



# Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02, Revision 03+Issue1 rev 6

23 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 Combella,	Avaya

## Editor(s):

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

## Related work:

This specification replaces or supersedes:

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

This specification is related to:

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

Deleted: March

**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.....	37	Deleted: 37
9 Java Annotations.....	38	Deleted: 37
9.1 @AllowsPassByReference.....	38	Deleted: 37
9.2 @Authentication.....	38	Deleted: 38
9.3 @Callback.....	39	Deleted: 38
9.4 @ComponentName.....	40	Deleted: 39
9.5 @Confidentiality.....	41	Deleted: 40
9.6 @Constructor.....	42	Deleted: 41
9.7 @Context.....	42	Deleted: 41
9.8 @Destroy.....	43	Deleted: 42
9.9 @EagerInit.....	43	Deleted: 42
9.10 @Init.....	44	Deleted: 43
9.11 @Integrity.....	44	Deleted: 43
9.12 @Intent.....	45	Deleted: 44
9.13 @OneWay.....	46	Deleted: 45
9.14 @PolicySets.....	46	Deleted: 45
9.15 @Property.....	47	Deleted: 46
9.16 @Qualifier.....	48	Deleted: 46
9.17 @Reference.....	49	Deleted: 47
9.17.1 Reinjection.....	51	Deleted: 48
9.18 @Remotable.....	53	Deleted: 49
9.19 @Requires.....	54	Deleted: 50
9.20 @Scope.....	55	Deleted: 51
9.21 @Service.....	56	Deleted: 52
10 WSDL to Java and Java to WSDL.....	58	Deleted: 53
10.1 JAX-WS Client Asynchronous API for a Synchronous Service.....	58	Deleted: 54
A. XML Schema: sca-interface-java.xsd.....	60	Deleted: 55
B. Java Classes and Interfaces.....	61	Deleted: 57
B.1 SCAClient Classes and Interfaces.....	61	Deleted: 57
B.1.1 SCAClient Interface.....	61	Deleted: 57
B.1.2 SCAClientFactory Class.....	62	Deleted: 59
B.1.3 SCAFactoryFinder class.....	63	Deleted: 59
B.1.4 SCAClient Classes and Interfaces - what does a vendor need to do?.....	69	Deleted: 60
C. Conformance Items.....	71	Deleted: 60
D. Acknowledgements.....	77	Deleted: 61
		Deleted: 62
		Deleted: 63
		Deleted: 67
		Deleted: 69
		Deleted: 75
		Deleted: March

E. Non-Normative Text ..... 78  
F. Revision History..... 79

**Deleted: 76**

**Deleted: 77**

**Deleted: March**

# 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> , <a href="http://www.ietf.org/rfc/rfc2119.txt">http://www.ietf.org/rfc/rfc2119.txt</a> , IETF RFC 2119, March 1997.        |
| [ASSEMBLY] | SCA Assembly Specification, <a href="http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf">http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf</a> |
| [SDO]      | SDO 2.1 Specification, <a href="http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf">http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf</a>              |
| [JAX-B]    | JAXB 2.1 Specification, <a href="http://www.jcp.org/en/jsr/detail?id=222">http://www.jcp.org/en/jsr/detail?id=222</a>   |
| [WSDL]     | WSDL Specification,<br>WSDL 1.1: <a href="http://www.w3.org/TR/wsdl">http://www.w3.org/TR/wsdl</a> ,<br>WSDL 2.0: <a href="http://www.w3.org/TR/wsdl20/">http://www.w3.org/TR/wsdl20/</a>               |
| [POLICY]   | SCA Policy Framework, <a href="http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf">http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf</a>             |

Deleted: March

- 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.

Deleted: March

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  
130     @Destroy  
131     public void stop() {  
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.

Deleted: March

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.

**Deleted:** March

## 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

Deleted: March

## 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".

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

Deleted: March

---

## 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

Deleted: March



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 }
362
```

```

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.

Deleted: March

## 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

Deleted: March

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.

**Deleted:** March

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

```

702

Deleted: March



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!");
```

Deleted: March

```
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 =

```

Deleted: March

```

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<B> 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<B> 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.

Deleted: March

- 981 • **getServices(Class<B> businessInterface, String referenceName)** – Returns a list of  
982 typed service proxies for a business interface type and a reference name.
- 983 • **getServiceReferences(Class<B> businessInterface, String referenceName)** –Returns a  
984 list typed service references for a business interface type and a reference name.
- 985 • **createSelfReference(Class<B> businessInterface)** – Returns a ServiceReference that can  
986 be used to invoke this component over the designated service.
- 987 • **createSelfReference(Class<B> 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<B> 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

**Deleted:** March

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?

### 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.

**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.

**Deleted:** March



## 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)
```

Deleted: March

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 `package org.oasisopen.sca.client;`

1157 `public abstract class SCAClientFactory {`

1158 `protected static SCAClientFactoryFinder defaultFactoryFinder;`

1159 `public static SCAClient newInstance() {...}`

1160 `public static SCAClient newInstance(Properties properties) {...}`

1161 `public static SCAClient newInstance(ClassLoader classLoader) {...}`

1162 `public static SCAClient newInstance(Properties properties,`

1163 `ClassLoader classLoader) {...}`

1164 `protected abstract SCAClient createSCAClient();`

1165 `}`

1174 **newInstance() method:**

Formatted: Indent: Before: 0.25"

Formatted: Indent: Before: 0.25"

Deleted: ¶

Deleted: March

1175 [Obtains a object implementing the SCAClient interface.](#)

1176 [Returns:](#)

1177 

- [object which implements the SCAClient interface](#)

1178 [Parameters:](#)

1179 

- [none](#)

1180 [Exceptions:](#)

1181 

- [none](#)

1182

1183 **[newInstance\(Properties\) method:](#)**

1184 [Obtains a object implementing the SCAClient interface, using a specified set of properties.](#)

1185 [Returns:](#)

1186 

- [object which implements the SCAClient interface](#)

1187 [Parameters:](#)

1188 

- [properties - a set of Properties that can be used when creating the object which](#)

1189 [implements the SCAClient interface.](#)

1190 [Exceptions:](#)

1191 

- [none](#)

1192

1193 **[newInstance\(Classloader\) method:](#)**

1194 [Obtains a object implementing the SCAClient interface using a specified classloader.](#)

1195 [Returns:](#)

1196 

- [object which implements the SCAClient interface](#)

1197 [Parameters:](#)

1198 

- [classLoader - a ClassLoader to use when creating the object which implements the](#)

1199 [SCAClient interface.](#)

1200 [Exceptions:](#)

1201 

- [none](#)

1202

1203 **[newInstance\(Properties, Classloader\) method:](#)**

1204 [Obtains a object implementing the SCAClient interface using a specified set of properties and a](#)

1205 [specified classloader.](#)

1206 [Returns:](#)

1207 

- [object which implements the SCAClient interface](#)

1208 [Parameters:](#)

1209 

- [properties - a set of Properties that can be used when creating the object which](#)

1210 [implements the SCAClient interface.](#)

1211 

- [classLoader - a ClassLoader to use when creating the object which implements the](#)

1212 [SCAClient interface.](#)

1213 [Exceptions:](#)

1214 

- [none](#)

1215

1216 **defaultFactory protected field:**  
1217 Provides a means by which a provider of an SCAClientFactory implementation can inject a factory  
1218 finder implementation into the abstract SCAClientFactory class - once this is done, future  
1219 invocations of the SCAClientFactory use the injected factory finder to locate and return an instance  
1220 of a subclass of SCAClientFactory.

## 1221 **8.10 SCAClientFactoryFinder Interface**

1222 The SCAClientFactoryFinder interface is a Service Provider Interface representing a  
1223 SCAClientFactory finder. SCA provides a default reference implementation of this interface. SCA  
1224 runtime vendors can create alternative implementations of this interface that use different class  
1225 loading or lookup mechanisms.

```
1226 package org.oasisopen.sca.client;  
1227  
1228 import java.util.Properties;  
1229 public interface SCAClientFactoryFinder {  
1230     SCAClientFactory find(Properties properties,  
1231                           ClassLoader classLoader);  
1232 }  
1233
```

## 1234 **8.11 SCAClientFactoryFinderImpl Class**

1235 This class is a default implementation of an SCAClientFactoryFinder, which is used to find an  
1236 implementation of an SCAClientFactory subclass, as used to obtain an SCAClient object for use by  
1237 a client. SCA runtime providers can replace this implementation with their own version.

```
1238 package org.oasisopen.sca.client.impl;  
1239  
1240 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder  
1241 {  
1242     ...  
1243     public SCAClientFactoryFinderImpl() {...}  
1244  
1245     public SCAClientFactory find(Properties properties,  
1246                                 ClassLoader classLoader)  
1247         throws ServiceRuntimeException {...}  
1248     ...  
1249 }
```

← - - - Formatted: Indent: First line:  
0.25"

### 1250 **SCAClientFactoryFinderImpl () method:**

1251 Public constructor for the SCAClientFactoryFinderImpl.

1252 Returns:

- 1253 • **SCAClientFactoryFinderImpl** which implements the SCAClientFactoryFinder interface

← - - - Formatted: Bullets and  
Numbering

1254 Parameters:

- 1255 • **none**

← - - - Formatted: Bullets and  
Numbering

1256 Exceptions:

- 1257 • **none**

← - - - Formatted: Bullets and  
Numbering

### 1258 **find (Properties, ClassLoader) method:**

1259 Obtains an implementation of the SCAClientFactory interface. It discovers a provider's  
1260 SCAClientFactory implementation by referring to the following information in this order:

- 1261 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on  
1262 the newInstance() method call if specified

← - - - Formatted: Bullets and  
Numbering

Deleted: March

- 1263 [2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties](#)
- 1264 [3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file](#)

1265 **Returns:**

- 1266 • [SCAClientFactory](#) implementation object

← - - - **Formatted:** Bullets and Numbering

1267 **Parameters:**

- 1268 • [properties](#) - a set of Properties that can be used when creating the object which
- 1269 [implements the SCAClientFactory interface.](#)
- 1270 • [classLoader](#) - a ClassLoader to use when creating the object which implements the
- 1271 [SCAClientFactory interface.](#)

← - - - **Formatted:** Bullets and Numbering

**Formatted:** Font: Not Bold, Not Italic, Complex Script  
Font: Not Bold, Not Italic

1272 **Exceptions:**

- 1273 • [ServiceRuntimeException](#), - if the SCAClientFactory implementation could not be found

← - - - **Formatted:** Bullets and Numbering

**Formatted:** Font: Not Bold, Not Italic, Complex Script  
Font: Not Bold, Not Italic

1274

1275 **8.12 NoSuchDomainException**

1276 [The following shows the NoSuchDomainException:](#)

```
1277 package org.oasisopen.sca;
1278
1279 public class NoSuchDomainException extends Exception {
1280     ...
1281 }
```

1282 [This exception indicates that the Domain specified could not be found.](#)

1283 **8.13 NoSuchServiceException**

1284 [The following shows the NoSuchServiceException:](#)

```
1285 package org.oasisopen.sca;
1286
1287 public class NoSuchServiceException extends Exception {
1288     ...
1289 }
```

1290 [This exception indicates that the service specified could not be found.](#)

**Deleted:** March

## 1291 9 Java Annotations

1292 This section provides definitions of all the Java annotations which apply to SCA.

1293 This specification places constraints on some annotations that are not detectable by a Java  
1294 compiler. For example, the definition of the @Property and @Reference annotations indicate that  
1295 they are allowed on parameters, but sections 8.14 and 8.15 constrain those definitions to  
1296 constructor parameters. An SCA runtime MUST verify the proper use of all SCA annotations and if  
1297 an annotation is improperly used, the SCA runtime MUST NOT run the component which uses the  
1298 invalid implementation code. [JCA90001]

1299 SCA annotations MUST NOT be used on static methods or on static fields. It is an error to use an  
1300 SCA annotation on a static method or a static field of an implementation class and the SCA  
1301 runtime MUST NOT instantiate such an implementation class. [JCA90002]

### 1302 9.1 @AllowsPassByReference

1303 The following Java code defines the @AllowsPassByReference annotation:

```
1304 package org.oasisopen.sca.annotation;  
1305  
1306 import static java.lang.annotation.ElementType.TYPE;  
1307 import static java.lang.annotation.ElementType.METHOD;  
1308 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1309 import java.lang.annotation.Retention;  
1310 import java.lang.annotation.Target;  
1311  
1312 @Target({TYPE, METHOD})  
1313 @Retention(RUNTIME)  
1314 public @interface AllowsPassByReference {  
1315  
1316 }  
1317  
1318
```

1319 The @AllowsPassByReference annotation is used on implementations of remotable interfaces to  
1320 indicate that interactions with the service from a client within the same address space are allowed  
1321 to use pass by reference data exchange semantics. The implementation promises that its by-value  
1322 semantics will be maintained even if the parameters and return values are actually passed by-  
1323 reference. This means that the service will not modify any operation input parameter or return  
1324 value, even after returning from the operation. Either a whole class implementing a remotable  
1325 service or an individual remotable service method implementation can be annotated using the  
1326 @AllowsPassByReference annotation.

1327 @AllowsPassByReference has no attributes

1328 The following snippet shows a sample where @AllowsPassByReference is defined for the  
1329 implementation of a service method on the Java component implementation class.

```
1330  
1331 @AllowsPassByReference  
1332 public String hello(String message) {  
1333     ...  
1334 }
```

### 1335 9.2 @Authentication

1336 The following Java code defines the @Authentication annotation:

```

1337
1338 package org.oasisopen.sca.annotation;
1339
1340 import static java.lang.annotation.ElementType.FIELD;
1341 import static java.lang.annotation.ElementType.METHOD;
1342 import static java.lang.annotation.ElementType.PARAMETER;
1343 import static java.lang.annotation.ElementType.TYPE;
1344 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1345 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1346
1347 import java.lang.annotation.Inherited;
1348 import java.lang.annotation.Retention;
1349 import java.lang.annotation.Target;
1350
1351 @Inherited
1352 @Target({TYPE, FIELD, METHOD, PARAMETER})
1353 @Retention(RUNTIME)
1354 @Intent(Authentication.AUTHENTICATION)
1355 public @interface Authentication {
1356     String AUTHENTICATION = SCA_PREFIX + "authentication";
1357     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
1358     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";
1359
1360     /**
1361      * List of authentication qualifiers (such as "message"
1362      * or "transport").
1363      *
1364      * @return authentication qualifiers
1365      */
1366     @Qualifier
1367     String[] value() default "";
1368 }

```

1369 The **@Authentication** annotation is used to indicate that the invocation requires authentication.  
1370 See the [section on Application of Intent Annotations](#) for samples and details.

### 1371 9.3 @Callback

1372 The following Java code defines the **@Callback** annotation:

```

1373
1374 package org.oasisopen.sca.annotation;
1375
1376 import static java.lang.annotation.ElementType.TYPE;
1377 import static java.lang.annotation.ElementType.METHOD;
1378 import static java.lang.annotation.ElementType.FIELD;
1379 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1380 import java.lang.annotation.Retention;
1381 import java.lang.annotation.Target;
1382
1383 @Target(TYPE, METHOD, FIELD)
1384 @Retention(RUNTIME)
1385 public @interface Callback {
1386
1387     Class<?> value() default Void.class;
1388 }
1389
1390

```

1391 The @Callback annotation is used to annotate a service interface with a callback interface by  
1392 specifying the Java class object of the callback interface as an attribute.

1393 The @Callback annotation has the following attribute:

- 1394 • **value** – the name of a Java class file containing the callback interface

1395

1396 The @Callback annotation can also be used to annotate a method or a field of an SCA  
1397 implementation class, in order to have a callback object injected. When used to annotate a  
1398 method or a field of an implementation class for injection of a callback object, the @Callback  
1399 annotation MUST NOT specify any attributes. [JCA90046]

1400 An example use of the @Callback annotation to declare a callback interface follows:

```
1401 package somepackage;  
1402 import org.oasisopen.sca.annotation.Callback;  
1403 import org.oasisopen.sca.annotation.Remotable;  
1404 @Remotable  
1405 @Callback(MyServiceCallback.class)  
1406 public interface MyService {  
1407  
1408     void someMethod(String arg);  
1409 }  
1410  
1411 @Remotable  
1412 public interface MyServiceCallback {  
1413  
1414     void receiveResult(String result);  
1415 }  
1416
```

1417 In this example, the implied component type is:

```
1418 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
1419     <service name="MyService">  
1420         <interface.java interface="somepackage.MyService"  
1421             callbackInterface="somepackage.MyServiceCallback"/>  
1422     </service>  
1423 </componentType>
```

## 1425 9.4 @ComponentName

1426 The following Java code defines the @ComponentName annotation:

1427

```
1428 package org.oasisopen.sca.annotation;  
1429  
1430 import static java.lang.annotation.ElementType.METHOD;  
1431 import static java.lang.annotation.ElementType.FIELD;  
1432 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1433 import java.lang.annotation.Retention;  
1434 import java.lang.annotation.Target;  
1435  
1436 @Target({METHOD, FIELD})  
1437 @Retention(RUNTIME)  
1438 public @interface ComponentName {  
1439  
1440 }  
1441
```



1442 The @ComponentName annotation is used to denote a Java class field or setter method that is  
1443 used to inject the component name.

1444 The following snippet shows a component name field definition sample.

```
1445  
1446 @ComponentName  
1447 private String componentName;  
1448
```

1449 The following snippet shows a component name setter method sample.

```
1450  
1451 @ComponentName  
1452 public void setComponentName(String name) {  
1453     //...  
1454 }
```

## 1455 9.5 @Confidentiality

1456 The following Java code defines the **@Confidentiality** annotation:

```
1457 package org.oasisopen.sca.annotations;  
1458  
1459 import static java.lang.annotation.ElementType.FIELD;  
1460 import static java.lang.annotation.ElementType.METHOD;  
1461 import static java.lang.annotation.ElementType.PARAMETER;  
1462 import static java.lang.annotation.ElementType.TYPE;  
1463 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1464 import static org.oasisopen.sca.Constants.SCA_PREFIX;  
1465  
1466 import java.lang.annotation.Inherited;  
1467 import java.lang.annotation.Retention;  
1468 import java.lang.annotation.Target;  
1469  
1470 @Inherited  
1471 @Target({TYPE, FIELD, METHOD, PARAMETER})  
1472 @Retention(RUNTIME)  
1473 @Intent(Confidentiality.CONFIDENTIALITY)  
1474 public @interface Confidentiality {  
1475     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
1476     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
1477     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";  
1478  
1479     /**  
1480      * List of confidentiality qualifiers such as "message" or  
1481      * "transport".  
1482      *  
1483      * @return confidentiality qualifiers  
1484      */  
1485     @Qualifier  
1486     String[] value() default "";  
1487 }  
1488
```

1489 The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.

1490 See the [section on Application of Intent Annotations](#) for samples and details.

## 1491 9.6 @Constructor

1492 The following Java code defines the **@Constructor** annotation:

```
1493 package org.oasisopen.sca.annotation;
1494
1495 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1496 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1497 import java.lang.annotation.Retention;
1498 import java.lang.annotation.Target;
1499
1500 @Target({CONSTRUCTOR})
1501 @Retention(RUNTIME)
1502 public @interface Constructor { }
```

1505 The @Constructor annotation is used to mark a particular constructor to use when instantiating a  
1506 Java component implementation. If a constructor of an implementation class is annotated with  
1507 @Constructor and the constructor has parameters, each of these parameters MUST have either a  
1508 @Property annotation or a @Reference annotation. [JCA90003]

1509 The following snippet shows a sample for the @Constructor annotation.

```
1510
1511 public class HelloServiceImpl implements HelloService {
1512     public HelloServiceImpl(){
1513         ...
1514     }
1515
1516     @Constructor
1517     public HelloServiceImpl(@Property(name="someProperty")
1518                             String someProperty ){
1519         ...
1520     }
1521
1522     public String hello(String message) {
1523         ...
1524     }
1525 }
1526
```

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

## 1527 9.7 @Context

1528 The following Java code defines the **@Context** annotation:

```
1529
1530 package org.oasisopen.sca.annotation;
1531
1532 import static java.lang.annotation.ElementType.METHOD;
1533 import static java.lang.annotation.ElementType.FIELD;
1534 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1535 import java.lang.annotation.Retention;
1536 import java.lang.annotation.Target;
1537
1538 @Target({METHOD, FIELD})
1539 @Retention(RUNTIME)
1540 public @interface Context {
1541
```

**Deleted:** March

1542 }  
1543

1544 The @Context annotation is used to denote a Java class field or a setter method that is used to  
1545 inject a composite context for the component. The type of context to be injected is defined by the  
1546 type of the Java class field or type of the setter method input argument; the type is either  
1547 **ComponentContext** or **RequestContext**.

1548 The @Context annotation has no attributes.

1549 The following snippet shows a ComponentContext field definition sample.

1550

```
1551 @Context  
1552 protected ComponentContext context;  
1553
```

1554 The following snippet shows a RequestContext field definition sample.

1555

```
1556 @Context  
1557 protected RequestContext context;
```

## 1558 9.8 @Destroy

1559 The following Java code defines the **@Destroy** annotation:

1560

```
1561 package org.oasisopen.sca.annotation;  
1562  
1563 import static java.lang.annotation.ElementType.METHOD;  
1564 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1565 import java.lang.annotation.Retention;  
1566 import java.lang.annotation.Target;  
1567  
1568 @Target(METHOD)  
1569 @Retention(RUNTIME)  
1570 public @interface Destroy {  
1571  
1572 }  
1573
```

1574 The @Destroy annotation is used to denote a single Java class method that will be called when the  
1575 scope defined for the implementation class ends. A method annotated with @Destroy MAY have  
1576 any access modifier and MUST have a void return type and no arguments. [JCA90004]

1577 If there is a method annotated with @Destroy that matches the criteria for the annotation, the  
1578 SCA runtime MUST call the annotated method when the scope defined for the implementation  
1579 class ends. [JCA90005]

1580 The following snippet shows a sample for a destroy method definition.

1581

```
1582 @Destroy  
1583 public void myDestroyMethod() {  
1584     ...  
1585 }
```

## 1586 9.9 @EagerInit

1587 The following Java code defines the **@EagerInit** annotation:

1588

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

1602 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]

## 1606 9.10 @Init

1607 The following Java code defines the **@Init** annotation:

```
1608
1609 package org.oasisopen.sca.annotation;
1610
1611 import static java.lang.annotation.ElementType.METHOD;
1612 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1613 import java.lang.annotation.Retention;
1614 import java.lang.annotation.Target;
1615
1616 @Target(METHOD)
1617 @Retention(RUNTIME)
1618 public @interface Init {
1619
1620 }
1621
1622 }
```

1623 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]

1626 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]

1629 The following snippet shows an example of an init method definition.

```
1630
1631 @Init
1632 public void myInitMethod() {
1633     ...
1634 }
```

## 1635 9.11 @Integrity

1636 The following Java code defines the **@Integrity** annotation:

1637

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.

Deleted: March

```

1638 package org.oasisopen.sca.annotation;
1639
1640 import static java.lang.annotation.ElementType.FIELD;
1641 import static java.lang.annotation.ElementType.METHOD;
1642 import static java.lang.annotation.ElementType.PARAMETER;
1643 import static java.lang.annotation.ElementType.TYPE;
1644 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1645 import static org.oasisopen.Constants.SCA_PREFIX;
1646
1647 import java.lang.annotation.Inherited;
1648 import java.lang.annotation.Retention;
1649 import java.lang.annotation.Target;
1650
1651 @Inherited
1652 @Target({TYPE, FIELD, METHOD, PARAMETER})
1653 @Retention(RUNTIME)
1654 @Intent(Integrity.INTEGRITY)
1655 public @interface Integrity {
1656     String INTEGRITY = SCA_PREFIX + "integrity";
1657     String INTEGRITY_MESSAGE = INTEGRITY + ".message";
1658     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";
1659
1660     /**
1661      * List of integrity qualifiers (such as "message" or "transport").
1662      *
1663      * @return integrity qualifiers
1664      */
1665     @Qualifier
1666     String[] value() default "";
1667 }

```

1669 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no tampering of the messages between client and service).

1671 See the [section on Application of Intent Annotations](#) for samples and details.

## 1672 9.12 @Intent

1673 The following Java code defines the **@Intent** annotation:

```

1674 package org.oasisopen.sca.annotation;
1675
1676 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
1677 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1678 import java.lang.annotation.Retention;
1679 import java.lang.annotation.Target;
1680
1681 @Target({ANNOTATION_TYPE})
1682 @Retention(RUNTIME)
1683 public @interface Intent {
1684     /**
1685      * The qualified name of the intent, in the form defined by
1686      * {@link javax.xml.namespace.QName#toString}.
1687      * @return the qualified name of the intent
1688      */
1689     String value() default "";
1690
1691     /**
1692

```

Deleted: March

```

1693     * The XML namespace for the intent.
1694     * @return the XML namespace for the intent
1695     */
1696     String targetNamespace() default "";
1697
1698     /**
1699     * The name of the intent within its namespace.
1700     * @return name of the intent within its namespace
1701     */
1702     String localPart() default "";
1703 }
1704

```

1705 The @Intent annotation is used for the creation of new annotations for specific intents. It is not  
 1706 expected that the @Intent annotation will be used in application code.

1707 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to  
 1708 define new intent annotations.

### 1709 9.13 @OneWay

1710 The following Java code defines the **@OneWay** annotation:

```

1711
1712 package org.oasisopen.sca.annotation;
1713
1714 import static java.lang.annotation.ElementType.METHOD;
1715 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1716 import java.lang.annotation.Retention;
1717 import java.lang.annotation.Target;
1718
1719 @Target(METHOD)
1720 @Retention(RUNTIME)
1721 public @interface OneWay {
1722
1723
1724 }
1725

```

1726 The @OneWay annotation is used on a Java interface or class method to indicate that invocations  
 1727 will be dispatched in a non-blocking fashion as described in the section on Asynchronous  
 1728 [Programming](#).

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

1729 The @OneWay annotation has no attributes.

1730 The following snippet shows the use of the @OneWay annotation on an interface.

```

1731 package services.hello;
1732
1733 import org.oasisopen.sca.annotation.OneWay;
1734
1735 public interface HelloService {
1736     @OneWay
1737     void hello(String name);
1738 }

```

### 1739 9.14 @PolicySets

1740 The following Java code defines the **@PolicySets** annotation:

```

1741
1742 package org.oasisopen.sca.annotation;

```

**Deleted:** March

```

1743
1744 import static java.lang.annotation.ElementType.FIELD;
1745 import static java.lang.annotation.ElementType.METHOD;
1746 import static java.lang.annotation.ElementType.PARAMETER;
1747 import static java.lang.annotation.ElementType.TYPE;
1748 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1749
1750 import java.lang.annotation.Retention;
1751 import java.lang.annotation.Target;
1752
1753 @Target({TYPE, FIELD, METHOD, PARAMETER})
1754 @Retention(RUNTIME)
1755 public @interface PolicySets {
1756     /**
1757      * Returns the policy sets to be applied.
1758      *
1759      * @return the policy sets to be applied
1760      */
1761     String[] value() default "";
1762 }
1763

```

1764 The **@PolicySets** annotation is used to attach one or more SCA Policy Sets to a Java  
1765 implementation class or to one of its subelements.

1766 See the [section "Policy Set Annotations"](#) for details and samples.

## 1767 9.15 @Property

1768 The following Java code defines the **@Property** annotation:

```

1769 package org.oasisopen.sca.annotation;
1770
1771 import static java.lang.annotation.ElementType.METHOD;
1772 import static java.lang.annotation.ElementType.FIELD;
1773 import static java.lang.annotation.ElementType.PARAMETER;
1774 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1775 import java.lang.annotation.Retention;
1776 import java.lang.annotation.Target;
1777
1778 @Target({METHOD, FIELD, PARAMETER})
1779 @Retention(RUNTIME)
1780 public @interface Property {
1781
1782     String name() default "";
1783     boolean required() default true;
1784 }
1785

```

1786 The @Property annotation is used to denote a Java class field, a setter method, or a constructor  
1787 parameter that is used to inject an SCA property value. The type of the property injected, which  
1788 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or  
1789 the type of the input parameter of the setter method or constructor.

1790 The @Property annotation can be used on fields, on setter methods or on a constructor method  
1791 parameter. However, **the @Property annotation MUST NOT be used on a class field that is declared  
1792 as final. [JCA90011]**

1793 Properties can also be injected via setter methods even when the @Property annotation is not  
1794 present. However, **the @Property annotation MUST be used in order to inject a property onto a  
1795 non-public field. [JCA90012]** In the case where there is no @Property annotation, the name of the  
1796 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.

**Deleted:** March

1797 Where there is both a setter method and a field for a property, the setter method is used.

1798 The @Property annotation has the following attributes:

- 1799 • **name (optional)** – the name of the property. For a field annotation, the default is the  
1800 name of the field of the Java class. For a setter method annotation, the default is the  
1801 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a  
1802 @Property annotation applied to a constructor parameter, there is no default value for the  
1803 name attribute and the name attribute MUST be present. [JCA90013]
- 1804 • **required (optional)** – a boolean value which specifies whether injection of the property  
1805 value is required or not, where true means injection is required and false means injection  
1806 is not required. Defaults to true. For a @Property annotation applied to a constructor  
1807 parameter, the required attribute MUST have the value true. [JCA90014]

1808

1809 The following snippet shows a property field definition sample.

1810

```
1811 @Property(name="currency", required=true)  
1812 protected String currency;
```

1813

1814 The following snippet shows a property setter sample

1815

```
1816 @Property(name="currency", required=true)  
1817 public void setCurrency( String theCurrency ) {  
1818     ....  
1819 }
```

1820

1821 For a @Property annotation, if the the type of the Java class field or the type of the input  
1822 parameter of the setter method or constructor is defined as an array or as any type that extends  
1823 or implements java.util.Collection, then the SCA runtime MUST introspect the component type of  
1824 the implementation with a <property/> element with a @many attribute set to true, otherwise  
1825 @many MUST be set to false. [JCA90047]

1826 The following snippet shows the definition of a configuration property using the @Property  
1827 annotation for a collection.

```
1828 ...  
1829 private List<String> helloConfigurationProperty;  
1830  
1831 @Property(required=true)  
1832 public void setHelloConfigurationProperty(List<String> property) {  
1833     helloConfigurationProperty = property;  
1834 }  
1835 ...
```

## 1836 9.16 @Qualifier

1837 The following Java code defines the @Qualifier annotation:

```
1838 package org.oasisopen.sca.annotation;  
1839  
1840 import static java.lang.annotation.ElementType.METHOD;  
1841 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

1842  
1843

**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.

**Deleted:** March



```

1844 import java.lang.annotation.Retention;
1845 import java.lang.annotation.Target;
1846
1847 @Target(METHOD)
1848 @Retention(RUNTIME)
1849 public @interface Qualifier {
1850 }
1851

```

1852 The @Qualifier annotation is applied to an attribute of a specific intent annotation definition,  
1853 defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the  
1854 intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the  
1855 intent has qualifiers. [JCA90015]

1856 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to  
1857 define new intent annotations.

## 1858 9.17 @Reference

1859 The following Java code defines the **@Reference** annotation:

```

1860
1861 package org.oasisopen.sca.annotation;
1862
1863 import static java.lang.annotation.ElementType.METHOD;
1864 import static java.lang.annotation.ElementType.FIELD;
1865 import static java.lang.annotation.ElementType.PARAMETER;
1866 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1867 import java.lang.annotation.Retention;
1868 import java.lang.annotation.Target;
1869 @Target({METHOD, FIELD, PARAMETER})
1870 @Retention(RUNTIME)
1871 public @interface Reference {
1872
1873     String name() default "";
1874     boolean required() default true;
1875 }
1876

```

1877 The @Reference annotation type is used to annotate a Java class field, a setter method, or a  
1878 constructor parameter that is used to inject a service that resolves the reference. The interface of  
1879 the service injected is defined by the type of the Java class field or the type of the input parameter  
1880 of the setter method or constructor.

1881 The @Reference annotation MUST NOT be used on a class field that is declared as final.  
1882 [JCA90016]

1883 References can also be injected via setter methods even when the @Reference annotation is not  
1884 present. However, the @Reference annotation MUST be used in order to inject a reference onto a  
1885 non-public field. [JCA90017] In the case where there is no @Reference annotation, the name of  
1886 the reference is the same as the name of the field or setter.

1887 Where there is both a setter method and a field for a reference, the setter method is used.

1888 The @Reference annotation has the following attributes:

- 1889 • **name : String (optional)** – the name of the reference. For a field annotation, the default is  
1890 the name of the field of the Java class. For a setter method annotation, the default is the  
1891 JavaBeans property name corresponding to the setter method name. For a @Reference  
1892 annotation applied to a constructor parameter, there is no default for the name attribute  
1893 and the name attribute MUST be present. [JCA90018]

- 1894
- **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]
- 1895
- 1896
- 1897

1898

1899 The following snippet shows a reference field definition sample.

1900

```
1901 @Reference(name="stockQuote", required=true)
1902 protected StockQuoteService stockQuote;
```

1903

1904 The following snippet shows a reference setter sample

1905

```
1906 @Reference(name="stockQuote", required=true)
1907 public void setStockQuote( StockQuoteService theSQService ) {
1908     ...
1909 }
```

1910

1911 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.

1915

```
1916 package services.hello;
1917
1918 private HelloService helloService;
1919
1920 @Reference(name="helloService", required=true)
1921 public setHelloService(HelloService service) {
1922     helloService = service;
1923 }
1924
1925 public void clientMethod() {
1926     String result = helloService.hello("Hello World!");
1927     ...
1928 }
```

1929

1930 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.

1933

```
1934 <?xml version="1.0" encoding="ASCII"?>
1935 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1936
1937     <!-- Any services offered by the component would be listed here -->
1938     <reference name="helloService" multiplicity="1..1">
1939         <interface.java interface="services.hello.HelloService"/>
1940     </reference>
1941
1942 </componentType>
```

1943

1944 If the type of a reference is not an array or any type that extends or implements  
1945 java.util.Collection, then the SCA runtime MUST introspect the component type of the  
1946 implementation with a <reference/> element with @multiplicity= 0..1 if the @Reference  
1947 annotation required attribute is false and with @multiplicity=1..1 if the @Reference  
1948 annotation required attribute is true. [JCA90020]

1949 If the type of a reference is defined as an array or as any type that extends or implements  
1950 java.util.Collection, then the SCA runtime MUST introspect the component type of the  
1951 implementation with a <reference/> element with @multiplicity=0..n if the @Reference  
1952 annotation required attribute is false and with @multiplicity=1..n if the @Reference annotation  
1953 required attribute is true. [JCA90021]

1954 The following fragment from a component implementation shows a sample of a service reference  
1955 definition using the @Reference annotation on a java.util.List. The name of the reference is  
1956 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the  
1957 services referenced by the helloServices reference. In this case, at least one HelloService needs  
1958 to be present, so **required** is true.

```
1959 @Reference(name="helloServices", required=true)  
1960 protected List<HelloService> helloServices;  
1961  
1962 public void clientMethod() {  
1963     ...  
1964     for (int index = 0; index < helloServices.size(); index++) {  
1965         HelloService helloService =  
1966             (HelloService)helloServices.get(index);  
1967         String result = helloService.hello("Hello World!");  
1968     }  
1969     ...  
1970 }  
1971 }  
1972 }  
1973 }
```

1974 The following snippet shows the XML representation of the component type reflected from for the  
1975 former component implementation fragment. There is no need to author this component type in  
1976 this case since it can be reflected from the Java class.

```
1977  
1978 <?xml version="1.0" encoding="ASCII"?>  
1979 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
1980  
1981     <!-- Any services offered by the component would be listed here -->  
1982     <reference name="helloServices" multiplicity="1..n">  
1983         <interface.java interface="services.hello.HelloService"/>  
1984     </reference>  
1985  
1986 </componentType>
```

1987  
1988 An unwired reference with a multiplicity of 0..1 MUST be presented to the implementation code by  
1989 the SCA runtime as null. [JCA90022] An unwired reference with a multiplicity of 0..n MUST be  
1990 presented to the implementation code by the SCA runtime as an empty array or empty collection.  
1991 [JCA90023]

## 1992 9.17.1 Reinjection

1993 References MAY be reinjected by an SCA runtime after the initial creation of a component if the  
1994 reference target changes due to a change in wiring that has occurred since the component was  
1995 initialized. [JCA90024]

1996 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 component type of the implementation with a <reference/> ... [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 ... [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 ... [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 a ... [4]

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

**Deleted:** March

1997 1. The component MUST NOT be STATELESS scoped.

1998 2. The reference MUST use either field-based injection or setter injection. References that are

1999 injected through constructor injection MUST NOT be changed.

2000 [JCA90025]

2001 Setter injection allows for code in the setter method to perform processing in reaction to a change.

2002 If a reference target changes and the reference is not reinjected, the reference MUST continue to

2003 work as if the reference target was not changed. [JCA90026]

2004 If an operation is called on a reference where the target of that reference has been undeployed,

2005 the SCA runtime SHOULD throw an InvalidServiceException. [JCA90027] If an operation is called

2006 on a reference where the target of the reference has become unavailable for some reason, the

2007 SCA runtime SHOULD throw a ServiceUnavailableException. [JCA90028] If the target service of

2008 the reference is changed, the reference MAY continue to work, depending on the runtime and the

2009 type of change that was made. [JCA90029] If it doesn't work, the exception thrown will depend on

2010 the runtime and the cause of the failure.

2011 A ServiceReference that has been obtained from a reference by ComponentContext.cast()

2012 corresponds to the reference that is passed as a parameter to cast(). If the reference is

2013 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue

2014 to work as if the reference target was not changed. [JCA90030] If the target of a ServiceReference

2015 has been undeployed, the SCA runtime SHOULD throw a InvalidServiceException when an

2016 operation is invoked on the ServiceReference. [JCA90031] If the target of a ServiceReference has

2017 become unavailable, the SCA runtime SHOULD throw a ServiceUnavailableException when an

2018 operation is invoked on the ServiceReference. [JCA90032] If the target service of a

2019 ServiceReference is changed, the reference MAY continue to work, depending on the runtime and

2020 the type of change that was made. [JCA90033] If it doesn't work, the exception thrown will

2021 depend on the runtime and the cause of the failure.

2022 A reference or ServiceReference accessed through the component context by calling getService()

2023 or getServiceReference() MUST correspond to the current configuration of the domain. This applies

2024 whether or not reinjection has taken place. [JCA90034] If the target of a reference or

2025 ServiceReference accessed through the component context by calling getService() or

2026 getServiceReference() has been undeployed or has become unavailable, the result SHOULD be a

2027 reference to the undeployed or unavailable service, and attempts to call business methods

2028 SHOULD throw an InvalidServiceException or a ServiceUnavailableException. [JCA90035] If the

2029 target service of a reference or ServiceReference accessed through the component context by

2030 calling getService() or getServiceReference() has changed, the returned value SHOULD be a

2031 reference to the changed service. [JCA90036]

2032 The rules for reference reinjection also apply to references with a multiplicity of 0..n or 1..n. This

2033 means that in the cases where reference reinjection is not allowed, the array or Collection for a

2034 reference of multiplicity 0..n or multiplicity 1..n MUST NOT change its contents when changes

2035 occur to the reference wiring or to the targets of the wiring. [JCA90037] In cases where the

2036 contents of a reference array or collection change when the wiring changes or the targets change,

2037 then for references that use setter injection, the setter method MUST be called by the SCA

2038 runtime for any change to the contents. [JCA90038] A reinjected array or Collection for a

2039 reference MUST NOT be the same array or Collection object previously injected to the component.

2040 [JCA90039]

2041

**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 constructor injection MUST NOT be changed.

**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 constructor injection MUST NOT be changed.

**Deleted:** If the target service of the reference is changed, the referen ... [6]

**Deleted:** If the target service of the reference is changed, the referen ... [7]

**Deleted:** If the target service of a ServiceReference is ... [8]

**Deleted:** If the target service of a ServiceReference is ... [9]

**Deleted:** A reference or ServiceReference accessed through the compon ... [10]

**Deleted:** A reference or ServiceReference accessed through the compon ... [11]

**Deleted:** If the target of a reference or ServiceReference ac ... [12]

**Deleted:** If the target of a reference or ServiceReference ac ... [13]

**Deleted:** If the target service of a reference or ServiceReference ac ... [14]

**Deleted:** If the target service of a reference or ServiceReference ac ... [15]

**Deleted:** In cases where the contents of a reference array or collection c ... [16]

**Deleted:** In cases where the contents of a reference array or collection c ... [17]

**Deleted:** March

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

2042

## 2043 9.18 @Remotable

2044 The following Java code defines the **@Remotable** annotation:

2045

```
2046 package org.oasisopen.sca.annotation;
```

2047

```
2048 import static java.lang.annotation.ElementType.TYPE;
```

```
2049 import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
2050 import java.lang.annotation.Retention;
```

```
2051 import java.lang.annotation.Target;
```

2052

2053

```
2054 @Target(TYPE)
```

```
2055 @Retention(RUNTIME)
```

```
2056 public @interface Remotable {
```

2057

2058

2059

2060 The @Remotable annotation is used to specify a Java service interface as remotable. A remotable  
 2061 service can be published externally as a service and MUST be translatable into a WSDL portType.  
 2062 [JCA90040]

2063 The @Remotable annotation has no attributes.

2064 The following snippet shows the Java interface for a remotable service with its @Remotable  
 2065 annotation.

Deleted: March

```

2066 package services.hello;
2067
2068 import org.oasisopen.sca.annotation.*;
2069
2070 @Remotable
2071 public interface HelloService {
2072     String hello(String message);
2073 }
2074
2075

```

2076 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**  
2077 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

2078 Complex data types exchanged via remotable service interfaces need to be compatible with the  
2079 marshalling technology used by the service binding. For example, if the service is going to be  
2080 exposed using the standard Web Service binding, then the parameters can be JAXB [JAX-B] types  
2081 or they can be Service Data Objects (SDOs) [SDO].

2082 Independent of whether the remotable service is called from outside of the composite that  
2083 contains it or from another component in the same composite, the data exchange semantics are  
2084 **by-value**.

2085 Implementations of remotable services can modify input data during or after an invocation and  
2086 can modify return data after the invocation. If a remotable service is called locally or remotely, the  
2087 SCA container is responsible for making sure that no modification of input data or post-invocation  
2088 modifications to return data are seen by the caller.

2089 The following snippet shows a remotable Java service interface.

```

2090
2091 package services.hello;
2092
2093 import org.oasisopen.sca.annotation.*;
2094
2095 @Remotable
2096 public interface HelloService {
2097     String hello(String message);
2098 }
2099
2100 package services.hello;
2101
2102 import org.oasisopen.sca.annotation.*;
2103
2104 @Service(HelloService.class)
2105 public class HelloServiceImpl implements HelloService {
2106     public String hello(String message) {
2107         ...
2108     }
2109 }
2110
2111

```

## 2112 9.19 @Requires

2113 The following Java code defines the **@Requires** annotation:

```

2114 package org.oasisopen.sca.annotation;
2115
2116 import static java.lang.annotation.ElementType.FIELD;
2117

```

Deleted: March

```

2118 import static java.lang.annotation.ElementType.METHOD;
2119 import static java.lang.annotation.ElementType.PARAMETER;
2120 import static java.lang.annotation.ElementType.TYPE;
2121 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2122
2123 import java.lang.annotation.Inherited;
2124 import java.lang.annotation.Retention;
2125 import java.lang.annotation.Target;
2126
2127 @Inherited
2128 @Retention(RUNTIME)
2129 @Target({TYPE, METHOD, FIELD, PARAMETER})
2130 public @interface Requires {
2131     /**
2132      * Returns the attached intents.
2133      *
2134      * @return the attached intents
2135      */
2136     String[] value() default "";
2137 }
2138

```

2139 The **@Requires** annotation supports general purpose intents specified as strings. Users can also  
2140 define specific intent annotations using the @Intent annotation.

2141 See the [section "General Intent Annotations"](#) for details and samples.

## 2142 9.20 @Scope

2143 The following Java code defines the **@Scope** annotation:

```

2144 package org.oasisopen.sca.annotation;
2145
2146 import static java.lang.annotation.ElementType.TYPE;
2147 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2148 import java.lang.annotation.Retention;
2149 import java.lang.annotation.Target;
2150
2151 @Target(TYPE)
2152 @Retention(RUNTIME)
2153 public @interface Scope {
2154
2155     String value() default "STATELESS";
2156 }

```

2157 The @Scope annotation MUST only be used on a service's implementation class. It is an error to  
2158 use this annotation on an interface. [JCA90041]

2159 The @Scope annotation has the following attribute:

- 2160 • **value** – the name of the scope.  
2161 SCA defines the following scope names, but others can be defined by particular Java-  
2162 based implementation types:  
2163 STATELESS  
2164 COMPOSITE  
2165 For 'STATELESS' implementations, a different implementation instance can be used to  
2166 service each request. Implementation instances can be newly created or be drawn from a  
2167 pool of instances.

2168 The default value is STATELESS.

2169 The following snippet shows a sample for a COMPOSITE scoped service implementation:

```

2170 package services.hello;

```

```

2171
2172 import org.oasisopen.sca.annotation.*;
2173
2174 @Service(HelloService.class)
2175 @Scope("COMPOSITE")
2176 public class HelloServiceImpl implements HelloService {
2177
2178     public String hello(String message) {
2179         ...
2180     }
2181 }
2182

```

## 9.21 @Service

2183  
2184 The following Java code defines the **@Service** annotation:

```

2185 package org.oasisopen.sca.annotation;
2186
2187 import static java.lang.annotation.ElementType.TYPE;
2188 import static java.lang.annotation.RetentionPolicy.RUNTIME;
2189 import java.lang.annotation.Retention;
2190 import java.lang.annotation.Target;
2191
2192 @Target(TYPE)
2193 @Retention(RUNTIME)
2194 public @interface Service {
2195
2196     Class<?>[] interfaces() default {};
2197     Class<?> value() default Void.class;
2198 }
2199

```

2200 The @Service annotation is used on a component implementation class to specify the SCA services  
2201 offered by the implementation. **An implementation class need not be declared as implementing all  
2202 of the interfaces implied by the services declared in its @Service annotation, but all methods of all  
2203 the declared service interfaces MUST be present. [JCA90042]** A class used as the implementation  
2204 of a service is not required to have a @Service annotation. If a class has no @Service annotation,  
2205 then the rules determining which services are offered and what interfaces those services have are  
2206 determined by the specific implementation type.

2207 The @Service annotation has the following attributes:

- 2208 • **interfaces (1..1)** – The value is an array of interface or class objects that are exposed as  
2209 services by this component implementation.
- 2210 • **value** – A shortcut for the case when the class provides only a single service interface -  
2211 contains a single interface or class object that is exposed as a service by this component  
2212 implementation.

2213 **A @Service annotation MUST only have one of the interfaces attribute or value attribute specified.  
2214 [JCA90043]**

2215  
2216 **A @Service annotation with no attributes MUST be ignored, it is the same as not having the  
2217 annotation there at all. [JCA90044]**

2218 The **service names** of the defined services default to the names of the interfaces or class, without  
2219 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.**

**Deleted: March**



2220 **A component implementation MUST NOT have two services with the same Java simple name.**  
2221 **[JCA90045]** If a Java implementation needs to realize two services with the same Java simple  
2222 name then this can be achieved through subclassing of the interface.

2223 The following snippet shows an implementation of the HelloService marked with the @Service  
2224 annotation.

```
2225 package services.hello;  
2226  
2227 import org.oasisopen.sca.annotation.Service;  
2228  
2229 @Service(HelloService.class)  
2230 public class HelloServiceImpl implements HelloService {  
2231  
2232     public void hello(String name) {  
2233         System.out.println("Hello " + name);  
2234     }  
2235 }  
2236
```

2237

## 10 WSDL to Java and Java to WSDL

2238  
2239  
2240

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

2241  
2242  
2243  
2244  
2245  
2246

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

**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.

2247  
2248  
2249  
2250  
2251  
2252

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

**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.

2253  
2254

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

- No support for holders

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

2255  
2256  
2257

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

2258

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

2259  
2260  
2261  
2262  
2263

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

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

2264  
2265  
2266  
2267  
2268  
2269  
2270  
2271

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

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

2272  
2273

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

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

2274  
2275  
2276  
2277

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

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

2278  
2279

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

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

2280  
2281  
2282

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

**Deleted:** March

2283 c. a return type of Future<?>  
2284 then C is a JAX-WS callback method that isn't part of the SCA interface contract.  
2285 As an example, an interface can be defined in WSDL as follows:

```
2286 <!-- WSDL extract -->  
2287 <message name="getPrice">  
2288   <part name="ticker" type="xsd:string"/>  
2289 </message>  
2290  
2291 <message name="getPriceResponse">  
2292   <part name="price" type="xsd:float"/>  
2293 </message>  
2294  
2295 <portType name="StockQuote">  
2296   <operation name="getPrice">  
2297     <input message="tns:getPrice"/>  
2298     <output message="tns:getPriceResponse"/>  
2299   </operation>  
2300 </portType>
```

2301  
2302 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2303 // asynchronous mapping  
2304 @WebService  
2305 public interface StockQuote {  
2306   float getPrice(String ticker);  
2307   Response<Float> getPriceAsync(String ticker);  
2308   Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);  
2309 }
```

2310  
2311 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2312 // synchronous mapping  
2313 @WebService  
2314 public interface StockQuote {  
2315   float getPrice(String ticker);  
2316 }
```

2317  
2318 **SCA runtimes MUST support the use of the JAX-WS client asynchronous model.** [JCA100009] In  
2319 the above example, if the client implementation uses the asynchronous form of the interface, the  
2320 two additional getPriceAsync() methods can be used for polling and callbacks as defined by the  
2321 JAX-WS specification.

2322

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

```
2323 <?xml version="1.0" encoding="UTF-8"?>
2324 <!-- (c) Copyright SCA Collaboration 2006 -->
2325 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2326         targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2327         xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2328         elementFormDefault="qualified">
2329
2330     <include schemaLocation="sca-core.xsd"/>
2331
2332     <element name="interface.java" type="sca:JavaInterface"
2333             substitutionGroup="sca:interface"/>
2334     <complexType name="JavaInterface">
2335         <complexContent>
2336             <extension base="sca:Interface">
2337                 <sequence>
2338                     <any namespace="##other" processContents="lax"
2339                         minOccurs="0" maxOccurs="unbounded"/>
2340                 </sequence>
2341                 <attribute name="interface" type="NCName" use="required"/>
2342                 <attribute name="callbackInterface" type="NCName"
2343                         use="optional"/>
2344                 <anyAttribute namespace="##any" processContents="lax"/>
2345             </extension>
2346         </complexContent>
2347     </complexType>
2348 </schema>
2349
```

2350  
2351  
2352  
2353  
2354  
2355  
2356  
2357  
2358  
2359  
2360  
2361  
2362  
2363  
2364  
2365  
2366  
2367  
2368  
2369  
2370  
2371  
2372  
2373  
2374  
2375  
2376  
2377  
2378  
2379  
2380  
2381  
2382  
2383  
2384  
2385  
2386  
2387  
2388  
2389  
2390  
2391  
2392  
2393  
2394  
2395  
2396  
2397  
2398  
2399

## **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

Deleted: March

2400  
2401

Deleted: D

## 2402 **B.1.2 SCAClientFactory Class**

2403 SCA provides an abstract base class SCAClientFactory. Vendors can provide subclasses of this class  
2404 which create objects that implement the SCAClient interface suitable for linking to services in their SCA  
2405 runtime.

2406

```
2407 /*  
2408 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.  
2409 * OASIS trademark, IPR and other policies apply.  
2410 */
```

2411

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

2412

```
import java.util.Properties;
```

2413

2414

```
import org.oasisopen.sca.client.SCAClientFactoryFinder;
```

2415

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

2416

2417

```
/**  
2419 * The SCAClientFactory can be used by non-SCA managed code to  
2420 * lookup services that exist in a SCADomain.
```

2421

```
* @see SCAClientFactoryFinderImpl  
2422 * @see SCAClient
```

2423

2424

```
* @author OASIS Open  
2425 */
```

2426

2427

```
public abstract class SCAClientFactory {
```

2428

2429

```
    /**  
2430      * The SCAClientFactoryFinder.  
2431      * Provides a means by which a provider of an SCAClientFactory  
2432      * implementation can inject a factory finder implementation into  
2433      * the abstract SCAClientFactory class - once this is done, future  
2434      * invocations of the SCAClientFactory use the injected factory  
2435      * finder to locate and return an instance of a subclass of  
2436      * SCAClientFactory.  
2437      */
```

2438

```
    protected static SCAClientFactoryFinder factoryFinder;
```

2439

2440

```
    /**  
2441      * Creates a new instance of the SCAClient that can be  
2442      * used to lookup SCA Services.
```

2443

```
     * @return A new SCAClient  
2444      */
```

2445

```
    public static SCAClient newInstance() {  
2446         return newInstance(null, null);  
2447     }
```

2448

2449

```
    /**  
2450      * Creates a new instance of the SCAClient that can be  
2451      * used to lookup SCA Services.  
2452      *
```

2453

2454

Deleted: March

```

2455     * @param properties Properties that may be used when
2456     * creating a new instance of the SCAClient
2457     * @return A new SCAClient instance
2458     */
2459     public static SCAClient newInstance(Properties properties) {
2460         return newInstance(properties, null);
2461     }
2462
2463     /**
2464     * Creates a new instance of the SCAClient that can be
2465     * used to lookup SCA Services.
2466     *
2467     * @param classLoader ClassLoader that may be used when
2468     * creating a new instance of the SCAClient
2469     * @return A new SCAClient instance
2470     */
2471     public static SCAClient newInstance(ClassLoader classLoader) {
2472         return newInstance(null, classLoader);
2473     }
2474
2475     /**
2476     * Creates a new instance of the SCAClient that can be
2477     * used to lookup SCA Services.
2478     *
2479     * @param properties Properties that may be used when
2480     * creating a new instance of the SCAClient
2481     * @param classLoader ClassLoader that may be used when
2482     * creating a new instance of the SCAClient
2483     * @return A new SCAClient instance
2484     */
2485     public static SCAClient newInstance(Properties properties,
2486                                         ClassLoader classLoader) {
2487         final SCAClientFactoryFinder finder =
2488             factoryFinder != null ? factoryFinder :
2489             new SCAClientFactoryFinderImpl();
2490         final SCAClientFactory factory
2491             = finder.find(properties, classLoader);
2492         return factory.createSCAClient();
2493     }
2494
2495     /**
2496     * This method is invoked to create a new SCAClient instance.
2497     *
2498     * @return A new SCAClient instance
2499     */
2500     protected abstract SCAClient createSCAClient();
2501 }

```

### 2503 **B.1.3 SCAClientFactoryFinder interface**

2504 The SCAClientFactoryFinder interface is a Service Provider Interface representing a SCAClientFactory  
2505 finder. SCA provides a default reference implementation of this interface. SCA runtime vendors can  
2506 create alternative implementations of this interface that use different class loading or lookup mechanisms.

```

2507 /*
2508 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
2509 * OASIS trademark, IPR and other policies apply.
2510 */

```

**Deleted:** protected

**Formatted:** Font: (Default)  
Courier New, Complex Script  
Font: Courier New, 10 pt

**Formatted:** Space Before: 0  
pt, After: 0 pt, Don't adjust  
space between Latin and  
Asian text, Don't adjust space  
between Asian text and  
numbers

**Formatted:** Bullets and  
Numbering

**Formatted:** Normal

**Deleted:** March

```

2511 package org.oasisopen.sca.client;
2512
2513
2514 import java.util.Properties;
2515
2516 /* A Service Provider Interface representing a SCAClientFactory finder.
2517 * SCA provides a default reference implementation of this interface.
2518 * SCA runtime vendors can create alternative implementations of this
2519 * interface that use different class loading or lookup mechanisms.
2520 */
2521 public interface SCAClientFactoryFinder {
2522
2523     SCAClientFactory find(Properties properties,
2524                            ClassLoader classLoader);
2525 }

```

Formatted: Normal

Formatted: Bullets and Numbering

#### 2526 **B.1.4 SCAClientFactoryFinderImpl class**

2527 This class provides a default implementation for finding a provider's SCAClientFactory implementation  
2528 class. It is used if the provider does not inject its SCAClientFactoryFinder implementation class into the  
2529 base SCAClientFactory class.

2530 It discovers a provider's SCAClientFactory implementation by referring to the following information in this  
2531 order:

- 2532 1. The org.oasisopen.sca.client.SCAClientFactory property from the Properties specified on the  
2533 newInstance() method call if specified
- 2534 2. The org.oasisopen.sca.client.SCAClientFactory property from the System Properties
- 2535 3. The META-INF/services/org.oasisopen.sca.client.SCAClientFactory file

```

2536 /*
2537 * Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
2538 * OASIS trademark, IPR and other policies apply.
2539 */

```

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.¶

```

2540 package org.oasisopen.sca.client.impl;
2541
2542 import org.oasisopen.sca.client.SCAClientFactoryFinder;
2543
2544 import java.io.BufferedReader;
2545 import java.io.Closeable;
2546 import java.io.IOException;
2547 import java.io.InputStream;
2548 import java.io.InputStreamReader;
2549 import java.net.URL;
2550 import java.util.Properties;
2551
2552 import org.oasisopen.sca.ServiceRuntimeException;
2553 import org.oasisopen.sca.client.SCAClientFactory;

```

Formatted: French France

```

2554
2555 /**
2556 * This is a default implementation of an SCAClientFactoryFinder which is
2557 * used to find an implementation of the SCAClientFactory interface.
2558 *
2559 * @see SCAClientFactoryFinder
2560 * @see SCAClientFactory
2561 *
2562 * @author OASIS Open
2563 */
2564 public class SCAClientFactoryFinderImpl implements SCAClientFactoryFinder {

```

Deleted: March



```

2565
2566 /**
2567  * The name of the System Property used to determine the SPI
2568  * implementation to use for the SCAClientFactory.
2569  */
2570 private static final String SCA_CLIENT_FACTORY_PROVIDER_KEY =
2571 SCAClientFactory.class.getName();
2572
2573 /**
2574  * The name of the file loaded from the ClassPath to determine
2575  * the SPI implementation to use for the SCAClientFactory.
2576  */
2577 private static final String SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE
2578  = "META-INF/services/" + SCA_CLIENT_FACTORY_PROVIDER_KEY;
2579
2580 /**
2581  * Public Constructor
2582  */
2583 public SCAClientFactoryFinderImpl() {
2584  }
2585
2586 /**
2587  * Creates an instance of the SCAClientFactorySPI implementation.
2588  * This discovers the SCAClientFactorySPI Implementation and instantiates
2589  * the provider's implementation.
2590  *
2591  * @param properties Properties that may be used when creating a new
2592  * instance of the SCAClient
2593  * @param classLoader ClassLoader that may be used when creating a new
2594  * instance of the SCAClient
2595  * @return new instance of the SCAClientFactory
2596  * @throws ServiceRuntimeException Failed to create SCAClientFactory
2597  * Implementation.
2598  */
2599 public SCAClientFactory find(Properties properties,
2600                              ClassLoader classLoader)
2601  throws ServiceRuntimeException {
2602      if (classLoader == null) {
2603          classLoader = getThreadContextClassLoader();
2604      }
2605      final String factoryImplClassName =
2606          discoverProviderFactoryImplClass(properties, classLoader);
2607      final Class<? extends SCAClientFactory> factoryImplClass
2608          = loadProviderFactoryClass(factoryImplClassName,
2609                                    classLoader);
2610      final SCAClientFactory factory =
2611          instantiateSCAClientFactoryClass(factoryImplClass);
2612      return factory;
2613  }
2614
2615 /**
2616  * Gets the Context ClassLoader for the current Thread.
2617  *
2618  * @return The Context ClassLoader for the current Thread.
2619  */
2620 private static ClassLoader getThreadContextClassLoader() {
2621      final ClassLoader threadClassLoader =
2622          Thread.currentThread().getContextClassLoader();

```

Deleted: March

```

2623     return threadClassLoader;
2624 }
2625
2626 /**
2627  * Attempts to discover the class name for the SCAClientFactorySPI
2628  * implementation from the specified Properties, the System Properties
2629  * or the specified ClassLoader.
2630  *
2631  * @return The class name of the SCAClientFactorySPI implementation
2632  * @throw ServiceRuntimeException Failed to find implementation for
2633  * SCAClientFactorySPI.
2634  */
2635 private static String
2636 discoverProviderFactoryImplClass(Properties properties,
2637 ClassLoader classLoader)
2638 throws ServiceRuntimeException {
2639     String providerClassName =
2640         checkPropertiesForSPIClassName(properties);
2641     if (providerClassName != null) {
2642         return providerClassName;
2643     }
2644
2645     providerClassName =
2646         checkPropertiesForSPIClassName(System.getProperties());
2647     if (providerClassName != null) {
2648         return providerClassName;
2649     }
2650
2651     providerClassName = checkMETAINFServicesForSIPClassName(classLoader);
2652     if (providerClassName == null) {
2653         throw new ServiceRuntimeException(
2654             "Failed to find implementation for SCAClientFactory");
2655     }
2656
2657     return providerClassName;
2658 }
2659
2660 /**
2661  * Attempts to find the class name for the SCAClientFactorySPI
2662  * implementation from the specified Properties.
2663  *
2664  * @return The class name for the SCAClientFactorySPI implementation
2665  * or <code>null</code> if not found.
2666  */
2667 private static String
2668 checkPropertiesForSPIClassName(Properties properties) {
2669     if (properties == null) {
2670         return null;
2671     }
2672
2673     final String providerClassName =
2674         properties.getProperty(SCA_CLIENT_FACTORY_PROVIDER_KEY);
2675     if (providerClassName != null && providerClassName.length() > 0) {
2676         return providerClassName;
2677     }
2678
2679     return null;
2680 }

```

```

2681
2682 /**
2683  * Attempts to find the class name for the SCAClientFactorySPI
2684  * implementation from the META-INF/services directory
2685  *
2686  * @return The class name for the SCAClientFactorySPI implementation or
2687  * <code>null</code> if not found.
2688  */
2689 private static String checkMETA-INFServicesForSIPClassName(ClassLoader cl)
2690 {
2691     final URL url =
2692         cl.getResource(SCA_CLIENT_FACTORY_PROVIDER_META_INF_SERVICE);
2693     if (url == null) {
2694         return null;
2695     }
2696
2697     InputStream in = null;
2698     try {
2699         in = url.openStream();
2700         BufferedReader reader = null;
2701         try {
2702             reader =
2703                 new BufferedReader(new InputStreamReader(in, "UTF-8"));
2704
2705             String line;
2706             while ((line = readNextLine(reader)) != null) {
2707                 if (!line.startsWith("#") && line.length() > 0) {
2708                     return line;
2709                 }
2710             }
2711
2712             return null;
2713         } finally {
2714             closeStream(reader);
2715         }
2716     } catch (IOException ex) {
2717         throw new ServiceRuntimeException(
2718             "Failed to discover SCAClientFactory provider", ex);
2719         } finally {
2720             closeStream(in);
2721         }
2722     }
2723
2724 /**
2725  * Reads the next line from the reader and returns the trimmed version
2726  * of that line
2727  *
2728  * @param reader The reader from which to read the next line
2729  * @return The trimmed next line or <code>null</code> if the end of the
2730  * stream has been reached
2731  * @throws IOException I/O error occurred while reading from Reader
2732  */
2733 private static String readNextLine(BufferedReader reader)
2734 throws IOException {
2735
2736     String line = reader.readLine();
2737     if (line != null) {
2738         line = line.trim();

```

```

2739     }
2740     return line;
2741 }
2742
2743 /**
2744  * Loads the specified SCAClientFactory Implementation class.
2745  *
2746  * @param factoryImplClassName The name of the SCAClientFactory
2747  * Implementation class to load
2748  * @return The specified SCAClientFactory Implementation class
2749  * @throws ServiceRuntimeException Failed to load the SCAClientFactory
2750  * Implementation class
2751  */
2752 private static Class<? extends SCAClientFactory>
2753 loadProviderFactoryClass(String factoryImplClassName,
2754 ClassLoader classLoader)
2755 throws ServiceRuntimeException {
2756
2757     try {
2758         final Class<?> providerClass =
2759             classLoader.loadClass(factoryImplClassName);
2760         final Class<? extends SCAClientFactory> providerFactoryClass =
2761             providerClass.asSubclass(SCAClientFactory.class);
2762         return providerFactoryClass;
2763     } catch (ClassNotFoundException ex) {
2764         throw new ServiceRuntimeException(
2765             "Failed to load SCAClientFactory implementation class "
2766             + factoryImplClassName, ex);
2767     } catch (ClassCastException ex) {
2768         throw new ServiceRuntimeException(
2769             "Loaded SCAClientFactory implementation class "
2770             + factoryImplClassName
2771             + " is not a subclass of "
2772             + SCAClientFactory.class.getName() , ex);
2773     }
2774 }
2775
2776 /**
2777  * Instantiate an instance of the specified SCAClientFactorySPI
2778  * Implementation class.
2779  *
2780  * @param factoryImplClass The SCAClientFactorySPI Implementation
2781  * class to instantiate.
2782  * @return An instance of the SCAClientFactorySPI Implementation class
2783  * @throws ServiceRuntimeException Failed to instantiate the specified
2784  * specified SCAClientFactorySPI Implementation class
2785  */
2786 private static SCAClientFactory
2787 instantiateSCAClientFactoryClass(
2788 Class<? extends SCAClientFactory> factoryImplClass)
2789 throws ServiceRuntimeException {
2790
2791     try {
2792         final SCAClientFactory provider = factoryImplClass.newInstance();
2793         return provider;
2794     } catch (Throwable ex) {
2795         throw new ServiceRuntimeException(
2796             "Failed to instantiate SCAClientFactory implementation class "

```

Deleted: March

```

2797         + factoryImplClass, ex);
2798     }
2799 }
2800
2801 /**
2802  * Utility method for closing Closeable Object.
2803  *
2804  * @param closeable The Object to close.
2805  */
2806 private static void closeStream(Closeable closeable) {
2807     if (closeable != null) {
2808         try{
2809             closeable.close();
2810         } catch (IOException ex) {
2811             throw new ServiceRuntimeException("Failed to close stream",
2812                                             ex);
2813         }
2814     }
2815 }
2816 }
2817
2818
2819

```

Formatted: Bullets and Numbering

## 2820 **B.1.5 SCAClient Classes and Interfaces - what does a vendor need to do?**

2821 The SCAClient classes and interfaces are designed so that vendors can provide their own  
 2822 implementation suited to the needs of their SCA runtime. This section describes the tasks that a vendor  
 2823 needs to consider in relation to the SCAClient classes and interfaces.

- 2824 • Implement their SCAClientFactory and SCAClient implementation classes

2825 Vendors need to provide an implementation of SCAClient that is capable of looking up Services in  
 2826 their SCA Runtime.

2827 Vendors need to subclass SCAClientFactory and implement the createSCAClient() method so  
 2828 that it creates an instance of their SCAClient implementation.

2829

2830

2831

2832

- 2833 • Configure the Vendor Implementation classes so they are used

2834 Vendors have several options:

2835 Option 1: Set System Property to point to the Vendor's implementation

2836

2837 Vendors set the org.oasisopen.sca.client.SCAClientFactory System Property to point to their  
 2838 implementation class and use the reference implementation of SCAClientFactoryFinder

2839

2840 Option 2: Provide a META-INF/services file

2841

2842 Vendors provide a META-INF/services/org.oasisopen.sca.client.SCAClientFactory file that points  
 2843 to their implementation class and use the reference implementation of SCAClientFactoryFinder

2844

2845 Option 3: Inject a vendor implementation of the SCAClientFactoryFinder interface into  
 2846 SCAClientFactory

2847

2848 Vendors inject an instance into the factoryFinder field of SCAClientFactory. The reference  
 2849

Deleted: March

2850  
2851

implementation of 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.

**Deleted:** March

2852

## C. Conformance Items

2853 This section contains a list of conformance items for the SCA Java Common Annotations and APIs  
2854 specification.

2855

Conformance ID	Description
[JCA20001]	Remotable Services MUST NOT make use of <b>method overloading</b> .
[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]

Deleted: March

- [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]

Deleted: March

[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.

2856

Deleted: March

---

2857 **D. Acknowledgements**

2858 The following individuals have participated in the creation of this specification and are gratefully  
2859 acknowledged:

2860 **Participants:**

2861 [Participant Name, Affiliation | Individual Member]

2862 [Participant Name, Affiliation | Individual Member]

2863

**Deleted:** March

---

## E. Non-Normative Text

**Deleted:** March

2865

## F. Revision History

2866 [optional; should not be included in OASIS Standards]

2867

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

Deleted: March

			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

2868



<b>Page 51: [1] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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 &lt;reference/&gt; 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.</p>		
<b>Page 51: [2] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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 &lt;reference/&gt; 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.</p>		
<b>Page 51: [3] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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 &lt;reference/&gt; 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.</p>		
<b>Page 51: [4] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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</p>		
<b>Page 51: [5] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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</p>		
<b>Page 52: [6] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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.</p>		
<b>Page 52: [7] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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.</p>		
<b>Page 52: [8] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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.</p>		
<b>Page 52: [9] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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.</p>		
<b>Page 52: [10] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>
<p>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.</p>		
<b>Page 52: [11] Deleted</b>	<b>Mike Edwards</b>	<b>6/23/2009 9:55:00 AM</b>

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 52: [12] Deleted** **Mike Edwards** **6/23/2009 9:55:00 AM**

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 52: [13] Deleted** **Mike Edwards** **6/23/2009 9:55:00 AM**

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 52: [14] Deleted** **Mike Edwards** **6/23/2009 9:55:00 AM**

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 52: [15] Deleted** **Mike Edwards** **6/23/2009 9:55:00 AM**

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 52: [16] Deleted** **Mike Edwards** **6/23/2009 9:55:00 AM**

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 52: [17] Deleted** **Mike Edwards** **6/23/2009 9:55:00 AM**

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.