



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02, Revision 03+Issue1 rev 5

22 June 2009

Deleted: 2

Deleted: 19

Deleted: 6

Deleted: March

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev3.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 can 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 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 can chose 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 might 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	10
2.2.2	Composite scope	11
3	Interface	12
3.1	Java interface element – <interface.java>	12
3.2	@Remotable	13
3.3	@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	SCAClient Interface and Related Classes	14
5	Error Handling	15
6	Asynchronous Programming	16
6.1	@OneWay	16
6.2	Callbacks	16
6.2.1	Using Callbacks	16
6.2.2	Callback Instance Management	18
6.2.3	Implementing Multiple Bidirectional Interfaces	18
6.2.4	Accessing Callbacks	19
7	Policy Annotations for Java	20
7.1	General Intent Annotations	20
7.2	Specific Intent Annotations	22
7.2.1	How to Create Specific Intent Annotations	22
7.3	Application of Intent Annotations	23
7.3.1	Inheritance And Annotation	23
7.4	Relationship of Declarative And Annotated Intents	25
7.5	Policy Set Annotations	25
7.6	Security Policy Annotations	26
7.6.1	Security Interaction Policy	26
7.6.2	Security Implementation Policy	27
8	Java API	30

8.1 Component Context.....	30	
8.2 Request Context.....	31	
8.3 ServiceReference.....	32	
8.4 ServiceRuntimeException.....	32	
8.5 ServiceUnavailableException.....	33	
8.6 InvalidServiceException.....	33	
8.7 Constants.....	33	
8.8 SCAClient Interface.....	33	
8.9 SCAClientFactory Class.....	34	
8.10 NoSuchDomainException.....	36	Deleted: 36
8.11 NoSuchServiceException.....	40	Deleted: 37
9 Java Annotations.....	41	Deleted: 37
9.1 @AllowsPassByReference.....	41	Deleted: 37
9.2 @Authentication.....	41	Deleted: 37
9.3 @Callback.....	42	Deleted: 38
9.4 @ComponentName.....	43	Deleted: 39
9.5 @Confidentiality.....	44	Deleted: 40
9.6 @Constructor.....	45	Deleted: 41
9.7 @Context.....	45	Deleted: 41
9.8 @Destroy.....	46	Deleted: 42
9.9 @EagerInit.....	46	Deleted: 42
9.10 @Init.....	47	Deleted: 43
9.11 @Integrity.....	47	Deleted: 43
9.12 @Intent.....	48	Deleted: 44
9.13 @OneWay.....	49	Deleted: 45
9.14 @PolicySets.....	49	Deleted: 45
9.15 @Property.....	50	Deleted: 46
9.16 @Qualifier.....	51	Deleted: 47
9.17 @Reference.....	52	Deleted: 48
9.17.1 ReInjection.....	54	Deleted: 50
9.18 @Remotable.....	56	Deleted: 52
9.19 @Requires.....	57	Deleted: 53
9.20 @Scope.....	58	Deleted: 54
9.21 @Service.....	59	Deleted: 55
10 WSDL to Java and Java to WSDL.....	61	Deleted: 57
10.1 JAX-WS Client Asynchronous API for a Synchronous Service.....	61	Deleted: 57
A. XML Schema: sca-interface-java.xsd.....	63	Deleted: 59
B. Java Classes and Interfaces.....	64	Deleted: 60
B.1 SCAClient Classes and Interfaces.....	64	Deleted: 60
B.1.1 SCAClient Interface.....	64	Deleted: 60
B.1.2 SCAClientFactory Class.....	65	Deleted: 60
B.1.3 SCAFactoryFinder class.....	66	Deleted: 61
B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?.....	72	Deleted: 62
C. Conformance Items.....	73	Deleted: 67
D. Acknowledgements.....	79	Deleted: 69
		Deleted: 75

E. Non-Normative Text 80

F. Revision History..... 81

Deleted: 76

Deleted: 77

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 can 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 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 can choose 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 and 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].

Comment [ME1]: This sentence needs to be removed

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/
[POLICY]	SCA Policy Framework, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf

- 44 **[JSR-250]** Common Annotation for Java Platform specification (JSR-250),
45 <http://www.jcp.org/en/jsr/detail?id=250>
46 **[JAX-WS]** JAX-WS 2.1 Specification (JSR-224),
47 <http://www.jcp.org/en/jsr/detail?id=224>
48 **[JAVABEANS]** JavaBeans 1.01 Specification,
49 <http://java.sun.com/javase/technologies/desktop/javabeans/api/>
50

51 **1.3 Non-Normative References**

- 52 **[EBNF-Syntax]** Extended BNF syntax format used for formal grammar of constructs
53 <http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation>

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

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

66 2.1.2 Java Semantics of a Remotable Service

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

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

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

Deleted: Remotable Services MUST NOT make use of **method overloading**.

Deleted: Remotable Services MUST NOT make use of **method overloading**.

77 2.1.3 Java Semantics of a Local Service

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

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

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

```
83 package services.hello;  
84 public interface HelloService {  
85     String hello(String message);  
86 }  
87
```

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

90 The data exchange semantic for calls to local services is **by-reference**. This means that
91 implementation code which uses a local interface needs to be written with the knowledge that
92 changes made to parameters (other than simple types) by either the client or the provider of the
93 service are visible to the other.

94 **2.1.4 @Reference**

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

98 **2.1.5 @Property**

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

102 **2.2 Implementation Scopes: @Scope, @Init, @Destroy**

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

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

109 This document defines two scopes:

- 110 • STATELESS
- 111 • COMPOSITE

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

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

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

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

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

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

```
124 @Init  
125 public void start() {  
126     ...  
127 }  
128  
129 @Destroy  
130 public void stop() {  
131     ...  
132 }  
133  
134
```

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

137 **2.2.1 Stateless scope**

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

140 The concurrency model for the stateless scope is single threaded. This means that the SCA
141 runtime MUST ensure that a stateless scoped implementation instance object is only ever
142 dispatched on one thread at any one time. [JCA20002] In addition, within the SCA lifecycle of a
143 stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of
144 one business method. [JCA20003] Note that the SCA lifecycle might not correspond to the Java
145 object lifecycle due to runtime techniques such as pooling.

146 2.2.2 Composite scope

147 For a composite scope implementation instance, the SCA runtime MUST ensure that all service
148 requests are dispatched to the same implementation instance for the lifetime of the containing
149 composite. [JCA20004] The lifetime of the containing composite is defined as the time it becomes
150 active in the runtime to the time it is deactivated, either normally or abnormally.

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

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

Deleted: When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.

Deleted: When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.

Deleted: If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.

Deleted: If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.

158 3 Interface

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

160 3.1 Java interface element – <interface.java>

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

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

168 The following is the pseudo-schema for the `interface.java` element

169

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

171

172 The `interface.java` element has the following attributes:

- 173 • **interface : NCName (1..1)** – the Java interface class to use for the service interface. The
174 value of the `@interface` attribute MUST be the fully qualified name of the Java interface
175 class. [JCA30001]
- 176 • **callbackInterface : NCName (0..1)** – the Java interface class to use for the callback
177 interface. The value of the `@callbackInterface` attribute MUST be the fully qualified name
178 of a Java interface used for callbacks. [JCA30002]

179

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

181

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

184

185 Here, the Java interface is defined in the Java class file

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

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

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

199

200

Deleted: The value of the
@interface attribute MUST
be the fully qualified name
of the Java interface class

Deleted: The value of the
@interface attribute MUST
be the fully qualified name
of the Java interface class

Deleted: The value of the
@callbackInterface
attribute MUST be the fully
qualified name of a Java
interface used for callbacks

Deleted: The value of the
@callbackInterface
attribute MUST be the fully
qualified name of a Java
interface used for callbacks

201 3.2 @Remotable

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

206 3.3 @Callback

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

211 4 Client API

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

214 4.1 Accessing Services from an SCA Component

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

219 4.1.1 Using the Component Context API

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

223 4.2 Accessing Services from non-SCA component implementations

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

226 4.2.1 **SCAClient** Interface and Related Classes

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

230 Objects which implement the SCAClient interface are obtained using the SCAClientFactory class.

232 The following is a sample of the code that a client would use:

```
233 import org.oasisopen.sca.client.SCAClient;  
234 import org.oasisopen.sca.client.SCAClientFactory;  
235 import com.foo.HelloService;  
236  
237 public void someMethod() {  
238  
239     String serviceURI = "SomeHelloServiceURI";  
240     URI domainURI = new URI("SomeDomainURI");  
241  
242     ...  
243     SCAClient scaClient = SCAClientFactory.newInstance();  
244     HelloService helloService =  
245         scaClient.getService(HelloService.class,  
246                              serviceURI, domainURI);  
247     String reply = helloService.sayHello("Mark");  
248     ...  
249 }
```

250 For details about the SCAClient interface and its related classes see the section "SCAClient
251 Interface" and the section "SCAClientFactory Class".

252

Deleted: ComponentContext

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

Deleted: d

Deleted: ¶

Deleted: ComponentContext context = // obtained via host environment-specific means ¶ HelloService helloService = ¶ context.getService(HelloService.class, "HelloService"); ¶ String result = helloService.hello("Hello World!"); ¶

Formatted: Bullets and Numbering

253 5 Error Handling

254 Clients calling service methods can experience business exceptions and SCA runtime exceptions.

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

257 SCA runtime exceptions are raised by the SCA runtime and signal problems in management of
258 component execution or problems interacting with remote services. The SCA runtime exceptions
259 are [defined in the Java API section](#).

260 6 Asynchronous Programming

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

- 268 • support for non-blocking method calls
- 269 • callbacks

270 Each of these topics is discussed in the following sections.

271 6.1 @OneWay

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

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

278 For a Java client to make a non-blocking call to methods that either return values or which throw
279 exceptions, a Java client can use the JAX-WS asynchronous client API model that is described in
280 section 9. It is considered to be a best practice that service designers define one-way methods as
281 often as possible, in order to give the greatest degree of binding flexibility to deployers.

282 6.2 Callbacks

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

- 287 • an interface for the provided service
- 288 • a callback interface that is provided by the client

289 Callbacks can be used for both remotable and local services. Either both interfaces of a
290 bidirectional service are remotable, or both are local. It is illegal to mix the two, as defined in the
291 SCA Assembly specification [SCA Assembly].

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

296 6.2.1 Using Callbacks

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

302 The following example shows a scenario in which bidirectional interfaces and callbacks could be
303 used. A client requests a quotation from a supplier. To process the enquiry and return the

304 quotation, some suppliers might need additional information from the client. The client does not
305 know which additional items of information will be needed by different suppliers. This interaction
306 can be modeled as a bidirectional interface with callback requests to obtain the additional
307 information.

```
308 package somepackage;
309 import org.osoa.sca.annotation.Callback;
310 import org.osoa.sca.annotation.Remotable;
311 @Remotable
312 @Callback(QuotationCallback.class)
313 public interface Quotation {h
314     double requestQuotation(String productCode, int quantity);
315 }
316
317 @Remotable
318 public interface QuotationCallback {
319     String getState();
320     String getZipCode();
321     String getCreditRating();
322 }
323
```

324 In this example, the `requestQuotation` operation requests a quotation to supply a given quantity
325 of a specified product. The `QuotationCallback` interface provides a number of operations that the
326 supplier can use to obtain additional information about the client making the request. For
327 example, some suppliers might quote different prices based on the state or the zip code to which
328 the order will be shipped, and some suppliers might quote a lower price if the ordering company
329 has a good credit rating. Other suppliers might quote a standard price without requesting any
330 additional information from the client.

331 The following code snippet illustrates a possible implementation of the example service, using the
332 `@Callback` annotation to request that a callback proxy be injected.

```
333 @Callback
334 protected QuotationCallback callback;
335
336 public double requestQuotation(String productCode, int quantity) {
337     double price = getPrice(productCode, quantity);
338     double discount = 0;
339     if (quantity > 1000 && callback.getState().equals("FL")) {
340         discount = 0.05;
341     }
342     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
343         discount += 0.05;
344     }
345     return price * (1-discount);
346 }
347 }
348
```

349 The code snippet below is taken from the client of this example service. The client's service
350 implementation class implements the methods of the `QuotationCallback` interface as well as those
351 of its own service interface `ClientService`.

```
352
353 public class ClientImpl implements ClientService, QuotationCallback {
354
355     private QuotationService myService;
356
357     @Reference
358     public void setMyService(QuotationService service) {
359         myService = service;
360     }
361 }

```

```

360     }
361
362     public void aClientMethod() {
363         ...
364         double quote = myService.requestQuotation("AB123", 2000);
365         ...
366     }
367
368     public String getState() {
369         return "TX";
370     }
371     public String getZipCode() {
372         return "78746";
373     }
374     public String getCreditRating() {
375         return "AA";
376     }
377 }

```

378
379 In this example the callback is **stateless**, i.e., the callback requests do not need any information
380 relating to the original service request. For a callback that needs information relating to the
381 original service request (a **stateful** callback), this information can be passed to the client by the
382 service provider as parameters on the callback request.

383 6.2.2 Callback Instance Management

384 Instance management for callback requests received by the client of the bidirectional service is
385 handled in the same way as instance management for regular service requests. If the client
386 implementation has STATELESS scope, the callback is dispatched using a newly initialized
387 instance. If the client implementation has COMPOSITE scope, the callback is dispatched using the
388 same shared instance that is used to dispatch regular service requests.

389 As described in section 6.7.1, a stateful callback can obtain information relating to the original
390 service request from parameters on the callback request. Alternatively, a composite-scoped client
391 could store information relating to the original request as instance data and retrieve it when the
392 callback request is received. These approaches could be combined by using a key passed on the
393 callback request (e.g., an order ID) to retrieve information that was stored in a composite-scoped
394 instance by the client code that made the original request.

395 6.2.3 Implementing Multiple Bidirectional Interfaces

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

```

402 @Callback
403 protected MyService1Callback callback1;
404
405 @Callback
406 protected MyService2Callback callback2;

```

408
409 If a single callback has a type that is compatible with multiple declared callback fields, then all of
410 them will be set.

411 6.2.4 Accessing Callbacks

412 In addition to injecting a reference to a callback service, it is also possible to obtain a reference to
413 a Callback instance by annotating a field or method of type **ServiceReference** with the
414 **@Callback** annotation.

415
416 A reference implementing the callback service interface can be obtained using
417 `ServiceReference.getService()`.

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

```
419 @Callback
420 protected ServiceReference<MyCallback> callback;
421
422 public void someMethod() {
423     MyCallback myCallback = callback.getCallback();    ...
424
425     myCallback.receiveResult(theResult);
426 }
427
428
429
```

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

434 Alternatively, a callback can be retrieved programmatically using the **RequestContext** API. The
435 snippet below shows how to retrieve a callback in a method programmatically:

```
436 public void someMethod() {
437     MyCallback myCallback =
438         ComponentContext.getRequestContext().getCallback();
439     ...
440
441     myCallback.receiveResult(theResult);
442 }
443
444
445
```

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

```
448 @Context
449 protected RequestContext requestContext;
450
451 void receiveResult(Object theResult) {
452     Object refParams =
453         requestContext.getServiceReference().getCallbackID();
454     ...
455 }
456
```

457
458 This is necessary if the service implementation has COMPOSITE scope, because callback injection
459 is not performed for composite-scoped implementations.

460 7 Policy Annotations for Java

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

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

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

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

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

487 7.1 General Intent Annotations

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

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

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

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

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

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

```
500     public static final String SCA_PREFIX=  
501         "{http://docs.oasis-open.org/ns/opencsa/sca/200712}";  
502     public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
503     public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
504
```

505 Notice that, by convention, qualified intents include the qualifier as part of the name of the
506 constant, separated by an underscore. These intent constants are defined in the file that defines

507 an annotation for the intent (annotations for intents, and the formal definition of these constants,
508 are covered in a following section).

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

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

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

513

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

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

```
516     package com.foo;
517
518     import static org.oasisopen.sca.annotation.Confidentiality.*;
519     import static org.oasisopen.sca.annotation.Reference;
520     import static org.oasisopen.sca.annotation.Requires;
521
522     public class Foo {
523         @Requires(CONFIDENTIALITY)
524         @Reference
525         public void setBar(Bar bar) {
526             ...
527         }
528     }
529
```

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

```
533     package com.foo;
534
535     import static org.oasisopen.sca.Constants.*;
536     import static org.oasisopen.sca.annotation.Reference;
537     import static org.oasisopen.sca.annotation.Requires;
538
539     public class Foo {
540         @Requires(SCA_PREFIX+"confidentiality")
541         @Reference
542         public void setBar(Bar bar) {
543             ...
544         }
545     }
546
```

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

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

549 where

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

552

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

554 7.2 Specific Intent Annotations

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

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

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

```
566 @Integrity
```

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

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

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

572 The general form of specific intent annotations is:

```
573 '@' Intent ('(' qualifiers ')')?
```

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

```
575 Intent      ::= NCName  
576 qualifiers  ::= "" qualifier "" (',' qualifier "")*  
577 qualifier   ::= NCName ('.' qualifier)?  
578
```

579 7.2.1 How to Create Specific Intent Annotations

580 **SCA identifies annotations that correspond to intents by providing an @Intent annotation which**
581 **MUST be used in the definition of a specific intent annotation.** [JCA70001]

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

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

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

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

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

594 In this case, the attribute's definition needs to be marked with the @Qualifier annotation. The
595 @Qualifier tells SCA that the value of the attribute is treated as a qualifier for the intent
596 represented by the whole annotation. If more than one qualifier value is specified in an
597 annotation, it means that multiple qualified forms exist. For example:

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

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

Deleted: SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.

Deleted: SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.

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

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

604 7.3 Application of Intent Annotations

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

- 606 • Java class
- 607 • Java interface
- 608 • Method
- 609 • Field
- 610 • Constructor parameter

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

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

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

617 If an annotation is specified at both the class/interface level and the method or field level, then
618 the method or field level annotation completely overrides the class level annotation of the same
619 base intent name.

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

623 7.3.1 Inheritance And Annotation

624 The inheritance rules for annotations are consistent with the common annotation specification, JSR
625 250 [JSR-250]

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

```
628 package services.hello;  
629 import org.oasisopen.sca.annotation.Remotable;  
630 import org.oasisopen.sca.annotation.Integrity;  
631 import org.oasisopen.sca.annotation.Authentication;  
632  
633 @Integrity("transport")  
634 @Authentication  
635 public class HelloService {  
636     @Integrity  
637     @Authentication("message")  
638     public String hello(String message) {...}  
639  
640     @Integrity  
641     @Authentication("transport")  
642     public String helloThere() {...}  
643 }  
644  
645 package services.hello;  
646 import org.oasisopen.sca.annotation.Remotable;  
647 import org.oasisopen.sca.annotation.Confidentiality;  
648 import org.oasisopen.sca.annotation.Authentication;
```

```

649
650     @Confidentiality("message")
651     public class HelloChildService extends HelloService {
652         @Confidentiality("transport")
653         public String hello(String message) {...}
654         @Authentication
655         String helloWorld() {...}
656     }

```

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

658

659 The effective intent annotation on the **helloWorld** method of the **HelloChildService** is
660 Integrity("transport"), @Authentication, and @Confidentiality("message").

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

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

665 The effective intent annotation on the **hello** method of the **HelloService** is @Integrity and
666 @Authentication("message")

667

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

671

```

672 <?xml version="1.0" encoding="ASCII"?>
673 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
674           name="HelloServiceComposite" >
675     <service name="HelloService" requires="integrity/transport
676           authentication">
677       ...
678     </service>
679     <service name="HelloChildService" requires="integrity/transport
680           authentication confidentiality/message">
681       ...
682     </service>
683     ...
684
685     <component name="HelloServiceComponent">*
686       <implementation.java class="services.hello.HelloService"/>
687       <operation name="hello" requires="integrity
688             authentication/message"/>
689       <operation name="helloThere"
690             requires="integrity
691             authentication/transport"/>
692     </component>
693     <component name="HelloChildServiceComponent">*
694       <implementation.java
695             class="services.hello.HelloChildService" />
696       <operation name="hello"
697             requires="confidentiality/transport"/>
698       <operation name="helloThere" requires=" integrity/transport
699             authentication"/>
700       <operation name="helloWorld" requires="authentication"/>
701     </component>
702

```


703 ...
704
705 </composite>
706

707 Example 2b. Declaratives intents equivalent to annotated intents in Example 2a.
708

709 7.4 Relationship of Declarative And Annotated Intents

710 Annotated intents on a Java class cannot be overridden by declarative intents in a composite
711 document which uses the class as an implementation. This rule follows the general rule for intents
712 that they represent requirements of an implementation in the form of a restriction that cannot be
713 relaxed.

714 However, a restriction can be made more restrictive so that an unqualified version of an intent
715 expressed through an annotation in the Java class can be qualified by a declarative intent in a
716 using composite document.

717 7.5 Policy Set Annotations

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

721 Policy Sets can be applied directly to Java implementations using the **@PolicySets** annotation.
722 The @PolicySets annotation either takes the QName of a single policy set as a string or the name
723 of two or more policy sets as an array of strings:
724

```
725                   @PolicySets( "<policy set QName>" (, "<policy set QName>")* )
```

726
727 As for intents, PolicySet names are QNames – in the form of "{Namespace-URI}localPart".

728 An example of the @PolicySets annotation:

```
729  
730                   @Reference(name="helloService", required=true)  
731                   @PolicySets({ MY_NS + "WS_Encryption_Policy",  
732                                MY_NS + "WS_Authentication_Policy" })  
733                   public setHelloService(HelloService service) {  
734                    ...  
735                   }  
736
```

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

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

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

- 742 • Java class
- 743 • Java interface
- 744 • Method
- 745 • Field
- 746 • Constructor parameter

747 7.6 Security Policy Annotations

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

750 7.6.1 Security Interaction Policy

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

- 753 • @Integrity
- 754 • @Confidentiality
- 755 • @Authentication

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

- 757 • message
- 758 • transport

759 The formal definitions of the @Authentication, @Confidentiality and @Integrity annotations are
760 found in the sections [@Authentication](#), [@Confidentiality](#) and [@Integrity](#).

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

```
764  
765 package services.hello;  
766 //Interface for HelloService  
767 public interface HelloService {  
768     String hello(String helloMsg);  
769 }  
770  
771 package services.client;  
772 // Interface for ClientService  
773 public interface ClientService {  
774     public void clientMethod();  
775 }  
776  
777 // Implementation class for ClientService  
778 package services.client;  
779  
780 import services.hello.HelloService;  
781 import org.oasisopen.sca.annotation.*;  
782  
783 @Service(ClientService.class)  
784 public class ClientServiceImpl implements ClientService {  
785  
786     private HelloService helloService;  
787  
788     @Reference(name="helloService", required=true)  
789     @Integrity("message")  
790     @Authentication("message")  
791     public void setHelloService(HelloService service) {  
792         helloService = service;  
793     }  
794  
795     public void clientMethod() {  
796         String result = helloService.hello("Hello World!");
```

```
797     ...
798     }
799 }
```

800
801 Example 1. Usage of annotated intents on a reference.

802 7.6.2 Security Implementation Policy

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

- 806 • RunAs
807 Takes as a parameter a string which is the name of a Security role.
808 eg. @RunAs("Manager") Code marked with this annotation executes with the Security
809 permissions of the identified role.
810
- 811 • RolesAllowed
812 Takes as a parameter a single string or an array of strings which represent one or more
813 role names. When present, the implementation can only be accessed by principals whose
814 role corresponds to one of the role names listed in the @roles attribute. How role names
815 are mapped to security principals is implementation dependent (SCA does not define this).
816 eg. @RolesAllowed({"Manager", "Employee"})
817
- 818 • PermitAll
819 No parameters. When present, grants access to all roles.
820
- 821 • DenyAll
822 No parameters. When present, denies access to all roles.
823
- 824 • DeclareRoles
825 Takes as a parameter a string or an array of strings which identify one or more role names
826 that form the set of roles used by the implementation.
827 eg. @DeclareRoles({"Manager", "Employee", "Customer"})

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

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

830 7.6.2.1 Annotated Implementation Policy Example

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

```
832
833 package services.account;
834 @Remotable
835 public interface AccountService {
836     AccountReport getAccountReport(String customerID);
837     float fromUSDollarToCurrency(float value);
838 }
```

839
840 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,
841 plus the service references it makes and the settable properties that it has, along with a set of
842 implementation policy annotations:

```
843
844 package services.account;
```

```

845     import java.util.List;
846     import commonj.sdo.DataFactory;
847     import org.oasisopen.sca.annotation.Property;
848     import org.oasisopen.sca.annotation.Reference;
849     import org.oasisopen.sca.annotation.RolesAllowed;
850     import org.oasisopen.sca.annotation.RunAs;
851     import org.oasisopen.sca.annotation.PermitAll;
852     import services.accountdata.AccountDataService;
853     import services.accountdata.CheckingAccount;
854     import services.accountdata.SavingsAccount;
855     import services.accountdata.StockAccount;
856     import services.stockquote.StockQuoteService;
857     @RolesAllowed("customers")
858     @RunAs("accountants" )
859     public class AccountServiceImpl implements AccountService {
860
861         @Property
862         protected String currency = "USD";
863
864         @Reference
865         protected AccountDataService accountDataService;
866         @Reference
867         protected StockQuoteService stockQuoteService;
868
869         @RolesAllowed({"customers", "accountants"})
870         public AccountReport getAccountReport(String customerID) {
871
872             DataFactory dataFactory = DataFactory.INSTANCE;
873             AccountReport accountReport =
874                 (AccountReport)dataFactory.create(AccountReport.class);
875             List accountSummaries = accountReport.getAccountSummaries();
876
877             CheckingAccount checkingAccount =
878                 accountDataService.getCheckingAccount(customerID);
879             AccountSummary checkingAccountSummary =
880                 (AccountSummary)dataFactory.create(AccountSummary.class);
881
882             checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
883 );
884             checkingAccountSummary.setAccountType("checking");
885             checkingAccountSummary.setBalance(fromUSDollarToCurrency
886                 (checkingAccount.getBalance()));
887             accountSummaries.add(checkingAccountSummary);
888
889             SavingsAccount savingsAccount =
890                 accountDataService.getSavingsAccount(customerID);
891             AccountSummary savingsAccountSummary =
892                 (AccountSummary)dataFactory.create(AccountSummary.class);
893
894             savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
895             savingsAccountSummary.setAccountType("savings");
896             savingsAccountSummary.setBalance(fromUSDollarToCurrency
897                 (savingsAccount.getBalance()));
898             accountSummaries.add(savingsAccountSummary);
899
900             StockAccount stockAccount =
901                 accountDataService.getStockAccount(customerID);
902             AccountSummary stockAccountSummary =

```

```

903         (AccountSummary)dataFactory.create(AccountSummary.class);
904     stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
905     stockAccountSummary.setAccountType("stock");
906     float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
907         stockAccount.getQuantity();
908     stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
909     accountSummaries.add(stockAccountSummary);
910
911     return accountReport;
912 }
913
914 @PermitAll
915 public float fromUSDollarToCurrency(float value) {
916
917     if (currency.equals("USD")) return value;
918     if (currency.equals("EURO")) return value * 0.8f;
919     return 0.0f;
920 }
921 }

```

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

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

- 924 • @RolesAllowed("customers") - indicating that customers have access to the
- 925 implementation as a whole
- 926 • @RunAs("accountants") - indicating that the code in the implementation runs with the
- 927 permissions of accountants

928 The getAccountReport(..) method is marked with @RolesAllowed({"customers", "accountants"}),

929 which indicates that this method can be called by both customers and accountants.

930 The fromUSDollarToCurrency() method is marked with @PermitAll, which means that this method

931 can be called by any role.

932 8 Java API

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

934 8.1 Component Context

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

```
936
937 package org.oasisopen.sca;
938
939 public interface ComponentContext {
940     String getURI();
941
942     <B> B getService(Class<B> businessInterface, String referenceName);
943
944     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
945                                             String referenceName);
946
947     <B> Collection<B> getServices(Class<B> businessInterface,
948                               String referenceName);
949
950     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
951                                                         businessInterface, String referenceName);
952
953     <B> ServiceReference<B> createSelfReference(Class<B>
954                                             businessInterface);
955
956     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
957                                             String serviceName);
958
959     <B> B getProperty(Class<B> type, String propertyName);
960
961     <B, R extends ServiceReference<B>> R cast(B target)
962         throws IllegalArgumentException;
963
964     RequestContext getRequestContext();
965
966
967 }
```

- 968
- 969 • **getURI()** - returns the absolute URI of the component within the SCA domain
 - 970 • **getService(Class businessInterface, String referenceName)** - Returns a proxy for
971 the reference defined by the current component. The getService() method takes as its
972 input arguments the Java type used to represent the target service on the client and the
973 name of the service reference. It returns an object providing access to the service. The
974 returned object implements the Java interface the service is typed with.
975 **ComponentContext.getService method MUST throw an IllegalArgumentException if the**
976 **reference identified by the referenceName parameter has multiplicity of 0..n or**
977 **1..n.[JCA80001]**
 - 978 • **getServiceReference(Class businessInterface, String referenceName)** - Returns a
979 ServiceReference defined by the current component. This method MUST throw an
980 IllegalArgumentException if the reference has multiplicity greater than one.

- 981 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
982 typed service proxies for a business interface type and a reference name.
- 983 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
984 list typed service references for a business interface type and a reference name.
- 985 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
986 be used to invoke this component over the designated service.
- 987 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
988 ServiceReference that can be used to invoke this component over the designated service.
989 Service name explicitly declares the service name to invoke
- 990 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
991 property defined by this component.
- 992 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
993 there is no current request or if the context is unavailable. The
994 **ComponentContext.getRequestContext** method MUST return non-null when invoked during
995 the execution of a Java business method for a service operation or a callback operation, on
996 the same thread that the SCA runtime provided, and MUST return null in all other cases.
997 [JCA80002]
- 998 • **cast(B target)** - Casts a type-safe reference to a ServiceReference

999 A component can access its component context by defining a field or setter method typed by
1000 **org.oasisopen.sca.ComponentContext** and annotated with **@Context**. To access the target
1001 service, the component uses **ComponentContext.getService(..)**.

1002 The following shows an example of component context usage in a Java class using the @Context
1003 annotation.

```
1004 private ComponentContext componentContext;
1005
1006 @Context
1007 public void setContext(ComponentContext context) {
1008     componentContext = context;
1009 }
1010
1011 public void doSomething() {
1012     HelloWorld service =
1013     componentContext.getService(HelloWorld.class, "HelloWorldComponent");
1014     service.hello("hello");
1015 }
```

1016 8.2 Request Context

1017 The following shows the **RequestContext** interface:

```
1018
1019 package org.oasisopen.sca;
1020
1021 import javax.security.auth.Subject;
1022
1023 public interface RequestContext {
1024
1025     Subject getSecuritySubject();
1026
1027     String getServiceName();
1028     <CB> ServiceReference<CB> getCallbackReference();
1029     <CB> CB getCallback();
1030     <B> ServiceReference<B> getServiceReference();

```

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

Formatted: Bullets and Numbering

1031 }
1032 }
1033 }

1034 The RequestContext interface has the following methods:

- 1035 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 1036 • **getServiceName()** – Returns the name of the service on the Java implementation the
1037 request came in on
- 1038 • **getCallbackReference()** – Returns a service reference to the callback as specified by the
1039 caller. This method returns null when called for a service request whose interface is not
1040 bidirectional or when called for a callback request.
- 1041 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
1042 getCallbackReference() method, this method returns null when called for a service request
1043 whose interface is not bidirectional or when called for a callback request.
- 1044 • **getServiceReference()** – **When invoked during the execution of a service operation, the
1045 getServiceReference method MUST return a ServiceReference that represents the service
1046 that was invoked. When invoked during the execution of a callback operation, the
1047 getServiceReference method MUST return a ServiceReference that represents the callback
1048 that was invoked. [JCA80003]**

Comment [ME2]: Need a reference to JAAS here

Comment [ME3]: What happens if there is no JAAS subject?

Deleted: When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.

Deleted: When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.

1049 8.3 ServiceReference

1050 ServiceReferences can be injected using the @Reference annotation on a field, a setter method,
1051 or constructor parameter taking the type ServiceReference. The detailed description of the usage
1052 of these methods is described in the section on Asynchronous Programming in this document.

1053 The following Java code defines the **ServiceReference** interface:

```
1054 package org.oasisopen.sca;  
1055  
1056 public interface ServiceReference<B> extends java.io.Serializable {  
1057     B getService();  
1058     Class<B> getBusinessInterface();  
1060 }  
1061
```

1062 The ServiceReference interface has the following methods:

- 1063 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
1064 returned is guaranteed to implement the business interface for this reference. The value
1065 returned is a proxy to the target that implements the business interface associated with this
1066 reference.
- 1067 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
1068 this reference.

1069 8.4 ServiceRuntimeException

1070 The following snippet shows the **ServiceRuntimeException**.

```
1071  
1072 package org.oasisopen.sca;  
1073  
1074 public class ServiceRuntimeException extends RuntimeException {  
1075     ...  
1076 }  
1077
```

1078 This exception signals problems in the management of SCA component execution.

1079 8.5 ServiceUnavailableException

1080 The following snippet shows the *ServiceUnavailableException*.

```
1081 package org.oasisopen.sca;
1082
1083
1084 public class ServiceUnavailableException extends ServiceRuntimeException {
1085     ...
1086 }
1087
```

1088 This exception signals problems in the interaction with remote services. These are exceptions
1089 that can be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException
1090 that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since
1091 it most likely requires human intervention

1092 8.6 InvalidServiceException

1093 The following snippet shows the *InvalidServiceException*.

```
1094 package org.oasisopen.sca;
1095
1096
1097 public class InvalidServiceException extends ServiceRuntimeException {
1098     ...
1099 }
1100
```

1101 This exception signals that the ServiceReference is no longer valid. This can happen when the
1102 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1103 be resolved by retrying the operation and will most likely require human intervention.

1104 8.7 Constants

1105 The SCA *Constants* interface defines a number of constant values that are used in the SCA Java
1106 APIs and Annotations. The following snippet shows the Constants interface:

```
1107 package org.oasisopen.sca;
1108
1109 public interface Constants {
1110     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";
1111     String SCA_PREFIX = "{ "+SCA_NS+"}";
1112 }
1113
```

1114 8.8 SCAClient Interface

1115 The SCAClient interface can be used by client code to obtain a proxy reference object for a service
1116 within an SCA Domain, through which the client code can invoke operations of that service. This
1117 is particularly useful for client code that is running outside the SCA Domain containing the target
1118 service, for example where the code is "unmanaged" and is not running under an SCA runtime.

1119 The following shows the *SCAClient* interface:

```
1120 package org.oasisopen.sca.client;
1121
1122 public interface SCAClient {
1123
1124     <T> T getService(Class<T> interfaze,
1125                   String serviceURI,
1126                   URI domainURI)
```

1127 throws `NoSuchServiceException`, `NoSuchDomainException`;

1128]

1129 **getService method:**

1130 Obtains a proxy reference object for a specified target service in a specified SCA Domain.

1131 **Returns:**

- 1132 • **proxy object** which implements the business interface `T`
- 1133 Invocations of a business method of the proxy causes the invocation of the corresponding
- 1134 operation of the target service .

1135 **Parameters:**

- 1136 • **interfaze** - a Java interface class which is the business interface of the target service
- 1137 • **serviceURI** - a String containing the relative URI of the target service within its SCA
- 1138 Domain.
- 1139 Takes the form componentName/serviceName or can also take the extended form
- 1140 componentName/serviceName/bindingName to use a specific binding of the target service
- 1141 • **domainURI** - a URI for the SCA Domain containing the target service

1142 **Exceptions:**

- 1143 • **NoSuchServiceException** - thrown if a service with the relative URI **serviceURI** and a
- 1144 business interface which matches **interfaze** cannot be found in the Domain identified by
- 1145 **domainURI**
- 1146 • **NoSuchDomainException** - thrown if the Domain identified by **domainURI** cannot be
- 1147 found

1148

1149 **8.9 SCAClientFactory Class**

1150 The SCAClientFactory class provides the means for client code to obtain an object which

1151 implements the SCAClient interface which is used in turn to obtain a proxy reference object to a

1152 service within an SCA Domain.

1153 The SCAClientFactory is an abstract class which provides a set of static newInstance(...) methods

1154 which the client can invoke in order to obtain a object implementing the SCAClient interface.

1155 The SCAClientFactory class is as follows:

1156

1157 package org.oasisopen.sca.client;

1158

1159 import java.util.Properties;

1160 import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;

1161

1162 public abstract class SCAClientFactory {

1163

1164 protected static SCAClientFactoryFinderImpl defaultFactoryFinder;

1165

1166 public static SCAClient newInstance() {

1167 return newInstance(null, null);

1168 }

1169

1170

1171 public static SCAClient newInstance(Properties properties) {

1172 return newInstance(properties, null);

1173 }

1174

1175 public static SCAClient newInstance(ClassLoader classLoader) {

← --- Formatted: Indent: Before:
0.25"

```

1176     return newInstance(null, classLoader);
1177 }
1178
1179 public static SCAClient newInstance(Properties properties,
1180                                 ClassLoader classLoader) {
1181     final SCAClientFactoryFinderImpl finder =
1182     defaultFactoryFinder != null ? defaultFactoryFinder :
1183     new SCAClientFactoryFinderImpl();
1184     final SCAClientFactory factory
1185     = finder.find(properties, classLoader);
1186     return factory.createSCAClient();
1187 }
1188
1189 protected abstract SCAClient createSCAClient();
1190 }

```

Deleted: ¶

1191 **newInstance() method:**

1192 Obtains a object implementing the SCAClient interface.

1193 Returns:

- 1194 • **object** which implements the SCAClient interface

1195 Parameters:

- 1196 • **none**

1197 Exceptions:

- 1198 • **none**

1200 **newInstance(Properties) method:**

1201 Obtains a object implementing the SCAClient interface, using a specified set of properties.

1202 Returns:

- 1203 • **object** which implements the SCAClient interface

1204 Parameters:

- 1205 • **properties** - a set of Properties that can be used when creating the object which
1206 implements the SCAClient interface.

1207 Exceptions:

- 1208 • **none**

1210 **newInstance(Classloader) method:**

1211 Obtains a object implementing the SCAClient interface using a specified classloader.

1212 Returns:

- 1213 • **object** which implements the SCAClient interface

1214 Parameters:

- 1215 • **classLoader** - a ClassLoader to use when creating the object which implements the
1216 SCAClient interface.

1217 Exceptions:

- 1218 • **none**

1219

1220 **newInstance(Properties, Classloader) method:**
1221 Obtains a object implementing the SCAClient interface using a specified set of properties and a
1222 specified classloader.

1223 Returns:

- 1224 • **object** which implements the SCAClient interface

1225 Parameters:

- 1226 • **properties** - a set of Properties that can be used when creating the object which
1227 implements the SCAClient interface.
- 1228 • **classLoader** - a ClassLoader to use when creating the object which implements the
1229 SCAClient interface.

1230 Exceptions:

- 1231 • **none**

1232

1233 **defaultFactory protected field:**

1234 Provides a means by which a provider of an SCAClientFactory implementation can inject its
1235 implementation subclass into the abstract SCAClientFactory class - once this is done, future
1236 invocations of the SCAClientFactory return the implementation subclass.

1237 **8.10 SCAClientFactoryFinder Interface**

1238 The SCAClientFactoryFinder interface is a Service Provider Interface representing a
1239 SCAClientFactory finder. SCA provides a default reference implementation of this interface. SCA
1240 runtime vendors can create alternative implementations of this interface that use different class
1241 loading or lookup mechanisms.

1242 **package** org.oasisopen.sca.client;

1243 **import** java.util.Properties;

1244 **public interface** SCAClientFactoryFinder {

1245 _____ SCAClientFactory find(Properties properties,
1246 _____ ClassLoader classLoader);

1247 }
1248
1249

1250 **8.11 SCAClientFactoryFinderImpl Class**

1251 This class is a default implementation of an SCAClientFactoryFinder, which is used to find an
1252 implementation of the SCAClientFactory interface, as used to obtain an SCAClient object for use by
1253 a client. SCA runtime providers can replace this implementation with their own version.

1254 **package** org.oasisopen.sca.client.impl;

1255 **import** org.oasisopen.sca.client.SCAClientFactoryFinder;

1256 **import** java.io.BufferedReader;

1257 **import** java.io.Closeable;

1258 **import** java.io.IOException;

1259 **import** java.io.InputStream;

1260 **import** java.io.InputStreamReader;

1261 **import** java.net.URL;

1262 **import** java.util.Properties;

1263 **import** org.oasisopen.sca.SCARuntimeException;

1264 **import** org.oasisopen.sca.client.SCAClientFactory;

1265
1266
1267

Formatted: Bullets and
Numbering

Formatted: Body Text,Body
Text Char,Body Text Char1
Char1,Body Text Char Char
Char1,Body Text Char1 Char1
Char Char,Body Text Char
Char Char1 Char Char,Body
Text Char1 Char1 Char Char
Char Char,Body Text Char
Char Char1 Char Char Char
Char,Body Text Char1

Formatted: Indent: Before:
0.25"

Formatted: Body Text,Body
Text Char,Body Text Char1
Char1,Body Text Char Char
Char1,Body Text Char1 Char1
Char Char,Body Text Char
Char Char1 Char Char,Body
Text Char1 Char1 Char Char
Char Char,Body Text Char
Char Char1 Char Char Char
Char,Body Text Char1

Formatted: Bullets and
Numbering

Formatted: Body Text,Body
Text Char,Body Text Char1
Char1,Body Text Char Char
Char1,Body Text Char1 Char1
Char Char,Body Text Char
Char Char1 Char Char,Body
Text Char1 Char1 Char Char
Char Char,Body Text Char
Char Char1 Char Char Char
Char,Body Text Char1

Formatted: Indent: Before:
0.25"

Formatted: French France

```

1268 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder
1269 {
1270
1271 private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
1272 SCAClientFactory.class.getName();
1273
1274 private static final String
1275 SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
1276 = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
1277
1278 public SCAClientFactoryFinderImpl() {
1279 }
1280
1281 public SCAClientFactory find(Properties properties,
1282 ClassLoader classLoader) {
1283 if (classLoader == null) {
1284 classLoader = getThreadContextClassLoader();
1285 }
1286 final String factoryImplClassName =
1287 discoverProviderFactoryImplClass(properties, classLoader);
1288 final Class<? extends SCAClientFactory> factoryImplClass
1289 = loadProviderFactoryClass(factoryImplClassName,
1290 classLoader);
1291 final SCAClientFactory factory =
1292 instantiateSCAClientFactoryClass(factoryImplClass);
1293 return factory;
1294 }
1295
1296 private static ClassLoader getThreadContextClassLoader() {
1297 final ClassLoader threadClassLoader =
1298 Thread.currentThread().getContextClassLoader();
1299 return threadClassLoader;
1300 }
1301
1302 private static String
1303 discoverProviderFactoryImplClass(Properties properties,
1304 ClassLoader classLoader)
1305 throws SCARuntimeException {
1306 String providerClassName =
1307 checkPropertiesForSPIClassName(properties);
1308 if (providerClassName != null) {
1309 return providerClassName;
1310 }
1311
1312 providerClassName =
1313 checkPropertiesForSPIClassName(System.getProperties());
1314 if (providerClassName != null) {
1315 return providerClassName;
1316 }
1317
1318 providerClassName =
1319 checkMETAINFServicesForSIPClassName(classLoader);
1320 if (providerClassName == null) {
1321 throw new SCARuntimeException(
1322 "Failed to find implementation for SCAClientFactory");
1323 }
1324 }
1325

```

Formatted: Indent: Before: 0.58"

Formatted: Indent: Before: 0.58", First line: 0.42"

Formatted: Indent: Before: 0.25"

```

1326     return providerClassName;
1327 }
1328
1329 private static String
1330 checkPropertiesForSPIClassName(Properties properties) {
1331     if (properties == null) {
1332         return null;
1333     }
1334
1335     final String providerClassName =
1336     properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
1337     if (providerClassName != null && providerClassName.length() > 0) {
1338         return providerClassName;
1339     }
1340
1341     return null;
1342 }
1343
1344 private static String checkMETAINFServicesForSIPClassName(ClassLoader
1345 cl)
1346 {
1347     final URL url =
1348     cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
1349     if (url == null) {
1350         return null;
1351     }
1352
1353     InputStream in = null;
1354     try {
1355         in = url.openStream();
1356         BufferedReader reader = null;
1357         try {
1358             reader =
1359             new BufferedReader(new InputStreamReader(in,
1360                 "UTF-8"));
1361
1362             String line;
1363             while ((line = readNextLine(reader)) != null) {
1364                 if (!line.startsWith("#") && line.length() > 0) {
1365                     return line;
1366                 }
1367             }
1368
1369             return null;
1370         } finally {
1371             closeStream(reader);
1372         }
1373     } catch (IOException ex) {
1374         throw new SCARuntimeException(
1375             "Failed to discover SCAclientFactory provider", ex);
1376     } finally {
1377         closeStream(in);
1378     }
1379 }
1380
1381 private static String readNextLine(BufferedReader reader)
1382 throws IOException {
1383

```

```

1384     String line = reader.readLine();
1385     if (line != null) {
1386         line = line.trim();
1387     }
1388     return line;
1389 }
1390
1391 private static Class<? extends SCAClientFactory>
1392     loadProviderFactoryClass(String factoryImplClassName,
1393                             ClassLoader classLoader)
1394     throws SCARuntimeException {
1395
1396     try {
1397         final Class<?> providerClass =
1398             classLoader.loadClass(factoryImplClassName);
1399         final Class<? extends SCAClientFactory> providerFactoryClass =
1400             providerClass.asSubclass(SCAClientFactory.class);
1401         return providerFactoryClass;
1402     } catch (ClassNotFoundException ex) {
1403         throw new SCARuntimeException(
1404             "Failed to load SCAClientFactory implementation class "
1405             + factoryImplClassName, ex);
1406     } catch (ClassCastException ex) {
1407         throw new SCARuntimeException(
1408             "Loaded SCAClientFactory implementation class "
1409             + factoryImplClassName
1410             + " is not a subclass of "
1411             + SCAClientFactory.class.getName(), ex);
1412     }
1413 }
1414
1415 private static SCAClientFactory
1416     instantiateSCAClientFactoryClass(
1417         Class<? extends SCAClientFactory> factoryImplClass)
1418     throws SCARuntimeException {
1419
1420     try {
1421         final SCAClientFactory provider =
1422             factoryImplClass.newInstance();
1423         return provider;
1424     } catch (Throwable ex) {
1425         throw new SCARuntimeException(
1426             "Failed to instantiate SCAClientFactory implementation
1427             class " + factoryImplClass, ex);
1428     }
1429 }
1430
1431 private static void closeStream(Closeable closeable) {
1432     if (closeable != null) {
1433         try {
1434             closeable.close();
1435         } catch (IOException ex) {
1436             throw new SCARuntimeException("Failed to close stream",
1437                 ex);
1438         }
1439     }
1440 }
1441 }

```

Formatted: Body Text,Body Text Char,Body Text Char1 Char1,Body Text Char Char1,Body Text Char1 Char1 Char Char,Body Text Char Char Char1 Char Char,Body Text Char Char Char,Body Text Char Char Char1 Char Char Char Char,Body Text Char1

1442 **8.12 NoSuchDomainException**

1443 The following shows the NoSuchDomainException:

1444 `package org.oasisopen.sca;`

1445 `public class NoSuchDomainException extends Exception {`
1446 `...`
1447 `}`

1448 This exception indicates that the Domain specified could not be found.

Formatted: Bullets and Numbering

1450 **8.13 NoSuchServiceException**

1451 The following shows the NoSuchServiceException:

1452 `package org.oasisopen.sca;`

1453 `public class NoSuchServiceException extends Exception {`
1454 `...`
1455 `}`

1456 This exception indicates that the service specified could not be found.

Formatted: Bullets and Numbering

1458 9 Java Annotations

1459 This section provides definitions of all the Java annotations which apply to SCA.

1460 This specification places constraints on some annotations that are not detectable by a Java
1461 compiler. For example, the definition of the @Property and @Reference annotations indicate that
1462 they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to
1463 constructor parameters. An SCA runtime MUST verify the proper use of all SCA annotations and if
1464 an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the
1465 invalid implementation code. [JCA90001]

1466 SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an
1467 SCA annotation on a static method or a static field of an implementation class and the SCA
1468 runtime MUST NOT instantiate such an implementation class. [JCA90002]

1469 9.1 @AllowsPassByReference

1470 The following Java code defines the **@AllowsPassByReference** annotation:

```
1471  
1472 package org.oasisopen.sca.annotation;  
1473  
1474 import static java.lang.annotation.ElementType.TYPE;  
1475 import static java.lang.annotation.ElementType.METHOD;  
1476 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1477 import java.lang.annotation.Retention;  
1478 import java.lang.annotation.Target;  
1479  
1480 @Target({TYPE, METHOD})  
1481 @Retention(RUNTIME)  
1482 public @interface AllowsPassByReference {  
1483  
1484 }  
1485
```

1486 The **@AllowsPassByReference** annotation is used on implementations of remotable interfaces to
1487 indicate that interactions with the service from a client within the same address space are allowed
1488 to use pass by reference data exchange semantics. The implementation promises that its by-value
1489 semantics will be maintained even if the parameters and return values are actually passed by-
1490 reference. This means that the service will not modify any operation input parameter or return
1491 value, even after returning from the operation. Either a whole class implementing a remotable
1492 service or an individual remotable service method implementation can be annotated using the
1493 @AllowsPassByReference annotation.

1494 @AllowsPassByReference has no attributes

1495 The following snippet shows a sample where @AllowsPassByReference is defined for the
1496 implementation of a service method on the Java component implementation class.

```
1497  
1498 @AllowsPassByReference  
1499 public String hello(String message) {  
1500     ...  
1501 }
```

1502 9.2 @Authentication

1503 The following Java code defines the **@Authentication** annotation:

```

1504
1505 package org.oasisopen.sca.annotation;
1506
1507 import static java.lang.annotation.ElementType.FIELD;
1508 import static java.lang.annotation.ElementType.METHOD;
1509 import static java.lang.annotation.ElementType.PARAMETER;
1510 import static java.lang.annotation.ElementType.TYPE;
1511 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1512 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1513
1514 import java.lang.annotation.Inherited;
1515 import java.lang.annotation.Retention;
1516 import java.lang.annotation.Target;
1517
1518 @Inherited
1519 @Target({TYPE, FIELD, METHOD, PARAMETER})
1520 @Retention(RUNTIME)
1521 @Intent(Authentication.AUTHENTICATION)
1522 public @interface Authentication {
1523     String AUTHENTICATION = SCA_PREFIX + "authentication";
1524     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
1525     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";
1526
1527     /**
1528      * List of authentication qualifiers (such as "message"
1529      * or "transport").
1530      *
1531      * @return authentication qualifiers
1532      */
1533     @Qualifier
1534     String[] value() default "";
1535 }

```

1536 The **@Authentication** annotation is used to indicate that the invocation requires authentication.
1537 See the [section on Application of Intent Annotations](#) for samples and details.

1538 9.3 @Callback

1539 The following Java code defines the **@Callback** annotation:

```

1540
1541 package org.oasisopen.sca.annotation;
1542
1543 import static java.lang.annotation.ElementType.TYPE;
1544 import static java.lang.annotation.ElementType.METHOD;
1545 import static java.lang.annotation.ElementType.FIELD;
1546 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1547 import java.lang.annotation.Retention;
1548 import java.lang.annotation.Target;
1549
1550 @Target(TYPE, METHOD, FIELD)
1551 @Retention(RUNTIME)
1552 public @interface Callback {
1553     Class<?> value() default Void.class;
1554 }
1555
1556
1557

```

1558 The @Callback annotation is used to annotate a service interface with a callback interface by
1559 specifying the Java class object of the callback interface as an attribute.

1560 The @Callback annotation has the following attribute:

- 1561 • **value** – the name of a Java class file containing the callback interface

1562

1563 The @Callback annotation can also be used to annotate a method or a field of an SCA
1564 implementation class, in order to have a callback object injected. When used to annotate a
1565 method or a field of an implementation class for injection of a callback object, the @Callback
1566 annotation MUST NOT specify any attributes. [JCA90046]

1567 An example use of the @Callback annotation to declare a callback interface follows:

```
1568 package somepackage;  
1569 import org.oasisopen.sca.annotation.Callback;  
1570 import org.oasisopen.sca.annotation.Remotable;  
1571 @Remotable  
1572 @Callback(MyServiceCallback.class)  
1573 public interface MyService {  
1574     void someMethod(String arg);  
1575 }  
1576  
1577 @Remotable  
1578 public interface MyServiceCallback {  
1579     void receiveResult(String result);  
1580 }  
1581  
1582 }
```

1583

1584 In this example, the implied component type is:

```
1585 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
1586     <service name="MyService">  
1587         <interface.java interface="somepackage.MyService"  
1588             callbackInterface="somepackage.MyServiceCallback"/>  
1589     </service>  
1590 </componentType>
```

1592 9.4 @ComponentName

1593 The following Java code defines the @ComponentName annotation:

1594

```
1595 package org.oasisopen.sca.annotation;  
1596  
1597 import static java.lang.annotation.ElementType.METHOD;  
1598 import static java.lang.annotation.ElementType.FIELD;  
1599 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1600 import java.lang.annotation.Retention;  
1601 import java.lang.annotation.Target;  
1602  
1603 @Target({METHOD, FIELD})  
1604 @Retention(RUNTIME)  
1605 public @interface ComponentName {  
1606  
1607 }  
1608
```

1609 The @ComponentName annotation is used to denote a Java class field or setter method that is
1610 used to inject the component name.

1611 The following snippet shows a component name field definition sample.

```
1612  
1613 @ComponentName  
1614 private String componentName;  
1615
```

1616 The following snippet shows a component name setter method sample.

```
1617 @ComponentName  
1618 public void setComponentName(String name) {  
1619     //...  
1620 }  
1621
```

1622 9.5 @Confidentiality

1623 The following Java code defines the **@Confidentiality** annotation:

```
1624 package org.oasisopen.sca.annotations;  
1625  
1626 import static java.lang.annotation.ElementType.FIELD;  
1627 import static java.lang.annotation.ElementType.METHOD;  
1628 import static java.lang.annotation.ElementType.PARAMETER;  
1629 import static java.lang.annotation.ElementType.TYPE;  
1630 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1631 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
1632  
1633 import java.lang.annotation.Inherited;  
1634 import java.lang.annotation.Retention;  
1635 import java.lang.annotation.Target;  
1636  
1637 @Inherited  
1638 @Target({TYPE, FIELD, METHOD, PARAMETER})  
1639 @Retention(RUNTIME)  
1640 @Intent(Confidentiality.CONFIDENTIALITY)  
1641 public @interface Confidentiality {  
1642     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
1643     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
1644     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";  
1645  
1646     /**  
1647      * List of confidentiality qualifiers such as "message" or  
1648      * "transport".  
1649      *  
1650      * @return confidentiality qualifiers  
1651      */  
1652     @Qualifier  
1653     String[] value() default "";  
1654 }  
1655
```

1656 The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.

1657 See the [section on Application of Intent Annotations](#) for samples and details.

1658 9.6 @Constructor

1659 The following Java code defines the **@Constructor** annotation:

```
1660 package org.oasisopen.sca.annotation;
1661
1662 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1663 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1664 import java.lang.annotation.Retention;
1665 import java.lang.annotation.Target;
1666
1667 @Target(CONSTRUCTOR)
1668 @Retention(RUNTIME)
1669 public @interface Constructor { }
```

1672 The @Constructor annotation is used to mark a particular constructor to use when instantiating a
1673 Java component implementation. If a constructor of an implementation class is annotated with
1674 @Constructor and the constructor has parameters, each of these parameters MUST have either a
1675 @Property annotation or a @Reference annotation. [JCA90003]

1676 The following snippet shows a sample for the @Constructor annotation.

```
1677
1678 public class HelloServiceImpl implements HelloService {
1679     public HelloServiceImpl(){
1680         ...
1681     }
1682
1683     @Constructor
1684     public HelloServiceImpl(@Property(name="someProperty")
1685                             String someProperty ){
1686         ...
1687     }
1688
1689     public String hello(String message) {
1690         ...
1691     }
1692 }
1693
```

Comment [ME4]: There also needs to be a normative statement that at most 1 constructor can be annotated with @Constructor

1694 9.7 @Context

1695 The following Java code defines the **@Context** annotation:

```
1696 package org.oasisopen.sca.annotation;
1697
1698 import static java.lang.annotation.ElementType.METHOD;
1699 import static java.lang.annotation.ElementType.FIELD;
1700 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1701 import java.lang.annotation.Retention;
1702 import java.lang.annotation.Target;
1703
1704 @Target({METHOD, FIELD})
1705 @Retention(RUNTIME)
1706 public @interface Context { }
```

1709 }
1710

1711 The @Context annotation is used to denote a Java class field or a setter method that is used to
1712 inject a composite context for the component. The type of context to be injected is defined by the
1713 type of the Java class field or type of the setter method input argument; the type is either
1714 **ComponentContext** or **RequestContext**.

1715 The @Context annotation has no attributes.

1716 The following snippet shows a ComponentContext field definition sample.

1717

```
1718 @Context  
1719 protected ComponentContext context;  
1720
```

1721 The following snippet shows a RequestContext field definition sample.

1722

```
1723 @Context  
1724 protected RequestContext context;
```

1725 9.8 @Destroy

1726 The following Java code defines the **@Destroy** annotation:

1727

```
1728 package org.oasisopen.sca.annotation;  
1729  
1730 import static java.lang.annotation.ElementType.METHOD;  
1731 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1732 import java.lang.annotation.Retention;  
1733 import java.lang.annotation.Target;  
1734  
1735 @Target(METHOD)  
1736 @Retention(RUNTIME)  
1737 public @interface Destroy {  
1738  
1739 }  
1740
```

1741 The @Destroy annotation is used to denote a single Java class method that will be called when the
1742 scope defined for the implementation class ends. A method annotated with @Destroy MAY have
1743 any access modifier and MUST have a void return type and no arguments. [JCA90004]

1744 If there is a method annotated with @Destroy that matches the criteria for the annotation, the
1745 SCA runtime MUST call the annotated method when the scope defined for the implementation
1746 class ends. [JCA90005]

1747 The following snippet shows a sample for a destroy method definition.

1748

```
1749 @Destroy  
1750 public void myDestroyMethod() {  
1751     ...  
1752 }
```

1753 9.9 @EagerInit

1754 The following Java code defines the **@EagerInit** annotation:

1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772

```
package org.oasisopen.sca.annotation;  
  
import static java.lang.annotation.ElementType.TYPE;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
@Target(TYPE)  
@Retention(RUNTIME)  
public @interface EagerInit {
```

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

9.10 @Init

1773
1774

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

1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789

```
package org.oasisopen.sca.annotation;  
  
import static java.lang.annotation.ElementType.METHOD;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
  
@Target(METHOD)  
@Retention(RUNTIME)  
public @interface Init {
```

1790
1791
1792

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

1793
1794
1795

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

1796

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

1797
1798
1799
1800
1801

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

9.11 @Integrity

1802
1803
1804

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

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

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

```

1805 package org.oasisopen.sca.annotation;
1806
1807 import static java.lang.annotation.ElementType.FIELD;
1808 import static java.lang.annotation.ElementType.METHOD;
1809 import static java.lang.annotation.ElementType.PARAMETER;
1810 import static java.lang.annotation.ElementType.TYPE;
1811 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1812 import static org.oasisopen.Constants.SCA_PREFIX;
1813
1814 import java.lang.annotation.Inherited;
1815 import java.lang.annotation.Retention;
1816 import java.lang.annotation.Target;
1817
1818 @Inherited
1819 @Target({TYPE, FIELD, METHOD, PARAMETER})
1820 @Retention(RUNTIME)
1821 @Intent(Integrity.INTEGRITY)
1822 public @interface Integrity {
1823     String INTEGRITY = SCA_PREFIX + "integrity";
1824     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
1825     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
1826
1827     /**
1828      * List of integrity qualifiers (such as "message" or "transport").
1829      *
1830      * @return integrity qualifiers
1831      */
1832     @Qualifier
1833     String[] value() default "";
1834 }
1835

```

1836 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no tampering of the messages between client and service).

1838 See the [section on Application of Intent Annotations](#) for samples and details.

1839 9.12 @Intent

1840 The following Java code defines the **@Intent** annotation:

```

1841 package org.oasisopen.sca.annotation;
1842
1843 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
1844 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1845 import java.lang.annotation.Retention;
1846 import java.lang.annotation.Target;
1847
1848 @Target({ANNOTATION_TYPE})
1849 @Retention(RUNTIME)
1850 public @interface Intent {
1851     /**
1852      * The qualified name of the intent, in the form defined by
1853      * {@link javax.xml.namespace.QName#toString}.
1854      * @return the qualified name of the intent
1855      */
1856     String value() default "";
1857
1858     /**
1859

```



```

1860     * The XML namespace for the intent.
1861     * @return the XML namespace for the intent
1862     */
1863     String targetNamespace() default "";
1864
1865     /**
1866     * The name of the intent within its namespace.
1867     * @return name of the intent within its namespace
1868     */
1869     String localPart() default "";
1870 }
1871

```

1872 The @Intent annotation is used for the creation of new annotations for specific intents. It is not
 1873 expected that the @Intent annotation will be used in application code.

1874 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
 1875 define new intent annotations.

1876 9.13 @OneWay

1877 The following Java code defines the **@OneWay** annotation:

```

1878
1879 package org.oasisopen.sca.annotation;
1880
1881 import static java.lang.annotation.ElementType.METHOD;
1882 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1883 import java.lang.annotation.Retention;
1884 import java.lang.annotation.Target;
1885
1886 @Target(METHOD)
1887 @Retention(RUNTIME)
1888 public @interface OneWay {
1889
1890
1891 }
1892

```

1893 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
 1894 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
 1895 [Programming](#).

Comment [ME5]: Needs recasting in a normative form of statement

1896 The @OneWay annotation has no attributes.

1897 The following snippet shows the use of the @OneWay annotation on an interface.

```

1898 package services.hello;
1899
1900 import org.oasisopen.sca.annotation.OneWay;
1901
1902 public interface HelloService {
1903     @OneWay
1904     void hello(String name);
1905 }

```

1906 9.14 @PolicySets

1907 The following Java code defines the **@PolicySets** annotation:

```

1908 package org.oasisopen.sca.annotation;
1909

```

```

1910
1911 import static java.lang.annotation.ElementType.FIELD;
1912 import static java.lang.annotation.ElementType.METHOD;
1913 import static java.lang.annotation.ElementType.PARAMETER;
1914 import static java.lang.annotation.ElementType.TYPE;
1915 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1916
1917 import java.lang.annotation.Retention;
1918 import java.lang.annotation.Target;
1919
1920 @Target({TYPE, FIELD, METHOD, PARAMETER})
1921 @Retention(RUNTIME)
1922 public @interface PolicySets {
1923     /**
1924      * Returns the policy sets to be applied.
1925      *
1926      * @return the policy sets to be applied
1927      */
1928     String[] value() default "";
1929 }
1930

```

1931 The **@PolicySets** annotation is used to attach one or more SCA Policy Sets to a Java
1932 implementation class or to one of its subelements.

1933 See the [section "Policy Set Annotations"](#) for details and samples.

1934 9.15 @Property

1935 The following Java code defines the **@Property** annotation:

```

1936 package org.oasisopen.sca.annotation;
1937
1938 import static java.lang.annotation.ElementType.METHOD;
1939 import static java.lang.annotation.ElementType.FIELD;
1940 import static java.lang.annotation.ElementType.PARAMETER;
1941 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1942 import java.lang.annotation.Retention;
1943 import java.lang.annotation.Target;
1944
1945 @Target({METHOD, FIELD, PARAMETER})
1946 @Retention(RUNTIME)
1947 public @interface Property {
1948
1949     String name() default "";
1950     boolean required() default true;
1951 }
1952

```

1953 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1954 parameter that is used to inject an SCA property value. The type of the property injected, which
1955 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1956 the type of the input parameter of the setter method or constructor.

1957 The @Property annotation can be used on fields, on setter methods or on a constructor method
1958 parameter. However, **the @Property annotation MUST NOT be used on a class field that is declared
1959 as final. [JCA90011]**

1960 Properties can also be injected via setter methods even when the @Property annotation is not
1961 present. However, **the @Property annotation MUST be used in order to inject a property onto a
1962 non-public field. [JCA90012]** In the case where there is no @Property annotation, the name of the
1963 property is the same as the name of the field or setter.

Deleted: the @Property annotation MUST NOT be used on a class field that is declared as final.

Deleted: the @Property annotation MUST NOT be used on a class field that is declared as final.

1964 Where there is both a setter method and a field for a property, the setter method is used.

1965 The @Property annotation has the following attributes:

- 1966 • **name (optional)** – the name of the property. For a field annotation, the default is the
1967 name of the field of the Java class. For a setter method annotation, the default is the
1968 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a
1969 @Property annotation applied to a constructor parameter, there is no default value for the
1970 name attribute and the name attribute MUST be present. [JCA90013]
- 1971 • **required (optional)** – a boolean value which specifies whether injection of the property
1972 value is required or not, where true means injection is required and false means injection
1973 is not required. Defaults to true. For a @Property annotation applied to a constructor
1974 parameter, the required attribute MUST have the value true. [JCA90014]

1975

1976 The following snippet shows a property field definition sample.

1977

```
1978 @Property(name="currency", required=true)  
1979 protected String currency;
```

1980

1981 The following snippet shows a property setter sample

1982

```
1983 @Property(name="currency", required=true)  
1984 public void setCurrency( String theCurrency ) {  
1985     ....  
1986 }  
1987
```

1988 For a @Property annotation, if the the type of the Java class field or the type of the input
1989 parameter of the setter method or constructor is defined as an array or as any type that extends
1990 or implements java.util.Collection, then the SCA runtime MUST introspect the component type of
1991 the implementation with a <property/> element with a @many attribute set to true, otherwise
1992 @many MUST be set to false. [JCA90047]

1993 The following snippet shows the definition of a configuration property using the @Property
1994 annotation for a collection.

```
1995 ...  
1996 private List<String> helloConfigurationProperty;  
1997  
1998 @Property(required=true)  
1999 public void setHelloConfigurationProperty(List<String> property) {  
2000     helloConfigurationProperty = property;  
2001 }  
2002 ...
```

2003 9.16 @Qualifier

2004 The following Java code defines the @Qualifier annotation:

```
2005 package org.oasisopen.sca.annotation;  
2006  
2007 import static java.lang.annotation.ElementType.METHOD;  
2008 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
2009  
2010
```

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

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

```

2011 import java.lang.annotation.Retention;
2012 import java.lang.annotation.Target;
2013
2014 @Target(METHOD)
2015 @Retention(RUNTIME)
2016 public @interface Qualifier {
2017 }
2018

```

2019 The @Qualifier annotation is applied to an attribute of a specific intent annotation definition,
 2020 defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the
 2021 intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the
 2022 intent has qualifiers. [JCA90015]

2023 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
 2024 define new intent annotations.

2025 9.17 @Reference

2026 The following Java code defines the **@Reference** annotation:

```

2027
2028 package org.oasisopen.sca.annotation;
2029
2030 import static java.lang.annotation.ElementType.METHOD;
2031 import static java.lang.annotation.ElementType.FIELD;
2032 import static java.lang.annotation.ElementType.PARAMETER;
2033 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2034 import java.lang.annotation.Retention;
2035 import java.lang.annotation.Target;
2036 @Target({METHOD, FIELD, PARAMETER})
2037 @Retention(RUNTIME)
2038 public @interface Reference {
2039
2040     String name() default "";
2041     boolean required() default true;
2042 }
2043

```

2044 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
 2045 constructor parameter that is used to inject a service that resolves the reference. The interface of
 2046 the service injected is defined by the type of the Java class field or the type of the input parameter
 2047 of the setter method or constructor.

2048 The @Reference annotation MUST NOT be used on a class field that is declared as final.
 2049 [JCA90016]

2050 References can also be injected via setter methods even when the @Reference annotation is not
 2051 present. However, the @Reference annotation MUST be used in order to inject a reference onto a
 2052 non-public field. [JCA90017] In the case where there is no @Reference annotation, the name of
 2053 the reference is the same as the name of the field or setter.

2054 Where there is both a setter method and a field for a reference, the setter method is used.

2055 The @Reference annotation has the following attributes:

- 2056 • **name : String (optional)** – the name of the reference. For a field annotation, the default is
 2057 the name of the field of the Java class. For a setter method annotation, the default is the
 2058 JavaBeans property name corresponding to the setter method name. For a @Reference
 2059 annotation applied to a constructor parameter, there is no default for the name attribute
 2060 and the name attribute MUST be present. [JCA90018]

- 2061
- **required (optional)** – a boolean value which specifies whether injection of the service reference is required or not, where true means injection is required and false means injection is not required. Defaults to true. For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true. [JCA90019]
- 2062
- 2063
- 2064
- 2065

2066 The following snippet shows a reference field definition sample.

2067

```
2068 @Reference(name="stockQuote", required=true)
2069 protected StockQuoteService stockQuote;
```

2070

2071 The following snippet shows a reference setter sample

2072

```
2073 @Reference(name="stockQuote", required=true)
2074 public void setStockQuote( StockQuoteService theSQService ) {
2075     ...
2076 }
```

2077

2078 The following fragment from a component implementation shows a sample of a service reference using the @Reference annotation. The name of the reference is "helloService" and its type is HelloService. The clientMethod() calls the "hello" operation of the service referenced by the helloService reference.

2079

2080

2081

```
2082
2083 package services.hello;
2084
2085 private HelloService helloService;
2086
2087 @Reference(name="helloService", required=true)
2088 public setHelloService(HelloService service) {
2089     helloService = service;
2090 }
2091
2092 public void clientMethod() {
2093     String result = helloService.hello("Hello World!");
2094     ...
2095 }
2096
```

2097

2098

2099

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

2100

```
2101 <?xml version="1.0" encoding="ASCII"?>
2102 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
2103
2104     <!-- Any services offered by the component would be listed here -->
2105     <reference name="helloService" multiplicity="1..1">
2106         <interface.java interface="services.hello.HelloService"/>
2107     </reference>
2108
2109 </componentType>
2110
```

2111 **If the type of a reference is not an array or any type that extends or implements**
2112 **java.util.Collection, then the SCA runtime MUST introspect the component type of the**
2113 **implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference**
2114 **annotation required attribute is false and with @multiplicity=1..1 if the @Reference**
2115 **annotation required attribute is true. [JCA90020]**

2116 **If the type of a reference is defined as an array or as any type that extends or implements**
2117 **java.util.Collection, then the SCA runtime MUST introspect the component type of the**
2118 **implementation with a <reference/> element with @multiplicity=0..n if the @Reference**
2119 **annotation required attribute is false and with @multiplicity=1..n if the @Reference**
2120 **annotation required attribute is true. [JCA90021]**

2121 The following fragment from a component implementation shows a sample of a service reference
2122 definition using the @Reference annotation on a java.util.List. The name of the reference is
2123 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
2124 services referenced by the helloServices reference. In this case, at least one HelloService needs
2125 to be present, so **required** is true.

```
2126 @Reference(name="helloServices", required=true)  
2127 protected List<HelloService> helloServices;  
2128  
2129 public void clientMethod() {  
2130  
2131     ...  
2132     for (int index = 0; index < helloServices.size(); index++) {  
2133         HelloService helloService =  
2134             (HelloService)helloServices.get(index);  
2135         String result = helloService.hello("Hello World!");  
2136     }  
2137     ...  
2138 }  
2139  
2140 }
```

2141 The following snippet shows the XML representation of the component type reflected from for the
2142 former component implementation fragment. There is no need to author this component type in
2143 this case since it can be reflected from the Java class.

```
2144  
2145 <?xml version="1.0" encoding="ASCII"?>  
2146 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
2147  
2148     <!-- Any services offered by the component would be listed here -->  
2149     <reference name="helloServices" multiplicity="1..n">  
2150         <interface.java interface="services.hello.HelloService"/>  
2151     </reference>  
2152  
2153 </componentType>
```

2154
2155 **An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by**
2156 **the SCA runtime as null. [JCA90022] An unwired reference with a multiplicity of 0..n MUST be**
2157 **presented to the implementation code by the SCA runtime as an empty array or empty collection.**
2158 **[JCA90023]**

2159 9.17.1 Reinjection

2160 References MAY be reinjected by an SCA runtime after the initial creation of a component if the
2161 reference target changes due to a change in wiring that has occurred since the component was
2162 initialized. [JCA90024]

2163 **In order for reinjection to occur, the following MUST be true:**

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

Deleted: If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the compo... [1]

Deleted: If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the compo... [2]

Deleted: If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the compo... [3]

Deleted: An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null

Deleted: An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null

Deleted: An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty... [4]

Deleted: An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty... [5]

2164 1. The component MUST NOT be STATELESS scoped.

2165 2. The reference MUST use either field-based injection or setter injection. References that are

2166 injected through constructor injection MUST NOT be changed.

2167 [JCA90025]

2168 Setter injection allows for code in the setter method to perform processing in reaction to a change.

2169 If a reference target changes and the reference is not reinjected, the reference MUST continue to

2170 work as if the reference target was not changed. [JCA90026]

2171 If an operation is called on a reference where the target of that reference has been undeployed,

2172 the SCA runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called

2173 on a reference where the target of the reference has become unavailable for some reason, the

2174 SCA runtime SHOULD throw a ServiceUnavailableException. [JCA90028] If the target service of

2175 the reference is changed, the reference MAY continue to work, depending on the runtime and the

2176 type of change that was made. [JCA90029] If it doesn't work, the exception thrown will depend on

2177 the runtime and the cause of the failure.

2178 A ServiceReference that has been obtained from a reference by ComponentContext.cast()

2179 corresponds to the reference that is passed as a parameter to cast(). If the reference is

2180 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue

2181 to work as if the reference target was not changed. [JCA90030] If the target of a ServiceReference

2182 has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an

2183 operation is invoked on the ServiceReference. [JCA90031] If the target of a ServiceReference has

2184 become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an

2185 operation is invoked on the ServiceReference. [JCA90032] If the target service of a

2186 ServiceReference is changed, the reference MAY continue to work, depending on the runtime and

2187 the type of change that was made. [JCA90033] If it doesn't work, the exception thrown will

2188 depend on the runtime and the cause of the failure.

2189 A reference or ServiceReference accessed through the component context by calling getService()

2190 or getServiceReference() MUST correspond to the current configuration of the domain. This applies

2191 whether or not reinjection has taken place. [JCA90034] If the target of a reference or

2192 ServiceReference accessed through the component context by calling getService() or

2193 getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a

2194 reference to the undeployed or unavailable service, and attempts to call business methods

2195 SHOULD throw an InvalidServiceException or a ServiceUnavailableException. [JCA90035] If the

2196 target service of a reference or ServiceReference accessed through the component context by

2197 calling getService() or getServiceReference() has changed, the returned value SHOULD be a

2198 reference to the changed service. [JCA90036]

2199 The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This

2200 means that in the cases where reference reinjection is not allowed, the array or Collection for a

2201 reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes

2202 occur to the reference wiring or to the targets of the wiring. [JCA90037] In cases where the

2203 contents of a reference array or collection change when the wiring changes or the targets change,

2204 then for references that use setter injection, the setter method MUST be called by the SCA

2205 runtime for any change to the contents. [JCA90038] A reinjected array or Collection for a

2206 reference MUST NOT be the same array or Collection object previously injected to the component.

2207 [JCA90039]

2208

Deleted: In order for reinjection to occur, the following MUST be true:
 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 ... [6]

Deleted: In order for reinjection to occur, the following MUST be true:
 1. The component MUS ... [7]

Deleted: If the target service of the reference is changed, the reference MAY continue to work ... [8]

Deleted: If the target service of the reference is changed, the reference MAY continue to work ... [9]

Deleted: If the target service of a ServiceReference is changed, the refer ... [10]

Deleted: If the target service of a ServiceReference is changed, the refer ... [11]

Deleted: A reference or ServiceReference accessed through the component context by calling ... [12]

Deleted: A reference or ServiceReference accessed through the component context by calling ... [13]

Deleted: If the target of a reference or ServiceReference accessed through the compon ... [14]

Deleted: If the target of a reference or ServiceReference accessed through the compon ... [15]

Deleted: If the target service of a reference or ServiceReference accessed through the compon ... [16]

Deleted: If the target service of a reference or ServiceReference accessed through the compon ... [17]

Deleted: In cases where the contents of a reference array or collection change when the wiring cha ... [18]

Deleted: In cases where the contents of a reference array or collection change when the wiring cha ... [19]

Change event	Effect on		
	Injected Reference or ServiceReference	Existing ServiceReference Object**	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	can be reinjected (if other conditions* apply). If not reinjected, then it	continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.

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

2209

2210 9.18 @Remotable

2211 The following Java code defines the **@Remotable** annotation:

2212

```
2213 package org.oasisopen.sca.annotation;
2214
2215 import static java.lang.annotation.ElementType.TYPE;
2216 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2217 import java.lang.annotation.Retention;
2218 import java.lang.annotation.Target;
```

2219

2220

```
2221 @Target(TYPE)
2222 @Retention(RUNTIME)
2223 public @interface Remotable {
2224
2225 }
2226
```

2227 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable
 2228 service can be published externally as a service and MUST be translatable into a WSDL portType.
 2229 [JCA90040]

2230 The @Remotable annotation has no attributes.

2231 The following snippet shows the Java interface for a remotable service with its @Remotable
 2232 annotation.


```

2233 package services.hello;
2234
2235 import org.oasisopen.sca.annotation.*;
2236
2237 @Remotable
2238 public interface HelloService {
2239     String hello(String message);
2240 }
2241
2242

```

2243 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**
2244 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

2245 Complex data types exchanged via remotable service interfaces need to be compatible with the
2246 marshalling technology used by the service binding. For example, if the service is going to be
2247 exposed using the standard Web Service binding, then the parameters can be JAXB [JAX-B] types
2248 or they can be Service Data Objects (SDOs) [SDO].

2249 Independent of whether the remotable service is called from outside of the composite that
2250 contains it or from another component in the same composite, the data exchange semantics are
2251 **by-value**.

2252 Implementations of remotable services can modify input data during or after an invocation and
2253 can modify return data after the invocation. If a remotable service is called locally or remotely, the
2254 SCA container is responsible for making sure that no modification of input data or post-invocation
2255 modifications to return data are seen by the caller.

2256 The following snippet shows a remotable Java service interface.

```

2257
2258 package services.hello;
2259
2260 import org.oasisopen.sca.annotation.*;
2261
2262 @Remotable
2263 public interface HelloService {
2264     String hello(String message);
2265 }
2266
2267 package services.hello;
2268
2269 import org.oasisopen.sca.annotation.*;
2270
2271 @Service(HelloService.class)
2272 public class HelloServiceImpl implements HelloService {
2273     public String hello(String message) {
2274         ...
2275     }
2276 }
2277
2278

```

2279 9.19 @Requires

2280 The following Java code defines the **@Requires** annotation:

```

2281 package org.oasisopen.sca.annotation;
2282
2283 import static java.lang.annotation.ElementType.FIELD;
2284

```

```

2285     import static java.lang.annotation.ElementType.METHOD;
2286     import static java.lang.annotation.ElementType.PARAMETER;
2287     import static java.lang.annotation.ElementType.TYPE;
2288     import static java.lang.annotation.RetentionPolicy.RUNTIME;
2289
2290     import java.lang.annotation.Inherited;
2291     import java.lang.annotation.Retention;
2292     import java.lang.annotation.Target;
2293
2294     @Inherited
2295     @Retention(RUNTIME)
2296     @Target({TYPE, METHOD, FIELD, PARAMETER})
2297     public @interface Requires {
2298         /**
2299          * Returns the attached intents.
2300          *
2301          * @return the attached intents
2302          */
2303         String[] value() default "";
2304     }

```

2306 The **@Requires** annotation supports general purpose intents specified as strings. Users can also
 2307 define specific intent annotations using the @Intent annotation.

2308 See the [section "General Intent Annotations"](#) for details and samples.

2309 9.20 @Scope

2310 The following Java code defines the **@Scope** annotation:

```

2311     package org.oasisopen.sca.annotation;
2312
2313     import static java.lang.annotation.ElementType.TYPE;
2314     import static java.lang.annotation.RetentionPolicy.RUNTIME;
2315     import java.lang.annotation.Retention;
2316     import java.lang.annotation.Target;
2317
2318     @Target(TYPE)
2319     @Retention(RUNTIME)
2320     public @interface Scope {
2321
2322         String value() default "STATELESS";
2323     }

```

2324 The @Scope annotation MUST only be used on a service's implementation class. It is an error to
 2325 use this annotation on an interface. [JCA90041]

2326 The @Scope annotation has the following attribute:

- 2327 • **value** – the name of the scope.
- 2328 SCA defines the following scope names, but others can be defined by particular Java-
 2329 based implementation types:
- 2330 STATELESS
- 2331 COMPOSITE
- 2332 For 'STATELESS' implementations, a different implementation instance can be used to
 2333 service each request. Implementation instances can be newly created or be drawn from a
 2334 pool of instances.

2335 The default value is STATELESS.

2336 The following snippet shows a sample for a COMPOSITE scoped service implementation:

```

2337     package services.hello;

```

```

2338
2339 import org.oasisopen.sca.annotation.*;
2340
2341 @Service(HelloService.class)
2342 @Scope("COMPOSITE")
2343 public class HelloServiceImpl implements HelloService {
2344
2345     public String hello(String message) {
2346         ...
2347     }
2348 }
2349

```

9.21 @Service

2350
2351 The following Java code defines the **@Service** annotation:

```

2352 package org.oasisopen.sca.annotation;
2353
2354 import static java.lang.annotation.ElementType.TYPE;
2355 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2356 import java.lang.annotation.Retention;
2357 import java.lang.annotation.Target;
2358
2359 @Target(TYPE)
2360 @Retention(RUNTIME)
2361 public @interface Service {
2362
2363     Class<?>[] interfaces() default {};
2364     Class<?> value() default Void.class;
2365 }
2366

```

2367 The @Service annotation is used on a component implementation class to specify the SCA services
2368 offered by the implementation. **An implementation class need not be declared as implementing all
2369 of the interfaces implied by the services declared in its @Service annotation, but all methods of all
2370 the declared service interfaces MUST be present. [JCA90042]** A class used as the implementation
2371 of a service is not required to have a @Service annotation. If a class has no @Service annotation,
2372 then the rules determining which services are offered and what interfaces those services have are
2373 determined by the specific implementation type.

2374 The @Service annotation has the following attributes:

- 2375 • **interfaces (1..1)** – The value is an array of interface or class objects that are exposed as
2376 services by this component implementation.
- 2377 • **value** – A shortcut for the case when the class provides only a single service interface -
2378 contains a single interface or class object that is exposed as a service by this component
2379 implementation.

2380 **A @Service annotation MUST only have one of the interfaces attribute or value attribute specified.
2381 [JCA90043]**

2382
2383 **A @Service annotation with no attributes MUST be ignored, it is the same as not having the
2384 annotation there at all. [JCA90044]**

2385 The **service names** of the defined services default to the names of the interfaces or class, without
2386 the package name.

Deleted: An
implementation class need
not be declared as
implementing all of the
interfaces implied by the
services declared in its
@Service annotation, but
all methods of all the
declared service interfaces
MUST be present.

Deleted: An
implementation class need
not be declared as
implementing all of the
interfaces implied by the
services declared in its
@Service annotation, but
all methods of all the
declared service interfaces
MUST be present.

2387 **A component implementation MUST NOT have two services with the same Java simple name.**
2388 **[JCA90045]** If a Java implementation needs to realize two services with the same Java simple
2389 name then this can be achieved through subclassing of the interface.

2390 The following snippet shows an implementation of the HelloService marked with the @Service
2391 annotation.

```
2392 package services.hello;  
2393  
2394 import org.oasisopen.sca.annotation.Service;  
2395  
2396 @Service(HelloService.class)  
2397 public class HelloServiceImpl implements HelloService {  
2398     public void hello(String name) {  
2400         System.out.println("Hello " + name);  
2401     }  
2402 }  
2403
```

2404 10 WSDL to Java and Java to WSDL

2405 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL
2406 mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotable Java
2407 interfaces from WSDL portTypes and vice versa.

2408 **For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java**
2409 **interface as if it had a @WebService annotation on the class, even if it doesn't. [JCA100001] The**
2410 **SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for**
2411 **the @javax.jws.OneWay annotation. [JCA100002] For the WSDL-to-Java mapping, the SCA**
2412 **runtime MUST take the generated @WebService annotation to imply that the Java interface is**
2413 **@Remotable. [JCA100003]**

2414 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]
2415 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping
2416 from Java types to XML schema types. [JCA100004] SCA runtimes MAY support the SDO 2.1
2417 mapping from Java types to XML schema types. [JCA100005] Having a choice of binding
2418 technologies is allowed, as noted in the first paragraph of section 5 of the JSR 181 (version 2)
2419 specification, which is referenced by the JAX-WS specification.

2420 The JAX-WS mappings are applied with the following restrictions:

- 2421 • No support for holders

2422
2423 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous
2424 model is used.

2425 10.1 JAX-WS Client Asynchronous API for a Synchronous Service

2426 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
2427 application with a means of invoking that service asynchronously, so that the client can invoke a service
2428 operation and proceed to do other work without waiting for the service operation to complete its
2429 processing. The client application can retrieve the results of the service either through a polling
2430 mechanism or via a callback method which is invoked when the operation completes.

2431 **For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the**
2432 **additional client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100006] For**
2433 **SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional**
2434 **client-side asynchronous polling and callback methods defined by JAX-WS. [JCA100007] If the additional**
2435 **client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface**
2436 **which declares the type of a reference in the implementation, SCA Runtimes MUST NOT include these**
2437 **methods in the SCA reference interface in the component type of the implementation. [JCA100008]**
2438

2439 The additional client-side asynchronous polling and callback methods defined by JAX-WS are recognized
2440 in a Java interface as follows:

2441 For each method M in the interface, if another method P in the interface has

- 2442 a. a method name that is M's method name with the characters "Async" appended, and
- 2443 b. the same parameter signature as M, and
- 2444 c. a return type of Response<R> where R is the return type of M

2445 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

2446 For each method M in the interface, if another method C in the interface has

- 2447 a. a method name that is M's method name with the characters "Async" appended, and
- 2448 b. a parameter signature that is M's parameter signature with an additional final parameter of type
2449 AsyncHandler<R> where R is the return type of M, and

Deleted: For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.

Deleted: For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.

Deleted: The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.

Deleted: The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.

Deleted: For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.

Deleted: For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.

2450 c. a return type of Future<?>
2451 then C is a JAX-WS callback method that isn't part of the SCA interface contract.
2452 As an example, an interface can be defined in WSDL as follows:

```
2453 <!-- WSDL extract -->  
2454 <message name="getPrice">  
2455   <part name="ticker" type="xsd:string"/>  
2456 </message>  
2457  
2458 <message name="getPriceResponse">  
2459   <part name="price" type="xsd:float"/>  
2460 </message>  
2461  
2462 <portType name="StockQuote">  
2463   <operation name="getPrice">  
2464     <input message="tns:getPrice"/>  
2465     <output message="tns:getPriceResponse"/>  
2466   </operation>  
2467 </portType>
```

2468
2469 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2470 // asynchronous mapping  
2471 @WebService  
2472 public interface StockQuote {  
2473   float getPrice(String ticker);  
2474   Response<Float> getPriceAsync(String ticker);  
2475   Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);  
2476 }
```

2477
2478 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2479 // synchronous mapping  
2480 @WebService  
2481 public interface StockQuote {  
2482   float getPrice(String ticker);  
2483 }
```

2484
2485 **SCA runtimes MUST support the use of the JAX-WS client asynchronous model.** [JCA100009] In
2486 the above example, if the client implementation uses the asynchronous form of the interface, the
2487 two additional getPriceAsync() methods can be used for polling and callbacks as defined by the
2488 JAX-WS specification.

2489

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

```
2490 <?xml version="1.0" encoding="UTF-8"?>
2491 <!-- (c) Copyright SCA Collaboration 2006 -->
2492 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2493   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2494   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2495   elementFormDefault="qualified">
2496
2497   <include schemaLocation="sca-core.xsd"/>
2498
2499   <element name="interface.java" type="sca:JavaInterface"
2500     substitutionGroup="sca:interface"/>
2501   <complexType name="JavaInterface">
2502     <complexContent>
2503       <extension base="sca:Interface">
2504         <sequence>
2505           <any namespace="##other" processContents="lax"
2506             minOccurs="0" maxOccurs="unbounded"/>
2507         </sequence>
2508         <attribute name="interface" type="NCName" use="required"/>
2509         <attribute name="callbackInterface" type="NCName"
2510           use="optional"/>
2511         <anyAttribute namespace="##any" processContents="lax"/>
2512       </extension>
2513     </complexContent>
2514   </complexType>
2515 </schema>
2516
```

2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566

B. Java Classes and Interfaces

B.1 SCAClient Classes and Interfaces

B.1.1 SCAClient Interface

```
/*  
 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.  
 * OASIS trademark, IPR and other policies apply.  
 */  
package org.oasisopen.sca.client;  
  
import java.net.URI;  
  
import org.oasisopen.sca.NoSuchDomainException;  
import org.oasisopen.sca.NoSuchServiceException;  
  
/**  
 * Client side interface that can be used to lookup SCA Services within  
 * a SCA domain.  
 * <p>  
 * The SCAClientFactory is used to obtain an implementation instance of  
 * the SCAClient.  
 *  
 * @see SCAClientFactory  
 * @author OASIS Open  
 */  
public interface SCAClient {  
  
    /**  
     * Returns a reference proxy that implements the business interface <T>  
     * of a service in a domain  
     *  
     * @param serviceURI the relative URI of the target service. Takes the  
     * form componentName/serviceName.  
     * Can also take the extended form componentName/serviceName/bindingName  
     * to use a specific binding of the target service  
     *  
     * @param domainURI the URI of an SCA Domain.  
     * @param interfaze The business interface class of the service in the  
     * domain  
     * @param <T> The business interface class of the service in the domain  
     *  
     * @return a proxy to the target service, in the specified SCA Domain  
     * that implements the business interface <B>.  
     * @throws NoSuchServiceException Service requested was not found  
     * @throws NoSuchDomainException Domain requested was not found  
     */  
    <T> T getService(Class<T> interfaze, String serviceURI, URI domainURI)  
        throws NoSuchServiceException, NoSuchDomainException;  
}
```

Formatted: French France

Formatted: French France

2567
2568

Deleted: D

2569 **B.1.2 SCAClientFactory Class**

2570 SCA provides an abstract reference implementation of the SCAClientFactory class. Vendors can provide
2571 a subclass of this class which provides the capability of creating objects that implement the SCAClient
2572 interface suitable for linking to services in their SCA runtime.

2573

```
2574 /*  
2575 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.  
2576 * OASIS trademark, IPR and other policies apply.  
2577 */  
2578 package org.oasisopen.sca.client;  
2579  
2580 import java.util.Properties;  
2581  
2582 import org.oasisopen.sca.client.impl.SCAClientFactoryFinderImpl;  
2583  
2584 /**  
2585 * The SCAClientFactory can be used by non-SCA managed code to  
2586 * lookup services that exist in a SCADomain.  
2587 *  
2588 * @see SCAClientFactoryFinderImpl  
2589 * @see SCAClient  
2590 *  
2591 * @author OASIS Open  
2592 */  
2593  
2594 public abstract class SCAClientFactory {  
2595  
2596     /**  
2597     * The SCAClientFactoryFinder.  
2598     * A Vendor may use reflection to inject a their own version of an  
2599     * SCAClientFactoryFinder instance that will be used in the  
2600     * newInstance() methods rather than using the default one provided by  
2601     * SCAClientFactoryFinderImpl  
2602     */  
2603     protected static SCAClientFactoryFinderImpl defaultFactoryFinder;  
2604  
2605     /**  
2606     * Creates a new instance of the SCAClient that can be  
2607     * used to lookup SCA Services.  
2608     *  
2609     * @return A new SCAClient  
2610     */  
2611     public static SCAClient newInstance() {  
2612         return newInstance(null, null);  
2613     }  
2614  
2615     /**  
2616     * Creates a new instance of the SCAClient that can be  
2617     * used to lookup SCA Services.  
2618     *  
2619     * @param properties Properties that may be used when  
2620     * creating a new instance of the SCAClient  
2621     * @return A new SCAClient instance
```

```

2622     */
2623     public static SCAClient newInstance(Properties properties) {
2624         return newInstance(properties, null);
2625     }
2626
2627     /**
2628      * Creates a new instance of the SCAClient that can be
2629      * used to lookup SCA Services.
2630      *
2631      * @param classLoader    ClassLoader that may be used when
2632      * creating a new instance of the SCAClient
2633      * @return A new SCAClient instance
2634      */
2635     public static SCAClient newInstance(ClassLoader classLoader) {
2636         return newInstance(null, classLoader);
2637     }
2638
2639     /**
2640      * Creates a new instance of the SCAClient that can be
2641      * used to lookup SCA Services.
2642      *
2643      * @param properties    Properties that may be used when
2644      * creating a new instance of the SCAClient
2645      * @param classLoader    ClassLoader that may be used when
2646      * creating a new instance of the SCAClient
2647      * @return A new SCAClient instance
2648      */
2649     public static SCAClient newInstance(Properties properties,
2650                                        ClassLoader classLoader) {
2651         final SCAClientFactoryFinderImpl finder =
2652             defaultFactoryFinder != null ? defaultFactoryFinder :
2653             new SCAClientFactoryFinderImpl();
2654         final SCAClientFactory factory
2655             = finder.find(properties, classLoader);
2656         return factory.createSCAClient();
2657     }
2658
2659     /**
2660      * This method is invoked to create a new SCAClient instance.
2661      *
2662      * @return A new SCAClient instance
2663      */
2664     protected abstract SCAClient createSCAClient();
2665     }

```

Deleted: protected

2668 **B.1.3 SCAClientFactoryFinder interface**

Formatted: Bullets and Numbering

2669 The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory
2670 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can
2671 create alternative implementations of this interface that use different class loading or lookup mechanisms.

Formatted: Normal

```

2672     /**
2673      * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
2674      * OASIS trademark, IPR and other policies apply.
2675      */
2676

```

```

2677 package org.oasisopen.sca.client;
2678
2679 import java.util.Properties;
2680
2681 /* A Service Provider Interface representing a SCAClientFactory finder.
2682 * SCA provides a default reference implementation of this interface.
2683 * SCA runtime vendors can create alternative implementations of this
2684 * interface that use different class loading or lookup mechanisms.
2685 */
2686 public interface SCAClientFactoryFinder {
2687
2688     SCAClientFactory find(Properties properties,
2689                            ClassLoader classLoader);
2690
2691 }

```

Formatted: Normal

Formatted: Bullets and Numbering

2691 **B.1.4 SCAClientFactoryFinderImpl class**

2692 This class provides a default implementation for finding a provider's SCAClientFactory implementation
2693 class. It is used if the provider does not inject its SCAClientFactoryFinder implementation class into the
2694 base SCAClientFactory class.

2695 It discovers a provider's SCAClientFactory implementation by referring to the following information in this
2696 order:

- 2697 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the
2698 newInstance() method call if specified
- 2699 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 2700 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

```

2701 /*
2702 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
2703 * OASIS trademark, IPR and other policies apply.
2704 */

```

Deleted: Since this is a reference implementation, vendors are free to replace the SCAClientFactoryFinder class with an alternative implementation that provides the lookup mechanisms required for their SCA Runtime.¶¶

```

2705 package org.oasisopen.sca.client.impl;
2706
2707 import org.oasisopen.sca.client.SCAClientFactoryFinder;
2708
2709 import java.io.BufferedReader;
2710 import java.io.Closeable;
2711 import java.io.IOException;
2712 import java.io.InputStream;
2713 import java.io.InputStreamReader;
2714 import java.net.URL;
2715 import java.util.Properties;

```

Formatted: French France

```

2716
2717 import org.oasisopen.sca.SCARuntimeException;
2718 import org.oasisopen.sca.client.SCAClientFactory;
2719
2720 /**
2721 * This is a default implementation of an SCAClientFactoryFinder which is
2722 used
2723 * to find an implementation of the SCAClientFactory interface.
2724 *
2725 * @see SCAClientFactoryFinder
2726 * @see SCAClientFactory
2727 *
2728 * @author OASIS Open
2729 */
2730 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {

```

```

2731
2732 /**
2733  * The name of the System Property used to determine the SPI
2734  * implementation to use for the SCAClientFactory.
2735  */
2736 private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
2737 SCAClientFactory.class.getName();
2738
2739 /**
2740  * The name of the file loaded from the ClassPath to determine
2741  * the SPI implementation to use for the SCAClientFactory.
2742  */
2743 private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
2744  = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
2745
2746 /**
2747  * Public Constructor
2748  */
2749 public SCAClientFactoryFinderImpl() {
2750  }
2751
2752 /**
2753  * Creates an instance of the SCAClientFactorySPI implementation.
2754  * This discovers the SCAClientFactorySPI Implementation and instantiates
2755  * the provider's implementation.
2756  *
2757  * @param properties Properties that may be used when creating a new
2758  * instance of the SCAClient
2759  * @param classLoader ClassLoader that may be used when creating a new
2760  * instance of the SCAClient
2761  * @return new instance of the SCAClientFactorySPI
2762  * @throws SCARuntimeException Failed to create SCAClientFactorySPI
2763  * Implementation.
2764  */
2765 public SCAClientFactory find(Properties properties,
2766                              ClassLoader classLoader) {
2767     if (classLoader == null) {
2768         classLoader = getThreadContextClassLoader();
2769     }
2770     final String factoryImplClassName =
2771         discoverProviderFactoryImplClass(properties, classLoader);
2772     final Class<? extends SCAClientFactory> factoryImplClass
2773         = loadProviderFactoryClass(factoryImplClassName,
2774 classLoader);
2775     final SCAClientFactory factory =
2776         instantiateSCAClientFactoryClass(factoryImplClass);
2777     return factory;
2778  }
2779
2780 /**
2781  * Gets the Context ClassLoader for the current Thread.
2782  *
2783  * @return The Context ClassLoader for the current Thread.
2784  */
2785 private static ClassLoader getThreadContextClassLoader() {
2786     final ClassLoader threadClassLoader =
2787         Thread.currentThread().getContextClassLoader();
2788     return threadClassLoader;

```

```

2789     }
2790
2791     /**
2792     * Attempts to discover the class name for the SCAClientFactorySPI
2793     * implementation from the specified Properties, the System Properties
2794     * or the specified ClassLoader.
2795     *
2796     * @return The class name of the SCAClientFactorySPI implementation
2797     * @throws SCARuntimeException Failed to find implementation for
2798     * SCAClientFactorySPI.
2799     */
2800     private static String
2801     discoverProviderFactoryImplClass(Properties properties,
2802     ClassLoader classLoader)
2803     throws SCARuntimeException {
2804     String providerClassName =
2805     checkPropertiesForSPIClassName(properties);
2806     if (providerClassName != null) {
2807     return providerClassName;
2808     }
2809
2810     providerClassName =
2811     checkPropertiesForSPIClassName(System.getProperties());
2812     if (providerClassName != null) {
2813     return providerClassName;
2814     }
2815
2816     providerClassName = checkMETAINFOServicesForSIPClassName(classLoader);
2817     if (providerClassName == null) {
2818     throw new SCARuntimeException(
2819     "Failed to find implementation for SCAClientFactory");
2820     }
2821
2822     return providerClassName;
2823     }
2824
2825     /**
2826     * Attempts to find the class name for the SCAClientFactorySPI
2827     * implementation from the specified Properties.
2828     *
2829     * @return The class name for the SCAClientFactorySPI implementation
2830     * or <code>null</code> if not found.
2831     */
2832     private static String
2833     checkPropertiesForSPIClassName(Properties properties) {
2834     if (properties == null) {
2835     return null;
2836     }
2837
2838     final String providerClassName =
2839     properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
2840     if (providerClassName != null && providerClassName.length() > 0) {
2841     return providerClassName;
2842     }
2843
2844     return null;
2845     }
2846

```

```

2847     /**
2848     * Attempts to find the class name for the SCAClientFactorySPI
2849     * implementation from the META-INF/services directory
2850     *
2851     * @return The class name for the SCAClientFactorySPI implementation or
2852     * <code>null</code> if not found.
2853     */
2854     private static String checkMETAINFServicesForSIPClassName(ClassLoader cl)
2855     {
2856         final URL url =
2857             cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
2858         if (url == null) {
2859             return null;
2860         }
2861         InputStream in = null;
2862         try {
2863             in = url.openStream();
2864             BufferedReader reader = null;
2865             try {
2866                 reader =
2867                     new BufferedReader(new InputStreamReader(in, "UTF-8"));
2868                 String line;
2869                 while ((line = readNextLine(reader)) != null) {
2870                     if (!line.startsWith("#") && line.length() > 0) {
2871                         return line;
2872                     }
2873                 }
2874             }
2875             return null;
2876         } finally {
2877             closeStream(reader);
2878         }
2879         catch (IOException ex) {
2880             throw new SCARuntimeException(
2881                 "Failed to discover SCAClientFactory provider", ex);
2882         } finally {
2883             closeStream(in);
2884         }
2885     }
2886     }
2887     }
2888     /**
2889     * Reads the next line from the reader and returns the trimmed version
2890     * of that line
2891     *
2892     * @param reader The reader from which to read the next line
2893     * @return The trimmed next line or <code>null</code> if the end of the
2894     * stream has been reached
2895     * @throws IOException I/O error occurred while reading from Reader
2896     */
2897     private static String readNextLine(BufferedReader reader)
2898         throws IOException {
2899         String line = reader.readLine();
2900         if (line != null) {
2901             line = line.trim();
2902         }
2903     }
2904

```

```

2905     return line;
2906 }
2907
2908 /**
2909  * Loads the specified SCAClientFactory Implementation class.
2910  *
2911  * @param factoryImplClassName The name of the SCAClientFactory
2912  * Implementation class to load
2913  * @return The specified SCAClientFactory Implementation class
2914  * @throws SCARuntimeException Failed to load the SCAClientFactory
2915  * Implementation class
2916  */
2917 private static Class<? extends SCAClientFactory>
2918 loadProviderFactoryClass(String factoryImplClassName,
2919 ClassLoader classLoader)
2920 throws SCARuntimeException {
2921
2922     try {
2923         final Class<?> providerClass =
2924             classLoader.loadClass(factoryImplClassName);
2925         final Class<? extends SCAClientFactory> providerFactoryClass =
2926             providerClass.asSubclass(SCAClientFactory.class);
2927         return providerFactoryClass;
2928     } catch (ClassNotFoundException ex) {
2929         throw new SCARuntimeException(
2930             "Failed to load SCAClientFactory implementation class "
2931             + factoryImplClassName, ex);
2932     } catch (ClassCastException ex) {
2933         throw new SCARuntimeException(
2934             "Loaded SCAClientFactory implementation class "
2935             + factoryImplClassName
2936             + " is not a subclass of "
2937             + SCAClientFactory.class.getName() , ex);
2938     }
2939 }
2940
2941 /**
2942  * Instantiate an instance of the specified SCAClientFactorySPI
2943  * Implementation class.
2944  *
2945  * @param factoryImplClass The SCAClientFactorySPI Implementation
2946  * class to instantiate.
2947  * @return An instance of the SCAClientFactorySPI Implementation class
2948  * @throws SCARuntimeException Failed to instantiate the specified
2949  * specified SCAClientFactorySPI Implementation class
2950  */
2951 private static SCAClientFactory
2952 instantiateSCAClientFactoryClass(
2953 Class<? extends SCAClientFactory> factoryImplClass)
2954 throws SCARuntimeException {
2955
2956     try {
2957         final SCAClientFactory provider = factoryImplClass.newInstance();
2958         return provider;
2959     } catch (Throwable ex) {
2960         throw new SCARuntimeException(
2961             "Failed to instantiate SCAClientFactory implementation class "
2962             + factoryImplClass, ex);

```

```

2963     }
2964 }
2965
2966 /**
2967  * Utility method for closing Closeable Object.
2968  *
2969  * @param closeable The Object to close.
2970  */
2971 private static void closeStream(Closeable closeable) {
2972     if (closeable != null) {
2973         try{
2974             closeable.close();
2975         } catch (IOException ex) {
2976             throw new SCARuntimeException("Failed to close stream", ex);
2977         }
2978     }
2979 }
2980 }
2981
2982

```

2983 **B.1.5 SCAClient Classes and Interfaces - what does a vendor need to do?**

Formatted: Bullets and
Numbering

2984 The SCAClient classes and interfaces are designed so that vendors can provide their own
 2985 implementation suited to the needs of their SCA runtime. This section describes the tasks that a vendor
 2986 needs to consider in relation to the SCAClient classes and interfaces.

- 2987 • Implement their SCAClientFactory and SCAClient implementation classes

2988 Vendors need to provide an implementation of SCAClient that is capable of looking up Services in
 2989 their SCA Runtime.

2992 Vendors need to subclass SCAClientFactory and implement the createSCAClient() method so
 2993 that it creates an instance of their SCAClient implementation.

- 2996 • Configure the Vendor Implementation classes so they are used
 2997 Vendors have several options:

2999 Option 1: Set System Property to point to the Vendor's implementation

3001 Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their
 3002 implementation class and use the reference implementation of SCAClientFactoryFinder

3004 Option 2: Provide a META-INF/services file

3006 Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points
 3007 to their implementation class and use the reference implementation of SCAClientFactoryFinder

3009 Option 3: Inject a vendor implementation instance into SCAClientFactory

3011 Vendors inject an instance into the defaultFactory field of SCAClientFactory. The
 3012 SCAClientFactoryFinder is not used in this scenario.

Deleted: .
 Option 4: Provide a Vendor
 specific implementation of
 SCAClientFactoryFinder .
 .
 Vendors write a new
 implementation of
 SCAClientFactoryFinder and
 replace the reference
 implementation that is provided
 by SCA.

3014

C. Conformance Items

3015 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
3016 specification.

3017

Conformance ID	Description
[JCA20001]	Remotable Services MUST NOT make use of method overloading .
[JCA20002]	the SCA runtime MUST ensure that a stateless scoped implementation instance object is only ever dispatched on one thread at any one time.
[JCA20003]	within the SCA lifecycle of a stateless scoped implementation instance, the SCA runtime MUST only make a single invocation of one business method.
[JCA20004]	For a composite scope implementation instance, the SCA runtime MUST ensure that all service requests are dispatched to the same implementation instance for the lifetime of the containing composite.
[JCA20005]	When the implementation class is marked for eager initialization, the SCA runtime MUST create a composite scoped instance when its containing component is started.
[JCA20006]	If a method of an implementation class is marked with the @Init annotation, the SCA runtime MUST call that method when the implementation instance is created.
[JCA20007]	the SCA runtime MAY run multiple threads in a single composite scoped implementation instance object and the SCA runtime MUST NOT perform any synchronization.
[JCA30001]	The value of the @interface attribute MUST be the fully qualified name of the Java interface class
[JCA30002]	The value of the @callbackInterface attribute MUST be the fully qualified name of a Java interface used for callbacks
[JCA30003]	if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.
[JCA30004]	The interface.java element MUST conform to the schema defined in the sca-interface-java.xsd schema.
[JCA70001]	SCA identifies annotations that correspond to intents by providing an @Intent annotation which MUST be used in the definition of a specific intent annotation.
[JCA80001]	ComponentContext.getService method MUST throw an IllegalArgumentException if the reference identified by the referenceName parameter has multiplicity of 0..n or 1..n.

- [JCA80002] The ComponentContext.getRequestContext method MUST return non-null when invoked during the execution of a Java business method for a service operation or a callback operation, on the same thread that the SCA runtime provided, and MUST return null in all other cases.
- [JCA80003] When invoked during the execution of a service operation, the getServiceReference method MUST return a ServiceReference that represents the service that was invoked. When invoked during the execution of a callback operation, the getServiceReference method MUST return a ServiceReference that represents the callback that was invoked.
- [JCA90001] An SCA runtime MUST verify the proper use of all SCA annotations and if an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the invalid implementation code.
- [JCA90002] SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an SCA annotation on a static method or a static field of an implementation class and the SCA runtime MUST NOT instantiate such an implementation class.
- [JCA90003] If a constructor of an implementation class is annotated with @Constructor and the constructor has parameters, each of these parameters MUST have either a @Property annotation or a @Reference annotation.
- [JCA90004] A method annotated with @Destroy MAY have any access modifier and MUST have a void return type and no arguments.
- [JCA90005] If there is a method annotated with @Destroy that matches the criteria for the annotation, the SCA runtime MUST call the annotated method when the scope defined for the implementation class ends.
- [JCA90007] When marked for eager initialization with an @EagerInit annotation, the composite scoped instance MUST be created when its containing component is started.
- [JCA90008] A method marked with the @Init annotation MAY have any access modifier and MUST have a void return type and no arguments.
- [JCA90009] If there is a method annotated with @Init that matches the criteria for the annotation, the SCA runtime MUST call the annotated method after all property and reference injection is complete.
- [JCA90011] the @Property annotation MUST NOT be used on a class field that is declared as final.
- [JCA90012] the @Property annotation MUST be used in order to inject a property onto a non-public field.
- [JCA90013] For a @Property annotation applied to a constructor parameter, there is no default value for the name attribute and the name attribute MUST be present.

- [JCA90014] For a @Property annotation applied to a constructor parameter, the required attribute MUST have the value true.
- [JCA90015] The @Qualifier annotation MUST be used in a specific intent annotation definition where the intent has qualifiers.
- [JCA90016] The @Reference annotation MUST NOT be used on a class field that is declared as final.
- [JCA90017] the @Reference annotation MUST be used in order to inject a reference onto a non-public field.
- [JCA90018] For a @Reference annotation applied to a constructor parameter, there is no default for the name attribute and the name attribute MUST be present.
- [JCA90019] For a @Reference annotation applied to a constructor parameter, the required attribute MUST have the value true.
- [JCA90020] If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation required attribute is true.
- [JCA90021] If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true.
- [JCA90022] An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by the SCA runtime as null (either via injection or via API call).
- [JCA90023] An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection (either via injection or via API call).
- [JCA90024] References MAY be reinjected by an SCA runtime after the initial creation of a component if the reference target changes due to a change in wiring that has occurred since the component was initialized.
- [JCA90025] In order for reinjection to occur, the following MUST be true:
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.
- [JCA90026] If a reference target changes and the reference is not reinjected, the reference MUST continue to work as if the reference target was not changed.
- [JCA90027] If an operation is called on a reference where the target of that reference has been undeployed, the SCA runtime SHOULD throw an InvalidServiceException.

[JCA90028]

If an operation is called on a reference where the target of the reference has become unavailable for some reason, the SCA runtime SHOULD throw a ServiceUnavailableException.

[JCA90029]

If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

[JCA90030]

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

[JCA90031]

If the target of a ServiceReference has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an operation is invoked on the ServiceReference.

Deleted: [JCA90031]

Deleted: [JCA90031]

[JCA90032]

If the target of a ServiceReference has become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an operation is invoked on the ServiceReference.

Deleted: [JCA90032]

Deleted: [JCA90032]

[JCA90033]

If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

Deleted: [JCA90033]

Deleted: [JCA90033]

[JCA90034]

A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain. This applies whether or not reinjection has taken place.

Deleted: [JCA90034]

Deleted: [JCA90034]

[JCA90035]

If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.

Deleted: [JCA90035]

Deleted: [JCA90035]

[JCA90036]

If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.

Deleted: [JCA90036]

Deleted: [JCA90036]

[JCA90037]

in the cases where reference reinjection is not allowed, the array or Collection for a reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes occur to the reference wiring or to the targets of the wiring.

Deleted: [JCA90037]

Deleted: [JCA90037]

[JCA90038]

In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.

Deleted: [JCA90038]

Deleted: [JCA90038]

[JCA90039]

A reinjected array or Collection for a reference MUST NOT be the same array or Collection object previously injected to the component.

Deleted: [JCA90039]

Deleted: [JCA90039]

- [JCA90040] The @Remotable annotation is used to specify a Java service interface as remotable. A remotable service can be published externally as a service and MUST be translatable into a WSDL portType.
- [JCA90041] The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface.
- [JCA90042] An implementation class need not be declared as implementing all of the interfaces implied by the services declared in its @Service annotation, but all methods of all the declared service interfaces MUST be present.
- [JCA90043] A @Service annotation MUST only have one of the interfaces attribute or value attribute specified.
- [JCA90044] A @Service annotation with no attributes MUST be ignored, it is the same as not having the annotation there at all.
- [JCA90045] A component implementation MUST NOT have two services with the same Java simple name.
- [JCA90046] When used to annotate a method or a field of an implementation class for injection of a callback object, the @Callback annotation MUST NOT specify any attributes.
- [JCA90047] For a @Property annotation, if the the type of the Java class field or the type of the input parameter of the setter method or constructor is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <property/> element with a @many attribute set to true, otherwise @many MUST be set to false.
- [JCA100001] For the purposes of the Java-to-WSDL mapping algorithm, the SCA runtime MUST treat a Java interface as if it had a @WebService annotation on the class, even if it doesn't.
- [JCA100002] The SCA runtime MUST treat an @org.oasisopen.sca.annotation.OneWay annotation as a synonym for the @javax.jws.OneWay annotation.
- [JCA100003] For the WSDL-to-Java mapping, the SCA runtime MUST take the generated @WebService annotation to imply that the Java interface is @Remotable.
- [JCA100004] SCA runtimes MUST support the JAXB 2.1 mapping from Java types to XML schema types.
- [JCA100005] SCA runtimes MAY support the SDO 2.1 mapping from Java types to XML schema types.
- [JCA100006] For SCA service interfaces defined using interface.java, the Java interface MUST NOT contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.
- [JCA100007] For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional client-side asynchronous polling and callback methods defined by JAX-WS.

Deleted: [JCA90045]

Deleted: [JCA90045]

[JCA100008]

If the additional client-side asynchronous polling and callback methods defined by JAX-WS are present in the interface which declares the type of a reference in the implementation, SCA Runtimes MUST NOT include these methods in the SCA reference interface in the component type of the implementation.

[JCA100009]

SCA runtimes MUST support the use of the JAX-WS client asynchronous model.

3018

3019 **D. Acknowledgements**

3020 The following individuals have participated in the creation of this specification and are gratefully
3021 acknowledged:

3022 **Participants:**

3023 [Participant Name, Affiliation | Individual Member]

3024 [Participant Name, Affiliation | Individual Member]

3025

3027

F. Revision History

3028 [optional; should not be included in OASIS Standards]

3029

Revision	Date	Editor	Changes Made
1	2007-09-26	Anish Karmarkar	Applied the OASIS template + related changes to the Submission
2	2008-02-28	Anish Karmarkar	Applied resolution of issues: 4, 11, and 26
3	2008-04-17	Mike Edwards	Ed changes
4	2008-05-27	Anish Karmarkar David Booz Mark Combellack	Added InvalidServiceException in Section 7 Various editorial updates
WD04	2008-08-15	Anish Karmarkar	* Applied resolution of issue 9 (it was applied before, not sure by whom, but it was applied incorrectly) * Applied resolution of issue 12, 22, 23, 29, 31, 35, 36, 37, 44, 45 * Note that issue 33 was applied, but not noted, in a previous version * Replaced the osoa.org NS with the oasis-open.org NS
WD05	2008-10-03	Anish Karmarkar	* Fixed the resolution of issue 37 but re-adding the sentence: "However, the @... annotation must be used in order to inject a property onto a non-public field. -- in the @Property and @Reference section * resolution of issue 9 was applied incorrectly. Fixed that -- removed the requirement for throwing an exception on ComponentContext.getServiceReferences() when multiplicity of references > 1 * minor ed changes
cd01-rev1	2008-12-11	Anish Karmarkar	* Fixed reference style to [RFC2119] instead of [1]. * Applied resolutions of issues 20, 21, 41, 42, 43, 47, 48, 49.
cd01-rev2	2008-12-12	Anish Karmarkar	* Applied resolutions of issues 61, 71, 72, 73, 79, 81, 82, 84, 112
cd01-rev3	2008-12-16	David Booz	* Applied resolution of issues 56, 75, 111
cd01-rev4	2009-01-18	Anish Karmarkar	* Applied resolutions of issues 28, 52, 94, 96, 99, 101
cd02	2009-01-26	Mike Edwards	Minor editorial cleanup. All changes accepted.

			All comments removed.
cd02-rev1	2009-02-03	Mike Edwards	Issues 25+95 Issue 120
cd02-rev2	2009-02-08	Mike Edwards	Merge annotation definitions contained in section 10 into section 8 Move remaining parts of section 10 to section 7. Accept all changes.
cd02-rev3	2009-03-16	Mike Edwards	RFC2119 work and formal marking of all normative statements - all sections. Completion of Appendix B (list of all normative statements) Accept all changes

3030

Page 54: [1] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
If the type of a reference is not an array or any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference annotation required attribute is false and with @multiplicity=1..1 if the @Reference annotation required attribute is true.		
Page 54: [2] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true.		
Page 54: [3] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
If the type of a reference is defined as an array or as any type that extends or implements java.util.Collection, then the SCA runtime MUST introspect the component type of the implementation with a <reference/> element with @multiplicity=0..n if the @Reference annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation required attribute is true.		
Page 54: [4] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection		
Page 54: [5] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
An unwired reference with a multiplicity of 0..n MUST be presented to the implementation code by the SCA runtime as an empty array or empty collection		
Page 55: [6] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
In order for reinjection to occur, the following MUST be true:		
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.		
Page 55: [7] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
In order for reinjection to occur, the following MUST be true:		
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.		
Page 55: [8] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.		
Page 55: [9] Deleted	Mike Edwards	6/22/2009 4:09:00 PM
If the target service of the reference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.		
Page 55: [10] Deleted	Mike Edwards	6/22/2009 4:09:00 PM

If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

Page 55: [11] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of a ServiceReference is changed, the reference MAY continue to work, depending on the runtime and the type of change that was made.

Page 55: [12] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain. This applies whether or not reinjection has taken place.

Page 55: [13] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

A reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() MUST correspond to the current configuration of the domain. This applies whether or not reinjection has taken place.

Page 55: [14] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.

Page 55: [15] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a reference to the undeployed or unavailable service, and attempts to call business methods SHOULD throw an InvalidServiceException or a ServiceUnavailableException.

Page 55: [16] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.

Page 55: [17] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

If the target service of a reference or ServiceReference accessed through the component context by calling getService() or getServiceReference() has changed, the returned value SHOULD be a reference to the changed service.

Page 55: [18] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.

Page 55: [19] Deleted **Mike Edwards** **6/22/2009 4:09:00 PM**

In cases where the contents of a reference array or collection change when the wiring changes or the targets change, then for references that use setter injection, the setter method MUST be called by the SCA runtime for any change to the contents.