



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 01, Revision 4a3 + Issue 27

~~12086~~ December January 2008

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev42.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev24.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd01-rev42.pdf>

Previous Version:

Latest Version:

<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>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec.pdf>

Latest Approved Version:

Technical Committee:

OASIS Service Component Architecture / J (SCA-J) TC

Chair(s):

Simon Nash,	IBM
Michael Rowley,	BEA Systems
Mark Combella,	Avaya

Editor(s):

Ron Barack,	SAP
David Booz,	IBM
Mark Combella,	Avaya
Mike Edwards,	IBM
Anish Karmarkar,	Oracle
Ashok Malhotra,	Oracle
Peter Peshev,	SAP

Related work:

This specification replaces or supersedes:

- Service Component Architecture Java Annotations and APIs Specification Version 1.00, March 21 2007

This specification is related to:

- Service Component Architecture Assembly Model Specification Version 1.1
- Service Component Architecture Policy Framework Specification Version 1.1

Field Code Changed

Field Code Changed

Field Code Changed

Declared XML Namespace(s):

<http://docs.oasis-open.org/ns/opencsa/sca/200712>

Abstract:

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

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous and conversational 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 individual programming models may 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" or "Latest Approved 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>).

The non-normative errata page for this specification is located at <http://www.oasis-open.org/committees/sca-j/>.

Notices

Copyright © OASIS® 2005, 2008. 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 names "OASIS", [insert specific trademarked names and abbreviations here] are trademarks 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	11
1.1	Terminology	11
1.2	Normative References	11
1.3	Non-Normative References	12
2	Implementation Metadata	13
2.1	Service Metadata	13
2.1.1	@Service	13
2.1.2	Java Semantics of a Remotable Service	13
2.1.3	Java Semantics of a Local Service	13
2.1.4	@Reference	14
2.1.5	@Property	14
2.2	Implementation Scopes: @Scope, @Init, @Destroy	14
2.2.1	Stateless scope	15
2.2.2	Composite scope	15
2.2.3	Conversation scope	15
3	Interface	16
3.1	Java interface element – <interface.java>	16
3.2	@Remotable	17
3.3	@Conversational	17
3.4	@Callback	17
4	Client API	18
4.1	Accessing Services from an SCA Component	18
4.1.1	Using the Component Context API	18
4.2	Accessing Services from non-SCA component implementations	18
4.2.1	ComponentContext	18
5	Error Handling	19
6	Asynchronous and Conversational Programming	20
6.1	@OneWay	20
6.2	Conversational Services	20
6.2.1	ConversationAttributes	20
6.2.2	@EndsConversation	21
6.3	Passing Conversational Services as Parameters	21
6.4	Conversational Client	21
6.5	Conversation Lifetime Summary	22
6.6	Conversation ID	23
6.6.1	Application Specified Conversation IDs	23
6.6.2	Accessing Conversation IDs from Clients	23
6.7	Callbacks	23
6.7.1	Stateful Callbacks	23
6.7.2	Stateless Callbacks	25
6.7.3	Implementing Multiple Bidirectional Interfaces	26
6.7.4	Accessing Callbacks	26
6.7.5	Customizing the Callback	27

6.7.6 Customizing the Callback Identity	27
6.7.7 Bindings for Conversations and Callbacks	28
7 Java API	29
7.1 Component Context	29
7.2 Request Context	30
7.3 CallableReference	31
7.4 ServiceReference	32
7.5 Conversation	32
7.6 ServiceRuntimeException	33
7.7 NoRegisteredCallbackException	33
7.8 ServiceUnavailableException	33
7.9 InvalidServiceException	33
7.10 ConversationEndedException	34
8 Java Annotations	35
8.1 @AllowsPassByReference	35
8.2 @ Authentication	36
8.3 @Callback	36
8.4 @ComponentName	38
8.5 @Confidentiality	38
8.6 @Constructor	39
8.7 @Context	40
8.8 @Conversational	40
8.9 @ConversationAttributes	41
8.10 @ConversationID	42
8.11 @Destroy	42
8.12 @EagerInit	43
8.13 @EndsConversation	43
8.14 @Init	44
8.15 @Integrity	45
8.16 @OneWay	45
8.17 @PolicySets	46
8.18 @Property	47
8.19 @Qualifier	48
8.20 @Reference	48
8.20.1 Reinjection	51
8.21 @Remotable	52
8.22 @Requires	54
8.23 @Scope	54
8.24 @Service	55
8.25 Security Implementation Policy Annotations	56
9 WSDL to Java and Java to WSDL	58
9.1 JAX-WS Client Asynchronous API for a Synchronous Service	58
10 Policy Annotations for Java	60
10.1 General Intent Annotations	60
10.2 Specific Intent Annotations	62

10.2.1 How to Create Specific Intent Annotations.....	63
10.3 Application of Intent Annotations.....	64
10.3.1 Inheritance And Annotation.....	65
10.4 Relationship of Declarative And Annotated Intents.....	67
10.5 Policy Set Annotations.....	67
10.6 Security Policy Annotations.....	67
10.6.1 Security Interaction Policy.....	68
10.6.2 Security Implementation Policy.....	70
A. XML Schema: sca-interface-java.xsd.....	74
B. Conformance Items.....	75
C. Acknowledgements.....	76
D. Non-Normative Text.....	77
E. Revision History.....	78
1— Introduction.....	9
1.1 Terminology.....	9
1.2 Normative References.....	9
1.3 Non-Normative References.....	10
2— Implementation Metadata.....	11
2.1 Service Metadata.....	11
2.1.1 @Service.....	11
2.1.2 Java Semantics of a Remotable Service.....	11
2.1.3 Java Semantics of a Local Service.....	11
2.1.4 @Reference.....	12
2.1.5 @Property.....	12
2.2 Implementation Scopes: @Scope, @Init, @Destroy.....	12
2.2.1 Stateless scope.....	13
2.2.2 Composite scope.....	13
2.2.3 Conversation scope.....	13
3— Interface.....	14
3.1 Java interface element — <interface.java>.....	14
3.2 @Remotable.....	15
3.3 @Conversational.....	15
3.4 @Callback.....	15
4— Client API.....	16
4.1 Accessing Services from an SCA Component.....	16
4.1.1 Using the Component Context API.....	16
4.2 Accessing Services from non-SCA component implementations.....	16
4.2.1 ComponentContext.....	16
5— Error Handling.....	17
6— Asynchronous and Conversational Programming.....	18
6.1 @OneWay.....	18
6.2 Conversational Services.....	18
6.2.1 ConversationAttributes.....	18
6.2.2 @EndsConversation.....	19
6.3 Passing Conversational Services as Parameters.....	19

6.4 Conversational Client.....	19
6.5 Conversation Lifetime Summary	20
6.6 Conversation ID	21
6.6.1 Application Specified Conversation IDs	21
6.6.2 Accessing Conversation IDs from Clients	21
6.7 Callbacks	21
6.7.1 Stateful Callbacks	21
6.7.2 Stateless Callbacks	23
6.7.3 Implementing Multiple Bidirectional Interfaces	24
6.7.4 Accessing Callbacks	24
6.7.5 Customizing the Callback	25
6.7.6 Customizing the Callback Identity	25
6.7.7 Bindings for Conversations and Callbacks	26
7— Java API	27
7.1 Component Context	27
7.2 Request Context	28
7.3 CallableReference	29
7.4 ServiceReference	30
7.5 Conversation	30
7.6 ServiceRuntimeException	31
7.7 NoRegisteredCallbackException	31
7.8 ServiceUnavailableException	31
7.9 InvalidServiceException	31
7.10 ConversationEndedException	32
8— Java Annotations	33
8.1 @AllowsPassByReference	33
8.2 @Callback	34
8.3 @ComponentName	35
8.4 @Constructor	35
8.5 @Context	36
8.6 @Conversational	37
8.7 @ConversationAttributes	37
8.8 @ConversationID	38
8.9 @Destroy	39
8.10 @EagerInit	40
8.11 @EndsConversation	40
8.12 @Init	41
8.13 @OneWay	41
8.14 @Property	42
8.15 @Reference	43
8.15.1 Reinjection	46
8.16 @Remotable	47
8.17 @Scope	49
8.18 @Service	49
9— WSDL to Java and Java to WSDL	51

9.1 JAX-WS Client Asynchronous API for a Synchronous Service.....	51
10—Policy Annotations for Java.....	53
10.1 General Intent Annotations.....	53
10.2 Specific Intent Annotations.....	55
10.2.1 How to Create Specific Intent Annotations.....	56
10.3 Application of Intent Annotations.....	57
10.3.1 Inheritance And Annotation.....	58
10.4 Relationship of Declarative And Annotated Intents.....	59
10.5 Policy Set Annotations.....	60
10.6 Security Policy Annotations.....	60
10.6.1 Security Interaction Policy.....	60
10.6.2 Security Implementation Policy.....	63
A.—XML Schema: sca-interface-java.xsd.....	67
B.—Acknowledgements.....	68
C.—Non Normative Text.....	69
D.—Revision History.....	70
1—Introduction.....	7
1.1 Terminology.....	7
1.2 Normative References.....	7
1.3 Non-Normative References.....	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.....	11
2.2.2 Composite scope.....	11
2.2.3 Conversation scope.....	11
3—Interface.....	12
3.1 Java interface element ("interface.java").....	12
3.2 @Remotable.....	12
3.3 @Conversational.....	12
3.4 @Callback.....	12
4—Client API.....	13
4.1 Accessing Services from an SCA Component.....	13
4.1.1 Using the Component Context API.....	13
4.2 Accessing Services from non-SCA component implementations.....	13
4.2.1 ComponentContext.....	13
5—Error Handling.....	14
6—Asynchronous and Conversational Programming.....	15
6.1 @OneWay.....	15
6.2 Conversational Services.....	15

6.2.1 ConversationAttributes	15
6.2.2 @EndsConversation	16
6.3 Passing Conversational Services as Parameters	16
6.4 Conversational Client	16
6.5 Conversation Lifetime Summary	17
6.6 Conversation ID	18
6.6.1 Application Specified Conversation IDs	18
6.6.2 Accessing Conversation IDs from Clients	18
6.7 Callbacks	18
6.7.1 Stateful Callbacks	18
6.7.2 Stateless Callbacks	20
6.7.3 Implementing Multiple Bidirectional Interfaces	21
6.7.4 Accessing Callbacks	21
6.7.5 Customizing the Callback	22
6.7.6 Customizing the Callback Identity	22
6.7.7 Bindings for Conversations and Callbacks	23
7—Java API	24
7.1 Component Context	24
7.2 Request Context	25
7.3 CallableReference	26
7.4 ServiceReference	26
7.5 Conversation	27
7.6 ServiceRuntimeException	27
7.7 NoRegisteredCallbackException	28
7.8 ServiceUnavailableException	28
7.9 InvalidServiceException	28
7.10 ConversationEndedException	28
8—Java Annotations	30
8.1 @AllowsPassByReference	30
8.2 @Callback	30
8.3 @ComponentName	32
8.4 @Constructor	32
8.5 @Context	33
8.6 @Conversational	34
8.7 @ConversationAttributes	34
8.8 @ConversationID	35
8.9 @Destroy	36
8.10 @EagerInit	36
8.11 @EndsConversation	37
8.12 @Init	37
8.13 @OneWay	38
8.14 @Property	39
8.15 @Reference	40
8.15.1 Reinjection	43
8.16 @Remotable	44

8.17 @Scope	45
8.18 @Service	46
9— WSDL to Java and Java to WSDL	48
9.1 JAX-WS Client Asynchronous API for a Synchronous Service	48
10— Policy Annotations for Java	50
10.1 General Intent Annotations	50
10.2 Specific Intent Annotations	52
10.2.1 How to Create Specific Intent Annotations	53
10.3 Application of Intent Annotations	54
10.3.1 Inheritance And Annotation	55
10.4 Relationship of Declarative And Annotated Intents	56
10.5 Policy Set Annotations	57
10.6 Security Policy Annotations	57
10.6.1 Security Interaction Policy	57
10.6.2 Security Implementation Policy	60
A.— XML Schema: sca-interface java.xsd	64
B.— Acknowledgements	65
C.— Non-Normative Text	66
D.— Revision History	67

1 Introduction

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

Specifically, this specification covers:

1. Implementation metadata for specifying component services, references, and properties
2. A client and component API
3. Metadata for asynchronous and conversational 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 individual programming models may chose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

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

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

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] | SCA Assembly Specification, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf |
| [SDO] | SDO 2.1 Specification, http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf |
| [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 ,
WSDL 2.0: http://www.w3.org/TR/wsdl20/ |

43		[POLICY]	SCA Policy Framework, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf
44			
45		[JSR-250]	Common Annotation for Java Platform specification (JSR-250), http://www.jcp.org/en/jsr/detail?id=250
46			
47		[JAX-WS]	JAX-WS 2.1 Specification (JSR-224), http://www.jcp.org/en/jsr/detail?id=224
48			
49		[JAVABEANS]	JavaBeans 1.01 Specification,
50			http://java.sun.com/javase/technologies/desktop/javabeans/api/
51			

52 1.3 Non-Normative References

53		TBD	TBD
----	--	------------	-----

54 2 Implementation Metadata

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

57 2.1 Service Metadata

58 2.1.1 @Service

59
60 The **@Service annotation** is used on a Java class to specify the interfaces of the services
61 implemented by the implementation. Service interfaces are defined in one of the following ways:

- 62 • As a Java interface
- 63 • As a Java class
- 64 • As a Java interface generated from a Web Services Description Language [WSDL]
65 (WSDL portType (Java interfaces generated from a WSDL portType are always
66 **remotable**))

67 2.1.2 Java Semantics of a Remotable Service

68 A **remotable service** is defined using the @Remotable annotation on the Java interface that
69 defines the service. Remotable services are intended to be used for **coarse grained** services, and
70 the parameters are passed **by-value**. Remotable Services are not allowed to make use of method
71 **overloading**.

72 The following snippet shows an example of a Java interface for a remote service:

```
73 package services.hello;  
74 @Remotable  
75 public interface HelloService {  
76     String hello(String message);  
77 }  
78
```

79 2.1.3 Java Semantics of a Local Service

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

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

84 The following snippet shows an example of a Java interface for a local service:

```
85  
86 package services.hello;  
87 public interface HelloService {  
88     String hello(String message);  
89 }  
90
```

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

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

96 2.1.4 @Reference

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

100 2.1.5 @Property

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

104 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

111 This document defines three scopes:

- 112 • STATELESS
- 113 • CONVERSATION
- 114 • COMPOSITE

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

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

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

122 **@Destroy** specifies a method called when the scope ends.

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

125 The following snippet is an example showing a fragment of a service implementation annotated
126 with lifecycle methods:

```
127  
128     @Init  
129     public void start() {  
130         ...  
131     }  
132  
133     @Destroy  
134     public void stop() {  
135         ...
```

136 }
137

138 The following sections specify four standard scopes, which a Java-based implementation type may
139 support.

140 2.2.1 Stateless scope

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

143 The concurrency model for the stateless scope is single threaded. This means that the SCA
144 runtime MUST ensure that a stateless scoped implementation instance object is only ever
145 dispatched on one thread at any one time. In addition, within the SCA lifecycle of an instance, the
146 SCA runtime MUST only make a single invocation of one business method. Note that the SCA
147 lifecycle might not correspond to the Java object lifecycle due to runtime techniques such as
148 pooling.

149 2.2.2 Composite scope

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

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

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

160 2.2.3 Conversation scope

161 A **conversation** is defined as a series of correlated interactions between a client and a target
162 service. A conversational scope starts when the first service request is dispatched to an
163 implementation instance offering a conversational service. A conversational scope completes after
164 an end operation defined by the service contract is called and completes processing or the
165 conversation expires. A conversation may be long-running (for example, hours, days or weeks)
166 and the SCA runtime may choose to passivate implementation instances. If this occurs, the
167 runtime must guarantee that implementation instance state is preserved.

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

172 A conversational scoped class MUST NOT expose a service using a non-conversational interface.
173 When a service has a conversational interface it MUST be implemented by a conversation-scoped
174 component. If no scope is specified on the implementation, then conversation scope is implied.

175 The concurrency model for the conversation scope is multi-threaded. This means that the SCA
176 runtime MAY run multiple threads in a single conversational scoped implementation instance
177 object and it MUST NOT perform any synchronization.

178 3 Interface

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

180 3.1 Java interface element – (“<interface.java>”)

181 The Java interface element is used in SCDL files in places where an interface is declared in terms
182 of a Java interface class. The Java interface element identifies the Java interface class and
183 optionally identifies a callback interface, where the first Java interface represents the forward
184 (service) call interface and the second interface represents the interface used to call back from the
185 service to the client.

186
187 The following is the pseudo-schema for the interface.java element. The following snippet shows the
188 schema for the Java interface element:

```
190 <interface.java interface="NCName" callbackInterface="NCName"?- />
```

191
192 The interface.java element has the following attributes:

- 193 • ***@interface (1..1)*** – the Java interface class to use for the service interface. @interface
194 MUST be the fully qualified name of the Java interface class. MUST be the fully qualified
195 name of the Java interface class. [JCA30001]
- 196 • ***@callbackInterface (0..1)*** – the Java interface class to use for the callback interface.
197 @callbackInterface MUST be the fully qualified name of a Java interface used for
198 callbacks. MUST be the fully qualified name of a Java interface used for callbacks
199 [JCA30002]

200
201 The following snippet shows an example of the Java interface element:

```
202  
203 <interface.java interface="services.stockquote.StockQuoteService"  
204 callbackInterface="services.stockquote.StockQuoteServiceCallback"/>
```

205
206 Here, the Java interface is defined in the Java class file
207 ./services/stockquote/StockQuoteService.class, where the root directory is defined by the
208 contribution in which the interface exists. Similarly, the callback interface is defined in the Java
209 class file ./services/stockquote/StockQuoteServiceCallback.class.

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

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

Formatted: French (France)

Comment [ME1]: Issue 101 (and many other changes in this section)

Formatted: Font color: Red

Formatted: Bullets and Numbering

Formatted: Font color: Red

Formatted: Font color: Red

Comment [ME2]: Issue 79

223 | For the Java interface type system, ~~parameters and return types~~ of the service methods are
224 | ~~described using Java classes or simple Java types. Service Data Objects [SDO] are the preferred~~
225 | ~~form of Java class because of their integration with XML technologies.~~

226

227 3.2 @Remotable

228 The **@Remotable** annotation on a Java interface indicates that the interface is designed to be
229 used for remote communication. Remotable interfaces are intended to be used for **coarse**
230 **grained** services. Operations' parameters and return values are passed **by-value**. Remotable
231 Services are not allowed to make use of method **overloading**.

232 3.3 @Conversational

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

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

237 3.4 @Callback

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

242 4 Client API

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

245 4.1 Accessing Services from an SCA Component

246 An SCA component may obtain a service reference either through injection or programmatically
247 through the **ComponentContext** API. Using reference injection is the recommended way to
248 access a service, since it results in code with minimal use of middleware APIs. The
249 ComponentContext API is provided for use in cases where reference injection is not possible.

250 4.1.1 Using the Component Context API

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

254 4.2 Accessing Services from non-SCA component implementations

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

257 4.2.1 ComponentContext

258 Non-SCA client code can use the ComponentContext API to perform operations against a
259 component in an SCA domain. How client code obtains a reference to a ComponentContext is
260 runtime specific.

261 The following example demonstrates the use of the component Context API by non-SCA code:

```
262  
263 ComponentContext context = // obtained through host environment-specific means  
264 HelloService helloService =  
265     context.getService(HelloService.class, "HelloService");  
266 String result = helloService.hello("Hello World!");
```

267

5 Error Handling

268

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

269

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.

270

271

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](#).

272

273

274

6 Asynchronous and Conversational Programming

275
276
277
278
279
280
281

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

282
283
284

- support for non-blocking method calls
- conversational services
- callbacks

285

Each of these topics is discussed in the following sections.

286
287
288
289

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

290

6.1 @OneWay

291
292
293

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

294
295
296

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

297
298
299
300

For a Java client to make a non-blocking call to methods that either return values or which throw exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in section 9. 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.

301

6.2 Conversational Services

302
303
304

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

305

6.2.1 ConversationAttributes

306
307

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

308

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

309
310
311
312
313
314
315

```
package com.bigbank;  
  
import org.oesa.oasisopen.sca.annotations.ConversationAttributes;  
  
@ConversationAttributes(maxAge="30 days");  
public class LoanServiceImpl implements LoanService {  
  
}
```

316 6.2.2 @EndsConversation

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

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

325 If a conversation is ended with an explicit outbound call to an @EndsConversation method or
326 through a call to the ServiceReference.endConversation() method, then any subsequent call to an
327 operation on the service reference will start a new conversation. If the conversation ends for any
328 other reason (e.g. a timeout occurred), then until ServiceReference.getConversation().end() is
329 called, the ConversationEndedException is thrown by any conversational operation.

330 6.3 Passing Conversational Services as Parameters

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

334 A service provider may also create a service reference for itself that it can pass to other services.
335 A service implementation does this with a call to the createSelfReference(...) method:

```
336     interface ComponentContext{  
337         ...  
338         <B> ServiceReference<B> createSelfReference(Class  
339             businessInterface);  
340         <B> ServiceReference<B> createSelfReference(Class  
341             businessInterface, String serviceName);  
342     }
```

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

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

349 6.4 Conversational Client

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

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

```
357  
358 @Reference  
359 LoanService loanService;  
360 // Known to be conversational because the interface is marked as  
361 // conversational
```

```

362 public void applyForMortgage(Customer customer, HouseInfo houseInfo,
363                             int term)
364 {
365     LoanApplication loanApp;
366     loanApp = createApplication(customer, houseInfo);
367     loanService.apply(loanApp);
368     loanService.lockCurrentRate(term);
369 }
370
371 public boolean isApproved() {
372     return loanService.getLoanStatus().equals("approved");
373 }
374 public LoanApplication createApplication(Customer customer,
375                                         HouseInfo houseInfo) {
376     return ...;
377 }

```

378 6.5 Conversation Lifetime Summary

379 **Starting conversations**

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

- 381 • A @Reference to a conversational service is injected
- 382 • A call is made to CompositeContext.getServiceReference and then a method of the service
- 383 is called.

385 **Continuing conversations**

386 The client can continue an existing conversation, by:

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

392 **Ending conversations**

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

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

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

404 If a service reference is used after the service provider's conversation timeout has caused the
405 conversation to be ended, then `ConversationEndedException` is thrown. In order to use that
406 service reference for a new conversation, its `endConversation()` method must be called.
407

408 6.6 Conversation ID

409 Every conversation has a **conversation ID**. The conversation ID can be generated by the system,
410 or it can be supplied by the client component.

411 If a field or setter method is annotated with `@ConversationID`, then the conversation ID for the
412 conversation is injected. The type of the field is not necessarily `String`. System generated
413 conversation IDs are always strings, but application generated conversation IDs may be other
414 complex types.

415 6.6.1 Application Specified Conversation IDs

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

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

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

424 6.6.2 Accessing Conversation IDs from Clients

425 Whether the conversation ID is chosen by the client or is generated by the system, the client may
426 access the conversation ID by calling `getConversationID()` on the current conversation
427 object.

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

431 6.7 Callbacks

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

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

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

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

445 6.7.1 Stateful Callbacks

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

448 The following example interfaces show an interaction over a stateful callback.

```
449 package somepackage;
450 import org.esea.oasisopen.sca.annotations.Callback;
451 import org.esea.oasisopen.sca.annotations.Conversational;
452 import org.esea.oasisopen.sca.annotations.Remotable;
453 @Remotable
454 @Conversational
455 @Callback(MyServiceCallback.class)
456 public interface MyService {
457
458     void someMethod(String arg);
459 }
460
461 @Remotable
462 @Conversational
463 public interface MyServiceCallback {
464
465     void receiveResult(String result);
466 }
467
```

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

474 When an interface and its callback interface are both marked as conversational, then there is only
475 one conversation that applies in both directions and it has the same lifetime. In this case, if both
476 interfaces declare a @ConversationAttributes annotation, then only the annotation on the main
477 interface applies.

```
478 @Callback
479 protected MyServiceCallback callback;
480
481 @Reference
482 protected MyService backendService;
483
484 public void someMethod(String arg) {
485     backendService.someMethod(arg);
486 }
487
488 public void receiveResult(String result) {
489     callback.receiveResult(result);
490 }
491
```

492
493 This fragment must come from an implementation that offers two services, one that it offers to its
494 clients (MyService) and one that is used for receiving callbacks from the back end
495 (MyServiceCallback). The code snippet below is taken from the client of this service, which also
496 implements the methods defined in MyServiceCallback.

497


```

498
499 private MyService myService;
500
501 @Reference
502 public void setMyService(MyService service) {
503     myService = service;
504 }
505
506 public void aClientMethod() {
507     ...
508     myService.someMethod(arg);
509 }
510
511 public void receiveResult(String result) {
512     // code to process the result
513 }
514

```

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

523 6.7.2 Stateless Callbacks

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

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

```

534
535 private ServiceReference<MyService> myService;
536
537 @Reference
538 public void setMyService(ServiceReference<MyService> service) {
539     myService = service;
540 }
541
542 public void aClientMethod() {
543     String someKey = "1234";
544     ...
545
546     myService.setCallbackID(someKey);
547     myService.getService().someMethod(arg);
548 }
549
550 @Context RequestContext context;
551
552 public void receiveResult(String result) {

```

Formatted: French (France)

```
553     Object key = context.getServiceReference().getCallbackID();
554     // Lookup any relevant state based on "key"
555     // code to process the result
556 }
```

557
558 Just as with stateful callbacks, a service implementation gets access to the callback object by
559 annotating a field or setter method with the `@Callback` annotation, such as the following:

```
560  
561 @Callback
562 protected MyServiceCallback callback;
563
```

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

569 6.7.3 Implementing Multiple Bidirectional Interfaces

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

```
576  
577 @Callback
578 protected MyService1Callback callback1;
579  
580 @Callback
581 protected MyService2Callback callback2;
582
```

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

585 6.7.4 Accessing Callbacks

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

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

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

```
592  
593 @Callback
594 protected CallableReference<MyCallback> callback;
595  
596 public void someMethod() {
597  
598     MyCallback myCallback = callback.getCallback();    ...
599  
600     myCallback.receiveResult(theResult);
601 }
602
```

603 Alternatively, a callback may be retrieved programmatically using the **RequestContext** API. The
604 snippet below shows how to retrieve a callback in a method programmatically:

605

```
606 public void someMethod() {  
607     MyCallback myCallback =  
608         ComponentContext.getRequestContext().getCallback();  
609  
610     ...  
611  
612     myCallback.receiveResult(theResult);  
613 }  
614  
615
```

616 On the client side, the service that implements the callback can access the callback ID that was
617 returned with the callback operation by accessing the request context, as follows:

618

```
619 @Context  
620 protected RequestContext requestContext;  
621  
622 void receiveResult(Object theResult) {  
623     Object refParams =  
624         requestContext.getServiceReference().getCallbackID();  
625  
626     ...  
627 }
```

628

629 On the client side, the object returned by the `getServiceReference()` method represents the
630 service reference for the callback. The object returned by `getCallbackID()` represents the
631 identity associated with the callback, which may be a single String or may be an object (as
632 described below in "Customizing the Callback Identity").

633 6.7.5 Customizing the Callback

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

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

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

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

650 6.7.6 Customizing the Callback Identity

651 The identity that is used to identify a callback request is initially generated by the system.

652 However, it is possible to provide an application specified identity to identify the callback by calling

653 the ***ServiceReference.setCallbackID()*** method. This can be used both for stateful and for
654 stateless callbacks. The identity is sent to the service provider, and the binding must guarantee
655 that the service provider will send the ID back when any callback method is invoked.

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

660 **6.7.7 Bindings for Conversations and Callbacks**

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

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

668 7 Java API

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

670 7.1 Component Context

671 The following Java code defines the *ComponentContext* interface:

672

```
673 package org.oesa.oasisopen.sca;
674
675 public interface ComponentContext {
676     String getURI();
677
678     <B> B getService(Class<B> businessInterface, String referenceName);
679
680     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
681                                             String referenceName);
682
683     <B> Collection<B> getServices(Class<B> businessInterface,
684                                String referenceName);
685
686     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
687                                                            businessInterface, String referenceName);
688
689     <B> ServiceReference<B> createSelfReference(Class<B>
690                                                businessInterface);
691
692     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
693                                                String serviceName);
694
695     <B> B getProperty(Class<B> type, String propertyName);
696
697     <B, R extends CallableReference<B>> R cast(B target)
698         throws IllegalArgumentException;
699
700     RequestContext getRequestContext();
701
702
703 }
```

704

- 705 • **getURI()** - returns the absolute URI of the component within the SCA domain
- 706 • **getService(Class businessInterface, String referenceName)** - Returns a proxy for
707 the reference defined by the current component. The getService() method takes as its
708 input arguments the Java type used to represent the target service on the client and the
709 name of the service reference. It returns an object providing access to the service. The
710 returned object implements the Java interface the service is typed with. This method
711 MUST throw an IllegalArgumentException if the reference has multiplicity greater than
712 one.
- 713 • **getServiceReference(Class businessInterface, String referenceName)** - Returns a
714 ServiceReference defined by the current component. This method MUST throw an
715 IllegalArgumentException if the reference has multiplicity greater than one.

- 716 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
717 typed service proxies for a business interface type and a reference name.
- 718 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
719 list typed service references for a business interface type and a reference name.
- 720 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
721 be used to invoke this component over the designated service.
- 722 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
723 ServiceReference that can be used to invoke this component over the designated service.
724 Service name explicitly declares the service name to invoke
- 725 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
726 property defined by this component.
- 727 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
728 there is no current request or if the context is unavailable. This method MUST return non-
729 null when invoked during the execution of a Java business method for a service operation
730 or callback operation, on the same thread that the SCA runtime provided, and MUST
731 return null in all other cases.
- 732 • **cast(B target)** - Casts a type-safe reference to a CallableReference

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

736
737 The following shows an example of component context usage in a Java class using the @Context
738 annotation.

```
739 private ComponentContext componentContext;
740
741 @Context
742 public void setContext(ComponentContext context) {
743     componentContext = context;
744 }
745
746 public void doSomething() {
747     HelloWorld service =
748         componentContext.getService(HelloWorld.class, "HelloWorldComponent");
749     service.hello("hello");
750 }
751
```

Formatted: French (France)

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

755 7.2 Request Context

756 The following shows the **RequestContext** interface:

```
757
758 package org.easoaasisopen.sca;
759
760 import javax.security.auth.Subject;
761
762 public interface RequestContext {
763
764     Subject getSecuritySubject();
765
```

```

766     String getServiceName();
767     <CB> CallableReference<CB> getCallbackReference();
768     <CB> CB getCallback();
769     <B> CallableReference<B> getServiceReference();
770
771 }
772

```

773 The RequestContext interface has the following methods:

- 774 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 775 • **getServiceName()** – Returns the name of the service on the Java implementation the
776 request came in on
- 777 • **getCallbackReference()** – Returns a callable reference to the callback as specified by the
778 caller. This method returns null when called for a service request whose interface is not
779 bidirectional or when called for a callback request.
- 780 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
781 getCallbackReference() method, this method returns null when called for a service request
782 whose interface is not bidirectional or when called for a callback request.
- 783 • **getServiceReference()** – When invoked during the execution of a service operation, this
784 method MUST return a CallableReference that represents the service that was invoked.
785 When invoked during the execution of a callback operation, this method MUST return a
786 CallableReference that represents the callback that was invoked.

787 7.3 CallableReference

788 The following Java code defines the **CallableReference** interface:

```

789
790 package org.oasisopen.sca;
791
792 public interface CallableReference<B> extends java.io.Serializable {
793
794     B getService();
795     Class<B> getBusinessInterface();
796     boolean isConversational();
797     Conversation getConversation();
798     Object getCallbackID();
799 }
800

```

801 The CallableReference interface has the following methods:

- 802
- 803 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
804 returned is guaranteed to implement the business interface for this reference. The value
805 returned is a proxy to the target that implements the business interface associated with this
806 reference.
- 807 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
808 this reference.
- 809 • **isConversational()** – Returns true if this reference is conversational.
- 810 • **getConversation()** – Returns the conversation associated with this reference. Returns null if
811 no conversation is currently active.
- 812 • **getCallbackID()** – Returns the callback ID.

813 7.4 ServiceReference

814

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

818 The following Java code defines the ServiceReference interface:

819

```
820 | package org.eseeoasisopen.sca;  
821  
822 | public interface ServiceReference<B> extends CallableReference<B> {  
823 |  
824 |     Object getConversationID();  
825 |     void setConversationID(Object conversationId) throws  
826 |         IllegalStateException;  
827 |     void setCallbackID(Object callbackID);  
828 |     Object getCallback();  
829 |     void setCallback(Object callback);  
830 | }
```

831

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

833

- 834 • **getConversationID()** - Returns the id supplied by the user that will be associated with
835 future conversations initiated through this reference, or null if no ID has been set by the
836 user.
- 837 • **setConversationID(Object conversationId)** – Set the ID, supplied by the user, to associate
838 with any future conversation started through this reference. If the value supplied is null then
839 the id will be generated by the implementation. Throws an IllegalStateException if a
840 conversation is currently associated with this reference.
- 841 • **setCallbackID(Object callbackID)** – Sets the callback ID.
- 842 • **getCallback()** – Returns the callback object.
- 843 • **setCallback(Object callback)** – Sets the callback object.

844 7.5 Conversation

845 The following snippet defines Conversation:

846

```
847 | package org.eseeoasisopen.sca;  
848 |  
849 | public interface Conversation {  
850 |     Object getConversationID();  
851 |     void end();  
852 | }
```

853

854 The Conversation interface has the following methods:

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

859 7.6 ServiceRuntimeException

860 The following snippet shows the *ServiceRuntimeException*.

861

```
862 | package org.eseaoasisopen.sca;  
863  
864 | public class ServiceRuntimeException extends RuntimeException {  
865 |     ...  
866 | }
```

867
868 This exception signals problems in the management of SCA component execution.

869 7.7 NoRegisteredCallbackException

870 The following snippet shows the *NoRegisteredCallbackException*.

871

```
872 | package org.eseaoasisopen.sca;  
873  
874 | public class NoRegisteredCallbackException extends  
875 |     ServiceRuntimeException {  
876 |     ...  
877 | }
```

878 This exception signals a problem where an attempt is made to invoke a callback when a client
879 does not implement the Callback interface and no valid custom Callback has been specified via a
880 call to *ServiceReference.setCallback()*.

881 7.8 ServiceUnavailableException

882 The following snippet shows the *ServiceUnavailableException*.

883

```
884 | package org.eseaoasisopen.sca;  
885  
886 | public class ServiceUnavailableException extends ServiceRuntimeException {  
887 |     ...  
888 | }
```

889
890 This exception signals problems in the interaction with remote services. These are exceptions
891 that may be transient, so retrying is appropriate. Any exception that is a
892 *ServiceRuntimeException* that is *not* a *ServiceUnavailableException* is unlikely to be resolved by
893 retrying the operation, since it most likely requires human intervention

894 7.9 InvalidServiceException

895 The following snippet shows the *InvalidServiceException*.

896

```
897 | package org.eseaoasisopen.sca;  
898  
899 | public class InvalidServiceException extends ServiceRuntimeException {  
900 |     ...  
901 | }
```

902
903 This exception signals that the *ServiceReference* is no longer valid. This can happen when the
904 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
905 be resolved by retrying the operation and will most likely require human intervention.

906 7.10 ConversationEndedException

907 The following snippet shows the *ConversationEndedException*.

```
908  
909 | package org.esaaoasisopen.sca;  
910  
911 public class ConversationEndedException extends ServiceRuntimeException {  
912     ...  
913 }  
914
```

915 8 Java Annotations

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

917 This specification places constraints on some annotations that are not detectable by a Java
918 compiler. For example, the definition of the @Property and @Reference annotations indicate that
919 they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to
920 constructor parameters. An SCA runtime MUST verify the proper use of all annotations and if an
921 annotation is improperly used, the SCA runtime MUST NOT run the component which uses the
922 invalid implementation code.

923 SCA annotations are not allowed on static methods and static fields. It is an error to use an SCA
924 annotation on a static method or a static field of an implementation class and the SCA runtime
925 MUST NOT instantiate such an implementation class.

926 8.1 @AllowsPassByReference

927 The following Java code defines the *@AllowsPassByReference* annotation:

```
928  
929 package org.oasisopen.sca.annotations;  
930  
931 import static java.lang.annotation.ElementType.TYPE;  
932 import static java.lang.annotation.ElementType.METHOD;  
933 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
934 import java.lang.annotation.Retention;  
935 import java.lang.annotation.Target;  
936  
937 @Target({TYPE, METHOD})  
938 @Retention(RUNTIME)  
939 public @interface AllowsPassByReference {  
940  
941 }  
942
```

Formatted: English (U.S.)

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

951 *@AllowsPassByReference* has no attributes

952

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

955

```
956 @AllowsPassByReference  
957 public String hello(String message) {  
958     ...  
959 }  
960
```

961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

8.2 @ Authentication

The following Java code defines the `@Authentication` annotation:

```

package org.oasisopen.sca.annotations;

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;

@Inherited
@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RUNTIME)
@Intent(Authentication.AUTHENTICATION)
public @interface Authentication {
    String AUTHENTICATION = SCA_PREFIX + "authentication";
    String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
    String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";

    /**
     * List of authentication qualifiers (such as "message" or "transport").
     *
     * @return authentication qualifiers
     */
    @Qualifier
    String[] value() default "";
}

```

The `@Authentication` annotation is used to indicate that the invocation requires authentication. Please check [10.3 Application of Intent Annotations](#) for samples and details.

Formatted: Heading 2,H2, Indent: Left: -0.4"

8.28.3 @Callback

The following Java code defines shows the `@Callback` annotation:

```

package org.oasisopen.sca.annotations;

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

@Target(TYPE, METHOD, FIELD)
@Retention(RUNTIME)

```

Formatted: Bullets and Numbering

Formatted: English (U.S.)

```

1015 public @interface Callback {
1016     Class<?> value() default Void.class;
1017 }
1018
1019
1020

```

1021 The @Callback annotation is used to annotate a service interface with a callback interface, which
1022 takes the Java Class object of the callback interface as a parameter.

1023 The @Callback annotation has the following attribute:

- 1024 • **value** – the name of a Java class file containing the callback interface

1025

1026 The @Callback annotation may also be used to annotate a method or a field of an SCA
1027 implementation class, in order to have a callback object injected

1028

1029 The following snippet shows a @Callback annotation on an interface:

1030

```

1031 @Remotable
1032 @Callback(MyServiceCallback.class)
1033 public interface MyService {
1034     void someAsyncMethod(String arg);
1035 }
1036
1037

```

1038 An example use of the @Callback annotation to declare a callback interface follows:

1039

```

1040 package somepackage;
1041 import org.esea.oasisopen.sca.annotations.Callback;
1042 import org.esea.oasisopen.sca.annotations.Remotable;
1043 @Remotable
1044 @Callback(MyServiceCallback.class)
1045 public interface MyService {
1046     void someMethod(String arg);
1047 }
1048
1049 @Remotable
1050 public interface MyServiceCallback {
1051     void receiveResult(String result);
1052 }
1053
1054

```

1055

1056 In this example, the implied component type is:

1057

```

1058 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1059     <service name="MyService">
1060         <interface.java interface="somepackage.MyService"
1061             callbackInterface="somepackage.MyServiceCallback"/>
1062     </service>
1063 </componentType>
1064

```

1065 **8.38.4 @ComponentName**

1066 The following Java code defines the **@ComponentName** annotation:

```
1067  
1068 package org.esis.oasisopen.sca.annotations;  
1069  
1070 import static java.lang.annotation.ElementType.METHOD;  
1071 import static java.lang.annotation.ElementType.FIELD;  
1072 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1073 import java.lang.annotation.Retention;  
1074 import java.lang.annotation.Target;  
1075  
1076 @Target({METHOD, FIELD})  
1077 @Retention(RUNTIME)  
1078 public @interface ComponentName {  
1079  
1080 }  
1081
```

1082 The @ComponentName annotation is used to denote a Java class field or setter method that is
1083 used to inject the component name.

1084
1085 The following snippet shows a component name field definition sample.

```
1086  
1087 @ComponentName  
1088 private String componentName;  
1089
```

1090 The following snippet shows a component name setter method sample.

```
1091  
1092 @ComponentName  
1093 public void setComponentName(String name) {  
1094     //...  
1095 }  
1096
```

1097 **8.5 @Confidentiality**

1098
1099 The following Java code defines the **@Confidentiality** annotation:
1100

```
1101 package org.oasisopen.sca.annotations;  
1102  
1103 import static java.lang.annotation.ElementType.FIELD;  
1104 import static java.lang.annotation.ElementType.METHOD;  
1105 import static java.lang.annotation.ElementType.PARAMETER;  
1106 import static java.lang.annotation.ElementType.TYPE;  
1107 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1108 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
1109  
1110 import java.lang.annotation.Inherited;  
1111 import java.lang.annotation.Retention;  
1112 import java.lang.annotation.Target;  
1113  
1114 @Inherited
```

Formatted: Bullets and Numbering

Formatted: English (U.S.)

Formatted: Font color: Custom
Color(59,0,111))

Formatted: Heading 2,H2, Indent: Left: 0",
First line: 0", Automatically adjust right indent
when grid is defined, Adjust space between
Latin and Asian text, Adjust space between
Asian text and numbers

```

1115 @Target({TYPE, FIELD, METHOD, PARAMETER})
1116 @Retention(RUNTIME)
1117 @Intent(Confidentiality.CONFIDENTIALITY)
1118 public @interface Confidentiality {
1119     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
1120     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
1121     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
1122
1123     /**
1124      * List of confidentiality qualifiers (such as "message" or "transport").
1125      *
1126      * @return confidentiality qualifiers
1127      */
1128     @Qualifier
1129     String[] value() default "";
1130 }
1131 The @Confidentiality annotation is used to indicate that the invocation requires confidentiality.
1132 Please check 10.3 Application of Intent Annotations for samples and details.
1133
1134

```

8.48.6 @Constructor

The following Java code defines the `@Constructor` annotation:

```

1137 package org.oasisopen.sca.annotations;
1138
1139 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1140 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1141 import java.lang.annotation.Retention;
1142 import java.lang.annotation.Target;
1143
1144 @Target (CONSTRUCTOR)
1145 @Retention (RUNTIME)
1146 public @interface Constructor { }
1147
1148
1149 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1150 Java component implementation. If this constructor has parameters, each of these parameters
1151 MUST have either a @Property annotation or a @Reference annotation.
1152
1153 The following snippet shows a sample for the @Constructor annotation.
1154
1155 public class HelloServiceImpl implements HelloService {
1156     public HelloServiceImpl() {
1157         ...
1158     }
1159
1160     @Constructor
1161     public HelloServiceImpl(@Property(name="someProperty") String
1162     someProperty ) {
1163         ...
1164     }
1165
1166     public String hello(String message) {
1167         ...

```

Formatted: Bullets and Numbering

Formatted: English (U.S.)

1168 }
1169 }

1170 **8.58.7 @Context**

1171 The following Java code defines the *@Context* annotation:

```
1172  
1173 package org.eseaopen.sca.annotations;  
1174  
1175 import static java.lang.annotation.ElementType.METHOD;  
1176 import static java.lang.annotation.ElementType.FIELD;  
1177 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1178 import java.lang.annotation.Retention;  
1179 import java.lang.annotation.Target;  
1180  
1181 @Target({METHOD, FIELD})  
1182 @Retention(RUNTIME)  
1183 public @interface Context {  
1184  
1185 }  
1186
```

1187 The @Context annotation is used to denote a Java class field or a setter method that is used to
1188 inject a composite context for the component. The type of context to be injected is defined by the
1189 type of the Java class field or type of the setter method input argument; the type is either
1190 **ComponentContext** or **RequestContext**.

1191 The @Context annotation has no attributes.

1192
1193 The following snippet shows a ComponentContext field definition sample.

```
1194  
1195 @Context  
1196 protected ComponentContext context;  
1197
```

1198 The following snippet shows a RequestContext field definition sample.

```
1199  
1200 @Context  
1201 protected RequestContext context;
```

1202 **8.68.8 @Conversational**

1203 The following Java code defines the *@Conversational* annotation:

```
1204  
1205 package org.eseaopen.sca.annotations;  
1206  
1207 import static java.lang.annotation.ElementType.TYPE;  
1208 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1209 import java.lang.annotation.Retention;  
1210 import java.lang.annotation.Target;  
1211 @Target(TYPE)  
1212 @Retention(RUNTIME)  
1213 public @interface Conversational {  
1214 }  
1215
```

Formatted: Bullets and Numbering

Formatted: English (U.S.)

Formatted: French (France)

Formatted: Bullets and Numbering

Formatted: French (France)

1216 The @Conversational annotation is used on a Java interface to denote a conversational service
1217 contract.

1218 The @Conversational annotation has no attributes.

1219 The following snippet shows a sample for the @Conversational annotation.

```
1220 package services.hello;  
1221  
1222 import org.eesa.oasisopen.sca.annotations.Conversational;  
1223  
1224 @Conversational  
1225 public interface HelloService {  
1226     void setName(String name);  
1227     String sayHello();  
1228 }
```

1229 **8.78.9 @ConversationAttributes**

1230 The following Java code defines the **@ConversationAttributes** annotation:

```
1231  
1232 package org.eesa.oasisopen.sca.annotations;  
1233  
1234 import static java.lang.annotation.ElementType.TYPE;  
1235 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1236 import java.lang.annotation.Retention;  
1237 import java.lang.annotation.Target;  
1238  
1239 @Target (TYPE)  
1240 @Retention (RUNTIME)  
1241 public @interface ConversationAttributes {  
1242  
1243     String maxIdleTime() default "";  
1244     String maxAge() default "";  
1245     boolean singlePrincipal() default false;  
1246 }  
1247
```

1248 The @ConversationAttributes annotation is used to define a set of attributes which apply to
1249 conversational interfaces of services or references of a Java class. The annotation has the following
1250 attributes:

- 1251 • **maxIdleTime (optional)** - The maximum time that can pass between successive
1252 operations within a single conversation. If more time than this passes, then the container
1253 may end the conversation.
- 1254 • **maxAge (optional)** - The maximum time that the entire conversation can remain active.
1255 If more time than this passes, then the container may end the conversation.
- 1256 • **singlePrincipal (optional)** - If true, only the principal (the user) that started the
1257 conversation has authority to continue the conversation. The default value is false.

1258
1259 The two attributes that take a time express the time as a string that starts with an integer, is
1260 followed by a space and then one of the following: "seconds", "minutes", "hours", "days" or
1261 "years".

1262
1263 Not specifying timeouts means that timeouts are defined by the SCA runtime implementation,
1264 however it chooses to do so.

Formatted: Bullets and Numbering

Formatted: French (France)

1265
1266 The following snippet shows the use of the @ConversationAttributes annotation to set the
1267 maximum age for a Conversation to be 30 days.

```
1268 package service.shoppingcart;  
1269  
1270 import org.esea.oasisopen.sca.annotations.ConversationAttributes;  
1271  
1272 @ConversationAttributes (maxAge="30 days");  
1273 public class ShoppingCartServiceImpl implements ShoppingCartService {  
1274     ...  
1275 }  
1276
```

1277 **8.88.10 @ConversationID**

1278 The following Java code defines the @ConversationID annotation:

```
1279  
1280 package org.esea.oasisopen.sca.annotations;  
1281  
1282 import static java.lang.annotation.ElementType.METHOD;  
1283 import static java.lang.annotation.ElementType.FIELD;  
1284 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1285 import java.lang.annotation.Retention;  
1286 import java.lang.annotation.Target;  
1287  
1288 @Target({METHOD, FIELD})  
1289 @Retention(RUNTIME)  
1290 public @interface ConversationID {  
1291 }  
1292  
1293
```

1294 The @ConversationID annotation is used to annotate a Java class field or setter method that is
1295 used to inject the conversation ID. System generated conversation IDs are always strings, but
1296 application generated conversation IDs may be other complex types.

1297 The following snippet shows a conversation ID field definition sample.

```
1298  
1299 @ConversationID  
1300 private String conversationID;  
1301
```

1302 The type of the field is not necessarily String.

1303

1304 **8.98.11 @Destroy**

1305 The following Java code defines the @Destroy annotation:

```
1306  
1307 package org.esea.oasisopen.sca.annotations;  
1308  
1309 import static java.lang.annotation.ElementType.METHOD;  
1310 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1311 import java.lang.annotation.Retention;
```

Formatted: Bullets and Numbering

Formatted: English (U.S.)

Formatted: Bullets and Numbering

Formatted: English (U.S.)

```
1312 import java.lang.annotation.Target;
1313
1314 @Target (METHOD)
1315 @Retention (RUNTIME)
1316 public @interface Destroy {
1317
1318 }
1319
```

1320 The @Destroy annotation is used to denote a single Java class method that will be called when the
1321 scope defined for the implementation class ends. The method MAY have any access modifier and
1322 MUST have a void return type and no arguments.

1323 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1324 when the scope defined for the implementation class ends. If the implementation class has a
1325 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1326 NOT instantiate the implementation class.

1327

1328 The following snippet shows a sample for a destroy method definition.

1329

```
1330 @Destroy
1331 public void myDestroyMethod() {
1332     ...
1333 }
```

1334 ~~8.108.12~~ @EagerInit

1335 The following Java code defines the @EagerInit annotation:

1336

```
1337 package org.esea.oasisopen.sca.annotations;
1338
1339 import static java.lang.annotation.ElementType.TYPE;
1340 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1341 import java.lang.annotation.Retention;
1342 import java.lang.annotation.Target;
1343
1344 @Target (TYPE)
1345 @Retention (RUNTIME)
1346 public @interface EagerInit {
1347
1348 }
1349
```

1350 The @EagerInit annotation is used to annotate the Java class of a COMPOSITE scoped
1351 implementation for eager initialization. When marked for eager initialization, the composite scoped
1352 instance is created when its containing component is started.

1353 ~~8.118.13~~ @EndsConversation

1354 The following Java code defines the @EndsConversation annotation:

1355

```
1356 package org.esea.oasisopen.sca.annotations;
1357
1358 import static java.lang.annotation.ElementType.METHOD;
1359 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1360 import java.lang.annotation.Retention;
```

Formatted: Bullets and Numbering

Formatted: French (France)

Formatted: Danish

Formatted: Bullets and Numbering

Formatted: English (U.S.)

```

1361 | import java.lang.annotation.Target;
1362 |
1363 | @Target (METHOD)
1364 | @Retention (RUNTIME)
1365 | public @interface EndsConversation {
1366 |
1367 |
1368 | }
1369 |

```

1370 The @EndsConversation annotation is used to denote a method on a Java interface that is called
1371 to end a conversation.

1372 The @EndsConversation annotation has no attributes.

1373 The following snippet shows a sample using the @EndsConversation annotation.

```

1374 | package services.shoppingbasket;
1375 |
1376 | import org.eesa.oasisopen.sca.annotations.EndsConversation;
1377 |
1378 | public interface ShoppingBasket {
1379 |     void addItem(String itemID, int quantity);
1380 |
1381 |     @EndsConversation
1382 |     void buy();
1383 | }

```

1384 **8.128.14 @Init**

1385 The following Java code defines the @Init annotation:

```

1387 | package org.eesa.oasisopen.sca.annotations;
1388 |
1389 | import static java.lang.annotation.ElementType.METHOD;
1390 | import static java.lang.annotation.RetentionPolicy.RUNTIME;
1391 | import java.lang.annotation.Retention;
1392 | import java.lang.annotation.Target;
1393 |
1394 | @Target (METHOD)
1395 | @Retention (RUNTIME)
1396 | public @interface Init {
1397 |
1398 |
1399 | }
1400 |

```

1401 The @Init annotation is used to denote a single Java class method that is called when the scope
1402 defined for the implementation class starts. The method MAY have any access modifier and MUST
1403 have a void return type and no arguments.

1404 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1405 after all property and reference injection is complete. If the implementation class has a method
1406 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1407 instantiate the implementation class.

1408 The following snippet shows an example of an init method definition.

```

1410 | @Init
1411 | public void myInitMethod() {

```

Formatted: Bullets and Numbering

Formatted: English (U.S.)

1412
1413
1414

```
...  
}  
}
```

1415

8.15 @Integrity

1416

The following Java code defines the **@Integrity** annotation:

1417

1418

1419

1420

1421

1422

1423

1424

1425

1426

1427

1428

1429

1430

1431

1432

1433

1434

1435

1436

1437

1438

1439

1440

1441

1442

1443

1444

1445

1446

1447

1448

1449

1450

1451

1452

1453

```
package org.oasisopen.sca.annotations;  
  
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.Constants.SCA_PREFIX;  
  
import java.lang.annotation.Inherited;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
@Inherited  
@Target({TYPE, FIELD, METHOD, PARAMETER})  
@Retention(RUNTIME)  
@Intent(Integrity.INTEGRITY)  
public @interface Integrity {  
    String INTEGRITY = SCA_PREFIX + "integrity";  
    String INTEGRITY_MESSAGE = INTEGRITY + ".message";  
    String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";  
  
    /**  
     * List of integrity qualifiers (such as "message" or "transport").  
     *  
     * @return integrity qualifiers  
     */  
    @Qualifier  
    String[] value() default "";  
}
```

The **@Integrity** annotation is used to indicate that the invocation requires integrity. Please check

10.3 Application of Intent Annotations for samples and details.

1454

8.138.16 @OneWay

1455

The following Java code defines the **@OneWay** annotation:

1456

1457

1458

1459

1460

1461

1462

1463

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

Formatted: Font color: Black

Formatted: Heading 2,H2, Indent: Left: 0",
First line: 0", Automatically adjust right indent
when grid is defined, Adjust space between
Latin and Asian text, Adjust space between
Asian text and numbers

Formatted

Formatted: Bullets and Numbering

Formatted: English (U.S.)

```
1464 @Target(METHOD)
1465 @Retention(RUNTIME)
1466 public @interface OneWay {
1467
1468
1469 }
1470
```

1471 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1472 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1473 Programming.

1474 The @OneWay annotation has no attributes.

1475 The following snippet shows the use of the @OneWay annotation on an interface.

```
1476 package services.hello;
1477
1478 import org.oasisopen.sca.annotations.OneWay;
1479
1480 public interface HelloService {
1481     @OneWay
1482     void hello(String name);
1483 }
1484
```

1485 8.17 @PolicySets

1486
1487 The following Java code defines the [@PolicySets](#) annotation:

```
1488 package org.oasisopen.sca.annotations;
1489
1490 import static java.lang.annotation.ElementType.FIELD;
1491 import static java.lang.annotation.ElementType.METHOD;
1492 import static java.lang.annotation.ElementType.PARAMETER;
1493 import static java.lang.annotation.ElementType.TYPE;
1494 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1495
1496 import java.lang.annotation.Retention;
1497 import java.lang.annotation.Target;
1498
1499 @Target({TYPE, FIELD, METHOD, PARAMETER})
1500 @Retention(RUNTIME)
1501 public @interface PolicySets {
1502     /**
1503      * Returns the policy sets to be applied.
1504      *
1505      * @return the policy sets to be applied
1506      */
1507     String[] value() default "";
1508 }
1509
1510
```

1511 The [@PolicySet](#) annotation is used to describe SCA Policy Sets. Please check [10.5 Policy Set](#)
1512 [Annotations](#) for samples and details.

1513
1514

Formatted: Font color: Custom
Color(RGB(59,0,111))

Formatted: Heading 2,H2, Indent: Left: 0",
First line: 0", Automatically adjust right indent
when grid is defined, Adjust space between
Latin and Asian text, Adjust space between
Asian text and numbers

Formatted: English (U.S.)

1515 **8.148.18 @Property**

1516 The following Java code defines the **@Property** annotation:

1517

```
1518 package org.esis.oasisopen.sca.annotations;
1519
1520 import static java.lang.annotation.ElementType.METHOD;
1521 import static java.lang.annotation.ElementType.FIELD;
1522 import static java.lang.annotation.ElementType.PARAMETER;
1523 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1524 import java.lang.annotation.Retention;
1525 import java.lang.annotation.Target;
1526
1527 @Target({METHOD, FIELD, PARAMETER})
1528 @Retention(RUNTIME)
1529 public @interface Property {
1530
1531     String name() default "";
1532     boolean required() default true;
1533 }
1534
```

1535

1536 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1537 parameter that is used to inject an SCA property value. The type of the property injected, which
1538 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1539 the type of the input parameter of the setter method or constructor.

1540 The @Property annotation may be used on fields, on setter methods or on a constructor method
1541 parameter. **However, the @Property annotation MUST NOT be used on a class field that is declared**
1542 **as final.**

1543 Properties may also be injected via setter methods even when the @Property annotation is not
1544 present. However, the @Property annotation must be used in order to inject a property onto a
1545 non-public field. In the case where there is no @Property annotation, the name of the property is
1546 the same as the name of the field or setter.

1547 Where there is both a setter method and a field for a property, the setter method is used.

1548

1549 The @Property annotation has the following attributes:

- 1550 • **name (optional)** – the name of the property. For a field annotation, the default is the
1551 name of the field of the Java class. For a setter method annotation, the default is the
1552 JavaBeans property name [\[JAVABEANS\]](#) corresponding to the setter method name. For a
1553 constructor parameter annotation, there is no default and the name attribute MUST be
1554 present.
- 1555 • **required (optional)** – specifies whether injection is required, defaults to true. For a
1556 constructor parameter annotation, this attribute MUST have the value true.

1557

1558 The following snippet shows a property field definition sample.

1559

```
1560 @Property(name="currency", required=true)
1561 protected String currency;
```

1562

1563 The following snippet shows a property setter sample

Formatted: Bullets and Numbering

Formatted: English (U.S.)

```
1564
1565 @Property(name="currency", required=true)
1566 public void setCurrency( String theCurrency ) {
1567     ....
1568 }
```

1570 If the property is defined as an array or as any type that extends or implements
1571 *java.util.Collection*, then the implied component type has a property with a *many* attribute set to
1572 true.

1574 The following snippet shows the definition of a configuration property using the @Property
1575 annotation for a collection.

```
1576
1577 ...
1578 private List<String> helloConfigurationProperty;
1579
1580 @Property(required=true)
1581 public void setHelloConfigurationProperty(List<String> property) {
1582     helloConfigurationProperty = property;
1583 }
1584 ...
1585
```

1586 **8.19 @Qualifier**

1588 The following Java code defines the @Qualifier annotation:

```
1589 package org.oasisopen.sca.annotations;
1590
1591 import static java.lang.annotation.ElementType.METHOD;
1592 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1593
1594 import java.lang.annotation.Retention;
1595 import java.lang.annotation.Target;
1596
1597 @Target (METHOD)
1598 @Retention (RUNTIME)
1599 public @interface Qualifier {
1600 }
1601 }
```

1603 The @Qualifier annotation can be applied to an attribute as an @Intent annotation to indicate the
1604 attribute provides qualifiers for the intent. Please check **10.3 Application of Intent**
1605 **Annotations** for samples and details.

1608 **8.158.20 @Reference**

1609 The following Java code defines the @Reference annotation:

1610

Formatted: Font color: Custom
Color(59,0,111)

Formatted: Heading 2,H2, Indent: Left: 0",
First line: 0", Automatically adjust right indent
when grid is defined, Adjust space between
Latin and Asian text, Adjust space between
Asian text and numbers

Formatted: Bullets and Numbering


```

1611 | package org.oasisopen.sca.annotations;
1612
1613 | import static java.lang.annotation.ElementType.METHOD;
1614 | import static java.lang.annotation.ElementType.FIELD;
1615 | import static java.lang.annotation.ElementType.PARAMETER;
1616 | import static java.lang.annotation.RetentionPolicy.RUNTIME;
1617 | import java.lang.annotation.Retention;
1618 | import java.lang.annotation.Target;
1619 | @Target({METHOD, FIELD, PARAMETER})
1620 | @Retention(RUNTIME)
1621 | public @interface Reference {
1622
1623 |     String name() default "";
1624 |     boolean required() default true;
1625 | }
1626

```

Formatted: English (U.S.)

1627 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1628 constructor parameter that is used to inject a service that resolves the reference. The interface of
1629 the service injected is defined by the type of the Java class field or the type of the input parameter
1630 of the setter method or constructor.

1631 **The @Reference annotation MUST NOT be used on a class field that is declared as final.**

1632 References may also be injected via setter methods even when the @Reference annotation is not
1633 present. However, the @Reference annotation must be used in order to inject a reference onto a
1634 non-public field. In the case where there is no @Reference annotation, the name of the reference
1635 is the same as the name of the field or setter.

1636 Where there is both a setter method and a field for a reference, the setter method is used.

1637 The @Reference annotation has the following attributes:

- 1638 • **name (optional)** – the name of the reference. For a field annotation, the default is the
1639 name of the field of the Java class. For a setter method annotation, the default is the
1640 JavaBeans property name corresponding to the setter method name. For a constructor
1641 parameter annotation, there is no default and the name attribute MUST be present.
- 1642 • **required (optional)** – whether injection of service or services is required. Defaults to true.
1643 For a constructor parameter annotation, this attribute MUST have the value true.

1644

1645 The following snippet shows a reference field definition sample.

1646

```

1647 | @Reference(name="stockQuote", required=true)
1648 | protected StockQuoteService stockQuote;

```

1649

1650 The following snippet shows a reference setter sample

1651

```

1652 | @Reference(name="stockQuote", required=true)
1653 | public void setStockQuote( StockQuoteService theSQService ) {
1654 |     ...
1655 | }

```

1656

1657 The following fragment from a component implementation shows a sample of a service reference
1658 using the @Reference annotation. The name of the reference is "helloService" and its type is

1659 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1660 helloService reference.

```
1661
1662 package services.hello;
1663
1664 private HelloService helloService;
1665
1666 @Reference(name="helloService", required=true)
1667 public setHelloService(HelloService service) {
1668     helloService = service;
1669 }
1670
1671 public void clientMethod() {
1672     String result = helloService.hello("Hello World!");
1673     ...
1674 }
1675
```

1676 The presence of a @Reference annotation is reflected in the componentType information that the
1677 runtime generates through reflection on the implementation class. The following snippet shows
1678 the component type for the above component implementation fragment.

```
1679
1680 <?xml version="1.0" encoding="ASCII"?>
1681 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1682
1683     <!-- Any services offered by the component would be listed here -->
1684     <reference name="helloService" multiplicity="1..1">
1685         <interface.java interface="services.hello.HelloService"/>
1686     </reference>
1687
1688 </componentType>
1689
```

1690 If the reference is not an array or collection, then the implied component type has a reference
1691 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1692 attribute – 1..1 applies if required=true.

1693
1694 If the reference is defined as an array or as any type that extends or implements *java.util.Collection*,
1695 then the implied component type has a reference with a **multiplicity** of either *1..n* or *0..n*, depending
1696 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if
1697 required=true.

1698 The following fragment from a component implementation shows a sample of a service reference
1699 definition using the @Reference annotation on a java.util.List. The name of the reference is
1700 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1701 services referenced by the helloServices reference. In this case, at least one HelloService should
1702 be present, so **required** is true.

```
1703
1704
1705 @Reference(name="helloServices", required=true)
1706 protected List<HelloService> helloServices;
1707
1708 public void clientMethod() {
1709
1710     ...
1711     for (int index = 0; index < helloServices.size(); index++) {
1712         HelloService helloService =
```

```

1713         (HelloService)helloServices.get(index);
1714         String result = helloService.hello("Hello World!");
1715     }
1716     ...
1717 }
1718

```

1719 The following snippet shows the XML representation of the component type reflected from for the
1720 former component implementation fragment. There is no need to author this component type in
1721 this case since it can be reflected from the Java class.

```

1722
1723 <?xml version="1.0" encoding="ASCII"?>
1724 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1725
1726     <!-- Any services offered by the component would be listed here -->
1727     <reference name="helloServices" multiplicity="1..n">
1728         <interface.java interface="services.hello.HelloService"/>
1729     </reference>
1730
1731 </componentType>

```

1732
1733 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1734 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1735 of 0..N must be an empty array or collection.

1736 **8.15-18.20.1 Reinjection**

1737 References MAY be reinjected after the initial creation of a component if the reference target
1738 changes due to a change in wiring that has occurred since the component was initialized. In order
1739 for reinjection to occur, the following MUST be true:

- 1740 1. The component MUST NOT be STATELESS scoped.
- 1741 2. The reference MUST use either field-based injection or setter injection. References that are
1742 injected through constructor injection MUST NOT be changed. Setter injection allows for
1743 code in the setter method to perform processing in reaction to a change.
- 1744 3. If the reference has a conversational interface, then reinjection MUST NOT occur while the
1745 conversation is active.

1746 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1747 work as if the reference target was not changed.

1748 If an operation is called on a reference where the target of that reference has been undeployed,
1749 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference
1750 where the target of the reference has become unavailable for some reason, the SCA runtime
1751 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the
1752 reference MAY continue to work, depending on the runtime and the type of change that was made.
1753 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1754 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
1755 corresponds to the reference that is passed as a parameter to cast(). If the reference is
1756 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
1757 to work as if the reference target was not changed. If the target of a ServiceReference has been
1758 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is
1759 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the
1760 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the
1761 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to
1762 work, depending on the runtime and the type of change that was made. If it doesn't work, the
1763 exception thrown will depend on the runtime and the cause of the failure.

Formatted: Bullets and Numbering

1764
1765
1766
1767
1768
1769

1770
1771
1772
1773
1774
1775

1776

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

The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This means that in the cases listed above where reference reinjection is not allowed, the array or Collection for the reference MUST NOT change its contents. In cases where the contents of a reference collection MAY change, then for references that use setter injection, the setter method MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be the same array or Collection object previously injected to the component.

Change event	Effect on		
	Reference	Existing ServiceReference Object	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	MAY be reinjected (if other conditions* apply). If not reinjected, then it MUST continue to work as if the reference target was not changed.	MUST 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 SHOULD throw <code>InvalidServiceException</code> .	Business methods SHOULD throw <code>InvalidServiceException</code> .	Result SHOULD be a reference to the undeployed or unavailable service. Business methods SHOULD throw <code>InvalidServiceException</code> .
Target service changed	MAY 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.	MAY 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 SHOULD be a reference to the changed service.
* Other conditions: <ol style="list-style-type: none"> 1. The component MUST NOT be STATELESS scoped. 2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed. ** Result of invoking <code>ComponentContext.cast()</code> corresponds to the reference that is passed as a parameter to <code>cast()</code> .			

1777

1778

8.168.21 @Remotable

1779

The following Java code defines the `@Remotable` annotation:

1780

1781

```
package org.eseaopen.sca.annotations;
```

sca-javacaa-1.1-spec-cd01-rev4
Copyright © OASIS® 2005, 2008. All Rights Reserved.

18 January 2008
Page 52 of 78

Formatted: Bullets and Numbering

```
1782
1783 import static java.lang.annotation.ElementType.TYPE;
1784 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1785 import java.lang.annotation.Retention;
1786 import java.lang.annotation.Target;
```

Formatted: French (France)

```
1787
1788
1789 @Target (TYPE)
1790 @Retention (RUNTIME)
1791 public @interface Remotable {
1792
1793 }
1794
```

1795 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable
1796 service can be published externally as a service and must be translatable into a WSDL portType.

1797 The @Remotable annotation has no attributes.

1798

1799 The following snippet shows the Java interface for a remotable service with its @Remotable
1800 annotation.

```
1801 package services.hello;
1802
1803 import org.esea.oasisopen.sca.annotations.*;
1804
1805 @Remotable
1806 public interface HelloService {
1807
1808     String hello(String message);
1809 }
1810
```

1811 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
1812 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1813

1814 Complex data types exchanged via remotable service interfaces MUST be compatible with the
1815 marshalling technology used by the service binding. For example, if the service is going to be
1816 exposed using the standard Web Service binding, then the parameters MAY be JAXB [JAX-B] types
1817 or Service Data Objects (SDOs) [SDO].

1818 Independent of whether the remotable service is called from outside of the composite that
1819 contains it or from another component in the same composite, the data exchange semantics are
1820 **by-value**.

1821 Implementations of remotable services may modify input data during or after an invocation and
1822 may modify return data after the invocation. If a remotable service is called locally or remotely,
1823 the SCA container is responsible for making sure that no modification of input data or post-
1824 invocation modifications to return data are seen by the caller.

1825

1826 The following snippet shows a remotable Java service interface.

1827

```
1828 package services.hello;
1829
1830 import org.esea.oasisopen.sca.annotations.*;
1831
1832 @Remotable
```

```

1833 public interface HelloService {
1834     String hello(String message);
1835 }
1836
1837
1838 package services.hello;
1839
1840 import org.oasisopen.sca.annotations.*;
1841
1842 @Service(HelloService.class)
1843 public class HelloServiceImpl implements HelloService {
1844     public String hello(String message) {
1845         ...
1846     }
1847 }
1848
1849

```

1850 **8.22 @Requires**

1851 The following Java code defines the **@Requires** annotation:

```

1852 package org.oasisopen.sca.annotations;
1853
1854 import static java.lang.annotation.ElementType.FIELD;
1855 import static java.lang.annotation.ElementType.METHOD;
1856 import static java.lang.annotation.ElementType.PARAMETER;
1857 import static java.lang.annotation.ElementType.TYPE;
1858 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1859
1860 import java.lang.annotation.Inherited;
1861 import java.lang.annotation.Retention;
1862 import java.lang.annotation.Target;
1863
1864 @Inherited
1865 @Retention(RUNTIME)
1866 @Target({TYPE, METHOD, FIELD, PARAMETER})
1867 public @interface Requires {
1868     /**
1869      * Returns the attached intents.
1870      *
1871      * @return the attached intents
1872      */
1873     String[] value() default "";
1874 }
1875

```

1876 The **@Requires** annotation supports general purpose intents specified as strings. User may also define specific intents using **@Intent** annotation. Please check **10.1 General Intent Annotations** for samples and details.

1883 **8.178.23 @Scope**

1884 The following Java code defines the **@Scope** annotation:

1885

Formatted: Font color: Custom
Color(59,0,111)

Formatted: Heading 2,H2, Indent: Left: 0",
First line: 0", Automatically adjust right indent
when grid is defined, Adjust space between
Latin and Asian text, Adjust space between
Asian text and numbers

Formatted: Bullets and Numbering

```
1886 | package org.esis.oasisopen.sca.annotations;
1887
1888 | import static java.lang.annotation.ElementType.TYPE;
1889 | import static java.lang.annotation.RetentionPolicy.RUNTIME;
1890 | import java.lang.annotation.Retention;
1891 | import java.lang.annotation.Target;
```

Formatted: French (France)

```
1892
1893 | @Target (TYPE)
1894 | @Retention (RUNTIME)
1895 | public @interface Scope {
1896
1897 |     String value() default "STATELESS";
1898 | }
```

1899 The @Scope annotation may only be used on a service's implementation class. It is an error to use
1900 this annotation on an interface.

1901 The @Scope annotation has the following attribute:

- 1902 • **value** – the name of the scope.
1903 For 'STATELESS' implementations, a different implementation instance may be used to
1904 service each request. Implementation instances may be newly created or be drawn from a
1905 pool of instances.
1906 SCA defines the following scope names, but others can be defined by particular Java-
1907 based implementation types:
1908 STATELESS
1909 COMPOSITE
1910 CONVERSATION

1911 The default value is STATELESS, except for an implementation offering a @Conversational service,
1912 which has a default scope of CONVERSATION. See section 2.2 for more details of the SCA-defined
1913 scopes.

1914 The following snippet shows a sample for a CONVERSATION scoped service implementation:

```
1915 | package services.hello;
1916
1917 | import org.esis.oasisopen.sca.annotations.*;
1918
1919 | @Service (HelloService.class)
1920 | @Scope ("CONVERSATION")
1921 | public class HelloServiceImpl implements HelloService {
1922
1923 |     public String hello (String message) {
1924 |         ...
1925 |     }
1926 | }
1927
```

1928 **8.188.24 @Service**

Formatted: Bullets and Numbering

1929 The following Java code defines the @Service annotation:

```
1930
1931 | package org.esis.oasisopen.sca.annotations;
1932
1933 | import static java.lang.annotation.ElementType.TYPE;
1934 | import static java.lang.annotation.RetentionPolicy.RUNTIME;
1935 | import java.lang.annotation.Retention;
1936 | import java.lang.annotation.Target;
1937
1938 | @Target (TYPE)
```

Formatted: French (France)

```

1939 @Retention(RUNTIME)
1940 public @interface Service {
1941
1942     Class<?>[] interfaces() default {};
1943     Class<?> value() default Void.class;
1944 }
1945

```

1946 The @Service annotation is used on a component implementation class to specify the SCA services
 1947 offered by the implementation. The class need not be declared as implementing all of the
 1948 interfaces implied by the services, but all methods of the service interfaces must be present. A
 1949 class used as the implementation of a service is not required to have a @Service annotation. If a
 1950 class has no @Service annotation, then the rules determining which services are offered and what
 1951 interfaces those services have are determined by the specific implementation type.

1952 The @Service annotation has the following attributes:

- 1953 • **interfaces** – The value is an array of interface or class objects that should be exposed as
 1954 services by this component.
- 1955 • **value** – A shortcut for the case when the class provides only a single service interface.

1956 Only one of these attributes should be specified.

1957
 1958 A @Service annotation with no attributes is meaningless, it is the same as not having the
 1959 annotation there at all.

1960 The **service names** of the defined services default to the names of the interfaces or class, without
 1961 the package name.

1962 A component MUST NOT have two services with the same Java simple name. If a Java
 1963 implementation needs to realize two services with the same Java simple name then this can be
 1964 achieved through subclassing of the interface.

1965 The following snippet shows an implementation of the HelloService marked with the @Service
 1966 annotation.

```

1967 package services.hello;
1968
1969 import org.eesa.oasisopen.sca.annotations.Service;
1970
1971 @Service(HelloService.class)
1972 public class HelloServiceImpl implements HelloService {
1973
1974     public void hello(String name) {
1975         System.out.println("Hello " + name);
1976     }
1977 }
1978

```

1981 8.25 Security Implementation Policy Annotations

1982 [JSR 250 "Common Annotations for the Java Platform" defines the following annotations that can be](#)
 1983 [used for implementation security policy:](#)

1984 [javax.annotation.security.RunAs](#)
 1985 [javax.annotation.security.RolesAllowed](#)
 1986 [javax.annotation.security.PermitAll](#)
 1987

1988 | [javax.annotation.security.DenyAll](#)
1989 | [javax.annotation.security.DeclareRoles](#)

1990
1991 | [Based on JSR 250, the RunAs , DeclareRoles annotations can be specified on a class; the RolesAllowed](#)
1992 | [, PermitAll , DenyAll annotations can be specified on a class or on method\(s\).](#)

1993 | [Please check JSR250 and SCA Policy spec for details on the meaning of the annotations . Please check](#)
1994 | [the section \[10.6.2\] Security Implementation Policy for the details on how these annotations are](#)
1995 | [mapped into Policy framework.](#)
1996
1997

Formatted: Font: (Default) Verdana, 9 pt

Formatted: Font: (Default) Verdana, 9 pt

1998

9 WSDL to Java and Java to WSDL

1999
2000
2001

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

2002
2003
2004
2005
2006

For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a @WebService annotation on the class, even if it doesn't, and the @org.oesoasisopen.annotations.OneWay annotation should be treated as a synonym for the @javax.jws.OneWay annotation. For the WSDL-to-Java mapping, the generated @WebService annotation implies that the interface is @Remotable.

2007
2008
2009
2010
2011

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 and MAY support the SDO 2.1 mapping. 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.

2012

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

2013

- No support for holders

2014

2015
2016

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

2017

9.1 JAX-WS Client Asynchronous API for a Synchronous Service

2018
2019
2020
2021
2022

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.

2023
2024
2025
2026
2027

For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the Assembly specification. These methods are recognized as follows.

2028

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

2029

a. a method name that is M's method name with the characters "Async" appended, and

2030

b. the same parameter signature as M, and

2031

c. a return type of Response<R> where R is the return type of M

2032

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

2033

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

2034

a. a method name that is M's method name with the characters "Async" appended, and

2035

b. a parameter signature that is M's parameter signature with an additional final parameter of type

2036

AsyncHandler<R> where R is the return type of M, and

2037

c. a return type of Future<?>

2038

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

2039

As an example, an interface may be defined in WSDL as follows:

2040

```
<!-- WSDL extract -->
```

2041

```
<message name="getPrice">
```

```
2042 <part name="ticker" type="xsd:string"/>
2043 </message>
2044
2045 <message name="getPriceResponse">
2046 <part name="price" type="xsd:float"/>
2047 </message>
2048
2049 <portType name="StockQuote">
2050 <operation name="getPrice">
2051 <input message="tns:getPrice"/>
2052 <output message="tns:getPriceResponse"/>
2053 </operation>
2054 </portType>
```

2055

2056 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2057 // asynchronous mapping
2058 @WebService
2059 public interface StockQuote {
2060     float getPrice(String ticker);
2061     Response<Float> getPriceAsync(String ticker);
2062     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2063 }
```

2064

2065 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2066 // synchronous mapping
2067 @WebService
2068 public interface StockQuote {
2069     float getPrice(String ticker);
2070 }
```

2071

2072 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2073 example, if the client implementation uses the asynchronous form of the interface, the two
2074 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2075 WS specification.

2076

10 Policy Annotations for Java

2077
2078
2079
2080
2081

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.

2082
2083
2084
2085

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.

2086
2087
2088
2089
2090
2091
2092
2093

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

2094
2095
2096
2097
2098

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

2099
2100
2101
2102
2103

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

2104 10.1 General Intent Annotations

2105
2106
2107

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.

2108
2109
2110

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

2111

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

2112
2113

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

2114
2115
2116

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

2117
2118

```
public static final String SCA_PREFIX="{http://docs.oasis-  
open.org/ns/opencsa/sca/200712}";
```

2119

```
public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
```

2120

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

2121 Notice that, by convention, qualified intents include the qualifier as part of the name of the
2122 constant, separated by an underscore. These intent constants are defined in the file that defines
2123 an annotation for the intent (annotations for intents, and the formal definition of these constants,
2124 are covered in a following section).

2125 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

2126 An example of the @Requires annotation with 2 qualified intents (from the Security domain)
2127 follows:

```
2128  
2129     @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2130
2131 This attaches the intents "confidentiality.message" and "integrity.message".

2132 The following is an example of a reference requiring support for confidentiality:

```
2133     package  
2134     org.oasisopen.sca.annotation;org.oasisopen.sca.annotations;  
2135  
2136     import static org.oasisopen.sca.annotations.Confidentiality.*;  
2137  
2138     public class Foo {  
2139         @Requires(CONFIDENTIALITY)  
2140         @Reference  
2141         public void setBar(Bar bar) {  
2142             ...  
2143         }  
2144     }
```

2145 Users may also choose to only use constants for the namespace part of the QName, so that they
2146 may add new intents without having to define new constants. In that case, this definition would
2147 instead look like this:

```
2148     package  
2149     org.oasisopen.sca.annotation;org.oasisopen.sca.annotations;  
2150  
2151     import static org.oasisopen.sca.Constants.*;  
2152  
2153     public class Foo {  
2154         @Requires(SCA_PREFIX+"confidentiality")  
2155         @Reference  
2156         public void setBar(Bar bar) {  
2157             ...  
2158         }  
2159     }
```

2160
2161 The formal syntax for the @Requires annotation follows:

2162 @Requires("qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]}
2163 where
2164 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2
2165

2166 The following shows the formal definition of the @Requires annotation:
2167

```
2168 package org.esea.oasisopen.sca.annotations;  
2169 import static java.lang.annotation.ElementType.TYPE;  
2170 import static java.lang.annotation.ElementType.METHOD;  
2171 import static java.lang.annotation.ElementType.FIELD;  
2172 import static java.lang.annotation.ElementType.PARAMETER;  
2173 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2174 import java.lang.annotation.Retention;  
2175 import java.lang.annotation.Target;  
2176 import java.lang.annotation.Inherited;
```

```
2177  
2178 @Inherited  
2179 @Retention(RUNTIME)  
2180 @Target({TYPE, METHOD, FIELD, PARAMETER})
```

```
2181  
2182 public @interface Requires {  
2183     String[] value() default "";  
2184 }
```

2185 The SCA_NS constant is defined in the Constants interface:

```
2186 package org.esea.oasisopen.sca;  
2187  
2188 public interface Constants {  
2189     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";  
2190     String SCA_PREFIX = "{"+SCA_NS+"}";  
2191 }
```

2193 10.2 Specific Intent Annotations

2194 In addition to the general intent annotation supplied by the @Requires annotation described
2195 above, it is also possible to have Java annotations that correspond to specific policy intents. SCA
2196 provides a number of these specific intent annotations and it is also possible to create new specific
2197 intent annotations for any intent.

2198 The general form of these specific intent annotations is an annotation with a name derived from
2199 the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an
2200 attribute to the annotation in the form of a string or an array of strings.

Formatted: English (U.S.)

2201 For example, the SCA confidentiality intent described in [the section on General Intent Annotations](#)
2202 using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific
2203 @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent
2204 is:

```
2205     @Integrity
```

2206 An example of a qualified specific intent for the "authentication" intent is:

```
2207     @Authentication( {"message", "transport"} )
```

2208 This annotation attaches the pair of qualified intents: "authentication.message" and
2209 "authentication.transport" (the sca: namespace is assumed in this both of these cases –
2210 "http://docs.oasis-open.org/ns/opencsa/sca/200712").

2211 The general form of specific intent annotations is:

```
2212     @<Intent>[(qualifiers)]
```

2213 where Intent is an NCName that denotes a particular type of intent.

```
2214     Intent ::= NCName
```

```
2215     qualifiers ::= "qualifier" | {"qualifier" [, "qualifier" ] }
```

```
2216     qualifier ::= NCName | NCName/qualifier
```

2217

2218 10.2.1 How to Create Specific Intent Annotations

2219 SCA identifies annotations that correspond to intents by providing an @Intent annotation which
2220 must be used in the definition of an intent annotation.

2221 The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the
2222 String form of the QName of the intent. As part of the intent definition, it is good practice
2223 (although not required) to also create String constants for the Namespace, the Intent and for
2224 Qualified versions of the Intent (if defined). These String constants are then available for use with
2225 the @Requires annotation and it should also be possible to use one or more of them as
2226 parameters to the @Intent annotation.

2227 Alternatively, the QName of the intent may be specified using separate parameters for the
2228 targetNamespace and the localPart for example:

```
2229     @Intent(targetNamespace=SCA_NS, localPart="confidentiality").
```

2230 The definition of the @Intent annotation is the following:

2231

```
2232     package  
2233     org.oasisopen.sca.annotation;org.oasisopen.sca.annotations;  
2234     import static java.lang.annotation.ElementType.ANNOTATION_TYPE;  
2235     import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2236     import java.lang.annotation.Retention;  
2237     import java.lang.annotation.Target;  
2238     import java.lang.annotation.Inherited;
```

2239

```
2240     @Retention(RUNTIME)
```

```
2241     @Target(ANNOTATION_TYPE)
```

```
2242     public @interface Intent {
```

```
2243         String value() default "";
```

Formatted: English (U.S.)

```
2244     String targetNamespace() default "";
2245     String localPart() default "";
2246 }
```

2247 When an intent can be qualified, it is good practice for the first attribute of the annotation to be a
2248 string (or an array of strings) which holds one or more qualifiers.

2249 In this case, the attribute's definition should be marked with the @Qualifier annotation. The
2250 @Qualifier tells SCA that the value of the attribute should be treated as a qualifier for the intent
2251 represented by the whole annotation. If more than one qualifier value is specified in an
2252 annotation, it means that multiple qualified forms are required. For example:

```
2253     @Confidentiality({"message", "transport"})
```

2254 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
2255 are set for the element to which the confidentiality intent is attached.

2256 The following is the definition of the @Qualifier annotation.

2257

```
2258     package org.oasisopen.sca.annotations;
2259     import static java.lang.annotation.ElementType.METHOD;
2260     import static java.lang.annotation.RetentionPolicy.RUNTIME;
2261     import java.lang.annotation.Retention;
2262     import java.lang.annotation.Target;
2263     import java.lang.annotation.Inherited;
```

2264

```
2265     @Retention(RetentionPolicy.RUNTIME)
```

```
2266     @Target(ElementType.METHOD)
```

```
2267     public @interface Qualifier {
```

```
2268     }
```

2269

2270 Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific
2271 intent annotations are shown in [the section dealing with Security Interaction Policy](#).

2272

2273 10.3 Application of Intent Annotations

2274 The SCA Intent annotations can be applied to the following Java elements:

- 2275 • Java class
- 2276 • Java interface
- 2277 • Method
- 2278 • Field

2279 Where multiple intent annotations (general or specific) are applied to the same Java element, they
2280 are additive in effect. An example of multiple policy annotations being used together follows:

```
2281     @Authentication
```

```
2282     @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2283 In this case, the effective intents are "authentication", "confidentiality.message" and
2284 "integrity.message".

Formatted: English (U.S.)

2285 If an annotation is specified at both the class/interface level and the method or field level, then
2286 the method or field level annotation completely overrides the class level annotation of the same
2287 type.

2288 The intent annotation can be applied either to classes or to class methods when adding annotated
2289 policy on SCA services. Applying an intent to the setter method in a reference injection approach
2290 allows intents to be defined at references.

2291 10.3.1 Inheritance And Annotation

2292 The inheritance rules for annotations are consistent with the common annotation specification, JSR
2293 250.

2294 The following example shows the inheritance relations of intents on classes, operations, and super
2295 classes.

```
2296  
2297 package services.hello;  
2298 import org.esea.oasisopen.sca.annotations.Remotable;  
2299 import org.esea.oasisopen.sca.annotations.Integrity;  
2300 import org.esea.oasisopen.sca.annotations.Authentication;  
2301  
2302 @Integrity("transport")  
2303 @Authentication  
2304 public class HelloService {  
2305     @Integrity  
2306     @Authentication("message")  
2307     public String hello(String message) {...}  
2308  
2309     @Integrity  
2310     @Authentication("transport")  
2311     public String helloThere() {...}  
2312 }  
2313  
2314 package services.hello;  
2315 import org.esea.oasisopen.sca.annotations.Remotable;  
2316 import org.esea.oasisopen.sca.annotations.Confidentiality;  
2317 import org.esea.oasisopen.sca.annotations.Authentication;  
2318  
2319 @Confidentiality("message")  
2320 public class HelloChildService extends HelloService {  
2321     @Confidentiality("transport")  
2322     public String hello(String message) {...}  
2323     @Authentication  
2324     String helloWorld() {...}  
2325 }
```

2326 Example 2a. Usage example of annotated policy and inheritance.

2327

2328 The effective intent annotation on the helloWorld method is Integrity("transport"),
 2329 @Authentication, and @Confidentiality("message").

2330 The effective intent annotation on the hello method of the HelloChildService is
 2331 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),

2332 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity
 2333 and @Authentication("transport"), the same as in HelloService class.

2334 The effective intent annotation on the hello method of the HelloService is @Integrity and
 2335 @Authentication("message")

2336

2337 The listing below contains the equivalent declarative security interaction policy of the HelloService
 2338 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
 2339 Example 2a.

```

2340
2341 <?xml version="1.0" encoding="ASCII"?>
2342
2343 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2344           name="HelloServiceComposite" >
2345   <service name="HelloService" requires="integrity/transport
2346           authentication">
2347     ...
2348   </service>
2349   <service name="HelloChildService" requires="integrity/transport
2350           authentication confidentiality/message">
2351     ...
2352   </service>
2353   ...
2354
2355   <component name="HelloServiceComponent">*
2356     <implementation.java class="services.hello.HelloService"/>
2357     <operation name="hello" requires="integrity
2358           authentication/message"/>
2359     <operation name="helloThere"
2360 requires="integrity
2361           authentication/transport"/>
2362   </component>
2363   <component name="HelloChildServiceComponent">*
2364     <implementation.java
2365 class="services.hello.HelloChildService" />
2366     <operation name="hello"
2367 requires="confidentiality/transport"/>
2368     <operation name="helloThere" requires=" integrity/transport
2369           authentication"/>
2370     <operation name="helloWorld" requires="authentication"/>
2371   </component>
2372   ...
2373   ...
2374 </composite>
2375
  
```

2376 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.

2377

2378

2379 10.4 Relationship of Declarative And Annotated Intents

2380 Annotated intents on a Java class cannot be overridden by declarative intents either in a
2381 composite document which uses the class as an implementation or by statements in a component
2382 Type document associated with the class. This rule follows the general rule for intents that they
2383 represent fundamental requirements of an implementation.

2384 An unqualified version of an intent expressed through an annotation in the Java class may be
2385 qualified by a declarative intent in a using composite document.

2386

2387 10.5 Policy Set Annotations

2388 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for
2389 example, a concrete policy is the specific encryption algorithm to use when encrypting messages
2390 when using a specific communication protocol to link a reference to a service).

2391 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
2392 The @PolicySets annotation either takes the QName of a single policy set as a string or the name
2393 of two or more policy sets as an array of strings:
2394

```
2395     @PolicySets( "<policy set QName>" |  
2396               { "<policy set QName>" [, "<policy set QName>"] })
```

2397

2398 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2399 An example of the @PolicySets annotation:

2400

```
2401     @Reference(name="helloService", required=true)  
2402     @PolicySets({ MY_NS + "WS_Encryption_Policy",  
2403                MY_NS + "WS_Authentication_Policy" })  
2404     public setHelloService(HelloService service) {  
2405         . . .  
2406     }
```

2407 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
2408 using the namespace defined for the constant MY_NS.

2409 PolicySets must satisfy intents expressed for the implementation when both are present, according
2410 to the rules defined in [the Policy Framework specification \[POLICY\]](#).

2411 The SCA Policy Set annotation can be applied to the following Java elements:

- 2412 • Java class
- 2413 • Java interface
- 2414 • Method
- 2415 • Field

2416

2417 10.6 Security Policy Annotations

2418 This section introduces annotations for SCA's security intents, as defined in [the SCA Policy](#)
2419 [Framework specification \[POLICY\]](#).

2420

2421 10.6.1 Security Interaction Policy

2422 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply
2423 to the operation of services and references of an implementation:

- 2424 • @Integrity
- 2425 • @Confidentiality
- 2426 • @Authentication

2427 All three of these intents have the same pair of Qualifiers:

- 2428 • message
- 2429 • transport

2430 The following snippets shows the @Integrity, @Confidentiality and @Authentication annotations:

```
2431 package org.esea.oasisopen.sca.annotations;  
2432  
2433 import java.lang.annotation.*;  
2434 import static org.esea.oasisopen.sca.Constants.SCA_NS;  
2435  
2436 @Inherited  
2437 @Retention(RetentionPolicy.RUNTIME)  
2438 @Target({ElementType.TYPE, ElementType.METHOD,  
2439         ElementType.FIELD, ElementType.PARAMETER})  
2440 @Intent(Integrity.INTEGRITY)  
2441 public @interface Integrity {  
2442     String INTEGRITY = SCA_NS+"integrity";  
2443     String INTEGRITY_MESSAGE = INTEGRITY+".message";  
2444     String INTEGRITY_TRANSPORT = INTEGRITY+".transport";  
2445     @Qualifier  
2446     String[] value() default "";  
2447 }  
2448  
2449  
2450 package org.esea.oasisopen.sca.annotations;  
2451  
2452 import java.lang.annotation.*;  
2453 import static org.esea.oasisopen.sca.Constants.SCA_NS;  
2454  
2455 @Inherited  
2456 @Retention(RetentionPolicy.RUNTIME)  
2457 @Target({ElementType.TYPE, ElementType.METHOD,  
2458         ElementType.FIELD, ElementType.PARAMETER})  
2459 @Intent(Confidentiality.CONFIDENTIALITY)
```

```

2460 public @interface Confidentiality {
2461     String CONFIDENTIALITY = SCA_NS+"confidentiality";
2462     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY+".message";
2463     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+".transport";
2464     @Qualifier
2465     String[] value() default "";
2466 }
2467
2468
2469 package org.esea.oasisopen.sca.annotations;
2470
2471 import java.lang.annotation.*;
2472 import static org.esea.oasisopen.sca.Constants.SCA_NS;
2473
2474 @Inherited
2475 @Retention(RetentionPolicy.RUNTIME)
2476 @Target({ElementType.TYPE, ElementType.METHOD,
2477         ElementType.FIELD, ElementType.PARAMETER})
2478 @Intent(Authentication.AUTHENTICATION)
2479 public @interface Authentication {
2480     String AUTHENTICATION = SCA_NS+"authentication";
2481     String AUTHENTICATION_MESSAGE = AUTHENTICATION+".message";
2482     String AUTHENTICATION_TRANSPORT = AUTHENTICATION+".transport";
2483     @Qualifier
2484     String[] value() default "";
2485 }
2486
2487

```

2488 The following example shows an example of applying an intent to the setter method used to inject
2489 a reference. Accessing the hello operation of the referenced HelloService requires both
2490 "integrity.message" and "authentication.message" intents to be honored.

```

2491 //Interface for HelloService
2492 public interface service.hello.HelloService {
2493     String hello(String helloMsg);
2494 }
2495
2496 // Interface for ClientService
2497 public interface service.client.ClientService {
2498     public void clientMethod();
2499

```

```

2500     }
2501
2502     // Implementation class for ClientService
2503     package services.client;
2504
2505     import services.hello.HelloService;
2506
2507     import org.eseeoasisopen.sca.annotations.*;
2508
2509     @Service(ClientService.class)
2510     public class ClientServiceImpl implements ClientService {
2511
2512
2513         private HelloService helloService;
2514
2515         @Reference(name="helloService", required=true)
2516         @Integrity("message")
2517         @Authentication("message")
2518         public void setHelloService(HelloService service) {
2519             helloService = service;
2520         }
2521
2522         public void clientMethod() {
2523             String result = helloService.hello("Hello World!");
2524             ...
2525         }
2526     }
2527

```

Example 1. Usage of annotated intents on a reference.

10.6.2 Security Implementation Policy

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

- RunAs

Takes as a parameter a string which is the name of a Security role.
eg. @RunAs("Manager")

*Code marked with this annotation will execute with the Security permissions of the identified role.

- RolesAllowed

Takes as a parameter a single string or an array of strings which represent one or more role names. When present, the implementation can only be accessed by principals whose

Formatted: Indent: Left: 0.75", No bullets or numbering

Formatted: Bullets and Numbering

2545 role corresponds to one of the role names listed in the @roles attribute. How role names
2546 are mapped to security principals is implementation dependent (SCA does not define this).
2547 eg. @RolesAllowed({"Manager", "Employee"})

2548 • PermitAll

2549 No parameters. When present, grants access to all roles.
2550

2551 • DenyAll

2552 No parameters. When present, denies access to all roles.
2553

2554 • DeclareRoles

2555 Takes as a parameter a string or an array of strings which identify one or more role names
2556 that form the set of roles used by the implementation.
2557 eg. @DeclareRoles({"Manager", "Employee", "Customer"})

2558 (all these are declared in the Java package javax.annotation.security)

2559 None of these annotations will appear in the introspected componentType definitions, therefore, these
2560 policies are Java implementation specific policies which only Java implementation implementers needs
2561 to deal with.
2562
2563

2564 For a full explanation of these intents, see [the Policy Framework specification \[POLICY\]](#).

2565 10.6.2.1 Annotated Implementation Policy Example

2566 The following is an example showing annotated security implementation policy:

```
2567 package services.account;  
2568 @Remotable  
2569 public interface AccountService {  
2570     AccountReport getAccountReport(String customerID);  
2571 }  
2572
```

2573
2574 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,
2575 plus the service references it makes and the settable properties that it has, along with a set of
2576 implementation policy annotations:

```
2577 package services.account;  
2578 import java.util.List;  
2579 import commonj.sdo.DataFactory;  
2580 import org.esea.oasisopen.sca.annotations.Property;  
2581 import org.esea.oasisopen.sca.annotations.Reference;  
2582 import javax.annotation.security.org.esea.oasisopen.sca.annotations.RolesAllowed;  
2583 import javax.annotation.security.org.esea.oasisopen.sca.annotations.RunAs;  
2584 import javax.annotation.security.org.esea.oasisopen.sca.annotations.PermitAll;  
2585 import javax.annotation.security.org.esea.oasisopen.sca.annotations.PermitAll;  
2586 import services.accountdata.AccountDataService;  
2587 import services.accountdata.CheckingAccount;  
2588 import services.accountdata.SavingsAccount;  
2589 import services.accountdata.StockAccount;
```

```

2592 import services.stockquote.StockQuoteService;
2593 @RolesAllowed("customers")
2594 @RunAs("accountants" )
2595 public class AccountServiceImpl implements AccountService {
2596
2597     @Property
2598     protected String currency = "USD";
2599
2600     @Reference
2601     protected AccountDataService accountDataService;
2602     @Reference
2603     protected StockQuoteService stockQuoteService;
2604
2605     @RolesAllowed({"customers", "accountants"})
2606     public AccountReport getAccountReport(String customerID) {
2607
2608         DataFactory dataFactory = DataFactory.INSTANCE;
2609         AccountReport accountReport =
2610             (AccountReport) dataFactory.create(AccountReport.class);
2611         List accountSummaries = accountReport.getAccountSummaries();
2612
2613         CheckingAccount checkingAccount =
2614             accountDataService.getCheckingAccount(customerID);
2615         AccountSummary checkingAccountSummary =
2616             (AccountSummary) dataFactory.create(AccountSummary.class);
2617         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
2618 );
2619
2620         checkingAccountSummary.setAccountType("checking");
2621         checkingAccountSummary.setBalance(fromUSDollarToCurrency
2622             (checkingAccount.getBalance()));
2623         accountSummaries.add(checkingAccountSummary);
2624
2625         SavingsAccount savingsAccount =
2626             accountDataService.getSavingsAccount(customerID);
2627         AccountSummary savingsAccountSummary =
2628             (AccountSummary) dataFactory.create(AccountSummary.class);
2629         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
2630         savingsAccountSummary.setAccountType("savings");
2631         savingsAccountSummary.setBalance(fromUSDollarToCurrency
2632             (savingsAccount.getBalance()));
2633         accountSummaries.add(savingsAccountSummary);
2634
2635

```

Formatted: French (France)


```

2636     StockAccount stockAccount =
2637     accountDataService.getStockAccount(customerID);
2638     AccountSummary stockAccountSummary =
2639         (AccountSummary) dataFactory.create(AccountSummary.class);
2640     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
2641     stockAccountSummary.setAccountType("stock");
2642     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol())) *
2643         stockAccount.getQuantity();
2644     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
2645     accountSummaries.add(stockAccountSummary);
2646
2647     return accountReport;
2648 }
2649
2650 @PermitAll
2651 public float fromUSDollarToCurrency(float value) {
2652
2653     if (currency.equals("USD")) return value; else
2654     if (currency.equals("EURO")) return value * 0.8f; else
2655     return 0.0f;
2656 }
2657 }

```

2658 Example 3. Usage of annotated security implementation policy for the java language.

2659 In this example, the implementation class as a whole is marked:

- 2660 • @RolesAllowed("customers") - indicating that customers have access to the
- 2661 implementation as a whole
- 2662 • @RunAs("accountants") - indicating that the code in the implementation runs with the
- 2663 permissions of accountants

2664 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),
2665 which indicates that this method can be called by both customers and accountants.

2666 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method
2667 can be called by any role.

2668

2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) Copyright SCA Collaboration 2006 -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
  elementFormDefault="qualified">
  <include schemaLocation="sca-core.xsd"/>
  <element name="interface.java" type="sca:JavaInterface"
    substitutionGroup="sca:interface"/>
  <complexType name="JavaInterface">
    <complexContent>
      <extension base="sca:Interface">
        <sequence>
          <any namespace="##other" processContents="lax"
            minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="interface" type="NCName" use="required"/>
        <attribute name="callbackInterface" type="NCName"
          use="optional"/>
        <anyAttribute namespace="##any" processContents="lax"/>
      </extension>
    </complexContent>
  </complexType>
</schema>
```

Formatted: German (Germany)

Formatted: English (U.S.)

Formatted: French (France)

2697

B. Conformance Items

2698

This section contains a list of conformance items for the SCA Java Common Annotations and APIs specification.

2699

2700

Conformance ID

Description

[JCA30001]

@interface MUST be the fully qualified name of the Java interface class

Formatted: Font color: Red

[JCA30002]

@callbackInterface MUST be the fully qualified name of a Java interface used for callbacks

Formatted: Font color: Red

[JCA30003]

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.

Formatted: Font color: Red

2701

2702

B.C. Acknowledgements

2703
2704

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

2705

Participants:

2706
2707

[Participant Name, Affiliation | Individual Member]
[Participant Name, Affiliation | Individual Member]

2708

Formatted: Bullets and Numbering

G.D. Non-Normative Text

Formatted: Bullets and Numbering

2710
2711
2712

D.E. Revision History

[optional; should not be included in OASIS Standards]

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 Combellack	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	* 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	* 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	* 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

2713
2714