



Service Component Architecture Java Common Annotations and APIs Specification Version 1.1

Committee Draft 02, Revision 02 +Issue130

08 February 2009

Specification URIs:

This Version:

<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev2.html>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev2.doc>
<http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-cd02-rev2.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

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 choose to implement their own mappings of assembly model concepts using native APIs and idioms when appropriate.

Status:

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

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

For information on whether any patents have been disclosed that 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	6
1.1	Terminology	6
1.2	Normative References	6
1.3	Non-Normative References	7
2	Implementation Metadata	8
2.1	Service Metadata	8
2.1.1	@Service	8
2.1.2	Java Semantics of a Remotable Service	8
2.1.3	Java Semantics of a Local Service	8
2.1.4	@Reference	9
2.1.5	@Property	9
2.2	Implementation Scopes: @Scope, @Init, @Destroy	9
2.2.1	Stateless scope	9
2.2.2	Composite scope	10
3	Interface	11
3.1	Java interface element – <interface.java>	11
3.2	@Remotable	12
3.3	@Callback	12
4	Client API	13
4.1	Accessing Services from an SCA Component	13
4.1.1	Using the Component Context API	13
4.2	Accessing Services from non-SCA component implementations	13
4.2.1	ComponentContext	13
5	Error Handling	14
6	Asynchronous Programming	15
6.1	@OneWay	15
6.2	Callbacks	15
6.2.1	Using Callbacks	15
6.2.2	Callback Instance Management	17
6.2.3	Implementing Multiple Bidirectional Interfaces	17
6.2.4	Accessing Callbacks	18
7	Policy Annotations for Java	19
7.1	General Intent Annotations	19
7.2	Specific Intent Annotations	21
7.2.1	How to Create Specific Intent Annotations	21
7.3	Application of Intent Annotations	22
7.3.1	Inheritance And Annotation	22
7.4	Relationship of Declarative And Annotated Intents	24
7.5	Policy Set Annotations	24
7.6	Security Policy Annotations	25
7.6.1	Security Interaction Policy	25
7.6.2	Security Implementation Policy	26
8	Java API	29

8.1	Component Context	29
8.2	Request Context	30
8.3	ServiceReference	31
8.4	ServiceRuntimeException.....	31
8.5	ServiceUnavailableException	32
8.6	InvalidServiceException.....	32
8.7	Constants Interface	32
9	Java Annotations	33
9.1	@AllowsPassByReference	33
9.2	@Authentication	34
9.3	@Callback	34
9.4	@ComponentName	35
9.5	@Confidentiality.....	36
9.6	@Constructor.....	37
9.7	@Context.....	37
9.8	@Destroy.....	38
9.9	@EagerInit.....	39
9.10	@Init.....	39
9.11	@Integrity	40
9.12	@Intent	40
9.13	@OneWay	41
9.14	@PolicySet	42
9.15	@Property.....	42
9.16	@Qualifier.....	44
9.17	@Reference.....	44
	9.17.1 Reinjection.....	47
9.18	@Remotable.....	48
9.19	@Requires.....	49
9.20	@Scope	50
9.21	@Service	51
10	WSDL to Java and Java to WSDL	53
10.1	JAX-WS Client Asynchronous API for a Synchronous Service.....	53
A.	XML Schema: sca-interface-java.xsd.....	55
B.	Conformance Items	56
C.	Acknowledgements	57
D.	Non-Normative Text	58
E.	Revision History.....	59

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, client and implementation model defined in this specification are designed to be used by other SCA Java-related specifications in either a partial or complete fashion.

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

1.1 Terminology

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

1.2 Normative References

- | | |
|------------|---|
| [RFC2119] | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , http://www.ietf.org/rfc/rfc2119.txt , IETF RFC 2119, March 1997. |
| [ASSEMBLY] | SCA Assembly Specification, http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd01.pdf |
| [SDO] | SDO 2.1 Specification, http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf |
| [JAX-B] | JAXB 2.1 Specification, http://www.jcp.org/en/jsr/detail?id=222 |
| [WSDL] | WSDL Specification,
WSDL 1.1: http://www.w3.org/TR/wsdl ,
WSDL 2.0: http://www.w3.org/TR/wsdl20/ |
| [POLICY] | SCA Policy Framework, http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-01.pdf |

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

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

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

67 2.1.2 Java Semantics of a Remotable Service

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

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

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

79 2.1.3 Java Semantics of a Local Service

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

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

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

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

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

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

95 2.1.4 @Reference

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

99 2.1.5 @Property

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

103 2.2 Implementation Scopes: @Scope, @Init, @Destroy

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

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

110 This document defines two scopes:

- 111 • STATELESS
- 112 • COMPOSITE

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

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

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

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

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

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

```
125  
126     @Init  
127     public void start() {  
128         ...  
129     }  
130  
131     @Destroy  
132     public void stop() {  
133         ...  
134     }  
135
```

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

138 2.2.1 Stateless scope

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

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

147 **2.2.2 Composite scope**

148 All service requests are dispatched to the same implementation instance for the lifetime of the
149 containing composite. 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 A composite scoped implementation can also specify eager initialization using the *@EagerInit*
152 annotation. When marked for eager initialization, the composite scoped instance is created when
153 its containing component is started. If a method is marked with the @Init annotation, it is called
154 when the instance is created.

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 it MUST NOT perform any synchronization.

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
163 optionally identifies a callback interface, where the first Java interface represents the forward
164 (service) call interface and the second interface represents the interface used to call back from the
165 service to the client.

166
167 The following is the pseudo-schema for the interface.java element

```
168  
169 <interface.java interface="NCName" callbackInterface="NCName"? />  
170
```

171 The interface.java element has the following attributes:

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

177
178 The following snippet shows an example of the Java interface element:

```
179  
180 <interface.java interface="services.stockquote.StockQuoteService"  
181 callbackInterface="services.stockquote.StockQuoteServiceCallback"/>  
182
```

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

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

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

197
198

199 3.2 @Remotable

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

204 3.3 @Callback

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

209 4 Client API

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

212 4.1 Accessing Services from an SCA Component

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

217 4.1.1 Using the Component Context API

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

221 4.2 Accessing Services from non-SCA component implementations

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

224 4.2.1 ComponentContext

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

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

229

```
230 ComponentContext context = // obtained via host environment-specific means  
231 HelloService helloService =  
232     context.getService(HelloService.class, "HelloService");  
233 String result = helloService.hello("Hello World!");
```

234

5 Error Handling

235

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

236

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

237

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

238

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

239

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

240

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

241 6 Asynchronous Programming

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

- 249 • support for non-blocking method calls
- 250 • callbacks

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

252 6.1 @OneWay

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

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

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

263 6.2 Callbacks

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

- 268 • an interface for the provided service
- 269 • a callback interface that must be provided by the client

270 Callbacks can be used for both remotable and local services. Either both interfaces of a
271 bidirectional service must be remotable, or both must be local. It is illegal to mix the two.

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

276 6.2.1 Using Callbacks

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

282 The following example shows a scenario in which bidirectional interfaces and callbacks could be
283 used. A client requests a quotation from a supplier. To process the enquiry and return the
284 quotation, some suppliers might need additional information from the client. The client does not

285 know which additional items of information will be needed by different suppliers. This interaction
286 can be modeled as a bidirectional interface with callback requests to obtain the additional
287 information.

```
288 package somepackage;
289 import org.osoa.sca.annotation.Callback;
290 import org.osoa.sca.annotation.Remotable;
291 @Remotable
292 @Callback(QuotationCallback.class)
293 public interface Quotation {
294     double requestQuotation(String productCode, int quantity);
295 }
296
297 @Remotable
298 public interface QuotationCallback {
299     String getState();
300     String getZipCode();
301     String getCreditRating();
302 }
303
```

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

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

```
313 @Callback
314 protected QuotationCallback callback;
315
316 public double requestQuotation(String productCode, int quantity) {
317     double price = getPrice(productCode, quantity);
318     double discount = 0;
319     if (quantity > 1000 && callback.getState().equals("FL")) {
320         discount = 0.05;
321     }
322     if (quantity > 10000 && callback.getCreditRating().charAt(0) == 'A') {
323         discount += 0.05;
324     }
325     return price * (1-discount);
326 }
327
328
```

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

```
332 public class ClientImpl implements ClientService, QuotationCallback {
333
334     private QuotationService myService;
335
336     @Reference
337     public void setMyService(QuotationService service) {
338         myService = service;
339     }
340
```



```

341
342     public void aClientMethod() {
343         ...
344         double quote = myService.requestQuotation("AB123", 2000);
345         ...
346     }
347
348     public String getState() {
349         return "TX";
350     }
351     public String getZipCode() {
352         return "78746";
353     }
354     public String getCreditRating() {
355         return "AA";
356     }
357 }

```

358
359 In this example the callback is *stateless*, i.e., the callback requests do not need any information
360 relating to the original service request. For a callback that needs information relating to the
361 original service request (a *stateful* callback), this information can be passed to the client by the
362 service provider as parameters on the callback request..

363 6.2.2 Callback Instance Management

364 Instance management for callback requests received by the client of the bidirectional service is
365 handled in the same way as instance management for regular service requests. If the client
366 implementation has STATELESS scope, the callback is dispatched using a newly initialized
367 instance. If the client implementation has COMPOSITE scope, the callback is dispatched using the
368 same shared instance that is used to dispatch regular service requests.

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

375 6.2.3 Implementing Multiple Bidirectional Interfaces

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

```

382
383     @Callback
384     protected MyService1Callback callback1;
385
386     @Callback
387     protected MyService2Callback callback2;

```

388
389 If a single callback has a type that is compatible with multiple declared callback fields, then all of
390 them will be set.

391 6.2.4 Accessing Callbacks

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

395 A reference implementing the callback service interface can be obtained using
396 `ServiceReference.getService()`.
397

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

```
399 @Callback  
400 protected ServiceReference<MyCallback> callback;  
401  
402 public void someMethod() {  
403     MyCallback myCallback = callback.getCallback();    ...  
404     myCallback.receiveResult(theResult);  
405 }  
406  
407  
408  
409
```

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

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

```
416 public void someMethod() {  
417     MyCallback myCallback =  
418         ComponentContext.getRequestContext().getCallback();  
419     ...  
420     myCallback.receiveResult(theResult);  
421 }  
422  
423  
424  
425
```

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

```
428 @Context  
429 protected RequestContext requestContext;  
430  
431 void receiveResult(Object theResult) {  
432     Object refParams =  
433         requestContext.getServiceReference().getCallbackID();  
434     ...  
435 }  
436
```

437 This is necessary if the service implementation has COMPOSITE scope, because callback injection
438 is not performed for composite-scoped implementations.
439

440 7 Policy Annotations for Java

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

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

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

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

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

467

468 7.1 General Intent Annotations

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

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

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

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

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

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

```
481 public static final String SCA_PREFIX=  
482     "{http://docs.oasis-open.org/ns/opencsa/sca/200712}";  
483 public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";  
484 public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";  
485
```

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

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

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

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

494

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

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

```
497     package com.foo;
498
499     import static org.oasisopen.sca.annotation.Confidentiality.*;
500     import static org.oasisopen.sca.annotation.Reference;
501     import static org.oasisopen.sca.annotation.Requires;
502
503     public class Foo {
504         @Requires(CONFIDENTIALITY)
505         @Reference
506         public void setBar(Bar bar) {
507             ...
508         }
509     }
510
```

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

```
514     package com.foo;
515
516     import static org.oasisopen.sca.Constants.*;
517     import static org.oasisopen.sca.annotation.Reference;
518     import static org.oasisopen.sca.annotation.Requires;
519
520     public class Foo {
521         @Requires(SCA_PREFIX+"confidentiality")
522         @Reference
523         public void setBar(Bar bar) {
524             ...
525         }
526     }
527
```

528 The formal syntax for the @Requires annotation follows:

```
529     @Requires( "qualifiedIntent" (, "qualifiedIntent")* )
```

530 where

```
531     qualifiedIntent ::= QName(.qualifier)*
```

532

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

534 7.2 Specific Intent Annotations

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

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

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

```
546 @Integrity
```

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

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

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

552 The general form of specific intent annotations is:

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

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

```
555 Intent      ::= NCName  
556 qualifiers  ::= "qualifier" (, "qualifier")*  
557 qualifier   ::= NCName(qualifier)?  
558
```

559 7.2.1 How to Create Specific Intent Annotations

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

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

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

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

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

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

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

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

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

581 See section @Qualifier for the formal definition of the @Qualifier annotation.

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

584 7.3 Application of Intent Annotations

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

- 586 • Java class
- 587 • Java interface
- 588 • Method
- 589 • Field
- 590 • Constructor parameter

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

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

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

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

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

603 7.3.1 Inheritance And Annotation

604 The inheritance rules for annotations are consistent with the common annotation specification, JSR
605 250.

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

```
608 package services.hello;  
609 import org.oasisopen.sca.annotation.Remotable;  
610 import org.oasisopen.sca.annotation.Integrity;  
611 import org.oasisopen.sca.annotation.Authentication;  
612  
613 @Integrity("transport")  
614 @Authentication  
615 public class HelloService {  
616     @Integrity  
617     @Authentication("message")  
618     public String hello(String message) {...}  
619  
620     @Integrity  
621     @Authentication("transport")  
622     public String helloThere() {...}  
623 }  
624  
625 package services.hello;  
626 import org.oasisopen.sca.annotation.Remotable;  
627 import org.oasisopen.sca.annotation.Confidentiality;  
628 import org.oasisopen.sca.annotation.Authentication;
```

```

629
630     @Confidentiality("message")
631     public class HelloChildService extends HelloService {
632         @Confidentiality("transport")
633         public String hello(String message) {...}
634         @Authentication
635         String helloWorld() {...}
636     }

```

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

638

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

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

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

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

647

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

651

```

652 <?xml version="1.0" encoding="ASCII"?>
653 <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712"
654           name="HelloServiceComposite" >
655     <service name="HelloService" requires="integrity/transport
656           authentication">
657       ...
658     </service>
659     <service name="HelloChildService" requires="integrity/transport
660           authentication confidentiality/message">
661       ...
662     </service>
663     ...
664
665     <component name="HelloServiceComponent">*
666       <implementation.java class="services.hello.HelloService"/>
667       <operation name="hello" requires="integrity
668             authentication/message"/>
669       <operation name="helloThere"
670             requires="integrity
671             authentication/transport"/>
672     </component>
673     <component name="HelloChildServiceComponent">*
674       <implementation.java
675             class="services.hello.HelloChildService" />
676       <operation name="hello"
677             requires="confidentiality/transport"/>
678       <operation name="helloThere" requires="integrity/transport
679             authentication"/>
680       <operation name="helloWorld" requires="authentication"/>
681     </component>
682

```

```
683     ...
684
685     </composite>
```

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

688

689 7.4 Relationship of Declarative And Annotated Intents

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

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

697 7.5 Policy Set Annotations

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

701 Policy Sets can be applied directly to Java implementations using the *@PolicySets* annotation.
702 The @PolicySets annotation either takes the QName of a single policy set as a string or the name
703 of two or more policy sets as an array of strings:
704

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

706

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

708 An example of the @PolicySets annotation:

709

```
710     @Reference(name="helloService", required=true)
711     @PolicySets({ MY_NS + "WS_Encryption_Policy",
712                MY_NS + "WS_Authentication_Policy" })
713     public setHelloService(HelloService service) {
714         ...
715     }
716
```

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

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

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

- 722 • Java class
- 723 • Java interface
- 724 • Method
- 725 • Field
- 726 • Constructor parameter

727 7.6 Security Policy Annotations

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

730 7.6.1 Security Interaction Policy

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

- 733 • @Integrity
- 734 • @Confidentiality
- 735 • @Authentication

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

- 737 • message
- 738 • transport

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

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

```
744
745 package services.hello;
746 //Interface for HelloService
747 public interface HelloService {
748     String hello(String helloMsg);
749 }
750
751 package services.client;
752 // Interface for ClientService
753 public interface ClientService {
754     public void clientMethod();
755 }
756
757 // Implementation class for ClientService
758 package services.client;
759
760 import services.hello.HelloService;
761 import org.oasisopen.sca.annotation.*;
762
763 @Service(ClientService.class)
764 public class ClientServiceImpl implements ClientService {
765
766     private HelloService helloService;
767
768     @Reference(name="helloService", required=true)
769     @Integrity("message")
770     @Authentication("message")
771     public void setHelloService(HelloService service) {
772         helloService = service;
773     }
774
775     public void clientMethod() {
776         String result = helloService.hello("Hello World!");
```

```
777     ...
778     }
779 }
```

780
781 Example 1. Usage of annotated intents on a reference.

782 7.6.2 Security Implementation Policy

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

- 786 • RunAs

787 Takes as a parameter a string which is the name of a Security role.
788 eg. @RunAs("Manager")
789

- 790 • Code marked with this annotation will execute with the Security permissions of the
791 identified role.

- 792 • RolesAllowed

793 Takes as a parameter a single string or an array of strings which represent one or more
794 role names. When present, the implementation can only be accessed by principals whose
795 role corresponds to one of the role names listed in the @roles attribute. How role names
796 are mapped to security principals is implementation dependent (SCA does not define this).
797 eg. @RolesAllowed({"Manager", "Employee"})
798

- 799 • PermitAll

800 No parameters. When present, grants access to all roles.
801

- 802 • DenyAll

803 No parameters. When present, denies access to all roles.
804

- 805 • DeclareRoles

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

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

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

811 7.6.2.1 Annotated Implementation Policy Example

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

```
813
814 package services.account;
815 @Remotable
816 public interface AccountService {
817     AccountReport getAccountReport(String customerID);
818     float fromUSDollarToCurrency(float value);
819 }
```

820
821 The following is a full listing of the AccountServiceImpl class, showing the Service it implements,
822 plus the service references it makes and the settable properties that it has, along with a set of
823 implementation policy annotations:

824

```

825 package services.account;
826 import java.util.List;
827 import commonj.sdo.DataFactory;
828 import org.oasisopen.sca.annotation.Context;
829 import org.oasisopen.sca.annotation.Property;
830 import org.oasisopen.sca.annotation.Reference;
831 import org.oasisopen.sca.annotation.RolesAllowed;
832 import org.oasisopen.sca.annotation.RunAs;
833 import org.oasisopen.sca.annotation.PermitAll;
834 import services.accountdata.AccountDataService;
835 import services.accountdata.CheckingAccount;
836 import services.accountdata.SavingsAccount;
837 import services.accountdata.StockAccount;
838 import services.stockquote.StockQuoteService;
839 @RolesAllowed("customers")
840 @RunAs("accountants")
841 public class AccountServiceImpl implements AccountService {
842
843     @Context
844     protected ComponentContext context;
845
846     @Property
847     protected String currency = "USD";
848
849     @Reference
850     protected AccountDataService accountDataService;
851
852     @Reference
853     protected StockQuoteService stockQuoteService;
854
855     @RolesAllowed({"customers", "accountants"})
856     public AccountReport getAccountReport(String customerID) {
857
858         AccountService accountService =
859         context.createSelfReference(AccountService.class);
860
861         DataFactory dataFactory = DataFactory.INSTANCE;
862         AccountReport accountReport =
863         (AccountReport) dataFactory.create(AccountReport.class);
864         List accountSummaries = accountReport.getAccountSummaries();
865
866         CheckingAccount checkingAccount =
867         accountDataService.getCheckingAccount(customerID);
868         AccountSummary checkingAccountSummary =
869         (AccountSummary) dataFactory.create(AccountSummary.class);
870
871         checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber()
872 );
873         checkingAccountSummary.setAccountType("checking");
874
875         checkingAccountSummary.setBalance(accountService.fromUSDollarToCurrency
876 (checkingAccount.getBalance()));
877         accountSummaries.add(checkingAccountSummary);
878
879         SavingsAccount savingsAccount =
880         accountDataService.getSavingsAccount(customerID);
881         AccountSummary savingsAccountSummary =
882         (AccountSummary) dataFactory.create(AccountSummary.class);

```

Formatted: Font: Not Bold, Font color: Auto

Formatted: Highlight

Formatted: Font: Not Bold, Font color: Auto

Formatted: Indent: Left: 0"

Formatted: Highlight

Formatted: Highlight

Formatted: Highlight

Formatted: Indent: First line: 0.32"

```

883 _____
884 savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
885 savingsAccountSummary.setAccountType("savings");
886 savingsAccountSummary.setBalance(fromUSDollarToCurrency
887 (savingsAccount.getBalance()));
888 accountSummaries.add(savingsAccountSummary);
889 _____
890 StockAccount stockAccount =
891 accountDataService.getStockAccount(customerID);
892 AccountSummary stockAccountSummary =
893 (AccountSummary) dataFactory.create(AccountSummary.class);
894 stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
895 stockAccountSummary.setAccountType("stock");
896 float balance = (stockQuoteService.getQuote(stockAccount.getSymbol())) *
897 stockAccount.getQuantity();
898 stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
899 accountSummaries.add(stockAccountSummary); ...
900
901     return accountReport;
902 }
903
904 @PermitAll
905 public float fromUSDollarToCurrency(float value) {
906
907     if (currency.equals("USD")) return value;
908     if (currency.equals("EURO")) return value * 0.8f;
909     return 0.0f;
910 }
911 }

```

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

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

- 914 • @RolesAllowed("customers") - indicating that customers have access to the
- 915 implementation as a whole
- 916 • @RunAs("accountants") - indicating that the code in the implementation runs with the
- 917 permissions of accountants

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

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

922 8 Java API

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

924 8.1 Component Context

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

```
926
927 package org.oasisopen.sca;
928
929 public interface ComponentContext {
930
931     String getURI();
932
933     <B> B getService(Class<B> businessInterface, String referenceName);
934
935     <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
936                                             String referenceName);
937     <B> Collection<B> getServices(Class<B> businessInterface,
938                                String referenceName);
939
940     <B> Collection<ServiceReference<B>> getServiceReferences(Class<B>
941                                                         businessInterface, String referenceName);
942
943     <B> ServiceReference<B> createSelfReference(Class<B>
944                                               businessInterface);
945
946     <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
947                                               String serviceName);
948
949     <B> B getProperty(Class<B> type, String propertyName);
950
951     <B, R extends ServiceReference<B>> R cast(B target)
952         throws IllegalArgumentException;
953
954     RequestContext getRequestContext();
955
956
957 }
```

- 958
- 959 • **getURI()** - returns the absolute URI of the component within the SCA domain
 - 960 • **getService(Class businessInterface, String referenceName)** - Returns a proxy for
961 the reference defined by the current component. The getService() method takes as its
962 input arguments the Java type used to represent the target service on the client and the
963 name of the service reference. It returns an object providing access to the service. The
964 returned object implements the Java interface the service is typed with. This method
965 MUST throw an IllegalArgumentException if the reference has multiplicity greater than
966 one.
 - 967 • **getServiceReference(Class businessInterface, String referenceName)** - Returns a
968 ServiceReference defined by the current component. This method MUST throw an
969 IllegalArgumentException if the reference has multiplicity greater than one.

- 970 • **getServices(Class businessInterface, String referenceName)** – Returns a list of
971 typed service proxies for a business interface type and a reference name.
- 972 • **getServiceReferences(Class businessInterface, String referenceName)** –Returns a
973 list typed service references for a business interface type and a reference name.
- 974 • **createSelfReference(Class businessInterface)** – Returns a ServiceReference that can
975 be used to invoke this component over the designated service.
- 976 • **createSelfReference(Class businessInterface, String serviceName)** – Returns a
977 ServiceReference that can be used to invoke this component over the designated service.
978 Service name explicitly declares the service name to invoke
- 979 • **getProperty (Class type, String propertyName)** - Returns the value of an SCA
980 property defined by this component.
- 981 • **getRequestContext()** - Returns the context for the current SCA service request, or null if
982 there is no current request or if the context is unavailable. This method MUST return non-
983 null when invoked during the execution of a Java business method for a service operation
984 or callback operation, on the same thread that the SCA runtime provided, and MUST
985 return null in all other cases.
- 986 • **cast(B target)** - Casts a type-safe reference to a ServiceReference

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

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

```
992 private ComponentContext componentContext;
993
994 @Context
995 public void setContext(ComponentContext context) {
996     componentContext = context;
997 }
998
999 public void doSomething() {
1000     HelloWorld service =
1001         componentContext.getService(HelloWorld.class, "HelloWorldComponent");
1002     service.hello("hello");
1003 }
1004
```

1005 Similarly, non-SCA client code can use the ComponentContext API to perform operations against a
1006 component in an SCA domain. How the non-SCA client code obtains a reference to a
1007 ComponentContext is runtime specific.

1008 8.2 Request Context

1009 The following shows the **RequestContext** interface:

```
1010
1011 package org.oasisopen.sca;
1012
1013 import javax.security.auth.Subject;
1014
1015 public interface RequestContext {
1016
1017     Subject getSecuritySubject();
1018
1019     String getServiceName();

```

```

1020     <CB> ServiceReference<CB> getCallbackReference();
1021     <CB> CB getCallback();
1022     <B> ServiceReference<B> getServiceReference();
1023
1024 }
1025

```

1026 The RequestContext interface has the following methods:

- 1027 • **getSecuritySubject()** – Returns the JAAS Subject of the current request
- 1028 • **getServiceName()** – Returns the name of the service on the Java implementation the
1029 request came in on
- 1030 • **getCallbackReference()** – Returns a service reference to the callback as specified by the
1031 caller. This method returns null when called for a service request whose interface is not
1032 bidirectional or when called for a callback request.
- 1033 • **getCallback()** – Returns a proxy for the callback as specified by the caller. Similar to the
1034 getCallbackReference() method, this method returns null when called for a service request
1035 whose interface is not bidirectional or when called for a callback request.
- 1036 • **getServiceReference()** – When invoked during the execution of a service operation, this
1037 method MUST return a ServiceReference that represents the service that was invoked.
1038 When invoked during the execution of a callback operation, this method MUST return a
1039 CallableReference that represents the callback that was invoked.

1040 8.3 ServiceReference

1041 ServiceReferences can be injected using the @Reference annotation on a field, a setter method,
1042 or constructor parameter taking the type ServiceReference. The detailed description of the usage
1043 of these methods is described in the section on Asynchronous Programming in this document.

1044 The following Java code defines the **ServiceReference** interface:

```

1045 package org.oasisopen.sca;
1046
1047 public interface ServiceReference<B> extends java.io.Serializable {
1048
1049     B getService();
1050     Class<B> getBusinessInterface();
1051 }
1052

```

1053 The ServiceReference interface has the following methods:

- 1054
- 1055 • **getService()** - Returns a type-safe reference to the target of this reference. The instance
1056 returned is guaranteed to implement the business interface for this reference. The value
1057 returned is a proxy to the target that implements the business interface associated with this
1058 reference.
- 1059 • **getBusinessInterface()** – Returns the Java class for the business interface associated with
1060 this reference.

1061 8.4 ServiceRuntimeException

1062 The following snippet shows the **ServiceRuntimeException**.

```

1063
1064 package org.oasisopen.sca;
1065
1066 public class ServiceRuntimeException extends RuntimeException {

```

1067 ...
1068 }

1069 This exception signals problems in the management of SCA component execution.
1070

1071 8.5 ServiceUnavailableException

1072 The following snippet shows the *ServiceUnavailableException*.

```
1073 package org.oasisopen.sca;  
1074  
1075 public class ServiceUnavailableException extends ServiceRuntimeException {  
1076     ...  
1077 }  
1078  
1079
```

1080 This exception signals problems in the interaction with remote services. These are exceptions
1081 that can be transient, so retrying is appropriate. Any exception that is a *ServiceRuntimeException*
1082 that is *not* a *ServiceUnavailableException* is unlikely to be resolved by retrying the operation, since
1083 it most likely requires human intervention

1084 8.6 InvalidServiceException

1085 The following snippet shows the *InvalidServiceException*.

```
1086 package org.oasisopen.sca;  
1087  
1088 public class InvalidServiceException extends ServiceRuntimeException {  
1089     ...  
1090 }  
1091  
1092
```

1093 This exception signals that the *ServiceReference* is no longer valid. This can happen when the
1094 target of the reference is undeployed. This exception is not transient and therefore is unlikely to
1095 be resolved by retrying the operation and will most likely require human intervention.

1096 8.7 Constants

1097 The SCA *Constants* interface defines a number of constant values that are used in the SCA Java
1098 APIs and Annotations. The following snippet shows the *Constants* interface:

```
1099 package org.oasisopen.sca;  
1100  
1101 public interface Constants {  
1102     String SCA_NS="http://docs.oasis-open.org/ns/opencsa/sca/200712";  
1103     String SCA_PREFIX = "{"+SCA_NS+"}";  
1104 }  
1105
```


1106

9 Java Annotations

1107

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

1108

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

1109

1110

1111

1112

1113

1114

SCA annotations are not allowed on static methods and 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.

1115

1116

1117

9.1 @AllowsPassByReference

1118

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

1119

1120

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

1121

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

1122

```
import static java.lang.annotation.ElementType.METHOD;
```

1123

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

1124

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

1125

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

1126

1127

```
@Target({TYPE, METHOD})
```

1128

```
@Retention(RUNTIME)
```

1129

```
public @interface AllowsPassByReference {
```

1130

1131

```
}
```

1132

1133

1134

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

1135

1136

1137

1138

1139

1140

1141

1142

`@AllowsPassByReference` has no attributes

1143

1144

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

1145

1146

1147

```
@AllowsPassByReference
```

1148

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

1149

```
    ...
```

1150

```
}
```

1151 9.2 @Authentication

1152 The following Java code defines the **@Authentication** annotation:

```
1153 package org.oasisopen.sca.annotation;
1154
1155 import static java.lang.annotation.ElementType.FIELD;
1156 import static java.lang.annotation.ElementType.METHOD;
1157 import static java.lang.annotation.ElementType.PARAMETER;
1158 import static java.lang.annotation.ElementType.TYPE;
1159 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1160 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1161
1162 import java.lang.annotation.Inherited;
1163 import java.lang.annotation.Retention;
1164 import java.lang.annotation.Target;
1165
1166 @Inherited
1167 @Target({TYPE, FIELD, METHOD, PARAMETER})
1168 @Retention(RUNTIME)
1169 @Intent(Authentication.AUTHENTICATION)
1170 public @interface Authentication {
1171     String AUTHENTICATION = SCA_PREFIX + "authentication";
1172     String AUTHENTICATION_MESSAGE = AUTHENTICATION + ".message";
1173     String AUTHENTICATION_TRANSPORT = AUTHENTICATION + ".transport";
1174
1175     /**
1176      * List of authentication qualifiers (such as "message"
1177      * or "transport").
1178      *
1179      * @return authentication qualifiers
1180      */
1181     @Qualifier
1182     String[] value() default "";
1183 }
1184
```

1185 The **@Authentication** annotation is used to indicate that the invocation requires authentication.
1186 See the [section on Application of Intent Annotations](#) for samples and details.

1187 9.3 @Callback

1188 The following Java code defines shows the **@Callback** annotation:

```
1189 package org.oasisopen.sca.annotation;
1190
1191 import static java.lang.annotation.ElementType.TYPE;
1192 import static java.lang.annotation.ElementType.METHOD;
1193 import static java.lang.annotation.ElementType.FIELD;
1194 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1195 import java.lang.annotation.Retention;
1196 import java.lang.annotation.Target;
1197
1198 @Target(TYPE, METHOD, FIELD)
1199 @Retention(RUNTIME)
1200 public @interface Callback {
1201     Class<?> value() default Void.class;
1202 }
1203
```

1204 }

1205
1206

1207 The @Callback annotation is used to annotate a service interface with a callback interface, which
1208 takes the Java Class object of the callback interface as a parameter.

1209 The @Callback annotation has the following attribute:

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

1211

1212 The @Callback annotation can also be used to annotate a method or a field of an SCA
1213 implementation class, in order to have a callback object injected

1214

1215 The following snippet shows a @Callback annotation on an interface:

1216

```
1217 @Remotable  
1218 @Callback(MyServiceCallback.class)  
1219 public interface MyService {  
1220  
1221     void someAsyncMethod(String arg);  
1222 }  
1223
```

1224 An example use of the @Callback annotation to declare a callback interface follows:

1225

```
1226 package somepackage;  
1227 import org.oasisopen.sca.annotation.Callback;  
1228 import org.oasisopen.sca.annotation.Remotable;  
1229 @Remotable  
1230 @Callback(MyServiceCallback.class)  
1231 public interface MyService {  
1232  
1233     void someMethod(String arg);  
1234 }  
1235  
1236 @Remotable  
1237 public interface MyServiceCallback {  
1238  
1239     void receiveResult(String result);  
1240 }  
1241
```

1241

1242 In this example, the implied component type is:

1243

```
1244 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712" >  
1245     <service name="MyService">  
1246         <interface.java interface="somepackage.MyService"  
1247             callbackInterface="somepackage.MyServiceCallback"/>  
1248     </service>  
1249 </componentType>  
1250
```

1251 9.4 @ComponentName

1252 The following Java code defines the **@ComponentName** annotation:

```

1253
1254 package org.oasisopen.sca.annotation;
1255
1256 import static java.lang.annotation.ElementType.METHOD;
1257 import static java.lang.annotation.ElementType.FIELD;
1258 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1259 import java.lang.annotation.Retention;
1260 import java.lang.annotation.Target;
1261
1262 @Target({METHOD, FIELD})
1263 @Retention(RUNTIME)
1264 public @interface ComponentName {
1265
1266 }
1267

```

1268 The @ComponentName annotation is used to denote a Java class field or setter method that is
1269 used to inject the component name.

1270 The following snippet shows a component name field definition sample.

```

1271
1272 @ComponentName
1273 private String componentName;
1274

```

1275 The following snippet shows a component name setter method sample.

```

1276
1277 @ComponentName
1278 public void setComponentName(String name) {
1279     //...
1280 }

```

1281 9.5 @Confidentiality

1282 The following Java code defines the **@Confidentiality** annotation:

```

1283 package org.oasisopen.sca.annotations;
1284
1285 import static java.lang.annotation.ElementType.FIELD;
1286 import static java.lang.annotation.ElementType.METHOD;
1287 import static java.lang.annotation.ElementType.PARAMETER;
1288 import static java.lang.annotation.ElementType.TYPE;
1289 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1290 import static org.oasisopen.sca.Constants.SCA_PREFIX;
1291
1292 import java.lang.annotation.Inherited;
1293 import java.lang.annotation.Retention;
1294 import java.lang.annotation.Target;
1295
1296 @Inherited
1297 @Target({TYPE, FIELD, METHOD, PARAMETER})
1298 @Retention(RUNTIME)
1299 @Intent(Confidentiality.CONFIDENTIALITY)
1300 public @interface Confidentiality {
1301     String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
1302     String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
1303     String CONFIDENTIALITY_TRANSPORT = CONFIDENTIALITY + ".transport";
1304

```

```

1305
1306     /**
1307      * List of confidentiality qualifiers (such as "message" or
1308      * "transport").
1309      *
1310      * @return confidentiality qualifiers
1311      */
1312     @Qualifier
1313     String[] value() default "";
1314 }

```

1315 The **@Confidentiality** annotation is used to indicate that the invocation requires confidentiality.
1316 See the [section on Application of Intent Annotations](#) for samples and details.

1317 9.6 @Constructor

1318 The following Java code defines the **@Constructor** annotation:

```

1319 package org.oasisopen.sca.annotation;
1320
1321 import static java.lang.annotation.ElementType.CONSTRUCTOR;
1322 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1323 import java.lang.annotation.Retention;
1324 import java.lang.annotation.Target;
1325
1326 @Target (CONSTRUCTOR)
1327 @Retention (RUNTIME)
1328 public @interface Constructor { }
1329
1330

```

1331 The **@Constructor** annotation is used to mark a particular constructor to use when instantiating a
1332 Java component implementation. If this constructor has parameters, each of these parameters
1333 MUST have either a **@Property** annotation or a **@Reference** annotation.

1334 The following snippet shows a sample for the **@Constructor** annotation.

```

1335
1336 public class HelloServiceImpl implements HelloService {
1337
1338     public HelloServiceImpl() {
1339         ...
1340     }
1341
1342     @Constructor
1343     public HelloServiceImpl(@Property(name="someProperty")
1344                             String someProperty ) {
1345         ...
1346     }
1347
1348     public String hello(String message) {
1349         ...
1350     }
1351 }

```

1352 9.7 @Context

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

1354

```

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

```

1369 The @Context annotation is used to denote a Java class field or a setter method that is used to
1370 inject a composite context for the component. The type of context to be injected is defined by the
1371 type of the Java class field or type of the setter method input argument; the type is either
1372 **ComponentContext** or **RequestContext**.

1373 The @Context annotation has no attributes.

1374

1375 The following snippet shows a ComponentContext field definition sample.

1376

```

1377 @Context
1378 protected ComponentContext context;
1379

```

1380 The following snippet shows a RequestContext field definition sample.

1381

```

1382 @Context
1383 protected RequestContext context;

```

1384 9.8 @Destroy

1385 The following Java code defines the **@Destroy** annotation:

1386

```

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

```

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

1403 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1404 when the scope defined for the implementation class ends. If the implementation class has a

1405 method with an `@Destroy` annotation that does not match these criteria, the SCA runtime MUST
1406 NOT instantiate the implementation class.

1407 The following snippet shows a sample for a destroy method definition.

1408

```
1409 @Destroy  
1410 public void myDestroyMethod() {  
1411     ...  
1412 }
```

1413 9.9 @EagerInit

1414 The following Java code defines the `@EagerInit` annotation:

1415

```
1416 package org.oasisopen.sca.annotation;  
1417  
1418 import static java.lang.annotation.ElementType.TYPE;  
1419 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1420 import java.lang.annotation.Retention;  
1421 import java.lang.annotation.Target;  
1422  
1423 @Target(TYPE)  
1424 @Retention(RUNTIME)  
1425 public @interface EagerInit {  
1426  
1427 }  
1428
```

1429 The `@EagerInit` annotation is used to annotate the Java class of a COMPOSITE scoped
1430 implementation for eager initialization. When marked for eager initialization, the composite scoped
1431 instance is created when its containing component is started.

1432 9.10 @Init

1433 The following Java code defines the `@Init` annotation:

1434

```
1435 package org.oasisopen.sca.annotation;  
1436  
1437 import static java.lang.annotation.ElementType.METHOD;  
1438 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1439 import java.lang.annotation.Retention;  
1440 import java.lang.annotation.Target;  
1441  
1442 @Target(METHOD)  
1443 @Retention(RUNTIME)  
1444 public @interface Init {  
1445  
1446 }  
1447  
1448
```

1449 The `@Init` annotation is used to denote a single Java class method that is called when the scope
1450 defined for the implementation class starts. The method MAY have any access modifier and MUST
1451 have a void return type and no arguments.

1452 If there is a method that matches these criteria, the SCA runtime MUST call the annotated method
1453 after all property and reference injection is complete. If the implementation class has a method

1454 with an @Init annotation that does not match these criteria, the SCA runtime MUST NOT
1455 instantiate the implementation class.

1456 The following snippet shows an example of an init method definition.

```
1457  
1458 @Init  
1459 public void myInitMethod() {  
1460     ...  
1461 }
```

1462 9.11 @Integrity

1463 The following Java code defines the **@Integrity** annotation:

```
1464 package org.oasisopen.sca.annotation;  
1465  
1466 import static java.lang.annotation.ElementType.FIELD;  
1467 import static java.lang.annotation.ElementType.METHOD;  
1468 import static java.lang.annotation.ElementType.PARAMETER;  
1469 import static java.lang.annotation.ElementType.TYPE;  
1470 import static java.lang.annotation.RetentionPolicy.RUNTIME;  
1471 import static org.oasisopen.Constants.SCA_PREFIX;  
1472  
1473 import java.lang.annotation.Inherited;  
1474 import java.lang.annotation.Retention;  
1475 import java.lang.annotation.Target;  
1476  
1477 @Inherited  
1478 @Target({TYPE, FIELD, METHOD, PARAMETER})  
1479 @Retention(RUNTIME)  
1480 @Intent(Integrity.INTEGRITY)  
1481 public @interface Integrity {  
1482     String INTEGRITY = SCA_PREFIX + "integrity";  
1483     String INTEGRITY_MESSAGE = INTEGRITY + ".message";  
1484     String INTEGRITY_TRANSPORT = INTEGRITY + ".transport";  
1485  
1486     /**  
1487      * List of integrity qualifiers (such as "message" or "transport").  
1488      *  
1489      * @return integrity qualifiers  
1490      */  
1491     @Qualifier  
1492     String[] value() default "";  
1493 }  
1494  
1495
```

1496 The **@Integrity** annotation is used to indicate that the invocation requires integrity (ie no
1497 tampering of the messages between client and service).

1498 See the [section on Application of Intent Annotations](#) for samples and details.

1499 9.12 @Intent

1500 The following Java code defines the **@Intent** annotation:

```
1501 package org.oasisopen.sca.annotation;  
1502  
1503 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;  
1504
```



```

1505 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1506 import java.lang.annotation.Retention;
1507 import java.lang.annotation.Target;
1508
1509 @Target({ANNOTATION_TYPE})
1510 @Retention(RUNTIME)
1511 public @interface Intent {
1512     /**
1513      * The qualified name of the intent, in the form defined by
1514      * {@link javax.xml.namespace.QName#toString}.
1515      * @return the qualified name of the intent
1516      */
1517     String value() default "";
1518
1519     /**
1520      * The XML namespace for the intent.
1521      * @return the XML namespace for the intent
1522      */
1523     String targetNamespace() default "";
1524
1525     /**
1526      * The name of the intent within its namespace.
1527      * @return name of the intent within its namespace
1528      */
1529     String localPart() default "";
1530 }
1531

```

1532 The @Intent annotation is used for the creation of new annotations for specific intents. It is not
1533 expected that the @Intent annotation will be used in application code.

1534 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
1535 define new intent annotations.

1536 9.13 @OneWay

1537 The following Java code defines the *@OneWay* annotation:

```

1538
1539 package org.oasisopen.sca.annotation;
1540
1541 import static java.lang.annotation.ElementType.METHOD;
1542 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1543 import java.lang.annotation.Retention;
1544 import java.lang.annotation.Target;
1545
1546 @Target(METHOD)
1547 @Retention(RUNTIME)
1548 public @interface OneWay {
1549
1550
1551 }
1552

```

1553 The @OneWay annotation is used on a Java interface or class method to indicate that invocations
1554 will be dispatched in a non-blocking fashion as described in the section on Asynchronous
1555 Programming.

1556 The @OneWay annotation has no attributes.

1557 The following snippet shows the use of the @OneWay annotation on an interface.

```

1558 package services.hello;
1559
1560 import org.oasisopen.sca.annotation.OneWay;
1561
1562 public interface HelloService {
1563     @OneWay
1564     void hello(String name);
1565 }

```

1566 9.14 @PolicySets

1567 The following Java code defines the **@PolicySets** annotation:

```

1568 package org.oasisopen.sca.annotation;
1569
1570 import static java.lang.annotation.ElementType.FIELD;
1571 import static java.lang.annotation.ElementType.METHOD;
1572 import static java.lang.annotation.ElementType.PARAMETER;
1573 import static java.lang.annotation.ElementType.TYPE;
1574 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1575
1576 import java.lang.annotation.Retention;
1577 import java.lang.annotation.Target;
1578
1579 @Target({TYPE, FIELD, METHOD, PARAMETER})
1580 @Retention(RUNTIME)
1581 public @interface PolicySets {
1582     /**
1583      * Returns the policy sets to be applied.
1584      *
1585      * @return the policy sets to be applied
1586      */
1587     String[] value() default "";
1588 }
1589
1590

```

1591 The **@PolicySets** annotation is used to attach one or more SCA Policy Sets to a Java implementation class or to one of its subelements.

1592 See the [section "Policy Set Annotations"](#) for details and samples.

1594 9.15 @Property

1595 The following Java code defines the **@Property** annotation:

```

1596 package org.oasisopen.sca.annotation;
1597
1598 import static java.lang.annotation.ElementType.METHOD;
1599 import static java.lang.annotation.ElementType.FIELD;
1600 import static java.lang.annotation.ElementType.PARAMETER;
1601 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1602 import java.lang.annotation.Retention;
1603 import java.lang.annotation.Target;
1604
1605 @Target({METHOD, FIELD, PARAMETER})
1606 @Retention(RUNTIME)
1607 public @interface Property {
1608     String name() default "";
1609 }

```

```
1610     boolean required() default true;
1611 }
1612
```

1613 The @Property annotation is used to denote a Java class field, a setter method, or a constructor
1614 parameter that is used to inject an SCA property value. The type of the property injected, which
1615 can be a simple Java type or a complex Java type, is defined by the type of the Java class field or
1616 the type of the input parameter of the setter method or constructor.

1617 The @Property annotation can be used on fields, on setter methods or on a constructor method
1618 parameter. However, the @Property annotation MUST NOT be used on a class field that is declared
1619 as final.

1620 Properties can also be injected via setter methods even when the @Property annotation is not
1621 present. However, the @Property annotation must be used in order to inject a property onto a
1622 non-public field. In the case where there is no @Property annotation, the name of the property is
1623 the same as the name of the field or setter.

1624 Where there is both a setter method and a field for a property, the setter method is used.

1625 The @Property annotation has the following attributes:

- 1626 • **name (optional)** – the name of the property. For a field annotation, the default is the
1627 name of the field of the Java class. For a setter method annotation, the default is the
1628 JavaBeans property name [JAVABEANS] corresponding to the setter method name. For a
1629 constructor parameter annotation, there is no default and the name attribute MUST be
1630 present.
- 1631 • **required (optional)** – specifies whether injection is required, defaults to true. For a
1632 constructor parameter annotation, this attribute MUST have the value true.

1633

1634 The following snippet shows a property field definition sample.

1635

```
1636 @Property(name="currency", required=true)
1637 protected String currency;
```

1638

1639 The following snippet shows a property setter sample

1640

```
1641 @Property(name="currency", required=true)
1642 public void setCurrency( String theCurrency ) {
1643     ....
1644 }
```

1645

1646 If the property is defined as an array or as any type that extends or implements
1647 **java.util.Collection**, then the implied component type has a property with a **many** attribute set to
1648 true.

1649 The following snippet shows the definition of a configuration property using the @Property
1650 annotation for a collection.

```
1651 ...
1652 private List<String> helloConfigurationProperty;
1653
1654 @Property(required=true)
1655 public void setHelloConfigurationProperty(List<String> property) {
1656     helloConfigurationProperty = property;
1657 }
```

1658 ...

1659 9.16 @Qualifier

1660 The following Java code defines the **@Qualifier** annotation:

```
1661 package org.oasisopen.sca.annotation;
1662
1663 import static java.lang.annotation.ElementType.METHOD;
1664 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1665
1666 import java.lang.annotation.Retention;
1667 import java.lang.annotation.Target;
1668
1669 @Target(METHOD)
1670 @Retention(RUNTIME)
1671 public @interface Qualifier {
1672 }
1673
1674
```

1675 The @Qualifier annotation is applied to an attribute of a specific intent annotation definition,
1676 defined using the @Intent annotation, to indicate that the attribute provides qualifiers for the
1677 intent. The @Qualifier annotation MUST be used in a specific intent annotation definition where the
1678 intent has qualifiers.

1679 See the [section "How to Create Specific Intent Annotations"](#) for details and samples of how to
1680 define new intent annotations.

1681 9.17 @Reference

1682 The following Java code defines the **@Reference** annotation:

```
1683
1684 package org.oasisopen.sca.annotation;
1685
1686 import static java.lang.annotation.ElementType.METHOD;
1687 import static java.lang.annotation.ElementType.FIELD;
1688 import static java.lang.annotation.ElementType.PARAMETER;
1689 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1690
1691 import java.lang.annotation.Retention;
1692 import java.lang.annotation.Target;
1693 @Target({METHOD, FIELD, PARAMETER})
1694 @Retention(RUNTIME)
1695 public @interface Reference {
1696     String name() default "";
1697     boolean required() default true;
1698 }
1699
```

1700 The @Reference annotation type is used to annotate a Java class field, a setter method, or a
1701 constructor parameter that is used to inject a service that resolves the reference. The interface of
1702 the service injected is defined by the type of the Java class field or the type of the input parameter
1703 of the setter method or constructor.

1704 The @Reference annotation MUST NOT be used on a class field that is declared as final.

1705 References can also be injected via setter methods even when the @Reference annotation is not
1706 present. However, the @Reference annotation must be used in order to inject a reference onto a
1707 non-public field. In the case where there is no @Reference annotation, the name of the reference
1708 is the same as the name of the field or setter.

1709 Where there is both a setter method and a field for a reference, the setter method is used.

1710 The @Reference annotation has the following attributes:

- 1711 • **name (optional)** – the name of the reference. For a field annotation, the default is the
1712 name of the field of the Java class. For a setter method annotation, the default is the
1713 JavaBeans property name corresponding to the setter method name. For a constructor
1714 parameter annotation, there is no default and the name attribute MUST be present.
- 1715 • **required (optional)** – whether injection of service or services is required. Defaults to true.
1716 For a constructor parameter annotation, this attribute MUST have the value true.

1717

1718 The following snippet shows a reference field definition sample.

1719

```
1720 @Reference(name="stockQuote", required=true)  
1721 protected StockQuoteService stockQuote;
```

1722

1723 The following snippet shows a reference setter sample

1724

```
1725 @Reference(name="stockQuote", required=true)  
1726 public void setStockQuote( StockQuoteService theSQService ) {  
1727     ...  
1728 }
```

1729

1730 The following fragment from a component implementation shows a sample of a service reference
1731 using the @Reference annotation. The name of the reference is "helloService" and its type is
1732 HelloService. The clientMethod() calls the "hello" operation of the service referenced by the
1733 helloService reference.

1734

```
1735 package services.hello;  
1736  
1737 private HelloService helloService;  
1738  
1739 @Reference(name="helloService", required=true)  
1740 public setHelloService(HelloService service) {  
1741     helloService = service;  
1742 }  
1743  
1744 public void clientMethod() {  
1745     String result = helloService.hello("Hello World!");  
1746     ...  
1747 }  
1748
```

1749 The presence of a @Reference annotation is reflected in the componentType information that the
1750 runtime generates through reflection on the implementation class. The following snippet shows
1751 the component type for the above component implementation fragment.

1752

```
1753 <?xml version="1.0" encoding="ASCII"?>  
1754 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">  
1755     <!-- Any services offered by the component would be listed here -->  
1756
```

```

1757     <reference name="helloService" multiplicity="1..1">
1758         <interface.java interface="services.hello.HelloService"/>
1759     </reference>
1760
1761 </componentType>
1762

```

1763 If the reference is not an array or collection, then the implied component type has a reference
1764 with a multiplicity of either 0..1 or 1..1 depending on the value of the @Reference **required**
1765 attribute – 1..1 applies if required=true.

1766
1767 If the reference is defined as an array or as any type that extends or implements *java.util.Collection*,
1768 then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending
1769 on whether the **required** attribute of the @Reference annotation is set to true or false – 1..n applies if
1770 required=true.

1771
1772 The following fragment from a component implementation shows a sample of a service reference
1773 definition using the @Reference annotation on a java.util.List. The name of the reference is
1774 "helloServices" and its type is HelloService. The clientMethod() calls the "hello" operation of all the
1775 services referenced by the helloServices reference. In this case, at least one HelloService should
1776 be present, so **required** is true.

```

1777     @Reference(name="helloServices", required=true)
1778     protected List<HelloService> helloServices;
1779
1780     public void clientMethod() {
1781
1782         ...
1783         for (int index = 0; index < helloServices.size(); index++) {
1784             HelloService helloService =
1785                 (HelloService)helloServices.get(index);
1786             String result = helloService.hello("Hello World!");
1787         }
1788         ...
1789     }
1790
1791

```

1792 The following snippet shows the XML representation of the component type reflected from for the
1793 former component implementation fragment. There is no need to author this component type in
1794 this case since it can be reflected from the Java class.

```

1795
1796 <?xml version="1.0" encoding="ASCII"?>
1797 <componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200712">
1798
1799     <!-- Any services offered by the component would be listed here -->
1800     <reference name="helloServices" multiplicity="1..n">
1801         <interface.java interface="services.hello.HelloService"/>
1802     </reference>
1803
1804 </componentType>
1805

```

1806 At runtime, the representation of an unwired reference depends on the reference's multiplicity. An
1807 unwired reference with a multiplicity of 0..1 must be null. An unwired reference with a multiplicity
1808 of 0..N must be an empty array or collection.

1809 **9.17.1 Reinjection**

1810 References MAY be reinjected after the initial creation of a component if the reference target
 1811 changes due to a change in wiring that has occurred since the component was initialized. In order
 1812 for reinjection to occur, the following MUST be true:

- 1813 1. The component MUST NOT be STATELESS scoped.
- 1814 2. The reference MUST use either field-based injection or setter injection. References that are
 1815 injected through constructor injection MUST NOT be changed. Setter injection allows for
 1816 code in the setter method to perform processing in reaction to a change.

1817 If a reference target changes and the reference is not reinjected, the reference MUST continue to
 1818 work as if the reference target was not changed.

1819 If an operation is called on a reference where the target of that reference has been undeployed,
 1820 the SCA runtime SHOULD throw InvalidServiceException. If an operation is called on a reference
 1821 where the target of the reference has become unavailable for some reason, the SCA runtime
 1822 SHOULD throw ServiceUnavailableException. If the target of the reference is changed, the
 1823 reference MAY continue to work, depending on the runtime and the type of change that was made.
 1824 If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.

1825 A ServiceReference that has been obtained from a reference by ComponentContext.cast()
 1826 corresponds to the reference that is passed as a parameter to cast(). If the reference is
 1827 subsequently reinjected, the ServiceReference obtained from the original reference MUST continue
 1828 to work as if the reference target was not changed. If the target of a ServiceReference has been
 1829 undeployed, the SCA runtime SHOULD throw InvalidServiceException when an operation is
 1830 invoked on the ServiceReference. If the target of a ServiceReference has become unavailable, the
 1831 SCA runtime SHOULD throw ServiceUnavailableException when an operation is invoked on the
 1832 ServiceReference. If the target of a ServiceReference is changed, the reference MAY continue to
 1833 work, depending on the runtime and the type of change that was made. If it doesn't work, the
 1834 exception thrown will depend on the runtime and the cause of the failure.

1835 A reference or ServiceReference accessed through the component context by calling getService()
 1836 or getServiceReference() MUST correspond to the current configuration of the domain. This
 1837 applies whether or not reinjection has taken place. If the target has been undeployed or has
 1838 become unavailable, the result SHOULD be a reference to the undeployed or unavailable service,
 1839 and attempts to call business methods SHOULD throw an exception as described above. If the
 1840 target has changed, the result SHOULD be a reference to the changed service.

1841 The rules for reference reinjection also apply to references with a multiplicity of 0..N or 1..N. This
 1842 means that in the cases listed above where reference reinjection is not allowed, the array or
 1843 Collection for the reference MUST NOT change its contents. In cases where the contents of a
 1844 reference collection MAY change, then for references that use setter injection, the setter method
 1845 MUST be called for any change to the contents. The reinjected array or Collection MUST NOT be
 1846 the same array or Collection object previously injected to the component.

1847

	Effect on		
Change event	Reference	Existing ServiceReference Object	Subsequent invocations of ComponentContext.getServiceReference() or getService()
Change to the target of the reference	MAY be reinjected (if other conditions* apply). If not reinjected, then it MUST continue to work as if the reference target was not changed.	MUST continue to work as if the reference target was not changed.	Result corresponds to the current configuration of the domain.
Target service	Business methods SHOULD throw	Business methods SHOULD throw	Result SHOULD be a reference to the undeployed

undeployed	InvalidServiceException.	InvalidServiceException.	or unavailable service. Business methods SHOULD throw InvalidServiceException.
Target service changed	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	MAY continue to work, depending on the runtime and the type of change that was made. If it doesn't work, the exception thrown will depend on the runtime and the cause of the failure.	Result SHOULD be a reference to the changed service.
<p>* Other conditions:</p> <ol style="list-style-type: none"> 1. The component MUST NOT be STATELESS scoped. 2. The reference MUST use either field-based injection or setter injection. References that are injected through constructor injection MUST NOT be changed. <p>** Result of invoking ComponentContext.cast() corresponds to the reference that is passed as a parameter to cast().</p>			

1848

1849

9.18 @Remotable

1850

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

1851

1852

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

1853

1854

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

1855

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

1856

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

1857

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

1858

1859

1860

```
@Target (TYPE)
```

1861

```
@Retention (RUNTIME)
```

1862

```
public @interface Remotable {
```

1863

1864

```
}
```

1865

1866

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.

1867

1868

The @Remotable annotation has no attributes.

1869

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

1870

1871

```
package services.hello;
```

1872

1873

```
import org.oasisopen.sca.annotation.*;
```

1874

1875

```
@Remotable
```

1876

```
public interface HelloService {
```

1877

```
    String hello(String message);
```

1878

1879 }

1880

1881 The style of remotable interfaces is typically **coarse grained** and intended for **loosely coupled**

1882 interactions. Remotable service interfaces are not allowed to make use of method **overloading**.

1883 Complex data types exchanged via remotable service interfaces MUST be compatible with the

1884 marshalling technology used by the service binding. For example, if the service is going to be

1885 exposed using the standard Web Service binding, then the parameters MAY be JAXB [JAX-B] types

1886 or Service Data Objects (SDOs) [SDO].

1887 Independent of whether the remotable service is called from outside of the composite that

1888 contains it or from another component in the same composite, the data exchange semantics are

1889 **by-value**.

1890 Implementations of remotable services can modify input data during or after an invocation and

1891 can modify return data after the invocation. If a remotable service is called locally or remotely, the

1892 SCA container is responsible for making sure that no modification of input data or post-invocation

1893 modifications to return data are seen by the caller.

1894 The following snippet shows a remotable Java service interface.

```
1895
1896 package services.hello;
1897
1898 import org.oasisopen.sca.annotation.*;
1899
1900 @Remotable
1901 public interface HelloService {
1902
1903     String hello(String message);
1904 }
1905
1906 package services.hello;
1907
1908 import org.oasisopen.sca.annotation.*;
1909
1910 @Service(HelloService.class)
1911 public class HelloServiceImpl implements HelloService {
1912
1913     public String hello(String message) {
1914         ...
1915     }
1916 }
```

1917 9.19 @Requires

1918 The following Java code defines the **@Requires** annotation:

```
1919 package org.oasisopen.sca.annotation;
1920
1921 import static java.lang.annotation.ElementType.FIELD;
1922 import static java.lang.annotation.ElementType.METHOD;
1923 import static java.lang.annotation.ElementType.PARAMETER;
1924 import static java.lang.annotation.ElementType.TYPE;
1925 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1926
1927 import java.lang.annotation.Inherited;
1928 import java.lang.annotation.Retention;
1929 import java.lang.annotation.Target;
```

```

1931
1932 @Inherited
1933 @Retention(RUNTIME)
1934 @Target({TYPE, METHOD, FIELD, PARAMETER})
1935 public @interface Requires {
1936     /**
1937      * Returns the attached intents.
1938      *
1939      * @return the attached intents
1940      */
1941     String[] value() default "";
1942 }

```

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

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

1947 9.20 @Scope

1948 The following Java code defines the **@Scope** annotation:

```

1949 package org.oasisopen.sca.annotation;
1950
1951 import static java.lang.annotation.ElementType.TYPE;
1952 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1953 import java.lang.annotation.Retention;
1954 import java.lang.annotation.Target;
1955
1956 @Target(TYPE)
1957 @Retention(RUNTIME)
1958 public @interface Scope {
1959     String value() default "STATELESS";
1960 }

```

1962 The @Scope annotation MUST only be used on a service's implementation class. It is an error to use this annotation on an interface.

1964 The @Scope annotation has the following attribute:

- 1965 • **value** – the name of the scope.
- 1966 For 'STATELESS' implementations, a different implementation instance can be used to
- 1967 service each request. Implementation instances can be newly created or be drawn from a
- 1968 pool of instances.
- 1969 SCA defines the following scope names, but others can be defined by particular Java-
- 1970 based implementation types:
- 1971 STATELESS
- 1972 COMPOSITE

1973 The default value is STATELESS.

1974 The following snippet shows a sample for a COMPOSITE scoped service implementation:

```

1975 package services.hello;
1976
1977 import org.oasisopen.sca.annotation.*;
1978
1979 @Service(HelloService.class)
1980 @Scope("COMPOSITE")
1981 public class HelloServiceImpl implements HelloService {
1982     public String hello(String message) {

```

```
1984     ...
1985     }
1986 }
1987
```

1988 9.21 @Service

1989 The following Java code defines the **@Service** annotation:

```
1990 package org.oasisopen.sca.annotation;
1991
1992 import static java.lang.annotation.ElementType.TYPE;
1993 import static java.lang.annotation.RetentionPolicy.RUNTIME;
1994 import java.lang.annotation.Retention;
1995 import java.lang.annotation.Target;
1996
1997 @Target (TYPE)
1998 @Retention (RUNTIME)
1999 public @interface Service {
2000
2001     Class<?>[] interfaces() default {};
2002     Class<?> value() default Void.class;
2003 }
2004
```

2005 The @Service annotation is used on a component implementation class to specify the SCA services
2006 offered by the implementation. The class need not be declared as implementing all of the
2007 interfaces implied by the services, but all methods of the service interfaces must be present. A
2008 class used as the implementation of a service is not required to have a @Service annotation. If a
2009 class has no @Service annotation, then the rules determining which services are offered and what
2010 interfaces those services have are determined by the specific implementation type.

2011 The @Service annotation has the following attributes:

- 2012 • **interfaces** – The value is an array of interface or class objects that should be exposed as
2013 services by this component.
- 2014 • **value** – A shortcut for the case when the class provides only a single service interface.

2015 Only one of these attributes should be specified.

2016
2017 A @Service annotation with no attributes is meaningless, it is the same as not having the
2018 annotation there at all.

2019 The **service names** of the defined services default to the names of the interfaces or class, without
2020 the package name.

2021 A component MUST NOT have two services with the same Java simple name. If a Java
2022 implementation needs to realize two services with the same Java simple name then this can be
2023 achieved through subclassing of the interface.

2024 The following snippet shows an implementation of the HelloService marked with the @Service
2025 annotation.

```
2026 package services.hello;
2027
2028 import org.oasisopen.sca.annotation.Service;
2029
2030 @Service(HelloService.class)
2031 public class HelloServiceImpl implements HelloService {
2032
2033     public void hello(String name) {
```

```
2034         System.out.println("Hello " + name);
2035     }
2036 }
2037
```

2038

10 WSDL to Java and Java to WSDL

2039 The SCA Client and Implementation Model for Java applies the WSDL to Java and Java to WSDL
2040 mapping rules as defined by the JAX-WS specification [JAX-WS] for generating remotable Java
2041 interfaces from WSDL portTypes and vice versa.

2042 For the purposes of the Java-to-WSDL mapping algorithm, the interface is treated as if it had a
2043 @WebService annotation on the class, even if it doesn't, and the
2044 @org.oasisopen.sca.annotation.OneWay annotation should be treated as a synonym for the
2045 @javax.jws.OneWay annotation. For the WSDL-to-Java mapping, the generated @WebService
2046 annotation implies that the interface is @Remotable.

2047 For the mapping from Java types to XML schema types, SCA permits both the JAXB 2.1 [JAX-B]
2048 mapping and the SDO 2.1 [SDO] mapping. SCA runtimes MUST support the JAXB 2.1 mapping
2049 and MAY support the SDO 2.1 mapping. Having a choice of binding technologies is allowed, as
2050 noted in the first paragraph of section 5 of the JSR 181 (version 2) specification, which is
2051 referenced by the JAX-WS specification.

2052 The JAX-WS mappings are applied with the following restrictions:

- 2053 • No support for holders

2054

2055 **Note:** This specification needs more examples and discussion of how JAX-WS's client asynchronous
2056 model is used.

2057 10.1 JAX-WS Client Asynchronous API for a Synchronous Service

2058 The JAX-WS specification defines a mapping of a synchronous service invocation, which provides a client
2059 application with a means of invoking that service asynchronously, so that the client can invoke a service
2060 operation and proceed to do other work without waiting for the service operation to complete its
2061 processing. The client application can retrieve the results of the service either through a polling
2062 mechanism or via a callback method which is invoked when the operation completes.

2063 For SCA reference interfaces defined using interface.java, the Java interface MAY contain the additional
2064 client-side asynchronous polling and callback methods defined by JAX-WS. For SCA service interfaces
2065 defined using interface.java, the Java interface MUST NOT contain these methods. If these methods are
2066 present, SCA Runtimes MUST NOT include them in the SCA reference interface as defined by the
2067 Assembly specification. These methods are recognized as follows.

2068 For each method M in the interface, if another method P in the interface has

- 2069 a. a method name that is M's method name with the characters "Async" appended, and
- 2070 b. the same parameter signature as M, and
- 2071 c. a return type of Response<R> where R is the return type of M

2072 then P is a JAX-WS polling method that isn't part of the SCA interface contract.

2073 For each method M in the interface, if another method C in the interface has

- 2074 a. a method name that is M's method name with the characters "Async" appended, and
- 2075 b. a parameter signature that is M's parameter signature with an additional final parameter of type
2076 AsyncHandler<R> where R is the return type of M, and
- 2077 c. a return type of Future<?>

2078 then C is a JAX-WS callback method that isn't part of the SCA interface contract.

2079 As an example, an interface can be defined in WSDL as follows:

2080

2081

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

```
2082 <part name="ticker" type="xsd:string"/>
2083 </message>
2084
2085 <message name="getPriceResponse">
2086 <part name="price" type="xsd:float"/>
2087 </message>
2088
2089 <portType name="StockQuote">
2090 <operation name="getPrice">
2091 <input message="tns:getPrice"/>
2092 <output message="tns:getPriceResponse"/>
2093 </operation>
2094 </portType>
```

2095

2096 The JAX-WS asynchronous mapping will produce the following Java interface:

```
2097 // asynchronous mapping
2098 @WebService
2099 public interface StockQuote {
2100     float getPrice(String ticker);
2101     Response<Float> getPriceAsync(String ticker);
2102     Future<?> getPriceAsync(String ticker, AsyncHandler<Float>);
2103 }
```

2104

2105 For SCA interface definition purposes, this is treated as equivalent to the following:

```
2106 // synchronous mapping
2107 @WebService
2108 public interface StockQuote {
2109     float getPrice(String ticker);
2110 }
```

2111

2112 SCA runtimes MUST support the use of the JAX-WS client asynchronous model. In the above
2113 example, if the client implementation uses the asynchronous form of the interface, the two
2114 additional getPriceAsync() methods can be used for polling and callbacks as defined by the JAX-
2115 WS specification.

2116

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

```
2117 <?xml version="1.0" encoding="UTF-8"?>
2118 <!-- (c) Copyright SCA Collaboration 2006 -->
2119 <schema xmlns="http://www.w3.org/2001/XMLSchema"
2120       targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2121       xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200712"
2122       elementFormDefault="qualified">
2123
2124   <include schemaLocation="sca-core.xsd"/>
2125
2126   <element name="interface.java" type="sca:JavaInterface"
2127           substitutionGroup="sca:interface"/>
2128   <complexType name="JavaInterface">
2129     <complexContent>
2130       <extension base="sca:Interface">
2131         <sequence>
2132           <any namespace="##other" processContents="lax"
2133               minOccurs="0" maxOccurs="unbounded"/>
2134         </sequence>
2135         <attribute name="interface" type="NCName" use="required"/>
2136         <attribute name="callbackInterface" type="NCName"
2137                   use="optional"/>
2138         <anyAttribute namespace="##any" processContents="lax"/>
2139       </extension>
2140     </complexContent>
2141   </complexType>
2142 </schema>
```

2143

B. Conformance Items

2144 This section contains a list of conformance items for the SCA Java Common Annotations and APIs
2145 specification.

2146

Conformance ID	Description
[JCA30001]	@interface MUST be the fully qualified name of the Java interface class
[JCA30002]	@callbackInterface MUST be the fully qualified name of a Java interface used for callbacks
[JCA30003]	However, if the Java interface class identified by the @interface attribute does contain a Java @Callback annotation, then the Java interface class identified by the @callbackInterface attribute MUST be the same interface class.

2147

2148

C. Acknowledgements

2149

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

2150

2151

Participants:

2152

[Participant Name, Affiliation | Individual Member]

2153

[Participant Name, Affiliation | Individual Member]

2154

D. Non-Normative Text

2156

E. Revision History

2157 [optional; should not be included in OASIS Standards]

2158

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

			All comments removed.
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.

2159