RFC 124:

Blueprint Service

1.0

165 Pages

Abstract

The OSGi platform provides an attractive foundation for building enterprise applications. However it lacks a rich component model for declaring components within a bundle and for instantiating, configuring, assembling and decorating such components when a bundle is started. This RFC describes a set of core features required in an enterprise programming model and that are widely used outside of OSGi today when building enterprise (Java) applications. These features need to be provided on the OSGi platform for it to become a viable solution for the deployment of enterprise applications. The RFC is written in response to RFP 76

0 Document Information

0.1 Table of Contents

All Page Within This Box

0.2 Status

This document specifies the Press Release process for the OSGi Alliance, and requests discussion and suggestions for improvements. Distribution of this document is unlimited within the OSGi Alliance.

0.3 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.`

0.4 Revision History

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

Revision	Date	Comments
0.1 draft	Sep 13th 2007	First draft of this RFC Adrian Colyer, SpringSource, adrian.colyer@springsource.com
0.2 draft	Nov 28th 2007	Second draft of RFC2, incorporating design material from Spring Dynamic Modules 1.0 rc1

All Page Within This Box

Revision	Date	Comments
0.5 draft	Dec 21st 2007	Added first version of section 5.3. Defined mechanism for accessing ServiceReference objects for a collection.
0.8 draft	March 25th 2008	Addressed comments from January 2008 EEG meeting, completed section 5.3, added ModuleContextListener type.
0.9 draft	May 16th 2008	Addressed comments from John Wells and Alexandre Alves posted to EEG mailing list
0.9.1 draft	June 17th 2008	Addressed bugs • 702: don't proxy service references • 703 management of service references within a collection • 704 Class of a lazy init component is not reference until the component is about to be instantiated
0.9.2 draft	July 3rd 2008	Addressed bugs • 668: signature of destroy-method callback • 672: use "mandatory" and "optional" instead of 1..N etc. • 701: use 'Map' instead of 'Dictionary' • 667 remove 'legacy' spring constructs • 695 define API for accessing components and meta-data
0.9.3 draft	July 7th 2008	Addressed bugs • 699 specification of type conversions • Completed section 5.2.3, Module Context Events
0.9.4 draft	September 4th 2008	• Removed boilerplate text regarding press releases!! • Replaced a few stray references to „bean" with „component" Addressed bugs • 734 register module context as a service • 732 diagnostic events must be asynchronous • 744 dependency wait should be aborted on bundle stop • 738 determine when and how type converters are specified • 739 rethink how namespace handlers are defined • 668 signature of destroy method callback
0.9.5 draft	September 23rd 2008	• Changed name to „Blueprint Service" • Avoided over-specifying how requirement for namespace handler services should be determined • Added security section • Update schema URI and location to follow OSGi conventions

All Page Within This Box

Revision	Date	Comments
0.9.6 draft	October 17th 2008	Updates from OSGi BundleFest meeting in Montpelier: • 741 clarify the term Module • 705 clarify how extender actually loads classes • 788 comments from Graham Charters • 789 comments from Rick McGuire • 796 Bundle-Blueprint header allows directives to be specified multiple times • 795 additional comments from Rick McGuire • 800 clarify rules for constructor injection arguments • 805 managed-service and managed-service-factory elements should create services, not local components

0.9.7 draft	November 21st 2008	• 810 clarify interpretation of paths in Bundle-Blueprint header • 797 added osgi. Prefix to directives for Bunlde-SymbolicName • 811 context admin events should include a bundle property • 816 conflicting statement on inner component names • 817 how does module context handle aliases • 818 what is field and method injection • 819 ConstructorMetadataInjection not correct • 820 NamedParameterSpecification class • 822,823,824,825,826.830 – miscellaneous tidy up to doc of API classes • 858 use collections in preference to arrays • 861 clarify placement of different element types • 882 constructor resolution rules and factory methods • 888 schema for property types don't agree with specification • 910 remove alias concept and local attribute • 796 remove osgi. prefix from blueprint directives • 834 reference to wildcard *.xml still remaining • 846 error in grammar of manifest header • 848 inconsistent use of id and name • 841 osgi. prefix for scope types • 806 signature of ModuleContextListener methods • 859 ServiceExportComponentMetadata should return a dictionary • 833 conflicting statements about bundle state • 835 Inconsistent definition of wait for dependencies semantics • 839 ModuleContext.getComponent and lazy initialization • 840 ModuleContext.getComponent and scopes • 842 make registration/unregistration methods required on listener • 844 inconsistent timeout units • 845 reference to „ComponentRegistry“ should be „ComponentDefinitionRegistry“ • 847 EventAdmin events not consistently setting properties • 860 description element not defined • 869 getClassName javadoc updated for factory-component case • 870 need to allow both type and index on parameter specification • 871 order of ConstructorInjectionMetadata.getParameterSpecifications must be specified. • 875 is there a defined order for property injection?

All Page Within This Box

		• 886 default value of a placeholder when no value is found? • 864 startup handling of mandatory service references inconsistent • 866 BigInteger type conversion • 889 format for Locale conversion • 912 type conversion from String constructors • 838 primitive type conversions need more explicit definition • 918 clarify component state at injection time • 887 scope of property placeholders • 899 ModuleContext injection timing not defined
0.9.8	December 12th 2008	• 815 remove references to multiple extenders • 970 fixed a couple of typos • 928 clarified documentation of wait-for-dependencies and fixed bad attribute value for availability in schema • 862 LAZY_ACTIVATION and timeouts • 963 Described how listeners participate in dependency cycles • 942 ReferenceNameValue is not a suitable value option for ServiceExportComponentMetadata • 867 array type conversions • 836 key-type and value-type attributes for collections • 896 access to converters
0.9.9	January 7th 2009	• 862 qualfied that mandatory service timeouts are based on receipt of LAZY_ACTIVATION event for bundles with lazy activation. • 1003 assorted typos fixed • 1010 circular logic in lazy activation description • 973 handling of multiple service listener methods • 974 use Map not Dictionary in APIs • 986 removed errant references to local= attribute • 987 example of <ref> tag usage incorrect • 993 adding numbered headings to section 5.4.3.1 • 996 reference to Map in javadoc of CollectionBasedServiceReferenceMetadata deleted • 998 schema mixes identifiedType and TidentifiedType • 994 define scope of inner components

1.0	February 11th 2009	• 985 circular references • 1017 is lazy-init implied for prototype components? • 1018 interaction of depends-on and prototype scope • 1019 destroy-method calls for bundle-scoped components • 1020 do bundle scoped instances persist until module context destroy? • 1028 added example for case of unsatisfied optional dependency • 1035 fixed error in registration listener example • 1042 can <interfaces> element be used with ref. Collections? • 1043 should ref-set and ref-list have a timeout attribute • 1044 provide more explicit schema definition for listeners • 1045 document ref= attribute for listener tags • 1047 WAITING events for blocked service invocations • 1048 fixed bad reference to „osgi“ element in section 5.4.3.3 • 1051 use blueprint-compendium not blueprint/compendium in xmlns • 1054 allow limitation on final classes in section 5.4.2.1 • 1055 added constants for EventAdmin topics • 1070 clarified timing of CREATING event for bundles with LAZY_ACTIVATION • 1071 fixed broken cross-reference in section 5.5 • 1034 missing / in closing element tags • 1074 incorrect schema definition for default-properties element • 1077 replaced ModuleContextAware with moduleContext component • 1076 combine all implicit component definitions into one section • 1078 return value of NamespaceHandler.getSchemaLocation • 1080 confusing sentence regarding managed-properties and constructor-injection in section 5.7.3 • 992 Clarify usage of inner components • 1053 added attribute to specify desired member type for ref-set and ref-list • 1085 rename config-properties to cm-properties • 1084 handling of container managed properties on update • 1081 Reference state during bind/unbind callbacks • 1046 Clarify method signatures for bind and unbind • 1083 ListValue, SetValue, MapValue made consistent with other Value types • 1036 clarify the meaning of „request“ for prototype components • 1049 simplify sorting of reference collections • 1033 incomplete set of metadata Value types

		• 1090 access to pid for a managed-component • 1092 there is no requirement to use a fixed BSN • 1093 old reference to managed-service element replaced with managed-properties. • 1095 clarified behaviour of property-placeholder evaluation • 976 registering services with array property types • 975 array type specification • 1082 why is ParserContext.getEnclosingMetadata needed?

1 Introduction

In 2006 SpringSource (formerly known as Interface21), the company behind the Spring Framework ("Spring"), identified a complementary relationship between the application assembly and configuration features supported by Spring, and the modularity and versioning features of OSGi. Spring is primarily used to build enterprise Java applications. In this marketplace there is a need for a solution to versioning, simultaneous deployment of more than one version of a library, a better basis for dividing an application into modules, and a more flexible runtime and deployment model. OSGi provides a proven solution to these problems. The question became, how can enterprise application developers take advantage of OSGi (build enterprise applications as a set of OSGi bundles) when developing Spring applications?

In response to this challenge, the Spring Dynamic Modules project was born (formerly known as the Spring-OSGi project). Spring Dynamic Modules enables the use of Spring to configure both the internals of a bundle and also references between bundles. Even with little promotion the project quickly gathered a lot of attention. As of September 2007 there are over 800 users subscribed to the project's active discussion group. Enterprise developers have responded extremely positively to the direction being taken by the project. The Spring Dynamic Modules project is led by SpringSource, with committers from Oracle and BEA also active. The design of Spring Dynamic Modules has been influenced by discussion (both face-to-face and in the discussion group) with key personnel in the OSGi Alliance and from the equinox, Felix, and Knopflerfish OSGi implementations.

The strong interest in the Spring-OSGi project demonstrates that the enterprise Java market is attracted to the OSGi platform, and that the set of capabilities offered by Spring Dynamic Modules represent important additions to the OSGi platform. At the OSGi Enterprise Expert Group requirements meeting held in Dublin in January 2007 a working group was formed to create an RFP for adding these capabilities to OSGi. The resulting RFP, RFP 76, was accepted by the OSGi Alliance, and this RFC is written in response to the requirements documented there.

2 Application Domain

The primary domain addressed by this RFP is enterprise Java applications, though a solution to the requirements raised by the RFP should also prove useful in other domains. Examples of such applications include internet web applications providing contact points between the general public and a business or organization (for example, online stores, flight tracking, internet banking etc.), corporate intranet applications (customer-relationship management, inventory etc.), standalone applications (not web-based) such as processing stock feeds and financial data, and "front-office" applications (desktop trading etc.). The main focus is on server-side applications.

The enterprise Java marketplace revolves around the Java Platform, Enterprise Edition (formerly known as J2EE) APIs. This includes APIs such as JMS, JPA, EJB, JTA, Java Servlets, JSF, JAX-WS and others. The central component model of JEE is Enterprise JavaBeans (EJBs). In the last few years open source frameworks have become important players in enterprise Java. The Spring Framework is the most widely used component model, and Hibernate the most widely used persistence solution. The combination of Spring and Hibernate is in common use as the basic foundation for building enterprise applications. Other recent developments of note in this space include the EJB 3.0 specification , and the Service Component Architecture project (SCA).

Some core features of the enterprise programming models the market is moving to include:

- A focus on writing business logic in "regular" Java classes that are not required to implement certain APIs or contracts in order to integrate with a container
- Dependency injection: the ability for a component to be "given" its configuration values and references to any collaborators it needs without having to look them up. This keeps the component testable in isolation and reduces environment dependencies. Dependency injection is a special case of Inversion of Control.
- Declarative specification of enterprise services. Transaction and security requirements for example are specified in metadata (typically XML or annotations) keeping the business logic free of such concerns. This also facilitates independent testing of components and reduces environment dependencies.
- Aspects, or aspect-like functionality. The ability to specify in a single place behavior that augments the execution of one *or more* component operations.

In Spring, components are known as "beans" and the Spring container is responsible for instantiating, configuring, assembling, and decorating bean instances. The Spring container that manages beans is known as an "application context". Spring supports all of the core features described above.

2.1 Terminology and Abbreviations

1. Inversion of Control: a pattern in which a framework is in control of the flow of execution, and invokes user-code at appropriate points in the processing.
2. Dependency Injection: a form of inversion of control in which a framework is responsible for providing a component instance with its configuration values and with references to any collaborators it needs (instead of the component looking these up).
3. Aspect-oriented programming (AOP): a programming paradigm in which types known as "aspects" provide modular implementations of features that cut across many parts of an application. AspectJ is the best known AOP implementation.
4. Application Context: a Spring container that instantiates, configures, assembles and decorates component instances known as beans, also used to refer to an instance of a Spring container.

All Page Within This Box

5. Bean: a component in a Spring application context
6. JMS: Java Messaging Service
7. JPA: Java Persistence API
8. JavaServlets: Java standard for serving web requests
9. EJB: Enterprise JavaBeans component model defined by the Java Platform, Enterprise Edition
10. JTA: Java Transaction API
11. JSF: JavaServer Faces, component model for web user interfaces
12. JAX-WS: Java API for XML-based web services
13. Module context: a container instance responsible for instantiating, configuring, and managing components within a module. A bundle has 0..1 module contexts associated with it.
14. Managed component: a component instantiated and configured by a module context.

3 Problem Description

Enterprise application developers working with technologies such as those described in chapter 2 would like to be able to take advantage of the OSGi platform. The core features of enterprise programming models previously described must be retained for enterprise applications deployed in OSGi. The current OSGi specifications are lacking in the following areas with respect to this requirement:

- There is no defined component model for the internal content of a bundle. Declarative Services only supports the declaration of components that are publicly exposed.
- The configuration (property injection) and assembly (collaborator injection) support is very basic compared to the functionality offered by frameworks such as Spring.
- There is no model for declarative specification of services that cut across several components (aspects or aspect-like functionality)
- Components that interact with the OSGi runtime frequently need to depend on OSGi APIs, meaning that unit testing outside of an OSGi runtime is problematic
- The set of types and resources visible from the context class loader is unspecified. The context class loader is heavily used in enterprise application libraries
- Better tolerance of the dynamic aspects of the OSGi platform is required. The programming model should make it easy to deal with services that may come and go, and with collections of such services, via simple abstractions such as an injecting a constant reference to an object implementing a service interface, or to a managed collection of such objects. See the description of osgi:reference in the Spring Dynamic Modules specification for an example of the level of support required here.

Providing these capabilities on the OSGi platform will facilitate the adoption of OSGi as a deployment platform for enterprise applications. This should be done in a manner that is familiar to enterprise Java developers, taking into account the unique requirements of the OSGi platform. The benefits also extend to other (non-enterprise) OSGi applications that will gain the ability to write simpler, more testable bundles backed by a strong component model.

4 Requirements

1. The solution MUST enable the instantiation and configuration of components inside a bundle based on metadata provided by the bundle developer.
2. The solution SHOULD NOT require any special bundle activator or other code to be written inside the bundle in order to have components instantiated and configured.
3. The solution MAY choose to provide an extender bundle that is responsible for instantiating and configuring components inside a bundle with component metadata, when such bundles are started.
4. The solution SHOULD enable the creation of components inside a bundle to be deferred until the dependencies of those components are satisfied.
5. The solution MUST provide guarantees about the set of resources and types visible from the context class loader during both bundle initialization and when operations are invoked on services.
6. The solution MAY provide a means for components to obtain OSGi contextual information (such as access to a BundleContext) without requiring the programmer to depend on any OSGi “lookup” APIs. This is required so that components may be unit tested outside of an OSGi runtime.
7. The solution MUST provide a mechanism for a bundle component to be optionally exported as an OSGi service. It MAY provide scope management for exported service (for example, a unique service instance for each requesting bundle).
8. The solution MUST provide a mechanism for injecting a reference to an OSGi service into a bundle component. It SHOULD provide a constant service reference that the receiving component can use even if the target service backing the reference is changed at run time.
9. The solution MUST provide a mechanism for injecting a reference to a set of OSGi services into a bundle component. It SHOULD provide access to the matching OSGi services via a constant service reference that the receiving component can use even if the target services backing the reference change at run time.
10. The solution MUST provide a mechanism for service clients obtaining references as described to be notified when a backing target service is bound or unbound.
11. The solution SHOULD tolerate services in use being unregistered and support transparent rebinding to alternate services if so configured.
12. The solution SHOULD support configuration of bundle components with configuration data sourced from the OSGi Configuration Admin service. It SHOULD support re-injection of configuration value if configuration information is changed via the Configuration Admin service after the bundle components have been initially instantiated and configured.
13. The solution SHOULD provide a rich set of instantiation, configuration, assembly, and decoration options for components, compatible with that expected by enterprise programmers used to working with containers such as Spring.
14. The solution SHOULD allow multiple component instances to be created dynamically at runtime.
15. The solution SHOULD present a design familiar to enterprise Java developers.
16. The solution MUST enable bundles configured using the component model to co-exist with bundles using Declarative Services
17. The solution MUST define capabilities available on the OSGi minimum execution environment

18. The solution MAY define enhanced capabilities available on other execution environments, as long as there is a strict subset/superset relationship between the features offered in less capable execution environments and the features offered in more capable execution environments.

5 Solution

5.1 Architectural Overview

The runtime components to be created for a bundle, together with their configuration and assembly information, are specified declaratively in one or more configuration files contained within the bundle. This information is used at runtime to instantiate and configure the required components when the bundle is started. A bundle with such information present is known as a *managed bundle*. The configuration and assembly information can be regarded as a "blueprint" for creating the runtime components of the managed bundle, hence this RFC is named "Blueprint Service".

An extender bundle is responsible for observing the life cycle of such bundles. When a bundle is started, the extender creates a *module context*[1] for that bundle from its blueprint by processing the configuration files and instantiating, configuring, and assembling the components specified there. The module context is a lightweight container that manages the created components, known as *managed components*.When a managed bundle is stopped, the extender shuts down the module context, which causes the managed components within the context to be cleanly destroyed.

The declarative configuration for a bundle may also specify that certain of the bundle's managed components are to be exported as services in the OSGi service registry. In addition, it is possible to declare that a bundle component depends on a service or set of services obtained via the service registry, and to have those services dependency injected into the bundle component.

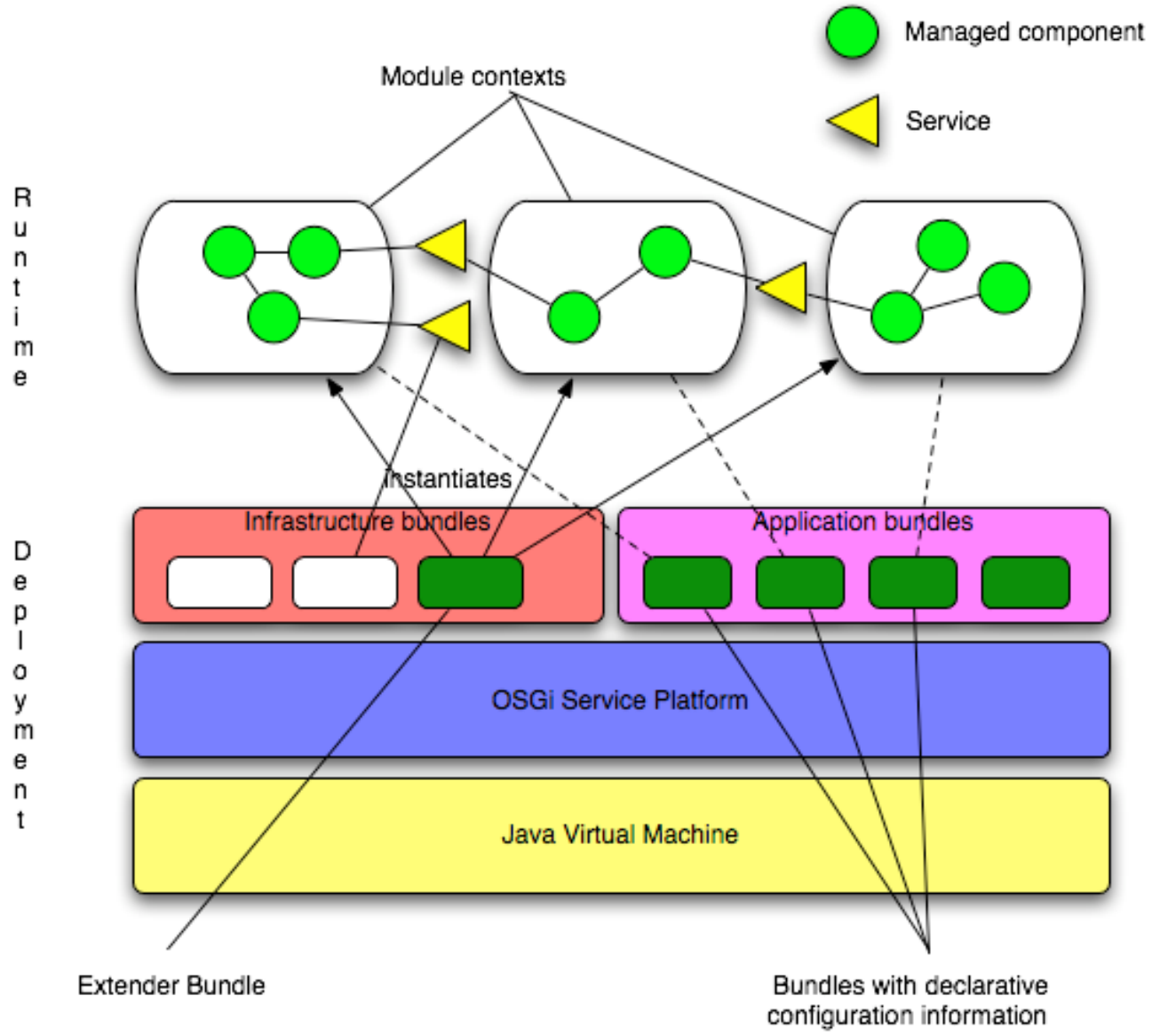
The solution therefore supports an application architecture in which modules are implemented as OSGi bundles with a module blueprint (the configuration information) and a runtime module context created from that blueprint. Modules are peers which interact via the service registry.

Figure 1 below provides a pictorial overview of the solution. Note that there is no reason an "infrastructure" bundle cannot also contain configuration information and have a module context automatically managed for it. From the perspective of the extender, all bundles are equal.

1 It is tempting to call this a "bundle context", but that could cause confusion with the BundleContext interface. In Spring this concept is know as an "application context".

All Page Within This Box

Figure 1 Solution Overview



The remainder of this section is structured as follows:

- Section 5.2 explains the relationship between bundles and module contexts and the role of the extender bundle
- Section 5.3 defines the configuration support for declaring components within a module context
- Section 5.4 defines how to export managed components as services in the service registry, and how to import references to services obtained via the registry
- Section 5.5 defines the interaction with the OSGi configuration administration service

All Page Within This Box

5.2 Module Context Life Cycle and the Extender Bundle

5.2.1 Module context creation and destruction

A Blueprint Service implementation must provide an extender bundle which manages the lifecycle of module contexts. This bundle is responsible for creating the module contexts for managed bundles (every ACTIVE managed bundle has one and only one associated module context). When the extender bundle is installed and started it looks for any existing managed bundles that are already in the ACTIVE state and creates module contexts on their behalf. In addition, it listens for bundle STARTED events and automatically creates a module context for any managed bundle that is subsequently started.

The extender bundle creates module contexts asynchronously. This behavior ensures that starting an OSGi Service Platform is fast and that bundles with service inter-dependencies do not cause deadlock on startup. A managed bundle may therefore transition to the ACTIVE state before its module context has been created. If module context creation fails for any reason then a context creation failure event will be published. The bundle remains in the ACTIVE state. There will be no services exported to the registry from the module context in this scenario.

When a module context has been successfully created, an instance of `org.osgi.service.blueprint.context.ModuleContext` describing the context is published in the service registry under the `org.osgi.service.blueprint.context.ModuleContext` interface. This service has a service property `osgi.blueprint.context.symbolicname` set to the bundle symbolic name of the bundle for which the context has been created. In addition a service property `osgi.blueprint.context.version` is set to the bundle version of the bundle for which the context has been created. The module context service must be published using the Bundle Context of the bundle it has been created for. This means that the symbolicname and version service properties are technically redundant because the same information can be determined by querying the ServiceRegistration object for the service to determine the bundle that published it. However, these properties are required to be published so that a service tracker can easily be given a filter expression that will wait for the publication of a module context service for a given bundle.

If a component to be created for a module context declares a mandatory dependency on the availability of certain OSGi services (see Section 5.4) then creation of the module context is blocked until the mandatory dependency can be satisfied through matching services available in the OSGi service registry. Since a service may come and go at any moment in an OSGi environment, this behavior only guarantees that all mandatory services were available at the moment creation of the module context began. One or more services may subsequently become unavailable again during the process of module context creation. Section 5.4 describes what happens when a mandatory service reference becomes unsatisfied.

A timeout applies to the wait for mandatory dependencies to be satisfied. If mandatory dependencies have not been satisfied before the timeout, then context creation fails. By default the timeout is set to 5 minutes, but this value can be configured using the timeout directive. See below for more information on manifest header entries and the available directives.

It is possible to change the module context creation semantics so that application context creation fails if all mandatory services are not immediately available upon startup.

When a managed bundle is stopped, the module context created for it is automatically destroyed. All services exported by the bundle will be unregistered (removed from the service registry) and any managed components within the module context that have specified destroy callbacks will have these invoked. Destruction of the module context for a STOPPING bundle happens synchronously. The context is guaranteed to be destroyed by the time the bundle transitions to the RESOLVED state.

If a managed bundle that has been stopped is subsequently re-started, a new module context will be created for it.

If the extender bundle is stopped, then all the module contexts created by the extender will be destroyed. Module contexts are shutdown in the following order:

1. Module contexts that do not export any services, or that export services that are not currently referenced, are shutdown in reverse order of bundle id. (Most recently installed bundles have their module contexts shutdown first).
2. Shutting down the module contexts in step (1) may have released service references these contexts were holding such that there are now additional module contexts that can be shutdown. If so, repeat step 1 again.
3. If there are no more active module contexts, we have finished. If there are active module contexts then there must be a cyclic dependency of references. The circle is broken by determining the highest ranking service exported by each context: the bundle with the lowest ranking service in this set (or in the event of a tie, the highest service id), is shut down. Repeat from step (1).

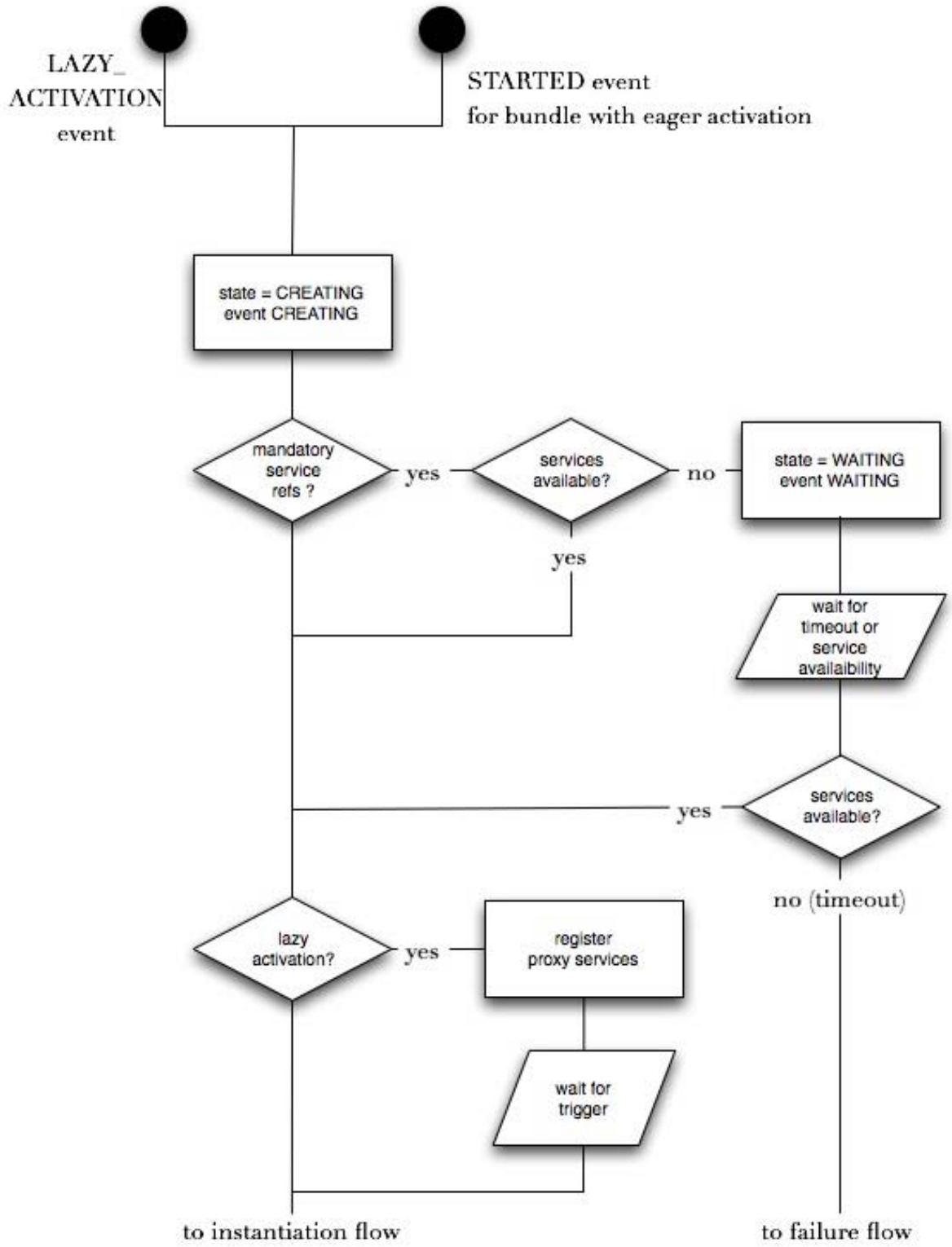
5.2.1.1 Lazy Activation

Since the OSGi R4.1 specification, a bundle may be marked as having lazy activation. Such a bundle issues a LAZY_ACTIVATION event when it is started, and then waits until a class is loaded from the bundle. On the first class load the STARTING event is issued and the Bundle Activator is invoked. When the activator returns the STARTED event is issued, and the bundle proceeds to the ACTIVE state.

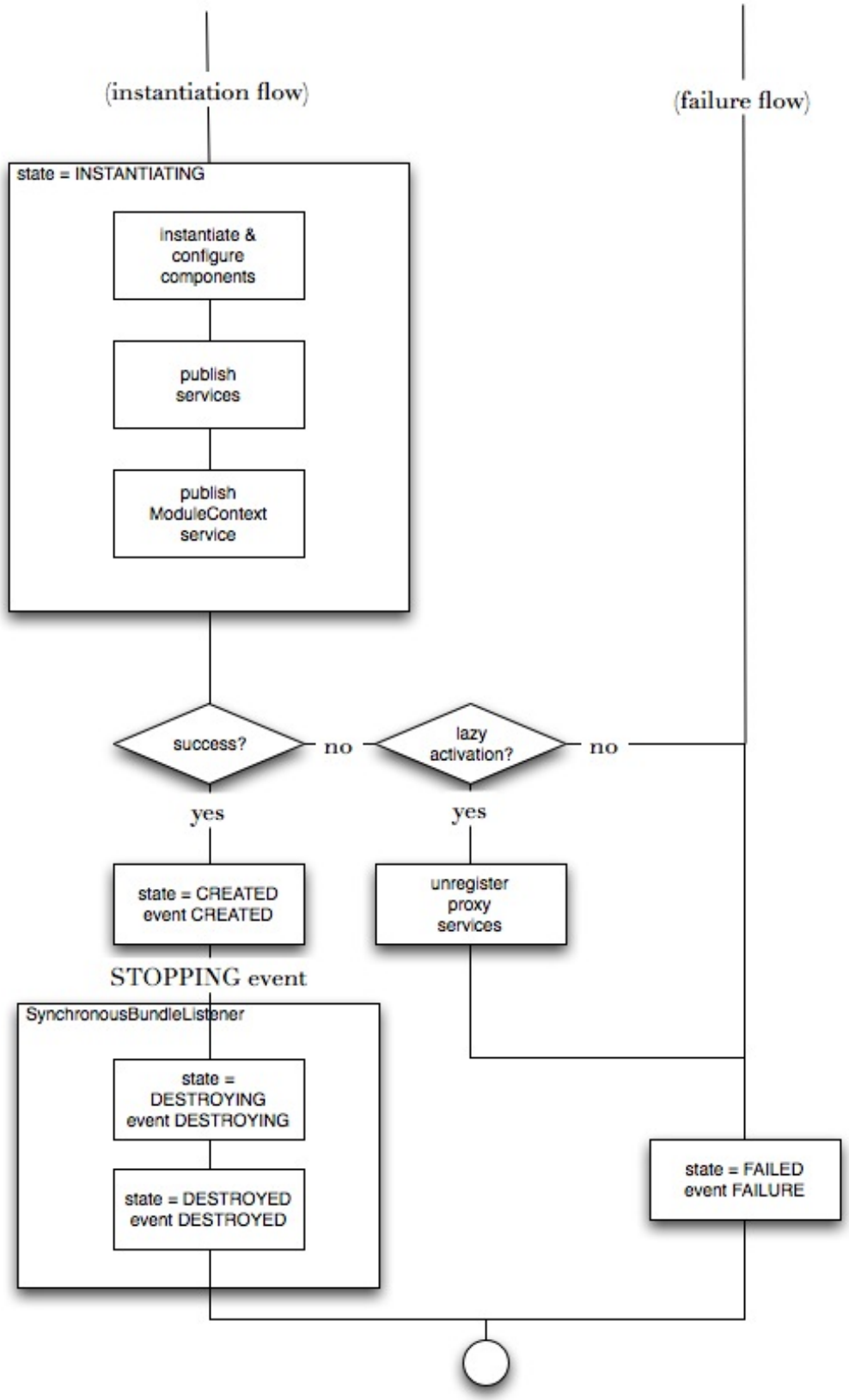
Module context instantiation is triggered when the STARTED event is issued for a given bundle, Therefore the module context for a lazily activated bundle will not be created until a class has been loaded from that bundle.

An additional consideration is the visibility of services registered by a module context when it is started. For a bundle that has been lazily activated but not yet started no module context has been created and hence the services are not yet published. Instead, placeholder services (ServiceFactory instances) are registered on the LAZY_ACTIVATION event so that these services can still be discovered and invoked by other bundles. The invocation of getService on a placeholder service factory also triggers the lazy bundle to be fully started. See section 5.4.1.6 for more details.

5.2.1.2 Module Life Cycle Summary



All Page Within This Box



All Page Within This Box

5.2.2 Manifest Headers for Managed Bundles

The extender recognizes a bundle as a *managed bundle* and will create an associated module context when the bundle is started if one or both of the following conditions is true:

- The bundle path contains a folder OSGI-INF/blueprint with one or more files in that folder with a '.xml' extension.
- META-INF/MANIFEST.MF contains a manifest header Bundle-Blueprint.

In the absence of the Bundle-Blueprint header the extender expects every ".xml" file in the OSGI-INF/blueprint folder to be a valid module context configuration file. A single module context is constructed from this set of files. Supporting multiple configuration files gives developers the option to separate out parts of their configuration. For example, security related definitions may be grouped together into a file of their own, or local component definitions and service import/export definitions split into separate files. From the perspective of the extender bundle, the choice of one or many configuration files makes no difference.

The Bundle-Blueprint manifest header may be used to specify an alternate set of configuration files. The resource paths are treated as relative resource paths and resolve to entries defined in the bundle and the set of attached fragments. When the Bundle-Blueprint header defines at least one configuration file location, any files in OSGI-INF/blueprint are ignored unless directly referenced from the Bundle-Blueprint header.

The syntax for the Bundle-Blueprint header value is:

```
header ::= clause (‘,’ clause)*
clause ::= path (‘;’ path)* (‘;’ parameter)*
```

Parameter can be either a directive or an attribute. This syntax is consistent with the OSGi Service Platform common header syntax defined in section 3.2.3 of the OSGi Service Platform Core Specification. The blueprint service itself defines no standard directives or attributes, but implementations of this specification may make use of this extension mechanism should they wish to do so.

For example, the manifest entry:

```
Bundle-Blueprint: config/account-data-context.xml, config/account-security-context.xml
```

will cause a module context to be instantiated using the configuration found in the files account-data-context.xml and account-security-context.xml in the bundle jar file.

Two directives are defined by the blueprint service specification to control the manner in which module context creation occurs. These directives are applied to the Bundle-SymbolicName header.

- `blueprint.wait-for-dependencies` (true|false)

  controls whether or not module context creation should wait for any mandatory service dependencies to be satisfied before proceeding (the default), or proceed immediately without waiting if dependencies are not satisfied upon startup.

  For example:

  ```
  Bundle-SymbolicName: org.osgi.foobar;blueprint.wait-for-dependencies:=false
  ```

  When wait-for-dependencies is false, context creation will begin immediately even if dependencies are not satisfied. For unsatisfied mandatory service references this means that the module context will proceed to the state it would be in had any such mandatory service references been available when module context creation began, but had subsequently become unavailable. The bundle will be in the ACTIVE state and any exported services (service components) that depend on the unsatisfied mandatory service references will

not be registered in the service registry until such time as the mandatory service reference becomes satisfied. Exported services that do not depend on the unsatisfied mandatory service reference will be registered in the service registry as usual. Service components that depend on the mandatory service reference will be injected with a service object that may not be backed by an actual service in the registry initially. See section 5.4 for more details.

- `blueprint.timeout (300000)`

  the time to wait (in milliseconds) for mandatory dependencies to be satisfied before giving up and failing module context creation. This setting is ignored if blueprint.wait-for-dependencies:=false is specified. The default is 5 minutes (300000 milliseconds).

  For example:

  ```
  Bundle-SymbolicName: org.osgi.foobar;blueprint.timeout:=60000
  ```

  Creates a module context that waits up to 1 minute (60 seconds) for its mandatory dependencies to appear.

5.2.3 Module Lifecycle Events

When a module context has been successfully created, the extender bundle must invoke the “contextCreated” operation of any registered services advertising support for the org.osgi.service.blueprint.context.ModuleContextListener interface. Only services with a compatible version of the interface will be invoked.

When creation of a module context fails for any reason, then the extender bundle must invoke the “contextCreationFailed” operation of any registered services advertising support for the org.osgi.service.blueprint.context.ModuleContextListener interface. Only services with a compatible version of the interface will be invoked.

Finer-grained information about the creation of module contexts is available if an EventAdmin service is available. When an EventAdmin service is available, events are published on the following topics:

- org/osgi/service/blueprint/context/CREATING – the extender has started to create a module context
- org/osgi/service/blueprint/context/CREATED – a module context has been successfully created
- org/osgi/service/blueprint/context/DESTROYING – the extender is destroying a module context
- org/osgi/service/blueprint/context/DESTROYED – a module context has been destroyed
- org/osgi/service/blueprint/context/WAITING – creation of a module context is waiting on the availability of a mandatory service, or a service invocation is waiting on the availability of a suitable backing service (see section 5.4.3).
- org/osgi/service/blueprint/context/FAILURE – creation of a module context has failed

All events are to be delivered asynchronously using the postEvent operation.

For each event the following properties are published:

- “bundle.symbolicName” (String) the symbolic name of the bundle for which the context is being created / destroyed.
- “bundle.id” (Long) the id of the bundle for which the context is being created / destroyed
- “bundle” (Bundle) the Bundle object of the bundle for which the context is being created or destroyed
- “bundle.version” (Version) the version of the bundle for which the context is being created / destroyed
- “timestamp” (Long) the time when the event occurred
- “extender.bundle” (Bundle) the Bundle object of the extender that is processing the context

All Page Within This Box

- “extender.bundle.id” (Long) the id of the extender bundle that is processing the context
- “extender.bundle.symbolicName” (String) the symbolic name of the extender bundle that is processing the context

In addition for a FAILURE event the “exception” property contains a Throwable detailing the failure cause. For a WAITING event, the “service.objectClass” (String[]) property details the interface type(s) of the service that the context is waiting on, and the “service.Filter” (String) property details the filter (if any). If the root cause of a FAILURE event is a timeout for a service wait, then the “service.objectClass” and “service.Filter” properties are also required to be present as part of the FAILURE event and these provide information about the service on which the wait failed. If the timeout occurred as a result of multiple services being unavailable then it is unspecified which of the unsatisfied references is reported in this manner.

The property names for module context events may be conveniently referenced using the constants defined in the org.osgi.service.event.EventConstants and org.osgi.service.blueprint.context.ModuleContextEventConstants interfaces.

A WAITING event is issued when a mandatory service is unavailable during context creation. An implementation may deliver one or more WAITING events for the same unsatisfied service reference before either the reference is satisfied, creation times out, or the bundle for which the context is being created is STOPPED.

If the bundle for which the module context is currently being created is STOPPED during the creation process then the DESTROYING and DESTROYED events will be published and any pending waits for that bundle (as notified by WAITING events) can be assumed to be cancelled.

5.3 Declaring Module Components

A module context configuration file contains component definitions using XML declarations from the `osgi` namespace (see section 5.7). As described in section 5.2.1.1, the complete blueprint for a module context may be comprised of definitions from multiple configuration files. There is at most one module context created for any given ACTIVE bundle.The module context container manages the lifecycle of these components. The basic structure of a configuration file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

   <description>
     Optional description for the blueprint defined in this file.
   </description>

   <component id="..." class="...">
     <!-- collaborators and configuration for this component go here -->
   </component>

   <component id="..." class="...">
     <!-- collaborators and configuration for this component go here -->
   </component>

   <!-- more component definitions go here... -->

</components>
```

If an error occurs during the processing of blueprint configuration files then creation of the context will fail with a ComponentDefinitionException (accessible via the “exception” property of the FAILURE event.

All Page Within This Box

A module container manages one or more *components* . These components are created using the configuration metadata (blueprint) that has been supplied to the container. Component definitions contain the following metadata:

• *a package-qualified class name:* typically this is the actual implementation class of the component being defined.

• component behavioral configuration elements, which state how the component should behave in the container (scope, lifecycle callbacks, and so forth).

• references to other components which are needed for the component to do its work; these references are also called *collaborators* or *dependencies*.

• other configuration settings to set in the newly created object. An example would be the number of connections to use in a component that manages a connection pool, or the size limit of the pool.

5.3.1 Naming Components

Every component has at most one id (also called an identifier, or name; these terms refer to the same thing). Component ids must be unique within a module.

Please note that you are not required to supply an id for a component. If no id is supplied explicitly, the container will generate a unique id for that component unless the component is an inner component (see **5.3.4.1.4 Inner Components** on page 29). Inner components are always anonymous.

5.3.2 Implicit Component Definitions

A module context contains a number of implicitly defined components with well-known names. A user defined component with the same name as one of these will override the implicit component definition, but it is not recommended to define such components.

5.3.2.1 ModuleContext component

The `ModuleContext` interface provides access to the component objects within the module context and to metadata describing the components within the context. A module context contains an implicitly defined component called '`moduleContext`' which supports the ModuleContext interface.

See section 5.8 for a description of the ModuleContext and ModuleContextAware interfaces.

The ModuleContext interface is useful for component lookup by name (where for example a component name has been specified via an external configuration mechanism such as the configuration admin service, or the component being looked up has a narrower scope than the component looking it up). It is also commonly used for introspection on the module context for management and other purposes.

Care should be taken with the use of the ModuleContext interface. In general the lookup pattern is discouraged and regular dependency injection is to be preferred where possible. When using the ModuleContext interface from within the module context it represents, the types of the components returned by the ModuleContext operations are guaranteed to be compatible with the caller. When using a ModuleContext interface from outside of the module context it represents (for example, after obtaining a reference to a ModuleContext via the service registry) then there is no guarantee of type compatibility or even visibility between the versions of the types of the returned components, and the versions of the types visible to the caller. Care must therefore be taken if casting the return value of ModuleContext.getComponent to a more specific type.

5.3.2.2 Bundle and BundleContext components

A module context contains two implicitly defined components, with ids “bundle” and “bundleContext” respectively. The “bundle” component is of type org.osgi.framework.Bundle and represents the bundle with which the module context is associated. The “bundleContext” component is of type org.osgi.framework.BundleContext and is the BundleContext object for the bundle with which the module context is associated.

All Page Within This Box

5.3.2.3 ConversionService component

A module context contains an implicitly defined component with id “conversionService” of type org.osgi.service.blueprint.convert.ConversionService, see section 5.3.4.1.6 for more information.

5.3.3 Instantiating Components

You can specify the type (or class) of object that is to be instantiated using the 'class' attribute of the <component/> element. The class element specifies the class of the component to be constructed in the common case where the container itself directly creates the component by calling its constructor reflectively (somewhat equivalent to Java code using the *'new'* operator). In the less common case where the container invokes a static, *factory* method on a class to create the component, the class property specifies the actual class containing the static factory method that is to be invoked to create the object (the type of the object returned from the invocation of the static factory method may be the same class or another class entirely, it doesn't matter).

Any types referenced in configuration elements and that need to be loaded as part of module context creation must be visible to the bundle defining the blueprint. All such types are loaded from the class space of the blueprint-defining bundle.

5.3.3.1 Instantiation using a constructor

When creating a component using the constructor approach, the class being created does not need to implement any specific interfaces or be coded in a specific fashion. Just specifying the component class should be enough. However, depending on what type of IoC you are going to use for that specific component, you may need a default (empty) constructor – this is required for “setter” injection.

You can specify your component class like so:

```
<component id="exampleComp" class="examples.Example"/>

<component name="anotherExample" class="examples.ExampleTwo"/>
```

The mechanism for supplying arguments to the constructor (if required), or setting properties of the object instance after it has been constructed, is described shortly.

5.3.3.2 Instantiation using a static factory method

When defining a component which is to be created using a static factory method, along with the class attribute which specifies the class containing the static factory method, another attribute named factory-method is needed to specify the name of the factory method itself. The container expects to be able to call this method (with an optional list of arguments as described later) and get back a live object, which from that point on is treated as if it had been created normally via a constructor. One use for such a component definition is to call static factories in legacy code.

The following example shows a component definition which specifies that the component is to be created by calling a factory-method. Note that the definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the createInstance() method must be a *static* method.

```
<component id="exampleComponent"  class="examples.ExampleComponent2"
    factory-method="createInstance"/>
```

The mechanism for supplying (optional) arguments to the factory method, or setting properties of the object instance after it has been returned from the factory, will be described shortly.

5.3.3.3 Instantiation using an instance factory method

In a fashion similar to instantiation via a static factory method, instantiation using an instance factory method is where a non-static method of an existing component from the container is invoked to create a new component. To

use this mechanism, the 'class' attribute must be left empty, and the 'factory-component' attribute must specify the name of a component[2] in the container that contains the instance method that is to be invoked to create the object. The name of the factory method itself must be set using the 'factory-method' attribute.

```
<!-- the factory component, which contains a method called createService() -->
<component id="serviceLocator" class="com.foo.DefaultServiceLocator">
  <!-- inject any dependencies required by this locator component -->
</component>

<!-- the component to be created via the factory component -->
<component id="exampleComponent"
  factory-component="serviceLocator"
  factory-method="createService"/>
```

5.3.4 Dependencies

A typical module is not made up of a single object (or component). Even the simplest of modules will no doubt have at least a handful of objects that work together. This next section explains how you go from defining a number of component definitions that stand alone to a fully realized module where objects work (or collaborate) together to achieve some goal.

5.3.4.1 Injecting Dependencies

The basic principle behind *Dependency Injection* (DI) is that objects define their dependencies (that is to say the other objects they work with) only through constructor arguments, arguments to a factory method, or properties which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually *inject* those dependencies when it creates the component. This is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the component itself being in control of instantiating or locating its dependencies on its own using direct construction of classes, or something like the *Service Locator* pattern.

5.3.4.1.1 Constructor Injection

Constructor-based DI is achieved by invoking a constructor with a number of arguments, each representing a dependency. Additionally, calling a factory method with specific arguments to construct the component can be considered almost equivalent, and the rest of this text will consider arguments to a constructor and arguments to a factory method similarly. Specifically, the <constructor-arg> element can also be used to specify arguments to factory methods, and the disambiguation rules for constructors also apply to disambiguation of overloaded factory methods.

The SimpleMovieLister class below an example of a class that could only be dependency injected using constructor injection. Notice that there is nothing *special* about this class.

```
public class SimpleMovieLister {
  // the SimpleMovieLister has a dependency on a MovieFinder
  private MovieFinder movieFinder;
```

[2] There are several ways to declare a named component. For example, a component may be declared using the <component> element, and also using the <reference> element defined in section 5.4.2. Namespace elements introduced by namespace handlers (see section 5.6) may provide alternate mechanisms for defining named components too.

```
  // a constructor so that the container can 'inject' a MovieFinder
  public SimpleMovieLister(MovieFinder movieFinder) {
    this.movieFinder = movieFinder;
  }
  // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

Constructor arguments are specified using the <constructor-arg> element. For example, an instance of the SimpleMovieLister class could be configured as follows:

```
  <component name="movieLister" class="SimpleMovieLister">
    <constructor-arg ref=”movieFinder”/>
  </component>
```

The ref attribute is used to refer to another component by name.

If the component class has multiple constructors and/or a constructor has multiple arguments then it becomes necessary to disambiguate the constructor to be invoked and the order in which the arguments are passed to the constructor. The disambiguation is done based on the constructor signature (number and type of arguments).

Consider the following class:

```
package x.y;
public class Foo {
  public Foo(Bar bar, Baz baz) {
    // ...
  }
}
```

Assuming that the Bar and Baz types only share java.lang.Object in common, the following configuration is valid:

```
  <component name="fooOne" class="x.y.Foo">
    <constructor-arg ref=”bar”/>
    <constructor-arg ref=”baz”/>
  </component>

  <component name="fooTwo" class="x.y.Foo">
    <constructor-arg ref=”baz”/>
    <constructor-arg ref=”bar”/>
  </component>

  <component id=”bar” class=”Bar”/>
  <component id=”baz’” class=”Baz”/>
```

If instead the constructor of Foo had signature (Object, Object) then there would be no way to determine based on type which argument each of the constructor args should be bound to. In this case the declaration order of the constructor-arg elements is used determine the order in which arguments are bound.

Constructor arguments can also have their index specified explicitly by use of the index attribute. For example:

```
<component id="exampleComponent" class="examples.ExampleComponent">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</component>
```

The value attribute here is used to specify a string value that will be converted to the type required by the constructor argument (see the section **5.3.4.1.7 Type Conversion** below).The declaration order of constructor-arg elements is not significant when the index attribute is used. The index attribute must be specified on every constructor-arg element if it is specified for any one of them. Indices are zero-based.

Consider a class that has multiple constructors with the same number of arguments, such as the class Multiple below:

```
class Multiple {
   public Multiple(String s1, String s2) {…}
   public Multiple(int i1, int i2) {…}
}
```

An attempt to configure an instance of Multiple as shown below will fail, because it is not possible to determine which constructor should be invoked.

```
<component id=”multiple” class=”Multiple”>
  <constructor-arg value=”123”/>
  <constructor-arg value=”456”/>
</component>
```

The type attribute can be used to disambiguate in such a case:

```
<component id=”multiple” class=”Multiple”>
  <constructor-arg type=”java.lang.String” value=”123”/>
  <constructor-arg type=”java.lang.String” value=”456”/>
</component>
```

5.3.4.1.2 Setter Injection

Setter-based DI is realized by calling setter methods on your components after invoking a no-argument constructor or no-argument static factory method to instantiate your component. It is also possible to mix both constructor-based and setter-based injection for the same component. For example, mandatory dependencies could be specified via constructor injection, and optional dependencies via setter injection. Setter injection is available for properties of the component defined following JavaBeans conventions. The order in which properties are injected is undefined.

Here is an example:

```
<component id="exampleComponent" class="examples.ExampleComponent">
  <property name="componentOne" ref=”anotherComponent”/>
  <property name="componentTwo" ref="yetAnotherComponent"/>
```

All Page Within This Box

```
  <property name="integerProperty" value="1"/>
</component>

<component id="anotherExampleComponent" class="examples.AnotherComponent"/>

<component id="yetAnotherComponent" class="examples.YetAnotherComponent"/>

public class ExampleComponent {
  private AnotherComponent compOne;
  private YetAnotherComponent compTwo;
  private int i;

  public void setComponentOne(AnotherComponent compOne) {
   this.compOne = compOne;
  }

  public void setComponentTwo(YetAnotherComponent compTwo) {
    this.compTwo = compTwo;
  }

  public void setIntegerProperty(int i) {
    this.i = i;
  }
}
```

As you can see, setters have been declared to match against the properties specified in the XML file, using JavaBeans conventions.

5.3.4.1.3 Properties and configuration details

Component properties and constructor arguments can be defined as either references to other managed components (collaborators), or values defined inline. A number of sub-element types are supported within the <property/> and <constructor-arg/> elements for just this purpose.

The **<value/>** element specifies a property or constructor argument as a human-readable string representation. The container converts these string values from a String to the actual type of the property or argument.

```
<component id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
   <!-- results in a setDriverClassName(String) call -->
   <property name="driverClassName">
     <value>com.mysql.jdbc.Driver</value>
   </property>
   <property name="url">
     <value>jdbc:mysql://localhost:3306/mydb</value>
   </property>
   <property name="username">
     <value>root</value>
   </property>
   <property name="password">
     <value>masterkaoli</value>
   </property>
</component>
```

For cases where you need a more specific type conversion than the type of the property or argument to be injected would imply, you can specify the fully-qualified name of the type that the string should be converted to

All Page Within This Box

using the optional “type” attribute of the value element. For example, if property is declared with type “Object”, but you want the value injected to be of type BigDecimal you could specify:

<value type=”java.math.BigDecimal”>12345</value>

The <property/> and <constructor-arg/> elements also support the use of the 'value' attribute, which can lead to much more succinct configuration. When using the 'value' attribute, the above component definition reads like so:

```
<component id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
       destroy-method="close">
  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
  <property name="username" value="root"/>
  <property name="password" value="masterkaoli"/>
</component>
```

The **idref** element is simply an error-proof way to pass the *id* of another component in the container (to a <constructor-arg/> or <property/> element). The motivation for injecting a component id as opposed to a component itself is that the recipient may then use the ModuleContext API (defined in section 5.5) to lookup the component by name at a later date (after instantiation and configuration). Lookup by name can be useful when the component being looked up has a narrower scope than the component referencing it (scopes are discussed later in section 5.3.5) – the component instance returned may vary with each lookup under these circumstances.

```
<component id="theTargetComponent" class="..."/>
<component id="theClientComponent" class="...">
  <property name="targetName">
    <idref component="theTargetComponent" />
  </property>
</component>
```

The above component definition snippet is *exactly* equivalent (at runtime) to the following snippet:

```
<component id="theTargetComponent" class="..." />
<component id="theClientComponent" class="...">
  <property name="targetName" value="theTargetComponent" />
</component>
```

The main reason the first form is preferable to the second is that using the idref tag allows the container to validate *at deployment time* that the referenced, named component actually exists. In the second variation, no validation is performed on the value that is passed to the 'targetName' property of the 'client' component. Any typo will only be discovered (with most likely fatal results) when the 'client' component is actually instantiated. If the 'client' component is a prototype component, this typo (and the resulting exception) may only be discovered long after the container is actually deployed.

The **ref** element is the final element allowed inside a <constructor-arg/> or <property/> definition element. It is used to set the value of the specified property to be a reference to another component managed by the container (a collaborator). As mentioned in a previous section, the referred-to component is considered to be a dependency of the component whose property is being set, and will be initialized on demand as needed (if it is a singleton component it may have already been initialized by the container) before the property is set.

Specifying the target component by using the component attribute of the <ref/> tag is the most general form, and will allow creating a reference to any component in the same module context (whether or not in the same XML file). The value of the 'component' attribute may be the same as either the 'id' attribute of the target component, or one of the values in the 'name' attribute of the target component.

```
<ref component="someComponent"/>
```

5.3.4.1.4 Inner Components

A <component/> element inside the <property/> or <constructor-arg/> elements is used to define a so-called *inner component*. An inner component definition does not need to have any id or name defined, if these attributes are specified they will be ignored by the container.

```
<component id="outer" class="...">
  <!-- instead of using a reference to a target component, simply define the target
       component inline -->
  <property name="target">
    <component class="com.example.Person"> <!-- this is the inner component -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </component>
  </property>
</component>
```

Note that in the specific case of inner components, the 'scope' attribute (see section 5.3.5) and any 'id' or 'name' attribute are effectively ignored. Inner components are *always* anonymous and always have *prototype* scope. Please also note that it is *not* possible to inject inner components into collaborating components other than the enclosing component. Inner components are are useful when:

- You have a component that should only be seen by its enclosing component, and
- You want to make it clear lexically that this is the case, and
- You want to avoid polluting the component namespace

5.3.4.1.5 Arrays

The `<array/>` element allows properties and arguments of array types to be defined and set. The `value-type` attribute specifies the type of the elements in the array. Nested inside the array element are the declarations for the array values themselves. An array value can be specified using any of the following elements:

- component
- reference
- service
- ref-set
- ref-list
- ref
- idref
- array
- list
- set
- map
- props
- value
- null

The reference, service, ref-set, and ref-list elements are introduced in section 5.4. Conversion will be attempted for each value to the type specified in the `array` element's `value-type` attribute.

All Page Within This Box

The following example shows the array element in use:

```
<component id="arrayExample" class="example.ArrayBean">

  <property name="myInts">
    <array value-type="int">
       <value>1</value>
       <value>2</value>
       <value>3</value>
       <value>5</value>
       <value>8</value>
    </array>
  </property>

  <property name="myStrings">
     <array value-type="java.lang.String">
       <value>one</value>
       <value>two</value>
       <value>three</value>
       <value>five</value>
       <value>eight</value>
     </array>
  </property>

</component>
```

5.3.4.1.6 Collections

The `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements allow properties and arguments of the Java Collection type `List`, `Set`, `Map`, and `Properties`, respectively, to be defined and set.

```
<component id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@example.org</prop>
        <prop key="support">support@example.org</prop>
        <prop key="development">development@example.org</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
        <value>a list element followed by a reference</value>
        <ref component="myDataSource" />
    </list>
  </property>
  <!-- results in a setSomeMap(java.util.Map) call -->
  <property name="someMap">
    <map>
        <entry>
            <key>
                <value>an entry</value>
            </key>
            <value>just some string</value>
```

All Page Within This Box

```
        </entry>
        <entry>
            <key>
                <value>a ref</value>
            </key>
            <ref component="myDataSource" />
        </entry>
    </map>
  </property>
  <!-- results in a setSomeSet(java.util.Set) call -->
  <property name="someSet">
    <set>
        <value>just some string</value>
        <ref component="myDataSource" />
    </set>
  </property>
</component>
```

Note that the value of a map key or value, property value, constructor argument value, or a set or list value can be any of the following elements:

- component
- reference
- service
- ref-set
- ref-list
- ref
- idref
- array
- list
- set
- map
- props
- value

The value of an array, map, property, set, or list value can also be <null/>. The key and value of a prop element are always strings. The reference, service, ref-set, and ref-list elements are introduced in section 5.4.

If you are using Java 5 or Java 6, you will be aware that it is possible to have strongly typed collections (using generic types). For example, it is possible to declare a Collection type such that it can only contain String elements. If you are dependency injecting a strongly-typed Collection into a component, you can take advantage of type-conversion support such that the elements of your strongly-typed Collection instances will be converted to the appropriate type prior to being added to the Collection. Conversion is attempted for key values and for values specified using either the value attribute or element. Implementations may optionally support type conversion from other value types (e.g. map, list, set and so on).
Consider the following class which has a strongly typed Map property “accounts”:

```
public class Foo {

    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
```

And the accompanying blueprint specification:

```
<components>
    <component id="foo" class="x.y.Foo">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </component>
</components>
```

When the 'accounts' property of the 'foo' component is being prepared for injection the string values '9.99', '2.75', and '3.99' will be converted into objects of type Float. This feature is only support on Java 5 or higher runtime environments.

Regardless of JDK level, you can also specify an explicit conversion to be attempted for array, map, list, and set values, and for map keys, using the value-type and key-type attributes. These work in a similar manner to the type attribute previously described for the value element. If a <map> element has a `key-type` specified then this is assumed to be the default type for all keys within the map unless overridden on an individual key basis (by using the type attribute of a value element nested inside a key element). If an <array>, <map>,< list>, or <set> element has a `value-type` specified then this is assumed to be the default type for all values within the collection, unless overridden on an individual value basis (by using the type attribute of the value element).

Finally, a <set> or <list> can also be injected into a property of array type (for example, int[]), providing that each entry in the list or set can be successfully converted to the base type of the array. Type conversion is discussed next.

5.3.4.1.7 Type Conversion

String values in module context configuration files must be converted to the type expected by an injection target (method or constructor parameter, or field) before being injected. The module context container supports a number of type conversions by default, and provides an extension mechanism for configuring additional type converters.

The default type conversions supported by the module context container are:

- String to any concrete type with a public String constructor
    - o If the target type has a public constructor that takes a single String argument, and no explicit converter is registered for that target type, then conversion is attempted by invoking the String constructor with the configuration value. This convention caters for many of the built-in JDK types such as BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short, File, URL, Date, and so on.
- String to primitive types. With the exception of boolean, which permits additional values (“yes”,”no”,”true”,”false”,”on”,”off”), the acceptable String formats for values are the same as defined for the String constructor of the associated wrapper type. Conversion to char supports the regular escape syntax for Java String literals.
- String to Boolean (permissible values are “yes”, “no”, “true”, “false”, “on”, “off”)String to Class
- String to Character (the string should contain only one character, escape syntax is the same as for Java string literals)
- String to Locale. The String should follow the syntax:

All Page Within This Box

Locale ::= language-code ("_" Country)+
Country ::= country-code ("_" variant-code)+
- Where language-code, country-code and variant-code are two character codes as defined by the javadoc for the Locale class.

- String to Pattern (only support on JDK 1.5 or later)
- String to java.lang.Properties (String must follow the format described in the JavaDoc for Properties.load[3])

Implementations of the blueprint service may also define additional built-in converters.

Users may define their own converters by implementing the `org.osgi.service.blueprint.convert.Converter` interface.

```
Interface Converter {

  public Class getTargetClass();

  public Object convert(Object source) throws Exception;

}
```

To register a converter it must be defined as a component and referenced from the `type-converters` element. Type converter elements are declared inside a components element before any component or other declarations. Type converter components may be declared directly inside the type-converters element using a component declaration, or referenced via a `ref` element. For example:

```
<type-converters>
  <!-- declare one or more type converters directly-->
  <component id="regionConverter" class="com.xyz.region.RegionConverter"/>

  <!—or simply reference components declared elsewhere -->
  <ref id="customDateConverter"/>
</type-converters>
```

All components declared within or referenced from a `type-converters` element must implement the `Converter` interface. Type converters are scoped to the module context and apply to all components defined for the context, regardless of the particular configuration file in which the converter or component may be defined.

Any converter declared in this way takes precedence over a built-in type converter when both are capable of performing the same conversion. If more than one explicitly declared type converter is capable of performing a given type conversion then:

- If the `targetClass` property (as returned by "getTargetClass") of one and only one declared type converter exactly matches the property type of the property to be configured, then that converter will be used
- Otherwise it is unspecified which of the candidate type converters will be used.

For example, if a component to be configured has a property of type `RegionCode`, and the type `EuropeanRegionCode` extends `RegionCode`, then:

- Given a registered type converter with `targetClass RegionCode` and a registered type converter with `targetClass EuropeanRegionCode`, the `RegionCode` converter will be used.

[3] See for example http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html#load(java.io.InputStream)

All Page Within This Box

- Given two registered type converters with `targetClass RegionCode` one of the two will be used, but it is unspecified which one it will be
- Given a registered type converter with `targetClass EuropeanRegionCode`, and another registered type converter with `targetClass AsianRegionCode` one of the two converters will be used, but it is unspecified which one it will be.

When declaring a component of type `Converter`, any configured properties of the converter component must rely only on the set of built-in type converters. This restriction also applies transitively to any components referenced by the converter component.

A bundle that exports a type implementing the Converter interface for use by other bundles should take care to specify the package of the targetClass type that the converter converts to in the uses clause for the package export. For example:

```
Export-Package: com.xyz.converters.region;uses:="com.xyz.region"
```

A bundle that references a type converter type defined locally does *not* need to export that type. When creating a module context, the extender bundle uses the class loader of the blueprint-defining bundle to load classes and resources for that context.

A module context contains an implicitly defined component with id “conversionService” of type org.osgi.service.blueprint.convert.ConversionService. Injecting a reference to the conversion service into a component provides access to the type conversions that the module context is able to perform.

5.3.4.1.8 Nulls

The `<null/>` element is used to handle null values. Empty arguments for properties and the like are treated as empty Strings. The following XML-based configuration metadata snippet results in the email property being set to the empty String value ("")

```
<component class="ExampleComponent">
  <property name="email"><value/></property>
</component>
```

This is equivalent to the following Java code: exampleComponent.setEmail(""). The special `<null>` element may be used to indicate a null value. For example:

```
<component class="ExampleComponent">
  <property name="email"><null/></property>
</component>
```

The above configuration is equivalent to the following Java code: exampleComponent.setEmail(null).

5.3.4.1.9 Configuration metadata shortcuts

The `<property/>`, `<constructor-arg/>`, and `<entry/>` elements all support a '`value`' attribute which may be used instead of embedding a full `<value/>` element. Therefore, the following:

```
<property name="myProperty">
  <value>hello</value>
</property>

<constructor-arg>
  <value>hello</value>
</constructor-arg>
```

All Page Within This Box

```
<entry key="myKey">
  <value>hello</value>
</entry>
```

are equivalent to:

```
<property name="myProperty" value="hello"/>

<constructor-arg value="hello"/>

<entry key="myKey" value="hello"/>
```

The <property/> and <constructor-arg/> elements support a similar shortcut 'ref' attribute which may be used instead of a full nested <ref/> element. Therefore, the following:

```
<property name="myProperty">
  <ref component="myComponent">
</property>

<constructor-arg>
  <ref component="myComponent">
</constructor-arg>
```

... are equivalent to:

```
<property name="myProperty" ref="myComponent"/>

<constructor-arg ref="myComponent"/>
```

Finally, the entry element allows a shortcut form to specify the key and/or value of the map, in the form of the 'key' / 'key-ref' and 'value' / 'value-ref' attributes. Therefore, the following:

```
<entry>
  <key>
    <ref component="myKeyComponent" />
  </key>
  <ref component="myValueComponent" />
</entry>
```

is equivalent to:

```
<entry key-ref="myKeyComponent" value-ref="myValueComponent"/>
```

5.3.4.1.10 Compound Property Names

Compound or nested property names are perfectly legal when setting component properties, as long as all components of the path except the final property name are not null. Consider the following component definition...

```
<component id="foo" class="foo.Bar">
  <property name="fred.bob.sammy" value="123" />
</component>
```

The foo component has a fred property which has a bob property, which has a sammy property, and that final sammy property is being set to the value 123. In order for this to work, the fred property of foo, and the bob property of fred must be non-null after the component is constructed, or a NullPointerException will be thrown.

5.3.4.2 Initialization guarantees

If a component A is directly or indirectly (for example via inclusion in a list or map) injected into a component B then in the absence of cyclic dependencies (see section 5.3.4.4), A is guaranteed to have been fully initialized before being injected.

5.3.4.3 Using depends-on

For most situations, the fact that a component is a dependency of another is expressed by the fact that one component is set as a property of another. This is typically accomplished with the <ref/> element. For the relatively infrequent situations where dependencies between components are less direct (for example, when a static initializer in a class needs to be triggered, such as database driver registration), the '`depends-on`' attribute may be used to explicitly force one or more components to be initialized before the component using this element is initialized. Find below an example of using the '`depends-on`' attribute to express a dependency on a single component.

```
<component id="compOne" class="ExampleComponent" depends-on="manager"/>
<component id="manager" class="ManagerComponent" />
```

If you need to express a dependency on multiple components, you can supply a comma-delimited list of component names as the value of the '`depends-on`' attribute.

```
 <component id="compOne" class="ExampleComponent" depends-on="manager,accountDao">
   <property name="manager" ref="manager" />
</component>
<component id="manager" class="ManagerComponent" />
<component id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

The '`depends-on`' attribute is used not only to specify an initialization time dependency, but also to specify the corresponding destroy time dependency (in the case of singleton components only). Dependent components that are defined in the 'depends-on' attribute will be destroyed after the relevant component itself being destroyed. This thus allows you to control shutdown order too.

5.3.4.4 Circular Dependencies

Dependency injection and the depends-on attribute create a dependency graph between components. An implementation of the blueprint service is not required to support cycles in the dependency graph. An implementation that does not support cycles should throw a ComponentDefinitionException (see section 5.5) if a cycle is detected. An implementation that provides some support for cyclic references may attempt to break cyclic dependencies by injecting a partially initialized (i.e. the object has been constructed, but all or some of the dependency injection setter methods may not yet have been invoked) component reference when possible. It is recommended that any such implementation issue a warning whenever such a situation occurs. When injecting a partially initialized component, the following guarantees must be made:

1. If a component declares an init method, it must be fully initialized before the init method is invoked
2. All components must be fully initialized before module context creation is complete (the CREATED event is posted, and the module context is published as a service).

If a cycle is detected that cannot be broken by such techniques, a ComponentDefinitionException must be thrown.

5.3.4.5 Lazily instantiated components

By default all singleton components in a module context will be pre-instantiated at startup. Pre-instantiation means that the module context will eagerly create and configure all of its singleton components as part of its initialization process. Generally this is a good thing, because it means that any errors in the configuration or in the surrounding environment will be discovered immediately (as opposed to possibly hours or even days down the line).

However, there are times when this behavior is not what is wanted. If you do not want a singleton component to be pre-instantiated you can selectively control this by marking a component definition as lazy-initialized. A lazily-initialized component is not created at startup and will instead be created when it is first requested.

Lazy loading is controlled by the '`lazy-init`' attribute on the `<component/>` element; for example:

```
<component id="lazy" class="com.foo.ExpensiveToCreateComponent" lazy-init="true"/>

<component name="not.lazy" class="com.foo.AnotherComponent"/>
```

The component named 'lazy' will not be eagerly pre-instantiated when the module context is starting up, whereas the 'not.lazy' component will be eagerly pre-instantiated.

Even though a component definition may be marked up as being lazy-initialized, if the lazy-initialized component is the dependency of a singleton component that is not lazy-initialized, then when the module context is eagerly pre-instantiating the singleton, it will have to satisfy all of the singletons dependencies, one of which will be the lazy-initialized component! In this situation a component that you have explicitly configured as lazy-initialized will in fact be instantiated at startup; all that means is that the lazy-initialized component is being injected into a non-lazy-initialized singleton component elsewhere.

It is also possible to control lazy-initialization at the container level by using the '`default-lazy-init`' attribute on the `<components/>` element; for example:

```
<components default-lazy-init="true">

    <!-- no components will be pre-instantiated... -->

</components>
```

The class referenced by the class attribute of a lazy-init component declaration is guaranteed not to be referenced in conjunction with that lazily initialized component until such time as the component is about to be instantiated.

5.3.5 Component Scopes

When you create a component definition what you are actually creating is a recipe for creating actual instances of the class defined by that component definition. The idea that a component definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular component definition, but also the scope of the objects created from a particular component definition. This approach is very powerful and gives you the flexibility to choose the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the Java class level. Components can be defined to be deployed in one of a number of scopes, specified using the scope attribute:

Scope Name	Description
Singleton	scopes a single component definition to a single object instance per module context

All Page Within This Box

Prototype	scopes a single component definition to any number of object instances
Bundle	scopes a single component definition to a single object per requesting client bundle

When a component is a singleton, only one shared instance of the component will be managed, and all requests for components with an id or ids matching that component definition will result in that one specific component instance being returned by the container.

<component id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

Singleton is the default scope if no scope is explicitly specified.

The non-singleton `prototype` scope of component deployment results in the creation of a new component instance every time the component is referenced for dependency injection, or is looked up using the `ModuleContext.getComponent` operation. Simply declaring a component with prototype scope does not on its own cause an instance to be created. As a rule of thumb, you should use the prototype scope for all components that are stateful, while the singleton scope should be used for stateless components.

In the following example `accountClientOne` and `accountClientTwo` will each be injected with their own unique instance of the `accountService` component:

```
<component id="accountService" class="com.foo.DefaultAccountService"
scope="prototype"/>

<component id=”accountClientOne” class=”…”>
  <property name=”accountService” ref=”accountService”/>
</component>

<component id=”accountClientTwo” class=”…”>
  <property name=”accountService” ref=”accountService”/>
</component>
```

Note that naming a component with prototype scope in a depends-on attribute value will trigger creation of an “orphaned” instance of the prototype component and is not generally recommended.

See section 5.4.1 for a discussion of the bundle scope. Implementations of this RFC are free to define additional scope values beyond those defined here.

5.3.6 Lifecycle

Specifying an `init-method` for a component enables a component to perform initialization work once all the necessary properties on a component have been set by the container. Specifying a `destroy-method` enables a component to get a callback when the module context containing it is destroyed. Destroy-method callbacks are not supported for components with a prototype scope – it is the responsibility of the application to manage the lifecycle of prototype instances after they have been created. If the destroy-method attribute is used with a bean of prototype scope, it will be ignored.

```
<component id="exampleInitComponent"
        class="examples.ExampleComponent" init-method="init"/>

public class ExampleComponent {
    public void init() {
        // do some initialization work
```

All Page Within This Box

```
    }
}


<component id="exampleDestroyComponent" class="examples.ExampleComponent" destroy-
method="cleanup"/>
public class ExampleComponent {
    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

The container can be configured to 'look' for named initialization and destroy callback method names on every component. This means that you, as an application developer, can simply write your application classes, use a convention of having an initialization callback called init() (or any other name of your choosing), and then (without having to configure each and every component with an 'init-method="init"' attribute) be safe in the knowledge that the container will call that method when the component is being created. To specify default init and destroy methods use the `default-init-method` and `default-destroy-method` attributes on the enclosing `components` element. For example:

```
<components default-init-method="onInit" default-destroy-method="onDestroy">
  <component id="someComponent" class="SomeClass">
    <!-- onInit() and onDestroy() methods will be called if implemented by
SomeClass -->
  </component>
</components>
```

A component that specifies its own init-method or destroy-method attribute overrides any value specified in the corresponding default attribute of the components element. An init-method or destroy-method attribute with an empty value indicates that no init or destroy method (respectively) should be invoked.

If the type of a component does not have a method corresponding to a default-init-method or default-destroy-method attribute value (and the component definition does not override these values) then this is not treated as an error and the init or destroy method invocation is simply omitted. If the type of a component does not have a method corresponding to an init-method or destroy-method attribute value then this is an error and context creation will fail.

The method referenced by an init-method or default-init-method attribute must return void, and take no arguments. The method referenced by a destroy-method or default-destroy-method attribute must return void, and take no arguments.

The invocation of init methods follows component creation order (which follows the dependency graph). When an init-method is invoked on a component, any components that have been injected into the component will have already been instantiated and had their init-methods called. The invocation of destroy methods also follows component creation order (which follows the dependency graph). Thus when a component has its destroy method invoked, all of the components that component depends upon are still fully functional.

If any component code invoked during component instantiation and configuration (for example, constructors, setter methods, init methods) throws an exception then this is considered as fatal and context creation will fail. Any already constructed components will be destroyed (including invocation of destroy-methods if specified).

If a destroy-method implementation throws an exception the module context should catch the exception and recover to continue destroying any remaining components. If an implementation supports logging then the exception should be logged.

5.4 Interacting with the Service Registry

The `osgi` namespace provides elements that can be used to export managed components as OSGi services, to define references to services obtained via the registry.

5.4.1 Exporting a managed component to the Service Registry

The `service` element is used to define a component representing an exported OSGi service. The service element can be declared anywhere that a component element could be used. At a minimum you must specify the managed component to be exported, and the service interface that the service advertises.

For example, the declaration

```
<service ref="componentToPublish" interface="com.xyz.MessageService"/>
```

exports the component with name componentToPublish as a service in the OSGi service registry with interface com.xyz.MessageService. The published service will have a service property with the name `osgi.service.blueprint.compname` set to the component id of the target component being registered (componentToPublish in this case).

The component defined by the `service` element is of type `org.osgi.framework.ServiceRegistration` and is the `ServiceRegistration` object resulting from registering the exported component with the OSGi service registry. By giving this component an XML id you can inject a reference to the `ServiceRegistration` object into other components if needed. For example:

```
<service id="myServiceRegistration"
        ref="componentToPublish" interface="com.xyz.MessageService"/>
```

As an alternative to exporting a named component, the component to be exported to the service registry may be defined as an anonymous inner component of the service element.

```
<service interface="com.xyz.MessageService">
  <component class="SomeClass">
     ...
  </component>
</service>
```

In this case the blueprint service will not advertise an osgi.blueprint.service.compname service property for the registered service.

If the component to be exported implements the `org.osgi.framework.ServiceFactory` interface then the ServiceFactory contract is honored as per section 5.6 of the OSGi Service Platform Core Specification. As an alternative to implementing this OSGi API, this RFC introduces a component scope known as *bundle* scope. When a component with bundle scope is exported as an OSGi service then one instance of the component will be created for each unique client (service importer) bundle that obtains a reference to it through the OSGi service registry. Once created, a component with bundle scope has its lifecycle managed by the module context and will be disposed (and any associated destroy-method invoked) when either the containing module context is disposed, or the reference count from the service-importing bundle drops to zero (e.g. that bundle is stopped). If the service importing bundle subsequently references the component again after previously releasing all references, a new instance will be created.

To declare a component with bundle scope use the scope attribute of the component element:

```
<service ref="compToBeExported" interface="com.xyz.MessageService"/>

<component id="compToBeExported" scope="bundle"
class="com.xyz.MessageServiceImpl"/>
```

A component with bundle scope that is not exported as a service simply behaves as a component with scope singleton.

5.4.1.1 Controlling the set of advertised service interfaces for an exported service

The OSGi Service Platform Core Specification defines the term service interface to represent the specification of a service's public methods. Typically this will be a Java interface, but the specification also supports registering service objects under a class name, so the phrase service interface can be interpreted as referring to either an interface or a class.

There are several options for specifying the service interface(s) under which the exported service is registered. The simplest mechanism, shown above, is to use the interface attribute to specify a fully-qualified interface name. To register a service under multiple interfaces the nested `interfaces` element can be used in place of the interface attribute.

```
<service ref="componentToBeExported">
  <interfaces>
     <value>com.xyz.MessageService</value>
     <value>com.xyz.MarkerInterface</value>
  </interfaces>
</service>
```

The `interface` attribute must not be used in conjunction with the `interfaces` element.

Using the `auto-export` attribute you can avoid the need to explicitly declare the service interfaces at all by analyzing the object class hierarchy and its interfaces.

The `auto-export` attribute can have one of four values:

- `disabled` : the default value; no auto-detected of service interfaces is undertaken and the interface attribute or interfaces element must be used instead.
- `interfaces` : the service will be registered using all of the Java interface types implemented by the component to be exported
- `class-hierarchy` : the service will be registered using the exported component's implementation type and super-types (up to but not including java.lang.Object).
- `all-classes` : the service will be registered using the exported component's implementation type and super-types (up to but not including java.lang.Object) plus all interfaces implemented by the component.

For example, to automatically register a component under all of the interfaces that it supports you would declare:

```
<service ref="componentToBeExported" auto-export="interfaces"/>
```

Given the interface hierarchy:

```
public interface SuperInterface {}

public interface SubInterface extends SuperInterface {}
```

then a service registered as supporting the `SubInterface` interface is not considered a match in OSGi when a lookup is done for services supporting the `SuperInterface` interface. For this reason it is a best practice to

All Page Within This Box

export all interfaces supported by the service being registered explicitly, using either the `interfaces` element or `auto-export="interfaces"`.

5.4.1.2 Controlling the set of advertised properties for an exported service

As previously described, an exported service is always registered with the service property `osgi.service.blueprint.compname` set to the name of the component being exported. Additional service properties can be specified using the nested `service-properties` element. The `service-properties` element contains key-value pairs to be included in the advertised properties of the service. The key must be a string value, and the value must be a type recognized by OSGi Filters. See section 5.5 of the OSGi Service Platform Core Specification for details of how property values are matched against filter expressions.

The `service-properties` element must contain at least one nested `entry` element. For example:

```
<service ref="componentToBeExported" interface="com.xyz.MyServiceInterface">
  <service-properties>
    <entry key="myOtherKey" value="aStringValue"/>
    <entry key="aThirdKey" value-ref="componentToExposeAsProperty"/>
  <service-properties>
</service>
```

See section 5.5 for details on how to register service properties sourced from the Configuration Admin service.

When registering a property for a service, you must explicity specify the value type unless you wish the value to be registered as a string. The OSGi specification (section 3.2.6) allows the following values for service properties:

- A primitive (int, long, float, double, byte, short, char, boolean). A value with one of these types can be specified using e.g.

```
<entry key="someKey">
  <value type="int">123</value>
</entry>
```

- One of the scalar types String, Integer, Long, Float, Double, Byte, Short, Character, Boolean. A value with one of these types can be specified using e.g.

```
<entry key="someKey">
  <value type="java.lang.Integer">123</value>
</entry>
```

  (Remember that no special treatment is needed for String values)

- An array of either the allowable primitive or scalar types. A value with one of these types can be specified using e.g.

```
<entry key="someKey">
  <array value-type="java.lang.String">
    <value>first</value>
    <value>second</value>
  </array>
</entry>
```

All Page Within This Box

- A Collection of scalar types. A value with one of these types can be specified using e.g.

```
<entry key="someKey">
  <set>
    <value>First</value>
    <value>Second</value>
  </set>
</entry>
```

5.4.1.3 The depends-on attribute

The optional depends-on attribute can be used to provide a comma-delimited list of component names for components that must be instantiated and configured before the service is published to the registry. The depends-on attribute is used to capture indirect dependencies required for operation of the service that cannot be explicitly determined through the configuration of the component to be exposed as a service. Dependencies listed in the depends-on clause only gate service registration, they do not propagate to the creation of the component referenced by the service element using the ref attribute.

5.4.1.4 The ranking attribute

When registering a service with the service registry, you may optionally specify a service ranking (see section 5.2.5 of the OSGi Service Platform Core Specification). When a bundle looks up a service in the service registry, given two or more matching services the one with the highest ranking will be returned. The default ranking value is zero. To explicitly specify a ranking value for the registered service, use the optional `ranking` attribute.

```
<service ref="componentToBeExported" interface="com.xyz.MyServiceInterface"
  ranking="9"/>
```

5.4.1.5 Registration Listener

The service defined by a `service` element is registered with the OSGi service registry when the module context is first created. It will be unregistered automatically when the bundle is stopped and the module context is disposed. Services are also unregistered and re-registered if a mandatory dependency of the service is unsatisfied or becomes satisfied again (see section 5.4.2). The `ServiceRegistration` object representing the service component is a proxy that always delegates to the current OSGi service platform `ServiceRegistration` for the service. If an operation is invoked on the `ServiceRegistraton` proxy at a time when there is no underlying `ServiceRegistration` (because the service has been unregistered, but not yet re-registered) then an `IllegalStateException` will be thrown.

If you need to take some action when a service is registered or unregistered then you can define a listener component using the nested `registration-listener` element.

The declaration of a registration listener must use either the `ref` attribute to refer to a top-level component definition, or declare an anonymous listener component inline. For example:

```
<service ref="componentToBeExported" interface="SomeInterface">
  <registration-listener ref="myListener"                    (1)
     registration-method="serviceRegistered"                 (2)
     unregistration-method="serviceUnregistered"/>           (2)
  <registration-listener
     registration-method=”onRegister”
```

All Page Within This Box

```
    unregistration-method=”onUnregister” >              (3)
    <component class="SomeListenerClass"/>              (4)
  </registration-listener>
</service>
```

(1) Listener declaration referring to a top-level component.

(2) The registration and unregistration methods to be invoked on the component referenced in (1).

(3) The registration and unregistration methods to be invoked on the component defined in (4)

(4) Listener component declared anonymously in-line.

The required `registration-method` and `unregistration-method` attributes specify the names of the methods defined on the listener component that are to be invoked during registration and unregistration. Registration and unregistration callback methods must have a signature matching with the following format:

```
public void anyMethodName(ServiceType serviceInstance, Map serviceProperties);
```

where `ServiceType` can be any type compatible with the exported service interface of the service.

The register callback is invoked when the service is initially registered at startup, and whenever it is subsequently re-registered. The unregister callback is invoked during the service unregistration process, no matter the cause (such as the owning bundle stopping). The `ServiceRegistration` proxy representing the service component is guaranteed to be backed by a valid `ServiceRegistration` object during register and unregister callbacks.

The registration/unregistration methods are only invoked when a service of a type compatible with the declared `ServiceType` is registered/unregistered.

The components refered to by registration-listener elements (either via the ref attribute or by direct inline declaration of a component) are treated as dependencies of the enclosing service component for the purposes of lifecycle management, dependency injection, and cycle detection. Given a service component `S`, with a listener `L`, it must however be possible to inject a reference to `S` into `L`. For example:

```
<service id="S" ref="SomeComponent" interface="I">
  <registration-listener
            registration-method="onRegister"
            unregistration-method="onUnregister"
            ref="L"/>
  </registration-listener>
</service>

<component id="L" class="C">
  <property name="serviceRegistration" ref="S"/>
</component>
```

5.4.1.6 Lazy Activation

When a bundle specifies lazy activation in its Bundle-ActivationPolicy manifest header, and that bundle is started with the “START_ACTIVATION_POLICY” flag set, then the bundle will wait in the STARTING state until the first class load from the bundle occurs. For a bundle with a module context blueprint that defines one or more services, this means that without special treatment, none of these services would be visible until a class load occurs. An important use case is to enable the first dereference of a service reference to trigger creation of the module context.

For a bundle that specifies the lazy activation policy and is started accordingly, the following steps are taken:

- The OSGi Service Platform creates a BundleContext object for the bundle
- The bundle state is moved to the STARTING state
- The LAZY_ACTIVATION event is fired
- A synchronous Blueprint Service event listener receives the LAZY_ACTIVATION event
    - If an EventAdmin service is present, an org/osgi/service/blueprint/context/CREATING event is published
    - The configuration metadata for the bundle is parsed, and any service elements are detected
    - The Blueprint Service waits for any mandatory service references of the context to be satisfied, according to the rules discussed in section 5.2. Any timeouts are relative to the receipt of the LAZY_ACTIVATION event (not the STARTING or STARTED events). Once the services are satisfied:
    - For any service that can be lazily registered, a lazy ServiceFactory is registered in the service registry to represent that service
        - A service can be lazily registered if the interface or interfaces under which the service is to be registered are explicitly specified and are Java interface types, and there are no registration listeners defined.
        - The lazy service advertises the same interfaces and properties as the ‘real’ service that will be created when the module context is created.
    - No components from the module context are instantiated during this process
    - The synchronous event listener returns

From this point, there are two triggers that can cause the module context to be created: a class load from the bundle, or a dereferencing of the ServiceReference (invocation of getService on the registered ServiceFactory for the service) for a lazy service.

In the first case:

- The system waits for a class load from the bundle to occur
- The normal STARTING event is fired by the framework
- The bundle is activated by the framework
- The STARTED event is fired and the framework moves the bundle to the ACTIVE state
- Asynchronous creation of the module context begins on receipt of the STARTED event
    - Any lazy ServiceFactory objects registered during LAZY_ACTIVATION are updated to return the true services defined in the module blueprint.

All Page Within This Box

In the second case:

- The ServiceFactory method getService is invoked on a lazy service (because a client has passed the ServiceReference obtained from the registry to the BundleContext.getService operation).
- The Bundle.start(START_TRANSIENT) operation is invoked to force the bundle to start immediately
- The normal STARTING event is fired by the framework
- The bundle is activated by the framework
- The STARTED event is fired and the framework moves the bundle to the ACTIVE state
- *Synchronous* creation of the module context begins
    - Any lazy ServiceFactory objects registered during LAZY_ACTIVATION are updated to return the true services defined in the module blueprint.
    - Module context creation completes
- The getService operation returns the true service from the newly created module context

If module context creation (however triggered) fails for any reason, then any registered placeholder services for that context must be unregistered.

Note that the potential for race conditions exists under concurrent class loading and/or service access. An implementation of the Blueprint Service must ensure that only one module context is instantiated under such conditions.

5.4.2 Defining References to OSGi Services

The Blueprint Service supports the declaration of components that represent services accessed via the OSGi Service Registry. In this manner references to OSGi services can be injected into bundle components. The service lookup is made using the service interface type that the service is required to support, plus an optional filter expression that matches against the service properties published in the registry.

For some use cases, a single matching service that meets the application requirements is all that is needed. The `reference` element defines a reference to a single service that meets the required specification. In other scenarios, especially when using the OSGi whiteboard pattern, references to all available matching services are required. The Blueprint Service supports the management of this set of references as a List or Set. References to services can be declared anywhere that a component element could be declared.

5.4.2.1 Referencing an individual service

The `reference` element is used to define a reference to a service in the service registry.

Since there can be multiple services matching a given description, the service returned is the service that would be returned by a call to BundleContext.getServiceReference. This means that the service with the highest ranking will be returned, or if there is a tie in ranking, the service with the lowest service id (the service registered first with the framework) is returned (please see Section 5 from the OSGi specification for more information on the service selection algorithm).

Interface attribute and interfaces element

The `interface` attribute identifies the service interface that a matching service must implement. For example, the following declaration creates a reference component called `messageService`, which is backed by the service returned from the service registry when querying it for a service offering the `MessageService` interface.

All Page Within This Box

```
<reference id="messageService" interface="com.xyz.MessageService"/>
```

Just as with the `service` element, when specifying multiple interfaces, use the nested `interfaces` element instead of the `interface` attribute:

```
<reference id="importedOsgiService">
  <interfaces>
     <value>com.xyz.MessageService</value>
     <value>com.xyz.MarkerInterface</value>
  <interfaces>
</reference>
```

It is illegal to use both the `interface` attribute and the `interfaces` element at the same time.

The component defined by the `reference` element implements all of the advertised service interfaces of the service *that are visible to the bundle*. Implementations of this RFC may choose to document a limitation that class-based (as opposed to interface-based) service interfaces that include final methods, or classes that are final themselves, are not supported.

Filter attribute

The optional `filter` attribute can be used to specify an OSGi filter expression and constrains the service registry lookup to only those services that match the given filter.

For example:

```
<reference id="asyncMessageService" interface="com.xyz.MessageService"
  filter="(asynchronous-delivery=true)"/>
```

will match only OSGi services that advertise the `MessageService` interface and have the property named asynchronous-delivery set to value 'true'.

Component name attribute

The `component-name` attribute is a convenient short-cut for specifying a filter expression that matches on the component name property automatically set when exporting a component using the `service` element.

For example:

```
<reference id="messageService" interface="com.xyz.MessageService"
   component-name="defaultMessageService"/>
```

will match only OSGi services that advertise the `MessageService` interface and have the property named `osgi.service.blueprint.compname` set to value `defaultMessageService`.

If both a filter attribute value and a component-name attribute value are specified, then matching services must satisfy the constraints of both.

Availability attribute

The `availability` attribute is used to specify whether or not a matching service is required at all times. An availability value of `mandatory` (the default) indicates that a matching service must always be available. An availability value of `optional` indicates that a matching service is not required at all times. A reference with

All Page Within This Box

mandatory availability is also known as a *mandatory service reference* and, by default, module context creation is deferred until the reference is satisfied.

Note: It is an error to declare a mandatory reference to a service that is also exported by the same bundle, this behavior can cause module context creation to fail through either deadlock or timeout.

Obtaining a ServiceReference object

If the property into which a reference component is to be injected has type `ServiceReference` (instead of the service interface supported by the reference), then an OSGi `ServiceReference for the service, as provided by the OSGi Service Platform in which the application is running,` will be injected in place of the service itself.

The injected service reference refers to the service instance satisfying the reference at the time the reference is injected. The ServiceReference object will not be updated if the backing service later changes. If there is no matching service instance at the time of injection (for example, the reference is to an optional service), then 'null' will be injected.

For example, given the following Java class declaration and component declarations:

```
public class ComponentWithServiceReference {
  private ServiceReference serviceReference;
  private SomeService service;
  // getters/setters ommitted
}
<reference id="service" interface="com.xyz.SomeService"/>
<component id="someComponent" class="ComponentWithServiceReference">
  <property name="serviceReference" ref="service"/>    (1)
  <property name="service" ref="service"/>             (2)
</component>
```

Then

(1) The `ServiceReference` object for the service obtained via the `reference` element will be injected into the `serviceReference` property.

(2) An object representing the service itself will be injected into the `service` property

5.4.2.2 Referencing a collection of services

Sometimes an application needs access not simply to any service meeting some criteria, but to all services meeting some criteria. The matching services may be held in a List or Set (optionally sorted).

The difference between using a List and a Set to manage the collection is one of equality. Two or more services published in the registry (and with distinct service ids) may be "equal" to each other, depending on the implementation of equals used by the service implementations. Only one such service will be present in a set, whereas all services returned from the registry will be present in a list. The `ref-set` and `ref-list` schema elements are used to define collections of services with set or list semantics respectively.

These elements support all of the attributes and nested elements defined for the `reference` element. An availability value of `optional` indicates that it is permissible for there to be no matching services. An availability value of `mandatory` indicates that at least one matching service is required at all times. Such a reference is considered a *mandatory reference* and any exported services from the same bundle (service defined

components) that depend on a mandatory reference will automatically be unregistered when the reference becomes unsatisfied, and re-registered when the reference becomes satisfied again.

The component defined by a `ref-list` element is of type `java.util.List`. The component defined by a `ref-set` element is of type `java.util.Set`.

The following example defines a component of type List that will contain all registered services supporting the `EventListener` interface:

```
<ref-list id="myEventListeners" interface="com.xyz.EventListener"/>
```

The members of the collection defined by the component are managed dynamically. As matching services are registered and unregistered in the service registry, the collection membership will be kept up to date. Each member of the collection supports the service interfaces that the corresponding service was registered with and that are visible to the bundle.

Sorted collections are also supported. To specify that a collection should be sorted, the optional `ordering-basis` attribute of `ref-list` and `ref-set` can be specified. The `ordering-basis` attribute can take one of two values: `service` or `service-reference`. If `service` is specified then sorting is based on the service instances in the collection. If `service-reference` is specified then sorting is based on the corresponding `ServiceReference` objects for those services. A comparator to use for the sorting can be specified using either the `comparator-ref` attribute, or the nested `comparator` element. The `comparator-ref` attribute is used to refer to a named component implementing `java.util.Comparator`. The comparator will be passed either the service objects or the ServiceReference objects, according to the value of the ordering-basis attribute. The `comparator` element can be used to define an inline component. For example:

```
<ref-set id="myServices" interface="com.xyz.MyService"
  ordering-basis="service"
  comparator-ref="someComparator"/>


<ref-list id="myOtherServices" interface="com.xyz.OtherService"
  ordering-basis="service-reference">
  <comparator>
     <component class="MyOtherServiceComparator"/>
  </comparator>
</ref-list>
```

If no comparator is specified (neither the comparator element nor comparator-ref attribute are used) then sorting occurs based on natural ordering. If a comparator is specified, but no ordering-basis, then the comparator will be passed the service instances.

Obtaining Service Reference Objects

If the property into which a reference set or list is to be injected is of type Collection<ServiceReference> or a subtype thereof (e.g. List<ServiceReference>, Set<ServiceReference>), then the injection collection will contain the ServiceReference objects for the matching services rather than the service objects themselves.

To support JDK 1.4 and below where generic types are not available, the optional `member-type` attribute of `ref-set` and ref-list allows you to specify explicitly the desired type of collection member. Permissable values are `service-instance` (the default behaviour) and `service-reference` (in which case the collection will contain ServiceReference objects).

5.4.3 Dealing with service dynamics

The component defined by a `reference` element is unchanged throughout the lifetime of the module context (the object reference remains constant). However, the OSGi service that backs the reference may come and go at any time. For a mandatory service reference, creation of the module context will block until a matching service is available. For an optional service reference (optional availability), the reference component will be created immediately, regardless of whether or not there is currently a matching service.

When the service backing a reference component goes away, an attempt is made to replace the backing service with another service matching the reference criteria. An application may be notified of a change in backing service by registering a listener (see section 5.4.3.2). If no matching service is available, then the reference is said to be unsatisfied. An unsatisfied mandatory service causes any exported service (service component) that depends on it to be unregistered from the service registry until such time as the reference is satisfied again.

When an operation is invoked on an unsatisfied reference component (either optional or mandatory), the invocation blocks until either the reference becomes satisfied or a timeout expires (whichever comes first). If an EventAdmin service is present then an org/osgi/service/blueprint/context/WAITING event is broadcast as described in section 5.2.3. The default timeout for service invocations is 5 minutes. The optional `timeout` attribute of the reference element enables an alternate timeout value (in milliseconds) to be specified. If no matching service becomes available within the timeout period, an unchecked `ServiceUnavailableException` is thrown. Specifying a timeout value of zero or less means that there will be no timeout period, and an unsatisfied service invocation will fail immediately.

Consider the following simple example:

```
<reference id="service" interface="com.xyz.SomeService" availability="optional" />

<component id="someComponent" class="SomeComponent">

  <property name="service" ref="service"/>

</component>
```

When the module context is created, and no service supporting the SomeService interface type is available in the registry, then a component instance is instantiated for the `service` component, but no backing service is set for it. The `someComponent` component is then instantiated and injected with a reference to the `service` component. At some point later, a service is published in the service registry that supports the SomeService interface. This service is transparently set as the backing service behind the `service` component. If an invocation is made on the `service` component when it has no backing service available, the timeout rules described above would then come into play.

Note that a module context with mandatory service references will by default wait for all mandatory service references to be satisfied before the context is initially created, but once the context has been created, a mandatory service reference that becomes unsatisfied does *not* cause the whole context to be disposed. The rationale is that having all mandatory services satisfied should be the normal case, and hence bringing up a context in a partially satisfied state is undesirable. However, once the context is created, temporary absences of mandatory services are tolerated to allow for administration operations and continuous operation of as much of the system as possible. For a reference collection with mandatory availability, the reference is considered satisfied is there is at least one matching service in the collection.

The `timeout` attribute is not supported by the `ref-set` and `ref-list` elements.

While a `reference` component will try to find a replacement if the backing service is unregistered, a reference collection-based component will simply remove the service from the collection. The recommend way of traversing a collection is by using an `Iterator`. During iteration, all `Iterators` held by the user will be transparently updated so it is possible to safely traverse the collection while it is being modified. Moreover, the Iterators will reflect all the changes made to the collection, even if they occurred after the Iterators were created (that is, during the iteration). Consider a case where a collection shrinks significantly (for example a large number of OSGi services are shutdown) right after an iteration started. To avoid dealing with the resulting 'dead' service

references, iterators do not take collection snapshots but instead are updated on each service event so they reflect the latest collection state, no matter how fast or slow the iteration is.

It is important to note that a service update will only influence Iterator operations that are executed after the event occurred. Services already returned by the iterator will not be updated even if the backing service has been unregistered. If an operation is invoked on such a service that has been unregistered, a `ServiceUnavailableException` will be thrown.

The Iterator contract is guaranteed: the `next()` method always obeys the result of the previous `hasNext()` invocation. Within this contract, an implementation is free to add additional matching elements into the collection during iteration, and to remove as yet unseen elements from the collection during iteration. A client may therefore see the return value of repeated calls to `hasNext()` change over time: for example after returning false a new member may be added to the collection causing a subsequent invocation of hasNext() to return true. The next() method always obeys the result of the previous hasNext() invocation, so if hasNext() returns true there is guaranteed to be an available object on a call to next().T

Any elements added to the collection during iteration over a sorted collection will only be visible if the iterator has not already passed their sort point.

Collections of ServiceReferences are managed in the same way as collections of the service objects themselves (i.e. ServiceReference objects may be added and removed dynamically so long as the Iterator contract is honored).

5.4.3.1 Mandatory dependencies

An exported service may depend, either directly or indirectly, on other services in order to perform its function. If one of these services is considered a mandatory dependency (has ‘mandatory’ availability) and the dependency can no longer be satisfied (because the backing service has gone away and there is no suitable replacement available) then the exported service that depends on it will be automatically unregistered from the service registry - meaning that it is no longer available to clients. If the mandatory dependency becomes satisfied once more (by registration of a suitable service), then the exported service will be re-registered in the service registry.

This automatic unregistering and re-registering of exported services based on the availability of mandatory dependencies only takes into account declarative dependencies. If exported service S depends on component A, which in turn depends on mandatory imported service M, and these dependencies are explicit in the module configuration file as per the example below, then when M becomes unsatisfied S will be unregistered. When M becomes satisfied again, S will be re-registered.

```
<service id="S" ref="A" interface="SomeInterface"/>


<component id="A" class="SomeImplementation">
   <property name="helperService" ref="M"/>
</component>


<reference id="M" interface="HelperService"
     availability="mandatory"/>
```

If however the dependency from A on M is not established through configuration as shown above, but instead at runtime through for example storing a reference to M in a field belonging to A without any involvement from the container, then this dependency is not tracked.

Automatic service registration and unregistration also applies to lazily registered services that have been registered on behalf of a bundle with a LAZY_ACTIVATION policy.

All Page Within This Box

5.4.3.2 Service Listeners

Applications that need to be aware of when a service backing a reference component is bound and unbound, or when a member is added to or removed from a collection, can register one or more listeners using the nested listener element. The `listener` element refers to a component (either by name using the `ref` attribute or nested `ref` element, or by defining one inline) that will receive bind and unbind notifications. The bind-method and unbind-method attributes indicate the operations to be invoked on the listener component during a bind or unbind event respectively.

For example:

```
<reference id="someService" interface="com.xyz.MessageService">
  <listener bind-method="onBind" unbind-method="onUnbind">
     <component class="MyCustomListener"/>
  </listener>
</reference>
```

The signature of a custom bind or unbind method must be one of:

```
public void anyMethodName(ServiceType service, Map properties);
public void anyMethodName(ServiceReference ref);
```

where `ServiceType` can be any type. The bind and unbind callbacks are invoked only if the `service` instance is assignable to a reference of type `ServiceType`. The `properties` parameter contains the set of properties that the service was registered with.

If the method signature has a single argument of type `ServiceReference` then the `ServiceReference` of the service will be passed to the callback in place of the service object itself.

When the listener is used with a `reference` declaration:

- A bind callback is invoked when the reference is initially bound to a backing service, and whenever the backing service is replaced by a new backing service.
- An unbind callback is only invoked when the current backing service is unregistered, and no replacement service is immediately available (i.e., the reference becomes unsatisfied).

When the listener is used with a collection declaration (set or list):

- A bind callback is invoked when a new service is added to the collection.
- An unbind callback is invoked when a service is unregistered and is removed from the collection.

Bind and unbind callbacks are made synchronously as part of processing an OSGi serviceChanged event for the backing OSGi service, and are invoked on the OSGi thread that delivers the corresponding OSGi ServiceEvent. In particular, the service or ServiceReference passed to a bind or unbind callback is valid for the lifetime of the callback. For collection based callbacks, the service is guaranteed to be available in the collection before a bind callback is invoked, and to remain in the collection until after an unbind callback has completed.

If a listener component defines multiple (overridden) methods with a name specified in a bind or unbind method attribute, then every method with a compatible signature (including both `ServiceType` and `ServiceReference` forms) is invoked on the corresponding event. The order of invocation of the matching methods is undefined.

The components refered to by listener elements (either via the ref attribute or by direct inline declaration of a component) are treated as dependencies of the enclosing service reference or collection component for the purposes of lifecycle management, dependency injection, and cycle detection. Given a reference component R, with a listener L, it must however be possible to inject a reference to R into L. For example:

All Page Within This Box

```
<reference id="R" interface="I">
  <listener bind-method="onBind"
            unbind-method="onUnbind"
            ref="L"/>
  </listener>
</reference>

<component id="L" class="C">
  <property name="observedService" ref="R"/>
</component>
```

5.4.3.3 Module-wide defaults for service references

The `components` element supports the setting of `default-availability` and `default-timeout` attribute values that then serve as the defaults for the `availability` attribute of the `reference`, `ref-set`, and `ref-list` elements, and the `timeout` attribute of the `reference` element respectively, when no value is specified.

5.5 Module Context API

5.6 Namespace Extension Mechanism

Third parties may contribute additional namespaces containing elements and attributes used to configure the components for a module context. These additional namespaces are referenced in the standard XML manner using the xmlns attribute of the top level element. The following configuration file references the osgi, osgix, and aop namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:osgix=http://www.osgi.org/xmlns/blueprint-compendium/v1.0.0
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
      http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
      http://www.osgi.org/xmlns/blueprint-compendium/v1.0.0 http://www.osgi.org/xmlns/blueprint-
compendium/v1.0.0/blueprint-compendium.xsd
      http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-
2.5.xsd">

…

</components>
```

In order to be able to interpret elements and attributes declared in third party namespaces, a *namespace handler* must be registered that the container can delegate to. Namespace handlers are registered in the OSGi service registry under the `org.osgi.service.blueprint.namespace.NamespaceHandler` interface. A registered namespace handler must also advertise the service property `osgi.service.blueprint.namespace.`The value of this property is an array of URIs for the schema that the namespace can handle (for example, http://www.springframework.org/schema/aop). When defining a namespace extension and schema the following conventions are recommended (but not required):

- A schema URI with a final path segment of the format “v[version]” – for example http://www.foobar.org/schema/foo/v1.0.0 references that version of the schema (1.0.0 in this case)

- A schema URI without such a final path segment references the latest version of the schema
- A namespace handler should publish in its osgi.service.blueprint.namespace property the URIs of each version of the schema it can handle, together with the unqualified URI if the handler can also handle the most recent version.

During the processing of a configuration file the container must determine the set of schemas used to define the elements in the configuration file. The resulting set of schema uris (for the above example this would be http://www.osgi.org/schema/service/blueprint, http://www.osgi.org/schema/service/blueprint-compendium, and http://www.springframework.org/schema/aop) determine the namespace handlers that must be present in order to process the file. For each schema uri the container creates a mandatory service reference for a service implementing the `NamespaceHandler` interface and with a matching `osgi.service.blueprint.namespace` property value. Creation of the context waits for these service references to be satisified in the same way as it would for any service reference explicitly declared via a `service` element – including generation of the wait events if an EventAsdmin service is available and the services are not immediately available.

The NamespaceHandler interface is defined as follows:

```
interface NamespaceHandler {
   /**
    * The URL where the xsd file for the schema may be found. Typically used to return a URL to a
    * bundle resource entry so as to avoid needing to lookup schemas remotely.
    * If null is returned then the schema location will be determined from the xsi:schemaLocation
attribute
    * value.
    */
   URL getSchemaLocation();

  /**
   * Called when a top-level (i.e. non-nested) element from the namespace is encountered.
   * Implementers may register component definitions themselves, and/or return a component definition
   * to be registered.
   */
   ComponentMetadata parse(org.w3c.dom.Element element, ParserContext context);

  /**
   * Called when an attribute or directly nested element is encountered. Implementors should parse the
   * supplied Node and decorated the provided component, returning the decorated component.
   */
   ComponentMetadata decorate(org.w3c.dom.Node node, ComponentMetadata component,
                              ParserContext context);
}
```

The `ParserContext` passed to the parse and decorate methods provides access to the `ComponentMetadata` of the enclosing component (if any) and to the `ComponentDefinitionRegistry`.

The `parse` callback is invoked when an element from the namespace is encountered that is not directly nested inside of a `component`, `service`, `reference`, `ref-list`, or `ref-set` element. Most commonly it is invoked

All Page Within This Box

when a top-level namespace element is encountered. Very often a custom namespace element serves as a convenient shortcut for creating a component definition. In this case where one namespace element corresponds to one component definition, the simplest implementation is often to return the new component definition as the return value of the parse callback. In more complex cases, a single namespace element may map to a collection of component definitions, or conditionally create components. In such circumstances the context passed to the parse method can be used to discover existing component definitions and to explicitly register new ones.

The `decorate` callback is invoked when an attribute from the handled namespace is encountered in an element that is not from the handled namespace. Decorate typically modifies the component definition for the component in which the attribute was encountered, and returns the modified definition. The `decorate` callback is also invoked when an element from the handled namespace is encountered nestly directly inside a `component`, `service`, `reference`, `ref-list`, or `ref-set` element.

Consider the following configuration snippet involving a hypothetical “cache” namespace:

```
<cache:lru-cache id="myCache"/>                          (1)

<component id="fooService" class="FooServiceImpl"
       cache:cache-return-values="true">                 (2)

    <cache:exclude>                                      (3)
      <cache:operation name="getVolatile"/>              (4)
    </cache:exclude>

    <property name="myProp" value="12"/>
</component>


<component id="barService" class="BarServiceImpl">
  <property name="localCache">
    <cache:lru-cache/>                                   (5)
  </property>
</component>
```

Given a namespace handler registered to handle the cache namespace, then at:

(1) The `parse` method will be invoked, and will typically return a new component definition defining a cache component.

(2) The `decorate` method will be invoked passing in the `cache-return-values` attribute `Node`, and the `ComponentMetadata` for the `fooService` component.

(3) The `decorate` method will be invoked passing in the `exclude` element `Node`, and the `ComponentMetadata` for the `fooService` component.

(4) There is no callback for this line, it is the responsibility of the `decorate` method at (3) to process the contents of the `exclude` element

(5) The `parse` method will be invoked and would typically return a new component definition defining a cache component. From the `ParserContext` passed to the method, `getEnclosingComponent` will return the component definition for the `barService` component.

5.6.1 Date Namespace Example

The following example illustrates how the Blueprint Service could be extended to process elements from a "date" namespace. In our example, the date namespace contains only one element, `dateformat`. The namespace handler will enable component configuration files to include declarations such as:

```
<date:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

and such declarations will in effect be equivalent to the following component definition:

```
<component id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>

</component>
```

The basic steps to implement the handler are as follows:

1. Create the schema file for the extension, and package it inside a bundle
2. Write a NamespaceHandler implementation and fill in the parse method
3. Export the namespace handler as a service in the service registry

5.6.1.1 Creating and packaging the schema file

The namespace handler will be packaged inside its own bundle. Inside the bundle, at location schemas/date.xsd[4] is placed the following file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.com/schema/myns"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:osgi="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    targetNamespace="http://www.mycompany.com/schema/date/v1.0.0"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

   <xsd:import namespace="http://www.osgi.org/xmlns/blueprint/v1.0.0"/>

   <xsd:element name="dateformat">
      <xsd:complexType>
         <xsd:complexContent>
            <xsd:extension base="osgi:identifiedType">
               <xsd:attribute name="lenient" type="xsd:boolean"/>
               <xsd:attribute name="pattern" type="xsd:string" use="required"/>
            </xsd:extension>
         </xsd:complexContent>
      </xsd:complexType>
   </xsd:element>


</xsd:schema>
```

The bold line shows that our new `dateformat` element extends the basic `identifiedType` element from the osgi blueprint schema, meaning that it will have an `id` attribute.

[4] The location given here is just for illustration, you can place the file anywhere inside the bundle that you choose

All Page Within This Box

5.6.1.2 NamespaceHandler implementation

The complete source code for the DateNamespaceHandler implementation is shown below:

```
package org.osgi.service.blueprint.namespace.example;

import java.net.URL;
import java.text.SimpleDateFormat;

import org.osgi.framework.BundleContext;
import org.osgi.service.blueprint.namespace.NamespaceHandler;
import org.osgi.service.blueprint.namespace.ParserContext;
import org.osgi.service.blueprint.namespace.example.builder.MutableLocalComponentMetadata;
import org.osgi.service.blueprint.reflect.ComponentMetadata;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

/**
 * Sample handler for a "date" namespace based on the example given here:
 * http://static.springframework.org/spring/docs/2.5.x/reference/extensible-xml.html
 *
 * Handles "dateformat" elements with the following form:
 *
 * <date:dateformat id="dateFormat"
 *   pattern="yyyy-MM-dd HH:mm"
 *   lenient="true"/>
 *
 */
public class DateNamespaceHandler implements NamespaceHandler {

            /**
             * The schema file is packaged in the same bundle as this handler class
             * for convenience, in schemas/date.xsd
             */
            private static final String SCHEMA_LOCATION = "schemas/date.xsd";

            private final BundleContext bundleContext;

            public DateNamespaceHandler(BundleContext context) {
                        this.bundleContext = context;
            }

            /**
             * Use the bundleContext to return a URL for accessing the schema definition file
             */
            public URL getSchemaLocation(String namespace)  throws IllegalArgumentException {
                        return bundleContext.getBundle().getResource(SCHEMA_LOCATION);
            }

            /**
             * Date elements can never be used nested directly inside a component, so
             * nothing for us to do.
             */
            public ComponentMetadata decorate(Node node, ComponentMetadata component,
                                    ParserContext context) {
                        return null;
            }


            /**
             * Handle the "dateformat" tag (and others in time...)
```

All Page Within This Box

```
     */
    public ComponentMetadata parse(Element element, ParserContext context) {

      if (element.getLocalName().equals("dateformat")) {
      return parseDateFormat(element);
      }
      else {
      throw new IllegalStateException("Asked to parse unknown tag: " +
                element.getTagName());
      }
    }

    /**
     * A dataformat element defines a component, which could be a top-level named
     * component or an anonymous inner component.
     */
    private ComponentMetadata parseDateFormat(Element element) {
      String name = element.hasAttribute("id") ? element.getAttribute("id") : "";
      MutableLocalComponentMetadata componentMetadata =
        new MutableLocalComponentMetadata(name,SimpleDateFormat.class.getName());

      // required attribute pattern
      String pattern = element.getAttribute("pattern");
      componentMetadata.addConstructorArg(pattern,0);

      // this however is an optional property
      String lenient = element.getAttribute("lenient");
      if ((lenient != null) && !lenient.equals("")) {
         componentMetadata.addProperty("lenient", lenient);
      }

      return componentMetadata;
    }

}
```

This implementation relies on a helper class, MutableLocalComponentMetadata that we use to create an instance of the LocalComponentMetadata interface. Future versions of this specification may standardize such implementation classes, or an equivalent builder API.

```
package org.osgi.service.blueprint.namespace.example.builder;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.osgi.service.blueprint.reflect.ComponentMetadata;
import org.osgi.service.blueprint.reflect.ConstructorInjectionMetadata;
import org.osgi.service.blueprint.reflect.LocalComponentMetadata;
import org.osgi.service.blueprint.reflect.MethodInjectionMetadata;
import org.osgi.service.blueprint.reflect.ParameterSpecification;
import org.osgi.service.blueprint.reflect.PropertyInjectionMetadata;
import org.osgi.service.blueprint.reflect.TypedStringValue;
import org.osgi.service.blueprint.reflect.Value;

/**
 * "Just enough" implementation of a mutable LocalComponentMetadata to meet the
 * needs of our namespace handler...
 */
public class MutableLocalComponentMetadata implements LocalComponentMetadata {

    private String name;
```

```
private String className;
private String initMethodName = "";
private String destroyMethodName = "";
private ComponentMetadata factoryComponent = null;
private final List<ParameterSpecification> constructorSpec =
   new ArrayList<ParameterSpecification>();
private final List<PropertyInjectionMetadata> propertiesSpec =
   new ArrayList<PropertyInjectionMetadata>();;
private MethodInjectionMetadata factoryMethodMetadata = null;
private String scope = "singleton";
private boolean isLazy = false;
private Set<String> dependencies = new HashSet<String>();

public MutableLocalComponentMetadata(String name, String className) {
            this.name = name;
            this.className = className;
}

public String getName() {
            return name;
}

public void setName(String name) {
            this.name = name;
}

public String getClassName() {
            return className;
}

public void setClassName(String className) {
            this.className = className;
}

public ConstructorInjectionMetadata getConstructorInjectionMetadata() {
            return new ConstructorInjectionMetadata() {

                        public List getParameterSpecifications() {
                                    return constructorSpec;
                        }

            };
}

public void addConstructorArg(ParameterSpecification spec) {
            constructorSpec.add(spec);
}

// convenience method for String-based values
public void addConstructorArg(final String value , final int index) {
            constructorSpec.add(new ParameterSpecification() {

                        public int getIndex() {
                                    return index;
                        }

                        public String getTypeName() {
                                    return null;
                        }

                        public Value getValue() {
                                    return stringValue(value);
                        }

            });
}

public Collection getPropertyInjectionMetadata() {
```

```
                return propertiesSpec;
        }

        public void addProperty(final String name, final Value value) {
                propertiesSpec.add(new PropertyInjectionMetadata() {

                        public String getName() {
                                return name;
                        }

                        public Value getValue() {
                                return value;
                        }
                });
        }

        public void addProperty(final String name, final String value) {
                propertiesSpec.add(new PropertyInjectionMetadata() {

                        public String getName() {
                                return name;
                        }

                        public Value getValue() {
                                return stringValue(value);
                        }
                });
        }

        public String getInitMethodName() {
                return initMethodName;
        }

        public void setInitMethodName(String initMethodName) {
                this.initMethodName = initMethodName;
        }

        public String getDestroyMethodName() {
                return destroyMethodName;
        }

        public void setDestroyMethodName(String destroyMethodName) {
                this.destroyMethodName = destroyMethodName;
        }

        public ComponentMetadata getFactoryComponent() {
                return factoryComponent;
        }

        public void setFactoryComponent(ComponentMetadata factoryComponent) {
                this.factoryComponent = factoryComponent;
        }

        public MethodInjectionMetadata getFactoryMethodMetadata() {
                return factoryMethodMetadata;
        }

        public void setFactoryMethodMetadata(
                        MethodInjectionMetadata factoryMethodMetadata) {
                this.factoryMethodMetadata = factoryMethodMetadata;
        }

        public String getScope() {
                return scope;
        }

        public void setScope(String scope) {
                this.scope = scope;
```

```
        }

        public boolean isLazy() {
                return isLazy;
        }

        public void setLazy(boolean lazy) {
                this.isLazy = lazy;
        }

        public Set getExplicitDependencies() {
                return dependencies;
        }

        public void addDependency(String name) {
                dependencies.add(name);
        }

        public void removeDependency(String name) {
                dependencies.remove(name);
        }

        private TypedStringValue stringValue(final String val) {
                return new TypedStringValue() {

                        public String getStringValue() {
                                return val;
                        }

                        public String getTypeName() {
                                return null;
                        }};
        }
}
```

Such an implementation is easily generated using a modern IDE.

5.6.1.3 Registering the namespace handler

With the DateNamespaceHandler and MutableLocalComponentMetadata classes packaged up in the bundle classpath, the last thing to do is ensure that the namespace handler is registered as a service when the bundle is started. The easiest way to do this is to use the blueprint service. In `OSGI-INF/blueprint` create a file called e.g. `module-context.xml` with the following declarations:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.osgi.org/xmlns/blueprint/v1.0.0
      http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

  <component id="dateNamespaceHandler"
          class="org.osgi.service.blueprint.namespace.example.DateNamespaceHandler"/>


  <service ref="dateNamespaceHandler"
     interface="org.osgi.service.blueprint.namespace.NamespaceHandler">
    <service-properties>
      <entry key="osgi.service.blueprint.namespace">
        <array value-type="java.lang.String">
            <value>http://www.mycompany.com/schema/date/v1.0.0</value>
            <value>http://www.mycompany.com/schema/date</value>
        </array>
```

All Page Within This Box

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

</components>
```

5.7 Configuration Administration Service Support

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.

5.7.1 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.

5.7.2 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.

5.7.3 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;
```

All Page Within This Box

```
  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.

5.7.4 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.

All Page Within This Box

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>
```

All Page Within This Box

```
</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.

5.7.5 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.

5.8 APIs

- ConstructorInjectionMetadata
- ListValue
- LocalComponentMetadata
- MapValue
- MethodInjectionMetadata
- NullValue
- ParameterSpecification
- PropertiesValue
- PropertyInjectionMetadata
- ReferenceNameValue
- ReferenceValue
- RegistrationListenerMetadata
- ServiceExportComponentMetadata
- ServiceReferenceComponentMetadata
- SetValue
- TypedStringValue
- UnaryServiceReferenceComponentMetadata
- Value
- Constant Field Values
- Serialized Form

Packages	
org.osgi.service.blueprint.context	Blueprint Service Context Package Version 1.0.
org.osgi.service.blueprint.convert	Blueprint Service Type Conversion Package Version 1.0.
org.osgi.service.blueprint.namespace	Blueprint Namespace Package Version 1.0.
org.osgi.service.blueprint.reflect	Blueprint Reflection Package Version 1.0.

5.9 Package org.osgi.service.blueprint.context

Blueprint Service Context Package Version 1.0.

5.9.1.1 See:

Description

Interface Summary	
ModuleContext	ModuleContext providing access to the components, service exports, and service references of a module.
ModuleContextEventConstants	Event property names used in EventAdmin events published for a module context.
ModuleContextListener	

Exception Summary	
ComponentDefinitionException	Exception thrown when a configuration-related error occurs during creation of a module context.
NoSuchComponentException	Thrown when an attempt is made to lookup a component by name and no such named component exists in the module context.
ServiceUnavailableException	Thrown when an invocation is made on an OSGi service reference component, and a backing service is not available.

5.10 Package org.osgi.service.blueprint.context Description

Blueprint Service Context Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package:                    org.osgi.service.blueprint.context;
version="[1.0,2.0)"
```

This package defines the primary interface to a module context, `ModuleContext`. An instance of this type is available inside a module context as an implicitly defined component with name "moduleContext".

This package also declares the supporting exception types, listener, and constants for working with a module context.

5.11 Package org.osgi.service.blueprint.convert

Blueprint Service Type Conversion Package Version 1.0.

All Page Within This Box

5.11.1.1 See:
Description

Interface Summary	
ConversionService	Provides access to the type conversions (both predefined and user registered) that are defined for the module context
Converter	Implemented by type converters that extend the type conversion capabilties of a module context container.

5.12 Package org.osgi.service.blueprint.convert Description

Blueprint Service Type Conversion Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.blueprint.convert;
version="[1.0,2.0)"
```

This package defines the `Converter` interface used to implement type converters, and the `ConversionService` interface that provides access to registered type converters. A module context contains an implicitly defined component "conversionService" that is an instance of `ConversionService`.

5.13 Package org.osgi.service.blueprint.namespace

Blueprint Namespace Package Version 1.0.

5.13.1.1 See:
Description

All Page Within This Box

Interface Summary	
ComponentDefinitionRegistry	A registry of the component definitions for a given context.
NamespaceHandler	A namespace handler provides support for parsing custom namespace elements and attributes in module context configuration files.
ParserContext	A ParserContext provides contextual information to a NamespaceHandler when parsing an Element or Node from the namespace.

Exception Summary	
ComponentNameAlreadyInUseException	Exception thrown when an attempt is made to register a component with a name that is already in use by an existing component.

5.14 Package org.osgi.service.blueprint.namespace Description

Blueprint Namespace Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.blueprint.namespace;
version="[1.0,2.0)"
```

This package provides the top-level interfaces needed for implementing a namespace handler.

5.15 Package org.osgi.service.blueprint.reflect

Blueprint Reflection Package Version 1.0.

5.15.1.1 See:

Description

Interface Summary	
ArrayValue	An array-based value.
BindingListenerMetadata	Metadata for a listener interested in service bind and unbind events for a service reference.
CollectionBasedServiceReferenceComponentMetadata	Service reference that binds to a collection of matching services from the OSGi service registry.
ComponentMetadata	Metadata for a component defined within a given module context.
ComponentValue	A value represented by an anonymous local component definition - this could be a component, reference, reference-collection or service definition.
ConstructorInjectionMetadata	Metadata describing how to instantiate a component instance by invoking one of its constructors.
ListValue	A list-based value.
LocalComponentMetadata	Metadata for a component defined locally with a module context.
MapValue	A map-based value.
MethodInjectionMetadata	Metadata describing a method to be invoked as part of component configuration.
NullValue	A value specified to be null via the element.
ParameterSpecification	Metadata describing a parameter of a method or constructor and the value that is to be passed during injection.

PropertiesValue	A java.util.Properties based value
PropertyInjectionMetadata	Metadata describing a property to be injected.
ReferenceNameValue	A value which represents the name of another component in the module context.
ReferenceValue	A value which refers to another component in the module context by name.
RegistrationListenerMetadata	Metadata for a listener interested in service registration and unregistration events for an exported service.
ServiceExportComponentMetadata	Metadata representing a service to be exported by a module context.
ServiceReferenceComponentMetadata	Metadata describing a reference to a service that is to be imported into the module context from the OSGi service registry.
SetValue	A set-based value.
TypedStringValue	A simple string value that will be type-converted if necessary before injecting into a target.
UnaryServiceReferenceComponentMetadata	Service reference that will bind to a single matching service in the service registry.
Value	A value to inject into a field, property, method argument or constructor argument.

5.16 Package org.osgi.service.blueprint.reflect Description

Blueprint Reflection Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package:                    org.osgi.service.blueprint.reflect;
version="[1.0,2.0)"
```

This package provides a reflection-based view of the configuration information for a given module context. The top-level mapping between configuration elements and reflection types is as follows:

All Page Within This Box

LocalComponentMetadata, ComponentValue

Configuration Element

component reference ref-set,ref-list service constructor-arg property listener registration-listener array ref id-ref list map null props set value

Corresponding Reflection Type(s)

UnaryServiceReferenceComponentMetadata CollectionBasedServiceReferenceComponentMetadata ServiceExportComponentMetadata ConstructorInjectionMetadata, ParameterSpecification PropertyInjectionMetadata BindingListenerMetadata RegistrationListenerMetadata ArrayValue ReferenceValue ReferenceNameValue ListValue MapValue NullValue PropertiesValue SetValue TypedStringValue

---

org.osgi.service.blueprint.context

5.17 Interface ModuleContext

`public interface ModuleContext`

ModuleContext providing access to the components, service exports, and service references of a module. Only bundles in the ACTIVE state may have an associated ModuleContext. A given BundleContext has at most one associated ModuleContext. An instance of ModuleContext may be obtained from within a module context by implementing the ModuleContextAware interface on a component class. Alternatively you can look up ModuleContext services in the service registry. The Constants. BUNDLE_SYMBOLICNAME and Constants.BUNDLE_VERSION service properties can be used to determine which bundle the published ModuleContext service is associated with. A ModuleContext implementation must support safe concurrent access. It is legal for the set of named components and component metadata to change between invocations on the same thread if another thread is concurrently modifying the same mutable ModuleContext implementation object.

See Also:

`ModuleContextAware`, `Constants`

Field Summary	
static int	**BUNDLE_STOPPING** reason code for destroy method callback of a managed service factory created component, when the component is being disposed because the bundle is being stopped.
static int	**CONFIGURATION_ADMIN_OBJECT_DELETED** reason code for destroy method callback of a managed service factory created component, when the component is being disposed because the corresponding configuration admin object was deleted.

Method Summary	
org.osgi.framework.BundleContext	**getBundleContext**() Get the bundle context of the bundle this module context is associated with.
java.lang.Object	**getComponent**(java.lang.String name) Get the component instance for a given named component.
ComponentMetadata	**getComponentMetadata**(java.lang.String name) Get the component metadata for a given named component.
java.util.Set	**getComponentNames**() The set of component names recognized by the module context.
java.util.Collection	**getExportedServicesMetadata**() Get the service export metadata for every service exported by this module.
java.util.Collection	**getLocalComponentsMetadata**() Get the metadata for all components defined locally within this module.
java.util.Collection	**getReferencedServicesMetadata**() Get the service reference metadata for every OSGi service referenced by this module.

Field Detail

All Page Within This Box

5.17.1 CONFIGURATION_ADMIN_OBJECT_DELETED

5.17.2 `static final int` **`CONFIGURATION_ADMIN_OBJECT_DELETE`**D

reason code for destroy method callback of a managed service factory created component, when the component is being disposed because the corresponding configuration admin object was deleted.

See Also:

Constant Field Values

5.17.3 BUNDLE_STOPPING

`static final int` **`BUNDLE_STOPPIN`**G

reason code for destroy method callback of a managed service factory created component, when the component is being disposed because the bundle is being stopped.

See Also:

Constant Field Values

Method Detail

5.17.4 getComponentNames

5.17.5 `java.util.Set` **`getComponentNames`**`()`

The set of component names recognized by the module context.

Returns:

an immutable set (of Strings) containing the names of all of the components

within the module.

5.17.6 getComponent

`java.lang.Object` **`getComponent`**`(java.lang.String name)`

All Page Within This Box

Get the component instance for a given named component. If the component has not yet been instantiated, calling this operation will cause the component instance to be created and initialized. If the component has a prototype scope then each call to getComponent will return a new component instance. If the component has a bundle scope then the component instance returned will be the instance for the caller's bundle (and that instance will be instantiated if it has not already been created). Note: calling getComponent from logic executing during the instantiation and configuration of a component, before the init method (if specified) has returned, may trigger a circular dependency (for a trivial example, consider a component that looks itself up by name during its init method). Implementations of the Blueprint Service are not required to support cycles in the dependency graph and may throw an exception if a cycle is detected. Implementations that can support certain kinds of cycles are free to do so.

Parameters:

`name`- the name of the component for which the instance is to be retrieved.

Returns:

the component instance, the type of the returned object is dependent on the component definition, and may be determined by introspecting the component metadata.

Throws:

`NoSuchComponentException` - if the name specified is not the name of a component within the module.

5.17.7 getComponentMetadata

`ComponentMetadata` **`getComponentMetadata`**`(java.lang.String name)`

Get the component metadata for a given named component.

Parameters:

`name`- the name of the component for which the metadata is to be retrieved.

Returns:

the component metadata for the component.

Throws:

NoSuchComponentException - if the name specified is not the name of a component within the module.

5.17.8 getReferencedServicesMetadata

5.17.9 `java.util.Collection` **`getReferencedServicesMetadata`**`()`

Get the service reference metadata for every OSGi service referenced by this module.

Returns:

an immutable collection of ServiceReferenceComponentMetadata, with one entry for each referenced service.

5.17.10 getExportedServicesMetadata

5.17.11 `java.util.Collection` **`getExportedServicesMetadata`**`()`

Get the service export metadata for every service exported by this module.

Returns:

an immutable collection of ServiceExportComponentMetadata, with one entry for each service export.

5.17.12 getLocalComponentsMetadata

5.17.13 `java.util.Collection` **`getLocalComponentsMetadata`**`()`

Get the metadata for all components defined locally within this module.

Returns:

an immutable collection of LocalComponentMetadata, with one entry for each component.

5.17.14 getBundleContext

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

All Page Within This Box

Get the bundle context of the bundle this module context is associated with.

Returns:
the module's bundle context

---

org.osgi.service.blueprint.context

Interface ModuleContextEventConstants

`public interface` **`ModuleContextEventConstant`**`s`

Event property names used in EventAdmin events published for a module context.

Field Summary	
static java.lang.String	**BUNDLE_VERSION** The version property defining the bundle on whose behalf a module context event has been issued.
static java.lang.String	**EXTENDER_BUNDLE** The extender bundle property defining the extender bundle processing the module context for which an event has been issued.
static java.lang.String	**EXTENDER_ID** The extender bundle id property defining the id of the extender bundle processing the module context for which an event has been issued.
static java.lang.String	**EXTENDER_SYMBOLICNAME** The extender bundle symbolic name property defining the symbolic name of the extender bundle processing the module context for which an event has been issued.
static java.lang.String	**TOPIC_BLUEPRINT_EVENTS** Topic prefix for all events issued by the Blueprint Service

All Page Within This Box

static java.lang.String	**TOPIC_CREATED** Topic for Blueprint Service CREATED events
static java.lang.String	**TOPIC_CREATING** Topic for Blueprint Service CREATING events
static java.lang.String	**TOPIC_DESTROYED** Topic for Blueprint Service DESTROYED events
static java.lang.String	**TOPIC_DESTROYING** Topic for Blueprint Service DESTROYING events
static java.lang.String	**TOPIC_FAILURE** Topic for Blueprint Service FAILURE events
static java.lang.String	**TOPIC_WAITING** Topic for Blueprint Service WAITING events

Field Detail

BUNDLE_VERSION

```
static final java.lang.String BUNDLE_VERSION
```

The version property defining the bundle on whose behalf a module context event has been issued.

See Also:

`Version`, Constant Field Values

EXTENDER_BUNDLE

```
static final java.lang.String EXTENDER_BUNDLE
```

The extender bundle property defining the extender bundle processing the module context for which an event has been issued.

See Also:

All Page Within This Box

Bundle, Constant Field Values

EXTENDER_ID

```
static final java.lang.String EXTENDER_ID
```

The extender bundle id property defining the id of the extender bundle processing the module context for which an event has been issued.

See Also:
Constant Field Values

EXTENDER_SYMBOLICNAME

```
static final java.lang.String EXTENDER_SYMBOLICNAME
```

The extender bundle symbolic name property defining the symbolic name of the extender bundle processing the module context for which an event has been issued.

See Also:
Constant Field Values

TOPIC_BLUEPRINT_EVENTS

```
static final java.lang.String TOPIC_BLUEPRINT_EVENTS
```

Topic prefix for all events issued by the Blueprint Service

See Also:
Constant Field Values

TOPIC_CREATING

```
static final java.lang.String TOPIC_CREATING
```

Topic for Blueprint Service CREATING events

See Also:
Constant Field Values

TOPIC_CREATED

```
static final java.lang.String TOPIC_CREATED
```

Topic for Blueprint Service CREATED events

See Also:
Constant Field Values

TOPIC_DESTROYING

```
static final java.lang.String TOPIC_DESTROYING
```

Topic for Blueprint Service DESTROYING events

See Also:
Constant Field Values

TOPIC_DESTROYED

```
static final java.lang.String TOPIC_DESTROYED
```

Topic for Blueprint Service DESTROYED events

See Also:
Constant Field Values

TOPIC_WAITING

```
static final java.lang.String TOPIC_WAITING
```

Topic for Blueprint Service WAITING events

See Also:
Constant Field Values

TOPIC_FAILURE

```
static final java.lang.String TOPIC_FAILURE
```

Topic for Blueprint Service FAILURE events

See Also:
Constant Field Values

org.osgi.service.blueprint.context

5.18 Interface ModuleContextListener

```
public interface ModuleContextListener
```

Method Summary	
void	contextCreated(org.osgi.framework.Bundle forBundle)
void	contextCreationFailed(org.osgi.framework.Bundle forBundle, java.lang.Throwable rootCause)

Method Detail

5.18.1 contextCreated

```
void contextCreated(org.osgi.framework.Bundle forBundle)
```

5.18.2 contextCreationFailed

```
void            contextCreationFailed(org.osgi.framework.Bundle
                    forBundle,
                    java.lang.Throwable rootCause)
```

---

org.osgi.service.blueprint.context

5.18.3 Class ComponentDefinitionException

```
java.lang.Object
  └java.lang.Throwable
     └java.lang.Exception
        └java.lang.RuntimeException
           └
             org.osgi.service.blueprint.context.ComponentDefinitionException
```

5.18.3.1 *All Implemented Interfaces:*

java.io.Serializable

5.18.3.2 ***public class*** *ComponentDefinitionExceptio**n***

extends java.lang.RuntimeException

Exception thrown when a configuration-related error occurs during creation of a module context.

5.18.3.3 *See Also:*

Serialized Form

All Page Within This Box

Constructor Summary

`ComponentDefinitionException(java.lang.String explanation)`

Method Summary

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

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

Constructor Detail

5.18.3.4 ComponentDefinitionException

```
public ComponentDefinitionException(java.lang.String explanation)
```

org.osgi.service.blueprint.context

5.19 Class NoSuchComponentException

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └org.osgi.service.blueprint.context.NoSuchComponentException
```

5.19.1.1 All Implemented Interfaces:

java.io.Serializable

All Page Within This Box

5.19.1.2 public class NoSuchComponentException

extends java.lang.RuntimeException

Thrown when an attempt is made to lookup a component by name and no such named component exists in the module context.

5.19.1.3 See Also:

Serialized Form

Constructor Summary

NoSuchComponentException(java.lang.String componentName)

Method Summary

java.lang.String	getComponentName()
java.lang.String	getMessage()

Methods inherited from class java.lang.Throwable

fillInStackTrace, getCause, getLocalizedMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

5.19.2 NoSuchComponentException

public **NoSuchComponentException**(java.lang.String componentName)

All Page Within This Box

Method Detail

5.19.3 getComponentName

```
public java.lang.String getComponentName()
```

5.19.4 getMessage

```
public java.lang.String getMessage()
```

5.19.4.1 Overrides:

`getMessage` in class `java.lang.Throwable`

---

org.osgi.service.blueprint.context

5.19.5 Class ServiceUnavailableException

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └org.osgi.service.blueprint.context.ServiceUnavailableException
```

5.19.5.1 All Implemented Interfaces:

java.io.Serializable

5.19.5.2 ***public class*** *ServiceUnavailableException*

extends java.lang.RuntimeException

Thrown when an invocation is made on an OSGi service reference component, and a backing service is not available.

5.19.5.3 See Also:

Serialized Form

All Page Within This Box

Constructor Summary

`ServiceUnavailableException(java.lang.String message, java.lang.Class serviceType, java.lang.String filterExpression)`

Method Summary

`java.lang.String`	`getFilter()` The filter expression that a service would have needed to satisfy in order for the invocation to proceed.
`java.lang.Class`	`getServiceType()` The type of the service that would have needed to be available in order for the invocation to proceed.

Methods inherited from class java.lang.Throwable

`fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

5.19.5.4 ServiceUnavailableException

```
public ServiceUnavailableException(java.lang.String message,
                                   java.lang.Class serviceType,
                                   java.lang.String filterExpression)
```

Method Detail

5.19.5.5 getServiceType

```
public java.lang.Class getServiceType()
```

All Page Within This Box

The type of the service that would have needed to be available in order for the invocation to proceed.

5.19.5.6 getFilter

```
public java.lang.String getFilter()
```

The filter expression that a service would have needed to satisfy in order for the invocation to proceed.

---

org.osgi.service.blueprint.convert

Interface ConversionService

```
public interface ConversionService
```

Provides access to the type conversions (both predefined and user registered) that are defined for the module context

---

Method Summary

java.lang.Object	convert(java.lang.Object fromValue, java.lang.Class toType) Convert an object to an instance of the given class, using the built-in and user-registered type converters as necessary.

Method Detail

convert

```
java.lang.Object convert(java.lang.Object fromValue,
                         java.lang.Class toType)
                         throws java.lang.Exception
```

All Page Within This Box

Convert an object to an instance of the given class, using the built-in and user-registered type converters as necessary.

Parameters:
`fromValue`- the object to be converted `toType`- the type that the instance is to be converted to

Returns:
an instance of the class 'toType'

Throws: `java.lang.Exception`- if the conversion cannot succeed. This exception is checked because callers should expect that not all source objects can be successfully converted.

---

org.osgi.service.blueprint.convert

Interface Converter

`public interface` **`Converte`**r

Implemented by type converters that extend the type conversion capabilties of a module context container.

Method Summary	
java.lang.Object	**convert**(java.lang.Object source) Convert an object to an instance of the target class.
java.lang.Class	**getTargetClass**() The type that this converter converts String values into.

Method Detail

getTargetClass

```
java.lang.Class getTargetClass()
```

The type that this converter converts String values into.

Returns:

Class object for the class that this converter converts to

convert

```
java.lang.Object convert(java.lang.Object source)
                  throws java.lang.Exception
```

Convert an object to an instance of the target class.

Parameters:

`source`- the object to be converted

Returns:

an instance of the class returned by getTargetClass

Throws:

`java.lang.Exception`- if the conversion cannot succeed. This exception is checked because callers should expect that not all source objects can be successfully converted.

org.osgi.service.blueprint.namespace

5.20 Interface ComponentDefinitionRegistry

5.20.1 `public interface` **`ComponentDefinitionRegistry`**

A registry of the component definitions for a given context. Implementations of ComponentDefinitionRegistry are required to support concurrent access. The state of a component registry may change between invocations on the same thread. For example, a single thread invoking containsComponentDefinition("foo") and getting a return value of 'true' may see a return value of null on a subsequent call to getComponentDefinition("foo") if another thread has removed the component definition in the meantime.

Method Summary

boolean	**containsComponentDefinition**(java.lang.String name) Returns true iff the registry contains a component definition with the given name.
ComponentMetadata	**getComponentDefinition**(java.lang.String name) Get the component definition for the component with the given name.
java.util.Set	**getComponentDefinitionNames**() Get the names of all the registered components.
void	**registerComponentDefinition**(ComponentMetadata component) Register a new component definition.

All Page Within This Box

void	removeComponentDefinition(java.lang.String name) Remove a component definition from the registry.

Method Detail

5.20.2 containsComponentDefinition

boolean **containsComponentDefinition**(java.lang.String name)

Returns true iff the registry contains a component definition with the given name.

5.20.3 getComponentDefinition

ComponentMetadata **getComponentDefinition**(java.lang.String name)

Get the component definition for the component with the given name.

5.20.4 Returns:

the matching component definition if present, or null if no component with a matching name or alias is present.

5.20.5 getComponentDefinitionNames

java.util.Set **getComponentDefinitionNames**()

Get the names of all the registered components.

5.20.6 Returns:

an immutable set (of Strings) containing the names of all registered components.

All Page Within This Box

5.20.7 registerComponentDefinition

`void registerComponentDefinition(ComponentMetadata component)`

Register a new component definition.

5.20.8 Throws:

`ComponentNameAlreadyInUseException` - if the name of the component definition to be registered is already in use by an existing component definition.

5.20.9 removeComponentDefinition

`void removeComponentDefinition(java.lang.String name)`

Remove a component definition from the registry. If no matching component is present then this operation does nothing.

5.20.10 Parameters:

`name` - the name of the component to be removed.

---

org.osgi.service.blueprint.namespace

Interface NamespaceHandler

`public interface NamespaceHandler`

A namespace handler provides support for parsing custom namespace elements and attributes in module context configuration files. It manipulates component definitions and the component registry to implement the intended semantices of the namespace. Instances of NamespaceHandler are discovered through the service registry where they should be published with a service property org.osgi.module.context.namespace set to the schema URI of the schema that they handle. Implementations of NamespaceHandler are required to be thread-safe.

Method Summary	
ComponentMetadata	**decorate**(org.w3c.dom.Node node, ComponentMetadata component, ParserContext context) Called when an attribute or nested element is encountered.
java.net.URL	**getSchemaLocation**(java.lang.String namespace) Return the location of the schema for a given namespace.
ComponentMetadata	**parse**(org.w3c.dom.Element element, ParserContext context) Called when a top-level (i.e.

Method Detail

getSchemaLocation

`java.net.URL` **`getSchemaLocation`**`(java.lang.String namespace)`

Return the location of the schema for a given namespace.

Parameters:

`namespace`- one of the advertized URIs supported by this handler (as registered in the org.osgi.service.blueprint.namespace property of the service registration).

Returns:

The URL where the xsd file for the schema may be found. Typically used to return a URL to a bundle resource entry so as to avoid needing to lookup schemas remotely. If null is returned then the schema location will be determined from the xsi:schemaLocation attribute value.

Throws:

`java.lang.IllegalArgumentException`- if the namespace parameter is not a recognized namespace supported by this handler

parse

`ComponentMetadata` **`parse`**`(org.w3c.dom.Element element, ParserContext context)`

Called when a top-level (i.e. non-nested) element from the namespace is encountered. Implementers may register component definitions themselves, and/ or return a component definition to be registered.

Parameters:

`element` - the dom element from the namespace that has just been

encountered `context` - parser context giving access component registry and context information about the current parsing location.

Returns:

a component metadata instance to be registered for the context, or null if there are no additional component descriptions to register.

decorate

`ComponentMetadata` **`decorate`**`(org.w3c.dom.Node node, ComponentMetadata component, ParserContext context)`

Called when an attribute or nested element is encountered. Implementors should parse the supplied Node and decorate the provided component, returning the decorated component.

Parameters:

`node` - the dom Node from the namespace that has just been encountered `component` - the component metadata for the component in which the attribute or nested element was encountered `context` - parser context giving access component registry and context information about the current parsing location.

Returns:

the decorated component to replace the original, or simply the original component if no decoration is required.

---

org.osgi.service.blueprint.namespace

Interface ParserContext

`public interface ParserContext`

A ParserContext provides contextual information to a NamespaceHandler when parsing an Element or Node from the namespace.

Method Summary

ComponentDefinitionRegistry	**getComponentDefinitionRegistry**() The component definition registry containing all of the registered component definitions for this context
ComponentMetadata	**getEnclosingComponent**() The enclosing component definition in the context of which the source node is to be processed.
org.w3c.dom.Node	**getSourceNode**() The dom Node which we are currently processing

Method Detail

getSourceNode

`org.w3c.dom.Node` **`getSourceNode`**()

The dom Node which we are currently processing

---

getComponentDefinitionRegistry

`ComponentDefinitionRegistry` **`getComponentDefinitionRegistry`**()

The component definition registry containing all of the registered component definitions for this context

---

getEnclosingComponent

`ComponentMetadata` **`getEnclosingComponent`**()

The enclosing component definition in the context of which the source node is to be processed.

---

org.osgi.service.blueprint.namespace

5.21 Class ComponentNameAlreadyInUseException

```
java.lang.Object
  └java.lang.Throwable
     └java.lang.Exception
        └java.lang.RuntimeException
└org.osgi.service.blueprint.namespace.
ComponentNameAlreadyInUseException
```

5.21.1 All Implemented Interfaces:

java.io.Serializable

All Page Within This Box

5.21.2 public class **ComponentNameAlreadyInUseException**
extends java.lang.RuntimeException

Exception thrown when an attempt is made to register a component with a name that is already in use by an existing component.

5.21.3 See Also:
Serialized Form

Constructor Summary

ComponentNameAlreadyInUseException(java.lang.String name)

Method Summary

java.lang.String	getConflictingName()
java.lang.String	getMessage()

Methods inherited from class java.lang.Throwable

fillInStackTrace, getCause, getLocalizedMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

5.21.4 ComponentNameAlreadyInUseException

5.21.5 public **ComponentNameAlreadyInUseException**(java.lang.String name)

Method Detail

All Page Within This Box

5.21.6 getMessage

```
public java.lang.String getMessage()
```

5.21.7 Overrides:

getMessage in class java.lang.Throwable

5.21.8 getConflictingName

```
public java.lang.String getConflictingName()
```

---

org.osgi.service.blueprint.reflect

Interface ArrayValue

All Superinterfaces:

Value

```
public interface ArrayValue
```

extends Value

An array-based value. Members of the array are instances of Value.

Method Summary	
Value []	**getArray**() The array of Value objects
java.lang.String	**getValueType**() The value-type specified for the array

Method Detail

getValueType

java.lang.String **getValueType**()

The value-type specified for the array

getArray

Value[] **getArray**()

The array of Value objects

org.osgi.service.blueprint.reflect

Interface BindingListenerMetadata

public interface **BindingListenerMetadata**

Metadata for a listener interested in service bind and unbind events for a service reference.

All Page Within This Box

Method Summary	
java.lang.String	**getBindMethodName**() The name of the method to invoke on the listener component when a matching service is bound to the reference
Value	**getListenerComponent**() The component instance that will receive bind and unbind events.
java.lang.String	**getUnbindMethodName**() The name of the method to invoke on the listener component when a service is unbound from the reference.

Method Detail

getListenerComponent

Value **getListenerComponent**()

The component instance that will receive bind and unbind events. The returned value must reference a component and therefore be either a ComponentValue, ReferenceValue, or ReferenceNameValue.

Returns:

the listener component reference.

getBindMethodName

java.lang.String **getBindMethodName**()

The name of the method to invoke on the listener component when a matching service is bound to the reference

Returns:

the bind callback method name.

All Page Within This Box

getUnbindMethodName

`java.lang.String` **`getUnbindMethodName`**()

The name of the method to invoke on the listener component when a service is unbound from the reference.

Returns:
the unbind callback method name.

5.21.8.1 org.osgi.service.blueprint.reflect

Interface CollectionBasedServiceReferenceComponentMetadata

5.21.9 All Superinterfaces:
ComponentMetadata , ServiceReferenceComponentMetadata

5.21.10 `public interface`

`CollectionBasedServiceReferenceComponentMetadat`a

extends ServiceReferenceComponentMetadata

Service reference that binds to a collection of matching services from the OSGi service registry.

Field Summary	
static int	**MEMBER_TYPE_SERVICE_REFERENCES** Collection contains service references
static int	**MEMBER_TYPE_SERVICES** Collection contains service instances
static int	**ORDER_BASIS_SERVICE_REFERENCES** Create ordering based on comparison of service reference objects.
static int	**ORDER_BASIS_SERVICES** Create ordering based on comparison of service objects.

Fields inherited from interface org.osgi.service.blueprint.reflect. ServiceReferenceComponentMetadata
MANDATORY_AVAILABILITY, OPTIONAL_AVAILABILITY

All Page Within This Box

Method Summary	
java.lang.Class	getCollectionType() The type of collection to be created.
Value	getComparator() The comparator specified for ordering the collection, or null if no comparator was specified.
int	getMemberType() Whether the collection will contain service instances, or service references
int	getOrderingComparisonBasis() The basis on which to perform ordering, if specified.

Methods inherited from interface org.osgi.service.blueprint.reflect. ServiceReferenceComponentMetadata
getBindingListeners, getComponentName, getFilter, getInterfaceNames, getServiceAvailabilitySpecification

Methods inherited from interface org.osgi.service.blueprint.reflect.ComponentMetadata
getExplicitDependencies, getName

Field Detail

5.21.11 ORDER_BASIS_SERVICES

```
static final int ORDER_BASIS_SERVICES
```

Create ordering based on comparison of service objects.

5.21.12 See Also:

Constant Field Values

All Page Within This Box

5.21.13 ORDER_BASIS_SERVICE_REFERENCES

```
static final int ORDER_BASIS_SERVICE_REFERENCES
```

Create ordering based on comparison of service reference objects.

5.21.14 See Also:

Constant Field Values

5.21.15 MEMBER_TYPE_SERVICES

```
static final int MEMBER_TYPE_SERVICES
```

Collection contains service instances

5.21.16 See Also:

Constant Field Values

5.21.17 MEMBER_TYPE_SERVICE_REFERENCES

```
static final int MEMBER_TYPE_SERVICE_REFERENCES
```

Collection contains service references

5.21.18 See Also:

Constant Field Values

Method Detail

5.21.19 getCollectionType

```
java.lang.Class getCollectionType()
```

The type of collection to be created.

5.21.20 Returns:

Class object for the specified collection type (List, Set).

All Page Within This Box

5.21.21 getComparator

`Value getComparator()`

The comparator specified for ordering the collection, or null if no comparator was specified.

5.21.22 Returns:

if a comparator was specified then a Value object identifying the comparator (a ComponentValue, ReferenceValue, or ReferenceNameValue) is returned. If no comparator was specified then null will be returned.

5.21.23 getOrderingComparisonBasis

`int getOrderingComparisonBasis()`

The basis on which to perform ordering, if specified.

5.21.24 Returns:

one of ORDER_BASIS_SERVICES and ORDER_BASIS_SERVICE_REFERENCES

5.21.25 getMemberType

`int getMemberType()`

Whether the collection will contain service instances, or service references

---

---

org.osgi.service.blueprint.reflect

Interface ComponentMetadata

All Known Subinterfaces:

CollectionBasedServiceReferenceComponentMetadata, LocalComponentMetadata, ServiceExportComponentMetadata, ServiceReferenceComponentMetadata, UnaryServiceReferenceComponentMetadata

---

All Page Within This Box

`public interface` **`ComponentMetadata`**

Metadata for a component defined within a given module context.

See Also:

`LocalComponentMetadata`, `ServiceReferenceComponentMetadata`, `ServiceExportComponentMetadata`

Method Summary

`java.util.Set`	**`getExplicitDependencies`**`()` The names of any components listed in a "depends-on" attribute for this component.
`java.lang.String`	**`getName`**`()` The name of the component.

Method Detail

getName

`java.lang.String` **`getName`**`()`

The name of the component.

Returns:

component name. The component name may be null if this is an anonymously defined inner component.

---

getExplicitDependencies

`java.util.Set` **`getExplicitDependencies`**`()`

All Page Within This Box

The names of any components listed in a "depends-on" attribute for this component.

Returns:

an immutable set of component names for components that we have explicitly declared a dependency on, or an empty set if none.

---

org.osgi.service.blueprint.reflect

Interface ComponentValue

All Superinterfaces:

Value

`public interface ComponentValue`

extends Value

A value represented by an anonymous local component definition - this could be a component, reference, reference-collection or service definition.

Method Summary

ComponentMetadata	**getComponentMetadata**()

Method Detail

getComponentMetadata

ComponentMetadata **getComponentMetadata**()

All Page Within This Box

org.osgi.service.blueprint.reflect

Interface ConstructorInjectionMetadata

public interface **ConstructorInjectionMetadata**

Metadata describing how to instantiate a component instance by invoking one of its constructors.

Method Summary

java.util.List	**getParameterSpecifications**() The parameter specifications that determine which constructor to invoke and what arguments to pass to it.

Method Detail

getParameterSpecifications

java.util.List **getParameterSpecifications**()

The parameter specifications that determine which constructor to invoke and what arguments to pass to it.

Returns:

an immutable list of ParameterSpecification, or an empty list if the default constructor is to be invoked. The list is ordered by ascending parameter index. I.e., the first parameter is first in the list, and so on.

org.osgi.service.blueprint.reflect

Interface ListValue

All Page Within This Box

All Superinterfaces:
Value

`public interface ListValue`

extends Value

A list-based value. Members of the List are instances of Value.

Method Summary	
java.util.List	**getList**() The List (of Value objects) for this List-based value
java.lang.String	**getValueType**() The value-type specified for the list elements, or null if none given

Method Detail

getValueType

`java.lang.String getValueType()`

The value-type specified for the list elements, or null if none given

---

getList

`java.util.List getList()`

The List (of Value objects) for this List-based value

org.osgi.service.blueprint.reflect

5.22 Interface LocalComponentMetadata

All Superinterfaces:

ComponentMetadata

5.22.1 public interface **LocalComponentMetadata**

extends ComponentMetadata

Metadata for a component defined locally with a module context.

Field Summary	
static java.lang.String	**SCOPE_BUNDLE**
static java.lang.String	**SCOPE_PROTOTYPE**
static java.lang.String	**SCOPE_SINGLETON**

Method Summary	
java.lang.String	**getClassName**() The name of the class type specified for this component.

All Page Within This Box

ConstructorInjectionMetadata	**getConstructorInjectionMetadata**() The constructor injection metadata for this component.
java.lang.String	**getDestroyMethodName**() The name of the destroy method specified for this component, if any.
Value	**getFactoryComponent**() The component instance on which to invoke the factory method (if specified).
MethodInjectionMetadata	**getFactoryMethodMetadata**() The metadata describing how to create the component instance by invoking a method (as opposed to a constructor) if factory methods are used.
java.lang.String	**getInitMethodName**() The name of the init method specified for this component, if any.
java.util.Collection	**getPropertyInjectionMetadata**() The property injection metadata for this component.
java.lang.String	**getScope**() The specified scope for the component lifecycle.
boolean	**isLazy**() Is this component to be lazily instantiated?

Methods inherited from interface org.osgi.service.blueprint.reflect. ComponentMetadata
getExplicitDependencies, getName

Field Detail

5.22.2 SCOPE_SINGLETON

```
static final java.lang.String SCOPE_SINGLETON
```

See Also:

Constant Field Values

5.22.3 SCOPE_PROTOTYPE

```
static final java.lang.String SCOPE_PROTOTYPE
```

See Also:

Constant Field Values

5.22.4 SCOPE_BUNDLE

```
static final java.lang.String SCOPE_BUNDLE
```

See Also:

Constant Field Values

Method Detail

5.22.5 getClassName

```
java.lang.String getClassName()
```

The name of the class type specified for this component.

Returns:

the name of the component class. If no class was specified in the component definition (because the a factory component is used instead) then this method will return null.

5.22.6 getInitMethodName

```
java.lang.String getInitMethodName()
```

All Page Within This Box

The name of the init method specified for this component, if any.

Returns:

the method name of the specified init method, or null if no init method was specified.

5.22.7 getDestroyMethodName

`java.lang.String` **`getDestroyMethodName`**`()`

The name of the destroy method specified for this component, if any.

Returns:

the method name of the specified destroy method, or null if no destroy method was specified.

5.22.8 getConstructorInjectionMetadata

`ConstructorInjectionMetadata`
`getConstructorInjectionMetadata``()`

The constructor injection metadata for this component.

Returns:

the constructor injection metadata. This is guaranteed to be non-null and will refer to the default constructor if no explicit constructor injection was specified for the component.

5.22.9 getPropertyInjectionMetadata

`java.util.Collection` **`getPropertyInjectionMetadata`**`()`

The property injection metadata for this component.

Returns:

an immutable collection of PropertyInjectionMetadata, with one entry for each property to be injected. If no property injection was specified for this component then an empty collection will be returned.

All Page Within This Box

5.22.10 isLazy

`boolean isLazy()`

Is this component to be lazily instantiated?

Returns:

true, iff this component definition specifies lazy instantiation.

5.22.11 getFactoryMethodMetadata

`MethodInjectionMetadata getFactoryMethodMetadata()`

The metadata describing how to create the component instance by invoking a method (as opposed to a constructor) if factory methods are used.

Returns:

the method injection metadata for the specified factory method, or null if no factory method is used for this component.

5.22.12 getFactoryComponent

`Value getFactoryComponent()`

The component instance on which to invoke the factory method (if specified).

Returns:

when a factory method and factory component has been specified for this component, this operation returns the metadata specifying the component on which the factory method is to be invoked. When no factory component has been specified this operation will return null. A return value of null with a non-null factory method indicates that the factory method should be invoked as a static method on the component class itself. For a non-null return value, the Value object returned will be either a ComponentValue or ReferenceValue.

5.22.13 getScope

`java.lang.String getScope()`

The specified scope for the component lifecycle.

Returns:
a String indicating the scope specified for the component.

See Also:
SCOPE_SINGLETON, SCOPE_PROTOTYPE, SCOPE_BUNDLE

---

org.osgi.service.blueprint.reflect

Interface MapValue

All Superinterfaces:
Value

`public interface MapValue`

extends Value

A map-based value. Map keys are instances of Value, as are the Map entry values themselves.

Method Summary	
java.lang.String	**getKeyType**() The key-type specified for map keys, or null if none given
java.util.Map	**getMap**() The Map of Value->Value mappings for this map-based value
java.lang.String	**getValueType**() The value-type specified for map values, or null if none given

Method Detail

getValueType

`java.lang.String` **`getValueType`**()

The value-type specified for map values, or null if none given

---

getKeyType

`java.lang.String` **`getKeyType`**()

The key-type specified for map keys, or null if none given

---

getMap

`java.util.Map` **`getMap`**()

The Map of Value->Value mappings for this map-based value

org.osgi.service.blueprint.reflect

Interface MethodInjectionMetadata

`public interface MethodInjectionMetadata`

Metadata describing a method to be invoked as part of component configuration.

Method Summary

java.lang.String	**getName**() The name of the method to be invoked.
java.util.List	**getParameterSpecifications**() The parameter specifications that determine which method to invoke (in the case of overloading) and what arguments to pass to it.

Method Detail

getName

`java.lang.String getName()`

The name of the method to be invoked.

Returns:

the method name, overloaded methods are disambiguated by parameter

specifications.

---

getParameterSpecifications

`java.util.List` **`getParameterSpecifications`**()

The parameter specifications that determine which method to invoke (in the case of overloading) and what arguments to pass to it.

Returns:

an immutable List of ParameterSpecification, or an empty list if the method takes no arguments. The list is ordered by ascending parameter index. I.e., the first parameter is first in the list, and so on.

---

org.osgi.service.blueprint.reflect

Interface NullValue

All Superinterfaces:

Value

`public interface` **`NullValu`**e

extends Value

A value specified to be null via the element.

Field Summary	
static NullValue	**NULL**

Field Detail

NULL

static final NullValue **NULL**

org.osgi.service.blueprint.reflect

Interface ParameterSpecification

public interface **ParameterSpecificatio**n

Metadata describing a parameter of a method or constructor and the value that is to be passed during injection.

Method Summary

int	**getIndex**() The (zero-based) index into the parameter list of the method or constructor to be invoked for this parameter.
java.lang.String	**getTypeName**() The type to convert the value into when invoking the constructor or factory method.
Value	**getValue**() The value to inject into the parameter.

Method Detail

getValue

Value **getValue**()

The value to inject into the parameter.

Returns:

the parameter value

---

getTypeName

java.lang.String **getTypeName**()

The type to convert the value into when invoking the constructor or factory method. If no explicit type was specified on the component definition then this method returns null.

Returns:

the explicitly specified type to convert the value into, or null if no type was specified in the component definition.

All Page Within This Box

getIndex

`int getIndex()`

The (zero-based) index into the parameter list of the method or constructor to be invoked for this parameter. This is determined either by explicitly specifying the index attribute in the component declaration, or by declaration order of constructor-arg elements if the index was not explicitly set.

Returns:

the zero-based parameter index

---

org.osgi.service.blueprint.reflect

Interface PropertiesValue

All Superinterfaces:

Value

`public interface PropertiesValue`

extends Value A java.util.Properties based value

Method Summary

java.util.Properties	getPropertiesValue()

Method Detail

All Page Within This Box

getPropertiesValue

`java.util.Properties` **`getPropertiesValue`**()

---

org.osgi.service.blueprint.reflect

Interface PropertyInjectionMetadata

`public interface` **`PropertyInjectionMetadata`**

Metadata describing a property to be injected. Properties are defined following JavaBeans conventions.

Method Summary

`java.lang.String`	**`getName`**() The name of the property to be injected, following JavaBeans conventions.
`Value`	**`getValue`**() The value to inject the property with.

Method Detail

getName

`java.lang.String` **`getName`**()

The name of the property to be injected, following JavaBeans conventions.

Returns:
the property name.

getValue

`Value getValue()`

The value to inject the property with.

Returns:

the property value.

---

org.osgi.service.blueprint.reflect

Interface ReferenceNameValue

All Superinterfaces:

Value

`public interface ReferenceNameValue`

extends Value

A value which represents the name of another component in the module context. The name itself will be injected, not the component that the name refers to.

Method Summary

java.lang.String	**getReferenceName**()

Method Detail

getReferenceName

`java.lang.String` **`getReferenceName`**`()`

---

org.osgi.service.blueprint.reflect

Interface ReferenceValue

All Superinterfaces:

Value

`public interface` **`ReferenceValue`**

extends Value

A value which refers to another component in the module context by name.

Method Summary

`java.lang.String`	**`getComponentName`**`()` The name of the referenced component.

Method Detail

getComponentName

`java.lang.String` **`getComponentName`**`()`

The name of the referenced component.

org.osgi.service.blueprint.reflect

Interface RegistrationListenerMetadata

`public interface` **`RegistrationListenerMetadata`**

Metadata for a listener interested in service registration and unregistration events for an exported service.

Method Summary

`Value`	**`getListenerComponent`**`()` The component instance that will receive registration and unregistration events.
`java.lang.String`	**`getRegistrationMethodName`**`()` The name of the method to invoke on the listener component when the exported service is registered with the service registry.
`java.lang.String`	**`getUnregistrationMethodName`**`()` The name of the method to invoke on the listener component when the exported service is unregistered from the service registry.

Method Detail

getListenerComponent

`Value` **`getListenerComponent`**`()`

The component instance that will receive registration and unregistration events. The returned value must reference a component and therefore be either a ComponentValue, ReferenceValue, or ReferenceNameValue.

Returns:

All Page Within This Box

the listener component reference.

---

getRegistrationMethodName

`java.lang.String` **`getRegistrationMethodName`**()

The name of the method to invoke on the listener component when the exported service is registered with the service registry.

Returns:
the registration callback method name.

---

getUnregistrationMethodName

`java.lang.String` **`getUnregistrationMethodName`**()

The name of the method to invoke on the listener component when the exported service is unregistered from the service registry.

Returns:
the unregistration callback method name.

---

org.osgi.service.blueprint.reflect

Interface ServiceExportComponentMetadata

All Superinterfaces:
ComponentMetadata

`public interface` **`ServiceExportComponentMetadat`**a

extends ComponentMetadata

Metadata representing a service to be exported by a module context.

Field Summary	
static int	**EXPORT_MODE_ALL** Advertise all Java classes and interfaces in the exported component's type as service interfaces.
static int	**EXPORT_MODE_CLASS_HIERARCHY** Advertise all Java classes in the hierarchy of the exported component's type as service interfaces.
static int	**EXPORT_MODE_DISABLED** Do not auto-detect types for advertised service intefaces
static int	**EXPORT_MODE_INTERFACES** Advertise all Java interfaces implemented by the exported component as service interfaces.

Method Summary

int	**getAutoExportMode**() Return the auto-export mode specified.
Value	**getExportedComponent**() The component that is to be exported as a service.
java.util.Set	**getInterfaceNames**() The type names of the set of interface types that the service should be advertised as supporting, as specified in the component declaration.
int	**getRanking**() The ranking value to use when advertising the service
java.util.Collection	**getRegistrationListeners**() The listeners that have registered to be notified when the exported service is registered and unregistered with the framework.
java.util.Map	**getServiceProperties**() The user declared properties to be advertised with the service.

Methods inherited from interface org.osgi.service.blueprint.reflect.ComponentMetadata

getExplicitDependencies, getName

Field Detail

EXPORT_MODE_DISABLED

```
static final int EXPORT_MODE_DISABLED
```

Do not auto-detect types for advertised service intefaces

See Also:

Constant Field Values

All Page Within This Box

EXPORT_MODE_INTERFACES

`static final int` **`EXPORT_MODE_INTERFACE`**S

Advertise all Java interfaces implemented by the exported component as service interfaces.

See Also:

Constant Field Values

EXPORT_MODE_CLASS_HIERARCHY

`static final int` **`EXPORT_MODE_CLASS_HIERARCH`**Y

Advertise all Java classes in the hierarchy of the exported component's type as service interfaces.

See Also:

Constant Field Values

EXPORT_MODE_ALL

`static final int` **`EXPORT_MODE_AL`**L

Advertise all Java classes and interfaces in the exported component's type as service interfaces.

See Also:

Constant Field Values

Method Detail

getExportedComponent

`Value` **`getExportedComponent`**()

The component that is to be exported as a service. Value must refer to a component and therefore be either a ComponentValue or ReferenceValue.

Returns:
the component to be exported as a service.

---

getInterfaceNames

`java.util.Set` **`getInterfaceNames`**()

The type names of the set of interface types that the service should be advertised as supporting, as specified in the component declaration.

Returns:
an immutable set of (String) type names, or an empty set if using auto-export

---

getAutoExportMode

`int` **`getAutoExportMode`**()

Return the auto-export mode specified.

Returns:
One of EXPORT_MODE_DISABLED, EXPORT_MODE_INTERFACES, EXPORT_MODE_CLASS_HIERARCHY, EXPORT_MODE_ALL

---

getServiceProperties

`java.util.Map` **`getServiceProperties`**()

The user declared properties to be advertised with the service.

All Page Within This Box

Returns:

Map containing the set of user declared service properties (may be empty if no properties were specified).

---

getRanking

`int` **`getRanking`**`()`

The ranking value to use when advertising the service

Returns:

service ranking

---

getRegistrationListeners

`java.util.Collection` **`getRegistrationListeners`**`()`

The listeners that have registered to be notified when the exported service is registered and unregistered with the framework.

Returns:

an immutable collection of RegistrationListenerMetadata

---

org.osgi.service.blueprint.reflect

Interface ServiceReferenceComponentMetadata

All Superinterfaces:

ComponentMetadata

All Known Subinterfaces:

CollectionBasedServiceReferenceComponentMetadata, UnaryServiceReferenceComponentMetadata

---

`public interface` **`ServiceReferenceComponentMetadat`**a

extends ComponentMetadata

Metadata describing a reference to a service that is to be imported into the module context from the OSGi service registry.

Field Summary

static int	**MANDATORY_AVAILABILITY** A matching service is required at all times.
static int	**OPTIONAL_AVAILABILITY** A matching service is not required to be present.

Method Summary

java.util.Collection	**getBindingListeners**() The set of listeners registered to receive bind and unbind events for backing services.
java.lang.String	**getComponentName**() The value of the component name attribute, if specified.
java.lang.String	**getFilter**() The filter expression that a matching service must pass
java.util.Set	**getInterfaceNames**() The interface types that the matching service must support
int	**getServiceAvailabilitySpecification**() Whether or not a matching service is required at all times.

Methods inherited from interface org.osgi.service.blueprint.reflect. ComponentMetadata
getExplicitDependencies, getName

Field Detail

MANDATORY_AVAILABILITY

```
static final int MANDATORY_AVAILABILITY
```

A matching service is required at all times.

See Also:

Constant Field Values

OPTIONAL_AVAILABILITY

All Page Within This Box

`static final int` **`OPTIONAL_AVAILABILITY`**

A matching service is not required to be present.

See Also:

Constant Field Values

Method Detail

getServiceAvailabilitySpecification

`int` **`getServiceAvailabilitySpecification`**`()`

Whether or not a matching service is required at all times.

Returns:

one of MANDATORY_AVAILABILITY or OPTIONAL_AVAILABILITY

---

getInterfaceNames

`java.util.Set` **`getInterfaceNames`**`()`

The interface types that the matching service must support

Returns:

an immutable set of type names

---

getComponentName

`java.lang.String` **`getComponentName`**`()`

The value of the component name attribute, if specified.

Returns:

the component name attribute value, or null if the attribute was not specified

getFilter

`java.lang.String` **`getFilter`**()

The filter expression that a matching service must pass

Returns:
filter expression

getBindingListeners

`java.util.Collection` **`getBindingListeners`**()

The set of listeners registered to receive bind and unbind events for backing services.

Returns:
an immutable collection of registered BindingListenerMetadata

org.osgi.service.blueprint.reflect

Interface SetValue

All Superinterfaces:
Value

`public interface` **`SetValue`**

extends Value

A set-based value. Members of the set are instances of Value.

Method Summary	
java.util.Set	**getSet**() The Set (of Value objects) for this set-based value
java.lang.String	**getValueType**() The value-type specified for the set elements, or null if none given

Method Detail

getValueType

`java.lang.String` **`getValueType`**()

The value-type specified for the set elements, or null if none given

getSet

`java.util.Set` **`getSet`**()

The Set (of Value objects) for this set-based value

org.osgi.service.blueprint.reflect

Interface TypedStringValue

All Superinterfaces:

Value

`public interface` **`TypedStringValue`**

All Page Within This Box

extends Value

A simple string value that will be type-converted if necessary before injecting into a target.

Method Summary	
java.lang.String	**getStringValue**() The string value (unconverted) of this value).
java.lang.String	**getTypeName**() The name of the type to which this value should be coerced.

Method Detail

getStringValue

`java.lang.String` **`getStringValue`**()

The string value (unconverted) of this value).

---

getTypeName

`java.lang.String` **`getTypeName`**()

The name of the type to which this value should be coerced. May be null.

---

org.osgi.service.blueprint.reflect

Interface UnaryServiceReferenceComponentMetadata

All Superinterfaces:
ComponentMetadata , ServiceReferenceComponentMetadata

`public interface` **`UnaryServiceReferenceComponentMetadat`**a

extends ServiceReferenceComponentMetadata

Service reference that will bind to a single matching service in the service registry.

Field Summary

Fields inherited from interface org.osgi.service.blueprint.reflect. ServiceReferenceComponentMetadata

MANDATORY_AVAILABILITY, OPTIONAL_AVAILABILITY

Method Summary

long	**getTimeout**() Timeout for service invocations when a matching backing service is unavailable.

Methods inherited from interface org.osgi.service.blueprint.reflect. ServiceReferenceComponentMetadata

getBindingListeners, getComponentName, getFilter, getInterfaceNames, getServiceAvailabilitySpecification

Methods inherited from interface org.osgi.service.blueprint.reflect. ComponentMetadata

getExplicitDependencies, getName

Method Detail

getTimeout

long **getTimeout**()

Timeout for service invocations when a matching backing service is unavailable.

All Page Within This Box

Returns:

service invocation timeout in milliseconds

---

org.osgi.service.blueprint.reflect

Interface Value

All Known Subinterfaces:

ArrayValue, ComponentValue, ListValue, MapValue, NullValue, PropertiesValue, ReferenceNameValue, ReferenceValue, SetValue, TypedStringValue

---

`public interface Value`

A value to inject into a field, property, method argument or constructor argument.

Constant Field Values

5.22.14 Contents

- org.osgi.*

org.osgi.*

org.osgi.service.blueprint.context.ModuleContext		
public static final int	BUNDLE_STOPPING	2
public static final int	CONFIGURATION_ADMIN_OBJECT_DELETED	1

org.osgi.service.blueprint.context.ModuleContextEventConstants		
public static final java.lang.String	BUNDLE_VERSION	"bundle.version"
public static final java.lang.String	EXTENDER_BUNDLE	"extender.bundle"
public static final java.lang.String	EXTENDER_ID	"extender.bundle.id"
public static final java.lang.String	EXTENDER_SYMBOLICNAME	"extender.bundle.symbolicName"
public static final java.lang.String	TOPIC_BLUEPRINT_EVENTS	"org/osgi/service/blueprint"

All Page Within This Box

public static final java.lang.String	TOPIC_CREATED	"org/osgi/service/blueprint/context/CREATED"
public static final java.lang.String	TOPIC_CREATING	"org/osgi/service/blueprint/context/CREATING"
public static final java.lang.String	TOPIC_DESTROYED	"org/osgi/service/blueprint/context/DESTROYED"
public static final java.lang.String	TOPIC_DESTROYING	"org/osgi/service/blueprint/context/DESTROYING"
public static final java.lang.String	TOPIC_FAILURE	"org/osgi/service/blueprint/context/FAILURE"
public static final java.lang.String	TOPIC_WAITING	"org/osgi/service/blueprint/context/WAITING"

org.osgi.service.blueprint.reflect.CollectionBasedServiceReferenceComponentMetadata

public static final int	MEMBER_TYPE_SERVICE_REFERENCES	2

public static final int	MEMBER_TYPE_SERVICES	1
public static final int	ORDER_BASIS_SERVICE_REFERENCES	2
public static final int	ORDER_BASIS_SERVICES	1

org.osgi.service.blueprint.reflect.LocalComponentMetadata		
public static final java.lang.String	SCOPE_BUNDLE	"bundle"
public static final java.lang.String	SCOPE_PROTOTYPE	"prototype"
public static final java.lang.String	SCOPE_SINGLETON	"singleton"

org.osgi.service.blueprint.reflect.ServiceExportComponentMetadata		
public static final int	EXPORT_MODE_ALL	4
public static final int	EXPORT_MODE_CLASS_HIERARCHY	3
public static final int	EXPORT_MODE_DISABLED	1
public static final int	EXPORT_MODE_INTERFACES	2

org.osgi.service.blueprint.reflect.ServiceReferenceComponentMetadata		
public static final int	MANDATORY_AVAILABILITY	1
public static final int	OPTIONAL_AVAILABILITY	2

Serialized Form

Package org.osgi.service.blueprint.context

Class org.osgi.service.blueprint.context.ComponentDefinitionException extends java.lang.RuntimeException implements Serializable

Class org.osgi.service.blueprint.context.NoSuchComponentException extends java.lang.RuntimeException implements Serializable

Serialized Fields

componentName

`java.lang.String` **componentName**

Class org.osgi.service.blueprint.context.ServiceUnavailableException extends java.lang.RuntimeException implements Serializable

Serialized Fields

serviceType

`java.lang.Class` **serviceType**

---

filter

All Page Within This Box

java.lang.String **filte**r

Package org.osgi.service.blueprint.namespace

Class org.osgi.service.blueprint.namespace.ComponentNameAlreadyInUseException extends java.lang.RuntimeException implements Serializable

Serialized Fields

duplicateName

java.lang.String **duplicateNam**e

5.23 'osgi' Schema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified"
        version="1.0.0">

  <!-- Schema elements for core component declarations -->

  <xsd:complexType name="TidentifiedType" abstract="true">
    <xsd:attribute name="id" type="xsd:ID">
    </xsd:attribute>
  </xsd:complexType>

  <xsd:element name="components">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="description" minOccurs="0"/>
        <xsd:element ref="type-converters" minOccurs="0" maxOccurs="1"/>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
```

```
            <xsd:element ref="component"/>
            <xsd:element ref="ref-list"/>
            <xsd:element ref="ref-set"/>
            <xsd:element ref="reference"/>
            <xsd:element ref="service"/>
            <xsd:any namespace="##other" processContents="strict" minOccurs="0"
              maxOccurs="unbounded"/>
          </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="default-lazy-init" default="false" type="xsd:boolean"/>
        <xsd:attribute name="default-init-method" type="xsd:string"/>
        <xsd:attribute name="default-destroy-method" type="xsd:string"/>
        <xsd:attributeGroup ref="defaults" />
        <xsd:anyAttribute namespace="##other" processContents="lax"/>
    </xsd:complexType>
 </xsd:element>

<xsd:element name="description">
    <xsd:complexType mixed="true">
        <xsd:choice minOccurs="0" maxOccurs="unbounded"/>
    </xsd:complexType>
 </xsd:element>

<xsd:element name="type-converters">
                        <xsd:complexType mixed="true">
                  <xsd:choice minOccurs="0" maxOccurs="unbounded">
                    <xsd:element ref="component"/>
                    <xsd:element ref="ref"/>
                  </xsd:choice>
    </xsd:complexType>
</xsd:element>

<xsd:group name="componentElements">
    <xsd:sequence>
        <xsd:element ref="description" minOccurs="0"/>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="constructor-arg"/>
            <xsd:element ref="property"/>
            <xsd:any namespace="##other" processContents="strict" minOccurs="0"
            maxOccurs="unbounded"/>
        </xsd:choice>
    </xsd:sequence>
 </xsd:group>

<xsd:attributeGroup name="componentAttributes">
    <xsd:attribute name="class" type="xsd:string"/>
    <xsd:attribute name="scope" type="xsd:string"/>
    <xsd:attribute name="lazy-init" default="default" type="Tdefaultable-boolean"/>
    <xsd:attribute name="depends-on" type="xsd:string"/>
    <xsd:attribute name="init-method" type="xsd:string"/>
    <xsd:attribute name="destroy-method" type="xsd:string"/>
    <xsd:attribute name="factory-method" type="xsd:string"/>
    <xsd:attribute name="factory-component" type="xsd:string"/>
```

```
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:attributeGroup>

  <xsd:element name="component">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="TidentifiedType">
          <xsd:group ref="componentElements"/>
          <xsd:attributeGroup ref="componentAttributes"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>

  <xsd:group name="valueElement">
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="1">
        <xsd:element ref="component"/>
        <xsd:element ref="ref"/>
        <xsd:element ref="idref"/>
        <xsd:element ref="value"/>
        <xsd:element ref="null"/>
        <xsd:element ref="list"/>
        <xsd:element ref="set"/>
        <xsd:element ref="map"/>
        <xsd:element ref="array"/>
        <xsd:element ref="props"/>
        <xsd:element ref="ref-list"/>
        <xsd:element ref="ref-set"/>
        <xsd:element ref="reference"/>
        <xsd:element ref="service"/>
        <xsd:any namespace="##other" processContents="strict"
          minOccurs="0" maxOccurs="unbounded"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:group>

  <xsd:group name="keyElement">
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="1">
        <xsd:element ref="component"/>
        <xsd:element ref="ref"/>
        <xsd:element ref="idref"/>
        <xsd:element ref="value"/>
        <xsd:element ref="list"/>
        <xsd:element ref="set"/>
        <xsd:element ref="map"/>
        <xsd:element ref="array"/>
        <xsd:element ref="props"/>
        <xsd:element ref="ref-list"/>
        <xsd:element ref="ref-set"/>
        <xsd:element ref="reference"/>
        <xsd:element ref="service"/>
```

```
            <xsd:any namespace="##other" processContents="strict"
              minOccurs="0" maxOccurs="unbounded"/>
          </xsd:choice>
        </xsd:sequence>
      </xsd:group>


    <xsd:element name="constructor-arg">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="description" minOccurs="0"/>
                        <xsd:group ref="valueElement"/>
          </xsd:sequence>
          <xsd:attribute name="index" type="xsd:string"/>
          <xsd:attribute name="type" type="xsd:string"/>
          <xsd:attribute name="ref" type="xsd:string"/>
          <xsd:attribute name="value" type="xsd:string"/>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="property" type="TpropertyType"/>

      <xsd:element name="ref">
        <xsd:complexType>
          <xsd:complexContent>
            <xsd:restriction base="xsd:anyType">
              <xsd:attribute name="component" type="xsd:string" use="required"/>
            </xsd:restriction>
          </xsd:complexContent>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="idref">
        <xsd:complexType>
          <xsd:complexContent>
            <xsd:restriction base="xsd:anyType">
              <xsd:attribute name="component" type="xsd:string" use="required"/>
            </xsd:restriction>
          </xsd:complexContent>
        </xsd:complexType>
      </xsd:element>


    <xsd:element name="value">
        <xsd:complexType mixed="true">
          <xsd:choice minOccurs="0" maxOccurs="unbounded"/>
          <xsd:attribute name="type" type="xsd:string"/>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="null">
        <xsd:complexType mixed="true">
          <xsd:choice minOccurs="0" maxOccurs="unbounded"/>
```

```
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="list" type="TlistSetArrayType"/>
  <xsd:element name="set" type="TlistSetArrayType"/>
  <xsd:element name="map" type="TmapType"/>
  <xsd:element name="array" type="TlistSetArrayType"/>
  <xsd:element name="entry" type="TentryType"/>
  <xsd:element name="props" type="TpropsType"/>
  <xsd:element name="key">
    <xsd:complexType>
                    <xsd:group ref="keyElement"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="prop">
    <xsd:complexType mixed="true">
      <xsd:attribute name="key" type="xsd:string" use="required"/>
      <xsd:attribute name="value" type="xsd:string" use="optional"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="TpropertyType">
    <xsd:sequence>
      <xsd:element ref="description" minOccurs="0"/>
          <xsd:group ref="valueElement"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="ref" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:string"/>
  </xsd:complexType>

<!-- Collection Types -->

  <!-- base collection type -->
  <xsd:complexType name="TbaseCollectionType"/>

  <!-- base type for collections that have (possibly) typed nested values -->
  <xsd:complexType name="TtypedCollectionType">
    <xsd:complexContent>
      <xsd:extension base="TbaseCollectionType">
        <xsd:attribute name="value-type" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- 'map' element type -->
  <xsd:complexType name="TmapType">
    <xsd:complexContent>
      <xsd:extension base="TtypedCollectionType">
        <xsd:sequence>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="entry"/>
```

```
            </xsd:choice>
          </xsd:sequence>
          <xsd:attribute name="key-type" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- 'entry' element type -->
    <xsd:complexType name="TentryType">
      <xsd:sequence>
        <xsd:element ref="key" minOccurs="0"/>
        <xsd:group ref="valueElement"/>
      </xsd:sequence>
      <xsd:attribute name="key" type="xsd:string"/>
      <xsd:attribute name="key-ref" type="xsd:string"/>
      <xsd:attribute name="value" type="xsd:string"/>
      <xsd:attribute name="value-ref" type="xsd:string"/>
    </xsd:complexType>

    <!-- 'list' and 'set' collection type -->
    <xsd:complexType name="TlistSetArrayType">
      <xsd:complexContent>
        <xsd:extension base="TtypedCollectionType">
          <xsd:group ref="valueElement" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- 'props' collection type -->
    <xsd:complexType name="TpropsType">
      <xsd:complexContent>
        <xsd:extension base="TbaseCollectionType">
          <xsd:sequence>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
              <xsd:element ref="prop"/>
            </xsd:choice>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- simple internal types -->
    <xsd:simpleType name="Tdefaultable-boolean">
      <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="default"/>
        <xsd:enumeration value="true"/>
        <xsd:enumeration value="false"/>
      </xsd:restriction>
    </xsd:simpleType>

    <xsd:attributeGroup name="defaults">
      <xsd:attribute name="default-timeout" type="xsd:long" default="300000"/>
      <xsd:attribute name="default-availability" type="Tavailability" default="mandatory"/>
```

```
</xsd:attributeGroup>

<!-- reference -->
<xsd:element name="reference" type="TsingleReference"/>


<xsd:complexType name="Treference">
  <xsd:complexContent>
    <xsd:extension base="TidentifiedType">
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="interfaces" type="TlistSetArrayType" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="listener" type="Tlistener" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="interface" use="optional" type="xsd:token"/>
      <xsd:attribute name="filter" use="optional" type="xsd:string"/>
      <xsd:attribute name="component-name" type="xsd:string" use="optional"/>
      <xsd:attribute name="availability" use="optional" type="Tavailability" default="mandatory"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>


<xsd:complexType name="Tlistener">
            <xsd:choice>
                                    <xsd:element ref="ref"/>
            <xsd:element ref="component"/>
                                    <xsd:element ref="reference"/>
                                    <xsd:element ref="service"/>
            <xsd:sequence minOccurs="0" maxOccurs="1">
            <!-- nested component declaration -->
            <xsd:any namespace="##other" minOccurs="1" maxOccurs="1" processContents="skip"/>
            </xsd:sequence>
            </xsd:choice>

    <!-- shortcut for component references -->
    <xsd:attribute name="ref" type="xsd:string" use="optional"/>
    <xsd:attribute name="bind-method" type="xsd:token" use="required"/>
    <xsd:attribute name="unbind-method" type="xsd:token" use="required"/>
</xsd:complexType>

<!-- single reference -->
<xsd:complexType name="TsingleReference">
    <xsd:complexContent>
        <xsd:extension base="Treference">
            <xsd:attribute name="timeout" use="optional" type="xsd:long"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="Tavailability">
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="mandatory"/>
        <xsd:enumeration value="optional"/>
```

All Page Within This Box

```
    </xsd:restriction>
</xsd:simpleType>

<!-- reference collections (set, list) -->
<xsd:element name="ref-list" type="TreferenceCollection"/>

<xsd:element name="ref-set" type="TreferenceCollection"/>

<xsd:complexType name="TreferenceCollection">
  <xsd:complexContent>
    <xsd:extension base="Treference">
      <xsd:sequence minOccurs="0" maxOccurs="1">
        <xsd:element name="comparator" type="Tcomparator"/>
      </xsd:sequence>
      <xsd:attribute name="comparator-ref" type="xsd:string" use="optional"/>
                                <xsd:attribute name="member-type" type="TmemberType" use="optional"/>
                                <xsd:attribute name="ordering-basis" type="TorderingBasis" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="TmemberType">
    <xsd:restriction base="xsd:token">
       <xsd:enumeration value="service-instance"/>
       <xsd:enumeration value="service-reference"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="Tcomparator">
  <xsd:choice>
                              <xsd:element ref="ref"/>
            <xsd:element ref="component"/>
                              <xsd:element ref="reference"/>
                              <xsd:element ref="service"/>
    <xsd:sequence minOccurs="1" maxOccurs="1">
       <!-- nested component declaration -->
       <xsd:any namespace="##other" minOccurs="1" maxOccurs="1" processContents="skip"/>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>

<xsd:simpleType name="TorderingBasis">
  <xsd:restriction base="xsd:token">
       <xsd:enumeration value="service"/>
       <xsd:enumeration value="service-reference"/>
  </xsd:restriction>
</xsd:simpleType>


<!-- service -->
<xsd:element name="service" type="Tservice"/>

<xsd:complexType name="Tservice">
```

```
    <xsd:complexContent>
      <xsd:extension base="TidentifiedType">
        <xsd:choice>
                <xsd:group ref="serviceElements"/>
                <!-- nested component declaration -->
                <xsd:any namespace="##other" minOccurs="0" maxOccurs="1" processContents="skip"/>
        </xsd:choice>
        <xsd:attribute name="interface" type="xsd:token" use="optional"/>
        <xsd:attribute name="ref" type="xsd:string" use="optional"/>
        <xsd:attribute name="depends-on" type="xsd:string" use="optional"/>
        <xsd:attribute name="auto-export" type="TautoExportModes" default="disabled"/>
        <xsd:attribute name="ranking" type="xsd:int" default="0"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:group name="serviceElements">
      <xsd:sequence>
        <xsd:element name="interfaces" type="TlistSetArrayType" minOccurs="0"/>
        <xsd:element name="service-properties" minOccurs="0" type="TmapType"/>
        <xsd:element name="registration-listener" type="TserviceRegistrationListener"
             minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
  </xsd:group>

  <xsd:complexType name="TserviceRegistrationListener">
                <xsd:choice>
                                        <xsd:element ref="ref"/>
                <xsd:element ref="component"/>
                                        <xsd:element ref="reference"/>
                                        <xsd:element ref="service"/>
                <xsd:sequence minOccurs="0" maxOccurs="1">
                                <!-- nested component declaration -->
                                <xsd:any namespace="##other" minOccurs="1" maxOccurs="1"
processContents="skip"/>
                                </xsd:sequence>
                </xsd:choice>

      <!-- shortcut for component references -->
      <xsd:attribute name="ref" type="xsd:string" use="optional"/>
      <xsd:attribute name="registration-method" type="xsd:token" use="required"/>
      <xsd:attribute name="unregistration-method" type="xsd:token" use="required"/>
  </xsd:complexType>

  <xsd:simpleType name="TautoExportModes">
      <xsd:restriction base="xsd:token">
        <xsd:enumeration value="disabled"/>
        <xsd:enumeration value="interfaces"/>
        <xsd:enumeration value="class-hierarchy"/>
        <xsd:enumeration value="all-classes"/>
      </xsd:restriction>
  </xsd:simpleType>
```

```
</xsd:schema>
```

5.24 'osgix' Schema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns="http://www.osgi.org/xmlns/blueprint-compendium/v1.0.0"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:osgi="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        targetNamespace="http://www.osgi.org/xmlns/blueprint-compendium/v1.0.0"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified"
        version="1.0.0">

  <xsd:import namespace="http://www.osgi.org/xmlns/blueprint/v1.0.0"/>

  <!-- property placeholder -->

  <xsd:element name="property-placeholder" type="TpropertyPlaceholder"/>

  <xsd:complexType name="TpropertyPlaceholder">
    <xsd:complexContent>
      <xsd:extension base="osgi:TidentifiedType">
        <xsd:sequence minOccurs="0" maxOccurs="1">
          <!-- nested properties declaration -->
          <xsd:element name="default-properties" type="TdefaultProperties" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="persistent-id" type="xsd:string" use="required"/>
        <xsd:attribute name="placeholder-prefix" type="xsd:string" use="optional" default="${"/>
        <xsd:attribute name="placeholder-suffix" type="xsd:string" use="optional" default="}"/>
        <xsd:attribute name="defaults-ref" type="xsd:string" use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="TdefaultProperties">
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
          <xsd:element ref="osgi:property"/>
        </xsd:sequence>
  </xsd:complexType>


  <!--  managed-properties  -->

  <xsd:element name="managed-properties" type="TmanagedProperties"/>

  <xsd:complexType name="TmanagedProperties">
    <xsd:attribute name="persistent-id" type="xsd:string" use="required"/>
    <xsd:attribute name="update-strategy" type="TupdateStrategyType" use="optional"/>
    <xsd:attribute name="update-method" type="xsd:string" use="optional"/>
  </xsd:complexType>

  <xsd:simpleType name="TupdateStrategyType">
```

All Page Within This Box

```
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="none"/>
        <xsd:enumeration value="component-managed"/>
        <xsd:enumeration value="container-managed"/>
      </xsd:restriction>
    </xsd:simpleType>

  <!--  managed-service-factory -->

  <xsd:element name="managed-service-factory" type="TmanagedServiceFactory"/>

  <xsd:complexType name="TmanagedServiceFactory">
              <xsd:complexContent>
                        <xsd:extension base="osgi:TidentifiedType">
                                  <xsd:sequence>
                      <xsd:group ref="osgi:serviceElements"/>
            <xsd:element name="managed-component" type="TmanagedComponent" minOccurs="1"
maxOccurs="1"/>
                                  </xsd:sequence>
          <xsd:attribute name="factory-pid" type="xsd:string" use="required"/>
                    <xsd:attribute name="interface" type="xsd:token" use="optional"/>
          <xsd:attribute name="auto-export" type="osgi:TautoExportModes" default="disabled"/>
          <xsd:attribute name="ranking" type="xsd:int" default="0"/>
                    <xsd:anyAttribute namespace="##other" processContents="lax"/>
              </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

              <xsd:complexType name="TmanagedComponent">
                        <xsd:group ref="osgi:componentElements"/>
      <xsd:attribute name="class" type="xsd:string"/>
      <xsd:attribute name="init-method" type="xsd:string"/>
      <xsd:attribute name="destroy-method" type="xsd:string"/>
      <xsd:attribute name="factory-method" type="xsd:string"/>
      <xsd:attribute name="factory-component" type="xsd:string"/>
      <xsd:anyAttribute namespace="##other" processContents="lax"/>
              </xsd:complexType>


  <!-- cm-properties -->

  <xsd:element name="cm-properties" type="TcmProperties"/>

  <xsd:complexType name="TcmProperties">
      <xsd:attribute name="persistent-id" type="xsd:string" use="required"/>
      <xsd:attribute name="update" type="xsd:boolean" use="optional" default="false"/>
  </xsd:complexType>


</xsd:schema>
```

All Page Within This Box

6 Considered Alternatives

Todo: document considered alternatives for behavior of a mandatory reference that becomes unsatisfied.

6.1 Type Converters

Several alternative designs for registering type converters were considered:

6.1.1 Declaring type converters in a manifest header entry

The advantage of declaring type converters to be used when creating a module context for a bundle in the bundle's manifest is that the container configuration is cleanly separated from the component instantiation.

The disadvantages where that the manifest entry looked a little out of place, and that it was not possible to have any control over the instantiation and configuration of the converters themselves.

6.1.2 Registering type converters as services in the service registry

Using the service registry would allow a standard whiteboard pattern for finding type converters. It initially seems attractive but has one serious disadvantage: it's hard to create a compatible type space. Converters implement a simple interface that in its signature converts from Object to Object (no generics allowed). A type converter published in the service registry would advertise the "Converter" interface, and use a service property to indicate the name of the class it can convert to. But how do you guarantee that the version of the class the service uses is the same as the one that the bundle using the conversion service is bound too? This requires tricky walking through PackageAdmin before the converter can be safely used. Add to this the fact that sharing of type converter instances across bundles is of limited value, and the service registry approach becomes less attractive.

6.1.3 "Magic" component declarations

A design whereby simply declaring a component that implements the Converter interface (but not needing the nested type-converters element) was considered. The difficulty with this is that the container must discover all components implementing Converter before configuring any components, which forces a phased instantiation model on components and proves problematic with situations such as constructor args. In addition, walking the type hierarchy of a component instance to discover if it implements a certain interface can be problematic when supertypes are not visible to the bundle.

.

7 Security Considerations

7.1 Service Permissions

Service registration and lookup in the Blueprint Service is built upon the existing OSGi service infrastructure. This means that Service Permission applies regarding the ability to publish,find or bind services. A service element

All Page Within This Box

declaring a component to be published as a service requires that the declaring bundle has ServicePermission[<provides>, REGISTER] for each provided interface specified for the service.

If a service reference is declared and does not specify optional cardinality, the reference cannot be satisfied unless the declaring bundle has ServicePermission[<interface>, GET] for the specified interface in the reference.

If the reference specifies optional cardinality but the declaring bundle does not have ServicePermission[<interface>, GET] for the specified interface in the reference, no service must be bound for this reference.

7.2 Required Admin Permission

A Blueprint Service implementation bundle requires AdminPermission[*,CONTEXT] because it needs access to the Bundle's BundleContext object with the Bundle.getBundleContext() method.

7.3 Using hasPermission

An implementation of the Blueprint Service should do all publishing, finding and binding of services for a Module Context using the BundleContext of the corresponding bundle. This means that normal stack-based permission checks will check the implementation bundle and not the module context's bundle. Since the implementation is registering and getting services on behalf of a module context's bundle, the implementation must call the Bundle.hasPermission method to validate that a module context's bundle has the necessary permission to register or get a service.

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

8.2 Author's Address

All Page Within This Box

Name	Adrian Colyer
Company	SpringSource
Address	Kenneth Dibben House Enterprise Road Chilworth Southampton SO16 7NS ENGLAND
Voice	+44 2380 111500
e-mail	adrian.colyer@springsource.com

8.3 Acronyms and Abbreviations

8.4 End of Document