



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02 + Issue 27

26 January 2009

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02.pdf> (normative)

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 Combellack,	Avaya

Editor(s):

Ron Barack,	SAP
David Booz,	IBM
Mark Combellack,	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

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	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	13
3.3	@Conversational	13
3.4	@Callback	13
4	Client API	14
4.1	Accessing Services from an SCA Component	14
4.1.1	Using the Component Context API	14
4.2	Accessing Services from non-SCA component implementations	14
4.2.1	ComponentContext	14
5	Error Handling	15
6	Asynchronous and Conversational Programming	16
6.1	@OneWay	16
6.2	Conversational Services	16
6.2.1	ConversationAttributes	16
6.2.2	@EndsConversation	17
6.3	Passing Conversational Services as Parameters	17
6.4	Conversational Client	17
6.5	Conversation Lifetime Summary	18
6.6	Conversation ID	19
6.6.1	Application Specified Conversation IDs	19
6.6.2	Accessing Conversation IDs from Clients	19
6.7	Callbacks	19
6.7.1	Stateful Callbacks	19
6.7.2	Stateless Callbacks	21
6.7.3	Implementing Multiple Bidirectional Interfaces	22
6.7.4	Accessing Callbacks	22
6.7.5	Customizing the Callback	23

6.7.6 Customizing the Callback Identity	23
6.7.7 Bindings for Conversations and Callbacks	24
7 Java API	25
7.1 Component Context	25
7.2 Request Context	26
7.3 CallableReference	27
7.4 ServiceReference	28
7.5 Conversation	28
7.6 ServiceRuntimeException	29
7.7 NoRegisteredCallbackException	29
7.8 ServiceUnavailableException	29
7.9 InvalidServiceException	29
7.10 ConversationEndedException	30
8 Java Annotations	31
8.1 @AllowsPassByReference	31
8.2 @Callback	32
8.3 @ComponentName	34 ³³
8.4 @Constructor	35 ³³
8.5 @Context	36 ³⁴
8.6 @Conversational	36 ³⁵
8.7 @ConversationAttributes	37 ³⁵
8.8 @ConversationID	38 ³⁶
8.9 @Destroy	39 ³⁷
8.10 @EagerInit	39 ³⁸
8.11 @EndsConversation	40 ³⁸
8.12 @Init	40 ³⁹
8.13 @OneWay	41 ³⁹
8.14 @Property	43 ⁴⁰
8.15 @Reference	45 ⁴¹
8.15.1 Reinjection	47 ⁴⁴
8.16 @Remotable	49 ⁴⁵
8.17 @Scope	51 ⁴⁷
8.18 @Service	52 ⁴⁷
9 WSDL to Java and Java to WSDL	54 ⁴⁹
9.1 JAX-WS Client Asynchronous API for a Synchronous Service	54 ⁴⁹
10 Policy Annotations for Java	56 ⁵¹
10.1 General Intent Annotations	56 ⁵¹
10.2 Specific Intent Annotations	58 ⁵³
10.2.1 How to Create Specific Intent Annotations	59 ⁵⁴
10.3 Application of Intent Annotations	60 ⁵⁵
10.3.1 Inheritance And Annotation	61 ⁵⁶
10.4 Relationship of Declarative And Annotated Intents	62 ⁵⁷
10.5 Policy Set Annotations	63 ⁵⁸
10.6 Security Policy Annotations	63 ⁵⁸
10.6.1 Security Interaction Policy	63 ⁵⁸

10.6.2 Security Implementation Policy	<u>6661</u>
A. XML Schema: sca-interface-java.xsd.....	<u>7165</u>
B. Conformance Items	<u>7266</u>
C. Acknowledgements	<u>7367</u>
D. Non-Normative Text	<u>7468</u>
E. Revision History.....	<u>7569</u>

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, http://java.sun.com/javase/technologies/desktop/javabeans/api/
50		
51		

52 **1.3 Non-Normative References**

53	None	None
----	-------------	------

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

```
188  
189 <interface.java interface="NCName" callbackInterface="NCName"? />  
190
```

191 The interface.java element has the following attributes:

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

197
198 The following snippet shows an example of the Java interface element:

```
199  
200 <interface.java interface="services.stockquote.StockQuoteService"  
201 callbackInterface="services.stockquote.StockQuoteServiceCallback"/>  
202
```

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

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

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

217
218

219 3.2 @Remotable

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

224 3.3 @Conversational

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

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

229 3.4 @Callback

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

234 4 Client API

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

237 4.1 Accessing Services from an SCA Component

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

242 4.1.1 Using the Component Context API

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

246 4.2 Accessing Services from non-SCA component implementations

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

249 4.2.1 ComponentContext

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

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

```
254  
255 ComponentContext context = // obtained through host environment-specific means  
256 HelloService helloService =  
257     context.getService(HelloService.class, "HelloService");  
258 String result = helloService.hello("Hello World!");
```

259

5 Error Handling

260

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

261

Business exceptions are thrown by the implementation of the called service method, and are

262

defined as checked exceptions on the interface that types the service.

263

SCA runtime exceptions are raised by the SCA runtime and signal problems in management of

264

component execution or problems interacting with remote services. The SCA runtime exceptions

265

are [defined in the Java API section](#).

266

6 Asynchronous and Conversational Programming

267
268
269
270
271
272
273

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:

274
275
276

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

277

Each of these topics is discussed in the following sections.

278
279
280
281

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.

282

6.1 @OneWay

283
284
285

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.

286
287
288

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.

289
290
291
292

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.

293

6.2 Conversational Services

294
295
296

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.

297

6.2.1 ConversationAttributes

298
299

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

300

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

301
302
303
304
305
306
307

```
package com.bigbank;
import org.oasisopen.sca.annotations.ConversationAttributes;

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


308 6.2.2 @EndsConversation

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

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

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

322 6.3 Passing Conversational Services as Parameters

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

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

```
328     interface ComponentContext{  
329         ...  
330         <B> ServiceReference<B> createSelfReference(Class  
331             businessInterface);  
332         <B> ServiceReference<B> createSelfReference(Class  
333             businessInterface, String serviceName);  
334     }
```

335
336 The second variant, which takes an additional **serviceName** parameter, must be used if the
337 component implements multiple services.

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

341 6.4 Conversational Client

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

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

```
349  
350 @Reference  
351 LoanService loanService;  
352 // Known to be conversational because the interface is marked as  
353 // conversational
```

```

354 public void applyForMortgage(Customer customer, HouseInfo houseInfo,
355                             int term)
356 {
357     LoanApplication loanApp;
358     loanApp = createApplication(customer, houseInfo);
359     loanService.apply(loanApp);
360     loanService.lockCurrentRate(term);
361 }
362
363 public boolean isApproved() {
364     return loanService.getLoanStatus().equals("approved");
365 }
366 public LoanApplication createApplication(Customer customer,
367                                         HouseInfo houseInfo) {
368     return ...;
369 }

```

370 6.5 Conversation Lifetime Summary

371 **Starting conversations**

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

- 373 • A @Reference to a conversational service is injected
- 374 • A call is made to CompositeContext.getServiceReference and then a method of the service
- 375 is called.

377 **Continuing conversations**

378 The client can continue an existing conversation, by:

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

384 **Ending conversations**

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

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

392 If a method is invoked on a service reference after an @EndsConversation method has been called
393 then a new conversation will automatically be started. If

394 ServiceReference.getConversationID() is called after the @EndsConversation method is called,
395 but before the next conversation has been started, it returns null.

396 If a service reference is used after the service provider's conversation timeout has caused the
397 conversation to be ended, then `ConversationEndedException` is thrown. In order to use that
398 service reference for a new conversation, its `endConversation()` method must be called.
399

400 6.6 Conversation ID

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

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

407 6.6.1 Application Specified Conversation IDs

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

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

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

416 6.6.2 Accessing Conversation IDs from Clients

417 Whether the conversation ID is chosen by the client or is generated by the system, the client may
418 access the conversation ID by calling `getConversationID()` on the current conversation
419 object.

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

423 6.7 Callbacks

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

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

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

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

437 6.7.1 Stateful Callbacks

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

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

```
441 package somepackage;
442 import org.oasisopen.sca.annotations.Callback;
443 import org.oasisopen.sca.annotations.Conversational;
444 import org.oasisopen.sca.annotations.Remotable;
445 @Remotable
446 @Conversational
447 @Callback(MyServiceCallback.class)
448 public interface MyService {
449
450     void someMethod(String arg);
451 }
452
453 @Remotable
454 @Conversational
455 public interface MyServiceCallback {
456
457     void receiveResult(String result);
458 }
459
```

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

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

```
470 @Callback
471 protected MyServiceCallback callback;
472
473 @Reference
474 protected MyService backendService;
475
476 public void someMethod(String arg) {
477     backendService.someMethod(arg);
478 }
479
480 public void receiveResult(String result) {
481     callback.receiveResult(result);
482 }
483
```

484
485 This fragment must come from an implementation that offers two services, one that it offers to its
486 clients (MyService) and one that is used for receiving callbacks from the back end
487 (MyServiceCallback). The code snippet below is taken from the client of this service, which also
488 implements the methods defined in MyServiceCallback.

489

```

490
491 private MyService myService;
492
493 @Reference
494 public void setMyService(MyService service) {
495     myService = service;
496 }
497
498 public void aClientMethod() {
499     ...
500     myService.someMethod(arg);
501 }
502
503 public void receiveResult(String result) {
504     // code to process the result
505 }
506

```

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

515 6.7.2 Stateless Callbacks

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

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

```

526
527 private ServiceReference<MyService> myService;
528
529 @Reference
530 public void setMyService(ServiceReference<MyService> service) {
531     myService = service;
532 }
533
534 public void aClientMethod() {
535     String someKey = "1234";
536     ...
537
538     myService.setCallbackID(someKey);
539     myService.getService().someMethod(arg);
540 }
541
542 @Context RequestContext context;
543
544 public void receiveResult(String result) {

```

```

545     Object key = context.getServiceReference().getCallbackID();
546     // Lookup any relevant state based on "key"
547     // code to process the result
548 }

```

549
550 Just as with stateful callbacks, a service implementation gets access to the callback object by
551 annotating a field or setter method with the `@Callback` annotation, such as the following:

```

552 @Callback
553 protected MyServiceCallback callback;
554
555

```

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

561 6.7.3 Implementing Multiple Bidirectional Interfaces

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

```

568 @Callback
569 protected MyService1Callback callback1;
570
571 @Callback
572 protected MyService2Callback callback2;
573
574

```

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

577 6.7.4 Accessing Callbacks

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

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

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

```

583
584 @Callback
585 protected CallableReference<MyCallback> callback;
586
587 public void someMethod() {
588     MyCallback myCallback = callback.getCallback(); ...
589
590     myCallback.receiveResult(theResult);
591 }
592
593
594

```

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

597

```
598 public void someMethod() {  
599     MyCallback myCallback =  
600         ComponentContext.getRequestContext().getCallback();  
601     ...  
602     ...  
603     myCallback.receiveResult(theResult);  
604 }  
605  
606  
607
```

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

610

```
611 @Context  
612 protected RequestContext requestContext;  
613  
614 void receiveResult(Object theResult) {  
615     Object refParams =  
616         requestContext.getServiceReference().getCallbackID();  
617     ...  
618 }  
619
```

620

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

625 6.7.5 Customizing the Callback

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

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

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

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

642 6.7.6 Customizing the Callback Identity

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

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

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

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

652 **6.7.7 Bindings for Conversations and Callbacks**

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

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

660 7 Java API

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

662 7.1 Component Context

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

```
664
665 package org.oasisopen.sca;
666
667 public interface ComponentContext {
668     String getURI();
669
670     <B> B getService(Class<B> businessInterface, String referenceName);
671
672     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
673                                             String referenceName);
674
675     <B> Collection<B> getServices(Class<B> businessInterface,
676                                String referenceName);
677
678     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
679                                                           businessInterface, String referenceName);
680
681     <B> ServiceReference<B> createSelfReference(Class<B>
682                                               businessInterface);
683
684     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
685                                               String serviceName);
686
687     <B> B getProperty(Class<B> type, String propertyName);
688
689     <B, R extends CallableReference<B>> R cast(B target)
690         throws IllegalArgumentException;
691
692     RequestContext getRequestContext();
693
694
695 }
```

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

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

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

728

729 The following shows an example of component context usage in a Java class using the @Context
730 annotation.

```
731 private ComponentContext componentContext;
732
733 @Context
734 public void setContext(ComponentContext context) {
735     componentContext = context;
736 }
737
738 public void doSomething() {
739     HelloWorld service =
740     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
741     service.hello("hello");
742 }
743
```

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

747 7.2 Request Context

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

749

```
750 package org.oasisopen.sca;
751
752 import javax.security.auth.Subject;
753
754 public interface RequestContext {
755
756     Subject getSecuritySubject();
757
```

```

758     String getServiceName();
759     <CB> CallableReference<CB> getCallbackReference();
760     <CB> CB getCallback();
761     <B> CallableReference<B> getServiceReference();
762
763 }
764

```

765 The RequestContext interface has the following methods:

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

779 7.3 CallableReference

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

```

781
782 package org.oasisopen.sca;
783
784 public interface CallableReference<B> extends java.io.Serializable {
785
786     B getService();
787     Class<B> getBusinessInterface();
788     boolean isConversational();
789     Conversation getConversation();
790     Object getCallbackID();
791 }
792

```

793 The CallableReference interface has the following methods:

- 794
- 795 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
796 returned is guaranteed to implement the business interface for this reference. The value
797 returned is a proxy to the target that implements the business interface associated with this
798 reference.
- 799 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
800 this reference.
- 801 • **isConversational()** – Returns true if this reference is conversational.
- 802 • **getConversation()** – Returns the conversation associated with this reference. Returns null if
803 no conversation is currently active.
- 804 • **getCallbackID()** – Returns the callback ID.

805 7.4 ServiceReference

806

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

810 The following Java code defines the ServiceReference interface:

811

```
812 package org.oasisopen.sca;  
813  
814 public interface ServiceReference<B> extends CallableReference<B> {  
815  
816     Object getConversationID();  
817     void setConversationID(Object conversationId) throws  
818         IllegalStateException;  
819     void setCallbackID(Object callbackID);  
820     Object getCallback();  
821     void setCallback(Object callback);  
822 }
```

823

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

825

- 826 • **getConversationID()** - Returns the id supplied by the user that will be associated with
827 future conversations initiated through this reference, or null if no ID has been set by the
828 user.
- 829 • **setConversationID(Object conversationId)** – Set the ID, supplied by the user, to associate
830 with any future conversation started through this reference. If the value supplied is null then
831 the id will be generated by the implementation. Throws an IllegalStateException if a
832 conversation is currently associated with this reference.
- 833 • **setCallbackID(Object callbackID)** – Sets the callback ID.
- 834 • **getCallback()** – Returns the callback object.
- 835 • **setCallback(Object callback)** – Sets the callback object.

836 7.5 Conversation

837 The following snippet defines Conversation:

838

```
839 package org.oasisopen.sca;  
840  
841 public interface Conversation {  
842     Object getConversationID();  
843     void end();  
844 }
```

845

846 The Conversation interface has the following methods:

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

851 7.6 ServiceRuntimeException

852 The following snippet shows the *ServiceRuntimeException*.

853

```
854 package org.oasisopen.sca;  
855  
856 public class ServiceRuntimeException extends RuntimeException {  
857     ...  
858 }
```

859 This exception signals problems in the management of SCA component execution.
860

861 7.7 NoRegisteredCallbackException

862 The following snippet shows the *NoRegisteredCallbackException*.

863

```
864 package org.oasisopen.sca;  
865  
866 public class NoRegisteredCallbackException extends  
867     ServiceRuntimeException {  
868     ...  
869 }
```

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

873 7.8 ServiceUnavailableException

874 The following snippet shows the *ServiceUnavailableException*.

875

```
876 package org.oasisopen.sca;  
877  
878 public class ServiceUnavailableException extends ServiceRuntimeException {  
879     ...  
880 }
```

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

886 7.9 InvalidServiceException

887 The following snippet shows the *InvalidServiceException*.

888

```
889 package org.oasisopen.sca;  
890  
891 public class InvalidServiceException extends ServiceRuntimeException {  
892     ...  
893 }
```

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

898 7.10 ConversationEndedException

899 The following snippet shows the *ConversationEndedException*.

```
900  
901 package org.oasisopen.sca;  
902  
903 public class ConversationEndedException extends ServiceRuntimeException {  
904     ...  
905 }  
906
```

907 8 Java Annotations

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

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

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

918 8.1 @AllowsPassByReference

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

920

```
921 package org.oasisopen.sca.annotations;  
922  
923 import static java.lang.annotation.ElementType.TYPE;  
924 import static java.lang.annotation.ElementType.METHOD;  
925 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
926 import java.lang.annotation.Retention;  
927 import java.lang.annotation.Target;  
928  
929 @Target({TYPE, METHOD})  
930 @Retention(RUNTIME)  
931 public @interface AllowsPassByReference {  
932  
933 }  
934
```

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

943 *@AllowsPassByReference* has no attributes

944

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

947

```
948 @AllowsPassByReference  
949 public String hello(String message) {  
950     ...  
951 }  
952
```

953 **8.2 @ Authentication**

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

```
955 package org.oasisopen.sca.annotations;  
956  
957 import static java.lang.annotation.ElementType.FIELD;  
958 import static java.lang.annotation.ElementType.METHOD;  
959 import static java.lang.annotation.ElementType.PARAMETER;  
960 import static java.lang.annotation.ElementType.TYPE;  
961 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
962 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
963  
964 import java.lang.annotation.Inherited;  
965 import java.lang.annotation.Retention;  
966 import java.lang.annotation.Target;  
967  
968 @Inherited  
969 @Target({TYPE, FIELD, METHOD, PARAMETER})  
970 @Retention(RUNTIME)  
971 @Intent(Authentication.AUTHENTICATION)  
972 public interface Authentication {  
973     String AUTHENTICATION = SCA_PREFIX + "authentication";  
974     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";  
975     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";  
976  
977     /**  
978      * List of authentication qualifiers (such as "message" or "transport").  
979      *  
980      * @return authentication qualifiers  
981      */  
982     @Qualifier  
983     String[] value() default "";  
984 }  
985  
986 }  
987
```

988 The SCA_PREFIX constant is defined in the Constants interface:

```
989 package org.oasisopen.sca;  
990  
991 public interface Constants {  
992     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";  
993     String SCA_PREFIX = "{"+SCA_NS+"}";  
994 }  
995
```

996 The **@Authentication** annotation is used to indicate that the invocation requires authentication.
997 Please check **10.3 Application of Intent Annotations** for samples and details.
998
999

1000 **8.28.3 @Callback**

1001 The following Java code defines shows the **@Callback** annotation:
1002

Formatted: Bullets and Numbering


```
1003 package org.oasisopen.sca.annotations;
1004
1005 import static java.lang.annotation.ElementType.TYPE;
1006 import static java.lang.annotation.ElementType.METHOD;
1007 import static java.lang.annotation.ElementType.FIELD;
1008 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1009 import java.lang.annotation.Retention;
1010 import java.lang.annotation.Target;
```

```
1011
1012 @Target(TYPE, METHOD, FIELD)
1013 @Retention(RUNTIME)
1014 public @interface Callback {
1015
1016     Class<?> value() default Void.class;
1017 }
1018
1019
```

1020 The @Callback annotation is used to annotate a service interface with a callback interface, which
1021 takes the Java Class object of the callback interface as a parameter.

1022 The @Callback annotation has the following attribute:

- 1023 • **value** – the name of a Java class file containing the callback interface

1024

1025 The @Callback annotation may also be used to annotate a method or a field of an SCA
1026 implementation class, in order to have a callback object injected

1027

1028 The following snippet shows a @Callback annotation on an interface:

1029

```
1030 @Remotable
1031 @Callback(MyServiceCallback.class)
1032 public interface MyService {
1033
1034     void someAsyncMethod(String arg);
1035 }
1036
```

1037 An example use of the @Callback annotation to declare a callback interface follows:

1038

```
1039 package somepackage;
1040 import org.oasisopen.sca.annotations.Callback;
1041 import org.oasisopen.sca.annotations.Remotable;
1042 @Remotable
1043 @Callback(MyServiceCallback.class)
1044 public interface MyService {
1045
1046     void someMethod(String arg);
1047 }
1048
1049 @Remotable
1050 public interface MyServiceCallback {
1051
1052     void receiveResult(String result);
1053 }
1054
```

1055 In this example, the implied component type is:

```
1056
1057 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >
1058
1059     <service name="MyService">
1060         <interface.java interface="somepackage.MyService"
1061             callbackInterface="somepackage.MyServiceCallback"/>
1062     </service>
1063 </componentType>
```

1064 **8.38.4 @ComponentName**

1065 The following Java code defines the **@ComponentName** annotation:

```
1066
1067 package org.oasisopen.sca.annotations;
1068
1069 import static java.lang.annotation.ElementType.METHOD;
1070 import static java.lang.annotation.ElementType.FIELD;
1071 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1072 import java.lang.annotation.Retention;
1073 import java.lang.annotation.Target;
1074
1075 @Target({METHOD, FIELD})
1076 @Retention(RUNTIME)
1077 public @interface ComponentName {
1078
1079 }
1080
```

1081 The @ComponentName annotation is used to denote a Java class field or setter method that is
1082 used to inject the component name.

1083

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

1085

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

1089 The following snippet shows a component name setter method sample.

1090

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

1096 **8.5 @Confidentiality**

1097

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

1099

```
1100 package org.oasisopen.sca.annotations;
1101
1102 import static java.lang.annotation.ElementType.FIELD;
```

Formatted: Bullets and Numbering

```

1103 import static java.lang.annotation.ElementType.METHOD;
1104 import static java.lang.annotation.ElementType.PARAMETER;
1105 import static java.lang.annotation.ElementType.TYPE;
1106 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1107 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1108
1109 import java.lang.annotation.Inherited;
1110 import java.lang.annotation.Retention;
1111 import java.lang.annotation.Target;
1112
1113 @Inherited
1114 @Target({TYPE, FIELD, METHOD, PARAMETER})
1115 @Retention(RUNTIME)
1116 @Intent(Confidentiality.CONFIDENTIALITY)
1117 public @interface Confidentiality {
1118     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
1119     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
1120     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
1121
1122     /**
1123      * List of confidentiality qualifiers (such as "message" or "transport").
1124      *
1125      * @return confidentiality qualifiers
1126      */
1127     @Qualifier
1128     String[] value() default "";
1129 }
1130 The @Confidentiality annotation is used to indicate that the invocation requires confidentiality.
1131 Please check 10.3 Application of Intent Annotations for samples and details.
1132
1133

```

1134 **8.48.6 @Constructor**

1135 The following Java code defines the **@Constructor** annotation:

```

1136 package org.oasisopen.sca.annotations;
1137
1138 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1139 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1140 import java.lang.annotation.Retention;
1141 import java.lang.annotation.Target;
1142
1143 @Target (CONSTRUCTOR)
1144 @Retention (RUNTIME)
1145 public @interface Constructor { }
1146
1147

```

1148 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1149 Java component implementation. If this constructor has parameters, each of these parameters
1150 MUST have either a @Property annotation or a @Reference annotation.

1151 The following snippet shows a sample for the @Constructor annotation.

```

1152
1153 public class HelloServiceImpl implements HelloService {
1154
1155     public HelloServiceImpl () {

```

```

1156     ...
1157     }
1158
1159     @Constructor
1160     public HelloServiceImpl(@Property(name="someProperty") String
1161     someProperty ){
1162         ...
1163     }
1164
1165     public String hello(String message) {
1166         ...
1167     }
1168 }

```

1169 [8.58.7 @Context](#)

1170 The following Java code defines the *@Context* annotation:

```

1171
1172 package org.oasisopen.sca.annotations;
1173
1174 import static java.lang.annotation.ElementType.METHOD;
1175 import static java.lang.annotation.ElementType.FIELD;
1176 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1177 import java.lang.annotation.Retention;
1178 import java.lang.annotation.Target;
1179
1180 @Target({METHOD, FIELD})
1181 @Retention(RUNTIME)
1182 public @interface Context {
1183
1184 }
1185

```

1186 The @Context annotation is used to denote a Java class field or a setter method that is used to
1187 inject a composite context for the component. The type of context to be injected is defined by the
1188 type of the Java class field or type of the setter method input argument; the type is either
1189 **ComponentContext** or **RequestContext**.

1190 The @Context annotation has no attributes.

1191

1192 The following snippet shows a ComponentContext field definition sample.

1193

```

1194 @Context
1195 protected ComponentContext context;
1196

```

1197 The following snippet shows a RequestContext field definition sample.

1198

```

1199 @Context
1200 protected RequestContext context;

```

1201 [8.68.8 @Conversational](#)

1202 The following Java code defines the *@Conversational* annotation:

1203

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

```

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

```

1214 The @Conversational annotation is used on a Java interface to denote a conversational service contract.

1217 The @Conversational annotation has no attributes.

1218 The following snippet shows a sample for the @Conversational annotation.

```

1219 package services.hello;
1220
1221 import org.oasisopen.sca.annotations.Conversational;
1222
1223 @Conversational
1224 public interface HelloService {
1225     void setName(String name);
1226     String sayHello();
1227 }

```

1228 **8-78.9 @ConversationAttributes**

1229 The following Java code defines the *@ConversationAttributes* annotation:

```

1230
1231 package org.oasisopen.sca.annotations;
1232
1233 import static java.lang.annotation.ElementType.TYPE;
1234 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1235 import java.lang.annotation.Retention;
1236 import java.lang.annotation.Target;
1237
1238 @Target (TYPE)
1239 @Retention (RUNTIME)
1240 public @interface ConversationAttributes {
1241
1242     String maxIdleTime() default "";
1243     String maxAge() default "";
1244     boolean singlePrincipal() default false;
1245 }
1246

```

1247 The @ConversationAttributes annotation is used to define a set of attributes which apply to conversational interfaces of services or references of a Java class. The annotation has the following attributes:

- 1250 • **maxIdleTime (optional)** - The maximum time that can pass between successive operations within a single conversation. If more time than this passes, then the container may end the conversation.
- 1251
- 1252
- 1253 • **maxAge (optional)** - The maximum time that the entire conversation can remain active. If more time than this passes, then the container may end the conversation.
- 1254

Formatted: Bullets and Numbering

- 1255
- **singlePrincipal (optional)** – If true, only the principal (the user) that started the conversation has authority to continue the conversation. The default value is false.
- 1256

1257

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

1261

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

1264

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

1267

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

1276 **8.88.10 @ConversationID**

1277 The following Java code defines the **@ConversationID** annotation:

1278

```
1279 package org.oasisopen.sca.annotations;  
1280  
1281 import static java.lang.annotation.ElementType.METHOD;  
1282 import static java.lang.annotation.ElementType.FIELD;  
1283 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1284 import java.lang.annotation.Retention;  
1285 import java.lang.annotation.Target;
```

1286

```
1287 @Target({METHOD, FIELD})  
1288 @Retention(RUNTIME)  
1289 public @interface ConversationID {  
1290  
1291 }  
1292
```

1293 The @ConversationID annotation is used to annotate a Java class field or setter method that is
1294 used to inject the conversation ID. System generated conversation IDs are always strings, but
1295 application generated conversation IDs may be other complex types.

1296 The following snippet shows a conversation ID field definition sample.

1297

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

1300

1301 The type of the field is not necessarily String.

1302

Formatted: Bullets and Numbering

1303 [8.98.11 @Destroy](#)

1304 The following Java code defines the **@Destroy** annotation:

```
1305
1306 package org.oasisopen.sca.annotations;
1307
1308 import static java.lang.annotation.ElementType.METHOD;
1309 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1310 import java.lang.annotation.Retention;
1311 import java.lang.annotation.Target;
1312
1313 @Target(METHOD)
1314 @Retention(RUNTIME)
1315 public @interface Destroy {
1316 }
1317
1318
```

1319 The @Destroy annotation is used to denote a single Java class method that will be called when the
1320 scope defined for the implementation class ends. The method MAY have any access modifier and
1321 MUST have a void return type and no arguments.

1322 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1323 when the scope defined for the implementation class ends. If the implementation class has a
1324 method with an @Destroy annotation that does not match these criteria, the SCA runtime MUST
1325 NOT instantiate the implementation class.

1326
1327 The following snippet shows a sample for a destroy method definition.

```
1328
1329 @Destroy
1330 public void myDestroyMethod() {
1331     ...
1332 }
```

1333 [8.108.12 @EagerInit](#)

1334 The following Java code defines the **@EagerInit** annotation:

```
1335
1336 package org.oasisopen.sca.annotations;
1337
1338 import static java.lang.annotation.ElementType.TYPE;
1339 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1340 import java.lang.annotation.Retention;
1341 import java.lang.annotation.Target;
1342
1343 @Target(TYPE)
1344 @Retention(RUNTIME)
1345 public @interface EagerInit {
1346 }
1347
1348
```

1349 The **@EagerInit** annotation is used to annotate the Java class of a COMPOSITE scoped
1350 implementation for eager initialization. When marked for eager initialization, the composite scoped
1351 instance is created when its containing component is started.

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

1352 **8.118.13 @EndsConversation**

1353 The following Java code defines the *@EndsConversation* annotation:

```
1354
1355 package org.oasisopen.sca.annotations;
1356
1357 import static java.lang.annotation.ElementType.METHOD;
1358 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1359 import java.lang.annotation.Retention;
1360 import java.lang.annotation.Target;
1361
1362 @Target (METHOD)
1363 @Retention (RUNTIME)
1364 public @interface EndsConversation {
1365
1366 }
1367
1368
```

1369 The @EndsConversation annotation is used to denote a method on a Java interface that is called
1370 to end a conversation.

1371 The @EndsConversation annotation has no attributes.

1372 The following snippet shows a sample using the @EndsConversation annotation.

```
1373 package services.shoppingbasket;
1374
1375 import org.oasisopen.sca.annotations.EndsConversation;
1376
1377 public interface ShoppingBasket {
1378     void addItem(String itemID, int quantity);
1379
1380     @EndsConversation
1381     void buy();
1382 }

```

1383 **8.128.14 @Init**

1384 The following Java code defines the *@Init* annotation:

```
1385
1386 package org.oasisopen.sca.annotations;
1387
1388 import static java.lang.annotation.ElementType.METHOD;
1389 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1390 import java.lang.annotation.Retention;
1391 import java.lang.annotation.Target;
1392
1393 @Target (METHOD)
1394 @Retention (RUNTIME)
1395 public @interface Init {
1396
1397 }
1398
1399
```

1400 The @Init annotation is used to denote a single Java class method that is called when the scope
1401 defined for the implementation class starts. The method MAY have any access modifier and MUST
1402 have a void return type and no arguments.

1403 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1404 after all property and reference injection is complete. If the implementation class has a method
1405 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1406 instantiate the implementation class.

1407 The following snippet shows an example of an init method definition.

1408

```
1409 @Init  
1410 public void myInitMethod() {  
1411     ...  
1412 }  
1413
```

1414 **8.15 @Integrity**

1415

1416 The following Java code defines the *@Integrity* annotation:

1417

```
1418 package org.oasisopen.sca.annotations;  
1419  
1420 import static java.lang.annotation.ElementType.FIELD;  
1421 import static java.lang.annotation.ElementType.METHOD;  
1422 import static java.lang.annotation.ElementType.PARAMETER;  
1423 import static java.lang.annotation.ElementType.TYPE;  
1424 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1425 import static org.oasisopen.Constants.SCA_PREFIX;  
1426  
1427 import java.lang.annotation.Inherited;  
1428 import java.lang.annotation.Retention;  
1429 import java.lang.annotation.Target;  
1430  
1431 @Inherited  
1432 @Target({TYPE, FIELD, METHOD, PARAMETER})  
1433 @Retention(RUNTIME)  
1434 @Intent(Integrity.INTEGRITY)  
1435 public @interface Integrity {  
1436     String INTEGRITY = SCA_PREFIX + "integrity";  
1437     String INTEGRITY_MESSAGE = INTEGRITY + ".message";  
1438     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";  
1439  
1440     /**  
1441      * List of integrity qualifiers (such as "message" or "transport").  
1442      *  
1443      * @return integrity qualifiers  
1444      */  
1445     @Qualifier  
1446     String[] value() default "";  
1447 }  
1448
```

1449 The *@Integrity* annotation is used to indicate that the invocation requires integrity. Please check

1450 **10.3 Application of Intent Annotations** for samples and details.
1451

1452 **8.138.16 @OneWay**

1453 The following Java code defines the *@OneWay* annotation:

1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468

1469
1470
1471

1472

1473

1474
1475
1476
1477
1478
1479
1480
1481
1482

```
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;

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

}
```

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

The @OneWay annotation has no attributes.

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

```
package services.hello;

import org.oasisopen.sca.annotations.OneWay;

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

8.17 @PolicySets

The following Java code defines the *@PolicySets* 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 java.lang.annotation.Retention;
import java.lang.annotation.Target;

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

1507 }
1508

1509 [The @PolicySet annotation is used to describe SCA Policy Sets. Please check **10.5 Policy Set Annotations** for samples and details.](#)

1510

1511

1512 **8.148.18 @Property**

1513 The following Java code defines the *@Property* annotation:

1514

```
1515 package org.oasisopen.sca.annotations;  
1516  
1517 import static java.lang.annotation.ElementType.METHOD;  
1518 import static java.lang.annotation.ElementType.FIELD;  
1519 import static java.lang.annotation.ElementType.PARAMETER;  
1520 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1521 import java.lang.annotation.Retention;  
1522 import java.lang.annotation.Target;  
1523  
1524 @Target({METHOD, FIELD, PARAMETER})  
1525 @Retention(RUNTIME)  
1526 public @interface Property {  
1527  
1528     String name() default "";  
1529     boolean required() default true;  
1530 }  
1531
```

1532

1533 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1534 parameter that is used to inject an SCA property value. The type of the property injected, which
1535 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1536 the type of the input parameter of the setter method or constructor.

1537 The @Property annotation may be used on fields, on setter methods or on a constructor method
1538 parameter. However, the @Property annotation MUST NOT be used on a class field that is declared
1539 as final.

1540 Properties may also be injected via setter methods even when the @Property annotation is not
1541 present. However, the @Property annotation must be used in order to inject a property onto a
1542 non-public field. In the case where there is no @Property annotation, the name of the property is
1543 the same as the name of the field or setter.

1544 Where there is both a setter method and a field for a property, the setter method is used.

1545

1546 The @Property annotation has the following attributes:

- 1547 • **name (optional)** – the name of the property. For a field annotation, the default is the
1548 name of the field of the Java class. For a setter method annotation, the default is the
1549 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a
1550 constructor parameter annotation, there is no default and the name attribute MUST be
1551 present.
- 1552 • **required (optional)** – specifies whether injection is required, defaults to true. For a
1553 constructor parameter annotation, this attribute MUST have the value true.

1554

1555 The following snippet shows a property field definition sample.

Formatted: English (U.S.)

Formatted: Bullets and Numbering

```
1556
1557     @Property(name="currency", required=true)
1558     protected String currency;
```

1559
1560 The following snippet shows a property setter sample

```
1561
1562     @Property(name="currency", required=true)
1563     public void setCurrency( String theCurrency ) {
1564         ....
1565     }
```

1566
1567 If the property is defined as an array or as any type that extends or implements
1568 *java.util.Collection*, then the implied component type has a property with a *many* attribute set to
1569 true.

1570
1571 The following snippet shows the definition of a configuration property using the @Property
1572 annotation for a collection.

```
1573
1574     ...
1575     private List<String> helloConfigurationProperty;
1576
1577     @Property(required=true)
1578     public void setHelloConfigurationProperty(List<String> property) {
1579         helloConfigurationProperty = property;
1580     }
1581     ...
1582
```

1583 8.19 @Qualifier

1584
1585 The following Java code defines the @Qualifier annotation:

```
1586     package org.oasisopen.sca.annotations;
1587
1588     import static java.lang.annotation.ElementType.METHOD;
1589     import static java.lang.annotation.RetentionPolicy.RUNTIME;
1590
1591     import java.lang.annotation.Retention;
1592     import java.lang.annotation.Target;
1593
1594     @Target(METHOD)
1595     @Retention(RUNTIME)
1596     public @interface Qualifier {
1597     }
```

1600 The @Qualifier annotation can be applied to an attribute as an @Intent annotation to indicate the
1601 attribute provides qualifiers for the intent. Please check 10.3 Application of Intent
1602 Annotations for samples and details.

1603

1604

1605 **8.158.20 @Reference**1606 The following Java code defines the **@Reference** annotation:

1607

```

1608 package org.oasisopen.sca.annotations;
1609
1610 import static java.lang.annotation.ElementType.METHOD;
1611 import static java.lang.annotation.ElementType.FIELD;
1612 import static java.lang.annotation.ElementType.PARAMETER;
1613 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1614 import java.lang.annotation.Retention;
1615 import java.lang.annotation.Target;
1616 @Target({METHOD, FIELD, PARAMETER})
1617 @Retention(RUNTIME)
1618 public @interface Reference {
1619
1620     String name() default "";
1621     boolean required() default true;
1622 }
1623

```

1624 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
 1625 constructor parameter that is used to inject a service that resolves the reference. The interface of
 1626 the service injected is defined by the type of the Java class field or the type of the input parameter
 1627 of the setter method or constructor.

1628 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1629 References may also be injected via setter methods even when the @Reference annotation is not
 1630 present. However, the @Reference annotation must be used in order to inject a reference onto a
 1631 non-public field. In the case where there is no @Reference annotation, the name of the reference
 1632 is the same as the name of the field or setter.

1633 Where there is both a setter method and a field for a reference, the setter method is used.

1634 The @Reference annotation has the following attributes:

- 1635 • **name (optional)** – the name of the reference. For a field annotation, the default is the
 1636 name of the field of the Java class. For a setter method annotation, the default is the
 1637 JavaBeans property name corresponding to the setter method name. For a constructor
 1638 parameter annotation, there is no default and the name attribute MUST be present.
- 1639 • **required (optional)** – whether injection of service or services is required. Defaults to true.
 1640 For a constructor parameter annotation, this attribute MUST have the value true.

1641

1642 The following snippet shows a reference field definition sample.

1643

```

1644 @Reference(name="stockQuote", required=true)
1645 protected StockQuoteService stockQuote;

```

1646

1647 The following snippet shows a reference setter sample

1648

```

1649 @Reference(name="stockQuote", required=true)
1650 public void setStockQuote( StockQuoteService theSQService ) {

```

1651 ...
1652 }

1653

1654 The following fragment from a component implementation shows a sample of a service reference
1655 using the @Reference annotation. The name of the reference is "helloService" and its type is
1656 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1657 helloService reference.

1658

```
1659 package services.hello;  
1660  
1661 private HelloService helloService;  
1662  
1663 @Reference(name="helloService", required=true)  
1664 public setHelloService(HelloService service) {  
1665     helloService = service;  
1666 }  
1667  
1668 public void clientMethod() {  
1669     String result = helloService.hello("Hello World!");  
1670     ...  
1671 }  
1672
```

1673 The presence of a @Reference annotation is reflected in the componentType information that the
1674 runtime generates through reflection on the implementation class. The following snippet shows
1675 the component type for the above component implementation fragment.

1676

```
1677 <?xml version="1.0" encoding="ASCII"?>  
1678 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
1679  
1680     <!-- Any services offered by the component would be listed here -->  
1681     <reference name="helloService" multiplicity="1..1">  
1682         <interface.java interface="services.hello.HelloService"/>  
1683     </reference>  
1684  
1685 </componentType>  
1686
```

1687 If the reference is not an array or collection, then the implied component type has a reference
1688 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1689 attribute – 1..1 applies if required=true.

1690

1691 If the reference is defined as an array or as any type that extends or implements **java.util.Collection**,
1692 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending
1693 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if
1694 required=true.

1695

1696 The following fragment from a component implementation shows a sample of a service reference
1697 definition using the @Reference annotation on a java.util.List. The name of the reference is
1698 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1699 services referenced by the helloServices reference. In this case, at least one HelloService should
1700 be present, so **required** is true.

1701

```
1702     @Reference(name="helloServices", required=true)
```

```

1703     protected List<HelloService> helloServices;
1704
1705     public void clientMethod() {
1706
1707         ...
1708         for (int index = 0; index < helloServices.size(); index++) {
1709             HelloService helloService =
1710                 (HelloService)helloServices.get(index);
1711             String result = helloService.hello("Hello World!");
1712         }
1713         ...
1714     }
1715

```

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

```

1719
1720 <?xml version="1.0" encoding="ASCII"?>
1721 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1722
1723     <!-- Any services offered by the component would be listed here -->
1724     <reference name="helloServices" multiplicity="1..n">
1725         <interface.java interface="services.hello.HelloService"/>
1726     </reference>
1727
1728 </componentType>

```

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

1733 **8-15-18.20.1 Reinjection**

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

- 1737 1. The component MUST NOT be STATELESS scoped.
- 1738 2. The reference MUST use either field-based injection or setter injection. References that are
1739 injected through constructor injection MUST NOT be changed. Setter injection allows for
1740 code in the setter method to perform processing in reaction to a change.
- 1741 3. If the reference has a conversational interface, then reinjection MUST NOT occur while the
1742 conversation is active.

1743 If a reference target changes and the reference is not reinjected, the reference MUST continue to
1744 work as if the reference target was not changed.

1745 If an operation is called on a reference where the target of that reference has been undeployed,
1746 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference
1747 where the target of the reference has become unavailable for some reason, the SCA runtime
1748 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the
1749 reference MAY continue to work, depending on the runtime and the type of change that was made.
1750 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1751 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
1752 corresponds to the reference that is passed as a parameter to cast(). If the reference is
1753 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
1754 to work as if the reference target was not changed. If the target of a ServiceReference has been

Formatted: Bullets and Numbering

1755 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is
 1756 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the
 1757 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the
 1758 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to
 1759 work, depending on the runtime and the type of change that was made. If it doesn't work, the
 1760 exception thrown will depend on the runtime and the cause of the failure.

1761 A reference or ServiceReference accessed through the component context by calling getService()
 1762 or getServiceReference() MUST correspond to the current configuration of the domain. This
 1763 applies whether or not reinjection has taken place. If the target has been undeployed or has
 1764 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
 1765 and attempts to call business methods SHOULD throw an exception as described above. If the
 1766 target has changed, the result SHOULD be a reference to the changed service.

1767 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
 1768 means that in the cases listed above where reference reinjection is not allowed, the array or
 1769 Collection for the reference MUST NOT change its contents. In cases where the contents of a
 1770 reference collection MAY change, then for references that use setter injection, the setter method
 1771 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be
 1772 the same array or Collection object previously injected to the component.

1773

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 InvalidServiceException.	Business methods SHOULD throw InvalidServiceException.	Result SHOULD be a reference to the undeployed or unavailable service. Business methods SHOULD throw InvalidServiceException.
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 ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().			

1774

1775

1776 [8.168.21 @Remotable](#)

1777 The following Java code defines the `@Remotable` annotation:

1778

```
1779 package org.oasisopen.sca.annotations;
1780
1781 import static java.lang.annotation.ElementType.TYPE;
1782 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1783 import java.lang.annotation.Retention;
1784 import java.lang.annotation.Target;
```

1785

1786

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

1792

1793 The `@Remotable` annotation is used to specify a Java service interface as remotable. A remotable
1794 service can be published externally as a service and must be translatable into a WSDL portType.

1795 The `@Remotable` annotation has no attributes.

1796

1797 The following snippet shows the Java interface for a remotable service with its `@Remotable`
1798 annotation.

```
1799 package services.hello;
1800
1801 import org.oasisopen.sca.annotations.*;
1802
1803 @Remotable
1804 public interface HelloService {
1805
1806     String hello(String message);
1807
1808 }
```

1808

1809 The style of remotable interfaces is typically *coarse grained* and intended for *loosely coupled*
1810 interactions. Remotable service interfaces are not allowed to make use of method *overloading*.

1811

1812 Complex data types exchanged via remotable service interfaces MUST be compatible with the
1813 marshalling technology used by the service binding. For example, if the service is going to be
1814 exposed using the standard Web Service binding, then the parameters MAY be JAXB [JAX-B] types
1815 or Service Data Objects (SDOs) [SDO].

1816 Independent of whether the remotable service is called from outside of the composite that
1817 contains it or from another component in the same composite, the data exchange semantics are
1818 *by-value*.

1819 Implementations of remotable services may modify input data during or after an invocation and
1820 may modify return data after the invocation. If a remotable service is called locally or remotely,
1821 the SCA container is responsible for making sure that no modification of input data or post-
1822 invocation modifications to return data are seen by the caller.

1823

Formatted: Bullets and Numbering

1824 The following snippet shows a remotable Java service interface.

1825

```
1826 package services.hello;
1827
1828 import org.oasisopen.sca.annotations.*;
1829
1830 @Remotable
1831 public interface HelloService {
1832     String hello(String message);
1833 }
1834
1835 package services.hello;
1836
1837 import org.oasisopen.sca.annotations.*;
1838
1839 @Service(HelloService.class)
1840 public class HelloServiceImpl implements HelloService {
1841     public String hello(String message) {
1842         ...
1843     }
1844 }
1845
1846
1847
```

1848 8.22 @Requires

1849

1850 The following Java code defines the [*@Requires*](#) annotation:

1851

```
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](#)
1877 | [specific intents using @Intent annotation. Please check 10.1 General Intent Annotations](#)
1878 | [for samples and details.](#)
1879 |
1880 |

1881 | **8.178.23 @Scope**

1882 | The following Java code defines the **@Scope** annotation:

```
1883 |  
1884 | package org.oasisopen.sca.annotations;  
1885 |  
1886 | import static java.lang.annotation.ElementType.TYPE;  
1887 | import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1888 | import java.lang.annotation.Retention;  
1889 | import java.lang.annotation.Target;  
1890 |  
1891 | @Target (TYPE)  
1892 | @Retention (RUNTIME)  
1893 | public @interface Scope {  
1894 |  
1895 |     String value() default "STATELESS";  
1896 | }  
1897 |
```

1897 | The @Scope annotation may only be used on a service's implementation class. It is an error to use
1898 | this annotation on an interface.

1899 | The @Scope annotation has the following attribute:

- 1900 | • **value** – the name of the scope.
1901 | For 'STATELESS' implementations, a different implementation instance may be used to
1902 | service each request. Implementation instances may be newly created or be drawn from a
1903 | pool of instances.
1904 | SCA defines the following scope names, but others can be defined by particular Java-
1905 | based implementation types:
1906 | STATELESS
1907 | COMPOSITE
1908 | CONVERSATION

1909 | The default value is STATELESS, except for an implementation offering a @Conversational service,
1910 | which has a default scope of CONVERSATION. See section [2.22-2](#) for more details of the SCA-
1911 | defined scopes.

1912 | The following snippet shows a sample for a CONVERSATION scoped service implementation:

```
1913 | package services.hello;  
1914 |  
1915 | import org.oasisopen.sca.annotations.*;  
1916 |  
1917 | @Service (HelloService.class)  
1918 | @Scope ("CONVERSATION")  
1919 | public class HelloServiceImpl implements HelloService {  
1920 |  
1921 |     public String hello (String message) {  
1922 |         ...  
1923 |     }  
1924 | }  
1925 |
```

Formatted: Bullets and Numbering

1926 **8.188.24 @Service**

Formatted: Bullets and Numbering

1927 The following Java code defines the **@Service** annotation:

1928

```
1929 package org.oasisopen.sca.annotations;  
1930  
1931 import static java.lang.annotation.ElementType.TYPE;  
1932 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1933 import java.lang.annotation.Retention;  
1934 import java.lang.annotation.Target;  
1935  
1936 @Target(TYPE)  
1937 @Retention(RUNTIME)  
1938 public @interface Service {  
1939  
1940     Class<?>[] interfaces() default {};  
1941     Class<?> value() default Void.class;  
1942 }  
1943
```

1944 The @Service annotation is used on a component implementation class to specify the SCA services
1945 offered by the implementation. The class need not be declared as implementing all of the
1946 interfaces implied by the services, but all methods of the service interfaces must be present. A
1947 class used as the implementation of a service is not required to have a @Service annotation. If a
1948 class has no @Service annotation, then the rules determining which services are offered and what
1949 interfaces those services have are determined by the specific implementation type.

1950 The @Service annotation has the following attributes:

- 1951 • **interfaces** – The value is an array of interface or class objects that should be exposed as
1952 services by this component.
- 1953 • **value** – A shortcut for the case when the class provides only a single service interface.

1954 Only one of these attributes should be specified.

1955

1956 A @Service annotation with no attributes is meaningless, it is the same as not having the
1957 annotation there at all.

1958 The **service names** of the defined services default to the names of the interfaces or class, without
1959 the package name.

1960 A component MUST NOT have two services with the same Java simple name. If a Java
1961 implementation needs to realize two services with the same Java simple name then this can be
1962 achieved through subclassing of the interface.

1963 The following snippet shows an implementation of the HelloService marked with the @Service
1964 annotation.

```
1965 package services.hello;  
1966  
1967 import org.oasisopen.sca.annotations.Service;  
1968  
1969 @Service(HelloService.class)  
1970 public class HelloServiceImpl implements HelloService {  
1971  
1972     public void hello(String name) {  
1973         System.out.println("Hello " + name);  
1974     }  
1975 }  
1976
```

1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994

8.25 Security Implementation Policy Annotations

JSR 250 "Common Annotations for the Java Platform" defines the following annotations that can be used for implementation security policy:

[javax.annotation.security.RunAs](#)

[javax.annotation.security.RolesAllowed](#)

[javax.annotation.security.PermitAll](#)

[javax.annotation.security.DenyAll](#)

[javax.annotation.security.DeclareRoles](#)

Based on JSR 250, the RunAs , DeclareRoles annotations can be specified on a class; the RolesAllowed , PermitAll , DenyAll annotations can be specified on a class or on method(s).

Please check JSR250 and SCA Policy spec for details on the meaning of the annotations . Please check the section [10.6.2] Security Implementation Policy for the details on how these annotations are mapped into Policy framework.

1995

9 WSDL to Java and Java to WSDL

1996
1997
1998

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.

1999
2000
2001
2002
2003

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.oasisopen.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.

2004
2005
2006
2007
2008

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.

2009

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

2010

- No support for holders

2011

2012
2013

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

2014

9.1 JAX-WS Client Asynchronous API for a Synchronous Service

2015
2016
2017
2018
2019

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.

2020
2021
2022
2023
2024

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.

2025

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

2026

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

2027

b. the same parameter signature as M, and

2028

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

2029

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

2030

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

2031

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

2032

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

2033

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

2034

c. a return type of Future<?>

2035

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

2036

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

2037
2038

```
<!-- WSDL extract -->
<message name="getPrice">
```

```
2039 <part name="ticker" type="xsd:string"/>
2040 </message>
2041
2042 <message name="getPriceResponse">
2043 <part name="price" type="xsd:float"/>
2044 </message>
2045
2046 <portType name="StockQuote">
2047 <operation name="getPrice">
2048 <input message="tns:getPrice"/>
2049 <output message="tns:getPriceResponse"/>
2050 </operation>
2051 </portType>
```

2052

2053 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2054 // asynchronous mapping
2055 @WebService
2056 public interface StockQuote {
2057     float getPrice(String ticker);
2058     Response<Float> getPriceAsync(String ticker);
2059     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2060 }
```

2061

2062 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2063 // synchronous mapping
2064 @WebService
2065 public interface StockQuote {
2066     float getPrice(String ticker);
2067 }
```

2068

2069 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2070 example, if the client implementation uses the asynchronous form of the interface, the two
2071 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2072 WS specification.

2073

10 Policy Annotations for Java

2074
2075
2076
2077
2078

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.

2079
2080
2081
2082

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.

2083
2084
2085
2086
2087
2088
2089
2090

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.

2091
2092
2093
2094
2095

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.

2096
2097
2098
2099

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.

2100

2101 10.1 General Intent Annotations

2102
2103

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.

2104

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

2105
2106
2107

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:

2108

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

2109
2110

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.

2111
2112
2113

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:

2114
2115

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

2116

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

2117

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


2118 Notice that, by convention, qualified intents include the qualifier as part of the name of the
2119 constant, separated by an underscore. These intent constants are defined in the file that defines
2120 an annotation for the intent (annotations for intents, and the formal definition of these constants,
2121 are covered in a following section).

2122 Multiple intents (qualified or not) are expressed as separate strings within an array declaration.

2123 An example of the @Requires annotation with 2 qualified intents (from the Security domain)
2124 follows:

```
2125  
2126     @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2127
2128 This attaches the intents "confidentiality.message" and "integrity.message".

2129 The following is an example of a reference requiring support for confidentiality:

```
2130     package org.oasisopen.sca.annotations;  
2131  
2132     import static org.oasisopen.sca.annotations.Confidentiality.*;  
2133  
2134     public class Foo {  
2135         @Requires(CONFIDENTIALITY)  
2136         @Reference  
2137         public void setBar(Bar bar) {  
2138             ...  
2139         }  
2140     }
```

2141 Users may also choose to only use constants for the namespace part of the QName, so that they
2142 may add new intents without having to define new constants. In that case, this definition would
2143 instead look like this:

```
2144     package org.oasisopen.sca.annotations;  
2145  
2146     import static org.oasisopen.sca.Constants.*;  
2147  
2148     public class Foo {  
2149         @Requires(SCA_PREFIX+"confidentiality")  
2150         @Reference  
2151         public void setBar(Bar bar) {  
2152             ...  
2153         }  
2154     }
```

2155
2156 The formal syntax for the @Requires annotation follows:

```
2157     @Requires( "qualifiedIntent" | {"qualifiedIntent" [, "qualifiedIntent"]}
```

2158 where

2159 qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2

2160

2161 The following shows the formal definition of the @Requires annotation:

2162

```
2163 package org.oasisopen.sca.annotations;  
2164 import static java.lang.annotation.ElementType.TYPE;  
2165 import static java.lang.annotation.ElementType.METHOD;  
2166 import static java.lang.annotation.ElementType.FIELD;  
2167 import static java.lang.annotation.ElementType.PARAMETER;  
2168 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2169 import java.lang.annotation.Retention;  
2170 import java.lang.annotation.Target;  
2171 import java.lang.annotation.Inherited;
```

2172

```
2173 @Inherited  
2174 @Retention(RUNTIME)  
2175 @Target({TYPE, METHOD, FIELD, PARAMETER})
```

2176

```
2177 public @interface Requires {  
2178     String[] value() default "";  
2179 }
```

2180 The SCA_NS constant is defined in the Constants interface:

2181

2182

```
2183 package org.oasisopen.sca;  
  
2184 public interface Constants {  
2185     String SCA_NS="http://docs.oasis-open.org/ns/opensca/sca/200712";  
2186     String SCA_PREFIX = "("+SCA_NS+")";  
2187 }
```

2187

2188 10.2 Specific Intent Annotations

2189 In addition to the general intent annotation supplied by the @Requires annotation described
2190 above, it is also possible to have Java annotations that correspond to specific policy intents. SCA
2191 provides a number of these specific intent annotations and it is also possible to create new specific
2192 intent annotations for any intent.

2193 The general form of these specific intent annotations is an annotation with a name derived from
2194 the name of the intent itself. If the intent is a qualified intent, qualifiers are supplied as an
2195 attribute to the annotation in the form of a string or an array of strings.

2196 For example, the SCA confidentiality intent described in [the section on General Intent Annotations](#)
2197 using the @Requires(CONFIDENTIALITY) intent can also be specified with the specific
2198 @Confidentiality intent annotation. The specific intent annotation for the "integrity" security intent
2199 is:

2200 @Integrity

2201 An example of a qualified specific intent for the "authentication" intent is:

2202 @Authentication({"message", "transport"})

2203 This annotation attaches the pair of qualified intents: "authentication.message" and
2204 "authentication.transport" (the sca: namespace is assumed in this both of these cases –
2205 "http://docs.oasis-open.org/ns/opencsa/sca/200712").

2206 The general form of specific intent annotations is:

2207 @<Intent>[(qualifiers)]

2208 where Intent is an NCName that denotes a particular type of intent.

2209 Intent ::= NCName

2210 qualifiers ::= "qualifier" | {"qualifier" [, "qualifier"] }

2211 qualifier ::= NCName | NCName/qualifier

2212

2213 10.2.1 How to Create Specific Intent Annotations

2214 SCA identifies annotations that correspond to intents by providing an @Intent annotation which
2215 must be used in the definition of an intent annotation.

2216 The @Intent annotation takes a single parameter, which (like the @Requires annotation) is the
2217 String form of the QName of the intent. As part of the intent definition, it is good practice
2218 (although not required) to also create String constants for the Namespace, the Intent and for
2219 Qualified versions of the Intent (if defined). These String constants are then available for use with
2220 the @Requires annotation and it should also be possible to use one or more of them as
2221 parameters to the @Intent annotation.

2222 Alternatively, the QName of the intent may be specified using separate parameters for the
2223 targetNamespace and the localPart for example:

2224 @Intent(targetNamespace=SCA_NS, localPart="confidentiality").

2225 The definition of the @Intent annotation is the following:

2226

2227 ~~package org.oasisopen.sca.annotations;~~

2228 ~~import static java.lang.annotation.ElementType.ANNOTATION_TYPE;~~

2229 ~~import static java.lang.annotation.RetentionPolicy.RUNTIME;~~

2230 ~~import java.lang.annotation.Retention;~~

2231 ~~import java.lang.annotation.Target;~~

2232 ~~import java.lang.annotation.Inherited;~~

2233

2234 ~~@Retention(RUNTIME)~~

2235 ~~@Target(ANNOTATION_TYPE)~~

2236 ~~public @interface Intent {~~

2237 ~~String value() default "";~~

2238 ~~String targetNamespace() default "";~~

2239 ~~String localPart() default "";~~

2240 ~~}~~

2241 When an intent can be qualified, it is good practice for the first attribute of the annotation to be a
2242 string (or an array of strings) which holds one or more qualifiers.

2243 In this case, the attribute's definition should be marked with the @Qualifier annotation. The
2244 @Qualifier tells SCA that the value of the attribute should be treated as a qualifier for the intent
2245 represented by the whole annotation. If more than one qualifier value is specified in an
2246 annotation, it means that multiple qualified forms are required. For example:

```
2247 @Confidentiality({"message", "transport"})
```

2248 implies that both of the qualified intents "confidentiality.message" and "confidentiality.transport"
2249 are set for the element to which the confidentiality intent is attached.

2250 ~~The following is the definition of the @Qualifier annotation.~~

```
2251  
2252 package org.oasisopen.sca.annotations;  
2253 import static java.lang.annotation.ElementType.METHOD;  
2254 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2255 import java.lang.annotation.Retention;  
2256 import java.lang.annotation.Target;  
2257 import java.lang.annotation.Inherited;  
2258  
2259 @Retention(RetentionPolicy.RUNTIME)  
2260 @Target(ElementType.METHOD)  
2261 public @interface Qualifier {  
2262     +  
2263
```

2264 Examples of the use of the @Intent and the @Qualifier annotations in the definition of specific
2265 intent annotations are shown in [the section dealing with Security Interaction Policy](#).

2266

2267 10.3 Application of Intent Annotations

2268 The SCA Intent annotations can be applied to the following Java elements:

- 2269 • Java class
- 2270 • Java interface
- 2271 • Method
- 2272 • Field

2273 Where multiple intent annotations (general or specific) are applied to the same Java element, they
2274 are additive in effect. An example of multiple policy annotations being used together follows:

```
2275 @Authentication  
2276 @Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

2277 In this case, the effective intents are "authentication", "confidentiality.message" and
2278 "integrity.message".

2279 If an annotation is specified at both the class/interface level and the method or field level, then
2280 the method or field level annotation completely overrides the class level annotation of the same
2281 type.

2282 The intent annotation can be applied either to classes or to class methods when adding annotated
2283 policy on SCA services. Applying an intent to the setter method in a reference injection approach
2284 allows intents to be defined at references.

2285 10.3.1 Inheritance And Annotation

2286 The inheritance rules for annotations are consistent with the common annotation specification, JSR
2287 250.

2288 The following example shows the inheritance relations of intents on classes, operations, and super
2289 classes.

```
2290  
2291 package services.hello;  
2292 import org.oasisopen.sca.annotations.Remotable;  
2293 import org.oasisopen.sca.annotations.Integrity;  
2294 import org.oasisopen.sca.annotations.Authentication;  
2295  
2296 @Integrity("transport")  
2297 @Authentication  
2298 public class HelloService {  
2299     @Integrity  
2300     @Authentication("message")  
2301     public String hello(String message) {...}  
2302  
2303     @Integrity  
2304     @Authentication("transport")  
2305     public String helloThere() {...}  
2306 }  
2307  
2308 package services.hello;  
2309 import org.oasisopen.sca.annotations.Remotable;  
2310 import org.oasisopen.sca.annotations.Confidentiality;  
2311 import org.oasisopen.sca.annotations.Authentication;  
2312  
2313 @Confidentiality("message")  
2314 public class HelloChildService extends HelloService {  
2315     @Confidentiality("transport")  
2316     public String hello(String message) {...}  
2317     @Authentication  
2318     String helloWorld() {...}  
2319 }
```

2320 Example 2a. Usage example of annotated policy and inheritance.

2321

2322 The effective intent annotation on the helloWorld method is Integrity("transport"),
2323 @Authentication, and @Confidentiality("message").

2324 The effective intent annotation on the hello method of the HelloChildService is
 2325 @Integrity("transport"), @Authentication, and @Confidentiality("transport"),
 2326 The effective intent annotation on the helloThere method of the HelloChildService is @Integrity
 2327 and @Authentication("transport"), the same as in HelloService class.
 2328 The effective intent annotation on the hello method of the HelloService is @Integrity and
 2329 @Authentication("message")
 2330
 2331 The listing below contains the equivalent declarative security interaction policy of the HelloService
 2332 and HelloChildService implementation corresponding to the Java interfaces and classes shown in
 2333 Example 2a.

```

2334
2335 <?xml version="1.0" encoding="ASCII"?>
2336
2337 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2338           name="HelloServiceComposite" >
2339   <service name="HelloService" requires="integrity/transport
2340           authentication">
2341     ...
2342   </service>
2343   <service name="HelloChildService" requires="integrity/transport
2344           authentication confidentiality/message">
2345     ...
2346   </service>
2347   ...
2348
2349   <component name="HelloServiceComponent">*
2350     <implementation.java class="services.hello.HelloService"/>
2351     <operation name="hello" requires="integrity
2352           authentication/message"/>
2353     <operation name="helloThere"
2354 requires="integrity
2355           authentication/transport"/>
2356   </component>
2357   <component name="HelloChildServiceComponent">*
2358     <implementation.java
2359 class="services.hello.HelloChildService" />
2360     <operation name="hello"
2361 requires="confidentiality/transport"/>
2362     <operation name="helloThere" requires=" integrity/transport
2363           authentication"/>
2364     <operation name="helloWorld" requires="authentication"/>
2365   </component>
2366   ...
2367   ...
2368
2369 </composite>
  
```

2370 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.
 2371
 2372

2373 10.4 Relationship of Declarative And Annotated Intents

2374 Annotated intents on a Java class cannot be overridden by declarative intents either in a
 2375 composite document which uses the class as an implementation or by statements in a component

2376 Type document associated with the class. This rule follows the general rule for intents that they
2377 represent fundamental requirements of an implementation.

2378 An unqualified version of an intent expressed through an annotation in the Java class may be
2379 qualified by a declarative intent in a using composite document.

2380

2381 10.5 Policy Set Annotations

2382 The SCA Policy Framework uses Policy Sets to capture detailed low-level concrete policies (for
2383 example, a concrete policy is the specific encryption algorithm to use when encrypting messages
2384 when using a specific communication protocol to link a reference to a service).

2385 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
2386 The @PolicySets annotation either takes the QName of a single policy set as a string or the name
2387 of two or more policy sets as an array of strings:
2388

```
2389     @PolicySets( "<policy set QName>" |  
2390                 { "<policy set QName>" [, "<policy set QName>" ] })
```

2391

2392 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

2393 An example of the @PolicySets annotation:

2394

```
2395     @Reference(name="helloService", required=true)  
2396     @PolicySets({ MY_NS + "WS_Encryption_Policy",  
2397                 MY_NS + "WS_Authentication_Policy" })  
2398     public setHelloService(HelloService service) {  
2399         . . .  
2400     }
```

2401 In this case, the Policy Sets WS_Encryption_Policy and WS_Authentication_Policy are applied, both
2402 using the namespace defined for the constant MY_NS.

2403 PolicySets must satisfy intents expressed for the implementation when both are present, according
2404 to the rules defined in [the Policy Framework specification \[POLICY\]](#).

2405 The SCA Policy Set annotation can be applied to the following Java elements:

- 2406 • Java class
- 2407 • Java interface
- 2408 • Method
- 2409 • Field

2410

2411 10.6 Security Policy Annotations

2412 This section introduces annotations for SCA's security intents, as defined in [the SCA Policy](#)
2413 [Framework specification \[POLICY\]](#).

2414

2415 10.6.1 Security Interaction Policy

2416 The following interaction policy Intents and qualifiers are defined for Security Policy, which apply
2417 to the operation of services and references of an implementation:

- 2418 • @Integrity
- 2419 • @Confidentiality
- 2420 • @Authentication

2421 All three of these intents have the same pair of Qualifiers:

- 2422 • message
- 2423 • transport

2424 The following snippets shows the ~~@Integrity~~, ~~@Confidentiality~~ and ~~@Authentication~~ annotations:

```

2425 package org.oasisopen.sca.annotations;
2426
2427 import java.lang.annotation.*;
2428 import static org.oasisopen.sca.Constants.SCA_NS;
2429
2430 @Inherited
2431 @Retention(RetentionPolicy.RUNTIME)
2432 @Target({ElementType.TYPE, ElementType.METHOD,
2433         ElementType.FIELD, ElementType.PARAMETER})
2434 @Intent(Integrity.INTEGRITY)
2435 public @interface Integrity {
2436     String INTEGRITY = SCA_NS+"integrity";
2437     String INTEGRITY_MESSAGE = INTEGRITY+".message";
2438     String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
2439     @Qualifier
2440     String[] value() default "";
2441 }
2442
2443
2444 package org.oasisopen.sca.annotations;
2445
2446 import java.lang.annotation.*;
2447 import static org.oasisopen.sca.Constants.SCA_NS;
2448
2449 @Inherited
2450 @Retention(RetentionPolicy.RUNTIME)
2451 @Target({ElementType.TYPE, ElementType.METHOD,
2452         ElementType.FIELD, ElementType.PARAMETER})
2453 @Intent(Confidentiality.CONFIDENTIALITY)
2454 public @interface Confidentiality {
2455     String CONFIDENTIALITY = SCA_NS+"confidentiality";
2456     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY+".message";

```



```

2457     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY+".transport";
2458     @Qualifier
2459     String[] value() default "";
2460     }
2461
2462
2463 package org.oasisopen.sca.annotations;
2464
2465 import java.lang.annotation.*;
2466 import static org.oasisopen.sca.Constants.SCA_NS;
2467
2468 @Inherited
2469 @Retention(RetentionPolicy.RUNTIME)
2470 @Target({ElementType.TYPE, ElementType.METHOD,
2471           ElementType.FIELD, ElementType.PARAMETER})
2472 @Intent(Authentication.AUTHENTICATION)
2473 public @interface Authentication {
2474     String AUTHENTICATION = SCA_NS+"authentication";
2475     String AUTHENTICATION_MESSAGE = AUTHENTICATION+".message";
2476     String AUTHENTICATION_TRANSPORT = AUTHENTICATION+".transport";
2477     @Qualifier
2478     String[] value() default "";
2479     }
2480
2481

```

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

```

2485
2486 //Interface for HelloService
2487 public interface service.hello.HelloService {
2488     String hello(String helloMsg);
2489 }
2490
2491 // Interface for ClientService
2492 public interface service.client.ClientService {
2493     public void clientMethod();
2494 }
2495
2496 // Implementation class for ClientService
2497 package services.client;

```

```

2498
2499     import services.hello.HelloService;
2500
2501     import org.oasisopen.sca.annotations.*;
2502
2503     @Service(ClientService.class)
2504     public class ClientServiceImpl implements ClientService {
2505
2506
2507         private HelloService helloService;
2508
2509         @Reference(name="helloService", required=true)
2510         @Integrity("message")
2511         @Authentication("message")
2512         public void setHelloService(HelloService service) {
2513             helloService = service;
2514         }
2515
2516         public void clientMethod() {
2517             String result = helloService.hello("Hello World!");
2518             ...
2519         }
2520     }

```

Example 1. Usage of annotated intents on a reference.

10.6.2 Security Implementation Policy

SCA defines java implementation honors the set 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. The @runAs annotations can be mapped to <runAs> element that is defined in the policy specification. Any code so annotated will run with the permissions of that role. How runAs role names are mapped to security principals is implementation dependent.

E.g. the above @RunAs annotation can be mapped as if the following policySet is defined and attached to the level where the annotation applies:

```

<policySet name="runas_manager">
  <securityIdentity>
    <runAs role="Manager">
  </securityIdentity>

```

2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595

</policySet

•

• RolesAllowed

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

The @RolesAllowed annotation can be mapped to <allow> element that is defined in the policy specification. It indicates that access is granted only to principals whose role corresponds to one of the role names listed in the @roles attribute. How role names are mapped to security principals is SCA Runtime implementation dependent. E.g. the above @RolesAllowed annotation can be mapped as if the following policySet is defined and attached to the level where the annotation applies:

```
<policySet name="allow_manager_employee">  
  <authorization>  
    <allow roles="Manager Employee">  
  </authorization>  
</policySet>
```

•

• PermitAll

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

- The @PermitAll annotation can be mapped to <permitAll> element that is defined in the policy specification. It indicates to grant access to all principals respectively. E.g. the above @PermitAll annotation can be mapped as if the following policySet is defined and attached to the level where the annotation applies: <policySet name="permitAll">

```
• <authorization>
```

```
• <permitAll/>
```

```
• </authorization>
```

```
• </policySet>
```

•

• DenyAll

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

- The @DenyAll annotation can be mapped to <denyAll> element that is defined in the policy specification. It indicates to deny access to all principals respectively.
- E.g. the above @DenyAll annotation can be mapped as if the following policySet is defined and attached to the level where the annotation applies:

```
• <policySet name="denyAll">
```

```
• <authorization>
```

```
• <denyAll/>
```

```
• </authorization>
```

```
• </policySet>
```

•

• DeclareRoles

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

There is no mapping to elements defined in policy specifications.

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

Formatted: Bullets and Numbering

Formatted: SC.6.208911

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

Formatted: Indent: Left: 0.5", No bullets or numbering

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

Formatted: Bullets and Numbering

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

Formatted: Bullets and Numbering

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

Formatted: Bullets and Numbering

2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639

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

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

Formatted: Indent: Left: 0.25"

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

10.6.2.1 Annotated Implementation Policy Example

The following is an example showing annotated security implementation policy:

```
package services.account;
@Remotable
public interface AccountService {
    AccountReport getAccountReport (String customerID);
}
```

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

```
package services.account;
import java.util.List;
import commonj.sdo.DataFactory;
import org.oasisopen.sca.annotations.Property;
import org.oasisopen.sca.annotations.Reference;
import org.oasisopen.sca.annotations.javax.annotation.security.RolesAllowed;
import org.oasisopen.sca.annotations.javax.annotation.security.RunAs;
import org.oasisopen.sca.annotations.javax.annotation.security.PermitAll;
import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;
@RolesAllowed("customers")
@RunAs("accountants" )
public class AccountServiceImpl implements AccountService {

    @Property
    protected String currency = "USD";

    @Reference
    protected AccountDataService accountDataService;

    @Reference
```

```

2640     protected StockQuoteService stockQuoteService;
2641
2642     @RolesAllowed({"customers", "accountants"})
2643     public AccountReport getAccountReport(String customerID) {
2644
2645         DataFactory dataFactory = DataFactory.INSTANCE;
2646         AccountReport accountReport =
2647             (AccountReport)dataFactory.create(AccountReport.class);
2648         List accountSummaries = accountReport.getAccountSummaries();
2649
2650         CheckingAccount checkingAccount =
2651             accountDataService.getCheckingAccount(customerID);
2652         AccountSummary checkingAccountSummary =
2653             (AccountSummary)dataFactory.create(AccountSummary.class);
2654         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
2655 );
2656
2657         checkingAccountSummary.setAccountType("checking");
2658         checkingAccountSummary.setBalance(fromUSDollarToCurrency
2659             (checkingAccount.getBalance()));
2660         accountSummaries.add(checkingAccountSummary);
2661
2662         SavingsAccount savingsAccount =
2663             accountDataService.getSavingsAccount(customerID);
2664         AccountSummary savingsAccountSummary =
2665             (AccountSummary)dataFactory.create(AccountSummary.class);
2666
2667         savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
2668         savingsAccountSummary.setAccountType("savings");
2669         savingsAccountSummary.setBalance(fromUSDollarToCurrency
2670             (savingsAccount.getBalance()));
2671         accountSummaries.add(savingsAccountSummary);
2672
2673         StockAccount stockAccount =
2674             accountDataService.getStockAccount(customerID);
2675         AccountSummary stockAccountSummary =
2676             (AccountSummary)dataFactory.create(AccountSummary.class);
2677         stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
2678         stockAccountSummary.setAccountType("stock");
2679         float balance= (stockQuoteService.getQuote(stockAccount.getSymbol())) *
2680             stockAccount.getQuantity();
2681         stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
2682         accountSummaries.add(stockAccountSummary);
2683

```

```
2684     return accountReport;
2685 }
2686
2687 @PermitAll
2688 public float fromUSDollarToCurrency(float value) {
2689
2690     if (currency.equals("USD")) return value; else
2691     if (currency.equals("EURO")) return value * 0.8f; else
2692     return 0.0f;
2693 }
2694 }
```

2695 Example 3. Usage of annotated security implementation policy for the java language.

2696 In this example, the implementation class as a whole is marked:

- 2697 • @RolesAllowed("customers") - indicating that customers have access to the
- 2698 implementation as a whole
- 2699 • @RunAs("accountants") - indicating that the code in the implementation runs with the
- 2700 permissions of accountants

2701 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),
2702 which indicates that this method can be called by both customers and accountants.

2703 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method
2704 can be called by any role.

2705

2706

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

```
2707 <?xml version="1.0" encoding="UTF-8"?>
2708 <!-- (c) Copyright SCA Collaboration 2006 -->
2709 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2710         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2711         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2712         elementFormDefault="qualified">
2713
2714     <include schemaLocation="sca-core.xsd"/>
2715
2716     <element name="interface.java" type="sca:JavaInterface"
2717             substitutionGroup="sca:interface"/>
2718     <complexType name="JavaInterface">
2719         <complexContent>
2720             <extension base="sca:Interface">
2721                 <sequence>
2722                     <any namespace="##other" processContents="lax"
2723 minOccurs="0" maxOccurs="unbounded"/>
2724                 </sequence>
2725                 <attribute name="interface" type="NCName" use="required"/>
2726                 <attribute name="callbackInterface" type="NCName"
2727 use="optional"/>
2728                 <anyAttribute namespace="##any" processContents="lax"/>
2729             </extension>
2730         </complexContent>
2731     </complexType>
2732 </schema>
2733
```

2734

B. Conformance Items

2735 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
2736 specification.

2737

Conformance ID	Description
[JCA30001]	@interface MUST be the fully qualified name of the Java interface class
[JCA30002]	@callbackInterface MUST be the fully qualified name of a Java interface used for callbacks
[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.

2738

2739

C. Acknowledgements

2740

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

2741

acknowledged:

2742

Participants:

2743

[Participant Name, Affiliation | Individual Member]

2744

[Participant Name, Affiliation | Individual Member]

2745

D. Non-Normative Text

2747

E. Revision History

2748 [optional; should not be included in OASIS Standards]

2749

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combella	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	* 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
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup. All changes accepted.

			All comments removed.
--	--	--	-----------------------

2750
2751

