RFC 156 - Blueprint Service Configuration Admin Support

Draft

22 Pages

Abstract

This RFC extends the Blueprint service to support Configuration Admin.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 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 8.1.

```
Source code is shown in this typeface.
```

0.6 Revision History

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

Revision	Date	Comments
0	12/18/09	First draft. Glyn Normington, VMware, gnormington@vmware.com
0.1	08/01/10	Convert David Bosschaert's requirements into problem descriptions for discussion. Glyn Normington
0.2	01/18/10	Rework following review at Southampton F2F. Start on the technical solution section as input to the prototype. Glyn Normington
0.2.1	01/20/10	Explain the technical solution. Glyn Normington
0.2.2	01/28/10	Revise the technical solution and explain that it is simply the starting point for a prototype. Move property placeholder section to considered alternatives. Glyn Normington
0.3	03/09/10	Enumerate the requirements. Glyn Normington
0.3.1	03/15/10	Note RFC 144 in section 3.2 Glyn Normington
0.4	06/25/10	Update technical solution to reflect London F2F discussion except that updating by deleting and recreating a component is relegated to considered alternatives. Glyn Normington
0.4.1	06/29/10	Clarify technical solution. Glyn Normington
0.5	11/22/10	Revise technical solution based on meeting with IBM and discussion at Walldorf F2F. Acknowledge the “lazy config” requirement from OSGi bug 1793. Glyn Normington
0.6	05/26/11	Incorporate bug 1738. Mark Nuttall

1 Introduction

Blueprint components need to be configured using Configuration Admin. Spring DM, which inspired the Blueprint service, has some support for Config Admin which will be used to inform this design.

This RFC relies on the support for namespaces being added to the Blueprint Service by RFC 155 – see reference [3].

The work to standardize Config Admin support began in RFC 124 - see reference [4]. However, the requirements and technical solution did not stabilize in time for R4.2.

This RFC captures the requirements for Config Admin support and proposes a solution.

Note that this RFC will be developed in parallel with a prototype implementation based on the Blueprint service reference implementation.

2 Application Domain

The Blueprint service enables an OSGi bundle to define Java objects known as *components* using a XML configuration file stored in the bundle. Components may be published as services in the service registry and may consume services from the service registry.

The Config Admin service provides a standard way of supplying configuration to OSGi bundles. Configuration is grouped into key->value dictionaries identified by Persistent Id (PID).

The Blueprint and Config Admin services are described in the R4.2 Service Compendium - see reference [5].

2.1 Dictionaries

The Configuration Admin service allows the creation, dissemination, and persistence of configurations for objects (which may be bundles, devices, or other resources) in an OSGi framework environment.

Configurations are represented as Config Admin *dictionaries* mapping *keys* to *values*.

We use the term `key' rather than property name to avoid ambiguity with `component properties' which are basic to the Blueprint service.

2.2 Configuration Admin

Config Admin is a collection of dictionaries indexed by *persistent identity* or *PID*.

Updating a configuration adds a new PID and dictionary or updates the dictionary associated with an existing PID.

Deleting a configuration removes a PID and its dictionary.

3 Problem Description

Add concrete use cases that are actually required. Avoid the temptation to make up use cases.

3.1 Updates

Although the full set of use cases for update will require a callback, the simple case should be possible without a callback. The basic problem is to define a mapping (per PID) from key to property. This mapping can be provided by a callback or by a declaration.

Spring DM supports *bean managed* and *container managed* update strategies (which are no longer mutually exclusive in Spring DM 2.0).

The bean managed update strategy is a bulk update callback to a bean method. This provides flexibility of locking and could also support the incremental update defined by RFC 150 – see reference [6].
The container managed update strategy is less flexible. All the properties in the updated Config Admin dictionary are set in turn inside a single synchronized block (on the bean object).

This approach to synchronization is questionable as it exposes the object's concurrency control mechanism to callers. It can also lead to deadlocks as there is no way to drop the monitor if it is necessary to make an “alien” call to another class.

3.2 Relationship of Components to PIDs

There needs to be an arbitrary relationship between PIDs and configured components. A single component must be able to consume properties from zero or more PIDs. A single PID must be capable of configuring zero or more components.

Config Admin has a basic restriction that distinct bundles may not access the same PID (and some other restrictions on Managed Factories which may not be relevant) but, apart from that, Spring DM satisfies these requirements. RFC 144 will lift this restriction.

3.3 Config Admin Dictionary Key to Bean Property Mappings

It should be possible to map Config Admin keys to property names. For example, a key 'com.acme.size' could be mapped to a property called 'size'.

Spring DM supports this via the bean-managed update strategy. The callback method can invoke whatever property setters it likes.

3.4 Defaults

There needs to be a way of specifying default values for keys which are not supplied in a PID's Config Admin dictionary.

Spring DM supports specifying default property value in a cm-properties element which may be used as input to a property placeholder configurer. The following example demonstrates this.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:osgi="http://www.springframework.org/schema/osgi"
            xmlns:osgix="http://www.springframework.org/schema/osgi-compendium"
            xmlns:context="http://www.springframework.org/schema/context"
            xsi:schemaLocation="[omitted]">

            <!--
              Obtain properties from Config Admin dictionary with PID=caprops. The
              dictionary has the following properties: day=7, snow=true,
              dayOfWeek=Thursday, year=2010. Note: properties are injected when the
              bean is created but are not updated subsequently if the Config Admin
              dictionary changes.
            -->
            <osgix:cm-properties id="caprops.cfg" persistent-id="caprops">

                        <!-- Provide some default property values -->
                        <prop key="month">January</prop>
                        <prop key="year">1970</prop>

            </osgix:cm-properties>

            <!--
              Property placeholder configurer referencing properties from Config
              Admin.
            -->
            <context:property-placeholder properties-ref="caprops.cfg" />

            <!-- Bean referencing properties using placeholders. -->
            <bean id="aBean" class="rfc0156.ca0.Bean">

                        <!-- int day comes from Config Admin  -->
                        <property name="day" value="${day}"></property>

                        <!-- String month comes from default -->
                        <property name="month" value="${month}"></property>

                        <!-- int year comes from Config Admin overriding default -->
                        <property name="year" value="${year}"></property>

                        <!--  boolean snow comes from Config Admin -->
                        <property name="snow" value="${snow}"/>

            </bean>

</beans>
```

Spring also supports specifying property values for a bean which can be used in conjunction with Spring DM to provide default values for properties which are omitted from a Config Admin dictionary. The following example demonstrates this. The <bean> and <property> elements are supported by the Blueprint service.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:osgi="http://www.springframework.org/schema/osgi"
            xmlns:osgix="http://www.springframework.org/schema/osgi-compendium"
            xmlns:context="http://www.springframework.org/schema/context"
```

```
        xsi:schemaLocation="[omitted]">

        <!-- Bean referencing managed properties using Config Admin. -->
        <bean id="aBean" class="rfc0156.dynamic.DynamicBean">
                <osgix:managed-properties persistent-id="caprops-dyn"
                        update-strategy="container-managed" />

                <!-- Provide some default property values -->
                <property name="month" value="January" />
                <property name="year" value="1970" />

        </bean>

</beans>
```

3.5 Property Types

It should be possible to support all the data types supported by Configuration Admin. Section 104.4.2 of the Config Admin spec limits values to the following types:

```
type ::= simple | vector | arrays

simple ::= String | Integer | Long | Float | Double | Byte |
                Short | Character | Boolean

primitive ::= long | int | short | char | byte | double |
                float | boolean

arrays ::= primitive '[]' | simple '[]'

vector ::= Vector of simple
```

By default, Spring DM supports certain types other than String such as int and boolean. All data types supported by both Blueprint and Config Admin MUST be supported.

The filter syntax in section 3.2 of the core specification may help here.

4 Requirements

REQ-MULTI-PID-COMP: It MUST be possible to configure a given component using zero or more PIDs.

REQ-MULTI-COMP-PID: It MUST be possible to configure zero or more components using a given PID.

REQ-MAP-KEYS-TO-PROPS: It SHOULD be possible to specify how Config Admin dictionary keys are mapped to component property names.

REQ-DEFAULT-PROPS: It MUST be possible to specify default values for keys which are not supplied in a PID's Config Admin dictionary.

REQ-PROP-TYPES: It SHOULD be possible to support properties of all the data types supported by Config Admin.

REQ-UPDATE: When the Config Admin dictionary for a given PID is updated, any components configured using that PID MUST be reconfigured with the new dictionary.

REQ-LAZY-CONFIG: It SHOULD be possible to wait for configuration to become available. See OSGi bug 1793 for background.

5 Technical Solution

The technical solution will be developed in parallel with a prototype. In order for the prototyping to begin, the following starting point is proposed.

We need to describe how dictionary keys and component property names are related, how to specify defaults when a dictionary does not provide all the property values a component requires, how to reference multiple dictionaries from a component, and what to do when any of the referenced dictionaries are updated or deleted.

The following sections examine the solutions to these requirements. The solution uses a custom namespace indicated by the “cm:” namespace qualifier.

5.1 Specifying a Config Admin PID

The requirement **REQ-MULTI-COMP-PID** states:

> It MUST be possible to configure zero or more components using a given PID.

A PID may be specified at any of three levels: the whole blueprint, a particular component, a particular property of a particular component.

A PID for a whole blueprint is specified on the blueprint element and applies to all the blueprint XML files in a given bundle, for example:

```
<blueprint cm:pid=”some.pid” ...>
    ...
</blueprint>
```

If conflicting PIDs are specified on distinct blueprint elements in the same bundle, this is an error and it is not defined which PID(s) are used for the whole blueprint. *The prototype may tighten up the rules in this case.*

A PID for a component is specified on the bean element and overrides any PID specified for the whole blueprint in the sense that a lookup using a given Config Admin dictionary key tries the component PID first and only if the key is not found tries the blueprint PID. The following is a simple example of a PID specified for a component:

```
<bean id="bean" cm:pid="other.pid" ...>
    ...
</bean>
```

A PID for a particular property of a particular component is specified on the property element and overrides any PIDs specified for the component and for the whole blueprint in the sense that a lookup using a given Config Admin dictionary key tries the property PID first, and if the key is not found, tries any component PID next, and only if the key is again not found, tries any blueprint PID. The following is a simple example of a PID specified for a property:

```
<bean id="bean" cm:pid="other.pid" ...>
    <property name="myProp" cm:pid="yet.another.pid" cm:key="org.foo.prop"/>
</bean>
```

5.2 Specifying Multiple Config Admin PIDs

The requirement **REQ-MULTI-PID-COMP** in RFC 156 states:

> It MUST be possible to configure a given component using zero or more PIDs.

We allow a list of one or more PIDs to be specified by the cm:pid attribute of a property element, a component element, or a blueprint element. PIDs later in the list take precedence over PIDs earlier in the list – essentially the list is searched backwards.

For example, the following component element specifies two PIDs with pid2 taking precedence over pid1.

```
<bean ... cm:pid="pid1 pid2" ...>
```

5.3 Mapping Property Names to Keys

The requirement REQ-MAP-KEYS-TO-PROPS states:

> It SHOULD be possible to specify how Config Admin dictionary keys are mapped to component property names.

We overload the property element to specify the Config Admin dictionary key for a given component property.

Note there is no requirement to be able to map multiple dictionary keys to the same property name as that would instantly create confusion when both keys exist but are mapped to distinct values.

For example, the following property map a Config Admin dictionary key to a corresponding property:

```
<property name="size" … cm:key="com.acme.size"/>
```

Note that this solution does not provide a way to avoid specifying both the property and the key when the two are equal. This may not be critical since the two are specified as attributes of the same element and so are less likely to get out of step than if they were in separate elements. *However, the prototype should explore the possibility of omitting the name attribute when the key is specified and the two are equal. This would require deferring part of the validation of the blueprint until after namespace processing has finished.*

5.4 Default Configuration

Requirement **REQ-DEFAULT-PROPS** states**:**

> It MUST be possible to specify default values for keys which are not supplied in a PID's Config Admin dictionary.

Configuration defaults are specified using the value attribute of the property element. Such a default value applies if and only if the Config Admin dictionary key is not found in any PID specified for the property, any PID specified for the component, and any PID specified for the whole blueprint.

For example, the following component specifies default values for two properties.

```
<bean id="def" osgix=”some.pid”>
    <property name="host" value="localhost" cm:key=”host” />
    <property name="port" value="8080" cm:key=”port” />
</bean>
```

5.5 Update

The requirement **REQ-UPDATE** states:

> When the Config Admin dictionary for a given PID is updated, any components configured using that PID MUST be reconfigured with the new dictionary.

When the contents of Config Admin changes a component that was configured using Config Admin may need to be updated. Changes in Config Admin include new PIDs being created, dictionaries associated with existing PIDs being updated, PIDs being deleted, or arbitrary combinations of creations, updates and deletions.

If a change occurs in Config Admin which means that a component would have been configured differently than it was originally, then the component properties may need to be updated. Two strategies are supported: restarting the whole blueprint and setting updated properties. The default strategy is to restart the whole blueprint. The strategy may be specified on a blueprint element. If conflicting strategies are specified, the blueprint restart strategy wins.

5.5.1 Update by Restarting the Blueprint

This is the simplest strategy for the component developer and is the default strategy. A change to any of the Config Admin PIDs which are referenced by the blueprint XML files of a given bundle causes the whole blueprint to be destroyed and recreated if and only if this is specified on at least one of the blueprint elements of the bundle:

```
<blueprint ... cm:updateStrategy=”restart” >
```

Again a batch of changes, as enabled by RFC 150, should be applied with one restart. This may be implemented by remembering the versions associated with the PIDs applied to a blueprint.

The feasibility of restarting the whole blueprint, which is likely to be required in other circumstances such as when a custom namespace goes away, is to be validated by the prototype.

5.5.2 Update by Setting Properties

When an update occurs, property setters are called to apply updates for all the PIDs specified by a blueprint. This ensures that a batch of updates, as enabled by RFC 150, is applied in one step since update notifications are sent after all the Config Admin PIDs in the batch are updated.

This strategy requires that the components of the blueprint, or at least those which have properties provided by Config Admin, are thread safe. Apart from protecting the internal state of these components, thread safety ensures that updates made by a given thread are published safely and are visible to other threads.

The calling of property setters is optionally bracketed by calls to user-defined begin update and end update methods. If either or both of these methods are not specified, then the corresponding methods are not called.

Begin update and end update methods must be parameterless with a void return type. If either of the named methods does not exist or has the wrong signature, it is ignored (although the implementation may choose to log diagnostics).

The configuration changes for a component bracketed by begin update and end update methods correspond to either a single update of a Config Admin PID or a batch of updates as defined by RFC 150.

A component may specify begin and end update methods in XML syntax using *beginUpdate* and *endUpdate* attributes which name methods of the component's class. These attributes are optional.

For example, the following component element specifies both begin update and end update methods.

```
<component ... cm:beginUpdate="beginMethod" cm:endUpdate="endMethod" >
```

This strategy may be specified on a blueprint element as follows:

```
<blueprint ... cm:updateStrategy=”setProperties” >
```

5.6 Non-String Property Types

The requirement **REQ-PROP-TYPES** states:

> It SHOULD be possible to support properties of all the data types supported by Config Admin.

The solution, likely to be inspired by Spring DM, will be specified after the initial prototype is complete.

5.7 Lazy Configuration

The requirement **REQ-LAZY-CONFIG** states:

> *It SHOULD be possible to wait for configuration to become available.*

This requirement introduces the need for a

```
<cm:availability ::= mandatory | optional />
```

attribute at the blueprint, bean and property level. Timeouts from mandatory elements not being satisfied would be handled in the same way as service references. There will also need to be a

```
<cm::default-availability ::= optional | mandatory />
```

element at the blueprint level, with a default value of 'optional'.

~~*The solution, likely to be inspired by Spring DM, will be specified after the initial prototype is complete.*~~

6 Considered Alternatives

6.1 PropertyMapping based solution

The following sections show an earlier iteration of the technical solution.

6.1.1 Mapping Property Names to Keys

The requirement REQ-MAP-KEYS-TO-PROPS states:

> It SHOULD be possible to specify how Config Admin dictionary keys are mapped to component property names.

We introduce the concept of a *property mapping* from property names to dictionary keys.

A property mapping is the opposite way round to the requirement so that it is a function. There is no requirement to be able to map, for example, two dictionary keys to the same property name as that would instantly create confusion when a single dictionary contained both keys but mapped to distinct values.

There is a fixed, default way of mapping property names to dictionary keys which applies to any property names not covered by a property mapping: any property name not covered by a property mapping is mapped to the dictionary key equal to the property name.

A property mapping is specified in XML as part of a *configSelector* element (described in section Error: Reference source not found) enclosing a series of *propertyMapping* elements, each of which maps a single property name, identified by a *property* attribute, to a key identified by a *key* attribute.

If a given configSelector encloses more than one propertyMapping element for a given property name, the first propertyMapping encountered is honoured and the others are ignored.

For example, the following property mapping maps two Config Admin dictionary keys to corresponding properties:

```
<osgix:propertyMapping property="size" key="com.acme.size"/>
<osgix:propertyMapping property="color" key="com.acme.color"/>
```

6.1.2 Configuration Selection

The requirement **REQ-MULTI-COMP-PID** states:

> It MUST be possible to configure zero or more components using a given PID.

A *configuration selector* identifies a PID and optionally specifies a property mapping. A configuration selector has an identifier so that it may be referred to by zero or more components.

A dictionary identified by the selector's PID is mapped using the selector's property mapping to produce some component properties (destined for injection into components).

If the selector's PID does not exist in Config Admin, then no component properties are produced.

A configuration selector is specified in XML using a *configSelector* element identified by an *id* attribute, specifying a PID by a *persistent-id* attribute, and optionally enclosing one or more propertyMapping elements.

For example, the following configuration selector specifies its identity, a PID, and a property mapping.

```
<osgix:configSelector id="cs1" persistent-id="com.acme.aPid">
    <osgix:propertyMapping property="p1" key="key1" />
    <osgix:propertyMapping property="p2" key="key2" />
</osgix:configSelector>
```

6.1.3 Default Configuration

Requirement **REQ-DEFAULT-PROPS** states**:**

> It MUST be possible to specify default values for keys which are not supplied in a PID's Config Admin dictionary.

Configuration defaults are specified as a mapping from property name to default value.

Configuration defaults are specified in XML using a series of *property* elements, each of which specifies a property name identified by a *name* attribute and a default value identified by a *value* attribute.

If more than one property element specifies a given property name, the first property element encountered is honoured and the others are ignored.

Values are specified as strings and for non-String properties, a type conversion is attempted at runtime. If the type conversion fails, the default is not applied (although implementations may log diagnostics).

For example, the following default configuration specifies default values for two properties.

```
<osgix:defaults id="def">
    <osgix:property name="p1" value="default1" />
    <osgix:property name="p2" value="default2" />
</osgix:defaults>
```

6.1.4 Component Configuration

The requirement **REQ-MULTI-PID-COMP** in RFC 156 states:

> It MUST be possible to configure a given component using zero or more PIDs.

We allow a component to specify a sequence of configuration selectors which will be used to configure the component. The configuration selectors are applied *in order* so that later configuration selectors in the sequence take precedence over earlier configuration selectors.

In addition, we allow a component optionally to specify a default configuration which provides values for properties not determined by any sequence of configuration selectors.

Components may specify a sequence of configuration selectors in XML using a *configSelectorRefs* attribute which consists of a comma separated sequence of one or more configuration selector element identifiers. Components may specify a default configuration using a *defaultsRef* attribute which consists of a single defaults element identifier.

For example, the following component element specifies a sequence of two configuration selectors and a default configuration.

```
<component ... osgix:configSelectorRefs="cs1,cs2" osgix:defaultsRef="def" >
```

6.2 Property Placeholders

The popular ${variable} syntax shown in the example below is a Spring property placeholder. Supporting this syntax would add considerable complexity, such as additional lifecycle states, into the Blueprint service specification and implementations.

Specifying individual properties using placeholders is usable for simple use cases with few properties, but it may not be ideal for supporting Config Admin updates.

The point here is to make the syntax as readable and intuitive as the property placeholder syntax.

The following example shows the use of property placeholders:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:osgi="http://www.springframework.org/schema/osgi"
            xmlns:osgix="http://www.springframework.org/schema/osgi-compendium"
            xmlns:context="http://www.springframework.org/schema/context"
            xsi:schemaLocation="[omitted]">

            <!--
              Obtain properties from Config Admin dictionary with PID=caprops.
            -->
            <osgix:cm-properties id="caprops.cfg" persistent-id="caprops" />


            <!--
              Property placeholder configurer referencing properties from Config
              Admin.
            -->
            <context:property-placeholder properties-ref="caprops.cfg" />

            <!-- Bean referencing properties using placeholders. -->
            <bean id="aBean" class="rfc0156.ca0.Bean">

                        <!-- int day comes from Config Admin  -->
                        <property name="day" value="${day}"></property>

            </bean>

</beans>
```

6.3 RFC 124 Configuration Administration Service Support

RFC 124 proposed the following technical solution, closely modeled on the Config Admin support in Spring DM.

The `osgix` namespace defines configuration elements and attributes supporting the OSGi Compendium Services. Currently the only service with dedicated support in this namespace is the Configuration Admin service.

6.3.1 Property Placeholder Support

Component property values may be sourced from the OSGi Configuration Administration service. This support is enabled via the `property-placeholder` element. The property placeholder element provides for replacement of delimited string values (placeholders) in component property expressions with values sourced from the configuration administration service. The required `persistent-id` attribute specifies the persistent identifier to be used as the key for the configuration dictionary. The default delimiter for placeholder strings is "${...}". Delimited strings can then be used for any property value of any component, and will be replaced will the configuration administration value with the given key.

Given the declarations:

```
<osgix:property-placeholder persistent-id="com.xyz.myapp"/>

<component id="someComponent" class="AClass">
  <property name="timeout" value="${timeout}"/>
</component>
```

Then the timeout property of someComponent will be set using the value of the timeout entry in the configuration dictionary registered under the com.xyz.myapp persistent id.

The placeholder strings are evaluated at the time that the component is instantiated. The evaluation results in a new string which is then interpreted as if it were the originally declared value. Changes to the properties made via Configuration Admin subsequent to the creation of the component do not result in re-injection of property values. See the `managed-properties` and `managed-service-factory` elements if you require this level of integration. The `placeholder-prefix` and `placeholder-suffix` attributes can be used to change the delimiter strings used for placeholder values. It is a configuration error to define multiple property-placeholder elements using the same prefix and suffix, and a ComponentDefinitionException will be thrown during context creation if such conflicting declarations are found.

It is possible to specify a default set of property values to be used in the event that the configuration dictionary does not contain an entry for a given key. The `defaults-ref` attribute can be used to refer to a named component of Properties or Map type. Instead of referring to an external component, the `default-properties` nested element may be used to define an inline set of properties.

```
<osgix:property-placeholder persistent-id="com.xyz.myapp">
  <osgix:default-properties>
     <property name="productCategory" value="E792"/>
     <property name="businessUnit" value="811"/>
  </osgix:default-properties>
</osgix:property-placeholder>
```

Property placeholder declarations have module context scope, and apply to any matching placeholder string regardless of the particular configuration file of the module the property placeholder and placeholder string declarations happen to be in. If a property referenced via a placeholder definition is not defined in the configuration dictionary, and no default value has been specified, then a runtime ComponentDefinitionException will be thrown during module context creation.

The `persistent-id` attribute must refer to the persistent-id of an OSGi ManagedService, it is a configuration error to specify a factory persistent id referring to a ManagedServiceFactory.

Placeholder expressions can be used in any attribute value, as the whole or part of the value text.

6.3.2 Publishing Configuration Admin properties with exported services

Using the property-placeholder support it is easy to publish any named configuration-admin property as a property of a service exported to the service registry. For example:

```
<service interface="MyInterface" ref="MyService">
  <service-properties>
    <entry key="akey" value="${property.placeholder.key}"/>
```

```
  </service-properties>
</service>
```

To publish all of the public properties registered under a given persistent-id as properties of an exported service, without having to explicitly list all of those properties up-front, use the nested `cm-properties` element.

```
<service interface="org.osgi.service.cm.ManagedService" ref="MyManagedService">
  <service-properties>
    <osgix:cm-properties persistent-id="pid"/>
  </service-properties>
</service>
```

Only public properties registered under the pid (properties with a key that does not start with ".") will be published. To have the advertised service properties updated when the configuration stored under the given persistent id is update, specify the optional `update="true"` attribute value.

6.3.3 Managed Properties

The `managed-properties` element can be nested inside a component declaration in order to configure component properties based on the configuration information stored under a given persistent id. It has one mandatory attribute, `persistent-id`.

An example usage of managed properties follows:

```
<component id=”myComponent” class=”AClass”>
  <osgix:managed-properties persistent-id="com.xyz.messageservice"/>
  <!-- other component declarations as needed …-->
</component>
```

For each key in the dictionary stored by configuration admin under the given persistent id, if the component type has a property with a matching name (following JavaBeans conventions), then that component property will be dependency injected with the value stored in configuration admin under the key.

If the definition of AClass from the example above is as follows:

```
public class AClass {
  private int amount;
  public void setAmount(int amount) { this.amount = amount; }
  public int getAmount() { return this.amount; }
}
```

and the configuration dictionary stored under the pid com.xyz.messageservice contains an entry “amount”->”200”, then the setAmount method will be invoked on the component instance during configuration, passing in the value 200.

If a property value is defined both in the configuration dictionary stored in the Configuration Admin service, and in a property element declaration nested in the component element, then the value from Configuration Admin takes precedence. Property values specified via property elements can therefore be treated as default values to be used if none is available through Configuration Admin.

The configuration data stored in Configuration Admin may be updated after the component has been created. By default, any updates post-creation will be ignored. To receive configuration updates, the `update-strategy` attribute can be used with a value of either `component-managed` or `container-managed`.

The default value of the optional `update-strategy` attribute is `none`. If an update strategy of `component-managed` is specified then the `update-method` attribute must also be used to specify the name of a method defined on the component class that will be invoked if the configuration for the component is updated. The update method must have one of the following signatures:

```
public void anyMethodName(Map properties)

public void anyMethodName(Map<String,?> properties); // for Java 5
```

When an update strategy of `container-managed` is specified then the container will re-inject component properties by name based on the new properties received in the update. For container-managed updates, the component class must provide setter methods for the component properties that it wishes to have updated. For each property in the updated configuration dictionary where the component class has a matching setter method, the setter method will be called with the new value.

6.3.4 Managed Service Factories

The Configuration Admin service supports a notion of a Managed Service Factory (see section 104.6 in the Compendium Specification). A managed service factory is identified by a factory pid, Configuration objects can be associated with the factory. Configuration objects associated with the factory can be added or removed at any point.

The `managed-service-factory` element defines a managed set of services. For each configuration object associated with the factory pid of the managed service factory, an anonymous component instance is created and registered as a service. The lifecycle of these component instances is tied to the lifecycle of the associated configuration objects. If a new configuration object is associated with the factory pid, a new component instance is created and registered as a service. If a configuration object is deleted or disassociated from the factory pid then the corresponding component instance is destroyed.

The attributes of the `managed-service-factory` element are:

- `id` (required)
- `factory-pid` (required) – this specifies the persistent id of the managed service factory in the Configuration Admin service
- `interface`, `auto-export`, and `ranking`, all with the same semantics as the attributes with the corresponding names defined on the <service> element. These attributes apply to each service registered on behalf of the managed service factory.

Optionally nested inside the managed-service-factory element are the `interfaces`, `service-properties`, and `registration-listener` elements, with the same syntax and semantics as when used nested inside of a service element.

A single nested `managed-component` element is required inside the `managed-service-factory`. The managed component declaration defines the component template for the component instances to be created and exposed as services. Managed component supports the same set of nested elements as for a `component`, and a subset of the attributes: `class`, `init-method`, `destroy-method`, `factory-method` and `factory-component`.

The signature of a destroy-method for a managed-component must follow the format:

```
public void anyMethodName(int reasonCode);
```

where reason code is one of:

- `ModuleContext.CONFIGURATION_ADMIN_OBJECT_DELETED`
- `ModuleContext.BUNDLE_STOPPING`

To have the properties of a managed component configured from the properties stored in its associated configuration object, simply use the nested managed-properties element as in the following example. The pid for the configuration object is automatically generated by the Configuration Admin service, and is different for each managed component instance. When the managed component instance is published as a service, the `service.pid` property is set to the value of the pid for its associated configuration object. A convention is adopted that specifying an empty string for the value of the `persistent-id` attribute when used nested inside of a `managed-service-factory` means “the persistent id of my associated configuration object”.

```
<managed-service-factory id="fooFactory" factory-pid="my.pid" interface="Foo">
  <managed-component class="SomeClass">
    <managed-properties persistent-id="" update-strategy=”container-managed”/>
    <property name="foo" ref="someOtherComponent"/>
  </managed-component>
</managed-service-factory>
```

Given the above definition, an instance of SomeClass will be created for each configuration object associated with the managed service factory “my.pid”. The instances are dependency injected with the properties found in the configuration object dictionary, and the property “foo” is also dependency injected with a reference to “someOtherComponent”. Each instance is registered as a service advertising the Foo interface.

The same convention of using an empty `persistent-id` attribute value applies to the `config-properties` too when nested inside a `managed-service-factory` element. The following example will publish all of the public properties from the associated configuration object as service properties of the service published for the associated managed component.

```
<managed-service-factory id="fooFactory" factory-pid="my.pid" interface="Foo">
  <osgi:service-properties>
    <config-properties persistent-id=””/>
  </osgi:service-properties>
  <managed-component class="SomeClass">
    <property name="foo" ref="someOtherComponent"/>
  </managed-component>
</managed-service-factory>
```

The component defined by a managed-service-factory is of type Map<ServiceRegistration,Object> and contains one entry for each service published by it where the key is the service registration object, and the value is the service itself. The Map membership is dynamically managed as configuration objects (and hence their associated services) come and go.

A typical use case for the managed service factory element might be to publish a DataSource service for each configuration object associated with the “data.source” factory pid. Adminstrators can then define, configure and publish new DataSource services simply by updating configuration information in the Configuration Admin service.

6.3.5 Direct access to configuration data

If you need to work directly with the configuration data stored under a given persistent id or factory persistent id, the easiest way to do this is to register a service that implements either the ManagedService or ManagedServiceFactory interface and specify the pid that you are interested in as a service property. For example:

```
<service interface="org.osgi.service.cm.ManagedService" ref="MyManagedService">
  <service-properties>
    <entry key="service.pid" value="my.managed.service.pid"/>
  </service-properties>
</service>
<component id="myManagedService" class="com.xyz.MyManagedService"/>
```

where the class MyManagedService implements org.osgi.service.cm.ManagedService.

6.4 Update option to recycle a component

This option was proposed as a simplified way of updating a component. The component would be deleted and recreated when its configuration changed. However this considerably complicates the lifecycle behaviour of components when we consider components which reference the deleted component (and components which reference those components, and so on transitively). Essentially this approach re-introduces the complexity of re-injecting properties into components.

7 Security Considerations

None.

8 Document Support

8.1 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]. RFC 155 is available in the OSGi Alliance subversion repository in rfcs/rfc0155/rfc-155-blueprint-namespaces.pdf.

[4]. RFC 124 v1.0 available in the OSGi Alliance subversion repository in rfcs/rfc0124/rfc-124.pdf.

[5]. OSGi Release 4 Version 4.2 Service Compendium.

[6]. RFC 150 “Configuration Admin Enhancements” is available in the OSGi Alliance subversion repository in rfcs/rfc0150/rfc-0150-ConfigAdminEnhancements.pdf.

8.2 Author’s Address

Name	Glyn Normington
Company	VMware
Address	SpringSource, Kenneth Dibben House, Enterprise Road, Chilworth, Southampton SO16 7NS, UK.
Voice	+44-2380-111512
e-mail	gnormington@vmware.com

8.3 Acronyms and Abbreviations

Configuration Admin is usually referred to as Config Admin (but never as Configuration Administration).

8.4 End of Document