

SCA Service Component Architecture

Client and Implementation Model Specification

for Java

SCA Version 0.9, November 2005

Technical Contacts:	Michael Beisiegel	IBM Corporation
	Henning Blohm	SAP AG
	Dave Booz	IBM Corporation
	Jean-Jacques Dubray	SAP AG
	Mike Edwards	IBM Corporation
	Bruce Ge	Siebel Systems, Inc.
	Mike Greenberg	IONA Technologies plc.
	Dan Kearns	Siebel Systems, Inc.
	Mike Lehmann	Oracle Corporation
	Jim Marino	BEA Systems, Inc.
	Martin Nally	IBM Corporation
	Greg Pavlik	Oracle Corporation
	Michael Rowley	BEA Systems, Inc.
	Adi Sakala	IONA Technologies plc.
	Ken Tam	BEA Systems, Inc.
	Lance Waterman	Sybase, Inc.

Copyright Notice

© Copyright BEA Systems, Inc., International Business Machines Corp, IONA Technologies, Oracle USA Inc, SAP AG., Siebel Systems, Inc., Sybase, Inc. 2005. All rights reserved.

License

The Service Component Architecture Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy and display the Service Component Architecture Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Component Architecture Specification, or portions thereof, that you make:

1. A link or URL to the Service Component Architecture Specification at these locations:

- <http://dev2dev.bea.com/technologies/commonj/index.jsp>
- <http://www.ibm.com/developerworks/library/specification/ws-sca/>
- <http://www.iona.com/devcenter/sca/>
- <http://oracle.com/technology/webservices/sca>
- <https://www.sdn.sap.com/>
- <http://www.sybase.com/sca>

2. The full text of this copyright notice as shown in the Service Component Architecture Specification.

BEA, IBM, IONA, Oracle, SAP, Siebel, Sybase (collectively, the "Authors") agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Component Architecture Specification.

THE Service Component Architecture SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE SERVICE DATA OBJECTS SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Component Architecture Specification or its contents without specific, written prior permission. Title to copyright in the Service Component Architecture Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

BEA is a registered trademark of BEA Systems, Inc.

IONA and IONA Technologies are registered trademarks of IONA Technologies plc.

Oracle is a registered trademark of Oracle Corporation.

SAP is a registered trademark of SAP AG.

Siebel is a registered trademark of Siebel Systems, Inc.

Sybase is a registered trademark of Sybase, Inc.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Table of Contents

Copyright Notice	ii
License	ii
Status of this Document	iii
1. Client and Implementation Model.....	1
1.1. Introduction	1
1.2. Basic Component Implementation Model	1
1.2.1. Implementing a Service	1
1.2.2. Implementing a Configuration Property	9
1.2.3. Component Type and Component	11
1.3. Basic Client Model.....	12
1.3.1. Accessing Services from Component Implementations	12
1.3.2. Accessing Services from non-SCA component implementations	16
1.3.3. Calling Service Methods	17
1.4. Error Handling.....	17
1.5. Asynchronous Programming	18
1.5.1. Nonblocking Calls	18
1.5.2. Conversational Services	18
1.5.3. Callbacks	24
1.5.4. Bindings for Conversations and Callbacks.....	29
1.6. Java API	29
1.6.1. Current Module Context	29
1.6.2. Module Context.....	30
1.6.3. Request Context	31
1.6.4. ServiceReference	31
1.6.5. No Registered Callback Exception	32
1.6.6. Service Runtime Exception	32
1.6.7. Service Unavailable Exception	32
1.6.8. Session Ended Exception	32
1.7. Java Annotations	33
1.7.1. @AllowsPassByReference	33
1.7.2. @Callback	34
1.7.3. @ComponentName.....	34
1.7.4. @ComponentMetadata	35
1.7.5. @Context.....	35
1.7.6. @Destroy.....	36
1.7.7. @Init.....	36
1.7.8. @Property.....	37

1.7.9.	@Reference.....	38
1.7.10.	@Remotable	39
1.7.11.	@Scope	39
1.7.12.	@Service.....	40
1.7.13.	@Session	41
1.7.14.	@SessionID	42
1.8.	WSDL 2 Java and Java 2 WSDL.....	42
1.9.	Proposed Future Additions to the Specification	43
2.	Appendix.....	44
2.1.	More on the Stateful Resource Pattern	44
2.2.	References.....	45

1. Client and Implementation Model

1.1. Introduction

This document describes the SCA Client and Implementation Model for the Java™ programming language.

The SCA Java implementation model describes how to implement SCA components in Java. A component implementation itself can also be a client to other services provided by other components or external services. The document describes how a Java implemented component gets access to services and calls their methods.

The document also explains how non-SCA Java components can be clients to services provided by other components or external services. The document shows how those non-SCA Java component implementations get access to services and call their methods.

This document defines implementation meta-data 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 meta-data that is represented by annotations can also be defined using a component type side file, as defined in the SCA Assembly Specification.

Note: There may be a few places where the current definition of the component type side file cannot represent all of the meta-data defined in the annotations listed in this document. These will be fixed before the final specification is complete.

1.2. Basic Component Implementation Model

This section describes how SCA components are implemented using the Java programming language. The sections shows how a Java implementation based component can implement a local or remotable service, and how the implementation can be made configurable through properties.

The SCA component implementation model provides a series of annotations which can be placed in the code to mark significant elements of the implementation which are used by the SCA runtime.

Every component implementation is itself also a client of services, this aspect of a component implementation is described in the basic client model section.

1.2.1. Implementing a Service

A component implementation based on a Java class (a **Java implementation**) provides one or more services.

The services provided by the Java implementation have an interface which is defined in one of the following ways:

- A Java interface (which is SCA's preferred style)
- A Java class (used for easy integration of existing Java classes which can only be used for the implementation of local services. See section on [Implementing a Local Service](#))

- A Java interface generated from a [Web Services Description Language \[5\]](#) (WSDL) portType (Java interfaces generated from a WSDL portType are always remotable. See section on [Implementing a Remotable Service](#))

The requirement on a Java implementation is that it implements all the operations defined by the service interface. If the service interface is defined by a Java interface the Java class based component implementation can either implement that Java interface, or just implement all the operations of the interface.

All Java class based component implementations must implement or have a default **zero-argument constructor**.

The **@Service annotation** type is used on a Java class component implementation to specify the interfaces of the services implemented by the implementation. A class that is intended to be used as the implementation of a service is not required to have an @Service annotation. If it has no such annotation, then it is assumed that all implemented interfaces that have been annotated as @Remotable are the service interfaces provided by the component. If none of the implemented interfaces is remotable, then by default the component offers a single service whose type is the implementation class.

The @Service annotation has the following attribute:

- **interfaces** – The value is an array of interface or class objects that should be exposed as services by this component.
- **value** – A shortcut for when only a single service interface is provided. Only one of the attributes should be specified.

An @Service with no attributes is meaningless; it's the same as not having it there at all.

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

If a Java implementation needs to realize two services with the same interface, then this is achieved through subclassing of the interface. The subinterface must not add any methods. Both interfaces are listed in the @Service annotation of the Java implementation class.

The following snippets show the Java service interface and the Java implementation class of a Java implementation.

```
package services.hello;

public interface HelloService {
    String hello(String message);
}

package services.hello;

import org.osoa.sca.annotations.*;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {
```

```

    public String hello(String message) {
        ...
    }
}

```

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

```

<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.oesa.org/xmlns/sca/0.9">
    <service name="HelloService">
        <interface.java interface="services.hello.HelloService"/>
    </service>
</componentType>

```

The following snippet shows a Java implementation class where the service interface is defined by the class itself.

```

package services.hello;

import org.oesa.sca.annotations.*;

@Service(HelloServiceImpl.class)
public class HelloServiceImpl implements AnotherInterface {

    public String hello(String message) {
        ...
    }
}

```

Based on the default rules for the @Service annotation you could also write the following.

```

package services.hello;

public class HelloServiceImpl implements AnotherInterface {

    public String hello(String message) {
        ...
    }
}

```

The following snippet shows the component type for the former two component implementations with the service interface defined by the implementation class itself. There is no need to author this component type in this case since it can all be reflected from the Java class.

```

<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.oesa.org/xmlns/sca/0.9">
    <service name="HelloService">
        <interface.java interface="services.hello.HelloServiceImpl"/>
    </service>
</componentType>

```

```

    </service>
</componentType>

```

The following snippet shows a Java implementation class that implements two services.

```

package services.hello;

import org.osoa.sca.annotations.*;

@Service(interfaces={ HelloService.class, AnotherInterface.class})
public class HelloServiceImpl implements HelloService, AnotherInterface {

    public String hello(String message) {
        ...
    }
    ...
}

```

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

```

<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">

    <service name="HelloService">
        <interface.java interface="services.hello.HelloService"/>
    </service>
    <service name="AnotherService">
        <interface.java interface="services.hello.AnotherService"/>
    </service>

</componentType>

```

The following snippets show a Java implementation that implements two services that are typed by the same interface through subclassing.

```

package services.hello;

public interface HelloService {

    String hello(String message);
}

package services.hello;

public interface HelloService2 extends HelloService {
}

package services.hello;

import org.osoa.sca.annotations.*;

```

```

@Service(interfaces={ HelloService.class,HelloService2.class})
public class HelloServiceImpl implements HelloService {

    public String hello(String message) {
        ...
    }
}

```

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

```

<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">

    <service name="HelloService">
        <interface.java interface="services.hello.HelloService"/>
    </service>
    <service name="HelloService2">
        <interface.java interface="services.hello.HelloService"/>
    </service>

</componentType>

```

1.2.1.1. Implementing a Remotable Service

Remotable services are services that can be published through entry points. Published services can be accessed by clients outside of the module that contains the component that provides the service.

Whether a service is remotable is defined by the interface of the service. In the case of Java this is defined by adding the **@Remotable** annotation to the Java interface. A service whose interface is defined by a Java class is not remotable. Java interfaces generated from WSDL portTypes are remotable, see the [WSDL 2 Java and Java 2 WSDL](#) section for details.

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

```

package services.hello;

import org.osoa.sca.annotations.*;

@Remotable
public interface HelloService {

    String hello(String message);
}

```

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

Complex data types exchanged via remotable service interfaces must be compatible with the marshalling technology that is used by any binding that is used for the service. For example, if the service is going to be exposed using the standard web service binding, then the parameters must be Service Data Objects (SDOs) 2.0 [\[1\]](#) or JAXB [\[2\]](#) types.

Independent of whether the remotable service is called from outside a module or from another component in the same module the data exchange semantics are **by-value**.

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

A remotable service can declare on its interface whether it allows pass by reference data exchange semantics on calls to it, meaning that the by-value semantics can be maintained without requiring that the parameters be copied. The implementation of remotable services that allow pass by reference must not alter its input data during or after the invocation, and must not modify return data after invocation. The **@AllowsPassByReference** annotation on the implementation of a remotable service is used to either declare that calls to the whole interface or individual methods allow pass by reference.

The following snippets show a remotable Java service interface and Java component implementation that implements the service interface and allows pass by references.

```
package services.hello;

import org.osoa.sca.annotations.*;

@Remotable
public interface HelloService {

    String hello(String message);
}

package services.hello;

import org.osoa.sca.annotations.*;

@Service(HelloService.class)
@AllowsPassByReference
public class HelloServiceImpl implements HelloService {

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

The following snippet shows how you can apply the @AllowsPassByReference annotation to an individual method.

```
package services.hello;

import org.osoa.sca.annotations.*;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {

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

```

    ...
  }
}

```

1.2.1.2. Implementing a Local Service

A local service can only be called by clients that are part of the same module as the component that implements the local service. Local services cannot be published through entry points.

Whether a service is local is defined by the interface of the service. In the case of Java this is defined by a Java interface with no `@Remotable` annotation or a Java class.

The following snippet shows the Java interface for a local service.

```

package services.hello;

public interface HelloService {

    String hello(String message);
}

```

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

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

1.2.1.3. Semantics of an unannotated POJO

A class that has no annotations and no corresponding *ComponentType* side file can still be used by an SCA module. The class is considered to implement a single local service whose type is defined by the class (recall that local services may be typed using either Java interfaces or classes).

The properties and references of the class are considered to be represented by the *set* methods present in the class, irrespective of their access mode. For set methods that take a complex type, SCA may not know whether the set method expects a property or a reference, since both can be represented by complex Java types. However, if the parameter of the set method is defined using an `@Remotable` interface, then it must be a reference. When it isn't possible to classify the interface of the parameter in this way, it may be set using either a component property or reference, as long as the type conforms to the parameter type.

Set methods that take simple types are always represented as properties.

1.2.1.4. Implementing the Stateful Resource Pattern

The fact that a service has its state maintained by SCA is represented by an `@Scope` annotation on either the service's interface definition or on the service class itself. SCA refers to such a service as a scoped service.

The `@Scope` annotation has the following attribute:

- *value* – the name of the scope

Clients accessing a scoped service will get a service instance according to scope. The following are the currently supported scope values:

- **stateless** (*default*) – Each request is handled separately. Java instances may be drawn a pool of instances that are use to service requests.
- **request** – Services requests are delegated to the same Java instance for all local service invocations that occur while servicing a remote service request. The lifecycle of a request scope extends from the point a client “request” enters the SCA runtime and a thread services that request until the thread stops synchronously servicing the request. Lookups will return the same service instance for the duration of the request as request-scoped services are bound to the thread of execution on which they were created. It is illegal to declare that a **@remotable** interface is **request** scoped.
- **session** – Service requests are delegated to the same Java instance for all requests in the same “session”. The lifecycle of a session scope extends from the point a session is established with the SCA runtime until the session is ended or expired. Subsequent requests by the same client within the session will participate in the same SCA session scope. Lookups will therefore return the same session-scoped service instance for the duration of the session. SCA provides no further definition of session, but instead requires the hosting application environment to provide the meaning of “session” for each transport binding. The definition of “session” must always maintain the restriction that a single thread operates in the context of at most one session throughout the handling of a request. For example, in the context of a Java™ 2 Enterprise Edition (J2EE) application environment, session scope could be bound to an HTTP Session.
- **module** – Service requests are delegated to the same Java instance for all requests within the same “module”. The lifecycle of a module-scoped instance extends from the point where the module component is loaded in the SCA runtime such that it is ready to service requests until the module is unloaded or stopped. Registry lookups will return the same module-scoped service instance for the duration of the module.

Note: The scope model is extensible and allows for new scopes to be defined. A future version of the specifications will describe how to extend an SCA runtime to handle additional scopes.

For **stateless**, **request** or **session** scoped implementations, the SCA runtime will prevent concurrent execution of methods on an instance of that implementation. However, **module** scoped implementations must be able to handle multiple threads running its methods concurrently.

For bidirectional interfaces, which include both a service interface and a callback interface, the two interfaces must have compatible scopes. The interfaces are compatible if either of the scopes is stateless or if they are the same scope. All other cases are illegal.

The following snippet shows the interface and implementation of a session-scoped service.

```
package services.shoppingcart;

import org.osoa.sca.annotations.*;

@Scope("session")
public interface ShoppingCartService {

    void addToCart(Item item);

    ...
}
```

```

package services.shoppingcart;

import org.osoa.sca.annotations.*;

@Service(ShoppingCartService.class)
public class ShoppingCartServiceImpl implements ShoppingCartService {

    public void addToCart(Item item){
        ...
    }
    ...
}

```

A scoped service can implement **lifecycle methods** on its Java implementation class. These lifecycle methods get called when the scope defined for the service starts or ends. The Java implementation class of the service uses the **@Init** annotation for the method to be called once and only once at the start of its scope and after properties and references have been injected, and it uses the **@Destroy** annotation for the method to be called when its scope ends.

The following snippet shows a Java implementation class of a service with lifecycle methods.

```

package services.shoppingcart;

import org.osoa.sca.annotations.*;

@Service(ShoppingCartService.class)
public class ShoppingCartServiceImpl implements ShoppingCartService {

    public void addToCart(Item item){
        ...
    }

    @Init
    public void start() {
        ...
    }

    @Destroy
    public void stop() {
        ...
    }
}

```

1.2.2. Implementing a Configuration Property

Component implementations can be configured through properties. The **@Property** annotation on a field or setter method in the Java class is used to define a configuration property on a Java component implementation.

Note: At this time SCA does not support constructor injection.

The **@Property** annotation has the following attributes:

- **name** – the name of the property, defaults to the name of the field of the Java class
- **required** – specifies whether injection is required, defaults to false

The type of the Java class field or setter method can be a simple Java type or a complex Java type. Any complex type used for a property must be Java type that has been generated by SDO 2.0 [\[11\]](#) from an XML schema.

Note: In the future, either SCA or SDO [\[11\]](#) may create a more concise XML format for complex property types. There is no current support for `java.util.Map`s and other types.

The `@Property` annotation may be used irrespective of the access modifier of the field (even *private*), since the access modifier defines the contract with the client of the component, whereas the `@Property` annotation specifies a contract with the container of the component.

The following snippet shows the definition of a configuration property using the `@Property` annotation.

```
package services.hello;

import org.osoa.sca.annotations.*;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {

    @Property(name="helloConfigurationProperty", required=true)
    private String helloConfigurationProperty;

    public String hello(String message) {
        System.out.println(helloConfigurationProperty + message);
        ...
    }
}
```

If the property is defined as an array or as a *java.util.Collection*, then the implied component type has a property with a **many** attribute set to true.

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

```
package services.hello;

import java.util.List;
import org.osoa.sca.annotations.*;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {

    @Property(required=true)
    private List<String> helloConfigurationProperty;

    public String hello(String message) {
        System.out.println(helloConfigurationProperty + message);
    }
}
```

```

    ...
  }
}

```

1.2.3. Component Type and Component

For a Java component implementation, a component type can optionally be specified in a side file. The component type side file can be found with the same class loader that the component implementation is on. It is in a directory that corresponds to the namespace of the implementation. It has the same name as the implementation file, but with a `.componentType` extension instead of the `.class` extension.

The rules on how a component type side file adds to the component type information reflected from the component implementation are described as part of the SCA assembly model specification [3]. If the component type information is in conflict with the implementation, then it is an error.

If the component type side file specifies the service interface using a WSDL interface, then the Java class should implement the interface that would be generated by the JAX-WS mapping of WSDL to Java interfaces. See the section [WSDL 2 Java and Java 2 WSDL](#).

This Client and Implementation Model for Java extends the SCA Assembly model [3] providing support for the Java interface type system and support for the Java implementation type.

The following snippet shows the schema for the Java interface element used to type services and references of component types.

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

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

- ***interface*** - fully qualified name of the Java interface or Java class
- ***callbackInterface*** – the optional fully qualified name of the Java callback interface

The following snippet shows the schema for the Java implementation element used to define the implementation of a component.

```
<implementation.java class="NCName" />
```

The `implementation.java` element has the following attributes:

- ***implementation*** - fully qualified name of the Java class implementation of a component

The following snippets show the Java service interface and the Java implementation class of a Java implementation.

```

package services.hello;

public interface HelloService {

    String hello(String message);
}

```

```

}

package services.hello;

import org.osoa.sca.annotations.*;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService {

    public String hello(String message) {
        ...
    }
}

```

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

```

<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">

    <service name="HelloService">
        <interface.java interface="services.hello.HelloService"/>
    </service>

</componentType>

```

The following snippet shows the component using the implementation.

```

<?xml version="1.0" encoding="ASCII"?>
<module xmlns="http://www.osoa.org/xmlns/sca/0.9"

    name="HelloModule" >

    ...

    <component name="HelloService">
        <implementation.java class="services.hello.HelloServiceImpl"/>
    </component>

    ...

</module>

```

1.3. Basic Client Model

This section describes how to get access to SCA services from both SCA components and from non-SCA components. It also describes how to call methods of these services.

1.3.1. Accessing Services from Component Implementations

The following sections describe the ways for getting access to a service. The ways are:

- using reference injection
- using the module context.

Using reference injection is the preferred way to access a service, since it results in code with minimal use of middleware APIs. The other methods should be used in cases where reference injection is not possible.

1.3.1.1. Using Reference Injection

Getting access to a service using reference injection is realized by defining a Java class field typed by the interface of the service and annotated by an **@Reference** annotation.

The @Reference annotation has the following attributes:

- **name** – the name of the reference, defaults to the name of the field of the Java class
- **required** – whether injection of service or services is required

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

```
package services.client;

import services.hello.HelloService;

import org.osoa.sca.annotations.*;

@Service(ClientService.class)
public class ClientServiceImpl implements ClientService {

    @Reference(name="helloService", required=true)
    private HelloService helloService;

    public void clientMethod() {
        String result = helloService.hello("Hello World!");
        ...
    }
}
```

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

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">

    <service name="ClientService">
        <interface.java interface="services.client.ClientService"/>
    </service>
    <reference name="helloService">
        <interface.java interface="services.hello.HelloService"/>
    </reference>

</componentType>
```

If the reference is defined as an array or as a **java.util.Collection**, then the implied component type has a reference with a **multiplicity** of either **1..n** or **0..n**, depending on whether the **required** attribute of the **@Reference** annotation is set to true or false.

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

```
package services.client;

import java.util.List;

import org.osoa.sca.annotations.*;

import services.hello>HelloService;

@Service(ClientService.class)
public class ClientServiceImpl implements ClientService {

    @Reference(name="helloService", required=true)
    private List<HelloService> helloServices;

    public void clientMethod() {
        ...
        HelloService helloService = (HelloService)helloServices.get(index);
        String result = helloService.hello("Hello World!");
        ...
    }
}
```

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

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9">

    <service name="ClientService">
        <interface.java interface="services.client.ClientService"/>
    </service>
    <reference name="helloServices" multiplicity="1..n">
        <interface.java interface="services.hello>HelloService"/>
    </reference>

</componentType>
```

1.3.1.2. Using Module Context

When a component needs access to a service whose name won't be known until runtime, the service can be accessed from the `ModuleContext`.

In order to access a service using the module context two things have to be done:

1. a field has to be defined to accept an injected module context
2. a method has to be called on the injected module context.

The module context gets injected by defining a field with a type of **ModuleContext** and with an **@Context** annotation. The type of context to be injected is determined from the type of the field; in this case, ModuleContext.

The following snippet shows the ModuleContext Java interface with its **locateService()** method.

```
package org.osoa.sca;

public interface ModuleContext {
    ...
    Object locateService(String serviceName);
}
```

The locateService() method takes as its input argument the name of the service and returns an object providing access to the service. The returned object implements the Java interface the service is typed with.

The named service has to be part of the same SCA module. The service name can be one of the following:

- `<component-name>/<service-name>`
 - the service-name is optional if the component has only one service defined
- `<externalService-name>`

The following shows a sample of a module context definition in a Java class using the @Context annotation. In the clientMethod() of the Java class the module context is used and the locateService() method is called on it passing the service name as input. The return of the locateService() method is cast to the interface defined for the service.

```
package services.client;

import org.osoa.sca.ModuleContext;
import org.osoa.sca.annotations.*;

import services.hello.HelloService;

@Service(ClientService.class)
public class ClientServiceImpl implements ClientService {

    @Context
    ModuleContext moduleContext;

    public void clientMethod() {

        HelloService helloService = (HelloService)moduleContext.locateService("HelloService");
        String result = helloService.hello("Hello World!");
        ...
    }
}
```

1.3.2. Accessing Services from non-SCA component implementations

The following sections describe how non-SCA components that are part of an SCA module get access to services.

1.3.2.1. Using Module Context

Non-SCA code that is part of an SCA module can use the ModuleContext in their implementations in order to find services. Non-SCA code is considered to be part of an SCA module if it is in the same class loader as the SCA module, or in any descendent class loader.

The following snippet shows the ModuleContext Java interface.

```
package org.osoa.sca;

public interface ModuleContext {

    ...

    Object locateService(String serviceName);
}
```

A non-SCA component can access the current ModuleContext through the CurrentModuleContext class, which is defined as follows:

```
package org.osoa.sca;

public final class CurrentModuleContext {
    public static ModuleContext getContext() { ... }
}
```

The non-SCA component would include a line like the following in its implementation to get access to the module context:

```
ModuleContext moduleContext = CurrentModuleContext.getContext();
```

Once the module context is available, the required service can be found using the locateService() method of the module context. The following sample shows the usage of the module context from a JSP to access a service and call a method against it.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"
    import="org.osoa.sca.*, commonj.sdo.*, services.hello.*"%>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<META name="GENERATOR" content="IBM Software Development Platform">
<META http-equiv="Content-Style-Type" content="text/css">
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<TITLE>index.jsp</TITLE>
</HEAD>
<BODY>
```

```

<%
...

ModuleContext moduleContext = CurrentModuleContext.getContext();
HelloService helloService = (HelloService)moduleContext.locateService("HelloService");
String result = helloService.hello("Hello World!");
...

%>
</BODY>
</HTML>

```

1.3.3. Calling Service Methods

The previous sections show the various options for getting access to a service. Once you have access to the service, calling a method of the service is like calling a method of a Java class.

The following snippet shows the invocation of the `getQuote` method of a service implementing the `StockQuote` interface.

```

package services.client;

import services.hello.HelloService;

import org.osoa.sca.annotations.*;

@Service(ClientService.class)
public class ClientServiceImpl implements ClientService {

    @Reference(name="helloService",required=true)
    private HelloService helloService;

    public void clientMethod() {
        String result = helloService.hello("Hello World!");
        ...
    }
}

```

If you have access to a service whose interface is marked as `remotable`, then on calls to methods of that service you will experience remote semantics. Arguments and return are passed **by-value** and you may get a ***ServiceUnavailableException***, which is a `RuntimeException`.

1.4. Error Handling

Clients calling service methods will experience business exceptions, and SCA runtime exceptions.

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

SCA runtime exceptions are raised by the SCA runtime and signal problems in the management of the execution of components, and in the interaction with remote services. Currently the following SCA runtime exceptions are defined:

- **ServiceRuntimeException** - signals problems in the management of the execution of SCA components.
- **ServiceUnavailableException** – signals problems in the interaction with remote services. This extends ServiceRuntimeException. These are exceptions that may be transient, so retrying is appropriate. Any exception that is a ServiceRuntimeException that is *not* a ServiceUnavailableException is unlikely to be resolved by retrying the operation, since it most likely requires human intervention.

1.5. Asynchronous Programming

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

1.5.1. Nonblocking Calls

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

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

SCA does not currently define a mechanism for making non-blocking calls to methods that return values or are declared to throw exceptions. It is recommended that service designers define one-way methods as often as possible, in order to give the greatest degree of binding flexibility to deployers.

1.5.2. Conversational Services

A frequent pattern that occurs during the execution of remotable services is that a conversation is started between the client of the service and the provider of the service. The conversation is a series of method invocations that all pertain to a single common topic. For example, a conversation may be the series of service calls that are necessary in order to apply for a bank loan.

Traditionally, it has been necessary for application programmers to write a significant amount of code in order to support this pattern. They must choose a unique key that will be used to identify the conversation, such as a loan ID, which they would use as a session ID. They must pass this session ID as part of any message that is part of the conversation. The service implementation has to make sure to store the state of the conversation at the end of each operation, using the session ID as the key, and look up the state of the conversation at the beginning of the next operation. If the application developer is developing for a clustered environment, then if they want the application to perform well, they must figure out how to route conversational requests to the node where the state is already in memory.

SCA makes it possible to design conversational services while leaving the details of ID generation, state management and routing to the SCA container. In SCA, a *session* is used to maintain information about a single conversation between a client and a remotable service. If the interface of the service has a scope

whose value is *session*, then the identity of the conversation (the session ID) does not need to be explicitly generated or passed as part of the body of messages, and the component that implements the service may store conversational state in its fields.

Note that the conversation identified by a session is always for a remotable service. When a remotable service has a session scope, then the session represents a conversation with one client of that remotable service. However, if a *local* service has a session scope, it means the local service is associated with the client of the last remotable service in the call stack, *not* with the client of the local service.

The following diagram illustrates the relationship between sessions and local and remotable services.

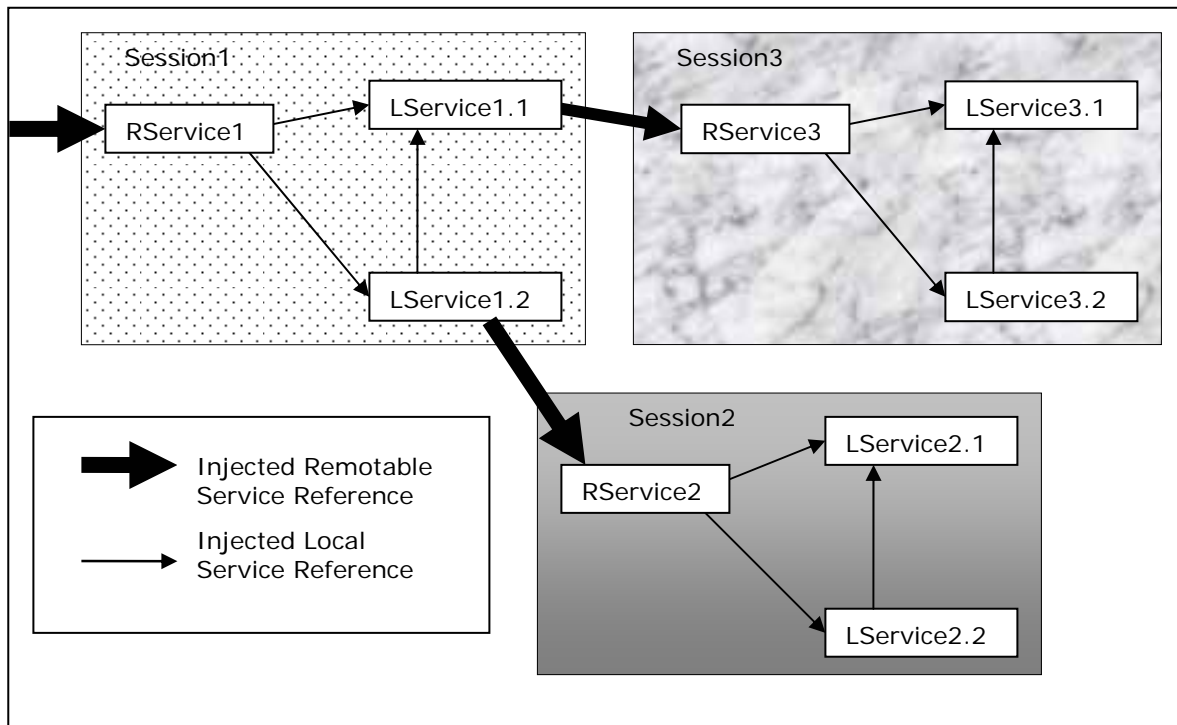


Figure 1: Session Scoped Remotable and Local Services

This diagram shows a variety of services, all of which are session scoped. The services named RService are remotable and the services named LService are local. Note that modules are not represented in this diagram. Each of the shaded areas could be in different modules or they could all be in the same module.

RService1 has injected references to LService1.1 and LService1.2. The reference that LService1.2 has to LService1.1 will refer to the same instance of LService1.1 that is referenced by RService1. However, the reference that LService1.2 has to RService2 starts a new conversation. Similarly for the reference that LService1.1 has to RService3.

There are times when a local service is called without an SCA remotable service earlier in the call stack, such as when a local service is called from a JSP. In these cases, a session is always considered to be present, but the semantics of the session are determined by the runtime environment. How the runtime environment starts and ends sessions will be standardized in a future specification. In the case of a JSP, the session would be tied to the HTTP session.

In the typical case, the lifetime of a session lasts until the client calls an operation that has been marked with an `@EndSession` annotation (or an `@EndSession` annotated callback method). The client is allowed to store the service reference object in a database and later load the object and continue using it. Typically, the service reference would be persisted by keeping it in the field of an object in the persistence tier of the application architecture (e.g. EJB3, JDO or Hibernate). However, a session may have a "lifetime" timeout, after which attempting to use the session will cause a `SessionEndedException` to be generated. A session may also be ended by calling the `endSession()` method on a service reference.

The following is an example interface that has been annotated as being conversational:

```
package com.bigbank;
import org.osoa.sca.annotations.Scope;
import org.osoa.sca.annotations.Remotable;

@Scope("session")
@Remotable
public interface LoanService {
    public void apply(LoanApplication application);
    public void lockCurrentRate(int termInYears);
    public void cancelApplication();
    public String getLoanStatus();
}
```

1.5.2.1. Conversational Client

There is no special coding required by the client of a conversational service. The developer of the client knows that the service is conversational by the *session* scope of the interface. The following shows an example client of the conversational service described above:

```
package com.independent;

import com.bigbank.LoanService;
import org.osoa.sca.annotations.Remotable;

@Remotable
public class BrokerImpl implements MortgageBroker {
    @Reference
    LoanService loanService;
    // Known to be conversational because the interface is conversational.

    public void applyForMortgage(Customer customer, HouseInfo houseInfo, int term)
    {
        LoanApplication loanApp;
        loanApp = createApplication(customer, houseInfo);
        loanService.apply(loanApp);
        loanService.lockCurrentRate(term);
    }

    public boolean isApproved() {
        return loanService.getLoanStatus().equals("approved");
    }

    public LoanApplication createApplication(Customer customer, HouseInfo
        houseInfo) {
        return ...;
    }
}
```

```
}

```

1.5.2.2. Conversational Service Provider

The provider of the conversational service also is not required to write any special code in order to be conversational. The fact that the service is conversational is determined by the fact that service implements a session-scoped remotable interface. However, the developer of the service knows that they can store data associated with the conversation in fields, and that data will be available throughout the conversation.

There are a few capabilities that are available to the implementation of the service, but are not required. If a field is annotated with `@SessionID`, then the session ID is injected onto the field.

```
@SessionID
private String mySessionID;
```

The type of the field is not necessarily `String`. System generated session IDs are always strings, but application generated session IDs may be other complex types, as described in the section below titled: "Application Specified Session IDs".

The service implementation class may also have an optional `@Session` annotation that has the following form:

```
@Session(maxIdleTime="10 minutes",
         maxAge="1 day",
         singlePrincipal=false)
```

[**Note:** Alternatively, each scope could be given its own annotation and do away with the general `@Scope` annotation, such as leaving out `@Scope("session")` and having `@SessionScope(maxAge="1 day")`. The attribute values should also be overridable using the implementation policy mechanism.]

The attributes of the annotation have the following meaning:

- ***maxIdleTime*** - The maximum time that can pass between operations within a single conversation. If more time than this passes, then the container may end the conversation.
- ***maxAge*** - The maximum time that the entire conversation can remain active. If more time than this passes, then the container may end the conversation.
- ***singlePrincipal*** – If true, only the principal (the user) that started the conversation has authority to continue the operation.

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

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

Here is an example implementation of a conversational service, with the business logic elided out:

```

package com.bigbank;
import org.osoa.sca.annotations.Session;
import org.osoa.sca.annotations.SessionID;

@Session(maxAge="30 days");
public class LoanServiceImpl implements LoanService {
    @SessionID private String loanID;

    public void apply(LoanApplication application) { ... }
    public void lockCurrentRate(int termInYears) { ... }
    public void cancelApplication() { ... }
    public String getLoanStatus() { ... }
}

```

1.5.2.3. *Methods that End the Conversation*

A method of a session scoped remotable service may be marked with an `@EndSession` annotation. This means that once this message has been called, no further methods may be called on the session so both the client and the service may free up resources that were associated with the session. It is also possible to mark a method on a callback interface (described later) as `@EndSession`, in order for the service provider to be the party that chooses to end the conversation. If a method is called after the conversation completes, the `SessionEndedException` (which extends `ServiceRuntimeException`) is thrown. This may also occur if there is a race condition between the client and the service provider calling their respective `@EndSession` methods.

1.5.2.4. *Passing Conversational Services as Parameters*

The service reference which represents a single conversation can be passed as a parameter to another service, even if that other service is remote. This may be used in order to allow one component to continue a conversation that had been started by another. At this point, service references may only be passed to other services within the same module.

A service provider may also create a service reference for itself that it can pass to other services. A service implementation does this with a call to

```
MethodContext.createServiceReferenceForSession(this).
```

```

interface ModuleContext {
    ...
    ServiceReference createServiceReferenceForSession(Object self);
    ServiceReference createServiceReferenceForSession(Object self, String
                                                    ServiceName);
}

```

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

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

1.5.2.5. *Conversation Lifetime Summary*

Starting conversations

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

A @reference to a conversational service is injected.

A call is made to `ModuleContext.newSession()`.

Continuing conversations

The client can continue an existing conversation, by:

Holding the service reference that was created when the conversation started.

Getting the service reference object passed as a parameter from another service, even remotely.

Loading a service reference that had been written to some form of persistent storage.

Getting the session ID from any source and calling:

```
ModuleContext.lookupServiceReference(serviceName, id)
```

Ending conversations

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

A server operation that has been annotated @EndConveration has been called.

The server calls an @EndSession method on the @Callback reference.

The server's conversation lifetime timeout occurs.

The client calls `ServiceReference.endSession()`.

If a method is invoked on a service reference after an @EndSession method has been called then a new session will automatically be started. If `ServiceReference.getSessionID()` is called after the @EndSession method is called, but before the next session has been started, it will return null.

If a service reference is used after the service provider's session timeout has caused the session to be ended, then `SessionEndedException` will be thrown. In order to use that service reference for a new session, its `endSession()` method must be called.

1.5.2.6. Application Specified Session IDs

It is also possible to take advantage of the state management aspects of conversational services while using a client-provided session ID. To do this, the client would not use injection, but would use the variant of the `newSession()` API that takes a session ID as a parameter. The `newSession()` method returns a conversational service reference, and has the following forms:

```
public interface ModuleContext {
    ...
    ServiceReference newSession(String serviceName);
    ServiceReference newSession(String serviceName, Object sessionId);
}
```

The service reference object returned from this method can be cast to the business interface of the service named by `serviceName`.

The session ID that is passed into this method should be an instance of either a `String` or an object that is serializable into XML, such as an SDO 2.0 [\[1\]](#) `DataObject`. The ID must be unique to the client component over all time. If the client is not an SCA component, then the ID must be globally unique.

Not all conversational service bindings support application-specified session IDs or may only support application-specified session IDs that are Strings. The WS-Addressing binding for conversational services serializes the session ID into XML for use as a reference parameter. See the section titled: "Bindings for Conversations and Callbacks".

1.5.2.7. Accessing Session IDs from Clients

Whether the session ID is chosen by the client or is generated by the system, the client may access the session ID of a conversation by calling the `ServiceReference.getSessionID()` method on the service reference for the conversation. Normally, the service references are kept as fields whose type is the type of the business interface of the service. In this case, the field would need to be cast to `ServiceReference` in order to get access to the `getSessionID` method.

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

1.5.3. Callbacks

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

- an interface for the provided service
- an interface that must be provided by the client

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

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

An example use of the `@Callback` annotation to declare a callback interface would be the following:

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Remotable;
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    public void someMethod(String arg);
}

@Remotable
public interface MyServiceCallback {

    public void receiveResult(String result);
}
```

In this particular example, the implied component type would be the following:

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/0.9" >
    <service name="MyService">
        <interface.java interface="somepackage.MyService"
            callbackInterface="somepackage.MyServiceCallback"/>
    </service>
</componentType>
```

1.5.3.1. Stateful Callbacks

A stateful callback represents a specific implementation instance of the component that is client of the service. The interface of a stateful callback has a scope that is either *session* or *module*. If the scope of the callback interface is stateful, then service that it is a callback for must either be stateless or must have the same scope as the callback. This is to prevent confusing situations that arise when mixing session and module scoped services in a conversation.

The following is an example service implementation (for the service and callback declared above) that uses the `@Callback` annotation to request that a stateful callback be injected. In this case it just passes the request on to some other component, essentially acting as an intermediary. Because the service is session scoped, the callback will still be available when the backend service sends back its asynchronous response.

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Reference;
public class MyServiceImpl implements MyService, MyServiceCallback {

    @Callback
    private MyServiceCallback callback;

    @Reference
    private MyService backendService;

    public void someMethod(String arg) {
        backendService.someMethod(arg);
    }
    public void receiveResult(String result) {
        callback.receiveResult(result);
    }
}
```

This component implements two services, one that it offers to its clients (`MyService`) and one that is used for receiving callbacks from the back end (`MyServiceCallback`). The client of this service would also implement `MyServiceCallback`.

```
package somepackage;
import org.osoa.sca.annotations.Reference;
public class SomeClientImpl implements AnotherService, MyServiceCallback {

    @Reference
```

```

private MyService myService;

public void aClientMethod() {
    ...
    myService.someMethod(arg);
}
public void receiveResult(String result) {
    // code to process the result
}
}

```

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

1.5.3.2. Stateless Callbacks

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

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

```

package somepackage;
import org.osoa.sca.annotations.Reference;

public class SomeClientImpl implements AnotherService, MyServiceCallback {

    @Reference
    private MyService myService;

    public void aClientMethod() {
        String someKey = "1234";
        ...
        ((ServiceReference)myService).setCallbackID(someKey);
        myService.someMethod(arg);
    }
    public void receiveResult(String result) {
        Object key = ((ServiceReference)myService).getCallbackID();
        // Lookup any relevant state based on "key"
        // code to process the result
    }
}

```

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

```

package somepackage;
import org.osoa.sca.annotations.Callback;
public class MyServiceImpl implements MyService {

    @Callback
    private MyServiceCallback callback;

    ...
}

```

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

1.5.3.3. *Implementing Multiple Bidirectional Interfaces*

Since it is possible for a single class to implement multiple services, it is also possible for callbacks to be defined for each of the services that it implements. The service implementation can include an injected field for each of its callbacks. The runtime injects the callback onto the appropriate field based on the type of the callback. The following is an example of a component that provides two services, each of which has a callback.

```

package somepackage;
import org.osoa.sca.annotations.Callback;
public class MyServiceImpl implements MyService1, MyService2 {

    @Callback
    private MyService1Callback callback1;

    @Callback
    private MyService2Callback callback2;
}

```

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

1.5.3.4. *Accessing Callbacks on the RequestContext*

Instead of using an annotated field to get access to a callback object, it is possible for a stateless service to access a specific callback from the RequestContext object in order to access a callback that is associated with the last request.

The RequestContext does not provide direct access to the callback object. Instead, RequestContext has a getServiceReference method. The service reference returned represents the service reference that the request was invoked on. So, a service implementation can get at the callback object associated with a request by calling RequestContext.getServiceReference().getCallback(). It is illegal for the service implementation to try to call the setCallback() on this service reference.

The callback that is accessed this way can be cast to the callback interface of the bidirectional service. On the service provider side, the callback object holds any reference parameters or other state that must be passed back to the client in order to provide the client with sufficient contextual information. The definition of `getService` looks like this:

```
package org.osoa.sca;

public interface RequestContext {
    ...
    ServiceReference getServiceReference();
}

```

A service implementation that uses the request context to access the callback would look like this:

```
package com.bigbank;
import org.osoa.sca.annotations.Context;

public class MyServiceImpl implements MyService {
    @Context;
    private ModuleContext moduleContext;

    public void someMethod() {
        RequestContext rc = moduleContext.getRequestContext();
        MyCallback callback = (MyCallback)rc.getServiceReference().getCallback();

        ...
        callback.receiveResult(theResult);
    }
}

```

On the client side, the service that implements the callback can access the callback ID (i.e. reference parameters) that was returned with the callback operation also by accessing the request context, as follows:

```
public class CallbackImpl implements MyCallback {
    @Context;
    private ModuleContext moduleContext;

    void receiveResult(Object theResult) {
        RequestContext rc = moduleContext.getRequestContext();
        Object refParams = rc.getServiceReference().getCallbackID();

        ...
    }
}

```

On the client side, the object returned by the `getServiceReference()` call represents the service reference that was used to send the original request. The object returned by `getCallbackID()` represents the identity associated with the callback, which may be a single String or may be an object that represents reference parameters.

1.5.3.5. Customizing the Callback

By default, the client component of a service is assumed to be the callback service for the bidirectional service. However, it is possible to change the callback by using the `ServiceReference.setCallback()` method. The object passed as the callback should implement the interface defined for the callback,

including any additional SCA semantics on that interface such as its scope and whether or not it is remotable.

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

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

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

1.5.3.6. Customizing the Callback Identity

The identity that is used to identify a callback request is, by default, generated by the system. However, it is possible to provide an application specified identity that should be used to identify the callback by calling the `ServiceReference.setCallbackID()` method. This can be used even either stateful or stateless callbacks. The identity will be sent to the service provider, and the binding must guarantee that the service provider will send the ID back when any callback method is invoked.

The callback identity has the same restrictions as the session ID. It should either be a string or an object that can be serialized into XML, such as an SDO 2.0 [\[1\]](#) `DataObject`. When the WS-Addressing conversation binding is used, the callback ID will be represented as reference parameters in the endpoint address that is sent as a reply-to address. However, other bindings may use other mechanisms.

1.5.4. Bindings for Conversations and Callbacks

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

WS-Addressing can be used to provide a session ID, but only for callbacks. WS-Addressing does not define a way for an identity to be provided by a client that identifies an entire conversation.

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

1.6. Java API

1.6.1. Current Module Context

The following snippet shows the `CurrentModuleContext` Java class.

```

package org.osoa.sca;

public final class CurrentModuleContext {

    public static ModuleContext getContext() {...}
}

```

The CurrentModuleContext Java class has the following methods:

- **getContext()** – returns the current module context

Note: The service provider aspect of the CurrentModuleContext implementation will be defined in a future specification.

1.6.2. Module Context

The following snippet shows the ModuleContext Java interface.

```

package org.osoa.sca;

import org.osoa.sca.model.Module;

public interface ModuleContext {

    String getName();
    String getURI();
    Module getMetaData();

    RequestContext getRequestContext();

    Object locateService(String serviceName);

    ServiceReference createServiceReferenceForSession(Object self);
    ServiceReference createServiceReferenceForSession(Object self, String
                                                         serviceName);

    ServiceReference newSession(String serviceName);
    ServiceReference newSession(String serviceName, Object sessionID);
}

```

The ModuleContext Java interface has the following methods:

- **getName()** – returns the name of the module
- **getURI()** – returns the absolute URI of the module component
- **getMetaData()** – returns the Module meta data object. The fully qualified name of the return type is **org.osoa.sca.model.Module**. The Module interface and all the other model interfaces are created from the SCA XML schema using the SDO 2.0 [\[1\]](#) XML schema to Java mapping rules. The returned object can also be cast to `commonj.sdo.DataObject`.
- **getRequestContext()** - returns the RequestContext object that corresponds to the last remotable service invocation in the call stack. Local service invocations do not affect the request context that is active. If this is called from code that is not part of an SCA component **null** is returned.
- **locateService()** – returns an object implementing the interface defined for the named service. Input to the method is the name of a service that is part of the same SCA module. The service name can be one of the following:

- `<component-name>/<service-name>`
 - the `service-name` is optional if the component has only one service defined
- `<externalService-name>`
- ***createServiceReferenceForSession()*** – takes a running instance of a service implementation and generates a service reference that can be used for conversations with that existing instance. The second variant, which takes an additional `serviceName` parameter, must be used if the component implements multiple services.
- ***newSession()*** – allows the creation of a `ServiceReference` for a service, the service name is provided as input. The second variant allows the creation of a `ServiceReference` with an application provided session ID.

1.6.3. Request Context

The following snippet shows the `RequestContext` Java interface.

```
package org.osoa.sca;

import java.util.Map;
import javax.security.auth.Subject;

public interface RequestContext {

    Subject getSecuritySubject();

    String getServiceName();

    ServiceReference getServiceReference();

}
```

The `RequestContext` Java interface has the following methods:

- ***getSecuritySubject()*** – returns the JAAS Subject of the current request
- ***getServiceName()*** – returns the name of the service on the Java implementation the request came in on
- ***getServiceReference()*** – returns the service reference that represents the service reference that the request was invoked on. A service implementation can get at the callback object associated with a request by calling `getServiceReference().getCallback()`. It is illegal for the service implementation to try to call the `setCallback()` on this service reference.

1.6.4. ServiceReference

Any object that implements the `ServiceReference` interface can also be cast to the business interface of the service that is targeted by the reference. All references to remotable services implement both the `ServiceReference` interface and the business interface.

```
package org.osoa.sca;

public interface ServiceReference {

    Object getSessionID();

    void endSession();

}
```

```

Object getCallbackID();
void setCallbackID(Object callbackID);

Object getCallback();
void setCallback(Object callback);
}

```

The ServiceReference Java interface has the following methods:

- **getSessionID()** – returns the session ID
- **endSession()** – ends the clients session with the referenced service
- **getCallbackID()** – returns the callback ID
- **setCallbackID()** – sets the callback ID
- **getCallback()** – returns the callback object
- **setCallback()** – sets the callback object

The detailed description of the usage of these methods is described in the section on Asynchronous Programming in this document.

1.6.5. No Registered Callback Exception

The following snippet shows the NoRegisteredCallbackException.

```

package org.osoa.sca;

public class NoRegisteredCallbackException extends ServiceRuntimeException {
    ...
}

```

1.6.6. Service Runtime Exception

The following snippet shows the ServiceRuntimeException.

```

package org.osoa.sca;

public class ServiceRuntimeException extends RuntimeException {
    ...
}

```

1.6.7. Service Unavailable Exception

The following snippet shows the ServiceRuntimeException.

```

package org.osoa.sca;

public class ServiceUnavailableException extends ServiceRuntimeException {
    ...
}

```

1.6.8. Session Ended Exception

The following snippet shows the SessionEndedException.

```

package org.osoa.sca;

public class SessionEndedException extends ServiceRuntimeException {
    ...
}

```

1.7. Java Annotations

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

1.7.1. @AllowsPassByReference

The following snippet shows the @AllowsPassByReference annotation type definition.

```

package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target({Type, METHOD})
@Retention(RUNTIME)
public @interface AllowsPassByReference {
}

```

The @AllowsPassByReference annotation is used on implementations of remotable interfaces to indicate that interactions with the service within the same module 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 the individual remotable service method implementation can be annotated using the @AllowsPassByReference annotation.

The following snippet shows a sample where @AllowsPassByReference is defined for the complete Java component implementation class.

```

@AllowsPassByReference
public class HelloServiceImpl implements HelloService {
    ...
}

```

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

```

@AllowsPassByReference
public String hello(String message) {
    ...
}

```

```
}

```

1.7.2. @Callback

The following snippet shows the @Callback annotation type definition:

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target(TYPE, METHOD, FIELD)
@Retention(RUNTIME)
public @interface Scope {

    String value() default null;
}

```

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

The @Callback annotation has the following attribute:

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

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

The following snippet shows a callback annotation on an interface:

```
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    public void someAsyncMethod(String arg);
}

```

1.7.3. @ComponentName

The following snippet shows the @ComponentName annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ComponentName {

```

```
}

```

The `@ComponentName` annotation type is used to annotate a Java class field or setter method that is used to inject the component name.

The following snippet shows a component name field definition sample.

```
@ComponentName
private String componentName;
```

1.7.4. @ComponentMetadata

The following snippet shows the `@ComponentMetadata` annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public interface ComponentMetadata {

}
```

The `@ComponentMetadata` annotation type is used to annotate a Java class field or setter method that is used to inject the component meta data object, which is an object that holds the current component's corresponding `<component>` element from the module definition file. The fully qualified name of the field type or setter method is *org.osoa.sca.model.Component*. The Component interface and all the other model interfaces are created from the SCA XML schema using the SDO 2.0 [\[1\]](#) XML schema to Java mapping rules. The returned object can also be cast to `commonj.sdo.DataObject`.

The following snippet shows a component meta data field definition sample.

```
@ComponentMetadata
private Component componentMetadata;
```

1.7.5. @Context

The following snippet shows the `@Context` annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

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

```
@Retention(RUNTIME)
public @interface Context {

}
```

The `@Context` annotation type is used to annotate a Java class field or setter method that is used to inject a module context. The type of context to be injected is defined by the type of the Java class field or type of the setter method input argument, the type is either `ModuleContext`.

The following snippet shows a `ModuleContext` field definition sample.

```
@Context
private ModuleContext moduleContext;
```

1.7.6. @Destroy

The following snippet shows the `@Destroy` annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

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

}
```

The `@Destroy` annotation type is used to annotate a Java class method that will be called when the scope defined for the local service implemented by the class ends. The method has to have no return and zero arguments. The annotated method has to be public.

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

```
@Destroy
void myDestroyMethod() {

    ...

}
```

1.7.7. @Init

The following snippet shows the `@Init` annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
```

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

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

    public boolean eager() default false;
}
```

The @Init annotation type is used to annotate a Java class method that will be called when the scope defined for the local service implemented by the class starts. The method has to have no return and zero arguments. The annotated method has to be public. The annotated method is called after all property and reference injection is complete.

The @Init annotation has the following attributes:

- **eager** - For scoped components, eager = true instructs the container to instantiate a component instance when its scope is initialized. When eager = false, component instances will be created when they are first referenced. The default is false.

The following snippet shows a sample for a init method definition.

```
@Init
void myInitMethod() {

    ...
}
```

1.7.8. @Property

The following snippet shows the @Property annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Property {

    public String name() default "";
    public boolean required() default false;
}
```

The @Property annotation type is used to annotate a Java class field or a setter method that is used to inject a configuration property value. The type of the property injected is defined by the type of the Java class field or the type of the setter method input argument.

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

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

The `@Property` annotation has the following attributes:

- ***name*** – the name of the property, defaults to the name of the field of the Java class
- ***required*** – specifies whether injection is required, defaults to false

The following snippet shows a property field definition sample.

```
@Property(name="currency", required=true)
private String currency;
```

The following snippet shows a property setter sample

```
@Property(name="currency", required=true)
public void setCurrency( String theCurrency );
```

1.7.9. @Reference

The following snippet shows the `@Reference` annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Reference {

    public String name() default "";
    public boolean required() default true;
}
```

The `@Reference` annotation type is used to annotate a Java class field or a setter method that is used to inject a service that resolves the reference. The interface of the service injected is defined by the type of the Java class field or the type of the setter method input argument.

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

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

The `@Reference` annotation has the following attributes:

- ***name*** – the name of the reference, defaults to the name of the field of the Java class
- ***required*** – whether injection of service or services is required

The following snippet shows a reference field definition sample.

```
@Reference(name="stockQuote", required=true)
private StockQuoteService stockQuote;
```

The following snippet shows a reference setter sample

```
@Reference(name="stockQuote", required=true)
public void setStockQuote( StockQuoteService theSQService );
```

1.7.10. @Remotable

The following snippet shows the @Remotable annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Remotable {

}
```

The @Remotable annotation type is used to annotate a Java service interface as remotable. Remotable service can be published using entry points and are translatable into WSDL portTypes.

The following snippet shows a sample for a remotable service interface definition.

```
@Remotable
public interface HelloService {

    ...

}
```

1.7.11. @Scope

The following snippet shows the @Scope annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;
```

```

@Target(TYPE)
@Retention(RUNTIME)
public @interface Scope {

    String value() default "stateless";
}

```

The @Scope annotation type is used on either the service's interface definition or on the service implementation class itself.

The @Scope annotation has the following attribute:

- **value** – the name of the scope, see section [Implementing the Stateful Resource Pattern](#) for the currently supported values

The following snippet shows a sample for a scoped service interface definition.

```

package services.shoppingcart;

@Scope("session")
public interface ShoppingCartService {

    void addToCart(Item item);
}

```

1.7.12. @Service

The following snippet shows the @Service annotation type definition.

```

package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Service {

    Class[] interfaces() default {};
    Class value() default null;
}

```

The @Service annotation type is used on a Java implementation to specify the interfaces of the services implemented by the implementation. A class that is intended to be used as the implementation of a service is not required to have an @Service annotation. If it has no such annotation, then it is assumed that all implemented interfaces that have been annotated as @Remotable are the service interfaces provided by the component. If none of the implemented interfaces is remotable, then by default the component offers a single service whose type is the implementation class.

The @Service annotation has the following attribute:

- **interfaces** – The value is an array of interface or class objects that should be exposed as services by this component.
- **value** – A shortcut for when only a single service interface is provided. Only one of the attributes should be specified.

An @Service with no attributes is meaningless, it's the same as not having it there at all.

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

The following snippet shows a sample that uses the service annotation.

```
package services.hello;

@Service(HelloService.class)
public class HelloServiceImpl implements HelloService, AnotherInterface {

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

1.7.13. @Session

The following snippet shows the @Session annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Session {

    public String maxIdleTime() default "";
    public String maxAge() default "";
    public boolean singlePrincipal() default false;
}
```

The @Session annotation type is used to annotate a Java class. The attributes of the annotation have the following meaning:

- **maxIdleTime** - The maximum time that can pass between operations within a single conversation. If more time than this passes, then the container may end the conversation.
- **maxAge** - The maximum time that the entire conversation can remain active. If more time than this passes, then the container may end the conversation.
- **singlePrincipal** – If true, only the principal (the user) that started the conversation has authority to continue the operation.

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

Not specifying timeouts means that timeouts are defined by the implementation of the SCA run-time, however it choose to do so.

The following snippet shows a component name field definition sample.

```
package service.shoppingcart;

import org.osoa.sca.annotations.*

@Session(maxAge="30 days");
public class ShoppingCartServiceImpl implements ShoppingCartService {
    ...
}
```

1.7.14. @SessionID

The following snippet shows the @SessionID annotation type definition.

```
package org.osoa.sca.annotations;

import java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface SessionID {

}
```

The @SessionID annotation type is used to annotate a Java class field or setter method that is used to inject the session ID. System generated session IDs are always strings, but application generated session IDs may be other complex types, as described in the section titled: "Application Specified Session IDs".

The following snippet shows a component name field definition sample.

```
@SessionID
private String sessionID;
```

1.8. WSDL 2 Java and Java 2 WSDL

The SCA Client and Implementation Model for Java applies the *WSDL to Java* and *Java to WSDL* mapping rules as defined by the [JAX-WS \[4\]](#) specification for generating remotable Java interfaces from WSDL portTypes and vice versa.

For the mapping from Java types to XML schema types SCA supports the SDO 2.0 [\[1\]](#) and JAXB [\[2\]](#) mapping.

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

- No support for holders

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

1.9. Proposed Future Additions to the Specification

The current version of the specification is acknowledged to be incomplete and there are a number of areas where future versions of the specification will be expanded to cover additional capabilities. Some of these capabilities include:

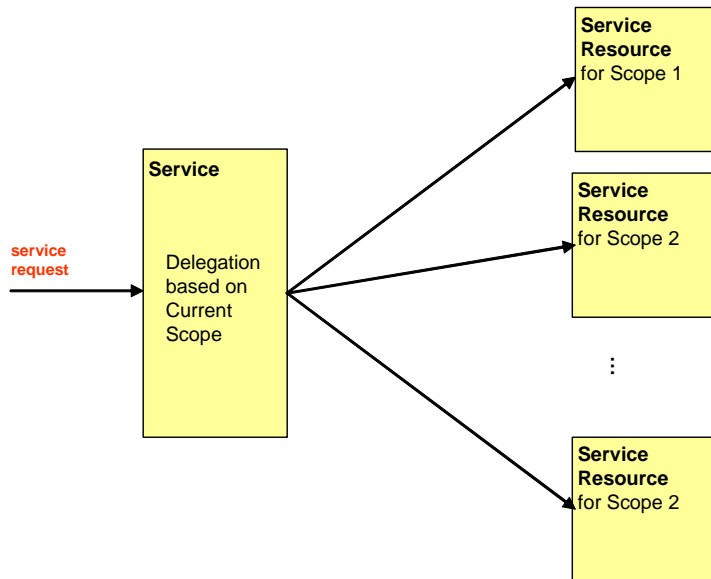
- Policy and QOS annotations including those for Transactions, Security and Reliable Messaging
- A Client asynchronous model for synchronous service operations.
- Provide a Service Provider Interface (SPI) to allow implementers of SCA container capabilities to plug-in implementations to an SCA runtime through a well-defined interface.

2. Appendix

2.1. More on the Stateful Resource Pattern

Services can be implemented using the *stateful resource pattern*. Requests received by the component implementing the service are delegated on a stateful service resource. Both the component implementing the service and the implementation of the stateful resource implement the service interface.

The following shows a service implementation based on the stateful resource pattern.



The realization of this pattern requires the implementation of two artifacts, the service and the stateful service resource. Since the implementation of the delegating service is repetitive, SCA provides a standard implementation for each of several kinds of scope, freeing the programmer to focus only on the implementation of the stateful resource.

2.2. References

[1] SDO 2.0 Specification

Any one of:

- <http://dev2dev.bea.com/technologies/commonj/index.jsp>
- <http://www.ibm.com/developerworks/library/specification/ws-sdo/>
- <http://oracle.com/technology/webservices/sca>
- <https://www.sdn.sap.com/>
- <http://www.xcalia/xdn/specs/sdo>
- <http://www.sybase.com/sca>

[2] JAXB Specification

<http://www.jcp.org/en/jsr/detail?id=31>

[3] SCA Assembly Specification

Any one of:

- <http://dev2dev.bea.com/technologies/commonj/index.jsp>
- <http://www.ibm.com/developerworks/library/specification/ws-sca/>
- <http://www.iona.com/devcenter/sca/>
- <http://oracle.com/technology/webservices/sca>
- <https://www.sdn.sap.com/>
- <http://www.sybase.com/sca>

[4] JAX-WS Specification

<http://jcp.org/en/jsr/detail?id=101>

[5] WSDL Specification

WSDL 1.1: <http://www.w3.org/TR/wsd/>

WSDL 2.0: <http://www.w3.org/TR/wsd20/>