SCA Service Component Architecture

Java Component Implementation Specification

SCA Version 1.00, 15 February 2007

Technical Contacts:

Ron Barack	SAP AG
Michael Beisiegel	IBM Corporation
Henning Blohm	SAP AG
Dave Booz	IBM Corporation
Jeremy Boynes	Independent
Ching-Yun Chao	IBM Corporation
Adrian Colyer	Interface21
Mike Edwards	IBM Corporation
Hal Hildebrand	Oracle
Sabin Ielceanu	TIBCO Software, Inc
Anish Karmarkar	Oracle
Daniel Kulp	IONA Technologies plc.
Ashok Malhotra	Oracle
Jim Marino	BEA Systems, Inc.
Michael Rowley	BEA Systems, Inc.
Ken Tam	BEA Systems, Inc
Scott Vorthmann	TIBCO Software, Inc
Lance Waterman	Sybase, Inc.

Copyright Notice

License

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.

Table of Contents

1. Java Implementation Type

1.1. Introduction

This specification extends the SCA Assembly Model [1] by defining how a Java class provides an implementation of an SCA component and how that class is used in SCA as a component implementation type.

This specification requires all the annotations and APIs as defined by the SCA Java Common Annotations and APIs specification [2]. All annotations and APIs referenced in this document are defined in the former unless otherwise specified. Moreover, the semantics defined in the Common Annotations and APIs specification are normative.

1.2. Java Implementation Type

This section specifies how a Java class provides an implementation of an SCA component, including its various attributes such as services, references, and properties. In addition, it details the use of metadata and the Java API defined in [2] in the context of a Java class used as a component implementation type,

1.2.1. Services

A component implementation based on a Java class may provide one or more services.

The services provided by a Java-based implementation may have an interface defined in one of the following ways:

- A Java interface
- A Java class
- A Java interface generated from a Web Services Description Language [3] (WSDL) portType.

Java implementation classes must implement all the operations defined by the service interface. If the service interface is defined by a Java interface, the Java-based component can either implement that Java interface, or implement all the operations of the interface.

A service whose interface is defined by a Java class (as opposed to a Java interface) is not remotable. Java interfaces generated from WSDL portTypes are remotable, see the WSDL 2 Java and Java 2 WSDL section of the SCA Java Common Annotations and API Specification for details.

A Java implementation type may specify the services it provides explicitly through the use of @Service. In certain cases as defined below, the use of @Service is not required and the services a Java implementation type offers may be inferred from the implementation class itself.

1.2.1.1. Use of @Service

Service interfaces may be specified as a Java interface. A Java class, which is a component implementation, may offer a service by implementing a Java interface specifying the service contract. As a Java class may implement multiple interfaces, some of which may not define SCA services, the @Service annotation can be used to indicate the services provided by the implementation and their corresponding Java interface definitions.

The following is an example of a Java service interface and a Java implementation, which provides a service using that interface:

Interface:

```
public interface HelloService {

        String hello(String message);
}
```

Implementation class:

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

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

The XML representation of the component type for this implementation is shown below for illustrative purposes. There is no need to author the component type as it can 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 Java implementation class itself, as opposed to an interface, may also define a service offered by a component. In this case, @Service may be used to explicitly declare the implementation class defines the service offered by the implementation. In this case, a component will only offer services declared by @Service. The following illustrates this:

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

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

In the above example, HelloWorldServiceImpl offers one service as defined by the public methods on the implementation class. The interface AnotherInterface in this case does not specify a service offered by the component. The following is an XML representation of the introspected component type:

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

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

</componentType>
```

@Service may be used to specify multiple services offered by an implementation as in:

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

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

The following snippet shows the introspected component type for this implementation.

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

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

</componentType>
```

1.2.1.2. Local and Remotable services

A Java service contract defined by an interface or implementation class may use @Remotable to declare that the service follows the semantics of remotable services as defined by the SCA Assembly Specification. The following example demonstrates the use of @Remotable:

```
package services.hello;

@Remotable
public interface HelloService {

      String hello(String message);
}
```

Unless @Remotable is declared, a service defined by a Java interface or implementation class is inferred to be a local service as defined by the SCA Assembly Model Specification.

If an implementation class has implemented interfaces that are not decorated with an @Remotable annotation, the class is considered to implement a single ***local*** service whose type is defined by the class (note that local services may be typed using either Java interfaces or classes).

An implementation class may provide hints to the SCA runtime about whether it can achieve pass-by-value semantics without making a copy by using the @AllowsPassByReference..

1.2.1.3. Introspecting services offered by a Java implementation

In the cases described below, the services offered by a Java implementation class may be determined through introspection, eliding the need to specify them using @Service. The following algorithm is used to determine how services are introspected from an implementation class:

If the interfaces of the SCA services are not specified with the @Service annotation on the implementation class, 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 implementation offers a single service whose type is the implementation class.*

1.2.1.4. Non-Blocking Service Operations

Service operations defined by a Java interface or implementation class may use @OneWay to declare that the SCA runtime must honor non-blocking semantics as defined by the SCA Assembly Specification when a client invokes the service operation.

1.2.1.5. Non-Conversational and Conversational Services

The Java implementation type supports all of the conversational service annotations as defined by the SCA Java Common Annotations and API Specification: @Conversational, @EndsConversation, and @ConversationAttributes.

The following semantics hold for service contracts defined by Java interface or implementation class. A service contract defined by a Java interface or implementation class is inferred to be non-conversational as defined by the SCA Assembly Specification unless it is decorated with @Conversational. In the latter case, @Conversational is used to declare that a component implementation offering the service implements conversational semantics as defined by the SCA Assembly Specification.

1.2.1.6. Callback Services

A callback interface is declared by using the @Callback annotation on the service interface implemented by a Java class.

1.2.2. References

References may be obtained through injection or through the ComponentContext API as defined in the SCA Java Common Annotations and API Specification. When possible, the preferred mechanism for accessing references is through injection.

1.2.2.1. Reference Injection

A Java implementation type may explicitly specify its references through the use of @Reference as in the following example:

```
public class ClientComponentImpl implements Client {
      private HelloService service;

      @Reference
      public void setHelloService(HelloService service) {
            this.service =  service;
      }
}
```

If @Reference marks a public or protected setter method, the SCA runtime is required to provide the appropriate implementation of the service reference contract as specified by the parameter type of the method. This must done by invoking the setter method an implementation instance. When injection occurs is defined by the scope of the implementation. However, it will always occur before the first service method is called.

If @Reference marks a public or protected field, the SCA runtime is required to provide the appropriate implementation of the service reference contract as specified by the field type. This must done by setting the field on an implementation instance. When injection occurs is defined by the scope of the implementation.

If @Reference marks a parameter on a constructor, the SCA runtime is required to provide the appropriate implementation of the service reference contract as specified by the constructor parameter during instantiation of an implementation instance.

References may also be determined by introspecting the implementation class according to the rules defined in Section **Error! Reference source not found.**.

References may be declared optional as defined by the Java Common Annotations and API Specification.

1.2.2.2. Dynamic Reference Access

References may be accessed dynamically through ComponentContext.getService() and ComponentContext.getServiceReference(..) methods as described in the Java Common Annotations and API Specification.

1.2.3. Properties

1.2.3.1. Property Injection

Properties may be obtained through injection or through the ComponentContext API as defined in the SCA Java Common Annotations and API Specification. When possible, the preferred mechanism for accessing propertoes is through injection.

A Java implementation type may explicitly specify its properties through the use of @Property as in the following example:

```
public class ClientComponentImpl implements Client {
      private int maxRetries;

      @Property
      public void setRetries(int maxRetries) {
            this. maxRetries = maxRetries;
      }
}
```

If @Property marks a public or protected setter method, the SCA runtime is required to provide the appropriate property value. This must done by invoking the setter method an implementation instance. When injection occurs is defined by the scope of the implementation.

If @Property marks a public or protected field, the SCA runtime is required to provide the appropriate property value. When injection occurs is defined by the scope of the implementation.

If @Property marks a parameter on a constructor, the SCA runtime is required to provide the appropriate property value during instantiation of an implementation instance.

Properties may also be determined by introspecting the implementation class according to the rules defined in Section **Error! Reference source not found.**.

Properties may be declared optional as defined by the Java Common Annotations and API Specification.

1.2.3.2. Dynamic Property Access

Properties may be accesses dynamically through ComponentContext. getProperty () method as described in the Java Common Annotations and API Specification.

1.2.4. Implementation Instance Instantiation

A Java implementation class must provide a public or protected constructor that can be used by the SCA runtime to instantiate implementation instances. The constructor may contain parameters; in the presence of such parameters, the SCA container will pass the applicable property or reference values when invoking the constructor. Any property or reference values not supplied in this manner will be set into the field or passed to the setter method associated with the property or reference before any service method is invoked.

The constructor to use is selected by the container as follows:

1. A declared constructor annotated with a @Constructor annotation.
2. A declared constructor that unambiguously identifies all property and reference values.

3. A no-argument constructor.

The @Constructor annotation must only be specified on one constructor; the SCA container must report an error if multiple constructors are annotated with @Constructor.

The property or reference associated with each parameter of a constructor is identified:

- by name in the @Constructor annotation (if present)
- through the presence of a @Property or @Reference annotation on the parameter declaration
- by uniquely matching the parameter type to the type of a property or reference

Cyclic references between components may be handled by the container in one of two ways:

- If any reference in the cycle is optional, then the container may inject a null value during construction, followed by injection of a reference to the target before invoking any service.
- The container may inject a proxy to the target service; invocation of methods on the proxy may result in a ServiceUnavailableException

The following are examples of legal Java component constructor declarations:

```
/** Simple class taking a single property value */
public class Impl1 {
    String someProperty;
    public Impl1(String propval) {...}
}

/** Simple class taking a property and reference in the constructor;
 * The values are not injected into the fields.
 *//
public class Impl2 {
    public String someProperty;
    public SomeService someReference;
    public Impl2(String a, SomeService b) {...}
}

/** Class declaring a named property and reference through the constructor */
public class Impl3 {
    @Constructor({"someProperty", "someReference"})
    public Impl3(String a, SomeService b) {...}
}

/** Class declaring a named property and reference through parameters */
public class Impl3b {
    public Impl3b(
        @Property("someProperty") String a,
        @Reference("someReference) SomeService b
        ) {...}
}

/** Additional property set through a method */
public class Impl4 {
    public String someProperty;
    public SomeService someReference;
    public Impl2(String a, SomeService b) {...}
    @Property public void setAnotherProperty(int x) {...}
}
```

1.2.5. Implementation Scopes and Lifecycle Callbacks

The Java implementation type supports all of the scopes defined in the Java Common Annotations and API Specification: STATELESS, REQUEST, CONVERSATION, and COMPOSITE. Implementations specify their scope through the use of the @Scope annotation as in:

```
@Scope(”COMPOSITE”)
public class ClientComponentImpl implements Client {
      // …
}
```

When the @Scope annotation is not specified on an implementation class, its scope is defaulted to STATELESS.

A Java component implementation specifies init and destroy callbacks by using @Init and @Destroy respectively. For example:

```
public class ClientComponentImpl implements Client {

      @Init
      public void init() {
            //…
      }

      @Destroy
      public void destroy() {
            //…
      }
}
```

1.2.5.1. *Java implementation classes that are CONVERSATION scoped may use @ConversationID to have the current conversation ID injected on a public or protected field or setter method. Alternatively, the Conversation API as defined in the Java Common Annotations and API Specification may be used to obtain the current conversation ID.***Conversational Implementation**

For the provider of a conversational service, there is the need to maintain state data between successive method invocations within a single conversation. For an Java implementation type, there are two possible strategies which may be used to handle this state data:

1. The implementation can be built as a stateless piece of code (essentially, the code expects a new instance of the code to be used for each method invocation). The code must then be responsible for accessing the conversationID of the conversation, which is maintained by the SCA runtime code. The implementation is then responsible for persisting any necessary state data during the processing of a method and for accessing the persisted state data when required, all using the conversationID as a key.

2. The implementation can be built as a stateful piece of code, which means that it stores any state data within the instance fields of the Java class. The implementation must then be declared as being of conversation scope using the @Scope annotation. This indicates to the SCA runtime that the implementation is stateful and that the runtime must perform correlation between client method invocations and a particular instance of the service implementation and that the runtime is also responsible for persisting and restoring the implementation instance if the runtime needs to clear the instance out of memory for any reason. (Note that conversations are potentially very long lived and that SCA runtimes may involve the use of clustered systems where a given instance object may be moved between nodes in the cluster over time, for load balancing purposes)

1.2.6. Accessing a Callback Service

Java implementation classes that require a callback service may use @Callback to have a reference to the callback service associated with the current invocation injected on a public or protected field or setter method.

1.2.7. Semantics of an Unannotated Implementation

The section defines the rules for determining properties and references for a Java component implementation that does not explicitly declare them using @Reference or @Property.

In the absence of @Property and @Reference annotations, the properties and references of a class are defined according to the following rules:

1. Public setter methods that are not included in any interface specified by an @Service annotation.
2. Protected setter methods
3. Public or protected fields unless there is a public or protected setter method for the same name

The following rules are used to determine whether an unannotated field or setter method is a property or reference:

1. If its type is simple, then it is a property.
2. If its type is complex, then if the type is an interface marked by @Remotable, then it is a reference; otherwise, it is a property.
3. Otherwise, if the type associated with the member is an array or a java.util.Collection, the basetype is the element type of the array or the parameterized type of the Collection; otherwise the basetype is the member type. If the basetype is an interface with an @Remotable or @Service annotation then the memberis defined as a reference. Otherwise, it is defined as a property.

The name of the reference or of the property is derived from the name found on the setter method or on the field.

1.2.8. Specifying the Java Implementation Type in an Assembly

The following defines the implementation element schema used for the Java implementation type:.

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

The implementation.java element has the following attributes:

- ***class (required)*** – the fully qualified name of the Java class of the implementation

1.2.9. Specifying the Component Type

For a Java implementation class, the component type is typically derived directly from introspection of the Java class .

A component type can optionally be specified in a side file. The component type side file is found with the same classloader that loaded the Java class. The side file must be located in a directory that corresponds to the namespace of the implementation and have the same name as the Java class, 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 [1]. If the component type information is in conflict with the implementation, it is an error.

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

2. Appendix

2.1. References

[1] SCA Assembly Specification

http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

[2] SCA Java Common Annotations and APIs

http://www.osoa.org/download/attachments/35/SCA_JavaCommonAnnotationsAndAPIs_V100.pdf