RFC 80 - Declarative Services

Final

53 Pages

Abstract

This RFC defines a declarative model for publishing, finding and binding services in the OSGi environment.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	29 May 2004	First Draft BJ Hargrave, IBM, hargrave@us.ibm.com
2nd draft	9 June 2004	Second Draft BJ Hargrave, IBM, hargrave@us.ibm.com
3rd draft	13 July 2004	Third Draft incorporating feedback from Nürnberg CPEG meeting. 1. Proxying service connection is no longer part of the spec proper, but the spec must support implementations which choose to proxy. 2. ComponentActivator interface was removed and replaced by a method convention that will be called via reflection. This avoids requiring public methods be added to the service object. 3. “Event” model added to “lookup” model for obtaining referenced services. 4. Added information on how ServiceFactory and ManagedServiceFactory function is addressed. 5. Added section to state that the Service Component runtime could be implemented outside of the framework (i.e. by a bundle). 6. Added Service Component enablement to allow for delayed enablement. BJ Hargrave, IBM, hargrave@us.ibm.com

Revision	Date	Comments
4th Draft	12 Sep 2004	Fourth Draft incorporating feedback from numerous reviewers as well as many changes to support the MEG application model. 1. Changed from single components.xml resource in META-INF directory to multiple files in the META-INF/components directory. This removed the bundle element from the xml. 2. Changed to support a per component xml property resources. This also allows the properties to be localized using normal PropertyResourceBundle naming. 3. Renamed service.name property to component.name. 4. Added com.isv prefix to names to use standard reverse domain naming style. 5. Added XML schema for component xml resources. 6. Removed specification of Bundle.getBundleContext method. How a Service Component runtime bundle gets a BundleContext from the OSGi framework is an implementation detail. BJ Hargrave, IBM, hargrave@us.ibm.com
Draft 5	4 October 2004	Fifth Draft incorporating feedback from Amsterdam CPEG meeting. 1. Updated XML schema to use namespace. This allows component descriptions to be embedded in larger XML files. This also removed the standard location in the JAR file for component descriptions. There is a new manifest header which names all the files containing component descriptions. Multiple scr:component elements can be placed in a single XML file. 2. Modified the properties handling to support multi-valued properties. The use of properties files to define properties is removed. The properties are now localized by using %variables like RFC 74. 3. Added ComponentFactory. 4. Added Javadoc for API. BJ Hargrave, IBM, hargrave@us.ibm.com
Draft 5.1	4 October 2004	Draft 5.1 adds a solution to allow a ComponentFactory to be registered as a wrapper type. BJ Hargrave, IBM, hargrave@us.ibm.com

Revision	Date	Comments
Draft 6	29 November 2004	Sixth Draft incorporating feedback from Nice CPEG meeting. 1. How SCR broadcasts FrameworkEvents is an implementation detail. 2. Removed localization since it was agreed that the properties should be localized "closer" to the actual presentation to the user. 3. Component factory renaming was removed. The value of the factory attribute is used as the value of the component.factory service property of the ComponentFactory service. 4. activate/deactivate/bind/unbind methods must now be public or protected. This is done to emulate "normal" inheritance behaviour. It also allows super methods to be called. 5. SCR must deactivate components when SCR bundle is stopped. 6. SCR must not maintain strong references to deactivated component instances to allow them (and possibly their bundle) to be garbage collected. 7. Made mention of RFC 103 regarding configurations. 8. Added support for properties to come from a properties file as well as property element. BJ Hargrave, IBM, hargrave@us.ibm.com
Final Review Draft	09 December 2004	Accepted all prior changes. Fixed some grammar errors and added manifest header examples. Recollected javadoc from membercvs. RFC is now in final review. BJ Hargrave, IBM, hargrave@us.ibm.com
	28 January 2005	Component implementation classes must have a public no argument (default) constructor. BJ Hargrave, IBM, hargrave@us.ibm.com
	20 March 2005	It is possible a component may obtain a partially activated component instance in its bind or activate method. This is possible if there is a circular dependency between components. BJ Hargrave, IBM, hargrave@us.ibm.com
	30 March 2005	Added ComponentContext.getServiceReference method to enable a component to easily obtain a reference to its service. Added component.id component property to easily identify unique component instances. BJ Hargrave, IBM, hargrave@us.ibm.com

Revision	Date	Comments
Final	27 May 2005	No Changes BJ Hargrave, hargrave@us.ibm.com

1 Introduction

Currently a bundle has to be active in order to offer a service(s) to other bundles. An active bundle consumes system resources such as ClassLoaders, loaded classes, created objects, etc. It is desirable to minimize the amount of resources consumed by a bundle, when a service(s) offered by it is not being used.

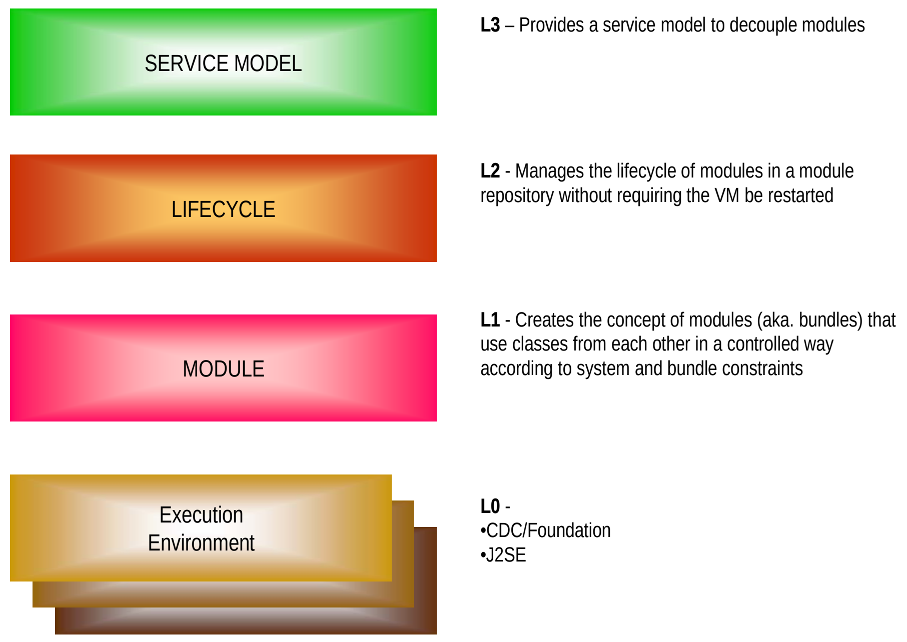
This RFC describes a declarative service model which allows a bundle to delay instantiating the service object until they are needed, thus minimizing the resource consumption at any point of time.

2 Application Domain

The OSGi framework from the OSGi Service Platform specification contains a procedural service model which provides a publish/find/bind model for using services. Though the OSGi environment is primarily a Java environment, OSGi specifications also provide support for JNI code.

In the OSGi environment, a service is declared by an interface and implemented by a class. This provides a desired separation between the details of the implementation of the service and the consumer of the service. As a result, the implementation of the service can be replaced by alternative implementations which meet the contract of the service interface. The alternative implementations may be bug fixes or functional enhancements, they may be from different vendors or they may offer different fidelity of service.

The OSGi framework may be conceived of using the following layering.


SERVICE MODEL
L3 – Provides a service model to decouple modules
LIFECYCLE
L2 - Manages the lifecycle of modules in a module repository without requiring the VM be restarted
MODULE
L1 - Creates the concept of modules (aka. bundles) that use classes from each other in a controlled way according to system and bundle constraints
Execution Environment
L0 -
•CDC/Foundation
•J2SE

L0 is a base Java runtime environment. L1 provides the concept of modules (aka. bundles). Each module declares its classloading interactions vis-à-vis other modules. L2 provides for the lifecycle of the modules to be managed in both a module repository and dynamically within the executing VM. Thus a module can be installed and made active without requiring the VM be restarted. Finally, L3 provide a publish/find/bind intra-VM service model to allow service consumers to be well decoupled from service providers.

The service model for the currently published OSGi specification (R3) is strictly a procedural model. A provider of a service must execute code which calls framework API to publish a service. A consumer of a service must execute code which calls framework API to find a service and then another framework API to bind to the service.

3 Problem Description

The current state of affairs has several areas in need of improvement.

1. There is no metadata within a bundle which describes the services it can provide and the services it will consume. So the framework does not know what services, if any, a bundle may provide or consume until the bundle executes and calls the framework. Note: The specification defines the Import-Service and Export-Service manifest headers, but they are not recognized at runtime by the framework. They are just hints to a management system.

2. In order to offer to provide a service, a bundle must be started and execute. This means the bundle must have a class loader, classes must be loaded and objects instantiated and some code executed which will explicitly publish the service. From a performance view, this is undesirable. We would rather be able to declaratively specify that a bundle will publish a service and delay creating class loaders, loading classes and instantiating objects until some other bundle actually desires to consume the service.

3. The current model requires that programmers be OSGi literate. As the OSGi service model API is specific to OSGi, it is problematic to entice programmers to use it. Furthermore, the OSGi environment can be a very dynamic environment. Services and bundles can come and go during the lifetime of the VM. Many programmers are not prepared to deal with such dynamism. So we desire a simplification of the programming model for using services in OSGi so that new programmers can declare their service relationships and then have them managed by the runtime rather than having to write code to call OSGi API.

4 Requirements

1. The OSGi specification has a legacy (3 public releases) that must be supported. “Legacy” bundles must be supported by the design for the OSGi declarative service model.

2. The OSGi market was and predominantly still is an constrained resource device market. So size matters. The service declaration should use an existing language to avoid having to develop new parsers and tools. It must be simple to parse with a very small parser.

3. The declarative service model must allow service objects to be lazily activated (and eagerly deactivated). This will allow the framework to delay creating service objects (and thus their classloader) until needed.

4. The programming model for using declarative services must be very simple and not require the programmer to learn a complicated API model.

5. The declarative services runtime should be implementable as a bundle running on an OSGi framework. This requirement will necessitate a means to access framework features not available through the public API.

5 Technical Solution

This RFC describes a declarative service model called ***Service Component***. This document also uses the generic term “component” to refer to Service Components.

5.1 Conceptual Model

5.1.1 What are Service Components

We define a Service Component to mean a set of function:

- Contained within a ***bundle***
- Whose ***lifecycle*** is contained within its bundle’s lifecycle
- Identified by an ***implementation*** class
- Optionally ***provides*** a service which is registered as an OSGi service
- Optionally ***references*** other OSGi services (which may be provided by other components)
- Optionally ***configured***
- Can be ***lazily activated***. This means the creation and activation of a component can be delayed until the component is actually used.

A Service Component can provide services to and/or reference services from other Service Components and other OSGi bundles. Services provided or used by a Service Component are always manifested as services registered in the OSGi service registry. Service Components are configured using the existing OSGi configuration infrastructure: Configuration Admin. A Service Component’s references to other services are not hard-coded. The references are declared via symbolic names whose target service is configurable. A default target service is also declared.

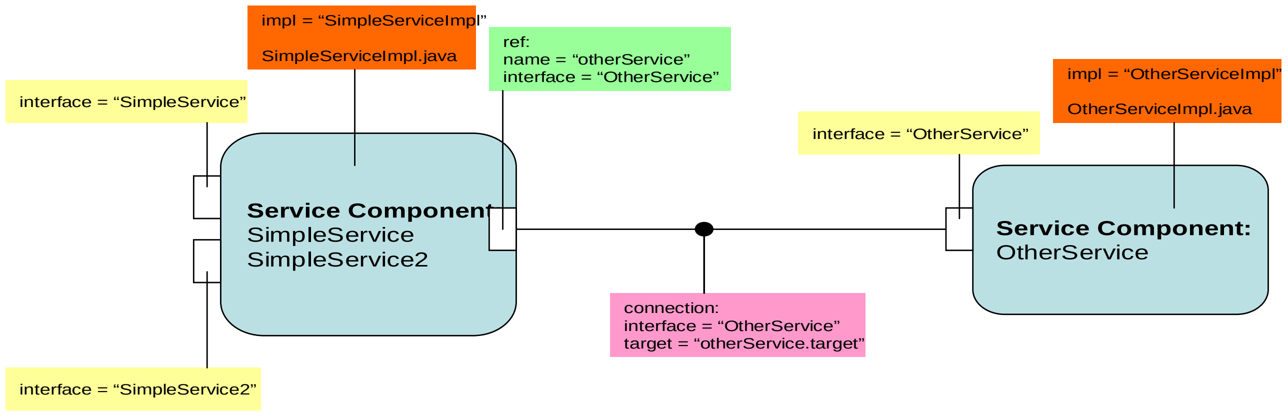
5.1.2 Example

This example shows the definition of two Service Components with the information used to resolve their references. An extended version of this example is discussed in more detail later.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="com.vendor.SimpleService"
   xmlns:scr=http://www.osgi.org/xmlns/scr/v1.0.0>
   <implementation class="com.vendor.simpleclient.impl.SimpleServiceImpl" />
   <service>
      <provide interface="org.simpleclient.interfaces.SimpleService"/>
      <provide interface="org.simpleclient.interfaces.SimpleService2"/>
   </service>
   <reference name="otherService"
         interface="org.simpleservice.interfaces.OtherService"/>
</scr:component>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="com.vendor.OtherService"
   xmlns:scr=http://www.osgi.org/xmlns/scr/v1.0.0>
   <implementation class="com.vendor.other.impl.OtherServiceImpl" />
   <service>
      <provide interface="org.simpleservice.interfaces.OtherService"/>
   </service>
</scr:component>
```

The following picture shows two Service Components with their interface, implementation, reference, and connection.


impl = “SimpleServiceImpl”
SimpleServiceImpl.java
ref:
name = “otherService”
interface = “OtherService”
impl = “OtherServiceImpl”
OtherServiceImpl.java
interface = “SimpleService”
interface = “OtherService”
Service Component
SimpleService
SimpleService2
Service Component:
OtherService
connection:
interface = “OtherService”
target = “otherService.target”
interface = “SimpleService2”

5.2 A Sample Application

In this chapter you get a first view of an application built using Service Components. The application illustrates the key Service Component programming model concepts.

Service Components (the components) are shown with their interfaces, references and implementations. The Service Component implementations in this example are all based on simple Java classes. You will see how a Service Component uses another Service Component using references and the Service Component programming model, and you will see how Service Component references are resolved by connecting them to other Service Components.

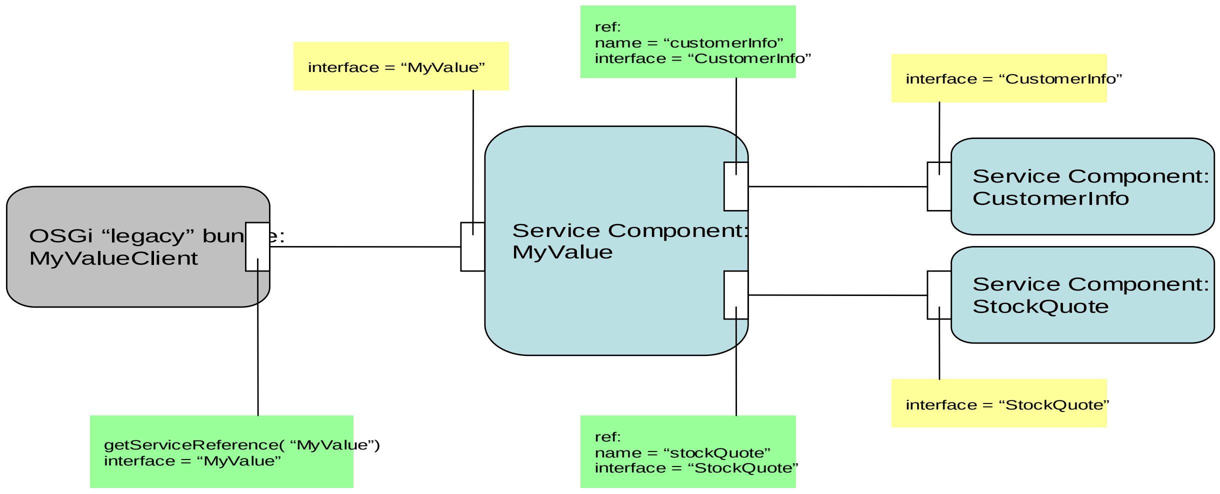
We also show how a “legacy”[1] OSGi bundle uses a Service Component.

5.2.1 Scenario

The application that we are looking at provides a MyValue Service Component which given my customer identification calculates my current worth based on the stock that I own. Users can access the MyValue Service Component via the MyValueClient which is some legacy OSGi bundle.

The following diagram shows the involved Service Components and legacy OSGi bundles with their interfaces and references, and how they are connected.

[1] In this document, we use the term “legacy” bundle to refer to a bundle which uses the traditional service API provided by the OSGi framework. This API includes the registerService, getServicesReference and getService methods of BundleContext.



The MyValue Service Component is a little business process that references two Service Components to accomplish its business goal. One is called CustomerInfo, given the customer identification it provides the symbol of the stock that I own as well as the number of shares. The other one is StockQuote, which given the stock symbol returns the current value of the stock.

MyValue is called by the MyValueClient bundle which provides the customer identification as input. The MyValue Service Component first calls the CustomerInfo Service Component, and with the returned symbol it then calls StockQuote. After that it multiplies the number of shares returned from the CustomerInfo Service Component with the quote returned by the StockQuote Service Component, and returns the result to the MyValueClient.

5.2.2 Development Process

We are using the "Notepad Development Process" to author a set of Service Components bundles in the file-system using a simple notepad editor. We develop the resources defined by the programming model in source form, e.g. Java, XML, and so on. Using the Notepad Development Process gives you an undisturbed view of the programming model.

The process for developing a Service Component bundle is as follows:

- Create a folder for each bundle in file-system to represent the OSGi bundle
- Create folders within each bundle folder to hold the Java code
- Add metadata to the bundle's manifest to specify the location of the component description
- Deploy bundles to an ***OSGi framework*** supporting a Service Component runtime (SCR).

5.2.3 Developing the Bundles

5.2.3.1 Bundle Folders

In this step we create the new bundle folders. The following shows the bundle folder's contents after this step is completed.

```
MyValue/
    <empty>
```

```
CustomerInfo/
    <empty>
```

```
StockQuote/
    <empty>
```

5.2.3.2 CustomerInfo Bundle

In this step we create the CustomerInfo Service Component bundle with its interface and implementation artifacts. The following shows the CustomerInfo folder contents after this step is completed.

```
CustomerInfo/
    META-INF/
         MANIFEST.MF
    com/isv/service/customerinfo/
         CustomerInfo.java
         Customer.java
    com/isv/service/customerinfo/impl/
         CustomerInfoImpl.java
         customerinfo.xml
```

The following snippet shows the Java interface and implementation class of the CustomerInfo Service Component. As you can see implementing a Service Component is about Java interfaces and simple Java classes.

The CustomerInfo Service Component implements a method called getCustomer. Input to the method is a customerID in form of a String, the return is an instance of the Java class Customer. In a real world scenario the implementation would take the customerID and use some business logic to interact with some data store. The data store returns the detailed customer information which in form of a Customer object becomes the return of the getCustomer method. The purpose of the CustomerInfo Service Component is to insulate all application business logic from the infrastructure details of accessing the data store, exposing a simple business interface to the application logic, which in this example is in a component called MyValue. In the sample implementation of getCustomer we just hard code the return of a Customer object.

```
package com.isv.service.customerinfo;

public interface CustomerInfo {
   public Customer getCustomer(String customerID);
}

package com.isv.service.customerinfo;

public class Customer {
   // implementation here...
}

package com.isv.service.customerinfo.impl;
import com.isv.service.customerinfo.*;

public class CustomerInfoImpl implements CustomerInfo {
   public Customer getCustomer(String customerID) {
       // here is where the interaction with business logic would go
       Customer cust = new Customer();

       cust.setCustNo(customerID);
       cust.setFirstName("Victor");
       cust.setLastName("Hugo");
       cust.setSymbol("IBM");
       cust.setNumShares(100);
       cust.setPostalCode("10589");
       cust.setErrorMsg("");

       return cust;
   }
}
```

In the next snippet we see the CustomerInfo bundle's component description. This is what makes CustomerInfo a Service Component. The resource containing the component description is specified by naming it on the `Service-Component` manifest header.

```
Service-Component: com/isv/service/customerinfo/impl/customerinfo.xml
```

The component description shows the interface that the Service Component implements as well as the implementation class.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="com.isv.service.customerinfo.CustomerInfo"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0">
  <implementation class="com.isv.service.customerinfo.impl.CustomerInfoImpl"/>
  <service>
     <provide interface="com.isv.service.customerinfo.CustomerInfo"/>
  </service>
</scr:component>
```

5.2.3.3 StockQuote Bundle

In this step we create the StockQuote Service Component bundle with its interface and implementation artifacts. The following shows the StockQuote folder contents after this step is completed.

```
StockQuote/
  META-INF/
     MANIFEST.MF
  com/isv/service/stockquote/
     StockQuote.java
  com/isv/service/stockquote/impl/
     StockQuoteInfoImpl.java
     stockquote.xml
```

The following snippet shows the Java interface and implementation class of the StockQuote Service Component. Again you can see implementing a Service Component is about Java interfaces and simple Java classes.

The StockQuote Service Component implements a method called getQuote. Data gets exchanged with this method in form of simple Java types, i.e. a String as input, and a float as return. In a real world scenario the implementation would take the symbol String input and use some communications to obtain the current quote.. The returned quote becomes the return of the getQuote method. In the sample snippet we just hard code the return of a quote.

```
package com.isv.service.stockquote;

public interface StockQuote {
   public float getQuote(String symbol);
}

package com.isv.service.stockquote.impl;
import com.isv.service.stockquote.*;

public class StockQuoteImpl implements StockQuote {
   public float getQuote(String symbol) {
       // here is where the interaction with business logic would go
      return 100.0f;
   }
}
```

In the next snippet we see the StockQuote bundle's component description. This is what makes StockQuote a Service Component. The resource containing the component description is specified by naming it on the `Service-Component` manifest header.

```
Service-Component: com/isv/service/stockquote/impl/stockquote.xml
```

The component description shows the interface that the Service Component implements as well as the implementation class.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="com.isv.service.stockquote.StockQuote"
       xmlns:scr=“http://www.osgi.org/xmlns/scr/v1.0.0”>
   <implementation class="com.isv.service.stockquote.impl.StockQuoteImpl"/>
   <service>
       <provide interface="com.isv.service.stockquote.StockQuote"/>
   </service>
</scr:component>
```

5.2.3.4 MyValue Bundle

In this step we create the MyValue Service Component bundle with its interface and implementation artifacts. The following shows the MyValue folder contents after this step is completed.

```
MyValue/
   META-INF/
      MANIFEST.MF
   com/isv/process/myvalue/
      MyValue.java
   com/isv/process/myvalue/impl/
      MyValueImpl.java
      myvalue.xml
```

The following snippet shows the Java interface of the MyValue Service Component. It has the getMyValue method that given my customerID returns my current worth.

```
package com.isv.process.myvalue;

public interface MyValue {
   public float getMyValue(String customerID) throws MyValueException;
}
```

In the next snippet we see the Java implementation class for the MyValue Service Component. It is a little process scripted in Java which uses the Service Component programming model to interact with the CustomerInfo and StockQuote services it references.

As you can see the MyValue Service Component implementation doesn't use the other Service Components explicitly, it uses logical reference names for locating them, i.e. "customerInfo", and "stockQuote". In the Serivce Component programming model there are two techniques by which a service can be acquired. The first technique, which may be thought of as a "lookup" model, requires the programmer to write one line of code to obtain a service, i.e. ***context.locateService("<reference-name>")***. The second technique, which may be thought of as an "event" model, requires the programmer to write a bind and unbind method for the service and specify those methods in the reference element of the component description. The bind method is called to provide the service object and the unbind method is called to withdraw the service object. Both techniques may be used and each has its positives and negatives. See 5.4.1 Obtaining a Service for more details.

The following snippet uses the "lookup" model.

```
package com.isv.process.myvalue.impl;
import org.osgi.service.component.ComponentContext;
import com.isv.process.myvalue.*;
import com.isv.service.stockquote.*;
import com.isv.service.customerinfo.*;

public class MyValueImpl implements MyValue {
   private ComponentContext context;

   // component activator
   protected void activate(ComponentContext context) {
      // prepare to be useable as a service
      this.context = context;
   }

   public float getMyValue(String customerID) throws MyValueException {
      // variables
      Customer customer = null;
      float quote = 0;
      float value = 0;

      // invoke
      CustomerInfo cInfo = (CustomerInfo)context.locateService("customerInfo");
      customer = cInfo.getCustomer(customerID);

      if (customer.getErrorMsg().equals("")) {
         // invoke
          StockQuote sQuote = (StockQuote)context.locateService("stockQuote");
          quote =  sQuote.getQuote(customer.getSymbol());

         // assign
          value = quote * customer.getNumShares();
      } else {
         // throw
          throw new MyValueException(customer.getErrorMsg());
      }
      // reply
      return value;
   }

   // component dectivator
   protected void deactivate(ComponentContext context) {
      // clean up; we are no longer a useable service
      this.context = null;
   }
}
```

In the next snippet we see the MyValue bundle’s component description. This is what makes MyValue a Service Component. The resource containing the component description is specified by naming it on the `Service-Component` manifest header.

```
Service-Component: com/isv/process/myvalue/impl/myvalue.xml
```

The component description shows the interface that the Service Component implements as well as the implementation class.

Here the new thing is that the MyValue Service Component is one that uses other Service Components, you see this expressed in the Service Component component description in the reference elements. The reference elements shows that the MyValue Service Component implementation will locate a Service Component under the name “customerInfo” which it expects to implement the CustomerInfo Java interface; it further is locating a Service Component under the name “stockQuote” which it expects to implement the StockQuote Java interface.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="com.isv.process.myvalue.MyValue"
    xmlns:scr=“http://www.osgi.org/xmlns/scr/v1.0.0”>
  <implementation class="com.isv.process.myvalue.impl.MyValueImpl" />

  <service>
    <provide interface="com.isv.process.myvalue.MyValue"/>
  </service>

  <reference name="customerInfo"
     interface="com.isv.service.customerinfo.CustomerInfo"
     cardinality="1..1"
     policy="static"
     target="(component.name=com.isv.service.customerinfo.CustomerInfo)"
  />

  <reference name="stockQuote"
     interface="com.isv.service.stockquote.StockQuote"
     cardinality="1..1"
     policy="static"
     target="(component.name=com.isv.service.stockquote.StockQuote)"
  />
</scr:component>
```

We now repeat the snippet but this time using the “event” model. This allows us to see how the code will look using the other technique. We add bind and unbind methods so the component can be provided with the services. In this example, we no longer need the ComponentContext object since all it was used for in the previous example was to obtain the services.

```java
package com.isv.process.myvalue.impl;
import com.isv.process.myvalue.*;
import com.isv.service.stockquote.*;
import com.isv.service.customerinfo.*;

public class MyValueImpl implements MyValue {
   private CustomerInfo cInfo;
   private StockQuote sQuote;

   protected void bindCustomerInfo(CustomerInfo cInfo) {
       this.cInfo = cInfo;
   }
   protected void unbindCustomerInfo(CustomerInfo cInfo) {
       this.cInfo = null;
   }

   protected void bindStockQuote(StockQuote sQuote) {
       this. sQuote = sQuote;
   }
   protected void unbindStockQuote(StockQuote sQuote) {
       this. sQuote = null;
   }

   public float getMyValue(String customerID) throws MyValueException {
      // variables
      Customer customer = null;
      float quote = 0;
      float value = 0;

      // invoke
      customer = cInfo.getCustomer(customerID);

      if (customer.getErrorMsg().equals("")) {
         // invoke
           quote =  sQuote.getQuote(customer.getSymbol());

         // assign
           value = quote * customer.getNumShares();
      } else {
         // throw
           throw new MyValueException(customer.getErrorMsg());
      }
      // reply
      return value;
   }
}
```

We now show the component description which has been modified to specify the bind and unbind methods. The SCR will call the bind and unbind methods to supply the Service Component with the referenced services.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="com.isv.process.myvalue.MyValue"
      xmlns:scr=“http://www.osgi.org/xmlns/scr/v1.0.0”>
   <implementation class="com.isv.process.myvalue.impl.MyValueImpl" />

   <service>
      <provide interface="com.isv.process.myvalue.MyValue"/>
   </service>

   <reference name="customerInfo"
       interface="com.isv.service.customerinfo.CustomerInfo"
       cardinality="1..1"
       policy="static"
       target="(component.name=com.isv.service.customerinfo.CustomerInfo)"
       bind=”bindCustomerInfo”
       unbind=”unbindCustomerInfo”
      />

   <reference name="stockQuote"
       interface="com.isv.service.stockquote.StockQuote"
       cardinality="1..1"
       policy="static"
       target="(component.name=com.isv.service.stockquote.StockQuote)"
       bind=”bindStockQuote”
       unbind=”unbindStockQuote”
   />
</scr:component>
```

5.2.3.5 Connections Between Service Components

Up to this point we authored three Service Components that don’t know about each other. What we know is that the MyValue Service Component needs two Service Components: one implementing the CustomerInfo and the other implementing the StockQuote interface to function.

The part that is missing is that we have to connect the references of MyValue to concrete Service Components that implement the Java interfaces expressed by the references. After a Service Component is activated, it is published by its interface names and properties in the OSGi service registry. To resolve a reference to specific services, the OSGi service registry is queried for services which match the reference’s **target** attribute. See 5.3.4 Service Connections for more information.

5.2.3.6 MyValueClient Bundle

We now discuss the MyValueClient bundle. This bundle is not a Service Component bundle but can still access Service Components as normal OSGi services using existing OSGi framework methods.

The following snippet shows code in the MyValueClient bundle which find and calls the MyClient Service Component using the MyValue interface.

```
...
   String customerID = "12345";
   ServiceTracker tracker = new ServiceTracker(context,
       "com.isv.process.myvalue.MyValue", null);
   tracker.open();
   MyValue myValue = (MyValue)tracker.getService();
   float value = myValue.getMyValue(customerID);
   tracker.close();
...
```

We are done with authoring our first Service Component bundles at this point. They are now ready for deployment on a Service Component enabled OSGi framework.

5.3 Service Component Model

This chapter describes the Service Component component model in more detail. We will show how you define a Service Component component, how you connect Service Components, and what makes a Service Component bundle.

5.3.1 Service Component Declaration

The Service Component is defined by a simplified declarative model for authoring and using OSGi services. A Service Component may define an OSGi service which will be published in the OSGi service registry after the Service Component is activated. A Service Component may also use other OSGi services including those defined by other Service Components.

The Service Component model simplifies the task of authoring OSGi services by using the component description to specify the details of the service and allowing the SCR to perform the work of publishing the service. This avoids the need to the programmer to write code to call OSGi framework methods and allows Service Components to be lazily loaded. As a result, bundles do not even have to provide a BundleActivator class to provide services.

The Service Component model also simplifies that task of using OSGi services. The SCR can manage the dependencies a Service Component has on other services and only activate the Service Component when it dependencies are met. It can also deactivate the Service Component if its dependencies no longer are satisfied. The Service Component model also allows the SCR to manage the dynamism of service dependencies.

Service Components are described using a simple-to-parse XML grammar described below. Footprint and simplicity requirements for the runtime dictate the use of a simple declarative data format. We choose XML as the language and use a simple schema that can be parsed with a very lightweight footprint parser.

The manifest header `Service-Component` must be specified in the bundle manifest. The value of the header is a comma seperated list of XML resources within the bundle.

```
Service-Component ::= description-entry ( ',' description-entry )*

description-entry ::= <fully qualified bundle entry name>
```

The SCR will parse each listed resource and process all component elements in the SCR XML namespace. All other elements and their attributes are ignored by the SCR. This allows component descriptions to be “embedded” in other XML documents. See 5.6 for the XML schema and namespace definition.

Fragment bundles may not describe components. Thus the SCR does not examine fragments for component descriptions.

The following outlines the content of the component description.

```
<scr:component name="component name"
   autoenable=”boolean”?
   factory=”component.factory property value”?
   xmlns:scr=“http://www.osgi.org/xmlns/scr/v1.0.0”>
   <implementation class="Java implementation class"/>

   <property name="property name" value="property value"?  type="property type"?>
       property value*
   </property>*
   <properties entry=”bundle entry name”/>*

   <service servicefactory=”boolean”?>
       <provide interface="Java interface type"/>+
   </service>?

   <reference name="reference name"
       interface="Java interface type"
       cardinality="reference cardinality"?
       policy="reference policy"?
       target=”target filter”?
       bind=”bind method”?
       unbind=”unbind method”?
   />*
</scr:component>
```

A component description uses the following major elements to define a Service Component. All Service Components are logically part of the bundle in which they are declared. A Service Component can only be activated if its containing bundle is started. If the containing bundle is stopped, then all contained Service Components are deactivated.

- ***component*** – Specifies a Service Component. A bundle can have multiple Service Components. Each Service Component is described in a separate component element. There can be multiple XML resources containing component elements and each XML resource can contain multiple component elements.

- ***service*** –The service element is optional. If present then the Service Component will be registered as a service only if the depenedencies of the component are met and the component can be activated. The service element groups provide elements.

- ***provide*** – A Service Component may implements one or more interfaces. Service Component interfaces are specified as Java language interfaces. Each ***provide element*** lists a Java interface implemented by the Service Component. There must be at least one provide element within the service element.

- ***reference*** – Service Components can make use of other Service Components or OSGi services. Service Components do not hard-code which other services they use – they declare "soft-links" called references. A Service Component has zero, one or more typed references to other services.

- ***implementation*** – A Service Component is implemented as a Java class. The Java class specified must implement the Java interfaces specified by any provide elements.

- ***property, properties*** – Service Components have properties associated with them. A Service Component's name is used to locate its configured properties in the OSGi Configuration Admin service. The property elements provide default or supplemental property values if not overridden by the properties retrieved from Configuration Admin.

The elements and their use in the Service Component model will be described in more detail in the following sections.

If the SCR detects an error when processing a component description, the SCR will broadcast as a FrameworkEvent.ERROR and will ignore the component description. Errors can include XML parsing errors and invalid descriptions.

5.3.1.1 Component

A Service Component has a name. The name is specified in the ***name attribute*** of the ***component element***. The value of the Service Component property, **component.name,** is set to the name of the Service Component. This property cannot be overriden by specified properties or configured properties.

The Service Component name uniquely identifies the Service Component within the OSGi framework. The component name is used as a PID to retrieve component properties from the OSGi Configuration Admin service if present. The properties for a Service Component are described in more detail in 5.3.1.5 Properties.

The ***autoenable*** attribute controls whether the component is enabled upon starting of the bundle. The default value is "true". If autoenable="false" is specified then the Service Component is disabled until ComponentContext.enableComponent is called. This can be useful when a bundle has to perform some lengthy initialization prior to making it's services available to other bundles. At the end of the initialization activity, the bundle can call the enableComponent method.

If the ***factory*** attribute is specified, instead of normal component activation, the SCR will register a ComponentFactory service which can be used to create instances of the component. See 5.3.3 Service Component Factory for more information.

In the next snippet we see a ***component*** description with the ***name*** attribute.

```
<component name="com.isv.service.customerinfo.CustomerInfo">
   ...
<component>
```

5.3.1.2 Service

A Service Component may implement one or more ***interfaces***. A Service Component interface is specified as a Java language interface by a ***provide element***. The Service Component implementation class must implement each specified interface. If the ***service element*** is present then at least one provide element must be specified and the Service Component will be published as an OSGi service if the Service Component dependencies are met. If one of the Service Component's dependencies becomes unavailable, then the Service Component's service will be unpublished.

Note: Service Components may be declared to provide a Java class as well as or instead of a Java interface. This is not recommended but is supported. If the at least one of the provide elements refers to a Java class, then the service connection cannot be proxied. See 5.3.4.3 Proxying Service Connections.

The ***parameter and return types*** of the Service Component methods are described using Java language classes, or simple Java language types.

In the next snippet we see the ***service*** element definition with a ***provide*** element with the ***interface*** attribute.

```
<service>

   <provide interface="com.isv.service.customerinfo.CustomerInfo"/>
</service>
```

5.3.1.3 Reference

A Service Component ***reference*** defines a dependency upon another service that will be located at run-time. The ***policy*** and ***cardinality*** of the reference will control how the Service Component is activated and deactivated relative to the availability of the dependent service.

A reference comprises the following attributes:

- ***name*** is a string adhering to the NMToken [3] definition which defines a component-specific reference name ("soft-link") for the service dependency. This attribute must be specified.

- ***interface*** is the fully qualified name of a Java language interface that is used by this Service Component to access the service. The service provided to the component must be type compatible with this interface. This attribute must be specified.

- ***cardinality*** indicates the number of services, matching this reference, which will bind to this Service Component. Possible values are: 0..1, 0..n, 1..1 (i.e. exactly one), 1..n (i.e. at least one). This attribute is optional. If it is not specified, then a cardinality of "1..1" is used.

- ***policy*** indicates when this Service Component may be activated[2] or deactivated relative to the availability of it referenced dependents. This attribute is optional. Possible values are: static, dynamic. If it is not specified, then a policy of “static” is used.

- ***target*** indicates the selection filter to be used to select the desired service for the reference. The value is an OSGi filter string which is used to select the target service(s) from the OSGi service registry. See 5.3.4 Service Connections for more information. The value of the target attribute will be overridden by the <reference-name>.target component property if present. This attribute is optional. If it is not specified and there is no <reference-name>.target component property, then the selection filter used to select the desired service is “(objectClass=”+<interface-name>+”)”.

- ***bind*** indicates the name of the component method to call for each selected service bound to this Service Component. This attrbute is optional. If the policy for this reference is “static” then the bind method will be called once for each bound service prior to calling the activate method. If the policy is “dynamic”, then the bind method may be called before and after calling the activate method. If the bind attribute is specificed, the unbind attribute must also be specified. See 5.4.1 for more information on bind and unbind.

- ***unbind*** indicates the name of the component method to call for each selected service unbound from this Service Component. This attrbute is optional. If the policy for this reference is “static” then the unbind method will be called once for each bound service after calling the deactivate method. If the policy is “dynamic”, then the unbind method may be called before and after calling the deactivate method. If the unbind attribute is specificed, the bind attribute must also be specified. See 5.4.1 for more information on bind and unbind.

The following table describes how the values of the ***cardinality*** and ***policy*** attributes affect the activation and deactivation of the Service Component and its binding to referenced services.

cardinality, ***policy***	**Comment**
1..1, static	The Service Component may be activated and bound to a **single** service which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component cannot be activated. **Bind**: Immediately prior to activation, the Service Component will bind to one service selected by the reference. **Unbind**: Immediately after deactivation, the Service Component will unbind from the bound service. **Deactivation**: If the service to which it is bound is unpublished, the Service Component will be deactivated.

[2] See 5.3.2 for more information on component activation and deactivation.

cardinality*, *policy	**Comment**
1..1, dynamic	The Service Component may be activated and bound to a **single** service which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component cannot be activated. **Bind**: Immediately prior to activation, the Service Component will bind to one service selected by the reference. **Rebind**: If the bound service is unpublished, the Service Component will remain activated and will dynamically unbind from the bound service and will dynamically bind to another service which is selected by the reference. **Unbind**: Immediately after deactivation, the Service Component will unbind from the bound service. **Deactivation**: If no services remain published which are selected by the reference, the Service Component will be deactivated.
1..n, static	The Service Component may be activated and bound to **at least one** service which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component cannot be activated. **Bind**: Immediately prior to activation, the Service Component will bind to all services selected by the reference. **Unbind**: Immediately after deactivation, the Service Component will unbind from any bound service. **Deactivation**: If any service to which it is bound is unpublished, the Service Component will be deactivated.

cardinality*, *policy	**Comment**
1..n, dynamic	The Service Component may be activated and bound to **at least one** service which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component cannot be activated. **Bind**: Immediately prior to activation, the Service Component will bind to all services selected by the reference. **Rebind**: The Service Component will dynamically bind to all services which are selected by the reference as they are published and dynamically unbind as they are unpublished. **Unbind**: Immediately after deactivation, the Service Component will unbind from any bound service. **Deactivation**: If no services remain published which are selected by the reference, the Service Component will be deactivated.
0..1, static	The Service Component may be activated and bound to **at most one** service which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component can still be activated without a bound service. **Bind**: Immediately prior to activation, the Service Component may bind to one service selected by the reference. **Unbind**: Immediately after deactivation, the Service Component will unbind from any bound service. **Deactivation**: If the service to which it is bound is unpublished, the Service Component will be deactivated.

cardinality*, *policy	**Comment**
0..1, dynamic	The Service Component may be activated and bound to **at most one** service which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component can still be activated without a bound service. **Bind**: Immediately prior to activation, the Service Component may bind to one service selected by the reference. **Rebind**: If the bound service is unpublished, the Service Component will remain activated and will dynamically unbind from the bound service and will dynamically bind to another service which is selected by the reference. **Unbind**: Immediately after deactivation, the Service Component will unbind from any bound service. **Deactivation**: The Service Component will be deactivated when its containing bundle is stopped.
0..n, static	The Service Component may be activated and bound to **any** service which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component can still be activated without a bound service. **Bind**: Immediately prior to activation, the Service Component will bind to all services selected by the reference. **Unbind**: Immediately after deactivation, the Service Component will unbind from any bound service. **Deactivation**: If any service to which it is bound is unpublished, the Service Component will be deactivated.

cardinality, ***policy***	**Comment**
0..n, dynamic	The Service Component may be activated and bound to **any** services which is selected by the reference. **Activation**: If no service is currently published which would be selected by the reference, the Service Component can still be activated without a bound service. **Bind**: Immediately prior to activation, the Service Component will bind to all services selected by the reference. **Rebind**: The Service Component will dynamically bind to all services which are selected by the reference as they are published and dynamically unbind as they are unpublished. **Unbind**: Immediately after deactivation, the Service Component will unbind from any bound service. **Deactivation**: The Service Component will be deactivated when its containing bundle is stopped.

The rules on how references get resolved to a suitable target service are described in 5.3.4 Service Connections.

5.3.1.4 Implementation

A Service Component is implemented as a Java language class. The data exchange semantics between a Service Component user and a Service Component implementation is *by-reference*.

The Java class specified must implement the Java interfaces specified by all ***provide*** elements and must be declared public and have a public default constructor so an instance of it can be created with `Class.newInstance`.

In the next snippet we see the ***implementation*** element with the ***class*** attribute.

```
<implementation class="com.isv.service.customerinfo.impl.CustomerInfoImpl"/>
```

5.3.1.5 Properties

A Service Component has a set of properties called the component properties. The component properties are specified in 3 places.

1. Properties specifed in the properties argument of ComponentFactory.newInstance method.
2. Properties retrieved from the OSGi Configuration Admin service having the **service.pid** matching the name of the Service Component. See 5.3.1.5.2 for more information.
3. ***property*** and ***properties*** elements in the Service Component's ***component*** element.

A property of a give name specified in the component factory's argument take precedence over the same property specified in the component's configuration which take precedence over the same property specified in ***property*** or ***properties*** elements. This precedence behavior allows certain default values to be specified in the ***property*** or ***properties*** elements and also allows the bundle to be configured to use different values. Thus the management system for the platform can configure more appropriate properties for a component including the desired connections between a Service Component and services upon which it depends.

property and ***properties*** elements are processed in top to bottom order. This allows later elements to override property values defined by earlier elements. There can be many ***property*** and ***properties*** elements and they may be interleaved.

The SCR always adds two properties, **component.name** and **component.id**, which cannot be overridden. The value of the **component.name** property is the name of the Service Component which is specified by the ***name*** attribute of the ***component*** element. The value of the **component.id** property is generated by SCR when a component instance is created. SCR assigns a unique value that is larger than all previously assigned values since SCR was started. These values are NOT persistent across restarts of SCR.

There are some special property names which are called **connection properties**. These are discussed in more detail in 5.3.4 Service Connections.

When the Service Component is activated, it is passed a ComponentContext object. The Service Component's implementation can access the component properties using the ComponentContext.getProperties method.

5.3.1.5.1 Multi-valued Property

The ***property*** element supports properties with multiple values.To specifiy multiple values for a property, do not use the ***value*** attribute. Instead specify the values as the body of the ***property*** element. If the ***value*** attribute is specified, the body of the ***property*** element is ignored.

The type of the value of the property will be an array based upon the ***type*** attribute.

- If the value of the ***type*** attribute is "String" then the type of property value will be String[]
- For the other values of the ***type*** attribute, the type of the property value will be a primitive array of the unboxed specified type. e.g. type="Integer" yields int[].

The body of the ***property*** element is treated as follows:

- All blank lines are ignored.
- All non-blank lines are stripped of leading and trailing spaces (String.trim). The result is added to the array.

So in the following example, the type of the value of the property "multi.valued.property" is String[] and the value is {"value1", "value2"}.

```
<property name="multi.valued.property" type=”String”>
   value1
   value2
</property>
```

5.3.1.5.2 Properties

The ***properties*** element references an entry in the bundle whose contents are a standard Java properties file. The entry is read and processed to obtain properties and their values.

```
<properties entry="com/isv/service/customerinfo/impl/customerinfo.properties" />
```

5.3.1.5.3 Configured Properties

The name of the component will be used to locate a Configuration or Configurations in the ConfigurationAdmin configuration database. The component name is used as a **service.pid** to look up the configurarion. A service.pid may be configured to be a ManagedService (single configuration) or a ManagedServiceFactory (multiple configurations). If no such configuration is present for the component name, then there are no configured properties for the component.

5.3.2 Service Component Activation and Deactivation

A Service Component can only be activated if the bundle containing it is started. Likewise, Service Components in a bundle are deactivated when the bundle is stopped.

A bundle written using Service Components will generally not need a traditional BundleActivator. The SCR will manage the activation/deactivation of components as necessary. This allows the SCR to delay the activation of service components until the component is actually used. This will also allow the framework to delay the creation of a bundle’s classloader. The SCR may also deactivate a component that is no longer being used. Service Components are always activated using a new object instance. This means that when a component is deactivated, that component instance will never be reused. (Note: This is the same behavior as BundleActivator instances.) The SCR must not maintain strong references to deactivated component instances to allow them to be garbage collected.

5.3.2.1 Service Registration

If a Service Component has a **service** element, i.e. the Service Component provides a service, then the SCR must register the service on behalf of the Service Component if the Service Component’s dependencies are met. The SCR can then delay the activation of the Service Component until the provided service is acquired via BundleContext.getService or a ComponentContext.locateService method. The SCR may deactivate the Service Component when it’s provided service is no longer being used. This can allow the bundle containing the component to be garbage collected. That is: the bundle’s classloader could be garbage collected because SCR maintains no strong references to any object or class from the bundle.

If a Service Component does not have a service element, i.e. the Service Component does not provide a service, or the Service Component is not a component factory, the SCR must activate the component if the bundle that contains the Service Component is started and the Service Component’s dependencies are met.

When the SCR registers a service on behalf of a Service Component, it must avoid causing a class load to occur from the Service Component's bundle. The SCR can ensure this by registering a ServiceFactory object with the framework. By registering a ServiceFactory object, the framework delays interface validation checking until the real service object is returned by the SCR after activating the Service Component.

5.3.2.2 Service Factory

The Service Component model supports a unique component instance for each bundle using the Service Component's service. This is in support of the ServiceFactory concept in the OSGi framework.

If the service element is specified with the attribute **servicefactory** set to "true", then a unqiue instance of the Service Component will be activated for each bundle which accesses the service. In this case the ComponentContext.getUsingBundle method will return the Bundle object of the bundle using the component instance.

If the service element is specified with the attribute **servicefactory** set to "false" or not specified, then all bundles which access the service will use the same instance of the Service Component. In this case the ComponentContext.getUsingBundle method will return null since multiple bundles may be using the component instance.

5.3.2.3 Configuration Admin

Service Components can receive properties from the Configuration Admin service. If a Service Component is activated and it's properties are updated in the Configuration Admin service, the SCR must deactivate the component and activate the component again using the new properties.

Configuration Admin supports both ManagedService and ManagedServiceFactory modes of use[3]. ManagedService provides a single set of properties while ManageServiceFactory provides multiple sets of properties per service.pid.

If a service.pid with the value of the component name is a factory pid, *i.e.* a ManagedServiceFactory, then the SCR will register the Service Component's service once for each Configuration associated with the factory pid. Each registered service will then be associated with a unique component instance and the pair will use the properties for the associated Configuration.

If a service.pid with the value of the component name is not a factory pid, *i.e.* a ManagedService, then the SCR will register the Service Component's service only once. The registered service will then be associated with a component instance and the pair will use the properties for the associated Configuration.

If ServiceFactory and ManagedServiceFactory are both used, then there will be multiple registered services for the Service Component, one per Configuration. Each registered service will then be associated with multiple component instances, one per bundle accessing the service.

5.3.3 Service Component Factory

A Service Component may be declared to use a component factory. This is done by specifiying the ***factory*** attribute on the ***component*** element. When the ***factory*** attribute is not specified, the SCR will create instances of

[3] SCR implementations will likely use RFC 103 [6] to avoid having to register MS and MSF services. In the case where no configuration for a pid exists, SCR would have to chose whether to register a MS or MSF service to be notified of the configuration being created. The use of ConfigurationListeners from RFC 103 will avoid this problem.

the Service Component as necessary. When the ***factory*** attribute is specifed, the SCR will register a component factory service for the Service Component on behalf of the Service Component.

A component factory service will be registered under the name “org.osgi.service.component.ComponentFactory” with the ***component.name*** service property set to the name of the component and the ***component.factory*** service property set to the value of the ***factory*** attribute.

If the component description contains a ***service*** element, then each created component instance will be registered as a service.

The use of ComponentFactory is incompatible with ManagedServiceFactory and ServiceFactory. This is because the SCR is not free to create component instances as necessary to support ManagedServiceFactory and ServiceFactory. It is an error in the component description to specify both ComponentFactory and ServiceFactory. It is also an error in the component description to specify ComponentFactory if a service.pid with the value of the component name is a factory pid, *i.e.* a ManagedServiceFactory.

5.3.4 Service Connections

Service connections are the connection between a Service Component’s service dependency specified by a ***reference*** element and the actual services (target services). How service connections are specified must:

1. Be simple for programmers to use in simple connection scenarios

2. Be configurable so that the connections can be established by deployers

3. Integrate with the existing OSGi configuration infrastructure

Connection properties allow configuration of connection between the provider of a service and the user of a service. This information is placed in the Service Component’s properties so that it can be configured “outside” of the bundle containing the Service Component. But reasonable values can be specified in **component** element so that the Service Component can be used without Configuration Admin.

5.3.4.1 Providing a Service

The following snippet contains a service element for a component.

```
<service>
   <provide interface="com.isv.service.customerinfo.CustomerInfo"/>
</service>
```

5.3.4.2 Referencing a Service

In order for a Service Component to use another service, we must be able to describe the other service to establish the connection. This description is done via a ***reference*** element. The first part of the connection description is specified by the name and interface attributes. They describe the reference name for the service used by the Service Component as well the Java interface the Service Component will use for the service. This information is coupled to the implementation of the Service Component since the Java interface and reference name will be used within the code.

The second part of the description is specified by target attribute which can be overridden by a **connection property** in the Service Component's properties. The connection property contain configurable information which is used to select the target service for the reference.

The name of the connection property is:

- ***<reference-name>*.target** – The value of this property is an OSGi filter string which is used to select the target service(s) from the OSGi service registry.

The following snippet contains a reference element for a service connection. In this example the value of the target attribute can be overridden by the **customerInfo.target** Service Component property.

```
<reference name="customerInfo"
   interface="com.isv.service.customerinfo.CustomerInfo"
   cardinality="1..1"
   policy="static"
   target="(component.name=com.isv.service.customerinfo.CustomerInfo)"
/>
```

5.3.4.3 Proxying Service Connections

The Service Component enabled OSGi framework may support proxied connections between the provider of a service and the consumer of the service. A proxied connection will allow the provider and consumer of the service to use different Java interfaces to represent the service. The Service Component providing the service only needs to provide all of the methods in the interface specified in the reference of the consuming Service Component. In particular, there is no requirement that the interface specified by the reference have the same name as the interface exposed by the Service Component, or that the interfaces be related in any way in an inheritance hierarchy. This is an important principle for loose coupling of components.

Proxied connections also provide other benefits such as avoiding the consumer of a service that has been deactivated from inadvertently maintaining on object reference to the service and thus preventing the deactivated Service Component from being garbage collected.

While a compliant SCR implementation does not have to support proxied connections, the Service Component specification must enable a compliant implementation to support it.

5.4 Service Component API

To support the Service Component model at runtime, some new API are defined. A ComponentContext interface is defined which can be used by a Service Component to interact with it execution context including locating services by reference name.

A component's implementation class may optional implement an activate method:

```
        protected void activate(ComponentContext context);
```

If a component implements this method, this method will be called when the component is activated to provide the component's ComponentContext object. If the activate method throws an exception, then the component will be considered to have failed to activate. The exception will be broadcast as a FrameworkEvent.ERROR. If the component is registered as a service and was being activated to produce an object to be used as a service object (i.e. a call to ComponentContext.locateService), then a ComponentException will be thrown by the locateService method. The component will then be considered to be deactivated and the unbind method(s) will be called for any bound services. Finally, if the component was registered as a service, the service will be unregistered.

A component's implementation class may optional implement a deactivate method:

```
protected void deactivate(ComponentContext context);
```

If a component implements this method, this method will be called when the component is deactivated. If the deactivate method throws an exception, then the exception will be broadcast as a FrameworkEvent.ERROR.

The activate and deactivate methods will be called by the SCR using reflection and must be at least protected accessible. These methods do not need to be public methods so that they do not appear as public methods on the component's provided service object. The SCR will look through the component's implementation class hierarchy for the first declaration of the method. If the method is declared protected or public, SCR will call the method.

5.4.1 Obtaining a Service

In the Serivce Component programming model there are two techniques by which an activated Service Component can obtain a referenced service.

5.4.1.1 Lookup Model

The first technique uses one of the ComponentContext methods: locateService or locateServices. This technique, which may be thought of as a "lookup" model, requires the Service Component to obtain it's ComponentContext object via the activate method described above. When the component needs a service object, it can call a locateService method to obtain the object. The returned object can be cast to the interface named in the reference. The component should not store the service object but should call a locateService method whenever the service object is required. For a service reference with a dynamic policy, this avoids holding a service object to a service which may have been unpublished after a locateService call.. This technique is very similar to the ServiceTracker model in which ServiceTracker.getService is called whenever a service object is needed. It is also similar to the JNDI Context.lookup model.

The locateService methods may throw ComponentException if the target serivce cannot be returned. This may occur if the target service is a component and that component throws an exception from its activate method. The locateServices methods may return a component instance which has been partially activated; *i.e.* has not completed its activate method. This is possible if there is a circular dependency between components. Therefore, during its activate method, a component should not call methods on a service object obtained from a locateService method since that service object may be a partially activated component instance.

```
CustomerInfo cInfo = (CustomerInfo)context.locateService("customerInfo");
```

If the cardinality of the reference is 0..n or 1..n, then more that one service may exist for the reference. The programmer can then use the locateServices method to access all the referenced services.

```
CustomerInfo[] cInfos = (CustomerInfo[])context.locateServices("customerInfo");
```

5.4.1.2 Event Model

The second technique is a bean-like model where the Service Component is called at specified methods to be notified of the availability and unavailablity of a service object. This technique, which may be thought of as an "event" model, requires the programmer to write a bind and unbind methods for the referenced service and specify those methods in the reference element of the component description. The bind method is called to provide the service object and the unbind method is called to withdraw the service object. The bind and unbind methods must take a single parameter. The service object must be assignable to the type of the parameter. With ManagedServiceFactory and ServiceFactory, there may be multiple instances of a component. The bind and unbind methods will be called on each component instance.

```
<reference name="customerInfo"
    interface="com.isv.service.customerinfo.CustomerInfo"
    bind=”bindCustomerInfo”
    unbind=”unbindCustomerInfo”
/>
```

```
protected void bindCustomerInfo(CustomerInfo cInfo) {
    this.cInfo = cInfo;
}
protected void unbindCustomerInfo(CustomerInfo cInfo) {
    this.cInfo = null;
}
```

If the cardinality of the reference is 0..n or 1..n, then more that one service may exist for the reference. In this case the bind method will be called multiple times, once for each service selected by the reference as it becomes available. Similarly, the unbind method will be called multiple times, once for each service selected by the reference as it becomes unavailable.

The bind method may be called before the Service Component is activated. Immediately prior to activating a Service Component, the bind method for each reference may be called if there is a service which matches the reference. A bind method may be passed a component instance which has been partially activated; *i.e.* has not had its activate method called. This is possible if there is a circular dependency between components. Therefore if a component’s bind method is called before that component’s activate method is called, the component’s bind method should not call methods on the object passed to the bind method since that object may be a partially activated component instance. If the service reference has a dynamic policy, the bind method for the reference may be called while the component is active. The unbind method may be called after the Service Component is deactivated. Immediately after deactivating a Service Component, the unbind method for each reference may be called if there is a service which matches the reference. If the service reference has a dynamic policy, the unbind method for the reference may be called while the component is active.

The bind and unbind methods will be called by the SCR using reflection and must be at least protected accessible. These methods do not need to be public methods so that they do not appear as public methods on the component's provided service object. The SCR will look through the component's implementation class hierarchy for the first declaration of the method. If the method is declared protected or public, SCR will call the method.

5.4.2 Accessing Component Properties

After a component is activated, it may access its properties by calling the ComponentContext.getProperties method. The returned Dictionary cannot be modified and contains all the Service Component properties.

```
Dictionary properties = context.getProperties();
```

5.4.3 Delayed Enablement

If a bundle needs to perform some lengthy initialization prior to making a Service Component's service available to other bundles, the bundle will need to declare that the Service Component is not auto enabled via the **autoenable** attribute of the **component** element. At the end of the initialization activity, the bundle must call the enableComponent method to enable the Service Components which were not auto enabled.

The argument to the enableComponent method must be either the name of a component declared by the bundle or null to indicate all components declared by the bundle.

The disableComponent method may be used if the bundle needs to disable components in preparation for stopping or any other reason.

Since the enableComponent method is part of the ComponentContext interface, a bundle must have an activated component in order to enable other components. This can be done by defining an autoenabled Service Component which does not register a service and does not reference any other services. Such a Service Component will be automatically activated when it's bundle is started.

5.5 Service Component Runtime

5.5.1 Relationship to OSGi Framework

The Service Component runtime (SCR) may be implemented as part of an OSGi framework or may be implemented as a bundle running on an OSGi framework along side Service Component bundles. In order to support implementing the SCR as a bundle, the framework must support such a bundle having access to other bundle's BundleContext objects. The SCR needs access to the BundleContext for the following reasons:

- To be able to register and get services on behalf of a Service Component bundle
- To interact with Configuration Admin on behalf of a Service Component bundle
- To provide a Service Component bundle with it's BundleContext when ComponentContext.getBundleContext is called

Since the BundleContext is a considered a private object to the bundle and would provide the capability for the receiver of the object to act as the bundle, there will be no specified way for the OSGi framework to provide a

BundleContext object to other bundles. This will be left as an implementation detail between a OSGi framework implementation and an SCR implementation.

Furthermore, the SCR will need at times to broadcast FrameworkEvent.ERROR. How the SCR interacts with the framework to broadcast these events is also an implementation detail between a OSGi framework implementation and an SCR implementation.

Note: There is no requirement that an SCR implemented as a bundle must run on any arbitrary OSGi framework implementation.

5.5.2 Starting and Stopping the SCR

When the SCR is implemented as a bundle running on top of the OSGi framework, any components activated by the SCR must be deactivated when the SCR bundle is stopped. When the SCR bundle is started, it must activate any components which must be automatically activated.

5.6 Service Component Description Schema

Here the is XML Schema for the component description. The master copy of the schema is in membercvs [5].

We use XML as the language to describe Service Components. The XML Namespace for Service Component descriptions is:

```
http://www.osgi.org/xmlns/scr/v1.0.0
```

The XML grammar specified by the schema can be easily parsed by very small XML parsers. For example, the JSR 172 (J2ME Web Services) XML parser is very small (~50KB) and namespace aware and can parse component descriptions.

5.6.1 XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.osgi.org/xmlns/scr/v1.0.0"
xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0">
    <annotation>
        <documentation xml:lang="en">
This is the XML Schema for component descriptions used by the Service Component Runtime (SCR).
Component description documents may be embedded in other XML documents. The SCR will
process all XML documents listed in the Service-Component manifest header of a bundle. The document
may contain a single component element or component elements embedded in a larger document.
Use of the namespace is optional if the document only contains a single component element.
Otherwise the namespace must be used.
        </documentation>
    </annotation>

    <element name="component" type="scr:Tcomponent"></element>

    <complexType name="Tcomponent">
        <sequence>
            <element name="implementation" type="scr:Timplementation" minOccurs="1"
maxOccurs="1"></element>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="property" type="scr:Tproperty"></element>
                <element name="properties" type="scr:Tproperties"></element>
            </choice>
            <element name="service" type="scr:Tservice" minOccurs="0" maxOccurs="1"></element>
            <element name="reference" type="scr:Treference" minOccurs="0"
maxOccurs="unbounded"></element>
```

```
        </sequence>
        <attribute name="autoenable" type="boolean" default="true" use="optional"></attribute>
        <attribute name="name" type="token" use="required"></attribute>
        <attribute name="factory" type="string" use="optional"></attribute>
    </complexType>

    <complexType name="Timplementation">
        <attribute name="class" type="token" use="required"></attribute>
    </complexType>

    <complexType name="Tproperty">
        <simpleContent>
            <extension base="string">
                <attribute name="name" type="string" use="required"></attribute>
                <attribute name="value" type="string" use="optional"></attribute>
                <attribute name="type" type="scr:TjavaTypes" use="optional" default="String"></attribute>
            </extension>
        </simpleContent>
    </complexType>

    <complexType name="Tproperties">
        <attribute name="entry" type="string" use="required"></attribute>
    </complexType>

    <complexType name="Tservice">
        <sequence>
            <element name="provide" type="scr:Tprovide" minOccurs="1" maxOccurs="unbounded"></element>
        </sequence>
        <attribute name="servicefactory" type="boolean" default="false" use="optional"></attribute>
    </complexType>

    <complexType name="Tprovide">
        <attribute name="interface" type="token" use="required"></attribute>
    </complexType>

    <complexType name="Treference">
        <attribute name="name" type="NMTOKEN" use="required"></attribute>
        <attribute name="interface" type="token" use="required"></attribute>
        <attribute name="cardinality" type="scr:Tcardinality" default="1..1" use="optional"></attribute>
        <attribute name="policy" type="scr:Tpolicy" use="optional" default="static"></attribute>
        <attribute name="target" type="string" use="optional"></attribute>
        <attribute name="bind" type="token" use="optional"></attribute>
        <attribute name="unbind" type="token" use="optional"></attribute>
    </complexType>

    <simpleType name="TjavaTypes">
        <restriction base="string">
            <enumeration value="String"></enumeration>
            <enumeration value="Long"></enumeration>
            <enumeration value="Double"></enumeration>
            <enumeration value="Float"></enumeration>
            <enumeration value="Integer"></enumeration>
            <enumeration value="Byte"></enumeration>
            <enumeration value="Char"></enumeration>
            <enumeration value="Boolean"></enumeration>
            <enumeration value="Short"></enumeration>
        </restriction>
    </simpleType>

    <simpleType name="Tcardinality">
        <restriction base="string">
            <enumeration value="0..1"></enumeration>
            <enumeration value="0..n"></enumeration>
            <enumeration value="1..1"></enumeration>
            <enumeration value="1..n"></enumeration>
        </restriction>
    </simpleType>

    <simpleType name="Tpolicy">
```

```
        <restriction base="string">
            <enumeration value="static"></enumeration>
            <enumeration value="dynamic"></enumeration>
        </restriction>
    </simpleType>

</schema>
```

5.7 Unresolved Issues

5.7.1 Introspection

Do we need an API for an external entity to inspect the components and their dependency state? This could be used to decide a service dependency is not resolved and attempt to install a bundle to resolve the dependency.

6 Javadoc

6.1 org.osgi.service.component

Interface ComponentContext

public interface **ComponentContext**

A ComponentContext object is used by a Service Component to interact with it execution context including locating services by reference name.

A component's implementation class may optional implement an activate method:

```
 protected void activate(ComponentContext context);
```

If a component implements this method, this method will be called when the component is activated to provide the component's ComponentContext object.

A component's implementation class may optional implement a deactivate method:

```
 protected void deactivate(ComponentContext context);
```

If a component implements this method, this method will be called when the component is deactivated.

The activate and deactivate methods will be called using reflection and must be at least protected accessible. These methods do not need to be public methods so that they do not appear as public methods on the component's provided service object. The methods will be located by looking through the component's implementation class hierarchy for the first declaration of the method. If the method is declared protected or public, the method will be called.

Method Summary

`void`	**`disableComponent`**`(java.lang.String name)` Disables the specified component name.
`void`	**`enableComponent`**`(java.lang.String name)` Enables the specified component name.
`org.osgi.framework.BundleContext`	**`getBundleContext`**`()` Returns the BundleContext of the bundle which contains this component.
`ComponentInstance`	**`getComponentInstance`**`()` Returns the ComponentInstance object for this component.
`java.util.Dictionary`	**`getProperties`**`()` Returns the component properties for this ComponentContext.
`ServiceReference`	**`getServiceReference`**`()` If this component is registered as a service using the `service` element, then this method returns the service reference of the service provided by this component.
`org.osgi.framework.Bundle`	**`getUsingBundle`**`()` If the component is registered as a service using the `servicefactory="true"` attribute, then this method returns the bundle using the service provided by this component.
`java.lang.Object`	**`locateService`**`(java.lang.String name)` Returns the service object for the specified service reference name.
`java.lang.Object[]`	**`locateServices`**`(java.lang.String name)` Returns the service objects for the specified service reference name.

Method Detail

6.1.1 getProperties

`public java.util.Dictionary` **`getProperties`**`()`

Returns the component properties for this ComponentContext.

Returns:

The properties for this ComponentContext. The properties are read only and cannot be modified.

6.1.2 locateService

`public java.lang.Object` **`locateService`**`(java.lang.String name)`

Returns the service object for the specified service reference name.

Parameters:

`name` - The name of a service reference as specified in a `reference` element in this component's description.

Returns:

A service object for the referenced service or `null` if the reference cardinality is `0..1` or `0..n` and no matching service is available.
Throws:
`ComponentException` - If the Service Component Runtime catches an exception while activating the target service.

6.1.3 locateServices

`public java.lang.Object[] `**`locateServices`**`(java.lang.String name)`

Returns the service objects for the specified service reference name.
Parameters:
`name` - The name of a service reference as specified in a `reference` element in this component's description.
Returns:
An array of service objects for the referenced service or `null` if the reference cardinality is `0..1` or `0..n` and no matching service is available.
Throws:
`ComponentException` - If the Service Component Runtime catches an exception while activating a target service.

6.1.4 getBundleContext

`public org.osgi.framework.BundleContext `**`getBundleContext`**`()`

Returns the BundleContext of the bundle which contains this component.
Returns:
The BundleContext of the bundle containing this component.

6.1.5 getUsingBundle

`public org.osgi.framework.Bundle `**`getUsingBundle`**`()`

If the component is registered as a service using the `servicefactory="true"` attribute, then this method returns the bundle using the service provided by this component.

This method will return `null` if the component is either:

- Not a service, then no bundle can be using it as a service.
- Is a service but did not specify the `servicefactory="true"` attribute, then all bundles will use this component.

Returns:
The bundle using this component as a service or `null`.

6.1.6 getComponentInstance

`public ComponentInstance `**`getComponentInstance`**`()`

Returns the ComponentInstance object for this component.
Returns:
The ComponentInstance object for this component.

6.1.7 enableComponent

`public void` **`enableComponent`**`(java.lang.String name)`

Enables the specified component name. The specified component name must be in the same bundle as this component.

Parameters:

`name` - The name of a component or `null` to indicate all components in the bundle.

6.1.8 disableComponent

`public void` **`disableComponent`**`(java.lang.String name)`

Disables the specified component name. The specified component name must be in the same bundle as this component.

Parameters:

`name` - The name of a component.

6.1.9 getServiceReference

`public ServiceReference` **`getServiceReference`**`()`

If this component is registered as a service using the `service` element, then this method returns the service reference of the service provided by this component.

This method will return `null` if this component is not registered as a service.

Returns:

The `ServiceReference` object for this component or `null` if this component is not registered as a service.

6.2 org.osgi.service.component
Interface ComponentFactory

---

public interface **ComponentFactory**

When a component is declared with the `factory` attribute on its `component` element, the Service Component Runtime will register a ComponentFactory service to allow instances of the component to be created rather than automatically create component instances as necessary.

Method Summary	
`ComponentInstance`	**`newInstance`**`(java.util.Dictionary properties)` Create a new instance of the component.

Method Detail

6.2.1 newInstance

`public ComponentInstance` **`newInstance`**`(java.util.Dictionary properties)`

Create a new instance of the component. Additional properties may be provided for the component instance.

Parameters:

`properties` - Additional properties for the component instance.
Returns:
A ComponentInstance object encapsulating the component instance. The returned component instance has been activated.

6.3 org.osgi.service.component
Interface ComponentInstance

public interface **ComponentInstance**

A ComponentInstance encapsulates an instance of a component. ComponentInstances are created whenever an instance of a component is created.

Method Summary

`void`	`dispose()` Dispose of this component instance.
`java.lang.Object`	`getInstance()` Returns the component instance.

Method Detail

6.3.1 dispose

`public void dispose()`

Dispose of this component instance. The instance will be deactivated. If the instance has already been deactivated, this method does nothing.

6.3.2 getInstance

`public java.lang.Object getInstance()`

Returns the component instance. The instance has been activated.
Returns:
The component instance or `null` if the instance has been deactivated.

6.4 org.osgi.service.component
Class ComponentException

```
java.lang.Object
  └─java.lang.Throwable
      └─java.lang.Exception
          └─java.lang.RuntimeException
              └─org.osgi.service.component.ComponentException
```

All Implemented Interfaces:
java.io.Serializable

public class **ComponentException**

extends java.lang.RuntimeException

Unchecked exception which may be thrown by the Service Component Runtime.

See Also:
    Serialized Form

Constructor Summary

`ComponentException(java.lang.String message)` Construct a new ComponentException with the specified message.
`ComponentException(java.lang.String message, java.lang.Throwable cause)` Construct a new ComponentException with the specified message and cause.
`ComponentException(java.lang.Throwable cause)` Construct a new ComponentException with the specified cause.

Method Summary

`java.lang.Throwable`	`getCause()` Returns the cause of this exception or `null` if no cause was specified when this exception was created.
`java.lang.Throwable`	`initCause(java.lang.Throwable cause)` The cause of this exception can only be set when constructed.

Methods inherited from class java.lang.Throwable
`fillInStackTrace, getLocalizedMessage, getMessage, getStackTrace, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

6.4.1 ComponentException

```
public ComponentException(java.lang.String message,
                          java.lang.Throwable cause)
```

Construct a new ComponentException with the specified message and cause.

Parameters:
    `message` - The message for the exception.
    `cause` - The cause of the exception. May be `null`.

6.4.2 ComponentException

`public **ComponentException**(java.lang.String message)`

Construct a new ComponentException with the specified message.

Parameters:

`message` - The message for the exception.

6.4.3 ComponentException

`public **ComponentException**(java.lang.Throwable cause)`

Construct a new ComponentException with the specified cause.

Parameters:

`cause` - The cause of the exception. May be `null`.

Method Detail

6.4.4 getCause

`public java.lang.Throwable **getCause**()`

Returns the cause of this exception or `null` if no cause was specified when this exception was created.

Returns:

The cause of this exception or `null` if no cause was specified.

6.4.5 initCause

`public java.lang.Throwable **initCause**(java.lang.Throwable cause)`

The cause of this exception can only be set when constructed.

Throws:

`java.lang.IllegalStateException` - This method will always throw an `IllegalStateException` since the cause of this exception can only be set when constructed.

6.5 org.osgi.service.component Interface ComponentConstants

---

public interface **ComponentConstants**

Defines standard names for Service Component constants.

Field Summary

`static java.lang.String`	**COMPONENT_FACTORY** A service registration property for a Service Component Factory.
`static java.lang.String`	**COMPONENT_ID** A component property for a Service Component that contains the generated id for the Service Component instance.
`static java.lang.String`	**COMPONENT_NAME** A service registration property for a Service Component.
`static java.lang.String`	**REFERENCE_TARGET_SUFFIX**

	A suffix for a service registration property for a reference target.
`static java.lang.String`	**`SERVICE_COMPONENT`** Manifest header (named "Service-Component") identifying the XML documents within the bundle containing the bundle's Service Component descriptions.

Field Detail

6.5.1 SERVICE_COMPONENT

`public static final java.lang.String` **`SERVICE_COMPONENT`**

Manifest header (named "Service-Component") identifying the XML documents within the bundle containing the bundle's Service Component descriptions.

The attribute value may be retrieved from the `Dictionary` object returned by the `Bundle.getHeaders` method.

See Also:
Constant Field Values

6.5.2 COMPONENT_NAME

`public static final java.lang.String` **`COMPONENT_NAME`**

A component property for a Service Component. It contains the name of the Service Component. The type of this property must be `String`.

See Also:
Constant Field Values

6.5.3 COMPONENT_ID

`public static final java.lang.String` **`COMPONENT_ID`**

A component property for a Service Component that contains the generated id for the Service Component instance. The type of this property must be `Long`.

The value of this property is assigned by the Service Component Runtime when a component instance is created. The Service Component Runtime assigns a unique value that is larger than all previously assigned values since the Service Component Runtime was started. These values are NOT persistent across restarts of the Service Component Runtime.

See Also:
Constant Field Values

6.5.4 COMPONENT_FACTORY

`public static final java.lang.String` **`COMPONENT_FACTORY`**

A service registration property for a Service Component Factory. It contains the value of the `factory` attribute. The type of this property must be `String`.

See Also:
Constant Field Values

6.5.5 REFERENCE_TARGET_SUFFIX

`public static final java.lang.String` **`REFERENCE_TARGET_SUFFIX`**

A suffix for a component property for a reference target. It contains the filter to select the target services for a reference. The type of this property must be `String`.

See Also:

Constant Field Values

6.5.6 Constant Field Values

org.osgi.service.component.ComponentConstants		
`public static final java.lang.String`	`COMPONENT_FACTORY`	`"component.factory"`
`public static final java.lang.String`	`COMPONENT_ID`	`"component.id"`
`public static final java.lang.String`	`COMPONENT_NAME`	`"component.name"`
`public static final java.lang.String`	`REFERENCE_TARGET_SUFFIX`	`".target"`
`public static final java.lang.String`	`SERVICE_COMPONENT`	`"Service-Component"`

7 Considered Alternatives

I attempted to decouple the component declarations from the service declarations. Unfortunately this proved unworkable. The main problem is that components require services. They may also provide services. When circular dependencies exists between components, for example two components each require the service provided by the other component, the circular dependency cannot be resolved by looking at a component's requirements in isolation from the service it may provide.

I had each component in a single XML file (in the META-INF/components dir) and then had properties files whose name was the same as the XML file. This was rejected for several reasons. First there was a desire for the component descriptions to be embedded in a larger XML document (e.g. the Meglet application descriptor) or for multiple component descriptions per file. Second we needed to support multi-valued properties (i.e. a property whose value is an array). So using property files to contain the properties and their localization would not support multi-valued properties.

Localization support was removed since it was felt that localization should be done closer to the actual presentation of the information to a user. Here is the former localization text. The localization attribute of the component element was also removed.

> Property values of type String may be localized. Property values that start with “%” are localized. The remainder of the property value is a localization key into a property localization resource. To use a % as the leading character in a property value and avoid localization, use “%%” which will be replaced by a single “%”.
>
> Property localization resources belong to a family that shares a common base name with additional suffixes which are derived from a locale name. The SCR searches for localization resources by appending suffixes to the base name according to the default locale as returned by java.util.Locale.getDefault. When locating a localization key within the family of localization resources, the resources must be examined from most specific suffix to least specific suffix for the given locale. This allows localization resources for more specific locales to contain different localizations or to not contain a localization so that it may be provided by the localization resource for a less specific locale. The order for locating localization keys in a family of localization resources is the order as specified by the java.util.ResourceBundle.getBundle method.
>
> The base name for the localization resources is specified by the ***localization*** attribute of the ***component*** element. If this attribute is not specified, then the value of the `Bundle-Localization` manifest header is used. If the manifest header is not specified, then the base name is “META-INF/bundle”.
>
> When searching for a localization resource, the SCR must first look in the bundle and then look in the currently attached fragment bundles. If no localization resource is found, the value of the property will be the value specified in the ***property*** element without the leading “%”. If an appropriate localization resource is found but no value is found for the localization key, the value of the property will be the value specified in the ***property*** element without the leading “%”.

---

8 Security Considerations

Declarative services are built upon the existing OSGi service infrastructure. This means that normal rules for ServicePermissions apply regarding the ability to publish, find or bind to services.

If a bundle does not have ServicePermission with the “register” action for an interface name specified by a provide element, then the Service Component will never be activated and other Service Components will not recognize it as able to resolve a service dependency.

If a bundle does not have ServicePermission with the “get” action for an interface name specified by a reference element and the reference has a static policy, then the Service Component will never be activated and other Service Components will not recognize it as able to resolve a service dependency.

If a bundle does not have ServicePermission with the “get” action for an interface name specified by a reference element and the reference has a dynamic policy, then the Service Component will can be activated but will not be able to access the referenced service.

Registering a Service Component service will not be done by the Service Component bundle calling the framework, so a normal stack check will not work. The same is true for getting services. Since the SCR is registering and getting services on behalf of a Service Component bundle, the SCR must use Bundle.hasPermission to validate that a Service Component bundle has the necessary permission to register or get a service.

9 Document Support

9.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. NMToken definition, http://www.w3.org/TR/1998/REC-xml-19980210#NT-Nmtoken

[4]. Automating Service Dependency Management in a Service-Oriented Component Model, http://www-adele.imag.fr/~cervante/papers/CBSE2003.pdf, H.Cervantes and R. S. Hall

[5]. Service Component Description Schema, http://membercvs.osgi.org/cgi-bin/cvsweb.cgi/~checkout~/xmlns/scr/scr.xsd?cvsroot=/cvshome/build

[6]. RFC 103 Configuration Listener, http://membercvs.osgi.org/rfcs/rfc0103/rfc-0103-ConfigurationListener.doc

9.2 Author’s Address

Name	BJ Hargrave
Company	IBM
Address	11501 Burnet Rd, Austin, TX 78758 USA
Voice	+1 512 838 8838
e-mail	hargrave@us.ibm.com

9.3 Acronyms and Abbreviations

[1]. SCR – Service Component Runtime

9.4 End of Document