OSGi Service Platform Release 4

Early Draft 2011.09

16 September 2011

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

DATE OF DISTRIBUTION: 16 September 2011

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

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

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

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

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

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable, worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future (“Future Specification”), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license

under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

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

Preface

This document is Early Draft 2011.09 of the OSGi Service Platform Release 4 specifications. As an *early* draft, it contains non-final specification work and it is not organized in the format normally associated with final release OSGi specifications. This document contains copies of OSGi design documents which either propose to modify existing published OSGi specifications from the OSGi Service Platform Release 4 specification documents or propose new specifications to potentially be incorporated in future OSGi Service Platform Release 4 Specification documents.

Since this early draft is not a complete specification document, the reader is expected to be familiar with OSGi Technology and the currently published OSGi Service Platform Release 4 specification documents. The reader should refer to http://www.osgi.org/About/Technology for more information on the OSGi Technology. There the reader can find a description of the OSGi Technology, as well as links to whitepapers and the OSGi Service Platform Release 4 specification documents, which are all available for download.

Pursuant to the Distribution and Feedback License above, the OSGi expert groups welcome your feedback on this early draft. Feedback can be provided by opening a bug at https://www.osgi.org/bugzilla/enter_bug.cgi?product=OSGi%20Specification.

BJ Hargrave
Chief Technical Officer
OSGi Alliance

RFC 112 OBR

Draft

67 Pages

Abstract

This document describes a bundle repository for the OSGi Alliance. This repository consists of a web site that hosts an XML resource that describes a federated repository managed the OSGi Alliance. This repository can be browsed on the web site. Additionally, the repository can be used directly from any OSGi Framework to deploy bundles from the repository (if supported by the bundle's licensing). This document defines the format of the XML and the OSGi service to access and use the repository.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 10.1.

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

0.3 Revision History

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

Revision	Date	Comments
Initial	DEC 22 2005	Peter Kriens, Initial draft
0.1	MAR 16 2006	Peter Kriens, Prepared for release.
0.2	FEB 20 2009	Hal Hildebrand, resurrected RFC from zombie status, added schema for OBR
0.3	FEB 25 2009	Hal Hildebrand, updated the API to reflect current state of Felix OBR implementation
0.4	JUNE 10 2010	David Savage, fixed formatting and numbering issues
0.5	JUNE 10 2010	David Savage, separated RepositoryAdmin interface and ResolverFactory interface to allow flexibility for providers. Added discussion on open issues
0.6	JUNE 25 2010	David Savage, added discussion of Resolver to open issues, updated discussion of uses, mention wiring API/RFC 154 relationship
0.7	JULY 13 2010	David Savage, moved various open issues to considered alternatives or technical solution as discussed in conference call. Updated API to latest proposal
0.8	JULY 23 2010	David Savage, Fleshed out build and management usecases, diagram updates and API
0.9	OCTOBER 19 2010	David Savage, Expand on capabilities model + examples, tidy up loose ends, some work on management usecases
0.10	OCTOBER 26 2010	David Savage, Change resolver api to be a stateless api and allow previous resolution state to be parsed explicitly in resolve method

Revision	Date	Comments
0.11	OCTOBER 16 2010	David Savage, Clarified API issues wrt framework context resolution. Started work on xml schema
0.12	JANUARY 19 2011	David Savage, updated API to remove ResolverFactory based on F2F feedback also opened discussion on Extends model. Further work on xml schema
0.13	FEBRUARY 2 2011	David Savage, Add Javadoc. Change extends to requirement with extend attribute set. Updates to match namespaces used in RFC 154. Change CapabilityProvider to return Iterable vs Iterator. Add uri as attribute on Resource and add discussion on when it is mandatory. Fix xmlns in xmlschema header. State RepositoryListener api is not testable in CT
0.14	MAY 2 2011	David Savage, Work in progress update to document to reflect Environment/Map-Wiring updates to API
0.15	MAY 13 2011	David Savage, Synchronize OBR API design with sub-systems design, prepare document for public draft.
0.16	JUNE 28 2011	David Savage, Updates after feedback from F2F, remove event/listener repository model, define osgi.identity and osgi.content namespaces, update repository discovery model to use findProviders/getContent, update xml schema, update examples to use new API, add RepositoryBuilder API
0.17	JULY 13 2011	David Savage, Clarify matches statement and findProviders usage, other minor formatting tidyup
0.18	JULY 22 2011	David Savage, update property types (remove uri); update uses definition; move generic extension:=true directive to considered alternatives in favour of namespace registry approach to extensions; move repository builder to considered alternatives; add discussion on osgi.content uri and repository.getContent; update javadoc after removal of Resource.getIdentity; update xml schema
0.19	AUGUST 19 2011	David Savage, add further discussion of domain model with respect to BundleRevision; Add notes on singleton, native code and execution environment as they relate to Environments; added more discussion around multiple cardinality requirements.
0.20	SEPTEMBER 8 2011	David Savage, Removed “repository management model” from domain object model; updated checksum encoding to Hex vs Base64; clarified type of osgi.content namespace attribute value; updated resolver api after much discussion on resolving requirements vs resources; updated code examples after API change; updated to latest javadoc; moved licensing to considered alternatives; document behavior for no effective requirements in resolution; add description of relative vs absolute URI usage in OBR xml; clarify that name is for information purposes only in OBR xml

Revision	Date	Comments
0.21	SEPTEMBER 15 2011	David Savage, add “name” to “osgi.content” namespace optional attributes and minor tidy up ready for public draft

1 Introduction

1.1 Acknowledgements

This document is based on the excellent work done by a number of contributors to this area including Richard S. Hall with the Oscar Bundle Repository, Robert Dunne and David Savage on the Nimble Resolver and the Sigil development framework, and Pascal Rapicault on the P2 provisioning platform.

1.2 Introduction

The uptake of the OSGi Specifications by the open source communities like Eclipse, Apache, and Knopflerfish has multiplied the number of available bundles. This is causing a confusing situation for end users because it is hard to find suitable bundles; there is currently no central repository.

This document addresses this lack of a repository. Not only describes it a concrete implementation of the OSGi Alliance’s repository (which will link member’s repositories), it also provides an XML format and service interface.

2 Application Domain

2.1 Introduction

OSGi specifications are being adopted at an increasing rate. The number of bundles available world wide is likely in the thousands, if not low ten thousands. Although many of these bundles are proprietary and not suitable for distribution, there are a large number of distributable bundles available. The current situation is that vendors have proprietary bundle repositories. However, there are a number of available solutions to downloading and installing bundles. These include:

- The Felix OSGi Bundle Repository which allows end users to discover bundles using a command line tool that runs on any OSGi Framework.

- The P2 provisioning platform used in Eclipse
- Maven repositories which are supported by a number of provisioning tools such as PAXRunner and Karaf
- Nimble Repositories which extend the OSGi Bundle Repository concepts to deal with active state dependencies.

Since bundles explicitly declare requirements in their manifest file, it is possible to define a bundle repository service that provides access to this metadata to enable remote reasoning about bundle dependencies.

In general, bundle requirements are satisfied by capabilities provided by other bundles, the environment, or other resources. Resolving bundle requirements to provided capabilities is a constraint solving process. Some constraints are of a simple provide/require nature, while other constraints can include notions of versions and version ranges. One of the more complex constraints is the *uses* directive, which is used by package exporters to constrain package importers.

The OSGi specification defines numerous types of bundle requirements, such as Import Package, Require Bundle, Fragment Host, and Execution Environment. However, it is expected that new types of requirements and capabilities for resolving them will be defined in the future. Additionally, not all capabilities will be provided by bundles; for example, screen size or available memory could be capabilities.

Conceptually, capabilities can simply be viewed as the properties or characteristics of a bundle or the environment and requirements can be viewed as a selection constraint over these capabilities. On the whole, requirements are more complex than capabilities. The selection constraint of a requirement has two orthogonal aspects: *multiple* and *optional*. For example, an imported package is not optional and not multiple, while an imported service could have *multiple* cardinality. Likewise, imported packages or services can be *mandatory* or *optional*.

Further, *extends* relationships allow a provider to *extend* another bundle. For example, a bundle fragment defines an *extends* relationship between a bundle and a host. Specifically, a given bundle requirement is a relationship that the bundle knows about in advance, as opposed to an extension, which may not have been known in advance by the bundle.

The process of resolving bundle requirements is complicated because it is non-trivial to find optimal solutions. The OSGi framework defines a run-time resolution process, which is concerned with many of the aspects described above. However, a provisioning resolution process for bundle discovery and deployment is also necessary, which is similar to the framework resolution process, but more generic.

2.2 Use cases

2.2.1 Provisioning

A repository can be used to simplify bundle provisioning by making it possible to create mechanisms to automate processing of deployment-related bundle requirements. The OSGi Framework already handles bundle requirement processing, such as resolving imported packages, required bundles, host bundles, and execution environments. However, the framework can only reason about and manage these requirements after bundles have been installed locally.

When a bundle is installed, all its requirements must be fulfilled. If its requirements can not be resolved, the bundle will fail to install or resolve. The missing requirements can potentially be resolved by installing other bundles; however, these bundles not only provide new capabilities, but they can also add new requirements that need to be resolved. This is a recursive process.

Downloaded bundles are usually licensed. Licensing issues are complex and dependent on the vendor of the bundle. The way a bundle is licensed may seriously affect the way the bundle can be downloaded. Many organizations require their employees to read the license before they download the actual artifact because many licenses contain an implicit agreement.

2.2.2 Build

It is also possible to use OSGi bundle meta data to resolve compile time dependencies as shown in the open source project Sigil. This use case follows a similar pattern to the provisioning usecase in that a known package is required to satisfy a compile time dependency so the bundle used to satisfy that dependency can be downloaded on demand.

As with provisioning during the build process it is often necessary for a developer to read and/or acknowledge a license before using a library.

Compared to the “static” repositories defined in the provisioning use case Sigil also treats the workspace as a repository from which to satisfy dependencies for other projects in the workspace.

2.2.3 Management

System administrators are tasked with managing the set of bundles used in a corporate environment. As they are often not the original developer of the bundles they greatly benefit from helpful diagnostic information that can tell them why a certain resolution failure occurs (In order to find out who to contact/blame to get the problem fixed).

System administrators require the ability to:

- Check that a given set of bundles can deploy within a framework – prior to deploying the application.
- Browse and search bundles from a repository to find specific characteristics such as bundles built on a certain date, or by a certain author.
- Navigate through bundle dependencies to find out what the impacts of deleting or upgrading a bundle may be
- Create repositories from various sources including existing enterprise archives, filesystems, public repositories such as maven central, etc.

Enterprise repositories are often federated to enable simplified management of sub sets of bundles, there may be many duplicate bundle entries in these federations. The rate of deployment to these federated repositories can vary depending on the work rate of the team managing the repository. For example in a finance environment front office traders are often releasing several artifacts a day to update algorithms used during trading where as back office systems are more likely to update on a weekly or monthly basis.

3 Problem Description

The problem this document addresses is that end users can not discover and deploy available bundles from a single, trusted, point of access.

3.1 Resolver Strategies

The resolver capability of OBR may do one of a number of things:

1. Resolve against an “empty” framework i.e. calculate the set of resources that are required to provide some top level function outside the current installed bundles. This usecase is applicable if you want to use OBR as a packaging tool - i.e. to create static lists of bundles to be deployed in "clean" frameworks.

2. Resolve against the current framework such that the resolution will resolve in the context of the existing bundles in the framework - i.e. deploy bundle x and y in a framework and drag in all bundles not yet deployed in the framework in the resolved state,

3.1.1 Existing resolver strategies

- The OBR implementation on Felix also supports resolving against an empty container (This being case (1) from above.

- In Sigil the resolver can find solutions against an empty initial set of dependencies (case 1 again) but in this case the use case is resolving OSGi bundles that will satisfy classpath requirements vs runtime requirements and the deployment process is different in that the target is a file system/eclipse classpath container vs an OSGi framework.

- In Nimble the resolver pulls bundles (or other resources – using pluggable deployment schemes) into a running framework (2 - though 1 is a trivial extension) based on requirements and policies and deploys bundles in different states (active/resolved/etc).

This specification will address both usecases.

4 Requirements

4.1 Functional

- Must not preclude browsing access to a bundle repository via a web server

- Provide access to a bundle repository so that bundles can be directly installed after discovery

- Handle dependency resolution so that bundles can be deployed without generating errors
- Allow repositories to be linked, creating a federated repository
- Allow multiple repositories to be used during resolution
- Allow ordering on repositories such that certain repositories are preferred over others during resolution
- Provide programmatic (service) access to the repository

4.2 Discovery

- Search bundles by keywords
- Search by namespace
- Provide filtering capabilities on execution environment
- Licensing conditions must be available before downloading the artifacts

4.3 Dependency Resolution

- Must be able to find bundles that can solve any unresolved requirements
- Must not preclude the ability to attach fragments or other extensions as part of a resolution
- Must handle all the requirements/capabilities and their directives as defined in the OSGi R4 specifications
- Must reuse the semantics of requirements/capabilities defined in RFC 154 where ever possible

4.4 Security

- A repository provider must be able to control the members of a federated repository.

4.5 Non Functional

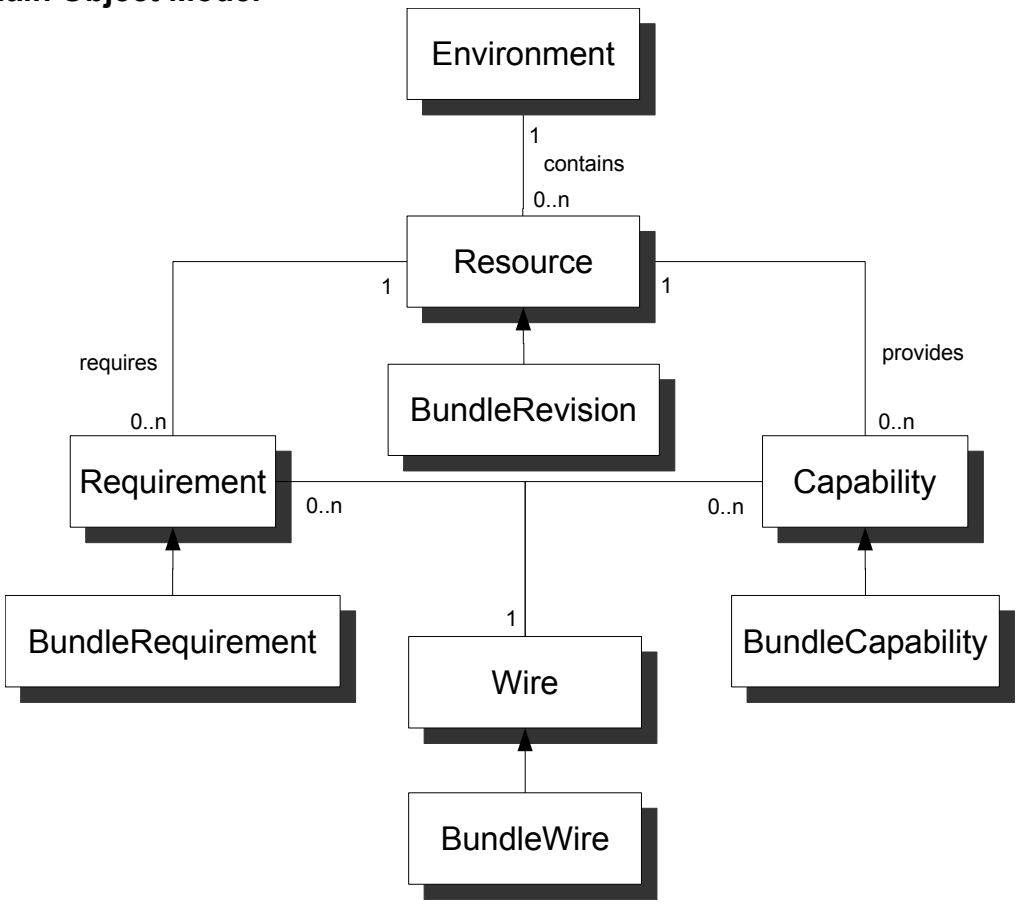
- The repository must be able to scale to ten thousand bundles
- Compliant with other OSGi services
- Easy to use
- It must be possible to implement a repository with a simple file. That is, a server must not be required

5 Technical Solution

5.1 Entities

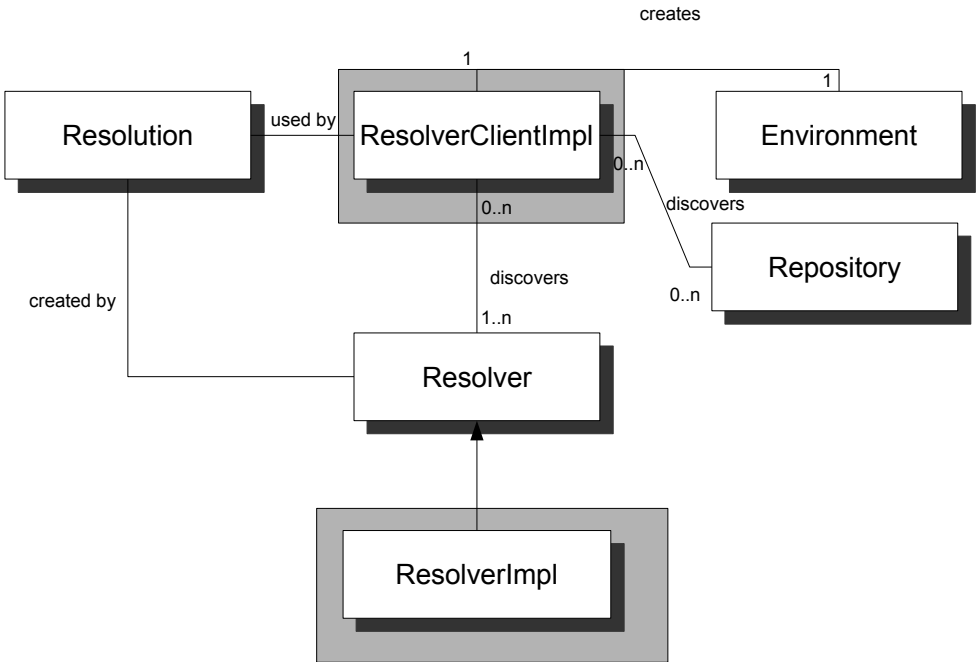
- Attribute – A map of key value properties that define the idenitity of a Capability
- Directive – A map of key value instructions that direct a Resolver in the processing of Requirements and Capabilities
- Capability – A namespaced set of attributes and directives
- Requirement – An assertion on a resource's capabilities defined by a set of directives
- Wire – A connection between a requirement and a capability associated with a resource
- Resource – A description of a bundle or other artifact that can be installed on a device. A resource provides capabilities and requires capabilities of other resources or the environment
- Environment – An environment provides options and constraints to the potential solution of a Resolver resolve operation
- Resolver – an object that can be used to find dependent and extension resources based on a set of requirements and a supplied environment
- Resolution – a Map of resources to wires
- Repository – Provides access to a set of resources
- Repository File – An XML file that can be referenced by a URL. The content contains meta data of resources and referrals to other repository files. It can be a static file or generated by a server.

5.1.1 Domain Object Model



The OSGi module layer is a concrete implementation of a generic wiring model. The recently added org.osgi.framework.wiring package has been extended in this design to build on this generic model and to reuse objects within the Resolver API.

5.1.2 Service Model



5.2 Overview

The key architecture of the OSGi Repository is a generic description of a resource and its dependencies. A resource can represent a physical or virtual resource. Physical resources include elements such as

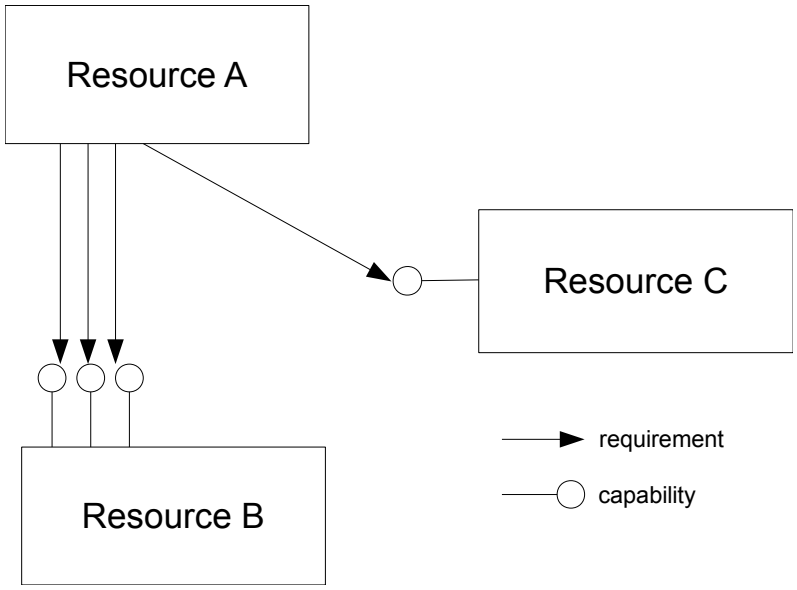
- A bundle,
- A certificate
- A configuration file

Virtual resources include elements such as:

- A service
- A particular bundle revision in a framework
- A process such as a web container

The purpose of the repository is to discover potential resources that can be instantiated by an OSGi framework and deploy these resources without causing install errors due to missing dependencies.

For this purpose, each resource has a list of requirements on other resources or the environment, a list of capabilities that are used to satisfy the requirements. Capabilities have Attributes and Directives that define how they behave, Requirements have Directives that select specific capabilities. This is depicted in the following picture.



5.2.1 Attributes and Directives

Attributes and directives are Maps of key value pairs that define behaviour of requirements and capabilities. In an directives map the keys and values are both String values. In an attributes map the keys are always String values, but the values may be one of the property types defined in the Core specification Filter Syntax (section 3.2.7).

5.2.2 Capabilties

A *capability* is anything that can be described with a set of attributes and directives. Examples of capabilities are:

- A package export
- A service export
- A fragment host
- A bundle
- A certificate
- A configuration record
- An Execution Environment
- A Display type
- Memory size
- Accessories

Capabilities are *namespaced*. The reason they are namespaced is so that they can only be provided to requirements with the same namespace. This is necessary because an attribute from two capabilities could have different meanings but still use the same name. To prevent these name clashes, the capabilities (and the

requirements that they can resolve) are namespaced. This specification defines namespaces necessary to handle the capability/requirements of the OSGi Bundle Manifest, generalised resource identity and content description.

Capabilities publish a set of attributes that identify the capability and allow requirements to be matched. In order to simplify the task of expressing capabilities in the rest of this document the following nomenclature will be defined:

```
capability <namespace> {

  <attributeName> = <attributeValue>

  <directiveName> := <directiveValue>

}
```

All capabilities must publish one identity attribute that matches the name of their namespace, e.g.

```
capability osgi.package {

  osgi.package = com.example

}
```

Along side attributes capabilities also support directives, directives provide information to a resolver to allow for extensions to resolution strategies for specific use cases.

The following capability directives are defined for all capabilities:

- `uses: packageName (, packageName)*` - The uses directive lists package names that are used by this capability. This information is intended to be used for uses constraints, see Package Constraints - section 3.7.5 of the Core Specification.

Capabilities can originate from other resources, but they can also be innate in the environment. This specification allows any bundle to dynamically provide capabilities to the environment.

5.2.3 Requirements

A requirement expressed as a filter on a resource. Just like a capability, a requirement is namespaced and support attributes and directives. All requirements should define a filter directive that matches the OSGi filter syntax.

A capability matches this requirement when all of the following are true:

- The specified capability has the same name space as this requirement.
- The filter specified by the filter directive of this requirement matches the attributes of the specified capability.
- The standard capability directives that influence matching and that apply to the name space are satisfied. See the capability mandatory directive.

Requirements also support directives that can be used to define behaviors such as optional imports, multiple cardinality, effective time, etc. The following requirement directives are defined:

```
resolution: dynamic | optional | mandatory (default mandatory)

cardinality: single | multiple (default single)

effective: <any-string> (default resolve)
```

- Mandatory requirements must have at least one valid capability match.
- Optional requirements will not fail a resolution if a provider is not found.
- Dynamic requirements are attached at runtime and a resolver may treat them as optional
- Requirements with a singular cardinality select one valid capability, if more than one is available then the resolver must choose the first capability it finds from the Environment.
- Requirements with a multiple cardinality can select one or more valid capabilities. It is up to the definition of a namespace as to whether there is a logical meaning to a multiple cardinalty requirement. For example osgi.wiring.package requirements should never have multiple cardinality as it has no meaning in that namespace. The resolver does not have to enforce the consistency of the namespace, it is up to the client to check that the resultant wiring has a well defined meaning.
- Requirements can be effective at different times, as an example a bundle may only have a dependency on another bundle when it is active.

5.2.4 Extensions

Requirements select a set of useful or required resources, an extension reverses this model; an Extension selects resources for which it might be useful. For example, a fragment can extend its host or weaving hook can add functionality to an existing class. In both cases, the bundle that provides the extension is aware of the host but the host not of the providers. Extensions are defined by the namespace for the corresponding capability and requirement. This specification currently only recognizes the `osgi.wiring.host` namespace as an extension namespace.

Resolvers must handle attachment of extension capabilities when building a wiring, the mechanism via which extensions are selected is explicitly not defined in this document. When an extension resource binds to a capability it delivers it's capabilities and requirements to the resource for that capability. In all other respects an extension acts like an ordinary requirement for the resolver.

5.2.5 Requirement and Extension examples

- Package imports may be optional, mandatory or dynamic but they cannot be multiple
- Require bundle imports may be optional or mandatory and they cannot be multiple
- A Fragment-Host is optional, may be singular or multiple and extends the capability it binds to
- A service dependency may be optional or mandatory and can be singular or multiple

5.2.6 Resource

A resource is a collection of requirements and capabilities, resources may be physical resources such as bundles, certificates, or properties files or virtual resources such as services, or processes.

A resource is identified by a special mandatory “osgi.identity” capability, which has the following attributes:

- osgi.identity - A name for the resource that is globally unique for the function of the resource. There can exists multiple resources with the same name but a different version or type. Two resources with the same type, name and version are considered to be identical. For a bundle, this is normally mapped to the Bundle-SymbolicName manifest header
- version - A version for the resource. This must be a version usable by the OSGi Framework version class. For a bundle this is mapped to the Bundle-Version manifest header
- type - The type of this resource for example a bundle has the namespace “osgi.bundle”. This maps to the RFC 154 concept of a namespace

A resource may also define an “osgi.content” capability, which has the following attributes:

- osgi.content – a String typed attribute that specifies where a resource may be downloaded from, this String must be a valid URI
- checksum – an optional attribute providing the Hex encoded checksum that can be used to verify that the resource was downloaded correctly
- checksum.algo – an optional attribute that specifies the algorithm used to calculate the checksum, if not specified SHA-256 is assumed
- copyright – an optional attribute providing a human readable statement of copyright for the resource that is being downloaded
- description – an optional attribute providing a human readable description of the resource
- documentation – an optional attribute specifying a String typed attribute that indicates a URL where documentation on this resource may be accessed
- license – an optional attribute providing machine readable form of license information. See section 3.2.1.10 of the OSGi Core Specification for information on it's format
- name – an optional attribute providing a human readable name of the resource
- scm – an optional String typed attribute that specifies a URL where the source control management for the resource is located
- size – an optional long typed attribute that specifies the size of the resource in bytes
- source – an optional String typed attribute that specifies the URL where source code for a resource is located

5.3 Repository

Repositories are services published to the OSGi registry they may be backed by a range of different technologies, including static XML files, databases, carrier pigeons, etc.

This specification does not provide any implementations of a Repository these are left open, however some suggested repository implementations are:

- SimpleRepository – takes an array of Resource objects to form an inmemory repository
- FederatedRepository – takes an array of sub repositories
- XMLRepository – takes an XML file URL as a construction parameter
- Directory based repository – Takes a file URL and traverses a directory to build a repository from a set of resources

The API of the Repository is:

- `Collection<Capability> findProviders(Requirement requirement)` – Find any capabilities that match the supplied requirement.
- `URL getContent(Resource resource)` - Lookup the URL where the supplied resource may be accessed, if any. Successive calls to this method do not have to return the same value, this allows for mirroring behaviors to be built into a repository.

5.4 Downloading a resource

Whilst the osgi.content namespace attribute is a URI string the direct conversion of this to a URL is discouraged as it is up to the repository how the URI is encoded and this may not have a direct URL mapping.

Instead the following code snippets demonstrate how to load a resource from a remote repository. The first snippet below searches for new resources and uses the Repository.getContent method to find a URL where the resource may be retrieved.

```
void loadResources(Requirement req, Repository[] repositories, Resolver resolver)
{
  ResourceTrackingEnvironment env = new ResourceTrackingEnvironment(repositories);

  Map<Resource, List<Wire>> delta = resolver.resolve(env, req);

  for (Resource res : delta.keySet()) {
    Repository rep = env.getRepository(res);
    URL url = rep.getContent(res);
    download(res, url);
  }
}
```

This example shows a simple environment implementation that records where a resource is retrieved from using a HashMap based approach.

```
public class ResourceTrackingEnvironment implements Environment {
  private final Repository[] repositories;
  private final HashMap<Resource, Repository> tracked = new HashMap<Resource, Repository>();

  public ResourceTrackingEnvironment(Repository[] repositories) {
    this.repositories = repositories;
  }

  @Override
  public Collection<Capability> findProviders(Requirement requirement) {
    ArrayList<Capability> providers = new ArrayList<Capability>();
    for (Repository rep : repositories) {
      Collection<Capability> found = rep.findProviders(requirement);
      for (Capability cap : found) {
        Resource res = cap.getResource();
        tracked.put(res, rep);
        providers.add(cap);
      }
    }
    return providers;
  }

  public Repository getRepository(Resource res) {
    return tracked.get(res);
  }

  @Override
  public Map<Resource, List<Wire>> getWiring() {
```

```
    return Collections.emptyMap();
  }

  @Override
  public boolean isEffective(Requirement requirement) {
    return true;
  }
}
```

5.5 Environment

In order for a resolver to solve a set of requirements it needs an environment to work against. The environment supplies the existing wires between resources that the resolver needs to take into account and provides additional capabilities that match requirements. The environment is a Java interface that can proxy many different underlying implementations.

The Environment interface provides the following methods:

- `Collection<Capability> findProviders(Requirement requirement)` - Find any capabilities that match the supplied requirement. A resolver should use the iteration order or the returned capability collection to infer preference in the case where multiple capabilities match a requirement. Capabilities at the start of the iteration are implied to be preferred over capabilities at the end.

- `boolean isEffective(Requirement requirement)` - Test if a given requirement should be wired in a given resolve operation. If this method returns false then the resolver should ignore this requirement during this resolve operation. The primary use case for this is to test the `effective` directive on the requirement, though implementations are free to use this for any other purposes.

- `Map<Resource, List<Wire>> getWiring()` - An immutable map of wires between resources.

An environment may be used to encapsulate a set of policies for a resolution, examples of these policies include:

- Preferred Resources – The iteration order of the findProviders result can be used to influence the resolver to prefer certain resources. For example this may be used to implement alternate resolution strategies where it is not the latest version of resource that is resolved but one known to have been through QA testing.

- Singletons – An environment must only return one instance of a singleton bundle per resolve call, if there are many singletons available to an environment the choice of which singleton is left open to the implementer

- Execution Environment – the environment may be used to filter out any bundles that are not supported by the target framework execution environment

- Native code – the environment can be used to filter out any osgi bundles that are not supported by the host operating system.

5.6 Resolving

The resolver is a complicated process requiring difficult choices that likely require user intervention and/or policies. This includes but is not limited to:

- The addition of optional resources.
- Attachment of fragments
- Resolve time policy for version ranges
- Decisions related to licensing, bundle size, performance etc

The implementation of the Resolver object can provide these capabilities as it sees fit. The mechanism by which a client configures these capabilities is not defined here and is instead left up to implementations to provide an API or a management interface as they see fit. Though one potential mechanism is to use requirement directives to influence the resolver strategy.

The Repository resolver is in many ways similar to the Framework resolver. Implementations may therefore strive to use the same code. However, the problem that the Framework resolver solves is subtly different from what the Repository resolver solves. First, the Repository resolver is more generic; it handles more than packages and bundles. This is the reason for the generic requirement/capability model instead of using the manifest directly. Second, the Framework creates a wiring between a set of installed bundles. In contrast, the Repository resolver installs a set of bundles. Despite these subtle differences, the logic behind these resolvers is very similar and can clearly share implementation code.

The API of the Resolver is:

Map<Resource, List<Wire>> resolve(Environment environment, Collection<? extends Resource> mandatoryResources, Collection<? extends Resource> optionalResources) - Attempt to resolve the resources based on the specified environment and return any new resources and wires to the caller.

The resolver considers two groups of resources:

- Mandatory - any resource in the mandatory group must be resolved, a failure to satisfy any mandatory requirement for these resources will result in a ResolutionException
- Optional - any resource in the optional group may be resolved, a failure to satisfy a mandatory requirement for a resource in this group will not fail the overall resolution but no resources or wires will be returned for this resource.

5.6.1 Delta

The resolve method returns the delta between the start state defined by Environment.getWiring() and the end resolved state, i.e. only new resources and wires are included. To get the complete resolution the caller can merge the start state and the delta using something like the following:

```
Map<Resource, List<Wire>> delta = resolver.resolve(env, resources, null);

Map<Resource, List<Wire>> wiring = env.getWiring();


for (Map.Entry<Resource, List<Wire>> e : delta.entrySet()) {
```

```
  Resource res = e.getKey();

  List<Wire> newWires = e.getValue();


  List<Wire> currentWires = wiring.get(res);

  if (currentWires != null) {

    newWires.addAll(currentWires);

  }


  wiring.put(res, newWires);

}
```

If no requirements are effective then the result of a resolution is amap containing the resource mapped to an empty list of wires, for example:

```
new HashMap(resource, Collections.emptyList());
```

5.6.2 Consistency

For a given resolve operation the parameters to the resolve method should be considered immutable. This means that resources should have constant capabilities and requirements and an environment should return a consistent set of capabilities, wires and effective requirements. The behavior of the resolver is not defined if resources or the environment supply inconsistent information.

5.7 Code Examples

5.7.1 Install into framework

```
  public void installIntoFramework(Resource resource)

      throws IllegalStateException, InterruptedException, BundleException {

    EnvironmentBuilder envBuilder = new EnvironmentBuilder();

    envBuilder.addEnvironment(_ctx);

    Environment framework = envBuilder.buildEnvironment();

    try {

      // attempt to resolve

      Map<Resource, List<Wire>> delta = _resolver

          .resolve(framework, Collections.singleton(resource), null);
```

```
      // for all resolved bundle revisions install them

      for (Resource r : delta.keySet()) {

        String type = (String)
r.getIdentity().getAttributes().get(ResourceConstants.IDENTITY_TYPE_ATTRIBUTE);



        if (ResourceConstants.IDENTITY_TYPE_BUNDLE.equals(type)) {

          String location = (String)
r.getCapabilities(ContentNamespace.CAPABILITY).iterator().next().getAttributes().get(ContentN
amespace.CAPABILITY);

          try {

             // NOTE better to use ResourceTrackingEnvironment approach above

            _ctx.installBundle(URI.create(location).toURL().toString());

          } catch (MalformedURLException e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

          }

        }

        else {

          System.err.println("No action defined for " + type);

        }

      }

    } catch (ResolutionException e) {

      System.out.println("Failure :(");

    }

  }
```

5.7.2 Resolve into an “empty” framework

```
  public void resolveIntoEmpty(Resource resource)

      throws InterruptedException {

    // build empty environment

    EnvironmentBuilder envBuilder = new EnvironmentBuilder();
```

```
    envBuilder.addRepositories(_ctx);

    Environment framework = envBuilder.buildEnvironment();

    // attempt to resolve
    _resolver.resolve(framework, Collections.singleton(resource), null);
  }
```

5.7.3 Resolve self into empty framework

```
  public void resolveSelfIntoEmptyFramework() throws InterruptedException {
    // build empty environment
    EnvironmentBuilder envBuilder = new EnvironmentBuilder();
    envBuilder.addRepositories(_ctx);
    Environment framework = envBuilder.buildEnvironment();

    // require the resource
    BundleReference ref = (BundleReference) getClass().getClassLoader();
    BundleRevision rev = ref.getBundle().adapt(BundleRevision.class);

    // attempt to resolve
    _resolver.resolve(framework, Collections.singleton(rev), null);
  }
```

5.7.4 Resolve a package into framework

```
  public void resolveAPackageIntoFramework() throws InterruptedException {
    // build framework environment
    EnvironmentBuilder envBuilder = new EnvironmentBuilder();
    envBuilder.addEnvironment(_ctx);
    Environment framework = envBuilder.buildEnvironment();
```

```
    // require the resource
    ResourceBuilder resBuilder = new ResourceBuilder();
    resBuilder.addRequirement("osgi.wiring.package",
        "(osgi.wiring.package=com.foo.api)");

    // attempt to resolve
    _resolver.resolve(framework, Collections.singleton(resBuilder.build()), null);
  }
```

5.7.5 Resolve Declarative Services provider into framework

```
  public void resolveDSProviderIntoFramework() throws InterruptedException {
    EnvironmentBuilder envBuilder = new EnvironmentBuilder();
    envBuilder.addEnvironment(_ctx);
    Environment framework = envBuilder.buildEnvironment();

    ResourceBuilder resBuilder = new ResourceBuilder();
    resBuilder.addRequirement("osgi.service.ds",
        "(provider=felix)");

    // attempt to resolve
    _resolver.resolve(framework, Collections.singleton(resBuilder.build()), null);
  }
```

5.8 XML Schema

This specification defines an XML schema to represent a repository, the purpose of this xml schema is for interchange between different vendors.

5.8.1 Namespace

The XML namespace is:

http://www.osgi.org/xmlns/obr/v1.0.0

```
<obr:repository name='Untitled' time='20051210072623.031'
```

```
    xmlns:obr="http://www.osgi.org/xmlns/obr/v1.0.0">

    …
```

5.8.2 The XML Structure

The following BNF describes the element structure of the XML file:

```
repository ::= (referral | resource) *

resource ::= require * capability *

require ::= attribute * directive *

capability ::= attribute * directive *
```

The xml schema is deliberately left open to extension via the use of <any/> and <anyAttribute/> elements so other schemas may expand on this model. A must-understand attribute should be used in extension schemas to define whether the extra elements constitute required or optional extension behaviour.

5.8.3 Repository

The <repository> tag is the outer tag of the XML document. It must contains the following attributes:

1. *name* – The name of the repository. The name may contain spaces and punctuation and is for informational purposes only.

2. *increment* – A long value counter to indicate the state of the repository, clients can use this to check if there have been any changes to the resources contained in this repository

The repository element can contain referral and resource elements.

```
<obr:repository name='Untitled' increment='1'
  xmlns:obr="http://www.osgi.org/xmlns/scr/v1.0.0">
</obr:repository>
```

5.8.4 Referral

A referral points to another repository XML file. The purpose of this element is to create a federation of repositories that can be accessed as a single repository. The referral element can have the following attributes:

1. *depth* – The depth of referrals this repository acknowledges. If the depth is 1, the referred repository must included but it must not follow any referrals from the referred repository. If the depth is more than one, referrals must be included up to the given depth. Depths of referred repositories must also be obeyed. For example, if the top repository specifies a depth of 5, and the 3 level has a depth of 1, then a repository included on level 5 must be discarded, even though the top repository would have allowed it. If this attribute is missing then an infinite depth is assumed.

2. *url* – The URL to the referred repository. The URL can be absolute or relative from the given repository's URL.

For example:

```
<referral depth="1" url=http://www.aqute.biz/bundles/repository.xml/>
```

5.8.5 Resource

The <resource> element describes a general resource with requirements, and capabilities. All resources must contain an osgi.identity capability.

5.8.6 Require

The <require> element describes one of the requirements that the enclosing resource has on its environment. A requirement is of a specific named type and contains a filter that is applied to all capabilities of the given type. Therefore, the requirement element has the following attributes:

- namespace – The namespace of the requirement. The filter must only be applied to capabilities that have the same name space

A requirement may contain zero or more attributes or directives, for example:

```
<require namespace='osgi.wiring.package'>

  <directive name='filter' value='(&amp;
(osgi.wiring.package=org.osgi.test.cases.util)(version&gt;=1.0.0))' />

</require>
```

This example requires that there is at least one exporter of the org.osgi.test.cases.util package with a version higher than 1.1.0

5.8.7 Capability

The capability element is a named set of attributes and directives. A capability can be used to resolve a requirement if the resource is included. A capability has the following attribute:

- *namespace – The namespace of the capability*

Only requirements with the same namespace must be able to match this capability.

The capability can contain two elements, attribute and directive.

5.8.8 Directives

Directives are empty xml elements which have the following attributes:

- name – The name of the directive
- value – The value of the directive (always a String)

5.8.9 Attributes

Attributes are empty xml elements which have the following attributes:

- name – The name of the property
- value – The value of the property
- type – The type of the property. This must be one of:

- string – A string value, which is the default.
- version – An OSGi version as implemented in the OSGi Version class.
- long – A java Long value
- double – A Java Double value
- list – A comma separated list of values. White space must be discarded, the values can not contain commas.

The following example shows a package export:

```
    <capability namespace='osgi.wiring.package'>
      <attribute value='org.eclipse.core.internal.resources'
name='osgi.wiring.package'/>
      <attribute value='0.0.0' type='version' name='version'/>
      <directive value='true' name='x-internal'/>
    </capability>
```

5.9 osgi.content URIs

The URI value of the osgi.content capability should be defined be relative to the repository xml as this allows simple mirroring behaviours. However absolute URI's are also supported. In order to resolve the underlying URL for an osgi.content URI defined in an OBR xml file a Repository implementation may use a mechanism something like the following:

```
public class XMLRepository implements Repository {

  private final URI xml;

  public XMLRepository(URI xml) {

    this.xml = xml;

  }



  @Override

  public URL getContent(Resource resource) {

    List<Capability> c = resource.getCapabilities("osgi.content");

    if (c.isEmpty()) return null;

    String uriStr = (String) c.get(0).getAttributes().get("osgi.content");

    try {

      return xml.resolve(uriStr).toURL();

    } catch (MalformedURLException e) {
```

```
      return null;

    }

  }


  @Override

  public Collection<Capability> findProviders(Requirement requirement) {

    // ...

  }

}
```

5.10 Sample XML File

```
<repository name='Untitled'
            increment='1'
            targetNamespace='http://www.osgi.org/xmlns/obr/v1.0.0'>
  <resource>
    <capability namespace='osgi.identity'>
      <attribute value='org.osgi.test.cases.tracker' name='osgi.identity' />
      <attribute value='1.0.0' name='version' type='version' />
      <attribute value='osgi.bundle' name='type'/>
    </capability>

    <capability namespace='osgi.content'>
      <attribute value='org.osgi.test.cases.tracker-3.0.0.jar' name='osgi.content'>
      <attribute value='4405' name='size' type='long' />
      <attribute value='http://www.osgi.org' name='documentation'/>
      <attribute value='Copyright (c) OSGi Alliance (2000, 2011). All Rights
Reserved.' name='copyright' />
    </capability>

    <capability namespace='osgi.wiring.bundle'>
      <attribute value='org.osgi.test.cases.tracker' name='osgi.wiring.bundle'/>
      <attribute value='1' name='manifest.version'/>
      <attribute value='3.0.0' name='version' type='version'/>
    </capability>

    <capability namespace='osgi.wiring.package'>
      <attribute value='org.osgi.test.cases.tracker' name='osgi.wiring.package' />
      <attribute value='0.0.0' name='version' type='version'/>
    </capability>

    <require namespace='osgi.wiring.package'>
      <directive value='(&amp;(osgi.wiring.package=org.osgi.test.cases.util)
(version&gt;=1.1.0))' name='filter' />
    </require>
  </resource>
</repository>
```

5.10.1 Bundle Wiring Mapping

This RFC uses the same mapping of capabilties and attributes as defined in section 7.4 of the Core specification

5.10.2 Bundle-ExecutionEnvironment

The Bundle Execution Environment header is mapped to a requirement. The capabilities of this requirement must be set by the environment. Each support environment is an element of a multi-valued property called ‘ee’ in a ‘ee’capability.

The filter must assert on ‘ee’ with the defined names for ee’s. For example, if the bundle can run on J2SE 1.4:

```
<require namespace="osgi.wiring.ee" filter="(|(ee=J2SE-1.4))"/>
```

This requirement is UNARY.

6 Javadoc

OSGi Javadoc

9/8/11 4:00 PM

Package Summary		Page
org.osgi.framework.resource	Framework Resource Package Version 1.0.	32
org.osgi.service.repository	Repository Package Version 1.0.	47
org.osgi.service.resolver	Resolver Package Version 1.0.	52

Package org.osgi.framework.resource

Framework Resource Package Version 1.0.

See:
Description

Interface Summary		Page
Capability	A capability that has been declared from a `Resource`.	33
Requirement	A requirement that has been declared from a `Resource` .	35
Resource	A resource is the representation of a uniquely identified and typed data.	37
Wire	A wire connecting a `Capability` to a `Requirement`.	45

Class Summary		Page
ResourceConstants	Defines standard names for the attributes, directives and name spaces for resources, capabilities and requirements.	38

Package org.osgi.framework.resource Description

Framework Resource 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.framework.resource; version="[1.0,2.0)"
```

Interface Capability

org.osgi.framework.resource

```
public interface Capability
```

A capability that has been declared from a Resource.

Version:
$Id: 5597c6dd01b34d5e3a2ec05f83b6f895f114d45a $
ThreadSafe

Method Summary		Page
Map<String,Object>	**getAttributes**() Returns the attributes of this capability.	34
Map<String,String>	**getDirectives**() Returns the directives of this capability.	33
String	**getNamespace**() Returns the name space of this capability.	33
Resource	**getResource**() Returns the resource declaring this capability.	34

Method Detail

getNamespace

```
String getNamespace()
```

Returns the name space of this capability.

Returns:
The name space of this capability.

---

getDirectives

```
Map<String,String> getDirectives()
```

Returns the directives of this capability.

Only the following list of directives have specified semantics:

- effective
- uses
- mandatory - only recognized for the osgi.wiring.bundle and osgi.wiring.package name spaces.
- exclude - only recognized for the osgi.wiring.package name space.
- include - only recognized for the osgi.wiring.package name space.

All other directives have no specified semantics and are considered extra user defined information. The OSGi Alliance reserves the right to extend the set of directives which have specified semantics.

Returns:
An unmodifiable map of directive names to directive values for this capability, or an empty map if this capability has no directives.

---

getAttributes

```
Map<String,Object> getAttributes()
```

Returns the attributes of this capability.

Returns:
An unmodifiable map of attribute names to attribute values for this capability, or an empty map if this capability has no attributes.

---

getResource

```
Resource getResource()
```

Returns the resource declaring this capability.

Returns:
The resource declaring this capability.

Interface Requirement

org.osgi.framework.resource

```
public interface Requirement
```

A requirement that has been declared from a Resource .

Version:
$Id: 46f600c32e563cfe617fb94ab4e308f5be2b9217 $
ThreadSafe

Method Summary		Page
Map<String,Object>	**getAttributes**() Returns the attributes of this requirement.	36
Map<String,String>	**getDirectives**() Returns the directives of this requirement.	35
String	**getNamespace**() Returns the name space of this requirement.	35
Resource	**getResource**() Returns the resource declaring this requirement.	36
boolean	**matches**(Capability capability) Returns whether the specified capability matches this requirement.	36

Method Detail

getNamespace

```
String getNamespace()
```

Returns the name space of this requirement.

Returns:
The name space of this requirement.

---

getDirectives

```
Map<String,String> getDirectives()
```

Returns the directives of this requirement.

Only the following list of directives have specified semantics:

- effective
- filter
- cardinality
- resolution
- visibility - only recognized for the osgi.wiring.bundle name space.

All other directives have no specified semantics and are considered extra user defined information. The OSGi Alliance reserves the right to extend the set of directives which have specified semantics.

Returns:
An unmodifiable map of directive names to directive values for this requirement, or an empty map if this requirement has no directives.

getAttributes

`Map<String,Object>` **`getAttributes`**`()`

Returns the attributes of this requirement.

Requirement attributes have no specified semantics and are considered extra user defined information.

Returns:
An unmodifiable map of attribute names to attribute values for this requirement, or an empty map if this requirement has no attributes.

getResource

`Resource` **`getResource`**`()`

Returns the resource declaring this requirement.

Returns:
The resource declaring this requirement.

matches

`boolean` **`matches`**`(Capability capability)`

Returns whether the specified capability matches this requirement.

A capability matches this requirement when all of the following are true:

- The specified capability has the same `name space` as this requirement.
- The filter specified by the `filter` directive of this requirement matches the `attributes of the specified capability`.
- The standard capability `directives` that influence matching and that apply to the name space are satisfied. See the capability `mandatory` directive.

Parameters:
`capability` - The capability to match to this requirement.
Returns:
`true` if the specified capability matches this this requirement; `false` otherwise.

Interface Resource

org.osgi.framework.resource

```
public interface Resource
```

A resource is the representation of a uniquely identified and typed data. A resources can be wired together via capabilities and requirements.

Version:
$Id: 56916cb2597067bdf63e757c078787eccebf3953 $
ThreadSafe

Method Summary		Page
List<Capability>	**getCapabilities**(String namespace) Returns the capabilities declared by this resource.	37
List<Requirement>	**getRequirements**(String namespace) Returns the requirements declared by this bundle resource.	37

Method Detail

getCapabilities

```
List<Capability> getCapabilities(String namespace)
```

Returns the capabilities declared by this resource.

Parameters:
`namespace` - The name space of the declared capabilities to return or `null` to return the declared capabilities from all name spaces.
Returns:
A list containing a snapshot of the declared `Capability`s, or an empty list if this resource declares no capabilities in the specified name space.

---

getRequirements

```
List<Requirement> getRequirements(String namespace)
```

Returns the requirements declared by this bundle resource.

Parameters:
`namespace` - The name space of the declared requirements to return or `null` to return the declared requirements from all name spaces.
Returns:
A list containing a snapshot of the declared `Requirement` s, or an empty list if this resource declares no requirements in the specified name space.

Class ResourceConstants

org.osgi.framework.resource

```
java.lang.Object
  └ org.osgi.framework.resource.ResourceConstants
```

```
final public class ResourceConstants
extends Object
```

Defines standard names for the attributes, directives and name spaces for resources, capabilities and requirements.

The values associated with these keys are of type `String`, unless otherwise indicated.

Version:
$Id: ab8d42db8d410c81bccb93effa990bed1f4414df $
Immutable

Field Summary		Page
static String	**CAPABILITY_EFFECTIVE_DIRECTIVE** A capability `directive` used to specify the effective time for the capability.	44
static String	**CAPABILITY_EXCLUDE_DIRECTIVE** A capability `directive` used to specify the comma separated list of classes which must not be allowed to be exported.	44
static String	**CAPABILITY_INCLUDE_DIRECTIVE** A capability `directive` used to specify the comma separated list of classes which must be allowed to be exported.	44
static String	**CAPABILITY_MANDATORY_DIRECTIVE** A capability `directive` used to specify the comma separated list of mandatory attributes which must be specified in the `filter` of a requirement in order for the capability to match the requirement.	44
static String	**CAPABILITY_USES_DIRECTIVE** A capability `directive` used to specify the comma separated list of `package` names a capability uses.	43
static String	**EFFECTIVE_ACTIVE** A directive value identifying a `capability` or `requirement` that is effective at active time.	42
static String	**EFFECTIVE_RESOLVE** A directive value identifying a `capability` or `requirement` that is effective at resolve time.	42
static String	**IDENTITY_NAMESPACE** Name space for the identity capability.	39
static String	**IDENTITY_SINGLETON_DIRECTIVE** An `identity` capability `directive` identifying if the resource is a singleton.	40
static String	**IDENTITY_TYPE_ATTRIBUTE** An `identity` capability attribute identifying the resource type.	40
static String	**IDENTITY_TYPE_BUNDLE** An `identity` capability `type` attribute value identifying the resource type as an OSGi bundle.	40
static String	**IDENTITY_TYPE_FRAGMENT** An `identity` capability `type` attribute value identifying the resource type as an OSGi fragment.	40
static String	**IDENTITY_TYPE_UNKNOWN** An `identity` capability `type` attribute value identifying the resource type as unknown.	40
static String	**IDENTITY_VERSION_ATTRIBUTE** An `identity` capability attribute identifying the `version` of the resource.	39

static String	**REQUIREMENT_CARDINALITY_DIRECTIVE** A requirement directive used to specify the cardinality for a requirement.	43
static String	**REQUIREMENT_CARDINALITY_MULTIPLE** A directive value identifying a multiple cardinality type.	43
static String	**REQUIREMENT_CARDINALITY_SINGULAR** A directive value identifying a singular cardinality type.	43
static String	**REQUIREMENT_EFFECTIVE_DIRECTIVE** A requirement directive used to specify the effective time for the requirement.	42
static String	**REQUIREMENT_FILTER_DIRECTIVE** A requirement directive used to specify a capability filter.	41
static String	**REQUIREMENT_RESOLUTION_DIRECTIVE** A requirement directive used to specify the resolution type for a requirement.	41
static String	**REQUIREMENT_RESOLUTION_MANDATORY** A directive value identifying a mandatory requirement resolution type.	41
static String	**REQUIREMENT_RESOLUTION_OPTIONAL** A directive value identifying an optional requirement resolution type.	42
static String	**REQUIREMENT_VISIBILITY_DIRECTIVE** A requirement directive used to specify the visibility type for a requirement.	42
static String	**REQUIREMENT_VISIBILITY_PRIVATE** A directive value identifying a private visibility type.	43
static String	**REQUIREMENT_VISIBILITY_REEXPORT** A directive value identifying a reexport visibility type.	43
static String	**WIRING_BUNDLE_NAMESPACE** Name space for bundle capabilities and requirements.	41
static String	**WIRING_HOST_NAMESPACE** Name space for host capabilities and requirements.	41
static String	**WIRING_PACKAGE_NAMESPACE** Name space for package capabilities and requirements.	40

Field Detail

IDENTITY_NAMESPACE

```
public static final String IDENTITY_NAMESPACE = "osgi.identity"
```

Name space for the identity capability. Each resource provides exactly one† identity capability that can be used to identify the resource. For identity capability attributes the following applies:

- The osgi.identity attribute contains the symbolic name of the resource.
- The version attribute contains the org.osgi.framework.Version of the resource.
- The type attribute contains the resource type.

A resource with a symbolic name provides exactly one † identity capability.

For a revision with a symbolic name the wiring for the revision provides exactly one† identity capability.

† A resource with no symbolic name must not provide an identity capability.

IDENTITY_VERSION_ATTRIBUTE

```
public static final String IDENTITY_VERSION_ATTRIBUTE = "version"
```

An identity capability attribute identifying the version of the resource. This attribute must be set to a value of type org.osgi.framework.Version. If the resource has no version then the value 0.0.0 must be used for the attribute.

IDENTITY_TYPE_ATTRIBUTE

```
public static final String IDENTITY_TYPE_ATTRIBUTE = "type"
```

An identity capability attribute identifying the resource type. This attribute must be set to a value of type String. if the resource has no type then the value unknown must be used for the attribute.

IDENTITY_TYPE_BUNDLE

```
public static final String IDENTITY_TYPE_BUNDLE = "osgi.bundle"
```

An identity capability type attribute value identifying the resource type as an OSGi bundle.

IDENTITY_TYPE_FRAGMENT

```
public static final String IDENTITY_TYPE_FRAGMENT = "osgi.fragment"
```

An identity capability type attribute value identifying the resource type as an OSGi fragment.

IDENTITY_TYPE_UNKNOWN

```
public static final String IDENTITY_TYPE_UNKNOWN = "unknown"
```

An identity capability type attribute value identifying the resource type as unknown.

IDENTITY_SINGLETON_DIRECTIVE

```
public static final String IDENTITY_SINGLETON_DIRECTIVE = "singleton"
```

An identity capability directive identifying if the resource is a singleton. A String value of "true" indicates the resource is a singleton; any other value or null indicates the resource is not a singleton.

WIRING_PACKAGE_NAMESPACE

```
public static final String WIRING_PACKAGE_NAMESPACE = "osgi.wiring.package"
```

Name space for package capabilities and requirements. For capability attributes the following applies:

- The osgi.wiring.package attribute contains the name of the package.
- The version attribute contains the the org.osgi.framework.Version of the package if one is specified or org.osgi.framework.Version.emptyVersion if not specified.
- The bundle-symbolic-name attribute contains the symbolic name of the resource providing the package if one is specified.
- The bundle-version attribute contains the org.osgi.framework.Version of resource providing the package if one is specified or org.osgi.framework.Version.emptyVersion if not specified.
- All other attributes are of type String and are used as arbitrary matching attributes for the capability.

A resource provides zero or more package capabilities (this is, exported packages) and requires zero or more package requirements (that is, imported packages).

WIRING_BUNDLE_NAMESPACE

```
public static final String WIRING_BUNDLE_NAMESPACE = "osgi.wiring.bundle"
```

Name space for bundle capabilities and requirements. For capability attributes the following applies:

- The osgi.wiring.bundle attribute contains the symbolic name of the bundle.
- The `bundle-version` attribute contains the `org.osgi.framework.Version` of the bundle if one is specified or `org.osgi.framework.Version.emptyVersion` if not specified.
- All other attributes are of type `String` and are used as arbitrary matching attributes for the capability.

A non-fragment resource with the `osgi.bundle` type `identity` provides exactly one † bundle `capability` (that is, the bundle can be required by another bundle). A fragment resource with the `osgi.fragment` type `identity` must not declare a bundle capability. A resource requires zero or more bundle `requirements` (that is, required bundles).

† A resource with no symbolic name must not provide a bundle capability.

WIRING_HOST_NAMESPACE

```
public static final String WIRING_HOST_NAMESPACE = "osgi.wiring.host"
```

Name space for host capabilities and requirements. For capability attributes the following applies:

- The osgi.wiring.host attribute contains the symbolic name of the bundle.
- The `bundle-version` attribute contains the `org.osgi.framework.Version` of the bundle if one is specified or `org.osgi.framework.Version.emptyVersion` if not specified.
- All other attributes are of type `String` and are used as arbitrary matching attributes for the capability.

A non-fragment resource with the with the `osgi.bundle` type `identity` provides zero or one † host `capabilities`. A fragment resource with the `osgi.fragment` type `identity` must not declare a host capability and must `declare` exactly one host requirement.

† A resource with no bundle symbolic name must not provide a host capability.

REQUIREMENT_FILTER_DIRECTIVE

```
public static final String REQUIREMENT_FILTER_DIRECTIVE = "filter"
```

A requirement `directive` used to specify a capability filter. This filter is used to match against a capability's `attributes`.

REQUIREMENT_RESOLUTION_DIRECTIVE

```
public static final String REQUIREMENT_RESOLUTION_DIRECTIVE = "resolution"
```

A requirement `directive` used to specify the resolution type for a requirement. The default value is `mandatory`.

See Also:
`mandatory`, `optional`

REQUIREMENT_RESOLUTION_MANDATORY

```
public static final String REQUIREMENT_RESOLUTION_MANDATORY = "mandatory"
```

A directive value identifying a mandatory requirement resolution type. A mandatory resolution type indicates that the requirement must be resolved when the resource is resolved. If such requirement cannot be resolved, the resource fails to resolve.

See Also:
REQUIREMENT_RESOLUTION_DIRECTIVE

REQUIREMENT_RESOLUTION_OPTIONAL

```
public static final String REQUIREMENT_RESOLUTION_OPTIONAL = "optional"
```

A directive value identifying an optional requirement resolution type. An optional resolution type indicates that the requirement is optional and the resource may be resolved without requirement being resolved.

See Also:
REQUIREMENT_RESOLUTION_DIRECTIVE

REQUIREMENT_EFFECTIVE_DIRECTIVE

```
public static final String REQUIREMENT_EFFECTIVE_DIRECTIVE = "effective"
```

A requirement directive used to specify the effective time for the requirement. The default value is resolve.

See Also:
resolve, active

EFFECTIVE_RESOLVE

```
public static final String EFFECTIVE_RESOLVE = "resolve"
```

A directive value identifying a capability or requirement that is effective at resolve time. Capabilities and requirements with an effective time of resolve are the only capabilities which are processed while resolving a resource.

See Also:
REQUIREMENT_EFFECTIVE_DIRECTIVE, CAPABILITY_EFFECTIVE_DIRECTIVE

EFFECTIVE_ACTIVE

```
public static final String EFFECTIVE_ACTIVE = "active"
```

A directive value identifying a capability or requirement that is effective at active time. Capabilities and requirements with an effective time of active are ignored while resolving a resource.

See Also:
REQUIREMENT_EFFECTIVE_DIRECTIVE, CAPABILITY_EFFECTIVE_DIRECTIVE

REQUIREMENT_VISIBILITY_DIRECTIVE

```
public static final String REQUIREMENT_VISIBILITY_DIRECTIVE = "visibility"
```

A requirement directive used to specify the visibility type for a requirement. The default value is private. This directive must only be used for requirements with the require bundle name space.

See Also:
private, reexport

REQUIREMENT_VISIBILITY_PRIVATE

```
public static final String REQUIREMENT_VISIBILITY_PRIVATE = "private"
```

A directive value identifying a private visibility type. A private visibility type indicates that any packages that are exported by the required bundle are not made visible on the export signature of the requiring bundle .

See Also:
REQUIREMENT_VISIBILITY_DIRECTIVE

REQUIREMENT_VISIBILITY_REEXPORT

```
public static final String REQUIREMENT_VISIBILITY_REEXPORT = "reexport"
```

A directive value identifying a reexport visibility type. A reexport visibility type indicates any packages that are exported by the required bundle are re-exported by the requiring bundle.

REQUIREMENT_CARDINALITY_DIRECTIVE

```
public static final String REQUIREMENT_CARDINALITY_DIRECTIVE = "cardinality"
```

A requirement directive used to specify the cardinality for a requirement. The default value is singular.

See Also:
multiple, singular

REQUIREMENT_CARDINALITY_MULTIPLE

```
public static final String REQUIREMENT_CARDINALITY_MULTIPLE = "multiple"
```

A directive value identifying a multiple cardinality type.

REQUIREMENT_CARDINALITY_SINGULAR

```
public static final String REQUIREMENT_CARDINALITY_SINGULAR = "singular"
```

A directive value identifying a singular cardinality type.

CAPABILITY_USES_DIRECTIVE

```
public static final String CAPABILITY_USES_DIRECTIVE = "uses"
```

A capability directive used to specify the comma separated list of package names a capability uses.

CAPABILITY_EFFECTIVE_DIRECTIVE

```
public static final String CAPABILITY_EFFECTIVE_DIRECTIVE = "effective"
```

A capability directive used to specify the effective time for the capability. The default value is resolve.

See Also:
resolve, active

CAPABILITY_MANDATORY_DIRECTIVE

```
public static final String CAPABILITY_MANDATORY_DIRECTIVE = "mandatory"
```

A capability directive used to specify the comma separated list of mandatory attributes which must be specified in the filter of a requirement in order for the capability to match the requirement. This directive must only be used for capabilities with the package, bundle, or host name space.

CAPABILITY_INCLUDE_DIRECTIVE

```
public static final String CAPABILITY_INCLUDE_DIRECTIVE = "include"
```

A capability directive used to specify the comma separated list of classes which must be allowed to be exported. This directive must only be used for capabilities with the package name space.

CAPABILITY_EXCLUDE_DIRECTIVE

```
public static final String CAPABILITY_EXCLUDE_DIRECTIVE = "exclude"
```

A capability directive used to specify the comma separated list of classes which must not be allowed to be exported. This directive must only be used for capabilities with the package name space.

Interface Wire

org.osgi.framework.resource

```
public interface Wire
```

A wire connecting a Capability to a Requirement.

Version:
$Id: 87d274ba376ed4f04b88b771d9c948998b211f5c $
ThreadSafe

Method Summary		Page
Capability	**getCapability**() Returns the Capability for this wire.	45
Resource	**getProvider**() Return the providing resource of the capability.	45
Requirement	**getRequirement**() Return the Requirement for this wire.	45
Resource	**getRequirer**() Return the requiring resource of the requirement.	46

Method Detail

getCapability

Capability **getCapability**()

Returns the Capability for this wire.

Returns:
The Capability for this wire.

---

getRequirement

Requirement **getRequirement**()

Return the Requirement for this wire.

Returns:
The Requirement for this wire.

---

getProvider

Resource **getProvider**()

Return the providing resource of the capability.

The resource returned may differ from the resource referenced by the capability.

Returns:
the providing resource.

getRequirer

Resource **getRequirer**()

Return the requiring resource of the requirement.

The resource returned my differ from the resource referenced by the requirement

Returns:
the requiring resource.

Package org.osgi.service.repository

Repository Package Version 1.0.

See:
Description

Interface Summary		Page
Repository	Represents a repository that contains `resources`.	51

Class Summary		Page
ContentNamespace	Constants for use in the "osgi.content" namespace.	48

Package org.osgi.service.repository Description

Repository Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.repository; version="[1.0,1.1)"
```

Class ContentNamespace

org.osgi.service.repository

```
java.lang.Object
  └ org.osgi.service.repository.ContentNamespace
```

```
final public class ContentNamespace
extends Object
```

Constants for use in the "osgi.content" namespace. This namespace is used to locate content via the Repository.findProviders(Requirement) method.

Field Summary		Page
String[]	**ATTRIBUTES** All attributes defined in this interface	50
String	**CAPABILITY** Namespace of the content capability	48
String	**CHECKSUM_ALGO_ATTRIBUTE** The checksum algorithm used to calculate the CHECKSUM_ATTRIBUTE if not specified this is assumed to be SHA-256 - TODO need default?	49
String	**CHECKSUM_ATTRIBUTE** Checksum attribute of a resource	48
String	**COPYRIGHT_ATTRIBUTE** A copyright statement for the resource	49
String	**DESCRIPTION_ATTRIBUTE** A human readable description of this resource	49
String	**DOCUMENTATION_URL_ATTRIBUTE** A URL where documentation for this resource can be accessed	49
String	**LICENSE_ATTRIBUTE** Provides an optional machine readable form of license information.	49
String	**SCM_URL_ATTRIBUTE** A URL where source control management for this resource is located	49
String	**SIZE_ATTRIBUTE** The size of this resource in bytes.	49
String	**SOURCE_URL_ATTRIBUTE** A URL where source code for this resource is located	49

Field Detail

CAPABILITY

```
public final String CAPABILITY = "osgi.content"
```

Namespace of the content capability

CHECKSUM_ATTRIBUTE

```
public final String CHECKSUM_ATTRIBUTE = "checksum"
```

Checksum attribute of a resource

CHECKSUM_ALGO_ATTRIBUTE

```
public final String CHECKSUM_ALGO_ATTRIBUTE = "checksumAlgo"
```

The checksum algorithm used to calculate the CHECKSUM_ATTRIBUTE if not specified this is assumed to be SHA-256 - TODO need default?

COPYRIGHT_ATTRIBUTE

```
public final String COPYRIGHT_ATTRIBUTE = "copyright"
```

A copyright statement for the resource

DESCRIPTION_ATTRIBUTE

```
public final String DESCRIPTION_ATTRIBUTE = "description"
```

A human readable description of this resource

DOCUMENTATION_URL_ATTRIBUTE

```
public final String DOCUMENTATION_URL_ATTRIBUTE = "documentation"
```

A URL where documentation for this resource can be accessed

LICENSE_ATTRIBUTE

```
public final String LICENSE_ATTRIBUTE = "license"
```

Provides an optional machine readable form of license information. See section 3.2.1.10 of the OSGi Core Specification for information on it's usage.

SCM_URL_ATTRIBUTE

```
public final String SCM_URL_ATTRIBUTE = "scm"
```

A URL where source control management for this resource is located

---

SIZE_ATTRIBUTE

```
public final String SIZE_ATTRIBUTE = "size"
```

The size of this resource in bytes.

SOURCE_URL_ATTRIBUTE

```
public final String SOURCE_URL_ATTRIBUTE = "source"
```

A URL where source code for this resource is located

ATTRIBUTES

```
public final String[] ATTRIBUTES
```

All attributes defined in this interface

Interface Repository

org.osgi.service.repository

```
public interface Repository
```

Represents a repository that contains resources.

Repositories may be registered as services and may be used as inputs to an Environment.findProviders(Requirement) operation.

Repositories registered as services may be filtered using standard service properties.

Version:
$Id: e08240df0c2b794b1e2e80cf8f2ef5c18a22adee $
ThreadSafe

Method Summary		Page
Collection<Capability>	**findProviders**(Requirement requirement) Find any capabilities that match the supplied requirement.	51
URL	**getContent**(Resource resource) Lookup the URL where the supplied resource may be accessed, if any.	51

Method Detail

findProviders

```
Collection<Capability> findProviders(Requirement requirement)
```

Find any capabilities that match the supplied requirement.

Parameters:
requirement - The requirement that should be matched
Returns:
A collection of capabilities that match the supplied requirement
Throws:
NullPointerException - if the requirement is null

---

getContent

```
URL getContent(Resource resource)
```

Lookup the URL where the supplied resource may be accessed, if any.

Successive calls to this method do not have to return the same value this allows for mirroring behaviors to be built into a repository.

Parameters:
resource - - The resource whose content is desired.
Returns:
The URL for the supplied resource or null if this resource has no binary content or is not accessible for any reason
Throws:
NullPointerException - if the resource is null

Package org.osgi.service.resolver

Resolver Package Version 1.0.

See:
Description

Interface Summary		Page
Environment	An environment provides options and constraints to the potential solution of a `Resolver.resolve(Environment, Collection, Collection)` operation.	53
Resolver	A resolver is a service interface that can be used to find resolutions for specified `resources` based on a supplied `Environment`.	57

Exception Summary		Page
ResolutionException	Indicates failure to resolve a set of requirements.	55

Package org.osgi.service.resolver Description

Resolver Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.resolver; version="[1.0,1.1)"
```

Interface Environment

org.osgi.service.resolver

```
public interface Environment
```

An environment provides options and constraints to the potential solution of a Resolver.resolve(Environment, Collection, Collection) operation.

Environments:

- Provide capabilities that the Resolver can use to satisfy requirements via the findProviders(Requirement) method
- Constrain solutions via the getWiring() method. A wiring consists of a map of existing resources to wires.
- Filter transitive requirements that are brought in as part of a resolve operation via the isEffective(Requirement).

An environment may be used to provide capabilities via local resources and/or remote repositories.

A resolver may call the findProviders(Requirement), isEffective(Requirement) and getWiring() method any number of times during a resolve using any thread. Environments may also be shared between several resolvers. As such implementors should ensure that this class is properly synchronized.

ThreadSafe

Method Summary		Page
Collection<Capability>	**findProviders**(Requirement requirement) Find any capabilities that match the supplied requirement.	53
Map<Resource,List<Wire>>	**getWiring**() An immutable map of wires between revisions.	54
boolean	**isEffective**(Requirement requirement) Test if a given requirement should be wired in a given resolve operation.	53

Method Detail

findProviders

```
Collection<Capability> findProviders(Requirement requirement)
```

Find any capabilities that match the supplied requirement.

A resolver should use the iteration order or the returned capability collection to infer preference in the case where multiple capabilities match a requirement. Capabilities at the start of the iteration are implied to be preferred over capabilities at the end.

Parameters:
requirement - the requirement that a resolver is attempting to satisfy
Returns:
an collection of capabilities that match the supplied requirement
Throws:
NullPointerException - if the requirement is null

---

isEffective

```
boolean isEffective(Requirement requirement)
```

Test if a given requirement should be wired in a given resolve operation. If this method returns false then the resolver should ignore this requirement during this resolve operation.

The primary use case for this is to test the `effective` directive on the requirement, though implementations are free to use this for any other purposes.

Parameters:
`requirement` - the Requirement to test
Returns:
true if the requirement should be considered as part of this resolve operation
Throws:
`NullPointerException` - if requirement is null

---

getWiring

`Map<Resource,List<Wire>> **getWiring**()`

An immutable map of wires between revisions. Multiple calls to this method for the same environment object must result in the same set of wires.

Returns:
the wires already defined in this environment

Class ResolutionException

org.osgi.service.resolver

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ org.osgi.service.resolver.ResolutionException
```

All Implemented Interfaces:
Serializable

```
public class ResolutionException
extends RuntimeException
```

Indicates failure to resolve a set of requirements.

If a resolution failure is caused by a missing mandatory dependency a resolver may include any requirements it has considered in the resolution exception. Clients may access this set of dependencies via the getUnresolvedRequirements() method.

Resolver implementations may subclass this class to provide extra state information about the reason for the resolution failure.

Immutable
ThreadSafe

Constructor Summary	Page
ResolutionException(String message) Creates an exception of type ResolutionException.	56
ResolutionException(String message, Throwable cause, Collection<Requirement> unresolvedRequirements) Creates an exception of type ResolutionException.	55
ResolutionException(Throwable cause) Creates an exception of type ResolutionException.	56

Method Summary		Page
Collection<Requirement>	**getUnresolvedRequirements**() May contain one or more unresolved mandatory requirements from mandatory resources.	56

Constructor Detail

ResolutionException

```
public ResolutionException(String message,
                           Throwable cause,
                           Collection<Requirement> unresolvedRequirements)
```

Creates an exception of type ResolutionException.

This method creates an ResolutionException object with the specified message, cause and unresolvedRequirements.

Parameters:

`message` - The message.
`cause` - The cause of this exception.
`unresolvedRequirements` - the requirements that are unresolved or null if no unresolved requirements information is provided.

ResolutionException

```
public ResolutionException(String message)
```

Creates an exception of type `ResolutionException`.

This method creates an `ResolutionException` object with the specified message.

Parameters:

`message` - The message.

ResolutionException

```
public ResolutionException(Throwable cause)
```

Creates an exception of type `ResolutionException`.

This method creates an `ResolutionException` object with the specified cause.

Parameters:

`cause` - The cause of this exception.

Method Detail

getUnresolvedRequirements

```
public Collection<Requirement> getUnresolvedRequirements()
```

May contain one or more unresolved mandatory requirements from mandatory resources.

This exception is provided for informational purposes and the specific set of requirements that are returned after a resolve failure is not defined.

Returns:

a collection of requirements that are unsatisfied

Interface Resolver

org.osgi.service.resolver

```
public interface Resolver
```

A resolver is a service interface that can be used to find resolutions for specified resources based on a supplied Environment.

Version:
$Id: 5735a30be6494040afe5a05cadf353cf0ce943a0 $
ThreadSafe

Method Summary		Page
Map<Resource,List<Wire>>	**resolve**(Environment environment, Collection<? extends Resource> mandatoryResources, Collection<? extends Resource> optionalResources) Attempt to resolve the resources based on the specified environment and return any new resources and wires to the caller.	57

Method Detail

resolve

```
Map<Resource,List<Wire>> resolve(Environment environment,
                                 Collection<? extends Resource> mandatoryResources,
                                 Collection<? extends Resource> optionalResources)
                          throws ResolutionException
```

Attempt to resolve the resources based on the specified environment and return any new resources and wires to the caller.

The resolver considers two groups of resources:

- Mandatory - any resource in the mandatory group must be resolved, a failure to satisfy any mandatory requirement for these resources will result in a ResolutionException
- Optional - any resource in the optional group may be resolved, a failure to satisfy a mandatory requirement for a resource in this group will not fail the overall resolution but no resources or wires will be returned for this resource.

Delta

The resolve method returns the delta between the start state defined by Environment.getWiring() and the end resolved state, i.e. only new resources and wires are included. To get the complete resolution the caller can merge the start state and the delta using something like the following:

```
Map<Resource, List<Wire>> delta = resolver.resolve(env, resources, null);
Map<Resource, List<Wire>> wiring = env.getWiring();

for (Map.Entry<Resource, List<Wire>> e : delta.entrySet()) {
  Resource res = e.getKey();
  List<Wire> newWires = e.getValue();

  List<Wire> currentWires = wiring.get(res);
  if (currentWires != null) {
    newWires.addAll(currentWires);
  }

  wiring.put(res, newWires);
}
```

Consistency

For a given resolve operation the parameters to the resolve method should be considered immutable. This means that resources should have constant capabilities and requirements and an environment should return a consistent set of capabilities, wires and effective requirements.

The behavior of the resolver is not defined if resources or the environment supply inconsistent information.

Parameters:

`environment` - the environment into which to resolve the requirements
`mandatoryResources` - The resources that must be resolved during this resolution step or null if no resources must be resolved
`optionalResources` - Any resources which the resolver should attempt to resolve but that will not cause an exception if resolution is impossible or null if no resources are optional.

Returns:

the new resources and wires required to satisfy the requirements

Throws:

`ResolutionException` - if the resolution cannot be satisfied for any reason
`NullPointerException` - if environment is null

7 OBR Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.osgi.org/xmlns/obr/v1.0.0/"
        xmlns:obr="http://www.osgi.org/xmlns/obr/v1.0.0/"
        elementFormDefault="unqualified"
        attributeFormDefault="unqualified"
        version="1.0.0">
    <complexType name="Repository">
        <sequence>
            <choice maxOccurs="unbounded" minOccurs="0">
                <element ref="obr:resource"></element>
                <element ref="obr:referral"></element>
            </choice>
            <any namespace="##any"
                 processContents="lax"
                 minOccurs="0"
                 maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="string">
            <annotation>
                <documentation>
                    The name of the repository. The name may contain
                    spaces and punctuation.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="increment" type="long">
            <annotation>
```

```
                <documentation>
                     An indication of when the repository was last changed. Client's
can check if a
                    repository has been updated by checking this increment value.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute/>
    </complexType>

    <complexType name="Resource">
        <annotation>
            <documentation>
                Describes a general resource with
                requirements and capabilities.
            </documentation>
        </annotation>
        <sequence>
            <element ref="obr:require" maxOccurs="unbounded"
                     minOccurs="0">
            </element>
            <element ref="obr:capability" maxOccurs="unbounded"
                     minOccurs="1">
            </element>
            <any namespace="##any"
                 processContents="lax"
                 minOccurs="0"
                 maxOccurs="unbounded"/>
        </sequence>
    </complexType>

    <complexType name="Referral">
        <annotation>
            <documentation>
                A referral points to another repository XML file. The
                purpose of this element is to create a federation of
                repositories that can be accessed as a single
                repository.
            </documentation>
        </annotation>
        <attribute name="depth" type="int" use="optional">
            <annotation>
                <documentation>
                    The depth of referrals this repository acknowledges.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="url" type="anyURI" use="required">
            <annotation>
                <documentation>
                    The URL to the referred repository. The URL can be
                    absolute or relative from the given repository's
                    URL.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute/>
    </complexType>

    <element name="repository" type="obr:Repository"></element>

    <element name="resource" type="obr:Resource"></element>

    <element name="referral" type="obr:Referral"></element>
```

```
    <simpleType name="Version">
        <annotation>
            <documentation>
                Version must follow the major, minor, micro, qualifier
                format as used the Framework's version class. Example is
                "1.0.4.R128"
            </documentation>
        </annotation>
        <restriction base="string">
            <pattern value="\d+(\.\d+((\.\d+)(\..+)?)?)?"></pattern>
        </restriction>
    </simpleType>

    <element name="require" type="obr:Require"></element>

    <element name="capability" type="obr:Capability"></element>

    <complexType name="Capability">
        <annotation>
            <documentation>
                A named set of type attributes and directives. A capability can be
                used to resolve a requirement if the resource is included.
            </documentation>
        </annotation>
        <sequence>
                                <element   ref="obr:attribute"   maxOccurs="unbounded"
minOccurs="0"></element>
                                <element   ref="obr:directive"   maxOccurs="unbounded"
minOccurs="0"></element>
            <any namespace="##any"
                 processContents="lax"
                 minOccurs="0"
                 maxOccurs="unbounded"/>
        </sequence>
        <attribute name="namespace" type="string">
            <annotation>
                <documentation>
                    Namespace of the capability. Only requirements with the
                    same namespace must be able to match this capability.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute/>
    </complexType>

    <complexType name="Require">
        <annotation>
            <documentation>
                A filter on a named set of capability attributes.
            </documentation>
        </annotation>
        <sequence>
                                <element   ref="obr:attribute"   maxOccurs="unbounded"
minOccurs="0"></element>
                                <element   ref="obr:directive"   maxOccurs="unbounded"
minOccurs="0"></element>
            <any namespace="##any"
                 processContents="lax"
                 minOccurs="0"
                 maxOccurs="unbounded"/>
        </sequence>
        <attribute name="namespace" type="string">
            <annotation>
```

```
            <documentation>
                Namespace of the requirements. Only capabilities with the
                same namespace must be able to match this requirement.
            </documentation>
        </annotation>
    </attribute>
    <anyAttribute/>
</complexType>

<element name="directive" type="obr:Directive"></element>

<element name="attribute" type="obr:Attribute"></element>

<complexType name="Attribute">
    <annotation>
        <documentation>
            A named value with an optional type that identifies
            a requirement or capability
        </documentation>
    </annotation>
    <sequence>
        <any namespace="##any"
             processContents="lax"
             minOccurs="0"
             maxOccurs="unbounded"/>

    </sequence>
    <attribute name="name" type="string">
        <annotation>
            <documentation>The name of the attribute</documentation>
        </annotation>
    </attribute>
    <attribute name="value" type="string">
        <annotation>
            <documentation>The value of the attribute</documentation>
        </annotation>
    </attribute>
    <attribute name="type" type="obr:PropertyType" default="string">
        <annotation>
            <documentation>The type of the attribute.</documentation>
        </annotation>
    </attribute>
    <anyAttribute/>
</complexType>

<complexType name="Directive">
    <annotation>
        <documentation>
            A named value of type string that instructs a resolver
            how to process a requirement or directive
        </documentation>
    </annotation>
    <sequence>
        <any namespace="##any"
             processContents="lax"
             minOccurs="0"
             maxOccurs="unbounded"/>

    </sequence>
    <attribute name="name" type="string">
        <annotation>
            <documentation>The name of the directive</documentation>
        </annotation>
    </attribute>
```

```
        <attribute name="value" type="string">
            <annotation>
                <documentation>The value of the directive</documentation>
            </annotation>
        </attribute>
        <anyAttribute/>
    </complexType>

    <simpleType name="PropertyType">
        <restriction base="string">
            <enumeration value="boolean"></enumeration>
            <enumeration value="string"></enumeration>
            <enumeration value="version"></enumeration>
            <enumeration value="uri"></enumeration>
            <enumeration value="long"></enumeration>
            <enumeration value="double"></enumeration>
            <enumeration value="list"></enumeration>
        </restriction>
    </simpleType>
    <attribute name="must-understand" type="boolean" default="false">
        <annotation>
            <documentation>
                 This attribute should be used by extensions to documents to require
that the document consumer
                understand the extension. This attribute must be qualified when used.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

8 Considered Alternatives

8.1 Licensing

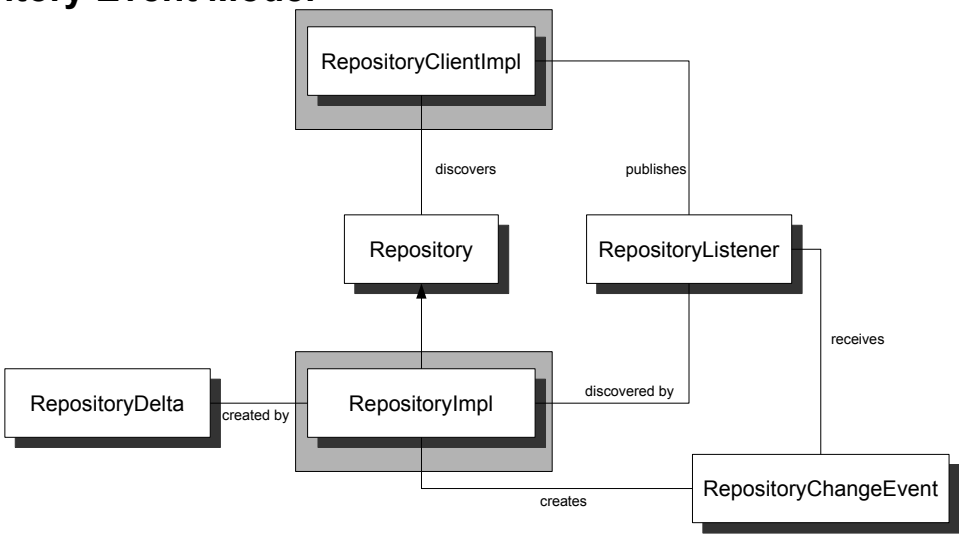
The value of the repository would be greatly enhanced if we would support a licensing model. Currently, certain bundles require the authentication so they can not be directly downloaded. This makes OBR like solutions for these bundles impossible in a standard way.

This is an interesting area for future research but it is beyond the scope of this RFC to tackle this problem here. It is felt that a licensing model may be layered on top of the current design at a later stage.

8.2 Problem Analysis

The resolver API is potentially useful during the development lifecycle and to help in runtime diagnostics when deployments go wrong. However due to the range of resolution algorithms that are possible, providing a meaningful general purpose API that can help users with diagnostic problems is very hard. This problem is delegated to a separate RFP.

8.3 Repository Event Model



8.3.1 Repository

- RepositoryDelta getDelta(long sinceIncrement) - Provides a mechanism to query the changes that have occurred to this repository since the specified increment.

8.3.2 RepositoryDelta

A repository delta lists the changes to a repository between to increments. The API of a RepositoryDelta is:

- List<Resource> getAddedResources();
- List<Resource> getChangedResources();
- List<Resource> getRemovedResources();

8.3.3 RepositoryListener and RepositoryChangeEvent

Depending on the underlying technology used to implement a Repository it may be possible to update the list of Resources that a Repository provides. If this happens it is necessary to inform the Resolver and any associated management tools of these changes. The RepositoryListener is a service interface that is looked up using the white board pattern by Repository implementations to notify of such changes:

The API of the RepositoryListener is:

- void repositoryChanged(RepositoryChangeEvent event);

The API of the RepositoryChangeEvent is:

- long getIncrement()
- Repository getRepository()

8.4 RepositoryBuilder

A repository builder service interface to allow third party code to simply build repositories by pointing at a URL location. Though useful the requirements for this service have not been fully captured so like the repository event model this work has been pushed out to a future RFP/RFC to ensure all relevant issues are covered.

The interface of a RepositoryBuilder is defined as follows:

- `Repository build(URL location)` – Attempts to build a Repository from the supplied location and returns a repository or null if this builder does not know how to build a repository from the supplied location

8.5 Extends:=true directive

It is possible to generically identify a requirement as an extension using a directive attribute. However this is not backwards compabitible with the current “osgi.wiring.host” BundleRequirement and is also over engineering as there are no other examples of extensions in use in OSGi at the present time. This design may be revisited in future if other extension types are defined.

8.6 Stateful resolver

The original OBR resolver used a stateful API design, however on discussion it has been agreed that a stateless API design is preferable as it is always possible to create a stateful wrapper around a stateless design but not to go the other way.

8.7 VersionRanges as explicit elements – separate from filters

Nimble and P2 encode the VersionRange as a separate element outside of the filter. This aids optimisation strategies but it is felt that early optimisation is a mistake for this API.

8.8 Garbage collection

Nimble supports the uninstall process which currently this specification is silent on as deployment has been removed as a goal for this specification. When a top level dependency is removed the transative set of dependencies that are no longer used is removed from the framework.

Discussion on conference calls suggested this is better handled in the Subsystem RFC work.

8.9 Separation of logical representation from physical representation

Both P2 and Nimble separate the concept of a logical unit of deployment from the physical artifact that is deployed. P2 uses the concept of an InstallableUnit with underlying artifacts. Nimble uses the concept of a Rule with underlying artifact.

Suggestion is to split resource into “part” with zero or more “resources”. A part is a logical unit of deployment that has requirements and capabilities. A resource is a physical artifact that can be accessed by URL and may have attributes such as size, checksum etc.

One use case for this separation is a service that requires some default configuration, an extreme example being a database and a database schema. In this scenario the service may explicitly require it's configuration at deployment time before it can be meaningfully instantiated. This is related to the concept of resource builders mentioned in 8.11.

Another usecase is deployment of composite bundles – here the deployment of an “internal” bundle requires different processing than a “top level” bundle and deployment is complicated by treating sub parts of the graph as logically separate entities.

Decision taken in conference call to remove deployment from this specification so this is not a concern for the current time.

8.9.1 Uses calculation

The default resolver strategy should take account of uses, however it should be an option available to the client at resolve time whether to calculate the uses constraints.

The uses problem space has been shown to be np complete [4]. and so can explode into a massively time consuming task. However in a large number of situations we've encountered in the wild the calculations are practically unnecessary at runtime as theoretical uses constraints never occur and are an artifact of classes cross cutting package boundaries.

From a practical point of view it is useful to be able to get a quick solution to enable the user to carry on coding/testing vs wait for the end of time to prevent a class mismatch that will never occur in running code and is only an artifact of limitations in a module implementation.

8.10 Query protocol

A query protocol is not needed as resources can be simply discovered using ldap matches on attributes, more complex searches are out of scope for this specification.

8.11 Relationship to DeploymentAdmin/ResourceBuilder

Ability to apply custom deployment steps to resources on installation is key for a general purpose deployer. However there are several rabbit holes in this area that need to be explored and the decision taken was taken during conference calls to remove deployment from this specification and push this to other specifications such as Subsystems.

8.12 Querying a Web Service Based Repository

The repository can become quite large in certain cases. So large that small environments cannot handle the full repository anymore. For scalability reasons, it is therefore necessary to query the repository to only receive smaller chunks. Server based repositories are recommended to support the following query parameters after the URL:

- keywords – A space separated (before URL encoding) list of keywords. This command must return all resources that match a keyword in the description, category, copyright, etc, case insensitive.
- requirement – A structured field. The first part is the name of the requirement, followed by a legal filter expression.
- category – A category

All fields can be repeated multiple times. The server should return the subset of the resources that match all fields. That is, all fields are anded together. However, the receiver must be able to handle resources that were not selected, that is, no assumption can be made the selection worked. The purpose of the selection criteria is a potential optimization.

As a further optimization, it is allowed to specify the resources that are already received. This a comma separated lost of repository ids. The server should not send these resources again. The name of this parameter is *knows*.

For example

```
http://www.aqute.biz/bundles/repository.xml?requirement=package:(\
package=org.osgi.util.measurement)&knows=1,2,3,4,9,102,89
```

This functionality has been moved out of this specification as it is possible to build this functionality on top of the existing design

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

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

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

[3]. JSR 124 J2EE Client Provisioning http://www.jcp.org/en/jsr/detail?id=124

[4]. Is the resolution problem in OSGi np complete? http://stackoverflow.com/questions/2085106/is-the-resolution-problem-in-osgi-np-complete/2133559#2133559

10.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery, Beaulieu, France
Voice	+33 467542167
e-mail	Peter.Kriens@aQute.biz
Name	Richard S. Hall
Company	
Address	Saginaw, Mi, USA
Voice	+1
e-mail	Heavy@ungoverned.org

Name	Hal Hildebrand
Company	Oracle Corporation
Address	500 Oracle Parkway, MS 2op946
Voice	+1 506 2055
e-mail	hal.hildebrand@oracle.com
Name	David Savage
Company	Paremus Ltd
Address	107-111 Fleet Street, London, EC4A 2AB, UK
Voice	+44 20 7993 8321
e-mail	david.savage@paremus.com

10.3 Acronyms and Abbreviations

10.4 End of Document

Subsystems

Draft

73 Pages

Abstract

This RFC proposes a design for OSGi Subsystems, where a subsystems is a collection of bundles with sharing and isolation semantics. The requirements for this RFC were defined and agreed in RFP 121.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 10.1.

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

0.3 Revision History

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

Revision	Date	Comments
Initial	*11 19 2009*	*Created from RFP 121* *Graham Charters, IBM charters@uk.ibm.com*
0.1	*08 01 2010*	*Added Subsystem Definition Headers* *Graham Charters, IBM charters@uk.ibm.com*
0.2	*23 07 2010*	*Added JavaDoc. Added architecture overview/entities. Added subsystem types overview. Started adding deployment manifest details. Added archive definitions. Added lots of TODOs!* *Graham Charters, IBM charters@uk.ibm.com*
0.3	*01 09 2010*	*Consideration of XML or Java Properties files in Alternatives.* *Artefacts now in OSGI-INF instead of META-INF.* *ResourceProcessor uses new Coordination Service.* *SubsystemAdmin uses Futures for long-running activities.* *Use of EventAdmin instead of SubsystemListeners.* *Some details on Subsystem life-cycle (incomplete)* *Graham Charters, IBM charters@uk.ibm.com*
0.4	*22 11 2010*	*Graham Charters, IBM charters@uk.ibm.com* *Updated life-cycle to include transitional states.* *Removed use of Futures (transitional states mean they're of limited use).* *Added ability to snoop on internal subsystem events.* *Clarified resolution limitation (no cycles).* *Updated resource processors api to align with subsystem life-cycle operations.* *Updated references to RFC 138 to cover framework hooks.*
0.5	*01 02 2011*	*Graham Charters, IBM charters@uk.ibm.com* *Changed resource processor operations to sychronous.*

Revision	Date	Comments
0.6	*7th April 2011*	*Graham Charters, IBM charters@uk.ibm.com* *Updates based on Tom Watson's comments and discussion at Austin face-to-face, e.g.:* • remove SubsystemAdmin, moving capabilities to Subsytem. • new ResourceOperation approach for ResourceProcessors.
0.7	*06/06/11*	*John Ross, IBM, jwross@us.ibm.com* • Removed references to Resource Processors, which are no longer in scope. • Added cancel() method to Subsystem interface. Also added several missing Subsystem.State constants. • Added missing SubsystemConstants.EventType.UNINSTALLING constant. • Removed references to SubsystemEventConstants. These constants are now a part of SubsystemConstants. • Made a few comments. Responded to a few comments.
0.8	*12th August 2011*	*Graham Charters, IBM charters@uk.ibm.com* • Collapsed headers into single Subsystem-* set. • Collapsed extensions to single .ssa extension. • Filled out deployment manifest definition • Added description of transitive dependency provisioning with concept of root subsystem provision-policy. • Added description on handling share resource states.

Revision	Date	Comments
0.9	9/12/11	*John Ross, IBM, jwross@us.ibm.com* • *Accepted changes from revision 0.8 in SVN.* • *Merged changes from 0.8 branch not in SVN.* • *Several minor fixes.* • *Added default header value information, including the subsystem URI and file naming conventions.* • *Removed references to Coordinator and Coordination.* • *Updated state table.* • *Added reference to Bundle Collision Hook and bsnversion=managed.* • *Added start-order directive to Subsystem-Content and Deployed-Content headers.* • *Added use of Repository services requirement.* • *Addressed typo's, etc., highlighted in Glyn Normington's email on 8/25/11.* • *Updated Javadoc.*

1 Introduction

The OSGi platform provides a very appealing deployment platform for a variety of applications. In many scenarios, and especially enterprises, an application can consist of a large number of bundles. Administrators typically think in terms of applications for tasks such as deployment, configuration and update, rather than individual bundles, and therefore there is a mismatch in concepts between those provided by the OSGi framework and those familiar to an administrator.

The problem domain was introduced and requirements agreed in RFP 121. This RFC proposes a solution to address those requirements. Input to the design in this RFC will be drawn from a number of existing sources:

- SpringSource dm Server - *applications*
- Apache Felix Karaf – *features*

- Apache Aries – *applications*
- Newton – *systems*
- Eclipse – *features/installable units*
- DeploymentAdmin – *deployment package*

2 Application Domain

When System Administrators, Deployers and other people think about software they typically think in terms of systems performing a certain business function. A System Administrator's job might be to make sure that these systems stay up and running. The sysadmin doesn't have intimate knowledge of how these systems are constructed, but (s)he does have knowledge of the applications that compose the systems. There might be an ordering application, a payment application and a stock checking application. The sysadmin knows that these applications exist and that they must be monitored to ensure that they are in a healthy state.

Other people may think of a system yet on a different level. The CIO of the company may have a higher level view. He sees the system described above as a single application unit: the company's web shop and views all of the applications that together run in his organization as 'The System'.

On the other end of the spectrum, a developer might be tasked with implementing the payment system. This system might in turn be built up of other applications. There might be a currency conversion application, a credit card charge application and an electronic bank transactions application.

Bottom line is that what one person sees as an application might be a mere component to the next person in the chain. Application is a nestable concept and as such it is referred to by the more general word ‘Subsystem’. What the people described above have in common is that a pure OSGi bundle is typically not what they view as an application. An OSGi bundle is a component used to build applications, but applications themselves, no matter from what level you look at it, are always bigger than a bundle.

So from a conceptual point of view, applications are nestable. However, this does not mean that the runtime realization of these nested applications needs to exactly follow the nested structure. In OSGi systems, applications are built up from OSGi bundles, and although it would be desirable to view an OSGi system in a way that makes sense to the person looking at it, this does not mean that the bundles themselves need to be nested as well. The application view could be a virtual, organizational layer over a potentially flat bag of bundles that realize the applications at runtime. In some cases layering might be needed to enforce separation but in other systems the layering might not be used.

An application level view over an OSGi system might look like Figure 1.

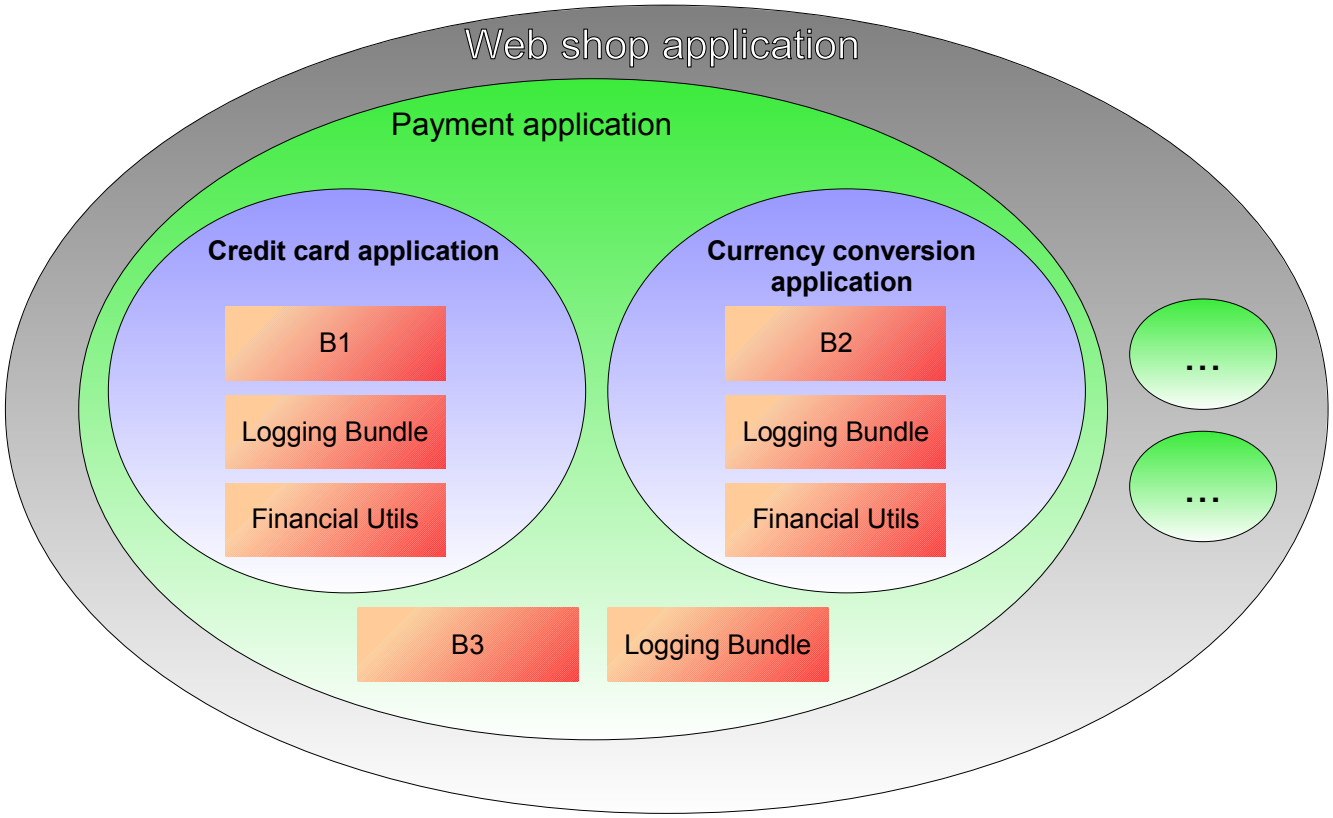


Figure 1: Application level logical view of a deployment

At runtime, the bundles composing this application might be deployed as a flat structure where the 'Logging Bundle' and 'Financial Utils' bundles are shared. This way of deploying the application is very efficient with regard to memory footprint. See Figure 2.

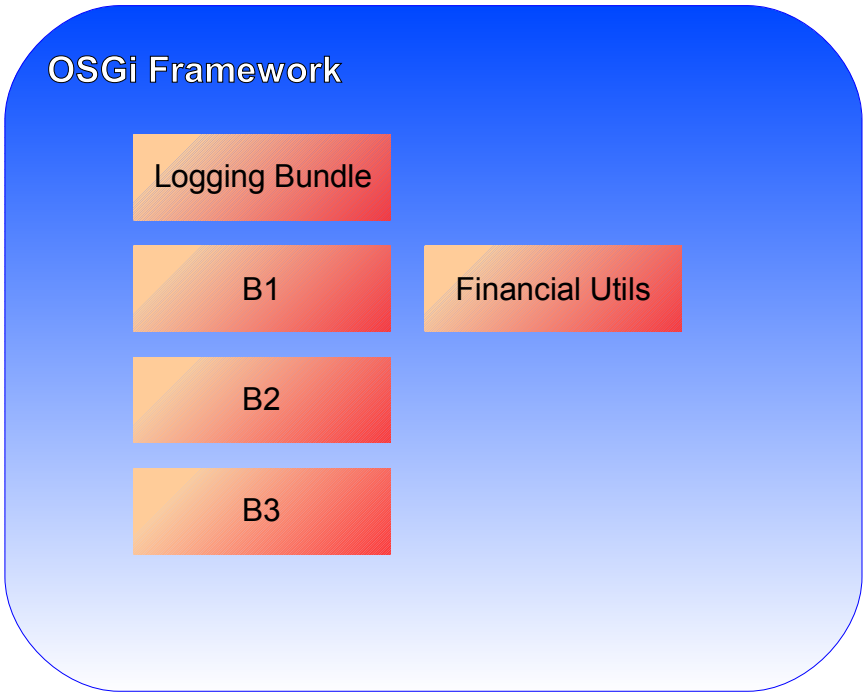


Figure 2: Runtime realization with sharing

Framework resolver hooks defined by RFC 138 could be used to isolate certain subsystems while still sharing other bundles, like the logging bundle. This provides another runtime deployment option for the same system, providing more isolation. See Figure 3.

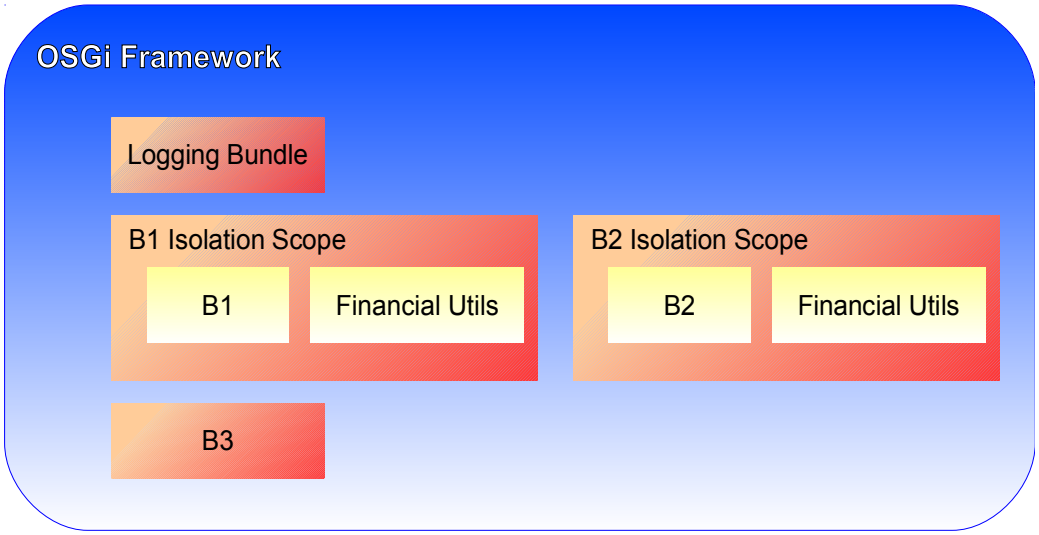


Figure 3: Runtime realization with less sharing and more isolation

Besides the applications installed in an OSGi framework, there are also bundles installed that are considered part of the OSGi framework infrastructure, but at runtime these bundles could become part of the application. Consider a bundle providing the logging service or a bundle providing the HTTP Service. OSGi Services are a great architecture for loose coupling, fostering sharing and re-use.

2.1 Related OSGi Specifications

2.1.1 OBR (RFC 112)

RFC 112 describes the OSGi Bundle Repository. This repository could provide an excellent place to store all the bundles required to deploy an application. It would therefore be important that any new designs resulting from this requirements document fully integrate and leverage the OBR.

However, using an OBR may not be right for every use-case so it should be optional. In some contexts development might start out relatively loose, using artifact repositories such as an OBR. Over time during the development process the dependencies in the project might harden and ultimately an OBR may not be appropriate any more. In a completely hardened situation dependencies might be obtained from a local directory created by an installer process or possibly a fixed corporate web site.

2.1.2 Framework Hooks (RFC 138)

Framework Hooks as proposed in RFC 138 provide a way of isolating or grouping subsystems within a single OSGi Framework. Hook configuration could be used to model an application at runtime, but there may also be implementations that do not require isolation so use of Framework Hooks should be optional.

2.1.3 Initial Provisioning

The Initial Provisioning spec does not relate to an application concept at all. It rather concerns how to deploy initial bundles on a device. This specification is unrelated to this RFC, but mentioned here to confirm that there is no potential overlap.

2.1.4 Deployment Admin

The Deployment Admin specification is about defining a file format to ship and deploy applications. Such an application is built up of multiple bundles and additional resources. Patching of applications, is also supported by this specification.

An additional feature provided by Deployment Admin, and which is considered useful for subsystems is the capability to roll back a deployment.

A major drawback of the Deployment Admin specification is that it explicitly prohibits sharing of bundles, which, besides preventing memory efficiency, could cause real problems when two applications use the same third-party bundle, as installing the same bundle twice is not allowed. Even different versions of the same bundle can not be installed simultaneously with Deployment Admin.

2.1.5 Application Admin

Application Admin provides the concept of an Application in OSGi. This specification could potentially be the basis of a runtime API into the Application Metadata and life-cycle system. It would most likely have to be extended to support all the use cases. Missing in the Application Admin spec is the Application Metadata, which should be declarative and available both inside the running OSGi framework as well as outside it to support OSGi tooling for subsystems.

Concepts missing from Application Admin:

- No file format to describe an application, purely API based
- No impact analysis for doing upgrades
- No interaction with OBR
- No application fingerprinting
- No runtime application extent analysis (what bundles beyond the core app set are used by this app)

2.1.6 Bundle Collision Hook (RFC 174)

The purpose of this hook is to provide a way to tell the framework what it should do when encountering a bundle with the same bsn and version. This new hook was necessary because the existing Find Hook could not be used during a bundle update because there is no valid bundle context at that time. The hook implementation will need to filter out any candidate having the same bsn and version as another bundle within the targeted region.

2.2 Terminology + Abbreviations

-
- *Root Subsystem – The top-most subsystem in the hierarchy and, therefore, has no parent. It is always available as the starting point for installing other subsystems and resides at the same level into which the subsystems implementation bundle was installed.*
- *Resource – A bundle, subsystem, or configuration.*
- *Content Resource – A resource that is part of the subsystem content.*
- ***Transitive Dependency** – A resource that is not part of the subsystem content but is necessary to satisfy content requirements (e.g., package, bundle, or service dependencies).*
-
- Unisolated Subsystem – A subsystem that does not restrict imported requirements or exported capabilities. It effectively defines a set of resources that are provisioned into the OSGi framework in a flat bundle space.
-
- Isolated Subsystem – A subsystem that restricts imported requirements and exported capabilities. The contents are isolated using an RFC 138 hook configuration.
-
- Implicitly Isolated Subsystem – A subsystem that implicitly restricts imported requirements and exported capabilities according to its type. For example, an application does not export any capabilities but will import any requirements not satisfied by the application's content.
-

- Explicitly Isolated Subsystem – A subsystem that explicitly restricts imported requirements and exported capabilities. The isolation policy is defined by the user within the subsystem manifest.

3 Problem Description

In today's OSGi framework, what is deployed is typically just a large set of bundles. To a person not familiar with the details of the design of these bundles it is often unclear what function these bundles perform and how they are related.

Some bundles might provide shared infrastructure (e.g. a bundle providing the OSGi Log service), while other bundles might together provide an application function (e.g. a set of bundles together implementing a web-based shopping application). Today it is not possible to find out from a high level what applications are installed in the OSGi framework. The only information available is the list of bundles.

The OSGi framework needs to be enhanced with a mechanism that makes it possible to declare applications. An application has a name meaningful to the deployer. It is typically composed of a number of key bundles and services, plus their dependencies. When the deployer requests the list of installed applications, he will get a manageable result that is typically much shorter than the list of installed bundles.

The deployer also needs to be able to perform actions on the application level. Installing, uninstalling, starting, and stopping should be possible on the level of an application.

A developer may wish to think in terms of an application consisting of a set of bundles and may wish to declare those bundles as belonging to the application.

3.1 Problem Scope

Graphical tools are out of scope for this RFC.

In scope would be the metadata needed to define the applications plus an API that would enable actions at the application level.

4 Requirements

The following requirements are taken from RFP 121, “Subsystem metadata and Lifecycle”. The requirement numbers used here are identical to those in RFP 121. Some requirements were ommitted from the final RFP and hence the numbering is intentionally not contiguous.

4.1 Subsystem Modeling

REQ 1. The solution MUST provide a means to describe a subsystem which can be accessed both inside a running OSGi framework (e.g. through an API) as well as outside of a running framework (e.g in a file).

REQ 2. The solution MUST define a subsystem definition format.

REQ 3. It MUST be possible to define a subsystem consisting of OSGi bundles.

REQ 4. It MUST be possible to make artifacts other than OSGi bundles part of a subsystem definition.

REQ 5. It MUST be possible to include a subsystem definition in another subsystem definition.

REQ 6. It MUST be possible to reference another subsystem in a subsystem definition, which makes the other subsystems a dependency.

REQ 7. It MUST be simple to define a scoped subsystem of the type defined in requirement 8 but with a default sharing policy that hides everything in the given subsystem from the parent of the given subsystem and makes all bundles (for wiring purposes only), packages and services visible inside the given subsystem from its parent.

REQ 8. It MUST be possible to scope subsystems. Scoped subsystems form a hierarchy. The subsystems, bundles, packages, and services belonging to a given scoped subsystem are visible within the scope of the given subsystem. A subsystem MUST have control over a sharing policy to selectively make packages and services visible to the parent of that subsystem. A subsystem MUST have control over a sharing policy to selectively make bundles (for wiring purposes only), packages and services visible inside that subsystem from its parent. Subsystems, bundles, packages, and services belonging to unscoped subsystems are always available to all peer subsystems (scoped or unscoped) and bundles.

REQ 9. Subsystems MUST be uniquely identifiable and versioned.

REQ 10. Subsystem definitions MUST be extensible. This will allow tools to store associated data along side the subsystem definitions.

4.2 Dependency Management

REQ 13. It MUST be possible to use constraints to declaratively identify bundles and other artifacts in a subsystem definition.

REQ 15. It MUST be possible to define a subsystem as a number of key bundles. The transitive dependencies of the subsystem are inferred from the meta-data of the key bundles.

4.3 Administration

REQ 18. The solution MUST provide an API to query the subsystems available in the OSGi container, their associated state (as a snapshot) and dependencies.

REQ 19. The fidelity of the subsystem states MUST be sufficient to capture the possible states of its constituents

REQ 20. The solution MUST provide an API to drive the subsystem life-cycle.

REQ 22. The solution SHOULD provide an API to do impact analysis regarding replacing one or more bundles or subsystems. This impact analysis should list the subsystems affected and describe how these subsystems are using the affected bundle: either by importing an interface or a class from it, purely through services or through the extender pattern. It should also describe existing transitive dependencies and new transitive dependencies.

REQ 24. The APIs that modify the system SHOULD not leave the system in an inconsistent state. .

4.4 Runtime

REQ 26. The solution MUST allow a single bundle to be part of multiple subsystems.

REQ 27. The solution MUST allow multiple versions of the same bundle.

REQ 28. Subsystems MUST have a well-defined life-cycle.

4.5 Development Support

REQ 29. The solution MUST support the development process by allowing bundles to be updated with other bundles that have the same version as the previous, similar to how Maven supports SNAPSHOT versions of artifacts.

4.6 RFC 138 Compatibility

REQ 30. The solution MUST be compatible with the scoping mechanisms possible through RFC 138.

5 Technical Solution

Subsystems enable a number of resources to be managed as a single entity, such as a set of bundles with associated configuration. Subsystems also allow selective sharing of capabilities and requirements (borrowing the terms from rfc 154), such as packages and services. For example, a subsystem might be defined in terms of a set of bundles that share a number of packages and services between themselves, but only share a subset of

those capabilities outside the subsystem. This selective sharing enables the creation of assets that are more coarse-grained than individual bundles, and exhibit their own modularity.

Subsystems simplifies the task for deploying the collection or resources by integrating the OSGi Bundle Repository (OBR) technology to aid resolution of the collection and defining a mechanism for describing and processing the resources to be deployed, including content and transitive dependency resources. Subsystems also describes how the life-cycle of the content and transitive resources are affected by the life-cycle of a subsystem.

Subsystems are designed to be easy to create for those familiar with bundle development. They therefore have many concepts and design choices in common with bundles. For example, a manifest format is used to define a subsystem, as well as describe its deployment, and they have a symbolic name and version for their identity.

The subsystem service design consists of the following elements:

1. Subsystem metadata for defining subsystems.

2. An API for the management of subsystems; install, uninstall, update, event notification.

5.1 Subsystems Architecture

The subsystems architecture is shown in figure 4.

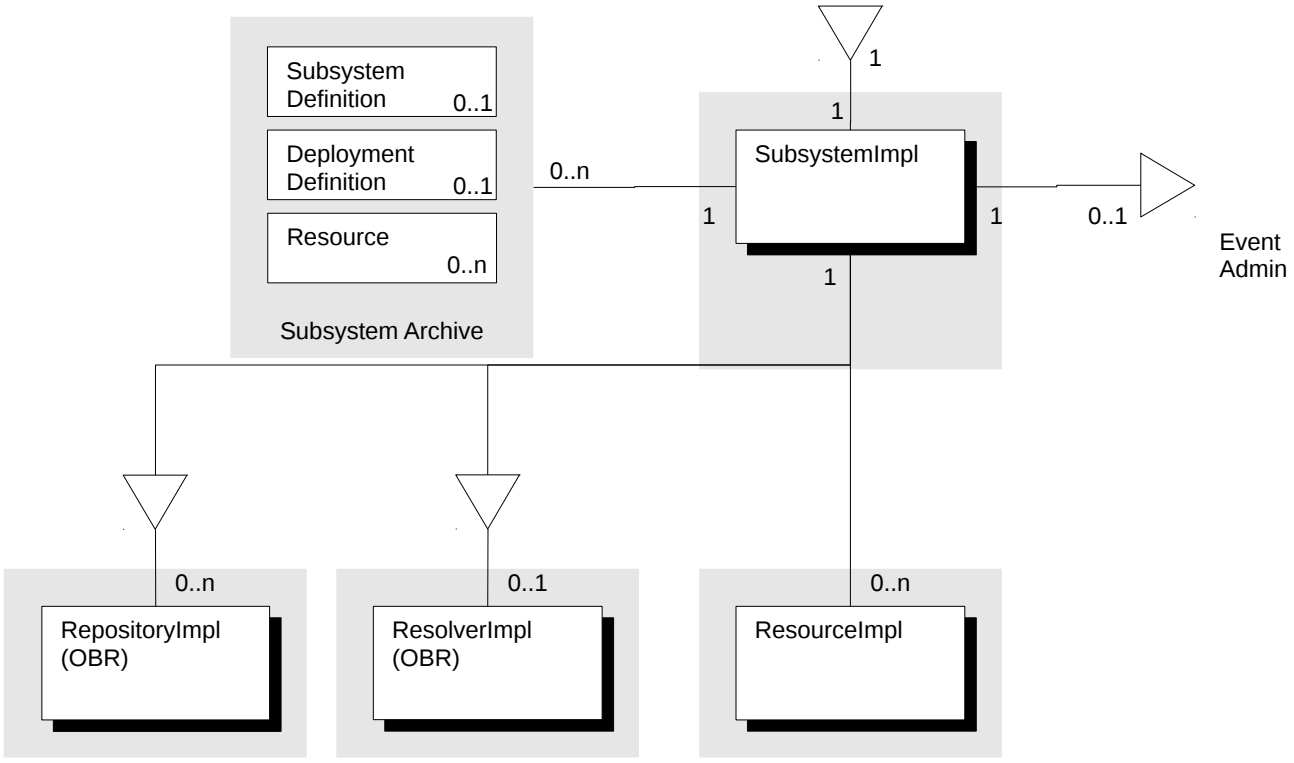


Figure 4: Subsystem Architecture

The main entities in the architecture are as follows:

- *Subsystem Definition* – A Subsystem Definition is a description of a subsystem that is processed by the Subsystem Admin service. For example, it describes the contents of the subsystem and, depending on the type of subsystem, it also describes any packages and services that are shared in our out. The subsystem definition is optional.

- *Deployment Definition* – A Deployment Definition describes the exact resources to provision for subsystem (i.e. the resolved content and any resources require to satisfy the subsystems dependencies), and any sharing poilicies (i.e. the packages and services that can be shared in or out of the subsystem). The deployment definition is optional.

- *Resource* – A Resource is any artifact, such as a bundle or configuration, that may be required when provisioning a subsystem.

- *Subsystem Archive* – An archive used to package a subsystem for installation into an OSGi runtime. The archive optionally contains a subsystem definition, a deployment definition, and resources.

- *Subsystem* – An implementation of this specification that provides the ability to install, uninstall and locate subsystems.

- *Repository* – Subsystems are not required to be, and typically won't be, transitively closed and therefore some amount of resolution will be required during installation to ensure all requirements are satisfied. Zero or more Repositories may be required in order to fully provision a Subsystem, including its transitive dependencies. Subsystems defines the use of repository services defined by the OBR specification, RFC 112. If OBR repositories are not available then a subsystem implementation is free to use others. If repository services are available, subsystems implementations must query them in service ranking order and include any found capabilities for possible use by a resolver. Implementations are free to give higher priority to capabilities found by other means.

- Resolver – A Resolver is used by the Subsystem implementation to determine whether the subsystem resolves (i.e. is transitively closed) or requires additional resources in order to satisfy all its requirements. The Resolver is defined by the OBR specification, RFC 112.

5.2 Subsystem Types

Three types of subsystems are defined in this specification; Application, Composite Bundle and Feature. Each differs in the way in which they share requirements and capabilities and are described in more detail in the next few sections.

5.2.1 Application Subsystems

An application subsystem is a subsystem with a sharing policy associated with what people would often consider to be an application. An application subsystem consists of a number of bundles and other supporting resources. The content bundles share package and service dependencies with each other. An application does not share any package or service capabilities to bundles outside the application. Any package or service requirements that are not satisfied by the content bundles themselves are automatically imported from outside the application.

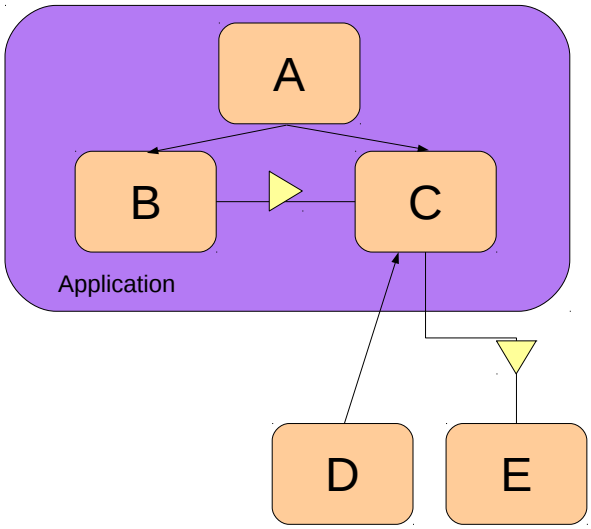


Figure 5: Application package and service sharing example.

An example of this point is shown in figure 5. An application consisting of 3 content bundles, A, B and C, is shown. Bundle A is exporting an API package to bundles B and C. Bundle C is providing a service which is being used by Bundle B. Bundle C also requires a package and service, neither of which are provided by bundles inside the application. These are therefore automatically shared into the application and in this example provided by bundles D and E, respectively.

5.2.2 Composite Subsystems

A composite subsystem is a subsystem with a fully explicit sharing policy. A composite consists of a set of content bundles and other supporting resources. These content bundles share package and service dependencies inside the composite. By default a composite does not share packages or services into or out of itself. Any packages or services that are to be shared into or out of the composite, must be explicitly identified in the composite subsystem definition.

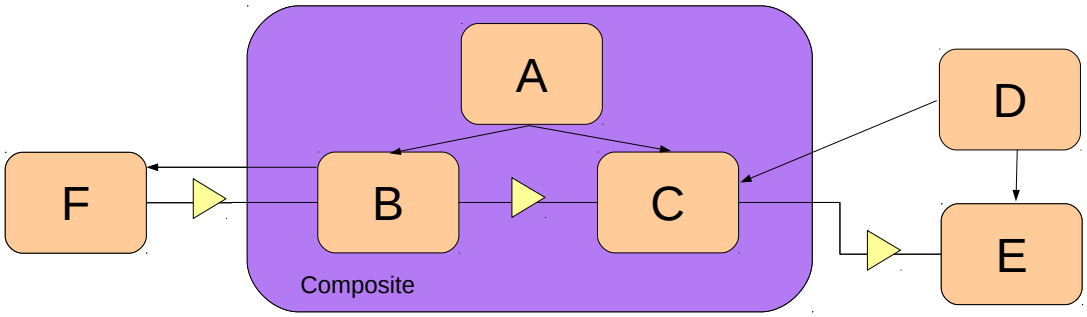


Figure 6: Composite package and service sharing example.

Figure 6 shows an example composite subsystem that is sharing packages and service both in and out of the composite for use by bundles inside and outside the composite, respectively. In this example, bundles A, B and C have various internal package and service dependencies. However, the composite also exports a package and service provided by bundle B that are being used by bundle F. In addition to this, the composite is importing a package provided by bundle D and a service provided by bundle E. Both of these are then being used by content bundle C.

It is worth noting that a consequence of the resolution process outlined in section 5.2.7 is that a composite subsystem is not permitted to be involved in a package dependency cycle with other peer subsystems or bundles. Such cycles will result in the subsystem failing to resolve.

5.2.3 Feature Subsystems

A feature subsystem is a subsystem that does not impose any isolation. A feature consists of a set of bundles and associated resources. The packages and services provided by the content bundles are all automatically made available to bundles outside the feature. Packages and services required by the content bundles can automatically be satisfied by bundles outside the feature. The content bundles can be required by bundles outside the feature and the content bundles can required by bundles from outside the feature. The main purpose of a feature subsystem is to enable the life-cycle management of a set of bundles.

Subsystem

An implementation of this specification provides a subsystem service for managing the life-cycle of subsystems. The service implements the org.osgi.service.subsystem.Subsystem interface (see section 6). Each subsystem service instance manages subsystems directly visibile to it in the subsystem hierarchy, as illustrated in figure 7.

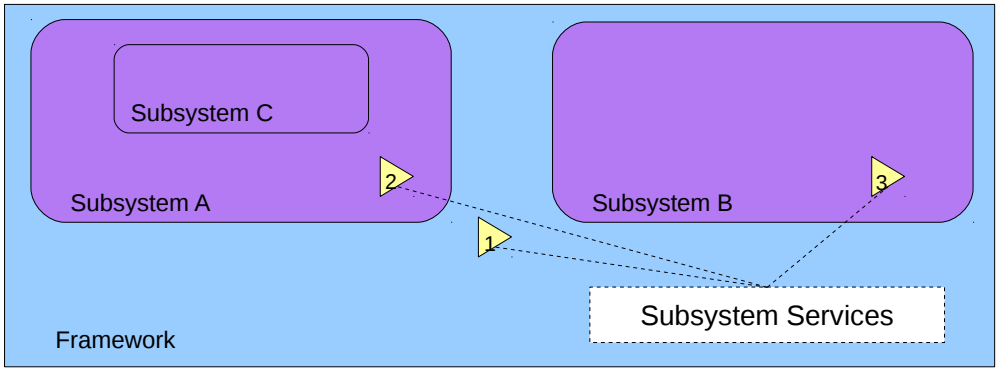


Figure 7: Subsystem Service instance model.

Figure 7 shows four Subsystem service instances registered, each numbered to aid description. In this example, at the outer-most level (the *root scope*), Subsystem service 1 is managing Subsystems A and B. A and B and be retrieved using **Subsystem.getChildren()**. Because this service is in the *root scope* a call to **Subsystem.getParent()** would return `null` as it has no parent subsystem. At the next level down, Subsystem service 2 is managing subsystem C and Subsystem service 3 is not managing any subsystems. A call to **Subsystem.getChildren()** on service 2 would return Subsystem C and **Subsystem.getParent()** would return Subsystem A. A call to **Subsystem.getChildren()** on service 3 would return and empty subsystem Collection as it has no child subsystems.

A subsystem implementation may choose to hide the subsystem service to prevent a subsystem's content bundles from creating their own subsystems. This is illustrated in Subsystem C, which does not contain a subsystem service.

Note, the Subsystem service view described above is logically what is expected, however a most likely implementation strategy would be to have a single service factory registration which is visible inside each subsystem and returns the appropriate subsystem instance based on the subsystem in which the requesting bundle lives.

5.2.4 Life-cycle

Subsystems have a life-cycle similar to that of Bundles. Operations on the Subsystem (e.g. install, uninstall, start and stop) cause it to move through its life-cycle. The operations also cause equivalent operations on the Subsystem content (e.g. start on a Subsystem will cause start on any content bundles). In these circumstances, the processing of the Subsystem content must also respect any life-cycle policy of the content resources. For example, starting a subsystem that contains bundle fragment resources must not require those fragments to start as this would be counter to the life-cycle policy of bundle fragments.

The use and behavior of subsystem life-cycle in the presence of the start level services is out of scope.

All subsystem states are defined by the `Subsystem.State` enum, for example, `Subsystem.State.INSTALLED`. For ease of reading these will hereafter be referred to by their short name, e.g. `INSTALLED`.

The life-cycle state of a subsystem is a reflection of the states of the contents and the last action performed through the subsystem api. For example, a Subsystem only transitions to `ACTIVE` when all its contents have successfully started, triggered by calling **start** on the subsystem. However, if a program uses the reflective api to change the state of an individual content bundle, this will not result in a change of state for of the subsystem. A

subsystem's state also reflects the cases where only a subset of it's contents have reached some target state. These states are referred to as 'transitional states'. Examples include `INSTALLING` and `RESOLVING`.

A subsystem's state is persistent, however, the state of their contents is not. Restarting a framework will cause subsystems to attempt to reach their previous states, which will try to bring the state of the content resources to the same state. For example, if a subsystem was `ACTIVE` when the framework was stopped, then when that framework is started again, the subsystem will try to reach the `ACTIVE` state again, which will result in an attempt to **start** all content resource, irrespective of their actual state when the framework was stopped.

Figure 8 shows the life-cycle for a Subsystem.

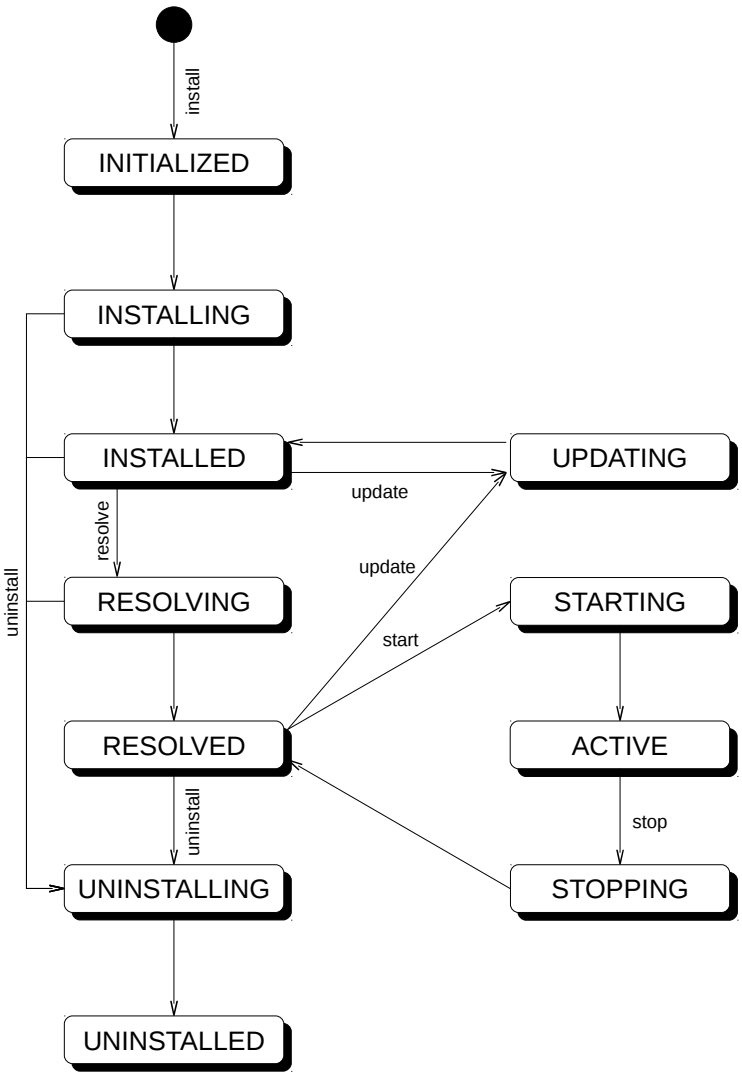


Figure 8: Subsystem Lifecycle

5.2.4.1 Events

When the EventAdmin service is available in the root scope, The subsystem runtime will produce events based on the Subsystem life-cycle. All events are delivered asynchronously. The Event Topic of these events is:

```
org/osgi/service/Subsystem/<event-type>
```

Whenever an event type is mentioned in this document, it should be assumed it belongs to the above topic, unless stated otherwise.

The following event types are generated based on the life-cycle of the Subsystem

Event Type	Description
INSTALLING	Indicates a subsystem has started installing
INSTALLED	Indicates a subsystem has completed installed
RESOLVING	Indicates a subsystem has started resolving.
RESOLVED	Indicates a subsystem has been resolved
STARTING	Indicates a subsystem has begun starting.
STARTED	Indicates a subsystem has become active.
STOPPING	Indicates a subsystem has begun stopping
STOPPED	Indicates a subsystem has been stopped

UPDATING	Indicates a subsystem has started updating
UPDATED	Indicates a subsystem has been updated
UNINSTALLING	Indicates a subsystem has started uninstalling.
UNINSTALLED	Indicates a subsystem has been uninstalled
CANCELING	Indicates that a subsystem operation is being canceled.
FAILED	Indicates a subsystem life-cycle operation (e.g. install, update) failed
CANCELED	Indicates an asynchronous subsystem operation was canceled (e.g. install or update).

Subsystem events contain the following properties to help identify the subsystem, the time at which the event occurred and any exceptions that may have occurred.

- `SUBSYSTEM_ID` – the id of the subsystem for which the event was being generated (from SubsystemConstants).
- `SUBSYSTEM_LOCATION`. - the location of the subsystem for which the event was generated (from SubsystemConstants).
- `SUBSYSTEM_SYMBOLICNAME` – the symbolic name of the subsystem from which the event was generated (from SubsystemConstants)
- `SUBSYSTEM_VERSION` – the symbolic name of the subsystem from which the event was generated (from SubsystemConstants)
- `TIMESTAMP` – the time at which the event was generated (from EventConstants).
- `SUBSYSTEM_STATE` – the current state of the subsystem (e.g. CANCELING) (from SubsystemConstants).
- `EXCEPTION` – contains any exception that caused the event (from EventConstants).
- `EXCEPTION_CLASS` – filled in as defined by event admin (from EventConstants).
- `EXCEPTION_MESSAGE` - filled in as defined by event admin (from EventConstants).

5.2.5 Listening for Internal Subsystem Events

Users of subsystems may need to be aware of the internal goings on of a subsystem. One use case for this is to be able to determine when a subsystem is 'ready'. Readiness may not necessarily directly correspond to one of the subsystem states outlined in section 5.2.4, For example, a library bundle might only need to reach the `RESOLVED` state, whereas a blueprint bundle might only be considered ready once its blueprint container service has been registered.

The subsystems runtime enables users to snoop on the inner workings of subsystems through EventAdmin. Events that occur inside a subsystem and which are not normally propagated outside the subsystem can be observed by registering an event handler in the subsystem service's root scope. Internal events are re-published using the following event topic pattern:

```
org/osgi/service/SubsystemInternals/<original-topic>
```

The use of the topic token “SubsystemInternals” keeps internal and external events in separate peer topic spaces, thus allowing wildcarding.

<original-topic> is the original topic of the event being re-published. The event published is the event admin event that would have been generated internally. For example, a bundle `STARTED` event would be published as:

```
org/osgi/service/SubsystemInternals/osg/osgi/framework/BundleEvent/STARTED
```

Note, because event admin is not subsystem aware, it is necessary to install event admin in every level at which event admin is to be used.

The following properties are added to the event to identify the subsystem from which the event came. These properties allow handlers to filter for a specific subsystem:

- `SUBSYSTEM_ID` – the id of the subsystem from which the event was generated (from SubsystemEventConstants).
- `SUBSYSTEM_LOCATION` - the location of the subsystem from which the event was generated (from SubsystemEventConstants).
- `SUBSYSTEM_SYMBOLICNAME` – the symbolic name of the subsystem from which the event was generated (from SubsystemEventConstants)
- `SUBSYSTEM_VERSION` – the symbolic name of the subsystem from which the event was generated (from SubsystemEventConstants)
- `TIMESTAMP` – the time at which the event was generated (from EventConstants). This value should be set only if not already present.
- `SUBSYSTEM_STATE` – the current state of the subsystem. This is used when the state is not indicated by the event type (e.g. CANCELING) (from EventConstants).
-

5.2.6 Installing a Subsystem

Subsystems are installed using one of the service's **install** methods. These methods carry the same arguments as BundleContext.installBundle and the same semantics, such as a null InputStream meaning the InputStream is created from the location identifier. Subsystem install operations are potentially long-running and are therefore asynchronous. Each install method returns a Subsystem with the installation process initiated, but not necessarily complete.

The following steps are required to install a subsystem:

1. If there is an existing subsystem containing the same location identifier as the Subsystem to be installed then the existing Subsystem is returned.
2. If this is a new install, then a new `Subsystem` is created with its id set to the next available value (ascending order)
3. The subsystem's state is set to `INSTALLING` and if EventAdmin is available, an event of type `INSTALLING` is fired.
4. The following installation steps are then started and performed asynchronously and the new subsystem is returned to the caller:
5. The subsystem content is read from the input stream.
6. If the subsystem requires isolation (i.e. is an application or a composite), then isolation is set up while the install is in progress, such that none of the content bundles can be resolved. This

isolation is not changed until the subsystem is explicitly requested to resolve (i.e. as a result of a **Subsystem.start()** operation).
7. If the subsystem does not include a deployment manifest (see 5.3.6), then the subsystem runtime must calculate one as described in section 5.3.6.17.
8. The resources identified in the deployment manifest are installed into the framework. All content resources are installed into the Subsystem, whereas transitive dependencies are installed into an ancestor subsystem as describe in section 5.2.6.1. If any resources fail to install, then failure is handled as described in 5.2.6.2. Transitive resources are free to resolve and start independent of the subsystem they were installed for.
9. The subsystem's state is set to `INSTALLED` and if EventAdmin is available an `INSTALLED` event is fired.

Note, if the subsystem requires isolation and it is in the `INSTALLING` or `INSTALLED` state, none of it's content bundles must be permitted to resolve.

5.2.6.1 Resource Installation

Two classes of resource are potentially installed during subsystem installation; content resources and transitive resources

Content Resources – these are resources identified by the Subsytem-Content (5.3.2.4) header of the subsystem manfiest. The deployment manifest Deployed-Content header (5.3.6.5) identifies the exact version of each content resource to be provisioned. Content resources are installed directly into the Subsystem.

Transitive Resources – these are the resources required to provide capabilities to satisfy requirements from the content resources that are not satisfied by capabilities provided by the content resources themselves or the capabilities provided by the target environment.

Transitive resources are provisioned into the ancestor hierarchy of the subsystem. The final destination of the transitive resources is determined by the subsystem or its ancestors. A subsystem states whether it is willing to hold transitive dependencies by specifying the “provision-policy:=acceptTransitive” directive on the Subsystem-Type header. Transitive dependencies are provisioned into the first subsystem subsystem with acceptTransitive encountered when traversing up the subsystem hierarchy, including the subsystem being provisioned. If no such subsystem is encountered, then the transitive dependencies are provisioned into the *root scope*.

Figure 9 shows a simple example of provisioning transitive dependencies. An application subsystem has been installed into a composite subsystem where the composite has opted to allow transitive dependencies. The application has three content bundles A, B and C, and two transitive dependency bundles, D and E. The two transitive dependencies have been provisioned into the parent composite subsystem.

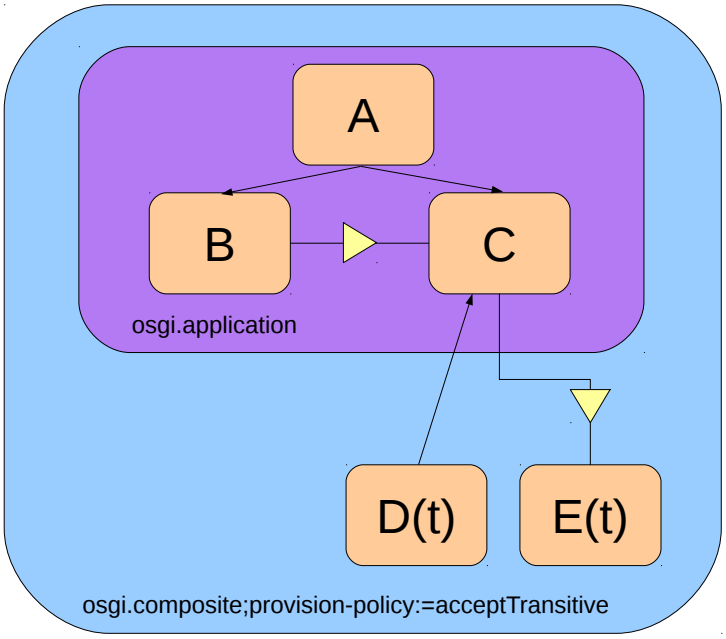


Figure 9: Simple transitive resource provisioning example

Figure 10 shows an example of the application from the previous example, but with the application opting in to hold transitive dependencies. Because the application itself is the first root encountered, the transitive bundles, D and E are installed into the application subsystem, rather than the parent.

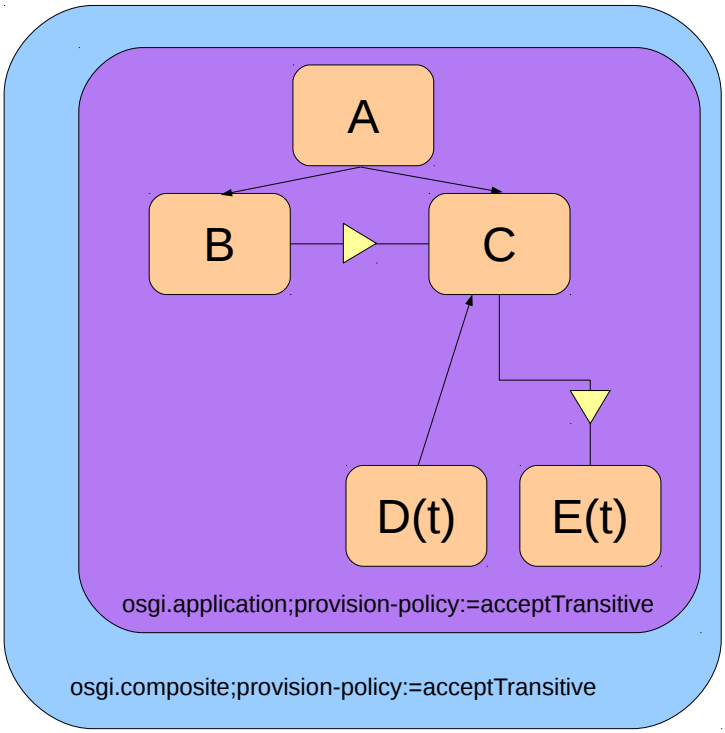


Figure 10: Transitive resource provisioning with multiple roots

Figure 11 shows an example where the isolation policy of one of the subsystems prevents the installed applications for accessing all the capabilities of its transitive dependencies. In this example, the first 'root' subsystem encounter is a two generations up from the application subsystem. This is where the transitive dependency bundles D and E are installed. However, the isolation imposed by an intermediate composite subsystem prevents the application from accessing the service provided by bundle E. This is considered a deployment error.

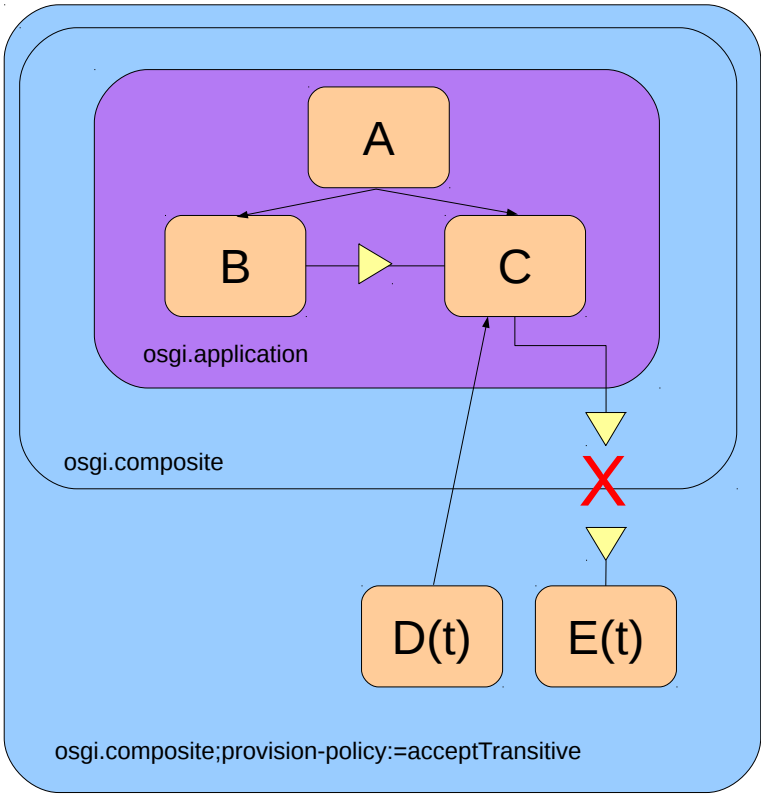


Figure 11: Transitive resource provisioning example with isolation error

5.2.6.2 Installation Failures

If the installation of a Subsystem fails, for example, due to problems opening an InputStream, then a subsystem `FAILED` event is fired. The exception that caused the failure is set in the event properties `EXCEPTION`, `EXCEPTION_CLASS` and `EXCEPTION_MESSAGE`. The subsystem id and location are also provided in the event to identify the failed subsystem, and finally, the timestamp is set to indicate when the failure occurred.

Failure to install one of the content resources (e.g. a content bundle) must result in a failure to install the subsystem and an attempt to undo any associated installation activities. The subsytem's state is then set to `UNINSTALLED` and the `UNINSTALLED` event fired via EventAdmin, if available.

5.2.6.3 bsnversion Property

The framework property `bsnversion` must be set to “managed” for subsystems. This will be the default value for the OSGi Core 4.4 release. Implementations will need to provide a Bundle Collision Hook as discussed in RFC 174.

5.2.7 Resolving a Subsystem

Resolution of a subsystem only happens as a result of an explicit request (i.e. by calling **Subsystem.start()**. Because start is an asynchronous operations there is no need for resolution to be done asynchronously. A subsystems state is only considered `RESOLVED` when all its content bundles are `RESOLVED`. The subsystem runtime issues a `RESOLVED` event when this condition is met.

To ensure a consistent and 'whole' view is presented for the composite subsystem externals, resolution happens in two steps. Resolution for Feature subsystems, which are transparent and Application subsystems, which do not export anything, can happen in a single step. Subsystem resolution occurs as follows:

1. Resolution is initiated with a call to **Subsystem.start()**.

2. If the subsystem is in the UNINSTALLING or UNINSTALLED state then a **SubsystemException** is thrown.

3. If the subsystem is already resolved, then no action is taken.

4. The subsystem's state is set to RESOLVING and a RESOLVING event is fired via EventAdmin, if available.

5. If the subsystem is one with isolation, then isolation is configured such that its content bundles are permitted to resolve against each other and any externals that it imports. If the subsystem is a composite then its exports are not yet made available for resolution.

6. An attempt is made to resolve the content bundles and any transitive dependencies that are not resolved (including subsystem dependencies).

7. If the subsystem fails to resolve (perhaps due to a content bundle not resolving) then the subsystem state is set to INSTALLED. A subsystem FAILED event is fired via EventAdmin, if available.

8. If the subsystem is a composite, then once all content bundles are RESOLVED, the subsystem makes any packages it exports available for resolution. Any subsequent resolution is then able to wire to packages provided by the composite subsystem. The subsystem's state is set to RESOLVED and a RESOLVED event is fired via EventAdmin, if available.

A consequence of this approach is that a composite subsystem is not permitted to be part of a package dependency cycle with other peer bundles or composite subsystems. If cycles exist then the composite subsystem will never resolve.

5.2.8 Starting a Subsystem

Subsystems are started by calling the Subsystem **start** method. Starting a subsystem performs the following steps:

1. If the Subsystem is in the UNINSTALLING or UNINSTALLED state, then a SubsystemException is thrown.

2. If the Subsystem is in the ACTIVE state, then the method returns immediately.

3. The starting process is initiated asynchronously and the method returns. From this point forward a framework restart must automatically start the subsystem (i.e. the subsystem's start state is persisted).

4. If there are existing content bundles that have not been resolved, then the runtime must attempt to resolve these first, as described in section 5.2.7. If the subsystem fails to resolve then the start does not continue.

5. Once all bundles are RESOLVED (i.e. the subsystem is resolved) then the subsystem's state is set to STARTING and if EventAdmin is available, an event of type STARTING is fired.

6. Resources are started in their start-level order irrespective of the framework start-level. The subsystem runtime is free to start bundles with the same start-level in any order it chooses. If a resource is already active (e.g. in the case of two intersecting feature subsystems), then it is skipped. Note, the subsystem content resources must only be transiently started, meaning the persistent start of the containing

subsystem will cause them to be started at the appropriate time. This avoids circumstances where content resources could be started ahead of the subsystem being set up.

7. Failure to start one of the content resources (e.g. a content bundle) must result in a failure to start the subsystem and an attempt to undo any associated activities. The subsystem state must be set to `RESOLVED`. The `FAILED` event is fired with the subsystem identification information, timestamp, and any exception information regarding the reason set in the appropriate event properties.

8. If all content resources successfully start (note, for some resource types, start may be a no-op, for example, in the case of a bundle fragment) then the subsystem's state is set to `ACTIVE`, and the `STARTED` event is fired via EventAdmin, if available.

Note, as each content bundle is activated, it may register and consume services before the entire subsystem is activated. This behavior is inconsistent for composite subsystems, where the design goal is to present a “whole view”, however, to achieve this would require complex alternatives, such as replaying service events.

5.2.9 Stopping a Subsystem

Subsystems are stopped by calling the Subsystem **stop** method. Stopping a subsystem performs the following steps:

1. If the Subsystem is in the `UNINSTALLING` or `UNINSTALLED` state then an `IllegalStateException` is thrown.

2. If the subsystem's state is not `STARTING` or `ACTIVE` then this method returns immediately.

3. The Subsystem's state is set to `STOPPING` and the `STOPPING` is fired via EventAdmin, if available.

4. The stopping process is initiated asynchronously and the method returns. From this point on, the subsystem stop state is persisted. The stopping process performs the following actions:

5. Resources are stopped in their reverse start-level order. The subsystems runtime is free to stop bundles with the same start-level in any order it chooses.

6. Failure to stop any resource must result in a `FAILED` event being fired with the subsystem identification information, timestamp, and any exception information regarding the reason set in the appropriate event properties. Stop processing continues to try to stop any remaining resources.

7. Once an attempt has been made to stop all resources, the Subsytem's state is set to `RESOLVED` and the `STOPPED` event is fired via EventAdmin, if available.

5.2.10 Stopping the Framework

When the framework is stopped, all subsystems must also be stopped before the start level starts taking down the bundles. This ensures subsystem stopping is managed appropriately by the subsystem runtime and not as an accidental and undesirable side-effect of the framework stopping individual bundles.

5.2.11 Unstalling a Subsystem

Subsystems are uninstalled by calling the Subsystem **uninstall** method. Uninstalling a subsystem results in it being put into the `UNINSTALLED` state and sending an `UNINSTALLED` event. The Framework must remove any resources related to this subsystem that it is able to remove. If this subsystem has exported any packages, the Framework must continue to make these packages available to their importing bundles or subsystems until the

PackageAdmin.refreshPackages method has been called or the Framework is relaunched. The following steps are required to uninstall a subsystem:

1. If this subsystem's state is UNINSTALLING or `UNINSTALLED` then this method returns immediately.
2. If this subsystem's state is `ACTIVE`, `STARTING` or `STOPPING`, this subsystem is stopped as described in section 5.2.9.
3. The Subsystem's state is set to `UNINSTALLING` and the `UNINSTALLING` event is fired via EventAdmin, if available.
4. The uninstall process is initiated asynchronously and the method returns. The uninstall process performs the following actions:
5. Resources are uninstalled.
6. Failure to uninstall any resource must result in a `FAILED` event being fired with the subsystem identification information, timestamp, and any exception information regarding the reason set in the appropriate event properties. Uninstall processing continues to try to uninstall any remaining resources.
7. Once an attempt has been made to uninstall all resources, the Subsytem state is set to `UNINSTALLED` and the `UNINSTALLED` event is fired via EventAdmin, if available.

5.2.12 Canceling a Subsystem operation

Asynchronous subsystem operations can be canceled through the Subsystem **cancel** method. Cancellation of a subsystem performs the following steps:

1. If the subsystem is not in a transitional state initiated through the Subsystem service (i.e. not `INSTALLING`, `UNINSTALLING`, `STARTING`, `UPDATING`, `STOPPING`), then an `IllegalStateException` is thrown.
2. It is permissible for an implementation to delay canceling if it is in the final steps of completing an in-flight operation. This would result in the cancel operation blocking briefly and then an `IllegalStateException` being thrown.
3. A `CANCELING` event is fired via EventAdmin, if available.
4. The cancel process is initiated asynchronously and the method returns. The cancel process then performs the following actions:
5. The runtime attempts to undo all actions performed as part of the current operation.
6. If the undo process fails for any resource, then a `FAILED` event is fired via EventAdmin, if available and the subsystems state remains in the transitional state.
7. If the undo process is successful, then the subsystem state is set to the state it was in prior to beginning the transition, and a `CANCELED` event is fired via EventAdmin, if available.

5.2.13 Retrieving Subsystems

Subsystems form a tree hierarchy. The child subsystems of a subsystem can be retrieved by calling the Subsystem **getChildren()** method. The parent subsystem of a subsystem can be retrieved by calling its **getParent()** method.

5.2.14 Shared Resource States

Resources can be shared between subsystems in two ways:

1. As common transitive resource dependencies between two sibling subsystems (see Figure 12).
2. As common content resources between two sibling feature subsystems (see Figure 13).

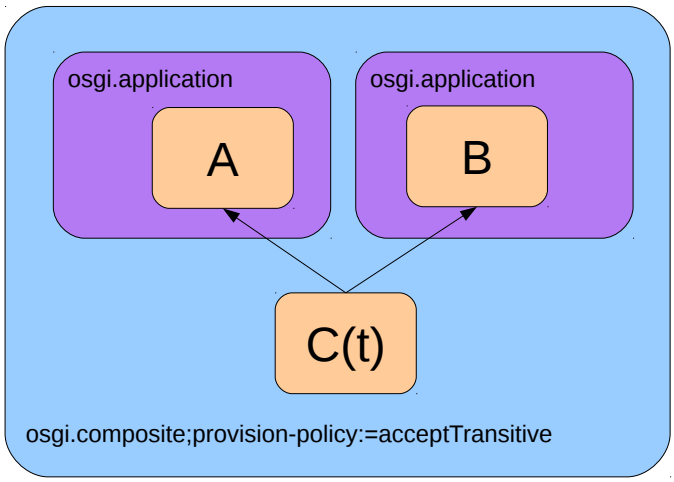


Figure 12: Two subsystems with a common transitive dependency

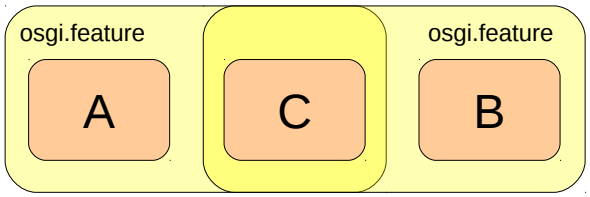


Figure 13: Two feature subsystems with shared common content

In both cases, the subsystems that cause the shared resource to be installed must share the responsibility for that resource's life-cycle. The shared resource life-cycle is managed such that it remains in the highest state required (see 5.2.14.1 for state precedence order) by each of the other resources with a dependency on that resource. This includes actions performed directly on the resource. For example, if the resource is installed and then is

subsequently used as a transitive dependency, uninstalling the thing which requires the resource as a transitive dependency does not unisnstall the resource.

Direct resource operations override the states required by things that depend on the resource. For example, performing an uninstall on the resource will uninstall that resource, irrespective of whether it is still required as a transitive dependency of another subsystem. This allow admin agents to perform updates of shared resources.

It's worth observing that when all things with requirements on transitive dependencies are uninstalled, their transitive dependencies will transition to uninstalled and this equates to garbage collection.

5.2.14.1 State Precedence

When managing the life-cycle of shared resources, a state precedence is used. The order of states is as follows:

1. ACTIVE
2. RESOLVED
3. INSTALLED
4. UNINSTALLED

In the case of a subsystem resource, transitional states are not considered. Instead, the target end state of the transition is used.

Figure 14 shows an example of two features with a common bundle dependency C. Feature F1 is ACTIVE and so its content bundles A and C are active. Feature F2 is RESOLVED. It's content bundle B which is not shared is also RESOLVED, but because its content bundle C is shared with F1 and F1's state is higher in the order, bundle C is in the ACTIVE state.

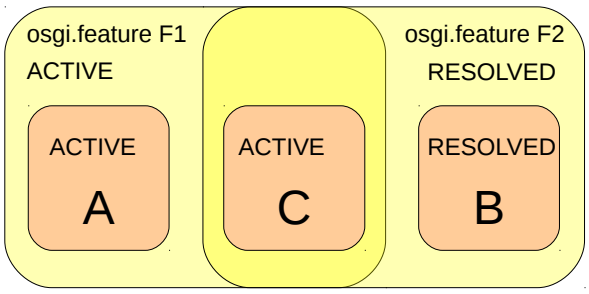


Figure 14: Shared resource state example

5.3 Subsystem Definitions

The following sections describe the headers used to describe subsystem definitions. As mentioned earlier, three types of subsystem are defined:

- Application – An implicitly scoped subsystem.
- Composite – An explicitly scoped subsystem.
- Feature – An unscoped subsystem.

A single set of headers is used to describe all subsystem types with one header identifying the type. Some headers are only valid for particular subsystem types.

- ***Type*** – the type of the subsystem, such as application, composite or feature.
- ***Identity*** – a symbolic name and version (Note, this is does not define uniqueness, as the same subsystem may be installed multiple times, although not in the same scope (i.e. they cannot be peers)).
- Name – A human readable name for the subsystem.
- ***Description*** – human readable information about the subsystem
- ***Content*** – The things that make up the subsystem content, such as bundles, configuration, and other subsystems.
- ***Bundle Sharing*** – ability to share bundles into or out of a subsystem.
- ***Package Sharing*** – the ability to share packages into or out of a subsystem. The defaults for packages sharing differ based on the type of subsystem (e.g. scoped vs unscoped).
- ***Service Sharing*** – the ability to share services into or out of a subsystem. The defaults for service sharing differ based on the type of subsystem (e.g. scoped vs unscoped).
- ***Nesting*** – the ability to define subsystems in terms of other subsystems.
- ***Transitive Closure*** – the ability to allow subsystems to define core content and have their remaining transitive dependencies calculated during deployment.
- ***Configuration*** – configuration for the subsystem and its contents.

5.3.1 Manifest Header Processing

Subsystem manifests follow the Jar manifest format, but to improve usability, there are two rules which are relaxed:

1. Line lengths can exceed the Jar manifest maximum of 72 bytes.

2. The last line is not required to be a newline.

5.3.2 Subsystem Manifest Headers

TODO: fit-n-finish on the headers (take a look at what we have for bundles and see what equivalents we need for subsystems.

5.3.2.1 Type Header

- `Subsystem-Type` – the type of the subsystem. This specification defines three subsystem types with the following values; `osgi.application`, `osgi.composite`, and `osgi.feature`. The default value is `osgi.application`. It is recommended that any new subsystem types have their names defined using a reverse domain naming scheme, e.g. `org.acme.subsystem.wibble`.

The following directive can be applied to the Subsystem-Type header:

- `provision-policy` – the provision-policy directive is used to describe additional qualities of the subsystem. This specification defines two values for the directive. 1) `root` – this means that the subsystem is considered to a be a root where its or its children's transitive resources can be provisioned. 2) `none` – there is no provision-policy and therefore the default subsystem behavior is unmodified. `None` is the default for this directive. The default value is `default`.

  Versioning Headers

- `Manifest-Version` – the version of the scheme for manifests that this manifest conforms to. This is the standard manifest header and is not defined by this specification. The default value is `1.0`.

- `Subsystem-ManifestVersion` – a subsystem manifest must conform to a version of the subsystem manifest headers. The version is defined by the OSGi specification and will be 1.0 for the first version.

5.3.2.2 Identity Headers

The following headers are used to specify the identity of asubsystem:

- `Subsystem-SymbolicName` – the symbolic name of the subsystem. Follows the same scheme and guidelines as `Bundle-SymbolicName`. To avoid confusion when viewing an environment that contains bundles and subsystems, it is recommended that different names be used for each. The default value is computed as described below.

- `Subsystem-Version` – the version of the subsystem (optional). Follows the same scheme and guidelines as `Bundle-Version`. The default value is `0.0.0` or computed as described below.

If not specified in the manifest, or the manifest is omitted, the following rules are defined for computing the default values of Subsystem-SymbolicName and Subsystem-Version. Manifest values override the defaults. If not otherwise specified, the default value of Subsystem-Version is always `0.0.0`. If the symbolic name is not provided in the manifest and cannot be computed by any of the following means, implementations may fail the installation.

1. When installing a subsystem via the subsystem interface (i.e. using one of the install methods), the following URI syntax must be used as the location in order to specify default values. Note that the optional URL component, if included, must be encoded.

subsystem-uri ::= 'subsystem://' (url)? '?' symbolic-name ('#' version)?

url ::= <see RFC 1738> // Retrieved via URI.getAuthority().

symbolic-name ::= <see OSGi Core Specification 1.3.2> // Retrieved via URI.getQuery().

version ::= <see OSGi Core Specification 3.2.5> // Retrieved via URI.getFragment().

2. When installing a subsystem with no content headers and containing other subsystem archives with no symbolic name or version, default values will be derived from the archive file names. The syntax is as follows.

   subsystem-archive ::= symbolic-name ('@' version) '.ssa'

3. When installing a subsystem with content headers having clauses pointing to other subsystems with no symbolic name or version, the path and version attribute of the corresponding clause will be used as the default symbolic name and version.

5.3.2.3 Informational Headers

The following headers are informational and are intended to be human readable:

- `Subsystem-Name` – the human readable subsystem name. Follows the same scheme and guidelines as `Bundle-Name`. The default value is the symbolic name.

- `Subsystem-Description` – the description of the subsystem. Follows the same scheme and guidelines as `Bundle-Description`.

5.3.2.4 Content Header

The following header identifies the content of a subsystem:

- `Subsystem-Content` – the content of the subsystem. This is the key content for the subsystem. At runtime, this content will be provisioned as a set which may or may not be isolated, depending on the type of the subsystem. See section 5.3.2.6 for the detailed definition of subsystem content.

5.3.2.5 Dependency Header

The following header is used to ensure a particular Bundle or Subsystem is used to satisfy implicit package dependencies of a subsystem (e.g. packages used by application or feature subsystems):

- `Preferred-Provider` – a set of bundles or subsystems to be preferred to satisfy the subsystem's implicit package imports. See section 5.3.2.7 for the detailed definition of use bundle.

Preferred providers are used to express a preference for a bundle or subsystem to satisfy implicit package dependencies. If a subsystem has an implicit import package that is satisfied by a bundle listed in Preferred-Provider then the package import should be tied to that bundle (e.g. through bundle-symbolic-name or bundle-version matching attributes or equivalent mechanism). Any bundles listed in Preferred-Provider that satisfy implicit imports are provisioned into the subsystem root (as is the case with any other bundles satisfying implicit package dependencies).

This mechanism enables subsystems to share bundles or subsystems with other subsystems whilst still ensuring they use the provider of the packages they expect.

5.3.2.6 Subsystem-Content

Subsystem content is a list of content resources that can be of various types. The default type is bundle. A bundle is identified by its symbolic name. For example

```
Subsystem-Content: com.acme.bundles.bundle1,
 com.acme.subsystems.subsystem1;type:=osgi.subsystem
```

The default value of Subsystem-Content is computed from all resource files included in the subsystem archive. It is an error to omit the content header and have no resource files in the archive.

The following matching attribute can be applied to bundle content:

- `version` – a version range used to select the bundle version of the bundle to use. This follows the OSGi version range scheme, including the default value of 0.0.0. For composite subsystems, this value must be a fixed version range (e.g. “[1.0, 1.0]”). This is due to the fact that there is an inextricable link between the versions on the explicit import and export statements made on a composite and the chosen versions of the content bundles. Allowing variability in the content versions

The following directives can be applied to bundle content:

- `resolution` – states whether the bundle must exist in order for the subsystem to be considered complete. A value of `mandatory` (the default) means the bundle must exist, a value of `optional` means the subsystem can be complete even if the bundle does not exist.

- `type` – indicates the type of the content. This value is used to uniquely identify the content resource. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundles). The default type is 'osgi.bundle'. A subsystem resource is identified with a type of 'osgi.subsystem'.

5.3.2.7 Preferred-Provider

Preferred provider is a list of bundles and/or subsystems that the subsystem explicitly uses packages from. A bundle or subsystem is identified by its symbolic name and an optional type directive. For example

```
Preferred-Provider: com.acme.bundles.bundle1,
 com.acme.subsystems.subsystem1;type:=osgi.subsystem
```

The following matching attribute can be applied:

- `version` – a version range used to select the version of the bundle or subsystem to use. This follows the OSGi version range scheme, including the default value of 0.0.0.

The following directive can be applied to the Preferred-Provider header:

- `type` – indicates the type of the preferred provider. This value is used to uniquely identify the provider when resolving the subsystem. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundle). The default type is 'osgi.bundle'. A subsystem resource is identified with a type of 'osgi.subsystem'.

Open Question: Is there a need for service affinity or other types of capability?

5.3.2.8 Explicit Sharing Headers

A composite subsystem can make explicit statements about service, package and bundle sharing. In fact, in order to share anything, it must make these explicit statements. These statements are made using the following headers. Note, these headers are not valid for application or feature subsystems as any sharing they support is completely implicit.

- `Import-Package` – specifies a list of package constraints to import into the composite. Uses the same syntax as bundle Import-Package. Any exported package from a bundle or composite that is a peer to this composite and which matches one of the constraints is available to satisfy any Import-Package constraints of the content bundles.

- `Export-Package` – specifies a list of package to export out of the composite. Uses the same syntax as bundle Export-Package. Any exports listed here must exactly match (i.e. they must have the same attributes and directives, but they can appear in a different order) an export from one of the content bundles.

- `Require-Bundle` – specifies a list of require bundle constraints to import into the composite. Uses the same syntax as Require-Bundle. Any bundle that is a sibling to the composite which matches one of the constraints is available to satisfy Require-Bundle constraints of the composite's content bundles.

- `Subsystem-ImportService` – specifies a list of service interfaces and optional service filters that control the services that are imported into the composite subsystem. Any services registered by peer bundles or exported by peer composites that match one of these filters are made available to the content bundles. The syntax for this header is as follows:

  ```
  Subsystem-ImportService ::= service-import ( ',' service-import )*
  service-import ::= interface-name ( ';' filter )?
  ```

- `Subsystem-ExportService` – specifies a list of service filters that control the services that are exported out of the composite. Any services registered by content bundles that match one of these filters are made available to peer bundles and composites. The syntax for this header is as follows:

  ```
  Subsystem-ExportService ::= service-export ( ',' service-export )*
  service-export ::= interface-name ( ';' filter )?
  ```

5.3.3 Subsystem Archive

A subsystem archive is a zip file ending in the extension .ssa (SubSystem Archive). Unlike jar archives, there are no special rules governing the order of the archive contents. A subsystem archive consists of the following:

1. The subsystem manifest. This is optional and if omitted will be calculated based on the rules defined in section Error: Reference source not found. The subsystem manifest is stored in the archive in a file named, OSGI-INF/SUBSYSTEM.MF.

2. A deployment manifest. This is optional and if omitted is calculated during provisioning of the subsystem, either by some management agent prior to calling **Subsystem.install** or as part of the **System.install** operation. It is anticipated that subsystem archives will rarely contain deployment manifests during development, but as the subsystem is tested prior to putting the subsystem into production, it will be common to add the deployment manifest to 'lock down' the bundles that are provisioned. The deployment manifest is stored in the archive in a file named, OSGI-INF/DEPLOYMENT.MF.

3. Any resources that may help with the provisioning of the subsystem. Note, it is feasible for a subsystem archive to contain no resources and have all it's contents and their dependencies provisioned from a repository. However, it is an error to install a subsystem whose archive contains no resources and no manifest.

5.3.4

5.3.5 Manifest Localization

TODO: Describe how headers are localized. This will follow the approach used for bundle manifest headers.

5.3.6 Deployment Manifest

A deployment manifest describes the bundles and resources that should be provisioned for a particular subsystem definition. The deployment manifest can be packaged with the subsystem or determined during deployment. A deployment manifest could be authored by hand but this is highly unlikely as the process is complex and would be error prone. Therefore, in most cases the deployment manifest will be calculated by a resolver, either out of band in some pre-deployment or pre-installation step, or during the installation of the subsystem, the latter making use of the OBR specification, identified in the subsystem architecture in figure 4.

5.3.6.1 Portability

A deployment manifest describes the bundles and resources that need to be provisioned to satisfy the subsystem definition. Resolution happens in the context of a target runtime, which could be a specific server instance, an empty framework, and so on. As such, a deployment manifest may not be transferable from one environment to another.

Whilst the format and location of deployment manifests is standardized it is not a requirement that a deployment manifest calculated for one runtime work in another. Installation of a subsystem can therefore determine whether the deployment manifest is valid and if not, choose to calculate a new deployment, or fail the installation. If the supplied deployment manifest is not valid and a valid deployment manifest cannot be calculated, then the installation be failed. Failing an installation is done by throwing a SubsystemException from the install method.

5.3.6.2 Design

The deployment manifest describes the resources to be provisioned for a subsystem, irrespective of the subsystem type. These resources fall into two categories:

1. Deployed content: the content that is to be deployed into the subsystem.

2. Provision resource: the resources to be provisioned outside the subsystem in support of its external dependencies.

A deployment manifest also describes configuration for the subsystem package and service isolation in the form of headers for the packages and services that are imported or exported by the subsystem. In the case of feature subsystems, which have no isolation, these headers are not used as they would simply list all packages and services imported or exported by the feature subsystem contents.

A deployment manifest is located in the OSGI-INF folder of the Subsystem archive file and is named DEPLOYMENT.MF.

As with Subsystem manifests, Deployment manifests follow the Jar manifest format, but to improve usability, there are two rules which are relaxed:

1. Line lengths can exceed the Jar manifest maximum of 72 bytes.

2. The last line is not required to be a newline.

5.3.6.3 Manifest Version Header

- `Manifest-Version` – the version of the scheme for manifest that this manifest conforms to. The default value is 1.0.

5.3.6.4 Content Header

- `Deployed-Content` – the content to be deployed for the subsystem. This is the exact content that is provisioned into the Composite Bundle that implements the Subsystem. The default value is computed from the Subsystem-Content header and resolution results. Basically, the value is the same except version ranges are replaced with exact versions. The exact details of the Deployed-Content header are described in section 5.3.6.5.

5.3.6.5 Deployed-Content

Deployed content is a list of exact content that can be of various types. The default type is bundle, including composite bundle. A bundle is identified by its symbolic name. For example

```
Deployed-Content: com.acme.bundles.bundle1,
 com.acme.bundles.bundle2
```

Each entry must uniquely identify the resource to be provisioned into the subsystem.

The following matching attribute can be applied to bundle content:

- `deployed_version` – the exact version of the resource to be deployed. Deployed version is a specific version, not a version range, hence the use of a new attribute name.

The following directive can be applied to resource content:

- `type` – indicates the type of the content. The default value is “osgi.bundle”. This specification defines “osgi.subsystem” for subsystems.

- start-order – The precedence the resource should have during the start sequence. Resources with lower start-order values are started before resources with higher values. Resources with the same start-order value may be started sequentially or in parallel. The default value is “1”.

5.3.6.6 Subsystem Identification

The subsystem to which the deployment manifest applies is identified by the subsystem's symbolic name and version headers, defined earlier. The specific headers for each subsystem type are as follows:

- `Subsystem-SymbolicName` and `Subsystem-Version`

The default values match the values of the headers with the same names in the subsystem manifest. It is an error if the deployment manifest is contained in a subsystem archive and the identification does not match that of the archive (either in the subsystem's definition manifest or one which would be generated through the defaulting rules).

5.3.6.7 Provisioning Header

- `Provision-Resource` – the resources to be provisioned in support of the subsystem's transitive dependencies. These are the specific bundles that are provisioned outside the Composite Bundle that implements the Subsystem. The default value is computed from the set of all resources necessary to resolve the subsystem minus any content resources defined in the Subsystem-Content header. The exact details of the Provision-Resource header are described in section 5.3.6.8.

5.3.6.8 Provision-Resource

Provision resource lists the resources to be provisioned in support of a subsystem's external dependencies. These dependencies can include package or service requirements.

Provision resource has one required matching attribute:

- `deployed_version` – the exact version of the resource to be deployed. Deployed version is a specific version, not a version range, hence the use of a new attribute name.

Provision resource has the following optional directive:

- `type` – indicates the type of the resource. This value is used to uniquely identify the resource. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundles). The default type is 'osgi.bundle'. A subsystem resource is identified with a type of 'osgi.subsystem'.

5.3.6.9 Package Imports

Isolating subsystems (i.e. applications and composites) describe the exact packages they import in their deployment manifests. They do this using the bundle Import-Package header. Any packages that match the Import-Package statement must be allowed to pass into the subsystem as candidates for resolving the subsystem contents. Features subsystems provide no isolation or resolution affinity and therefore there is no need to described their package imports in the deployment manifest. For composite subsystems, these are the Import-Package statements taken directly for the subsystem manifest, but for application subsystems these must be calculated as described below.

5.3.6.10 Application Import-Package Calculation

Application resolution requires their resolution to prefer packages provided by content bundles over those provided outside the application. For this reason, the deployment manifest only lists Import-Package statements from the content bundles that are not satisfied when resolving the application contents in isolation.

5.3.6.11 Package Exports

Only composite subsystems explicitly export packages. Any Export-Package statements in the composite's SUBSYSTEM.MF are re-iterated in the deployment manifest. Feature subsystems essentially export all packages and therefore there is not need to describe every packages exported by the feature's bundles in this header.

5.3.6.12 Bundle Requirements

Isolating subsystems (i.e. applications and composites) describe their bundle requirements in their deployment manifests. They do this using the bundle Require-Bundle header. Any bundles that match the Require-Bundle statement are allowed to pass into the subsystem to resolve Require-Bundle requirements from the content bundles. Features subsystems provide no isolation or resolution affinity and therefore there is no need to described their bundle requirements in the deployment manifest. For composite subsystems, these are the Require-Bundle statements taken directly for the subsystem manifest, but for application subsystems these must be calculated as described below.

5.3.6.13 Application Require-Bundle Calculation

Application resolution requires their resolution to prefer content bundles over those provided outside the application. For this reason, the deployment manifest only lists Require-Bundle statements from the content bundles that are not satisfied when resolving the application contents in isolation.

5.3.6.14 Service Imports

Composite subsystems explicitly import services for use by their content bundles. Any entries described in the Subsystem-ImportService header are mapped to entires in the Deployed-ServiceImport header. The mapping takes the service-import entries of the syntax described in section 5.3.2.8, and combines them into a single filter. So

```
service-import ::= interface-name ( ';' filter )?
```

becomes

```
service-import ::= combined-filter
combined-filter ::= '(&(objectClass=' interface-name ')' filter ')'
```

For example

```
Subsystem-ImportService: com.acme.ServiceInterface;filter="(size=large)"
```

Becomes

```
Deployed-ServiceImport: (&(objectClass=com.acme.ServiceInterface)(size=large))
```

Application subsystems do not explicitly import services. The services required to be imported into an application are essentially those required by the application's content bundles that are not also provided by the content bundles. There is no standard way to determine this, but a subsystem runtime is permitted to use it's own means to determine the services to import. Examples include resource metadata from a bundle repository, or analysis of bundle contents (e.g. Blueprint or Declarative Service XMLs). A subsystem runtime is then free to add Deployed-ServiceImport entries in order to complete the application's deployment manifest definition.

Feature subsystems do not isolate services and therefore there is no need to declare the services that are imported into the feature. All services outside the feature are available for use inside the feature.

5.3.6.15 Service Exports

Composite subsystems explicitly export services for use by peer bundles or subsystems. Any entries described in the Subsystem-ExportService header are mapped to entires in the Deployed-ServiceExport header. The mapping takes the service-export entries of the syntax described in section 5.3.2.8, and combines them into a single filter. So

```
service-export ::= interface-name ( ';' filter )?
```

becomes

```
service-export ::= combined-filter
combined-filter ::= '(&(objectClass=' interface-name ')' filter ')'
```

For example

```
Subsystem-ExportService: com.acme.ServiceInterface;filter="(size=large)"
```

Becomes

```
Deployed-ServiceExport: (&(objectClass=com.acme.ServiceInterface)(size=large))
```

Application subsystems do not export services outside the application and therefore it is an error to include the Deployed-ServiceExport header in an application deployment manifest.

Feature subsystems do not isolate services and therefore there is no need to declare the services that are exported into the feature. All services inside the feature are available for use outside the feature.

5.3.6.16 Provision-Resource

```
Manifest-Version: 1.0
Subsystem-SymbolicName: com.ibm.ws.eba.example.blog.app
Subsystem-Version: 1.0.0
Deployed-Content: com.ibm.ws.eba.example.blog.api;deployed-version=1.0
 .0,com.ibm.ws.eba.example.blog.persistence;deployed-version=1.0.0,com
 .ibm.ws.eba.example.blog.web;deployed-version=1.0.0,com.ibm.ws.eba.ex
 ample.blog.biz;deployed-version=1.0.0
Deployed-ServiceImport: (objectClass=com.ibm.ws.eba.example.blog.com
 ment.persistence.api.BlogCommentService)
Import-Package: com.ibm.json.java;version="1.0.0";bundle-symbolic-name
 ="com.ibm.json.java";bundle-version="[1.0.0,1.0.0]",javax.persistence
 ;version="1.0.0",javax.servlet.http;version="[2.5.0,3.0.0)",javax.ser
 vlet;version="[2.5.0,3.0.0)"
Provision-Resource: com.ibm.json.java;deployed-version=1.0.0
```

Figure 15: Example Deployment Manifest.

5.3.6.17 Deployment Manifest Creation

5.3.6.18

5.3.7 Configuration

Notes:

- Config Admin
- Security Permissions

5.4 Transitive Closure

Notes:

- Desire expressed to be able to control 'auto-provisioning'. Some debate as to whether that is expressed in the subsystem artefact or a policy of the environment into which the subsystem is deployed. The latter seemed more appropriate. - Management Agent can choose to do this or not.
- Need to be able to handle runtime requirements
- Need to be able to handle license requirements

5.4.1

5.5

6 JavaDoc

OSGi Javadoc

8/29/11 3:53 PM

Package Summary		Page
org.osgi.service.subsystem	Subsystem Package Version 1.0.	45

Package org.osgi.service.subsystem

Subsystem Package Version 1.0.

See:
Description

Interface Summary		Page
Subsystem	A representation of a subsystem in the framework.	46

Class Summary		Page
SubsystemConstants	Defines the constants used by subsystems.	55

Enum Summary		Page
Subsystem.State	The states of a subsystem in the framework.	52
SubsystemConstants.EVENT_TYPE	The subsystem lifecycle event types that can be produced by a subsystem.	62

Exception Summary		Page
SubsystemException	Exception thrown by Subsystem when a problem occurs.	66

Package org.osgi.service.subsystem Description

Subsystem Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.subsystem; version="[1.0,1.1)"
```

Interface Subsystem

org.osgi.service.subsystem

public interface **Subsystem**

A representation of a subsystem in the framework. A subsystem is a collection of bundles and/or other resource. A subsystem has isolation semantics. Subsystem types are defined that have different default isolation semantics. For example, an Application subsystem does not export any of the packages or services provided by its content bundles, and imports any packages or services that are required to satisfy unresolved package or service dependencies of the content bundles. A subsystem is defined using a manifest format.

ThreadSafe

Nested Class Summary		Page
static enum	**Subsystem.State** The states of a subsystem in the framework.	52

Method Summary		Page
void	**cancel**() Cancels the currently executing asynchronous life-cycle operation, if any.	47
Collection<Subsystem>	**getChildren**() Gets the subsystems managed by this service.	47
Collection<org.osgi.framework.resource.Resource>	**getConstituents**() Returns a snapshot of all Resources currently constituting this Subsystem.	47
Map<String,String>	**getHeaders**() Gets the headers used to define this subsystem.	47
Map<String,String>	**getHeaders**(String locale) Gets the headers used to define this subsystem.	48
String	**getLocation**() The location identifier used to install this subsystem through Subsystem.install.	48
Subsystem	**getParent**() Gets the parent Subsystem that scopes this subsystem instance.	48
Subsystem.State	**getState**() Gets the state of the subsystem.	48
long	**getSubsystemId**() Gets the identifier of the subsystem.	48
String	**getSymbolicName**() Gets the symbolic name of this subsystem.	49
org.osgi.framework.Version	**getVersion**() Gets the version of this subsystem.	49
Subsystem	**install**(String location) Install a new subsystem from the specified location identifier.	49
Subsystem	**install**(String location, InputStream content) Install a new subsystem from the specified InputStream object.	49
void	**start**() Starts the subsystem.	50
void	**stop**() Stops the subsystem.	50

void	**uninstall**() Uninstall the given subsystem.	51

Method Detail

cancel

```
void cancel()
     throws SubsystemException
```

Cancels the currently executing asynchronous life-cycle operation, if any.

Throws:

SubsystemException - - If this subsystem is not in one of the transitional states or the currently executing operation cannot be cancelled for any reason.

---

getChildren

```
Collection<Subsystem> getChildren()
```

Gets the subsystems managed by this service. This only includes the top-level Subsystems installed in the Framework, CoompositeBundle or Subsystem from which this service has been retrieved.

Returns:

The Subsystems managed by this service.

Throws:

IllegalStateException - If the subsystem is in the installing state or transitioned to the uninstalled state due to a failed installation.

---

getConstituents

```
Collection<org.osgi.framework.resource.Resource> getConstituents()
```

Returns a snapshot of all Resources currently constituting this Subsystem. If this Subsystem has no Resources, the Collection will be empty.

Returns:

A snapshot of all Resources currently constituting this Subsystem.

Throws:

IllegalStateException - If the subsystem is in the installing state or transitioned to the uninstalled state due to a failed installation.

---

getHeaders

```
Map<String,String> getHeaders()
```

Gets the headers used to define this subsystem. The headers will be localized using the locale returned by java.util.Locale.getDefault. This is equivalent to calling getHeaders(null).

Returns:

The headers used to define this subsystem.

Throws:

SecurityException - If the caller does not have the appropriate AdminPermission[this,METADATA] and the runtime supports permissions.

IllegalStateException - If the subsystem is in the installing state or transitioned to the uninstalled state due to a failed installation.

getHeaders

```
Map<String,String> getHeaders(String locale)
```

Gets the headers used to define this subsystem.

Parameters:
`locale` - The locale name to be used to localize the headers. If the locale is null then the locale returned by java.util.Locale.getDefault is used. If the value is the empty string then the returned headers are returned unlocalized.
Returns:
the headers used to define this subsystem, localized to the specified locale.
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

getLocation

```
String getLocation()
```

The location identifier used to install this subsystem through Subsystem.install. This identifier does not change while this subsystem remains installed, even after Subsystem.update. This location identifier is used in Subsystem.update if no other update source is specified.

Returns:
The string representation of the subsystem's location identifier.

getParent

```
Subsystem getParent()
```

Gets the parent Subsystem that scopes this subsystem instance.

Returns:
The Subsystem that scopes this subsystem or null if there is no parent subsystem (e.g. if the outer scope is the framework).
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

getState

```
Subsystem.State getState()
```

Gets the state of the subsystem.

Returns:
The state of the subsystem.

getSubsystemId

```
long getSubsystemId()
```

Gets the identifier of the subsystem. Subsystem identifiers are assigned when the subsystem is installed and are unique within the framework.

Returns:
The identifier of the subsystem.

---

getSymbolicName

```
String getSymbolicName()
```

Gets the symbolic name of this subsystem.

Returns:
The symbolic name of this subsystem.
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

---

getVersion

```
org.osgi.framework.Version getVersion()
```

Gets the version of this subsystem.

Returns:
The version of this subsystem.
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

---

install

```
Subsystem install(String location)
          throws SubsystemException
```

Install a new subsystem from the specified location identifier.

This method performs the same function as calling install(String, InputStream) with the specified location identifier and a null InputStream.

Parameters:
`location` - The location identifier of the subsystem to be installed.
Returns:
The installed subsystem.
Throws:
`SubsystemException` - If the subsystem could not be installed for any reason.
`SecurityException` - If the caller does not have the appropriate AdminPermission[installed subsystem,LIFECYCLE], and the Java Runtime Environment supports permissions.

---

install

```
Subsystem install(String location,
                  InputStream content)
          throws SubsystemException
```

Install a new subsystem from the specified InputStream object.

If the specified InputStream is null, the InputStream must be created from the specified location.

The specified location identifier will be used as the identity of the subsystem. Every installed subsystem is uniquely identified by its location identifier which is typically in the form of a URL.

TODO: Understand whether this all change when we can install the same bundle multiple times.

A subsystem and its contents must remain installed across Framework and VM restarts. The subsystem itself is installed atomically, however its contents are not.

The following steps are required to install a subsystem:

10. If there is an existing subsystem containing the same location identifier as the subsystem to be installed, then the existing subsystem is returned.
11. If this is a new install, then a new Subsystem is created with its id set to the next available value (ascending order).
12. The subsystem's state is set to INSTALLING and if EventAdmin is available, an event of type INSTALLING is fired.
13. The following installation steps are then started and performed asynchronously and the new subsystem is returned to the caller.
14. The subsystem content is read from the input stream.
15. If the subsystem requires isolation (i.e. is an application or a composite), then isolation is set up while the install is in progress, such that none of the content bundles can be resolved. This isolation is not changed until the subsystem is explicitly requested to resolve (i.e. as a result of a Subsystem.start() operation).
16. If the subsystem does not include a deployment manifest, then the subsystem runtime must calculate one.
17. The resources identified in the deployment manifest are installed into the framework. All content resources are installed into the Subsystem, whereas transitive dependencies are installed into an ancestor subsystem. If any resources fail to install, then the entire installation is failed. Transitive resources are free to resolve and start independent of the subsystem they were installed for.
18. The subsystem's state is set to INSTALLED and if EventAdmin is available an INSTALLED event is fired.

Parameters:

`location` - The location identifier of the subsystem to be installed.
`content` - The InputStream from where the subsystem is to be installed or null if the location is to be used to create the InputStream.

Returns:

The installed subsystem.

Throws:

`SubsystemException` - If the subsystem could not be installed for any reason.
`SecurityException` - If the caller does not have the appropriate AdminPermission[installed subsystem,LIFECYCLE], and the Java Runtime Environment supports permissions.

---

start

```
void start()
    throws SubsystemException
```

Starts the subsystem. The subsystem is started according to the rules defined for Bundles and the content bundles are enabled for activation.

Throws:

`SubsystemException` - If this subsystem could not be started.
`IllegalStateException` - If this subsystem has been uninstalled.
`SecurityException` - If the caller does not have the appropriate AdminPermission[this,EXECUTE] and the runtime supports permissions.

---

stop

```
void stop()
         throws SubsystemException
```

Stops the subsystem. The subsystem is stopped according to the rules defined for Bundles and the content bundles are disabled for activation and stopped.

Throws:

SubsystemException - If an internal exception is thrown while stopping the subsystem (e.g. a BundleException from Bundle.stop).
IllegalStateException - - If this subsystem has been uninstalled.
SecurityException - - If the caller does not have the appropriate AdminPermission[this,EXECUTE] and the runtime supports permissions.

---

uninstall

```
void uninstall()
        throws SubsystemException
```

Uninstall the given subsystem.

This method causes the Framework to notify other bundles and subsystems that this subsystem is being uninstalled, and then puts this subsystem into the UNINSTALLED state. The Framework must remove any resources related to this subsystem that it is able to remove. If this subsystem has exported any packages, the Framework must continue to make these packages available to their importing bundles or subsystems until the org.osgi.service.packageadmin.PackageAdmin.refreshPackages(org.osgi.framework.Bundle[]) method has been called or the Framework is relaunched. The following steps are required to uninstall a subsystem:

1. If this subsystem's state is UNINSTALLED then an IllegalStateException is thrown.
2. If this subsystem's state is ACTIVE, STARTING or STOPPING, this subsystem is stopped as described in the Subsystem.stop() method. If Subsystem.stop() throws an exception, a Framework event of type FrameworkEvent.ERROR is fired containing the exception.
3. This subsystem's state is set to UNINSTALLED.
4. A subsystem event of type SubsystemEvent.UNINSTALLED is fired.
5. This subsystem and any persistent storage area provided for this subsystem by the Framework are removed.

Throws:

SubsystemException - If the uninstall failed.
IllegalStateException - If the subsystem is already in the UNISTALLED state.
SecurityException - If the caller does not have the appropriate AdminPermission[this,LIFECYCLE] and the Java Runtime Environment supports permissions.

Enum Subsystem.State

org.osgi.service.subsystem

```
java.lang.Object
  └ java.lang.Enum<Subsystem.State>
      └ org.osgi.service.subsystem.Subsystem.State
```

All Implemented Interfaces:
Comparable<Subsystem.State>, Serializable

Enclosing class:
Subsystem

```
public static enum Subsystem.State
extends Enum<Subsystem.State>
```

The states of a subsystem in the framework. These states match those of a Bundle and are derived using the same rules as CompositeBundles. As such, they are more a reflection of what content bundles are permitted to do rather than an aggregation of the content bundle states.

Enum Constant Summary	Page
ACTIVE A subsystem is in the ACTIVE state when it has reached the beginning start-level (for starting it's contents), and all its persistently started content bundles that are resolved and have had their start-levels met have completed, or failed, their activator start method.	53
INSTALLED A subsystem is in the INSTALLED state when all resources are successfully installed.	53
INSTALLING A subsystem is in the INSTALLING state when it is initially created.	53
RESOLVED Â A subsystem is in the RESOLVED state when all resources are resolved.	53
RESOLVING Â A subsystem in the RESOLVING is allowed to have its content bundles resolved.	53
STARTING A subsystem is in the STARTING state when all its content bundles are enabled for activation.	53
STOPPING Â A subsystem in the STOPPING state is in the process of taking its its active start level to zero, stopping all the content bundles.	53
UNINSTALLED A subsystem is in the UNINSTALLED state when all its content bundles and uninstalled and its system bundle context is invalidated.	54
UNINSTALLING	54
UPDATING	54

Method Summary		Page
static Subsystem.State	**valueOf**(String name)	54
static Subsystem.State[]	**values**()	54

Enum Constant Detail

INSTALLING

public static final Subsystem.State **INSTALLING**

A subsystem is in the INSTALLING state when it is initially created.

INSTALLED

public static final Subsystem.State **INSTALLED**

A subsystem is in the INSTALLED state when all resources are successfully installed.

RESOLVING

public static final Subsystem.State **RESOLVING**

Â A subsystem in the RESOLVING is allowed to have its content bundles resolved.

RESOLVED

public static final Subsystem.State **RESOLVED**

Â A subsystem is in the RESOLVED state when all resources are resolved.

STARTING

public static final Subsystem.State **STARTING**

A subsystem is in the STARTING state when all its content bundles are enabled for activation.

ACTIVE

public static final Subsystem.State **ACTIVE**

A subsystem is in the ACTIVE state when it has reached the beginning start-level (for starting it's contents), and all its persistently started content bundles that are resolved and have had their start-levels met have completed, or failed, their activator start method.

STOPPING

public static final Subsystem.State **STOPPING**

Â A subsystem in the STOPPING state is in the process of taking its its active start level to zero, stopping all the content bundles.

UPDATING

```
public static final Subsystem.State UPDATING
```

UNINSTALLING

```
public static final Subsystem.State UNINSTALLING
```

UNINSTALLED

```
public static final Subsystem.State UNINSTALLED
```

A subsystem is in the UNINSTALLED state when all its content bundles and uninstalled and its system bundle context is invalidated.

Method Detail

values

```
public static Subsystem.State[] values()
```

---

valueOf

```
public static Subsystem.State valueOf(String name)
```

Class SubsystemConstants

org.osgi.service.subsystem

```
java.lang.Object
  └ org.osgi.service.subsystem.SubsystemConstants
```

```
public class SubsystemConstants
extends Object
```

Defines the constants used by subsystems.

Nested Class Summary		Page
static enum	SubsystemConstants.EVENT_TYPE The subsystem lifecycle event types that can be produced by a subsystem.	62

Field Summary		Page
static String	DEPLOYED_CONTENT Manifest header identifying the resources to be deployed.	56
static String	DEPLOYED_VERSION_ATTRIBUTE Manifest header attribute identifying the deployed version.	56
static String	EVENT_SUBSYSTEM_ID Key for the event property that holds the subsystem id.	57
static String	EVENT_SUBSYSTEM_LOCATION Key for the event property that holds the subsystem location.	57
static String	EVENT_SUBSYSTEM_SYMBOLICNAME Key for the event property that holds the subsystem symbolic name.	57
static String	EVENT_SUBSYSTEM_VERSION Key for the event property that holds the subsystem version.	57
static String	EVENT_SUBYSTEM_STATE Key for the event property that holds the subsystem state.	57
static String	EVENT_TOPIC The topic for subsystem event admin events.	57
static String	EVENT_TOPIC_INTERNALS The topic for subsystem internal event admin events.	57
static String	EXPORT_PACKAGE Manifest header identifying packages offered for export.	57
static String	IDENTITY_TYPE_ATTRIBUTE Manifest header attribute identifying the resource type.	58
static String	IDENTITY_TYPE_BUNDLE Manifest header attribute value identifying a bundle resource type.	58
static String	IDENTITY_TYPE_SUBSYSTEM Manifest header attribute value identifying a subsystem resource type.	58
static String	IMPORT_PACKAGE Manifest header identifying packages required for import.	58
static String	PREFERRED_PROVIDER Manifest header used to express a preference for particular resources to satisfy implicit package dependencies.	58
static String	PROVISION_RESOURCE Manifest header identifying the resources to be deployed to satisfy the transitive dependencies of a subsystem.	58

static String	**REQUIRE_BUNDLE** Manifest header identifying symbolic names of required bundles.	59
static String	**RESOLUTION_DIRECTIVE** Manifest header directive identifying the resolution type.	59
static String	**RESOLUTION_MANDATORY** Manifest header directive value identifying a mandatory resolution type.	59
static String	**RESOLUTION_OPTIONAL** Manifest header directive value identifying an optional resolution type.	59
static String	**START_LEVEL_DIRECTIVE** Manifest header directive identifying the start level.	59
static String	**SUBSYSTEM_CONTENT** The list of subsystem contents identified by a symbolic name and version.	59
static String	**SUBSYSTEM_DESCRIPTION** Human readable description.	59
static String	**SUBSYSTEM_EXPORTSERVICE** Manifest header identifying services offered for export.	60
static String	**SUBSYSTEM_IMPORTSERVICE** Manifest header identifying services required for import.	60
static String	**SUBSYSTEM_MANIFESTVERSION** The subsystem manifest version header must be present and equals to 1.0 for this version of applications.	60
static String	**SUBSYSTEM_NAME** Human readable application name.	60
static String	**SUBSYSTEM_SYMBOLICNAME** Symbolic name for the application.	60
static String	**SUBSYSTEM_TYPE** Manifest header identifying the subsystem type.	60
static String	**SUBSYSTEM_TYPE_APPLICATION** Manifest header value identifying an application subsystem.	60
static String	**SUBSYSTEM_TYPE_COMPOSITE** Manifest header value identifying a composite subsystem.	60
static String	**SUBSYSTEM_TYPE_FEATURE** Manifest header value identifying a feature subsystem.	61
static String	**SUBSYSTEM_VERSION** Version of the application.	61
static String	**VERSION_ATTRIBUTE** Manifest header attribute indicating a version or version range.	61

Field Detail

DEPLOYED_CONTENT

```
public static final String DEPLOYED_CONTENT = "Deployed-Content"
```

Manifest header identifying the resources to be deployed.

DEPLOYED_VERSION_ATTRIBUTE

```
public static final String DEPLOYED_VERSION_ATTRIBUTE = "deployed-version"
```

Manifest header attribute identifying the deployed version.

EVENT_SUBSYSTEM_ID

```
public static final String EVENT_SUBSYSTEM_ID = "subsystem.id"
```

Key for the event property that holds the subsystem id.

EVENT_SUBSYSTEM_LOCATION

```
public static final String EVENT_SUBSYSTEM_LOCATION = "subsystem.location"
```

Key for the event property that holds the subsystem location.

EVENT_SUBYSTEM_STATE

```
public static final String EVENT_SUBYSTEM_STATE = "subsystem.state"
```

Key for the event property that holds the subsystem state.

EVENT_SUBSYSTEM_SYMBOLICNAME

```
public static final String EVENT_SUBSYSTEM_SYMBOLICNAME = "subsystem.symbolicname"
```

Key for the event property that holds the subsystem symbolic name.

EVENT_SUBSYSTEM_VERSION

```
public static final String EVENT_SUBSYSTEM_VERSION = "subsystem.version"
```

Key for the event property that holds the subsystem version.

EVENT_TOPIC

```
public static final String EVENT_TOPIC = "org/osgi/service/Subsystem/"
```

The topic for subsystem event admin events.

EVENT_TOPIC_INTERNALS

```
public static final String EVENT_TOPIC_INTERNALS = "org/osgi/service/SubsystemInternals/"
```

The topic for subsystem internal event admin events.

EXPORT_PACKAGE

```
public static final String EXPORT_PACKAGE = "Export-Package"
```

Manifest header identifying packages offered for export.

See Also:
org.osgi.framework.Constants.EXPORT_PACKAGE

IDENTITY_TYPE_ATTRIBUTE

```
public static final String IDENTITY_TYPE_ATTRIBUTE = "type"
```

Manifest header attribute identifying the resource type. The default value is IDENTITY_TYPE_BUNDLE.

See Also:
org.osgi.framework.resource.ResourceConstants.IDENTITY_TYPE_ATTRIBUTE

IDENTITY_TYPE_BUNDLE

```
public static final String IDENTITY_TYPE_BUNDLE = "osgi.bundle"
```

Manifest header attribute value identifying a bundle resource type.

See Also:
org.osgi.framework.resource.ResourceConstants.IDENTITY_TYPE_BUNDLE

IDENTITY_TYPE_SUBSYSTEM

```
public static final String IDENTITY_TYPE_SUBSYSTEM = "osgi.subsystem"
```

Manifest header attribute value identifying a subsystem resource type.

IMPORT_PACKAGE

```
public static final String IMPORT_PACKAGE = "Import-Package"
```

Manifest header identifying packages required for import.

See Also:
org.osgi.framework.Constants.IMPORT_PACKAGE

PREFERRED_PROVIDER

```
public static final String PREFERRED_PROVIDER = "Preferred-Provider"
```

Manifest header used to express a preference for particular resources to satisfy implicit package dependencies.

PROVISION_RESOURCE

```
public static final String PROVISION_RESOURCE = "Provision-Resource"
```

Manifest header identifying the resources to be deployed to satisfy the transitive dependencies of a subsystem.

REQUIRE_BUNDLE

```
public static final String REQUIRE_BUNDLE = "Require-Bundle"
```

Manifest header identifying symbolic names of required bundles.

RESOLUTION_DIRECTIVE

```
public static final String RESOLUTION_DIRECTIVE = "resolution"
```

Manifest header directive identifying the resolution type. The default value is RESOLUTION_MANDATORY.

See Also:
org.osgi.framework.Constants.RESOLUTION_DIRECTIVE

RESOLUTION_MANDATORY

```
public static final String RESOLUTION_MANDATORY = "mandatory"
```

Manifest header directive value identifying a mandatory resolution type.

See Also:
org.osgi.framework.Constants.RESOLUTION_MANDATORY

RESOLUTION_OPTIONAL

```
public static final String RESOLUTION_OPTIONAL = "optional"
```

Manifest header directive value identifying an optional resolution type.

See Also:
org.osgi.framework.Constants.RESOLUTION_OPTIONAL

START_LEVEL_DIRECTIVE

```
public static final String START_LEVEL_DIRECTIVE = "start-level"
```

Manifest header directive identifying the start level.

SUBSYSTEM_CONTENT

```
public static final String SUBSYSTEM_CONTENT = "Subsystem-Content"
```

The list of subsystem contents identified by a symbolic name and version.

SUBSYSTEM_DESCRIPTION

```
public static final String SUBSYSTEM_DESCRIPTION = "Subsystem-Description"
```

Human readable description.

SUBSYSTEM_EXPORTSERVICE

```
public static final String SUBSYSTEM_EXPORTSERVICE = "Subsystem-ExportService"
```

Manifest header identifying services offered for export.

SUBSYSTEM_IMPORTSERVICE

```
public static final String SUBSYSTEM_IMPORTSERVICE = "Subsystem-ImportService"
```

Manifest header identifying services required for import.

SUBSYSTEM_MANIFESTVERSION

```
public static final String SUBSYSTEM_MANIFESTVERSION = "Subsystem-ManifestVersion"
```

The subsystem manifest version header must be present and equals to 1.0 for this version of applications.

SUBSYSTEM_NAME

```
public static final String SUBSYSTEM_NAME = "Subsystem-Name"
```

Human readable application name.

SUBSYSTEM_SYMBOLICNAME

```
public static final String SUBSYSTEM_SYMBOLICNAME = "Subsystem-SymbolicName"
```

Symbolic name for the application. Must be present.

SUBSYSTEM_TYPE

```
public static final String SUBSYSTEM_TYPE = "Subsystem-Type"
```

Manifest header identifying the subsystem type.

SUBSYSTEM_TYPE_APPLICATION

```
public static final String SUBSYSTEM_TYPE_APPLICATION = "osgi.application"
```

Manifest header value identifying an application subsystem.

SUBSYSTEM_TYPE_COMPOSITE

```
public static final String SUBSYSTEM_TYPE_COMPOSITE = "osgi.composite"
```

Manifest header value identifying a composite subsystem.

SUBSYSTEM_TYPE_FEATURE

```
public static final String SUBSYSTEM_TYPE_FEATURE = "osgi.feature"
```

Manifest header value identifying a feature subsystem.

SUBSYSTEM_VERSION

```
public static final String SUBSYSTEM_VERSION = "Subsystem-Version"
```

Version of the application. If not present, the default value is 0.0.0.

VERSION_ATTRIBUTE

```
public static final String VERSION_ATTRIBUTE = "version"
```

Manifest header attribute indicating a version or version range. The default value is `org.osgi.framework.Version.emptyVersion`.

Enum SubsystemConstants.EVENT_TYPE

org.osgi.service.subsystem

```
java.lang.Object
  └ java.lang.Enum<SubsystemConstants.EVENT_TYPE>
      └ org.osgi.service.subsystem.SubsystemConstants.EVENT_TYPE
```

All Implemented Interfaces:
Comparable<SubsystemConstants.EVENT_TYPE>, Serializable

Enclosing class:
SubsystemConstants

```
public static enum SubsystemConstants.EVENT_TYPE
extends Enum<SubsystemConstants.EVENT_TYPE>
```

The subsystem lifecycle event types that can be produced by a subsystem. See ? and Subsystem for details on the circumstances under which these events are fired.

Enum Constant Summary	Page
CANCELED Event type used to indicate that the operations was cancelled (e.g.	64
CANCELING Event type used to indicate that a subsystem operation is being cancelled.	64
FAILED Event type used to indicate that the operation failed (e.g.	65
INSTALLED Event type used to indicate a subsystem has been installed.	63
INSTALLING Event type used to indicate a subsystem is installing.	63
RESOLVED Event type used to indicate a subsystem has been resolved.	63
RESOLVING Event type used to indicate a subsystem is resolving.	63
STARTED Event type used to indicate a subsystem has been started.	63
STARTING Event type used to indicate a subsystem is starting.	63
STOPPED Event type used to indicate a subsystem has been stopped.	64
STOPPING Event type used to indicate a subsystem is stopping.	64
UNINSTALLED Event type used to indicate a subsystem has been uninstalled.	64
UNINSTALLING Event type used to indicate a subsystem is uninstalling.	64
UPDATED Event type used to indicate a subsystem has been updated.	64
UPDATING Event type used to indicate a subsystem is updating.	64

Method Summary		Page
static SubsystemConstants.EVENT_TYPE	**valueOf**(String name)	65
static SubsystemConstants.EVENT_TYPE[]	**values**()	65

Enum Constant Detail

INSTALLING

public static final SubsystemConstants.EVENT_TYPE **INSTALLING**

Event type used to indicate a subsystem is installing.

INSTALLED

public static final SubsystemConstants.EVENT_TYPE **INSTALLED**

Event type used to indicate a subsystem has been installed.

RESOLVING

public static final SubsystemConstants.EVENT_TYPE **RESOLVING**

Event type used to indicate a subsystem is resolving.

RESOLVED

public static final SubsystemConstants.EVENT_TYPE **RESOLVED**

Event type used to indicate a subsystem has been resolved.

STARTING

public static final SubsystemConstants.EVENT_TYPE **STARTING**

Event type used to indicate a subsystem is starting.

STARTED

public static final SubsystemConstants.EVENT_TYPE **STARTED**

Event type used to indicate a subsystem has been started.

STOPPING

public static final SubsystemConstants.EVENT_TYPE **STOPPING**

Event type used to indicate a subsystem is stopping.

STOPPED

public static final SubsystemConstants.EVENT_TYPE **STOPPED**

Event type used to indicate a subsystem has been stopped.

UPDATING

public static final SubsystemConstants.EVENT_TYPE **UPDATING**

Event type used to indicate a subsystem is updating.

UPDATED

public static final SubsystemConstants.EVENT_TYPE **UPDATED**

Event type used to indicate a subsystem has been updated.

UNINSTALLING

public static final SubsystemConstants.EVENT_TYPE **UNINSTALLING**

Event type used to indicate a subsystem is uninstalling.

UNINSTALLED

public static final SubsystemConstants.EVENT_TYPE **UNINSTALLED**

Event type used to indicate a subsystem has been uninstalled.

CANCELING

public static final SubsystemConstants.EVENT_TYPE **CANCELING**

Event type used to indicate that a subsystem operation is being cancelled.

CANCELED

public static final SubsystemConstants.EVENT_TYPE **CANCELED**

Event type used to indicate that the operations was cancelled (e.g. an install was cancelled).

FAILED

```
public static final SubsystemConstants.EVENT_TYPE FAILED
```

Event type used to indicate that the operation failed (e.g. an exception was thrown during installation).

Method Detail

values

```
public static SubsystemConstants.EVENT_TYPE[] values()
```

---

valueOf

```
public static SubsystemConstants.EVENT_TYPE valueOf(String name)
```

Class SubsystemException

org.osgi.service.subsystem

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ org.osgi.service.subsystem.SubsystemException
```

All Implemented Interfaces:
Serializable

```
public class SubsystemException
extends RuntimeException
```

Exception thrown by Subsystem when a problem occurs.

Constructor Summary	Page
SubsystemException() Construct a subsystem exception with no message.	66
SubsystemException(String message) Construct a subsystem exception specifying a message.	66
SubsystemException(String message, Throwable cause) Construct a subsystem exception specifying a message and wrapping an existing exception.	67
SubsystemException(Throwable cause) Construct a subsystem exception wrapping an existing exception.	66

Constructor Detail

SubsystemException

```
public SubsystemException()
```

Construct a subsystem exception with no message.

SubsystemException

```
public SubsystemException(String message)
```

Construct a subsystem exception specifying a message.

Parameters:
`message` - The message to include in the exception.

SubsystemException

```
public SubsystemException(Throwable cause)
```

Construct a subsystem exception wrapping an existing exception.

Parameters:
`cause` - The cause of the exception.

SubsystemException

```
public SubsystemException(String message,
                          Throwable cause)
```

Construct a subsystem exception specifying a message and wrapping an existing exception.

Parameters:

`message` - The message to include in the exception.
`cause` - The cause of the exception.

7 State Tables

7.1 State lock with cancelable operations

The following approach treats operations as uninterruptable with the exception of the explicit use of cancel:

1. Transitional states lock out other operations - causes them to be queued up.
2. If a queued operation cannot acquire the lock within a 'reasonable time period' then an exception is thrown for that operation.
3. Transitional states can be canceled through the subsystem cancel() operation which tries to return the subsystem to its previous state.
4. Errors are typically IllegalStateExceptions.
5. No-ops are shown by blank cells.

	cancel	install	uninstall	start	stop	update
INSTALLING	If installing aborted, UNINSTALLED. Otherwise, error.		Wait for Install to complete, then start Uninstall.	Wait for Install to complete then Start.		Wait for Install to complete then Update.
INSTALLED	Error		Uninstall	Resolve and Start.		Update
RESOLVING	If resolving aborted, INSTALLED. Otherwise, error.		Uninstall	Wait for resolve to complete then Start.		Wait for Resolve to complete then Update.
RESOLVED	Error		Uninstall	Start		Update
STARTING	If starting aborted, RESOLVED. Otherwise, error.		Wait for Start to complete then uninstall.		Wait for Start to complete then Stop.	Wait for Start to complete then Update.
ACTIVE	Error		Stop and Uninstall		Stop	Update
STOPPING	If stopping aborted, ACTIVE. Otherwise, error.		Wait for stop to complete then Uninstall.	Wait for Stop to complete then Start.		Wait for Stop to complete then Update.
UPDATING	If updating aborted, assume previous state. Otherwise, error.		Wait for Update to complete then Uninstall	Wait for Update to complete then Start.		Wait for Update to complete then Update.
UNINSTALLING	If uninstalling aborted, INSTALLED. Otherwise, error.	Error		Error	Error	Error
UNINSTALLED	Error	Error		Error	Error	Error

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

8.1 State Tables: Interruptable operations

Note: This approach was considered and rejected in favor of the simpler option outlined in section 7.1.

The following approach treats interrupts to transitional states as the norm. These interrupts are performed through the use of the Coordinator:

6. Transitional states are interrupted via Coordination. This can be done in one of two ways:
    1. A resource processor detects a problem and fails the Coordination.
    2. An admin agent chooses to cancel an in-flight operation (e.g. cancelling a start operation whilst in the STARTING state).
7. If the subsystem is in a state beyond (needs defining) that which would be achieved by processing the operation, then it is considered a no-op (e.g. install on an ACTIVE subsystem does not cause it to go to INSTALLED).
8. If an operation is called during a transitional state, and the transitional state is before the state that would be reached then it runs to completion and then performs the requested operation (e.g. start on an INSTALLING subsystem completes the install and then tries to reach ACTIVE).
9. Errors are typically IllegalStateExceptions.
10. No-ops are shown with blank cells.

	cancel	install	uninstall	start	stop	update
CREATED?		Install	Error	Install, Resolve and Start	Error	Error
INSTALLING	Fail the Coordination (end up in CREATED state? - bundles in UNINSTALLED?)		Fail Coordination (end up in CREATED state?)	Complete Install, then do Resolve and Start		Error
INSTALLED			Uninstall	Resolve and Start resources		Update
RESOLVING			Uninstall	Complete Resolve then Start		Complete Resolve then do Update
RESOLVED			Uninstall	Start		Update
STARTING	Fail the Coordination (end up in RESOLVED)		Fail the Start Coordination and Uninstall		Fail the Start (end up in RESOLVED)	Fail the Start then do Update
ACTIVE			Stop then Uninstall		Stop	Update
STOPPING	Fail the Coordination (end up in ACTIVE)		Complete the Stop then Uninstall	Fail the Stop Coordination (end up in ACTIVE)		Complete the Stop then do Update
UPDATING	Fail the Coordination (end up in state prior to update).		Fail the Update then Uninstall	Complete Update then do Start		Complete the Update then do next Update
UNINSTALLING	Fail the Coordination (end up in INSTALLED)	Error		Error	Error	Error
UNINSTALLED		Error		Error	Error	Error

8.2 Metadata Formats

XML was considered for the format of the Subsystem and Deployment definitions but rejected because the power and associated complexity/verbosity of XML was not required to express the subsystem information.

Java properties files were also considered for the format of the Subsystem and Deployment definitions but these suffered from the opposite problem from XML. Whilst Java properties files can use the same colon separator as Java manifests for name-value pairs and would therefore be familiar to OSGi developers, they do not have the concept of attributes and are therefore too simplistic to easily represent a Subsystem or Deployment definition.

8.3 Zip versus Jar

This design proposes the use of a zip file for the subsystem archive format. A design which used a jar format was considered for the following reason:

1. to enable subsystems to be installed as bundles using the extender pattern.
2. To enable the artefact to be signed.
3. To enable JarInputStream to be used to load them without needing to test first whether or not it was a zip.

After investigation the following conlusions were reached. Regarding 1, whist it seems attractive to manage a subsystem as a bundle, this would lead to strange lifecycle bundles living in the runtime. Also, these bundles would not be able to replace the need for a SubsystemAdmin services and therefore their value is limited. Regarding 2, the current belief is it is sufficient to sign the bundles contained within the archive, not the archive itself. If this is found not to be sufficient, then this discussion can be revisited. Regarding 3, it is simple to load a zip file using a JarInputStream and therefore this is a non-issue.

One concern that stems from the use of the jar format is the need to have a META-INF/MANIFEST.MF that serves no purpose other than to potentially confuse the developer.

8.4 Resource Processors

Resource processor design is inspired by deployment admin. A resource processor is a service that adds the ability to manage the life-cycle of resources on behalf of a subsystem. For example, a resource processor might manage subsystem configuration, or certificates.

8.4.1 Resource Processor Services

Each resource processor is registered in the service registry under the interface `org.osgi.service.resource.ResourceProcessor`. The types of resources a resource processor can handle are identified using the service property **osgi.resource.namespace**. The namespace values are the same as those used in the generic requirements and capabilities (RFC 154) and OBR (RFC 112), or should follow the best practice for defining custom namespaces. The type of this property is String+ to allow a single resource processor to provide support for multiple namespaces.

The Subsystem implementation must only use resource processor services that are visible to its implementation bundle. In other words the service must be registered in the *root scope*. The subsystem implementation must not look inside other scopes to find resource processors. Note, if a subsystem exports a resource processor up to the root scope then this resource processor becomes a candidate to be used by the subsystem runtime.

If there are multiple resource processors that support the same namespace, then the one returned by **getServiceReference** (i.e. highest ranking, lowest id (tie breaker)) must be used.

TBD: Describe how to 'observe' resource life-cycle without being 'the' resource processor (Resource Listener).

8.4.2 Resource Processor Life-cycle

TBD: Define what happens when you remove/replace a resource processor. Do we take down any dependent subsystems? Do we have a “forced” capability similar to that used by deployment admin?

8.4.3 Resource Processor Operations

Resource processors implement a **process** operation that is called for each of the life-cycle operations that can be performed on a subsystem itself (install, uninstall, start, stop and update). The **process** operation is passed a ResourceOperation that describes the following:

- The operation to be performed.
- The context of the operation (this can include a reference to the subsystem to which the resource belongs.
- The coordination under which the operation is being performed. If no coordination is provided, then the operation is performed independent of any others. If a coordination is provided then the resource processor must register as a participant and must fail the coordination if it fails to process the resource.

The subsystem runtime uses the coordinator service to coordinate the outcomes of these operations across resource processors for a single subsystem. The resource processor must process each request immediately and only return once the processing has completed. If a resource processor is notified that the coordination **failed** then it must cancel or attempt to undo any actions requested under that coordination. If the resource processor cannot undo all actions then it must throw an exception from the **Particpant.failed** call.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10Document Support

10.1 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

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	Guillaume Nodet
Company	Progress
Address	
Voice	
e-mail	gnodet@progress.com

Name	Graham Charters
Company	IBM
Address	
Voice	
e-mail	charters@uk.ibm.com

10.3 Acronyms and Abbreviations

10.4 End of Document

RFC 167 - SPI Service Loader support

Draft

19 Pages

Abstract

The Sun SPI model is a plug-in model widely used in the Java platform. However, there are issues with this model when used in an OSGi Framework.

This RFC proposes a solution to this problem in order to make the SPI model usable from within OSGi.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 11.1.

`Source code is shown in this typeface.`

0.3 Revision History

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

Revision	Date	Comments
Initial	October 2010	David Bosschaert, initial version
0.2	January 2011	David Bosschaert, enhanced technical solution section
0.3	March 2011	David Bosschaert, feedback from EEG concall review
0.4	April 2011	David Bosschaert, restricted solution to java.util.ServiceLoader, moved legacy support to Considered Alternatives. Also made the diagram in 5.8.1 more explicit.
0.5	June 2011	David Bosschaert, added content to sections 5.3, 5.4, 5.5, 6 and 7.

Revision	Date	Comments
0.6	August 2011	David Bosschaert, apply feedback from June F2F. Also removed MultiDelegationClassLoader example implementation code.
0.7	September 2011	David Bosschaert, incorporating feedback from BJ Hargrave

1 Introduction

The Sun SPI model is a plug-in model widely used in the Java platform. In fact, many implementations of JSR standards are plugged into the JRE using the Sun SPI model. However, there are issues with this model when used in an OSGi Framework.

This RFC proposes a solution to this problem in order to make the SPI model usable from within OSGi and enable this model for both existing applications that may have been developed outside of OSGi as well as for new applications that haven been developed as OSGi bundles.

2 Application Domain

The application domain is any OSGi-based Java application that uses a third-partly library which relies on the Sun SPI model.

There are two sides to the problem, the SPI provider side and the SPI consumer side.

2.1 SPI Providers

SPI providers are typically Jar files that contain an implementation of a standardized interface. An SPI provider could be OSGi-aware or non-OSGi-aware.

2.2 SPI Consumers

SPI consumers are clients of a standardized API that use it to obtain an implementation which is provided through the Sun SPI mechanism. These clients could be OSGi-aware or non-OSGi-aware.

3 Problem Description

In Java the most widely used pluggability model is the Sun SPI model. This model allows plugging in an implementation of typically a standardized interface such as a JAXP compliant XML parser, A JSR 311 compliant REST reader or writer or a SOAP implementation.

3.1 FactoryFinder

The SPI model is generally based on the following mechanics, although the implementations vary slightly across the board.

It generally uses a 'FactoryFinder' class which tries to obtain a Factory class which binds it to a certain implementation.

The Factory Finder typically does two things:

1. First it tries to work out the name of the factory class implementation for factory with id x.y.Z, typically using the following algorithm:
   a) It checks for the existence of a system property x.y.Z that might hold the class name.
   b) It looks for a well known properties file in $java.home/lib which might hold a value for the x.y.Z key.
   c) It tries to load the resource META-INF/services/x.y.Z using the Thread Context Classloader, the System Classloader or the classloader that loaded the Factory Finder class. If found it reads the implementation class name from the resource.
   d) If all of the above fails it uses a fall-back class name which is typically hard coded in the JRE implementation.
2. Once a class name has been obtained, the Factory Finder will try to load the class.
   a) In some cases an actual classloader is passed in to the Factory Finder in which case that classloader is used.
   b) Otherwise the Thread Context Classloader is tried or if that isn't set the System Classloader is used.
   c) If that fails the Factory Finder calls Class.forName() with its own classloader as the classloader argument.

This pattern has a number of disadvantages when used in an OSGi framework:

- A typical 3rd party library implementation that uses the SPI model relies on step 1c, where a resource is loaded from the META-INF/services directory through a classloader. In OSGi frameworks this resource is normally not visible outside of the library as only exported packages are made visible to classloaders outside of the bundle. Exporting the META-INF/services directory as a package is not an option as many libraries might have this directory but the OSGi framework can only resolve a package to a single bundle.

- Even if the class name can be obtained somehow in the OSGi framework, the loading of the implementation class will end up being challenging as it does not take the classloader of the bundle into account.

  This can often be worked around by setting the Thread Context Classloader to the bundle's classloader, but this requires modifications to the user's bundle code.

- To be loadable by another bundle the implementation class needs to be in a exported package of the providing bundle. This is generally undesirable as it exposes packages internal to an implementation outside its bundle, something highly discouraged in OSGi modularity.

- Besides the above issues, there is also a lifecycle issue in general with the SPI pattern as the Factory Finder often uses static variables and hence the Factory can only be set once in the life time of the VM in certain scenarios.

Since the Sun SPI model is highly prevalent in the Java library ecosystem and in fact the standard mechanism used within the JDK itself, an OSGi developer should not have to worry about getting around the problems with this mechanism. It should *just work*.

Other OSGi specifications have addressed this issue on a case-by-case basis, but a general solution to this problem is not available.

The OSGi Alliance should create a generic mechanism to deal with the Sun SPI model so that libraries utilizing this model can be used in an OSGi framework.

3.2 ServiceLoader

The JRE also contains a class called java.util.ServiceLoader which provides a general algorithm for this finding SPI implemetors. The algorithm is different than the one described above in that it only visits the META-INF/services. It doesn't involve system properties, nor files in the java home directory. It also doesn't have a default class name built in.

Additionally, the ServiceLoader class allows the client to obtain all the providers, where the FactoryFinder only selects a single one using the rules above.

3.3 JavaMail and non-empty constructors

The JavaMail API uses a SPI mechanism as well, but in a slightly different way. Classes are specified in a META-INF/javamail.properties file with a number of attributes, like this:

```
# JavaMail IMAP provider Sun Microsystems, Inc
protocol=imap; type=store; class=com.sun.mail.imap.IMAPStore; vendor=Sun;
protocol=imaps; type=store; class=com.sun.mail.imap.IMAPSSLStore; vendor=Sun;
```

The JavaMail framework expects the class specified to have a Constructor that has the following signature:

```
IMAPStore(Session session, URLName url)
```

When a consumer calls one of the Session.getStore() APIs this will result in the JavaMail framework to instantiate the correct class with the constructor as specified above.

During the RFP stage it has been decided that an anomaly like this would not be supported by this standard.

4 Requirements

4.1 Provider side Requirements

- SP01 – SPI service providers should be automatically registered in the service registry.
- SP02 – The solution should work with providers that are not OSGi-aware. These providers will need to be bundelized, however.
- SP03 – Providers need to opt-in using a Manifest Header.
- SP04 – Bundelized providers should support the full bundle lifecycle and should therefore be able to be uninstalled.

4.2 Client side Requirements

- SP10 – Non-OSGi-aware clients should work without modifications to the source code. It is not expected that these clients will get OSGi lifecycle benefits.
- SP11 – OSGi-aware Clients should be able to use OSGi services to obtain SPI implementations. These clients must get OSGi lifecycle benefits wrt to the SPI implementations.
- SP12 – The semantics for non-OSGi-aware clients must not change. For example clients must receive new SPI instances for every `ServiceLoader.load()` invocation.

5 Technical Solution

The technical solution aims to address both OSGi-aware use-cases as well as non-OSGi-aware use-cases.

Providers and consumers are described separately as they need to be handled in different ways.

5.1 Scope

The technical solution focuses on the how the bundle developer interacts with the system to get the desired behavior. There is a multititude of options available to achieve the specified behavior including RFC 159-based load-time weaving, static weaving, install-level interception, framework-level classloader control and possibly others. This RFC leaves the implementation choice to the implementor.

A pilot implementation is available in Apache Aries which uses an RFC 159 WeavingHook and ASM for the consumer side. On the provider side a BundleTracker is used.

5.2 Non-OSGi aware Providers

SPI Service Providers advertise their services through resources that are present in the `META-INF/services` directory of their jar file. This mechanism is supported by the JavaSE 6 `java.util.ServiceLoader` class and is also used in other implementations that predate Java 6. The following pattern is typically used (and this is the only one supported by this RFC): for every resource in the `META-INF/services` directory its name represents a Java class name or interface name. The contents of the resource is in text file format and lists implementation classes, one class per line. The implementation classes extend or implement the class or interface with the same name as the resource name.

For example, take a resource:

- `META-INF/services/org.acme.MyService`
  with contents:
  `org.acme.impl1.MyService1`
  `org.acme.impl2.MyService2`

This means that the jar file contains the classes MyService1 and MyService2 which implement or extend the MyService class. The MyService class does not need to be present in the jar file, but needs to be visible to the classloader that loads the jar file.

Note that it's quite common that the actual resource in the META-INF/services directory is actually located in a Jar file which is embedded in the bundle and visible throught the Bundle-ClassPath header. The discovery process needs to take this into account.

This RFC describes that Service Providers are handled in two ways by the system: firstly they are registered in the Service Registry to support OSGi-aware consumers, secondly the providing bundle or its classloader is recorded in an internal data structure which is used to support non-OSGi-aware consumers.

To enable OSGi support, Service Provider bundles need to opt in to the process by specifying the `SPI-Provider` header in the bundle manifest, using one of the following:

1. `SPI-Provider: *`

   When specified as such any services found in the `META-INF/services` directory will enabled.

2. `SPI-Provider: org.acme.Service1, org.acme.Service2`

   This allows a bundle to specify a subset of the services available in `META-INF/services` which should be enabled in OSGi.

5.2.1 OSGi Service Registrations

Service Providers are registered in the OSGi Service Registry as a ServiceFactory under the class name of the `META-INF/services` resource that declares them. Every consuming bundle will receive a separate instance of the service, which is similar to the semantics followed by `java.util.ServiceLoader.load()`.The objects will be instantiated using a no-arg constructor. The Service Factories are registered with the bundle context of the bundle that provides the implementation.

Service Properties

Property	*Data Type*	*Description*
spi.provider.url	String	A URL in String form that resolves to the resource that contained the service declaration. Besides being able to trace back where the service came from, the property also indicates that the service was registered through OSGi SPI support.

5.3 OSGi-aware Providers

OSGi-aware providers are bundles that provide services that could potentially be consumed via `java.util.ServiceLoader.load()`. OSGi-aware providers:

- register their own services in the OSGi Service Registry, so automatic registration of services found in `META-INF/services` is not needed.
- need to be able to play part in the `ServiceLoader.load()` lookup mechanism, for this they need to provide the relevant resource in the `META-INF/services` directory.

OSGi-aware providers need to opt-in to the lookup using a slightly alternative header:

```
ServiceLoader-Provider: *
```

The semantics of ServiceLoader-Provider are exactly the same as the semantics for SPI-Provider with the exception that no service will be registered in the Service Registry.

5.4 OSGi-aware Consumers

OSGi-aware consumers can obtain the services form the OSGi Service Registry. This is the preferred mechanism of interaction as it provides proper lifecycle support in case a service provider bundle is removed and/or a new service provider bundle is added to the system.

5.5 Non-OSGi-aware Consumers

Non-OSGi-aware consumers obtain service instances by calling `java.util.ServiceLoader.load()`.

Non-OSGi-aware Consumers can be made to work in on OSGi environment, but they do suffer from the disadvantage that they won't get lifecycle support, i.e. they will not get notified when a service provider is uninstalled. This is a consequence of the lack of lifecycle support in the design of `java.util.ServiceLoader`.

5.5.1 ServiceLoader.load(Class c)

As of Java 6 `java.util.ServiceLoader.load(Class c)` is the suggested API that SPI Service Consumers use. This is a static API of which returns an instance of the ServiceLoader class which in itself is Iterable; iterating through the result will produce the actual service instances found. ServiceLoader is a final class and has a private constructor so it's impossible to return an alternative implementation of ServiceLoader. We need to influence the way ServiceLoader works within its current design. ServiceLoader internally uses the ThreadContextClassLoader to find service implementations. Additionally, ServiceLoader provides a `load(Class c, ClassLoader cl)` method, which takes the classloader to use instead of relying on the TCCL. To make the API work in an OSGi environment an implementation has among its options:

- Use byte code manipulation to change the caller's invocation from `ServiceLoader.load(Class c)` to `ServiceLoader.load(Class c, ClassLoader cls)`

- Use byte code manipulation to set the TCCL for the duration of the `ServiceLoader.load(Class c)` call. In this case care must be taken to set the TCCL back to its previous value in a finally block after the call returns.

In both the cases above, the classloader provided must have visibility of the appropriate bundles for the duration of the `ServiceLoader.load()` call.

In addition, only bundles that provide class-compatible implementations of the service are considered to be appropriate. This can be verified by loading the requested class from the provider and checking that the class returned is the same object as the requesting class.

Note that there may be other ways in which to achieve the desired behavior, this is considered to be an implementation detail.

A consumer bundle that wishes to use this behavior for the duration of ServiceLoader.load(Class c) can simply opt-in by specifying the following header:

```
SPI-Consumer: *
```

The above includes any suitable provider bundle in the ServiceLoader.load() process.

5.5.1.1 Targeting specific bundles

To target a specific bundle or bundles to take part in `ServiceLoader.load()` requests, the provider bundle can be specified in the SPI-Consumer header. The bundle can be specified by Symbolic Name and Version or Bundle ID. Optionally a version can be specified.

```
SPI-Consumer: *;filter:=(bundle-symbolic-name=myBundle1)
```

load SPI providers from myBundle1, as no version is specified, the highest available version is selected.

```
SPI-Consumer: *;filter:=(|(&(bundle-symbolic-name=myBundle1)(bundle-
version="[1.2,1.3)"))(&(bundle-symbolic-name=myBundle2)(bundle-version=2)))
```

load SPI providers from myBundle1 with version between 1.2 inclusive and 1.3 exclusive or myBundle2 with version 2

```
SPI-Consumer: *;filter:=(bundle-id=0)
```

load SPI providers only from the bundle with ID 0, i.e. the system bundle.

5.5.1.2 Targeting specific bundles per service type

It is possible to distinguish which bundle or set of bundles should take part in `ServiceLoader.load()` requests based on the actual service type requested. The following header shows this:

```
SPI-Consumer:
org.acme.MyService1;filter:=(bundle=myBundle1),org.acme.MyService2;filter:=(|
(bundle=myBundle2)(bundle=myBundle3))
```

Calls to java.util.ServiceLoader.load(org.acme.MyService1.class) are only handled in the context of myBundle1, while calls to java.util.ServiceLoader.load(org.acme.MyService2.class) are handled in the context of myBundle2 and myBundle3.

5.5.2 SPI-Consumer header

In general, the SPI-Consumer header has the following syntax:

```
SPI-Consumer: (*|fqcn){;filter}{,(*|fqcn){;filter}}*
```

Supported values:

Element	*Example(s)*	*Description*
fqcn	* org.acme.MyService	When '*' is specified all ServiceLoader.load() calls are treated, when a class name is specified, only ServiceLoader.load() calls for that specific SPI are treated.

Supported filter attributes are:

Attribute	*Example(s)*	*Description*
bundle-symbolic-name	filter:=(bundle-symbolic-name=mySPiImpl1)	If specified only bundles with a matching symbolic name are eligible to provide the service to this consumer. If not specified any bundle potentially qualifies.
bundle-version	filter:=(&(bundle-symbolic-name=mySPiBundle)(bundle-version="[1.2,2)"))	If specified only bundles with the specified version or version range are eligible.
bundle-id	filter:=(bundle-id=) filter:=(|(bundle-id=2)(bundle-id=3))	The bundle ID or IDs of bundles that are eligible to provide the service.

5.6 Handling JRE-provided implementations

These can be selected by requesting that they be provided from bundle with ID 0.

If no bundle restriction is specified, JRE-provided implementations are returned as part of the ServiceLoader.load() call as normal.

TODO add a more detailed example

5.7 RFC 153 Capability and Requirement

To ensure the availability of the correct bundles at resolve-time, a bundle depending on this mechanism can declare this by specifying a Require-Capability attribute in its manifest:

```
Require-Capability: osgi.jre.serviceloader
```

The implementor of this mechanism should provide the capability. This can be done in the following ways:

- If support is provided at runtime, the entity providing the support should specify the following manifest header:
  `Provide-Capability: osgi.jre.serviceloader`

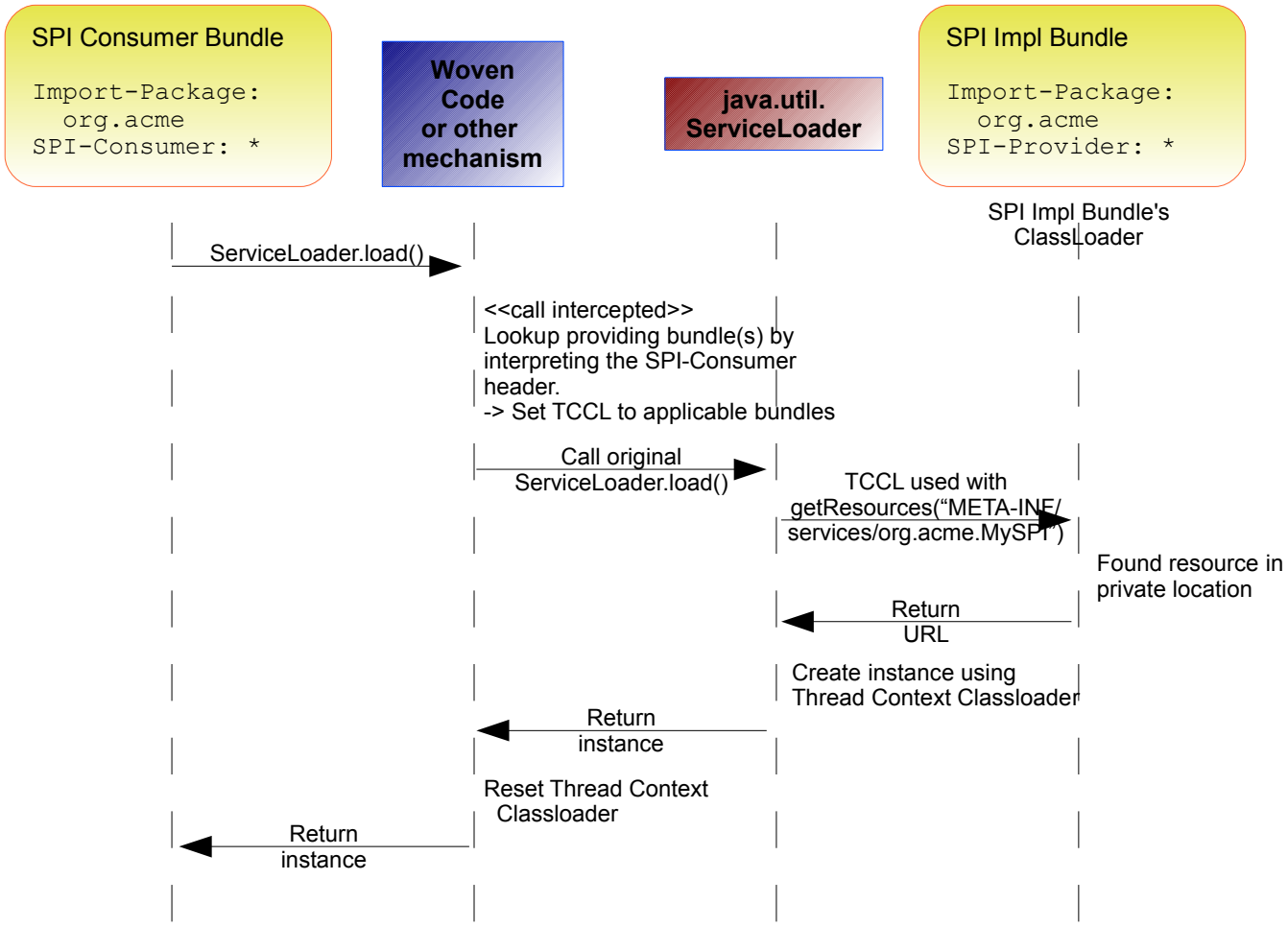
- If support is provided statically, e.g. through a tool that statically provides the necessary actions, the Require-Capability header should be removed from the depending bundle by the tool after having modified it.

This feature is currently limit to resolve-time matching. A management agent should ensure that the required bundles are started at runtime to enable the extension process.

5.8 Examples

5.8.1 Bundle directly using ServiceLoader.load()

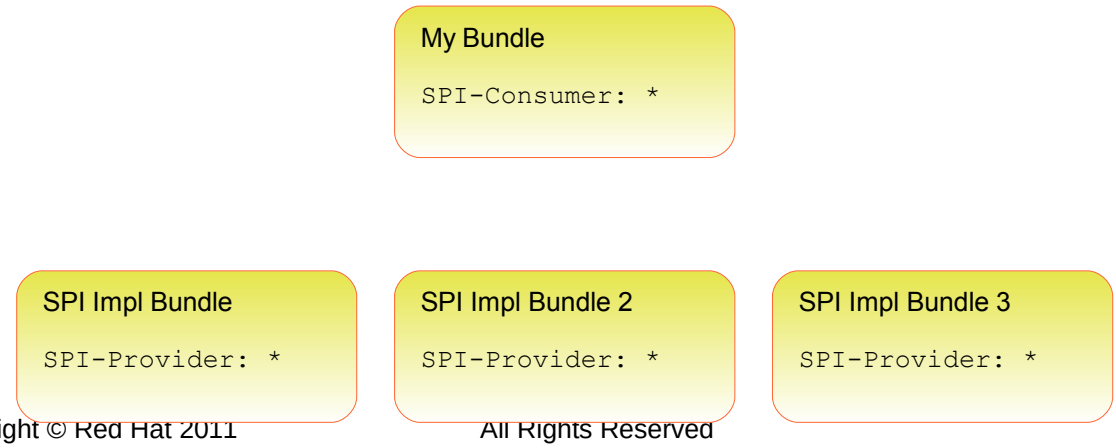
A consumer bundle uses the ServiceLoader.load() API directly.



** This diagram only depicts the logical flow. The actual loading of the resource and instantiation happens lazily on the ServiceLoader iterator callback.*

5.8.2 Bundle using a library that collects several service implementations using ServiceLoader.load()

The ServiceLoader.load() API can return multiple instances of the service. In turn these services should be allowed to come from various bundles.



ServiceLoader internally only calls ClassLoader.getResources() once, however the desired result can be achieved by using a delegation classloader that delegates to a number of other classloader, like the Error: Reference source not found from section Error: Reference source not found as the Thread Context Classloader for the duration of the ServiceLoader.load() call.

6 Command Line API

There are no new commands needed for this RFC. All the metadata is held in manifest headers and the processing is taken care of by the extender.

7 JMX API

There is no new JMX API needed for this RFC. All the metadata is held in manifest headers and these can already be accessed through JMX.

8 Initial Spec Chapter

Provide a link to where the Initial Spec Chapter can be found. The Initial Spec Chapter is typically written by someone other than the author(s) of this RFC and represents a rewrite of this document as close as possible to what will ultimately appear in the OSGi Specifications. It will be used by the Specification Editor as the basis for the ultimate specification chapter.

The spec template and writing guidelines can be found here:

https://www.osgi.org/members/svn/documents/trunk/templates/specification-template-oo.ott

https://www.osgi.org/members/svn/documents/trunk/templates/specwriting.pdf

9 Considered Alternatives

9.1 Technical Solution

9.1.1 'Legacy' consumers

Before the ServiceLoader class became part of the JRE in Java 6, a number of technologies existed that used their own proprietary classes to handle providers that advertise themselves through resources in the META-INF/services directory. Examples of such technologies include JAXP, JAXB, JAX-WS, media codecs and others. Although these technologies have their own implementation of a 'ServiceLoader' class, (often called 'FactoryFinder'), the mechanism that they use is roughly similar and is generally also influenced by the Thread Context Classloader. So a treatment similar to that of the `ServiceLoader.load(Class)` method is also helpful here.

Legacy consumers are also configured through the SPI-Consumer header, e.g to select the JAXP from a bundle with BSN apache-xerces, specify the header as follows:

```
SPI-Consumer: javax.xml.parsers.DocumentBuilderFactory#newInstance();bundle=apache-xerces
```

While JAXP can be selected from the JRE by obtaining it through the System Bundle:

```
SPI-Consumer: javax.xml.parsers.DocumentBuilderFactory#newInstance();bundleId=0
```

There is a difference in how `ServiceLoader.load()` is treated because ServiceLoader.load() is a generic mechanism: the service API is provided as the argument to `ServiceLoader.load()` while in the other cases the class on which the invocation is made is itself the API. To clarify, a simple header of

```
SPI-Consumer: javax.xml.parsers.DocumentBuilderFactory#newInstance()
```

Will enable the SPI consumer mechanism for any provider of the DocumentBuilderFactory service.

While

```
SPI-Consumer: java.util.ServiceLoader#load(java.lang.Class[org.acme.MySvc])
```

Will enable the mechanism for the MySvc service when obtained via `ServiceLoader.load()`.

9.2 Examples

9.2.1 Bundle using JAXP DocumentBuilderFactory.newInstance()

In this case the consumer bundle directly calls a JAXP API:

javax.xml.parsers.DocumentBuilderFactory.newInstance()

As this API deviates from the standard ServiceLoader.load() the SPI-Consumer header in the consumer bundle needs to specify the actual API.

JAXP Consumer Bundle

```
Import-Package javax.xml.parsers
SPI-Consumer:
  javax.xml.parsers.DocumentBuilderFactory#newInstance()
```

JAXP Impl Bundle

```
SPI-Provider: *
```

9.2.2 Bundle using library that uses ServiceLoader.load() and DocumentBuilder-Factory.newInstance()

An ordinary bundle uses an API which in turn uses both JAXP as well as another SPI provider bundle.

My Bundle

```
Import-Package:
  org.acme.foo
```

Library Bundle

```
Export-Pacakge: org.acme.foo
Import-Package javax.xml.parsers
SPI-Consumer: java.util.ServiceLoader#load(),
  javax.xml.parsers.DocumentBuilderFactory#newInstance()
```

JAXP Impl Bundle

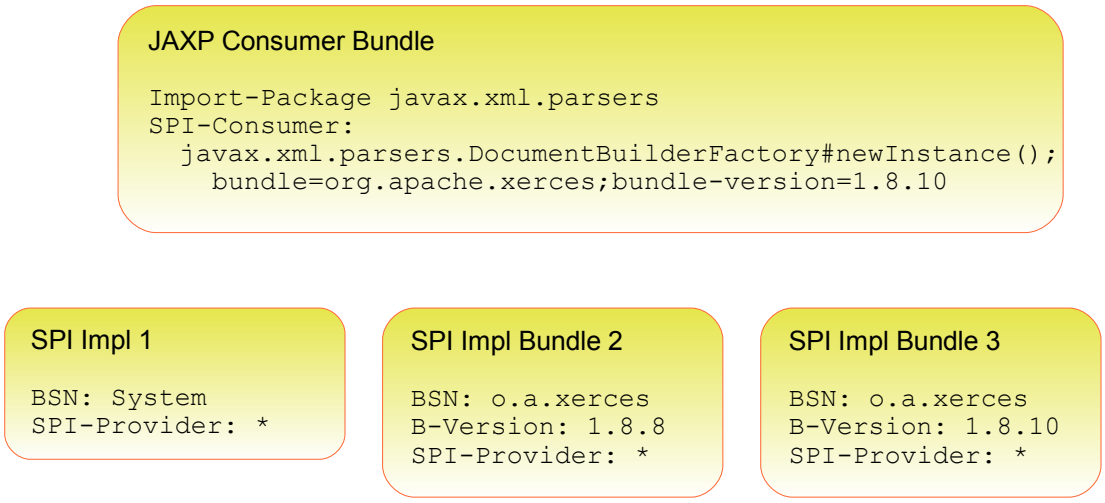
```
SPI-Provider: *
```

SPI Impl Bundle

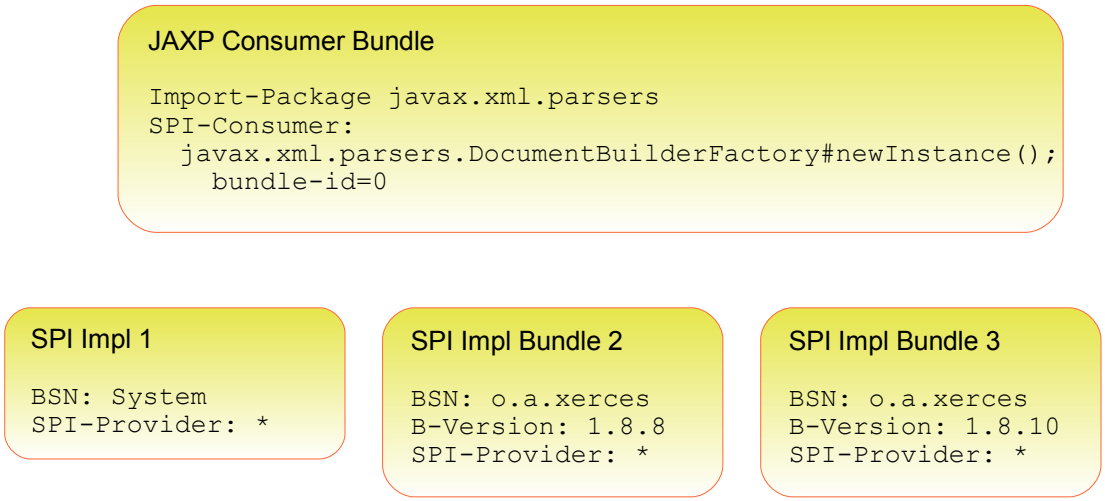
```
SPI-Provider: *
```

The SPI-Consumer header needs to be in the Library bundle. The consumer bundle that indirectly consumes the SPI services does not need any specific headers.

9.2.3 Bundle using JAXP from an Apache bundle, not from the System bundle nor any other provider



9.2.4 Bundle using JAXP from an from the System bundle not from any other provider bundle



9.2.5 JAXB anomaly

JAXB is initialized by calling JAXBContext.newInstance(). The files in the META-INF/services are indeed called javax.xml.bind.JAXBContext but the class name in there refers to a class called com.sun.xml.bind.v2.ContextFactory which is ***not*** a subclass of JAXBContext. On that class a method called createContext(String, Classloader, Map) is called which returns a JAXBContext object.

Open question: what will we do with JAXB? Will we attempt to support this?

9.2.6 Obtaining the SPI service from the OSGi Service Registry

10 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

Since a bundle may be running in a secure framework (i.e. installed SecurityManager), the woven code must take that into account.

At the weaving site, one must be careful writing code which requires a permission not granted to the bundle. Such code should be in the support bundles which can perform any necessary doPrivilege.

This does raise the runtime weaving issue of PackagePermission for any package imports added by the weaver. The weaver must also adjust the permissions of the bundle being woven to allow import of any referenced packages which are added to WovenClass.

11 Document Support

11.1 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

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

11.2 Author's Address

Name	David Bosschaert
Company	Red Hat
Address	6700 Cork Airport Business Park Kinsale Road Cork Ireland
Voice	+353 21 230 3400
e-mail	david@redhat.com

Name	
Company	
Address	
Voice	
e-mail	

11.3 Acronyms and Abbreviations

11.4 End of Document

RFC 169 - JMX Update

Draft

23 Pages

Abstract

Chapter 124 of the OSGi Enterprise Specification describes the how the OSGi Framework can be managed through the JMX Management Model. This RFC updates the OSGi-JMX specification to follow the OSGi 4.3 Core framework updates. Additionally it addresses a number of bugs filed in relation to the OSGi-JMX specification.

1 Document Information

1.1 Table of Contents

1.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 11.1.

`Source code is shown in this typeface.`

1.3 Revision History

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

Revision	Date	Comments
Initial	December 2010	David Bosschaert, Initial Version
0.1	January 2011	David Bosschaert, added JMX Notification to problem description.
0.2	January 2011	Alexandre Alves, First draft of technical solution
0.3	January, 2011	Alexandre Alves, Incorporating David's feedback.
0.4	February, 2011	Alexandre Alves, Incorporating feedback from Berlin's face-to-face meeting.
0.5	March, 2011	Alexandre Alves, Supporting Bundle Wiring changes
0.6	May, 2011	David Bosschaert, added additional bugs 87 and 109 from public OSGi Bugzilla.
0.7	June, 2011	Alexandre Alves, Java API update
0.8	August, 2011	Alexandre Alves, BundleRevisions MBean

Revision	Date	Comments
0.9	August/September, 2011	David Bosschaert, Small changes after re-reading the document and checking all the related bugs.

2 Introduction

A number of bugs and emailed suggestions have surfaced with regard to the OSGi-JMX specification. Additionally, the core framework has been significantly updated for version 4.3 with new APIs. This RFC aims to update the OSGi JMX specification to address the concerns and suggestions made, as well as support the updated Core Framework API where applicable.

3 Application Domain

This RFC applies to access to the OSGi Framework through the Java Management Extensions (JMX) Technology.

4 Problem Description

This RFC is based on a number of bugs and otherwise communicated enhancements and fixes to the exsting OSGi-JMX specification. There is no RFP that this relates to besides the original OSGi-JMX RFP 105.

4.1 OSGi 4.3 Core Framework Update (Bug 1762)

The OSGi 4.3 Core framework has been enhanced with a number of APIs such as `BundleRevision`, `BundleWiring`, `BundleWirings`, `FrameworkWiring` all of which are obtained through a the `Bundle.adapt()` method. The JMX API needs to updated accordingly, where applicable.

4.2 JBoss OSGi-JMX extensions

While working with the OSGi JMX support the JBoss OSGi framework enhanced the JMX support with the following:

4.2.1 BundleStateMBean

http://github.com/jbosgi/jbosgi-jmx/blob/master/api/src/main/java/org/jboss/osgi/jmx/BundleStateMBeanExt.java

4.2.1.1 Bug 1598 - [JMX] BundleStateMBean does not allow to introspect single bundle

Suggested API:

```
CompositeData getBundle(long bundleId) throws IOException;
```

4.2.1.2 Bug 1599 - [JMX] BundleStateMBean does not allow localized header access

Suggested API:

```
TabularData getHeaders(long bundleId, String locale) throws IOException;
```

4.2.1.3 Bug 1600 - [JMX] BundleStateMBean does not allow access to bundle properties

Suggested API:

```
CompositeData getProperty(long bundleId, String key) throws IOException;
```

This has been clarified to mean Framework properties.

4.2.1.4 Bug 1601 - [JMX] BundleStateMBean possible additions

This bug contains suggested enhancements which are useful for debugging/troubleshooting purposes. The suggested APIs include:

```
long loadClass(long bundleId, String name);
String getEntry(long bundleId, String path);
String getResource(long bundleId, String name);
String getDataFile(long bundleId, String filename);
```

The APIs are not intended for the client to actually use the content, but rather to inform the user *where* the contain is resolved to.

4.2.2 ServiceStateMBean

http://github.com/jbosgi/jbosgi-jmx/blob/master/api/src/main/java/org/jboss/osgi/jmx/ServiceStateMBeanExt.java

4.2.2.1 Bug 1602 - [JMX] ServiceStateMBean does not allow to introspect single service

Suggested API:

```
CompositeData getService(long serviceId)
```

4.2.2.2 Bug 1603 - [JMX] ServiceStateMBean does not allow to filter services

Suggested API:

```
TabularData getServices(String clazz, String filter) throws IOException;
```

4.2.3 FrameworkMBean

4.2.3.1 Bug 1606 - [JMX] FrameworkMBean.resolveBundles does not define behaviour for invalid bundle ids

4.3 Bug 1846 OSGi MBeans do not target a particular OSGi framework instance

The ObjectNames for the OSGi MBeans do not seem to consider the fact that there may be more than one running OSGi framework instances at a time in the same JVM and thus likely the same MBean server.

For example, the ObjectName for the FrameworkMBean is "osgi.core:type=framework,version=1.5" which has no indication of which OSGi framework instance it is target to.

4.4 Email from Jürgen Kissner / SAP

The OSGi Service Platform Enterprise Specification, Release 4, Version 4.2 specifies the BundleStateMBean interface (Section 124.9.2). The listBundles() method (124.9.2.79) is the only way I can find to get information about all bundles available in an OSGi runtime environment.

The problem that we have is that that method is rather expensive to use as it returns a collection of BUNDLE_TYPE structures - basically all header information for all bundles plus some state information. (See section 124.5.2 for the type definition).

We found that invoking that method takes several seconds on standard hardware with on OSGi runtime that contains around 100 bundles.

An important use case is to determine the status of certain bundles. Usually the bundle names are known. With the current interface we have to retrieve all information and then filter what we need. It would be preferable to be able to specify some kind of filter as in input argument or to restrict the output in some way (e.g. it would be sufficient to just return the Bundle-ID, name, version and state). Alternatively it would be helpful to retrieve a list of bundle-Ids, as that could be used to retrieve the additional information.

4.5 Bug 1592 UserAdminMBean - differences between spec and RI

there are differences between the spec description and RI for UserAdmin MBean;here is the list with problems I found:

•spec 124.13.1.14 defines that ROLE_TYPE contains NAME, TYPE and PROPERTIES; within the RI PROPERTIES field is missing

•spec 124.13.1.19 defines that USER_TYPE contains CREDENTIALS and NAME, TYPE, PROPERTIES coming from ROLE_TYPE; within the RI CREDENTIALS and PROPERTIES fields are missing

•spec 124.13.1.4 defines that GROUP_TYPE contains MEMBERS, REQUIRED_MEMBERS and CREDENTIALS, NAME, TYPE, PROPERTIES coming from USER_TYPE; within the RI CREDENTIALS and PROPERTIES fields are missing

•spec 124.13.1.1 defines that AUTORIZATION_TYPE contains NAME and TYPE; within the RI TYPE field is missing

Comment from Hal: Add property support to the next release of the spec.

4.6 Bug 1616 Add properties support

When I'm calling the method listServices of ServiceStateMBean, it returns object from TabularData type (SERVICES_TYPE). When I get row composite data type from it (SERVICE_TYPE), the attribute "Properties" is missing in it (such attribute is specified in the spec).

–A large amount of discussion including code snippets can be found in the bug

4.7 Bug 1645 Invalid RuntimeException handling in CT

This is a CT bug, it's mentioned here to ensure that it is dealt with along with the other JMX work.

4.8 Bug 1646 UserAdminMBean issues

```
> Two issues:
>
> 1) The description of UserAdminMBean does not define what should the GROUP_TYPE
> CompositeData contain if there are no member or required-members. That is, if
> the lookup on member of required-members should return null or an empty array.
> The CT assumes empty array but probably should handle both.
```

Hal: I believe it should only be an empty array. An empty array never throws a null pointer exception and always does the right thing.

```
> 2) UserAdminMBean.addMember() or .addRequiredMember() does not allow
> IllegalArgumentException to be raised when invalid group name is passed. This
> is inconsistent with other such methods in this mbean, for example,
> addCredential() or getGroup()
```

Hal: That's because, as other bugs have forced, there are no runtime exceptions to be thrown by the beans.

Patch available in the bug.

4.9 Bug 1647 BundleStateMBean.getRequiredBundles() clarification

Clarify the behaviour of BundleStateMBean.getRequiredBundles() in the spec.

4.10 Bug 1649 ConfigurationAdminMBean clarification

The specification does not say what should happen in the TabularType passed into the ConfigurationAdminMBean.update() and updateForLocation() functions is not of JmxConstants.PROPERTIES_TYPE.

Our implementation expects JmxConstants.PROPERTIES_TYPE type but the CT passes a slightly different type (i.e. the type name is "Properties" instead of "PROPERTIES").

Hal: Yes, this should be consistent.

4.11 Public Bug 87

```
Errata:
```

```
124.5.6    Boolean and boolean types are missed in property type syntax.
124.8.3.28 P_BOOLEAN type is missed in SCALAR list
124.9.4.6  NAME_ITEM type should be SimpleType.STRING
124.9.5.16 SERVICE_EVENT_TYPE misses BUNDLE_IDENTIFIER type.
BUNDLE_IDENTIFIER_ITEM is a part of SERVICE_EVENT_TYPE
124.9.5.25 IllegalArgumentException never throws in listServices() method

Unclear points:
- BundleEvents and ServiceEvents handling (124.9.2.2, 124.9.5.16). 124.2.3 can
contain information how to provide JMX notification (MBean should implement
javax.management.NotificationBroadcaster)
- There is no infomration how to register MBean for every instance of
Compendium Service (124.3.2, 124.3.4). What ObjectName key property should be
added to Compendium Service OBJECTNAME (",service.id=?")

Some small API improvements:
124.9.3.22, 124.9.3.27 - can throw IllegalArgumentException if the bundle
indicated does not exist
124.9.3.21, 124.9.3.28, 124.9.3.41 - can throw IllegalArgumentException if
locations/bundleIdentifiers and urls/newlevels arrays are null or have
different lenght
124.13.1.22 - can throw IllegalArgumentException if the groupname is not a
Group
124.13.1.32, 124.13.1.38, 124.13.1.40 - can throw IllegalArgumentException if
the filter is invalid
124.13.1.46 - can throw IllegalArgumentException if the groupname is not a
Group
124.13.1.50 - can throw IllegalArgumentException if the username is not a User
```

4.12 Public Bug 109

```
* 124.5.6 PROPERTY

> VALUE – (String) The value of the property. Values that contain white
> space (see Character.isWhitespace()), quote characters (both single
> quote ("'" \u0027) and double quote ('"' \u0022)), or
> backslashes ('\' \u005C), must be quoted. A string can be quoted with
> single or double quotes, any of the previously mentioned characters
> must be escaped with a backslash ('\' \u005C). Values must be trimmed
> of whitespace before usage.

A String containing a comma must also be quoted, at least with a vector or an
array (and a comma doesn't have to be escaped).

Is this a valid value, I really mean _value_ : "Vector of ", with an ending
space ?
Maybe not all -the values have to be trimmed of whitespaces before usage-
despite the above last phrase, only value received for a vector or an array.

No syntax explaining how to compose a VALUE in case of a vector or
an array, only an example.

I propose a comma separated list of all elements. Only one
comma allowed between elements, and whitespaces between elements must
be ignored :
```

```
1, 2, 3, 4
Sunday, Monday, Tuesday, Wednesday

Is this value allowed for String[]{"", "", ""} (3 times the empty String) ?
,,,
```

------- *Comment #1 From Pascal Perez 2010-11-25 17:28:25 UTC* [reply] -------

```
The last value is only a String with 2 commas.
```

5 Requirements

The requirements of this RFC are described in RFP 105.

6 Technical Solution

6.1 OSGi MBean Object Names

A Java process may launch more than one OSGi framework instance at a time; therefore it must be possible for each OSGi framework instance to register their own instances of the OSGi MBeans.

The distinction of OSGi MBeans across OSGi framework instances is accomplished through the addition of the key property *uuid* to the OSGi MBean Object Names defined originally by the JMX Management Model specification, version 1.0. The value of this property must be equal to the OSGi framework UUID of the launched framework instance.

For example, the following Object Name identifies the FrameworkMBean for the first launched instance of the Apache Felix implementation of the OSGi framework:

```
osgi.core:type=framework,version=1.5,uuid=f81d4fae-7dec-11d0-a765-00a0c91e6bf6,...
```

However, UUIDs are not user-friendly and thus hard to manage in JMX. Therefore, the key property *framework* is also added to the OSGi MBean Object Names. The *framework* property must be equal to the the Bundle-SymbolicName of the OSGi framework instance's System Bundle.

Following, we revisit the previous example to include the *framework* property:

```
osgi.core:type=framework,version=1.5,uuid=f81d4fae-7dec-11d0-a765-00a0c91e6bf6,
 framework=org.apache.felix.framework
```

Likewise, a possible Object Name for the BundleStateMBean is:

```
osgi.core:type=bundleState,version=1.5,uuid=f81d4fae-7dec-11d0-a765-00a0c91e6bf6
 framework=org.apache.felix.framework
```

The advantage of the *framework* property is that it can be used to simplify the querying for the MBeans using Object Name patterns (e.g. names with asterisks). For instance, the following query allows a client to find all FrameworkMBeans for a Felix implementation without having to rely on knowing the UUID:

```
ObjectName frameworkObjName =
 new ObjectName("osgi.core:type=framework,version=1.5,
  framework=org.apache.felix.framework,*");
mbeanServerConnection.queryMBeans(frameworkObjectName, null);
```

Furthermore, in many cases, a JMX client may appropriately assume that only a single instance of the OSGi framework exists in the managed system, as in the following example:

```
ObjectName frameworkObjName =
 new ObjectName("osgi.core:type=framework,version=1.5,*");
mbeanServerConnection.queryMBeans(frameworkObjectName, null);
```

The *uuid* and *framework* key properties are only applicable to OSGi JMX Package Version 2.0 and above. JMX clients using Version 1.0 should refrain from including these properties.

To maintain backward compatibility, a OSGi JMX Package Version 2.0 may register the first instantiation of an OSGi framework using both the Version 1.0 Object Names as well as the Object Names outlined in this specification. In other words, a JMX client may not specify the uuid and/or framework properties, and still retrieve the MBeans for a OSGi framework instance.

6.2 BundleStateMBean

The state of a bundle consists of several items, such as its exported packages, its imported packages, its headers, etc. Although each one of these items can be individually retrieved for a bundle, there is no single API to retrieve all of these collectively. Although this may not be a concern for local call invocations, it is typically a goal of JMX APIs to minimize the amount of information exchanged due to network constraints. Considering this, the following method is added to the BundleStateMBean class:

```
CompositeData getBundle(long bundleId) throws IOException;
```

The Composite Data returned is of type BUNDLE_TYPE.

It is often necessary to find the state of all bundles installed in an OSGi framework, however generally not all the state is needed, but rather a sub-set of it, which is application specific. For example, a management tool may be interested only on the symbolic name. The method listBundles() could be used to retrieve all the state, including the symbolic name, however listBundles() is expensive. Therefore, a variation of listBundles() is added to allow the selection of which items should be returned as part of the BUNDLE_TYPE composite data:

```
TabularData listBundles(String [] bundleTypeItems) throws IOException;
```

For example, to find out the symbolic name and the location of all installed bundles, one can issue the following operation:

```
listBundles(new String[]{BundleStateMBean.SYMBOLIC_NAME,
 BundleStateMBean.LOCATION});
```

Specification of an unsupported item causes an IllegalArgumentException to be thrown.

Even though the method listBundles() allows for the retrieval of all bundle identifiers, it is sometimes preferable to reference this information as a MBean attribute as attributes can provide JMX notifications when changed, therefore a BundleIds JMX attribute is added.

6.2.1 BundleStartLevel support

To complete BundleStartLevel support, the following operation is added:

```
boolean isActivationPolicyUsed( )
```

This operation maps to the equivalent method of the BundleStartLevel interface.

6.2.2 Headers

All the headers of a bundle can be retrieved with the method getHeaders(), but likewise there is no single method to retrieve a a single header. The following method is added to allow for this:

```
String getHeader(long bundleId, String key) throws IOException;
```

Bundle headers may be localized. In the absence of an argument indicating the locale to be used, such as in the case of the methods getHeaders(long bundleId) and getHeader(long bundleId, String key), the default locale is used. The following methods are added to the BundleStateMBean for dealing with locales:

```
TabularData getHeaders(long bundleId, String locale) throws IOException;
CompositeData getHeader(long bundleId, String key, String locale)
  throws IOException;
```

The interpretation of the locale argument is similar to that defined by the method Bundle.getHeaders(String locale).

6.2.3 Requiring Bundles

The methods getRequiredBundles(long bundleId) and getRequiringBundles(long bundleId) can be used to discover the relationship between bundles established by the use of the Require-Bundle manifest header.

For example, consider a bundle A with the following header:

```
Require-Bundle: B, C
```

Furthermore, bundle D defines the following header:

```
Require-Bundle: A
```

A call to getRequiredBundles() for bundle A would return the bundle identifiers for bundles B and C. Conversely, a call to getRequiringBundles() for bundle A would yield the bundle identifier for bundle D.

6.3 ServiceStateMBean

Tailing the pattern established for the BundleStateMBean, the following new methods are added to the ServiceStateMBean:

```
CompositeData getService(long serviceId) throws IOException;
```

The Composite Data returned is of type SERVICE_TYPE.

```
CompositeData getProperty(long serviceId, String key) throws IOException;
```

The Composite Data returned is of type PROPERTY_TYPE. Note that we could not simply return the value, as the value is encoded as a String and the item TYPE_ITEM is needed to understand how to decode it.

```
TabularData listServices(String clazz, String filter) throws IOException;
```

The Tabular Data returned is of type SERVICES_TYPE. The arguments clazz and filter have the semantic as of the method BundleContext.getAllServiceReferences().

Furthermore, the Tabular Data JmxConstants.PROPERTIES_TYPE is added to the SERVICE_TYPE composite data. By doing so, we now include all of the state of a service in the calls to listServices() and getService().

Finally, the following methods are used to allow the selection of which items are to be returned by the method listServices():

```
TabularData listServices(String clazz, String filter, String [] serviceTypeItems)
 throws IOException;
```

The Tabular Data returned is of type SERVICES_TYPE, however only contains the selected items relating to SERVICE_TYPE.

Likewise, for ease of use, notification support and performance, a ServiceIds JMX attribute is added:

```
long[] getServiceIds() throws IOException;
```

6.4 FrameworkMBean

The FrameworkMBean supports the batching of operations, such as installBundles(), and uninstallBundles(). These return a Composite Data of type BATCH_ACTION_RESULT_TYPE indicating the individual results of the batched operations.

The method resolveBundles() should likewise be changed to return a Composite Data of type BATCH_ACTION_RESULT_TYPE, however whereas the other batch operations need not return any additional information in case of success, in the case of the resolveBundles() a bundle may not resolve and still be considered a successful operation. In other words, a successful resolve operation results into a boolean true or false. An unsuccessful operation, such as in the case of an invalid bundle id, throws an exception and provides no result.

To accommodate this behavior, a new BATCH_RESOLVE_RESULT_TYPE Composite Data that includes an array of Boolean objects (i.e. SUCESS_ARRAY_TYPE) is defined for the return of the resolveBundles() method:

```
Item SUCCESS_ARRAY_ITEM = new Item(SUCCESS,
           "Whether the operation was successful",
           JmxConstants.BOOLEAN_ARRAY_TYPE);

CompositeType BATCH_RESOLVE_RESULT_TYPE = Item
           .compositeType(
                     "BUNDLE_RESOLVE_RESULT",
                     "This type encapsulates a bundle batch install action result",
                     BUNDLE_IN_ERROR_ID_ITEM,
                     COMPLETED_ITEM,
                     ERROR_ITEM,
                     REMAINING_ID_ITEM,
                     SUCCESS_ARRAY_ITEM);

CompositeData resolveBundles(long[] bundleIdentifiers) throws IOException;
```

Furthermore, a “refreshBundles() : CompositeData” method is added. This method is synchronous, and shall likewise make use of the BATCH_ACTION_RESULT_TYPE to return its results.

```
boolean refreshBundle(long bundleIdentifier) throws IOException;
CompositeData refreshBundles(long[] bundleIdentifiers) throws IOException;
```

The existing “refreshBundles(): void“ method is kept asynchronous, however it will post a JMX notification with its results mimicking the same format as of the synchronous refreshBundles() method.

The following operations are added to the FrameworkMBean to support the FrameworkWiring class:

```
long [] getDependencyClosure(long [] bundles) throws IOException
```

`long [] getRemovalPendingBundles() throws IOException`

Both methods return an array of bundle Ids, or an empty array.

6.4.1 Framework Properties support

To support obtaining framework properties the following method will be added to the FrameworkMBean:

`TabularData getProperties() throws IOException;`
Where the TabularData is structured as a `PROPERTIES_TYPE`.

6.5 BundleRevisionsMBean

A org.osi.framework.jmx.BundleRevisionsMBean is added to support the bundle wiring package.

It is named as follows:

`osgi.core:type=wiringState,version=1.0`

Although BundleRevisionsMBean provides an alternative approach to retrieving the same state as supported by the PackageAdminMBean, the latter is kept as it provides a more user-friendly API for dealing exclusively with packages.

6.5.1 API

```
/*
 * Copyright (c) OSGi Alliance (2009, 2010). All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.osgi.jmx.framework;

import javax.management.openmbean.ArrayType;
import javax.management.openmbean.CompositeData;
import javax.management.openmbean.CompositeType;
import javax.management.openmbean.SimpleType;
import javax.management.openmbean.TabularType;

import org.osgi.jmx.Item;
```

```
import org.osgi.jmx.JmxConstants;

/**
 * This MBean represents the bundle wiring state.
 *
 * Note that not all information from the BundleWiring Java API is provided.
 *
 * Particularly, the limitations are:
 *   - Cannot retrieve references to resources (e.g. class) of a particular
bundle wiring.
 *
 */
public interface BundleWiringMBean {
    /*
     * The Object Name for a Bundle State MBean.
     */

    String OBJECTNAME = JmxConstants.OSGI_CORE
        + ":type=wiringState,version=1.0";

    /*
     * Namespaces
     *
     */

    String BUNDLE_NAMESPACE = "osgi.wiring.bundle";

    String HOST_NAMESPACE = "osgi.wiring.host";

    String PACKAGE_NAMESPACE = "osgi.wiring.package";

    /*
     * Items, CompositeData, TabularData, ArrayTypes
     *
     */

    String KEY = "Key";

    Item KEY_ITEM = new Item(KEY, "The directive key", SimpleType.STRING);

    String VALUE = "Value";

    Item VALUE_ITEM = new Item(VALUE, "The directive value",
```

```
        SimpleType.STRING);

    CompositeType DIRECTIVE_TYPE = Item.compositeType("DIRECTIVE",
            "Describes a directive of a capability or requirement",
            KEY_ITEM, VALUE_ITEM);

    TabularType DIRECTIVES_TYPE = Item.tabularType("DIRECTIVES",
            "Describes the directives of a capability or requirement",
            DIRECTIVE_TYPE, KEY
            );

    String DIRECTIVES = "Directives";

    Item DIRECTIVES_ITEM = new Item(DIRECTIVES,
            "The directives of a capability or requirement",
            DIRECTIVES_TYPE);

    // REVIEW should we reuse from JmxConstants here or create our own?
    TabularType ATTRIBUTES_TYPE = Item.tabularType("ATTRIBUTES",
            "Describes attributes of a capability or requirement",
            JmxConstants.PROPERTY_TYPE, JmxConstants.KEY
            );

    String ATTRIBUTES = "Attributes";

    Item ATTRIBUTES_ITEM = new Item(ATTRIBUTES,
            "The attributes of a capability or requirement",
            ATTRIBUTES_TYPE);

    String NAMESPACE = "Namespace";

    Item NAMESPACE_ITEM = new Item(NAMESPACE,
            "The namespace of a capability or requirement",
            SimpleType.STRING);

    CompositeType BUNDLE_REQUIREMENT_TYPE =
        Item.compositeType("BUNDLE_REQUIREMENT",
                "Describes the live wired requirements of a bundle",
                ATTRIBUTES_ITEM, DIRECTIVES_ITEM, NAMESPACE_ITEM);

    CompositeType BUNDLE_CAPABILITY_TYPE =
        Item.compositeType("BUNDLE_CAPABILITY",
                "Describes the live wired capabilities of a bundle",
```

```
                ATTRIBUTES_ITEM, DIRECTIVES_ITEM, NAMESPACE_ITEM);

    String PROVIDER_BUNDLE_ID = "ProviderBundleId";

    Item PROVIDER_BUNDLE_ID_ITEM = new Item(PROVIDER_BUNDLE_ID,
                "The identifier of the bundle that is the provider of the
capability",
            SimpleType.LONG);

    String REQUIRER_BUNDLE_ID = "RequirerBundleId";

    Item REQUIRER_BUNDLE_ID_ITEM = new Item(REQUIRER_BUNDLE_ID,
                "The identifier of the bundle that is the requirer of the
requirement",
            SimpleType.LONG);

    String BUNDLE_REQUIREMENT = "BundleRequirement";

    Item BUNDLE_REQUIREMENT_ITEM = new Item(BUNDLE_REQUIREMENT,
            "The wired requirements of a bundle",
            BUNDLE_REQUIREMENT_TYPE);

    String BUNDLE_CAPABILITY = "BundleCapability";

    Item BUNDLE_CAPABILITY_ITEM = new Item(BUNDLE_CAPABILITY,
            "The wired capabilities of a bundle",
            BUNDLE_CAPABILITY_TYPE);

    String PROVIDER_BUNDLE_REVISION_ID = "ProviderBundleRevisionId";

                    Item     PROVIDER_BUNDLE_REVISION_ID_ITEM     =     new
Item(PROVIDER_BUNDLE_REVISION_ID,
             "A local id for the bundle revision that is the provider of the
capability",
            SimpleType.STRING);

    String REQUIRER_BUNDLE_REVISION_ID = "RequirerBundleRevisionId";

                  Item    REQUIRER_BUNDLE_REVISION_ID_ITEM    =         new
Item(REQUIRER_BUNDLE_REVISION_ID,
             "A local id for the bundle revision that is the requirer of the
requirement",
            SimpleType.STRING);
```

```
    /**
     * Describes the live association between a provider of
     *  a capability and a requirer of the corresponding requirement.
     */
    CompositeType BUNDLE_WIRE_TYPE =
        Item.compositeType("BUNDLE_WIRE",
                    "Describes the live association between a provider and a
requirer",
                BUNDLE_REQUIREMENT_ITEM,
                BUNDLE_CAPABILITY_ITEM,
                PROVIDER_BUNDLE_ID_ITEM,
                PROVIDER_BUNDLE_REVISION_ID_ITEM,
                REQUIRER_BUNDLE_ID_ITEM,
                REQUIRER_BUNDLE_REVISION_ID_ITEM
                );

    ArrayType<?> BUNDLE_WIRES_TYPE_ARRAY =
        Item.arrayType(1, BUNDLE_WIRE_TYPE);

    String BUNDLE_REVISION_ID = "BundleRevisionId";

    Item BUNDLE_REVISION_ID_ITEM = new Item(BUNDLE_REVISION_ID,
            "The local identifier of the bundle revision",
            SimpleType.STRING);

    String BUNDLE_WIRES_TYPE = "BundleWiresType";

    Item BUNDLE_WIRES_TYPE_ARRAY_ITEM = new Item(BUNDLE_WIRES_TYPE,
            "The bundle wires of a bundle revision",
            BUNDLE_WIRES_TYPE_ARRAY);

    String BUNDLE_ID = "BundleId";

    Item BUNDLE_ID_ITEM = new Item(BUNDLE_ID,
            "The bundle identifier of the bundle revision",
            SimpleType.STRING);

    ArrayType<?> REQUIREMENT_TYPE_ARRAY =
        Item.arrayType(1, BUNDLE_REQUIREMENT_TYPE);

    ArrayType<?> CAPABILITY_TYPE_ARRAY =
        Item.arrayType(1, BUNDLE_CAPABILITY_TYPE);
```

```
    String REQUIREMENTS = "Requirements";

    Item REQUIREMENTS_ITEM = new Item(REQUIREMENTS,
            "The bundle requirements of a bundle revision wiring",
            REQUIREMENT_TYPE_ARRAY);

    String CAPABILITIES = "Capabilities";

    Item CAPABILITIES_ITEM = new Item(CAPABILITIES,
            "The bundle capabilities of a bundle revision wiring",
            CAPABILITY_TYPE_ARRAY);

    CompositeType BUNDLE_WIRING_TYPE =
        Item.compositeType("BUNDLE_WIRING",
                 "Describes the runtime association between a provider and a
requirer",
                BUNDLE_ID_ITEM,               /* Long */
                BUNDLE_REVISION_ID_ITEM,      /* Long (local scope) */
                REQUIREMENTS_ITEM,            /* REQUIREMENT_TYPE [] */
                CAPABILITIES_ITEM,            /* CAPABILITIES_TYPE [] */
                BUNDLE_WIRES_TYPE_ARRAY_ITEM  /* BUNLDE_WIRE_TYPE [] */
                );

    ArrayType<?> BUNDLE_WIRING_TYPE_ARRAY =
        Item.arrayType(1, BUNDLE_WIRING_TYPE);

    ArrayType<?> REVISIONS_REQUIREMENT_TYPE_ARRAY =
        Item.arrayType(2, BUNDLE_REQUIREMENT_TYPE);

    ArrayType<?> REVISIONS_CAPABILITY_TYPE_ARRAY =
        Item.arrayType(2, BUNDLE_CAPABILITY_TYPE);


    /**
     * Returns the requirements for the current bundle revision.
     * The ArrayType is typed by the {@link #REQUIREMENT_TYPE_ARRAY}.
     *
     * @param bundleId
     * @param namespace
        * @return the declared requirements for the current revision of
<code>bundleId</code>
     * and <code>namespace</code>
```

```
     *
     */
    ArrayType<?> getCurrentRevisionDeclaredRequirements(long bundleId,
            String namespace);

    /**
     * Returns the capabilities for the current bundle revision.
     * The ArrayType is typed by the {@link #CAPABILITY_TYPE_ARRAY}
     *
     * @param bundleId
     * @param namespace
        * @return the declared capabilities for the current revision of
<code>bundleId</code>
     * and <code>namespace</code>
     */
    ArrayType<?> getCurrentRevisionDeclaredCapabilities(long bundleId,
            String namespace);

    /**
     * Returns the bundle wiring for the current bundle revision.
     * The ArrayType is typed by the {@link #BUNDLE_WIRING_TYPE}
     *
     * @param bundleId
     * @param namespace
     * @return the wires for the current revision of <code>bundleId</code>
     * and <code>namespace</code>
     */
    CompositeData getCurrentWiring(long bundleId, String namespace);

    /**
     * Returns the requirements for all revisions of the bundle.
                  *    The    ArrayType    is    typed    by    the    {@link
#REVISIONS_REQUIREMENT_TYPE_ARRAY}.
     * The requirements are in no particular order, and may change in
     *  subsequent calls to this operation.
     *
     * @param bundleId
     * @param namespace
     * @param inUse
          *  @return  the  declared  requirements  for  all  revisions  of
<code>bundleId</code>
     *
     */
```

```
    ArrayType<?> getRevisionsDeclaredRequirements(long bundleId,
            String namespace, boolean inUse);

    /**
     * Returns the capabilities for all revisions of the bundle.
                  *    The    ArrayType    is    typed    by    the    {@link
#REVISIONS_CAPABILITY_TYPE_ARRAY}
     * The capabilities are in no particular order, and may change in
     *  subsequent calls to this operation.
     *
     * @param bundleId
     * @param namespace
     * @param inUse
          *  @return  the  declared  capabilities  for  all  revisions  of
<code>bundleId</code>
     */
    ArrayType<?> getRevisionsDeclaredCapabilities(long bundleId,
            String namespace, boolean inUse);

    /**
     * Returns the bundle wirings for all revisions of the bundle.
     * The ArrayType is typed by the {@link #BUNDLE_WIRING_TYPE_ARRAY}
     * The bundle wirings are in no particular order, and may
     *  change in subsequent calls to this operations.
     *
     * @param bundleId
     * @param namespace
     * @return the wires for all revisions of <code>bundleId</code>
     */
    ArrayType<?> getRevisionsWiring(long bundleId, String namespace);

    /**
     * Returns a closure of all bundle wirings linked by their
     *  bundle wires, starting at <code>rootBundleId</code>.
     * The ArrayType is typed by the {@link #BUNDLE_WIRING_TYPE_ARRAY}
     * The bundle wirings are in no particular order, and may
     *  change in subsequent calls to this operation. Furthermore,
        * the  bundle  wiring  IDs  are  local  and  cannot  be  reused  across
invocations.
     *
     * @param rootBundleId
     * @param namespace
     * @return a closure of bundle wirings linked together by wires.
```

```
     */
    ArrayType<?> getWiringClosure(long rootBundleId, String namespace);

    /**
     * Returns true if capability provided by <code>provider</code> matches
     *  with the requirement being required by <code>requirer</code>.
     * The <code>provider</code>'s CompositeType is typed by the
     *  {@link #BUNDLE_CAPABILITY_TYPE}
     * The <code>requirer</code>'s CompositeType is typed by the
     *  {@link #BUNDLE_REQUIREMENT_TYPE}
     *
      * REVIEW This method would have worked better should the requirements
and
     *  capabilities have an ID
     *
     * @param requirer bundle id of the bundle requirer
     * @param provider bundle id of the bundle provider
     * @return true if capability matches with requirement.
     */
    boolean matches(CompositeType provider, CompositeType requirer);

}
```

7 Command Line API

If this specification would benefit from a command line interface, describe it here. Commands should be realized as described in RFC 147.

This section is optional and could also be provided in a separate RFC.

8 JMX API

RFC 139 describes the JMX API to the OSGi Framework and a number of standard OSGi Services. The JMX specification is also present as chapter 124 in the OSGi spec documents.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a JMX API? The expectation is that in most cases it would.

The JMX API for the design in this RFC should be described here and if there is no JMX API an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

This RFC is specifically about the JMX API.

9 Initial Spec Chapter

Provide a link to where the Initial Spec Chapter can be found. The Initial Spec Chapter is typically written by someone other than the author(s) of this RFC and represents a rewrite of this document as close as possible to what will ultimately appear in the OSGi Specifications. It will be used by the Specification Editor as the basis for the ultimate specification chapter.

The spec template and writing guidelines can be found here:

https://www.osgi.org/members/svn/documents/trunk/templates/specification-template-oo.ott

https://www.osgi.org/members/svn/documents/trunk/templates/specwriting.pdf

10 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

11 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

12 Document Support

12.1 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

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

12.2 Author's Address

Name	David Bosschaert
Company	Red Hat
Address	6700 Cork Airport Business Park Kinsale Road Cork Ireland
Voice	+353 21 230 3400
e-mail	david@redhat.com

Name	Alexandre Alves
Company	Oracle Corporation
Address	
Voice	+1 650-607-0878
e-mail	alex.alves@oracle.com

12.3 Acronyms and Abbreviations

12.4 End of Document

RFC 0172 Declarative Services Annotations

Final

24 Pages

Abstract

Define tooling annotations for Declarative Services.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 8.1.

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

0.3 Revision History

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

Revision	Date	Comments
Initial	28 Mar 2011	Initial draft based upon bnd annotations. BJ Hargrave, IBM
2nd draft	29 April 2011	Updated requirements from approved RFP 145. Renamed enum element ATLEASTONE to AT_LEAST_ONE. BJ Hargrave, IBM
3rd draft	3 August 2011	Removed folding of method name to lower case for reference name. BJ Hargrave, IBM
Final draft	5 August 2011	After CPEG call discussion, moving RFC to final for voting. BJ Hargrave, IBM
Final draft	9 September 2011	Removed old reference to deleted git branch. BJ Hargrave, IBM

1 Introduction

Declarative Services is a popular way of programming to OSGi in a POJO manner. The DS specification requires the programmer to supply a component description in XML for each Service Component authored in the bundle. This can be awkward as the information in the XML may be duplicate of information in the source code. There is opportunity for this information to be entered incorrectly or to get out of sync with the source when refactoring. Annotations are needed to further simplify programming with DS.

2 Application Domain

Declarative Services (chapter 121 in the OSGi specifications) defines a POJO programming model for OSGi services. This model requires Service Component class be implemented in a certain way and the XML component descriptions be authored.

SCR will process the XML component descriptions are runtime and process the components. Using the XML allows lazy loading of component implementation classes. SCR only needs to load the classes when necessary to properly process the components.

3 Problem Description

The XML component descriptions are separate from the code. This means the XML must contain the class names and method names for the component. So this information has to be repeated which opens things for errors due to typos and changes from code refactoring.

4 Requirements

1. Define annotations so that, for the majority of cases, the programmer can avoid writing an XML component description.
2. The annotations must not require any changes to the DS specification.
3. The annotations are to be processed at tool time rather than runtime to preserve the laziness of DS. (Processing the annotations at runtime would require loading all classes in the bundle to look for annotations or require some support for byte code processing the classes.)
4. Tools which understand the annotations must process them and generate the proper XML component descriptions which are added to the bundle along with the proper changes to the Service-Component manifest header.
5. The solution must not require DS users to use Java 5 language if they don't choose to use annotations.

6. The solution must work with compliant DS 1.1 implementations or later.
7. Tools may generate DS 1.0 name space component descriptions if the annotations do not require any DS 1.1 features.
8. Tools should be able to override the name space of the XML.
9. This RFC should communicate with stakeholders of other interested OSGi DI engines to try to align overlapping concepts.

5 Technical Solution

This document defines a new org.osgi.service.component.annotations package. This package defines the annotations and associated enums.

There is a class annotation, `@Component`, which marks up a component implementation class and specifies the information about the component. There are also several annotations for the defined methods of a component: `@Activate`, `@Modified`, `@Deactivate`, `@Reference`. `@Activate` marks the component's `activate` method. `@Modified` marks the component's `modified` method. `@Deactivate` marks the component's `deactivate` method. `@Reference` marks a `bind` method in the component and specifies information about the reference to be injected. It also specifies information about the matching `unbind` method.

Following is the javadoc which provides more detail on the annotations.

OSGi Javadoc

8/3/11 3:22 PM

Package Summary		Page
org.osgi.service.component.annotations	Service Component Annotations Package Version 1.0.	7

Package org.osgi.service.component.annotations

Service Component Annotations Package Version 1.0.

See:
Description

Enum Summary		Page
ConfigurationPolicy	Configuration Policy for the `Component` annotation.	13
ReferenceCardinality	Cardinality for the `Reference` annotation.	20
ReferencePolicy	Policy for the `Reference` annotation.	22

Annotation Types Summary		Page
Activate	Identify the annotated method as the `activate` method of a Service Component.	8
Component	Identify the annotated class as a Service Component.	9
Deactivate	Identify the annotated method as the `deactivate` method of a Service Component.	15
Modified	Identify the annotated method as the `modified` method of a Service Component.	16
Reference	Identify the annotated method as a `bind` method of a Service Component.	17

Package org.osgi.service.component.annotations Description

Service Component Annotations Package Version 1.0.

This package is not used at runtime. Annotated classes are processed by tools to generate Component Descriptions which are used at runtime.

Annotation Type Activate

org.osgi.service.component.annotations

```
@Retention(value=RetentionPolicy.CLASS)
@Target(value=ElementType.METHOD)
public @interface Activate
```

Identify the annotated method as the `activate` method of a Service Component.

The annotated method is the activate method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

Version:
$Id: b61a8aecb6d8df3d60a2e6b05e6021801b580331 $
See Also:
"The activate attribute of the component element of a Component Description."

Annotation Type Component

org.osgi.service.component.annotations

```
@Retention(value=RetentionPolicy.CLASS)
@Target(value=ElementType.TYPE)
public @interface Component
```

Identify the annotated class as a Service Component.

The annotated class is the implementation class of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

Version:
$Id: 6d6332dbb471a2bc582bf8fc9a9b7f3efd02bfc2 $
See Also:
"The component element of a Component Description."

Required Element Summary		Page
ConfigurationPolicy	**configurationPolicy** The configuration policy of this Component.	11
boolean	**enabled** Declares whether this Component is enabled when the bundle containing it is started.	10
String	**factory** The factory identifier of this Component.	10
boolean	**immediate** Declares whether this Component must be immediately activated upon becoming satisfied or whether activation should be delayed.	11
String	**name** The name of this Component.	9
String[]	**properties** Property entries for this Component.	11
String[]	**property** Properties for this Component.	11
Class<?>[]	**service** The types under which to register this Component as a service.	10
boolean	**servicefactory** Declares whether this Component uses the OSGi ServiceFactory concept and each bundle using this Component's service will receive a different component instance.	10

Element Detail

name

```
public abstract String name
```

The name of this Component.

If not specified, the name of this Component is the fully qualified type name of the class being annotated.

Default:
""
See Also:
"The name attribute of the component element of a Component Description."

service

```
public abstract Class<?>[] service
```

The types under which to register this Component as a service.

If no service should be registered, the empty value `{}` must be specified.

If not specified, the service types for this Component are all the *directly* implemented interfaces of the class being annotated.

Default:
{}
See Also:
"The service element of a Component Description."

factory

```
public abstract String factory
```

The factory identifier of this Component. Specifying a factory identifier makes this Component a Factory Component.

If not specified, the default is that this Component is not a Factory Component.

Default:
""
See Also:
"The factory attribute of the component element of a Component Description."

servicefactory

```
public abstract boolean servicefactory
```

Declares whether this Component uses the OSGi ServiceFactory concept and each bundle using this Component's service will receive a different component instance.

If `true`, this Component uses the OSGi ServiceFactory concept. If `false` or not specified, this Component does not use the OSGi ServiceFactory concept.

Default:
false
See Also:
"The servicefactory attribute of the service element of a Component Description."

enabled

```
public abstract boolean enabled
```

Declares whether this Component is enabled when the bundle containing it is started.

If `true`, this Component is enabled. If `false` or not specified, this Component is disabled.

Default:
true
See Also:
"The enabled attribute of the component element of a Component Description."

immediate

```
public abstract boolean immediate
```

Declares whether this Component must be immediately activated upon becoming satisfied or whether activation should be delayed.

If `true`, this Component must be immediately activated upon becoming satisfied. If `false`, activation of this Component is delayed. If this property is specified, its value must be `false` if the `factory()` property is also specified or must be `true` if the `service()` property is specified with an empty value.

If not specified, the default is `false` if the `factory()` property is specified or the `service()` property is not specified or specified with a non-empty value and `true` otherwise.

Default:
false
See Also:
"The immediate attribute of the component element of a Component Description."

configurationPolicy

```
public abstract ConfigurationPolicy configurationPolicy
```

The configuration policy of this Component.

Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID equals the name of the component.

If not specified, the `OPTIONAL` configuration policy is used.

Default:
ConfigurationPolicy.OPTIONAL
See Also:
"The configuration-policy attribute of the component element of a Component Description."

property

```
public abstract String[] property
```

Properties for this Component.

Each property string is specified as `"key=value"`. The type of the property value can be specified in the key as `key:type=value`. The type must be one of the property types supported by the type attribute of the property element of a Component Description.

To specify a property with multiple values, use multiple key, value pairs. For example, `"foo=bar", "foo=baz"`.

Default:
{}
See Also:
"The property element of a Component Description."

properties

```
public abstract String[] properties
```

Property entries for this Component.

Specifies the name of an entry in the bundle whose contents conform to a standard Java Properties File. The entry is read and processed to obtain the properties and their values.

Default:
{}
See Also:
"The properties element of a Component Description."

Enum ConfigurationPolicy

org.osgi.service.component.annotations

```
java.lang.Object
  └ java.lang.Enum<ConfigurationPolicy>
      └ org.osgi.service.component.annotations.ConfigurationPolicy
```

All Implemented Interfaces:
Comparable<ConfigurationPolicy>, Serializable

```
public enum ConfigurationPolicy
extends Enum<ConfigurationPolicy>
```

Configuration Policy for the Component annotation.

Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID is the name of the component.

Version:
$Id: 333d73c609620e0474ec143b6a9b9cba7ce271e7 $

Enum Constant Summary	Page
IGNORE Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.	14
OPTIONAL Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present.	13
REQUIRE There must be a corresponding Configuration object for the component configuration to become satisfied.	13

Method Summary		Page
static ConfigurationPolicy	**valueOf**(String name)	14
static ConfigurationPolicy[]	**values**()	14

Enum Constant Detail

OPTIONAL

```
public static final ConfigurationPolicy OPTIONAL
```

Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present.

REQUIRE

```
public static final ConfigurationPolicy REQUIRE
```

There must be a corresponding Configuration object for the component configuration to become satisfied.

IGNORE

```
public static final ConfigurationPolicy IGNORE
```

Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.

Method Detail

values

```
public static ConfigurationPolicy[] values()
```

---

valueOf

```
public static ConfigurationPolicy valueOf(String name)
```

Annotation Type Deactivate

org.osgi.service.component.annotations

```
@Retention(value=RetentionPolicy.CLASS)
@Target(value=ElementType.METHOD)
public @interface Deactivate
```

Identify the annotated method as the `deactivate` method of a Service Component.

The annotated method is the deactivate method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

Version:
$Id: 397f31967ac41b758dd65190962f620bf9a86bdc $
See Also:
"The deactivate attribute of the component element of a Component Description."

Annotation Type Modified

org.osgi.service.component.annotations

```
@Retention(value=RetentionPolicy.CLASS)
@Target(value=ElementType.METHOD)
public @interface Modified
```

Identify the annotated method as the `modified` method of a Service Component.

The annotated method is the modified method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

Version:
$Id: 2a93eb9a641883494d39ddb6ecdf42779d787f7a $
See Also:
"The modified attribute of the component element of a Component Description."

Annotation Type Reference

org.osgi.service.component.annotations

```
@Retention(value=RetentionPolicy.CLASS)
@Target(value=ElementType.METHOD)
public @interface Reference
```

Identify the annotated method as a `bind` method of a Service Component.

The annotated method is a bind method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

Version:
$Id: 7b1b6b10fc8b315d3c4fb5866a89e41c326bf639 $
See Also:
"The reference element of a Component Description."

Required Element Summary		Page
ReferenceCardinality	**cardinality** The cardinality of the reference.	18
String	**name** The name of this reference.	17
ReferencePolicy	**policy** The policy for the reference.	18
Class<?>	**service** The type of the service to bind to this reference.	17
String	**target** The target filter for the reference.	18
String	**unbind** The name of the unbind method which pairs with the annotated bind method.	18

Element Detail

name

```
public abstract String name
```

The name of this reference.

If not specified, the name of this reference is based upon the name of the method being annotated. If the method name begins with `set` or `add`, that is removed.

Default:
""
See Also:
"The name attribute of the reference element of a Component Description."

---

service

```
public abstract Class<?> service
```

The type of the service to bind to this reference.

If not specified, the type of the service to bind is based upon the type of the first argument of the method being annotated.

Default:
Object.class
See Also:
"The interface attribute of the reference element of a Component Description."

---

cardinality

```
public abstract ReferenceCardinality cardinality
```

The cardinality of the reference.

If not specified, the reference has a `1..1` cardinality.

Default:
ReferenceCardinality.MANDATORY
See Also:
"The cardinality attribute of the reference element of a Component Description."

---

policy

```
public abstract ReferencePolicy policy
```

The policy for the reference.

If not specified, the `STATIC` reference policy is used.

Default:
ReferencePolicy.STATIC
See Also:
"The policy attribute of the reference element of a Component Description."

---

target

```
public abstract String target
```

The target filter for the reference.

Default:
""
See Also:
"The target attribute of the reference element of a Component Description."

---

unbind

```
public abstract String unbind
```

The name of the unbind method which pairs with the annotated bind method.

To declare no unbind method, the value `"-"` must be used.

If not specified, the name of the unbind method is derived from the name of the annotated bind method. If the annotated method name begins with `set`, that is replaced with `unset` to derive the unbind method name. If the annotated method name begins with `add`, that is replaced with `remove` to derive the unbind method name. Otherwise, `un` is prefixed to the annotated method name to derive the unbind method name.

Default:
""

See Also:
"The unbind attribute of the reference element of a Component Description."

Enum ReferenceCardinality

org.osgi.service.component.annotations

```
java.lang.Object
  └ java.lang.Enum<ReferenceCardinality>
      └ org.osgi.service.component.annotations.ReferenceCardinality
```

All Implemented Interfaces:
Comparable<ReferenceCardinality>, Serializable

```
public enum ReferenceCardinality
extends Enum<ReferenceCardinality>
```

Cardinality for the `Reference` annotation.

Specifies if the reference is optional and if the component implementation support a single bound service or multiple bound services.

Version:
$Id: a86f2f25880cbd49111f9d8ab8279cbd5bc6e7c1 $

Enum Constant Summary	Page
AT_LEAST_ONE The reference is mandatory and multiple.	21
MANDATORY The reference is mandatory and unary.	20
MULTIPLE The reference is optional and multiple.	21
OPTIONAL The reference is optional and unary.	20

Method Summary		Page
static ReferenceCardinality	**valueOf**(String name)	21
static ReferenceCardinality[]	**values**()	21

Enum Constant Detail

OPTIONAL

```
public static final ReferenceCardinality OPTIONAL
```

The reference is optional and unary. That is, the reference has a cardinality of `0..1`.

MANDATORY

```
public static final ReferenceCardinality MANDATORY
```

The reference is mandatory and unary. That is, the reference has a cardinality of `1..1`.

MULTIPLE

```
public static final ReferenceCardinality MULTIPLE
```

The reference is optional and multiple. That is, the reference has a cardinality of `0..n`.

AT_LEAST_ONE

```
public static final ReferenceCardinality AT_LEAST_ONE
```

The reference is mandatory and multiple. That is, the reference has a cardinality of `1..n`.

Method Detail

values

```
public static ReferenceCardinality[] values()
```

valueOf

```
public static ReferenceCardinality valueOf(String name)
```

Enum ReferencePolicy

org.osgi.service.component.annotations

```
java.lang.Object
  └ java.lang.Enum<ReferencePolicy>
      └ org.osgi.service.component.annotations.ReferencePolicy
```

All Implemented Interfaces:
Comparable<ReferencePolicy>, Serializable

```
public enum ReferencePolicy
extends Enum<ReferencePolicy>
```

Policy for the Reference annotation.

Version:
$Id: 698f862af7645e7a5baf98070cfbf71223041241 $

Enum Constant Summary	Page
DYNAMIC The dynamic policy is slightly more complex since the component implementation must properly handle changes in the set of bound services.	22
STATIC The static policy is the most simple policy and is the default policy.	22

Method Summary		Page
static ReferencePolicy	**valueOf**(String name)	23
static ReferencePolicy[]	**values**()	23

Enum Constant Detail

STATIC

```
public static final ReferencePolicy STATIC
```

The static policy is the most simple policy and is the default policy. A component instance never sees any of the dynamics. Component configurations are deactivated before any bound service for a reference having a static policy becomes unavailable. If a target service is available to replace the bound service which became unavailable, the component configuration must be reactivated and bound to the replacement service.

DYNAMIC

```
public static final ReferencePolicy DYNAMIC
```

The dynamic policy is slightly more complex since the component implementation must properly handle changes in the set of bound services. With the dynamic policy, SCR can change the set of bound services without deactivating a component configuration. If the component uses the event strategy to access services, then the component instance will be notified of changes in the set of bound services by calls to the bind and unbind methods.

Method Detail

values

```
public static ReferencePolicy[] values()
```

---

valueOf

```
public static ReferencePolicy valueOf(String name)
```

---

6 Considered Alternatives

None.

7 Security Considerations

The annotations in this document are not processed at runtime and introduce no security concerns.

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

Name	BJ Hargrave
Company	IBM
e-mail	hargrave@us.ibm.com

8.3 Acronyms and Abbreviations

DS – Declarative Services

SCR – Service Component Runtime, the runtime for DS.

8.4 End of Document

RFC 174 Bundle Collision Hook

Final

12 Pages

Abstract

Define a bundle collision hook.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 11.1.

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

0.3 Revision History

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

Revision	Date	Comments
Initial	June 10 2011	Initial draft for bundle CollisionHook. Thomas Watson, IBM tjwatson@us.ibm.com
	June 23 2011	Document the “managed” configuration value for org.osgi.framework.bsnversion from https://www.osgi.org/members/bugzilla/show_bug.cgi?id=2020
	August 18 2011	Final Version

1 Introduction

This RFC details how a bundle can hook into the install and update operations to influence the decision to allow multiple installations of a bundle with the same symbolic name and version.

2 Application Domain

This design is targeted at bundles which need to observe and manipulate the install and update operations to allow or disallow multiple installations of a bundle with the same symbolic name and version. In general this will

be a highly specialized bundle writtn by systems programmers. This design is not intended to be used by so-called “normal” application bundles.

3 Problem Description

With the OSGi R4.3 Core specification the resolver and bundle event/find hooks were introduced to complement the existing service hooks which were introduced in the R4.2 Core specification. With the combination of the service, bundle and resolver hooks an additional layer on top of the framework can be implemented to provide such things as bundle grouping or scoping for isolation (aka RFC 152 – Subsystems).

The OSGi R4.3 Core specification also introduced a new framework configuration option called org.osgi.framework.bsnversion. The default value of this configuration option is “single” which specifies the framework will only allow a single bundle to be installed for a given symbolic name and version. The value of “multiple” completely disables this invariant and allows multiple bundles to be installed having the same symbolic name and version. This was a requirement for isolation layers (e.g. subsystems) to allow the same bundle to be installed multiple times into multiple isolation units (or scopes) within the same OSGi framework instance.

The problem with the current definition of org.osgi.framework.bsnversion is that it provides a global on/off switch for the policy to allow multiple bundles to be installed having the same symbolic name and version. Hooks are not allowed to influence the policy at all. An additional hook is needed to influence this policy.

4 Requirements

No RFP exists for this RFC. This RFC is a result of discussions from bug:

https://www.osgi.org/members/bugzilla/show_bug.cgi?id=2020

Significant issues were discovered in prototyping the idea in bug 2020. This resulted in the need for a more complicated solution and the need for a formal RFC to discuss the solution.

- The solution MUST allow certain bundles to decide if the installation of multiple bundles having the same symbolic name and version is allowed.
- The solution MUST allow certain bundles to decide if updating an existing bundle such that the updated bundle will have the same symbolic name and version as one or more other installed bundles is allowed.

- The solution MUST not change the current behavior of the framework when the value of 'single' or 'multiple' is used for the configuration value of osgi.framework.bsnversion. This implies a new configuration value is needed.

5 Technical Solution

This RFC introduces a new hook, called the bundle collision hook, which is used by the framework to determine if installations of multiple bundles having the same symbolic name and version is allowed.

5.1 Terminology

- Bundle Collision – When multiple bundles installed into a single framework have the same symbolic name and version.
- Target – The bundle performing the install or update operation. For install operations the target is the bundle associated with the BundleContext used to call one of the install methods. For update operations the target is the bundle used to call one of the update methods.
- Handler – The bundle that registers a bundle collision hook service to influence the policy to allow or disallow bundle collisions.
- Bundle Collision Hook - A bundle collision hook intercepts bundle install and update operations which will result in a bundle collision. A bundle collision hook may remove specific bundle collisions from influencing the installation or update of another bundle with the same symbolic name and version.

5.2 Managing Bundle Collisions

Currently the org.osgi.framework.bsnversion configuration property is used to specify whether multiple bundles having the same symbolic name and version (i.e. bundle collisions) may be installed. The problem with the current definition of org.osgi.framework.bsnversion is that it provides a global on/off (single/multiple) switch for the policy to allow multiple bundles to be installed having the same symbolic name and version. Hooks are not allowed to influence the policy at all. An additional hook is needed to influence this policy.

5.2.1 Managed Configuration Option

A new configuration value 'managed' is specified for the org.osgi.framework.bsnversion configuration property. When set to 'managed' the framework will use the new bundle collision hook for install and update operations which introduce a bundle collision. The bundle collision hooks are used by the framework to determine if the install or update operation should be allowed to continue or fail. The framework must only call registered bundle collision hooks if the 'managed' configuration value is used. If no bundle collision hooks are installed then the behavior of the framework is identical to when the 'single' configuration value is used for the org.osgi.framework.bsnversion configuration property. The default for org.osgi.framework.bsnversion is changed from 'single' to 'managed'.

5.3 Bundle Collision Hook

To intercept install and update operations which will result in a bundle collision a handler must register a bundle CollisionHook object as a service with the framework. When the 'managed' configuration value is used the framework must then call all registered hooks for each bundle install and update operation. The calling order of the hooks is defined by the reversed compareTo ordering of their Service Reference objects. That is, the service with the highest ranking number is called first. A collision hook is called for all install and update operations, if and only if the completion of the operation will result in a bundle collision.

The bundle Collision Hook interface has a single method:

- filterCollisions(int operationType, Bundle target, Collection<Bundle> collisionCandidates) – An install or update operation is in progress which will result in a possible bundle collision. The implementer of this method can optionally shrink the given collection of collision candidates.

The operationType parameter indicates the type of operation (installing or updating) which is being performed. Depending on the operation type the target bundle and collision candidates are the following:

- installing – The target is the bundle associated with the BundleContext used to call one of the install methods. The collision candidate collection contains the existing bundles installed which have the same symbolic name and version as the bundle being installed.

- updating – The target is the bundle used to call one of the update methods. The collision candidate collection contains the existing bundles installed which have the same symbolic name and version as the content the target bundle is being updated to.

This method can filter the list of collision candidates by removing potential collisions. Removing a collision candidate will allow the specified operation to succeed as if the collision candidate is not installed. If after calling all collision hooks the collision candidate collection is empty then the operation is allowed to proceed.

NOTE: It was decided to avoid the usage of a BundleEvent. A BundleEvent of type INSTALLED and UPDATED could have been used for the operation types INSTALLING and UPDATING respectively. The issue with this approach is that it required a real Bundle object to be available from the BundleEvent when in the middle of the INSTALLING operation. This seemed like a rather nasty thing to do especially since the bundle would not really be in a valid state yet (i.e. INSTALLED).

5.4 Java Doc

OSGi Javadoc

6/10/11 3:34 PM

Package Summary		Page
org.osgi.framework.hooks.bundle		8

Package org.osgi.framework.hooks.bundle

Interface Summary		Page
CollisionHook	OSGi Framework Bundle Collision Hook Service.	*9*

Interface CollisionHook

org.osgi.framework.hooks.bundle

```
public interface CollisionHook
```

OSGi Framework Bundle Collision Hook Service.

Bundles registering this service will be called during framework bundle install and update operations to determine if an install or update operation will result in a bundle symbolic name and version collision.

Version:
$Id: 64a98074fa8331b0cd21989f9c8583b99b2370cf $
ThreadSafe

Field Summary		Page
int	**INSTALLING** Specifies a bundle install operation is being performed.	9
int	**UPDATING** Specifies a bundle update operation is being performed.	9

Method Summary		Page
void	**filterCollisions**(int operationType, org.osgi.framework.Bundle target, Collection<org.osgi.framework.Bundle> collisionCandidates) Filter bundle collisions hook method.	9

Field Detail

INSTALLING

```
public static final int INSTALLING = 1
```

Specifies a bundle install operation is being performed.

UPDATING

```
public static final int UPDATING = 2
```

Specifies a bundle update operation is being performed.

Method Detail

filterCollisions

```
void filterCollisions(int operationType,
                      org.osgi.framework.Bundle target,
                      Collection<org.osgi.framework.Bundle> collisionCandidates)
```

Filter bundle collisions hook method. This method is called during the install or update operation. The operation type will be installing or updating. Depending on the operation type the target bundle and the collision candidate collection are the following:

- installing - The target is the bundle associated with the org.osgi.framework.BundleContext used to call one of the install methods. The collision candidate collection contains the existing bundles installed which have the same symbolic name and version as the bundle being installed.
- updating - The target is the bundle used to call one of the update methods. The collision candidate collection contains the existing bundles installed which have the same symbolic name and version as the content the target bundle is being updated to.

This method can filter the list of collision candidates by removing potential collisions. Removing a collision candidate will allow the specified operation to succeed as if the collision candidate is not installed.

Parameters:

operationType - the operation type. Must be the value of installing or updating.
target - the target bundle used to determine what collision candidates to filter.
collisionCandidates - a collection of collision candidates. The collection supports all the optional Collection operations except add and addAll. Attempting to add to the collection will result in an UnsupportedOperationException. The collection is not synchronized.

6 Command Line API

Command line API is not considered relevant to this design. Hooks are a very low level concept which should not be accessed by the command line.

7 JMX API

JMX API is not considered relevant to this design. Hooks are a very low level concept which should not be accessed by JMX.

8 Initial Spec Chapter

Provide a link to where the Initial Spec Chapter can be found. The Initial Spec Chapter is typically written by someone other than the author(s) of this RFC and represents a rewrite of this document as close as possible to

what will ultimately appear in the OSGi Specifications. It will be used by the Specification Editor as the basis for the ultimate specification chapter.

The spec template and writing guidelines can be found here:

https://www.osgi.org/members/svn/documents/trunk/templates/specification-template-oo.ott

https://www.osgi.org/members/svn/documents/trunk/templates/specwriting.pdf

9 Considered Alternatives

In bug 2020 (https://www.osgi.org/members/bugzilla/show_bug.cgi?id=2020) there is discussion of using the bundle FindHook to determine if the installation of multiple bundles having the same symbolic name and version is allowed. The idea was to use the bundle find hook to determine if the BundleContext installing the bundle had visibility to the bundles which collided with the symbolic name and version of the bundle being installed. This was a nice idea for detecting duplicate installations and allowing hooks to decide if the operation should succeed or not.

The issue with this approach is that there is no way to call the appropriate bundle FindHook when performing a bundle update. This is because bundle updates are performed without the use of a BundleContext. So there is no BundleContext perspective available in order to appropriately call the bundle FindHook. If the bundle FindHook would have taken a Bundle object instead of a BundleContext object then we may have been able to use it for updates also but the fact that a BundleContext is required makes this approach infeasible or incomplete.

10 Security Considerations

When Java permissions are in effect, this design is secured by ServicePermissions.

The bundle registering the various hook services must have the necessary ServicePermission.REGISTER. Since there are various hook services, we have fine grained control over what specific hooks a bundle can register.

11 Document Support

11.1 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

11.2 Author's Address

Name	Thomas Watson
Company	IBM Corporation
Address	
Voice	+1 512 286 9168
e-mail	tjwatson@us.ibm.com

11.3 Acronyms and Abbreviations

11.4 End of Document

RFC 0175 Version Update

Draft

22 Pages

Abstract

This RFC proposed an update to the Version and Version Range specifications and companion code. A new “snapshot” range of qualifiers is added to Version and a VersionRange class is introduced.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 9.1.

`Source code is shown in this typeface.`

0.3 Revision History

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

Revision	Date	Comments
Initial	25 July 2011	Initial Draft. API to be included later after initial reviews of the basic requirements and design. BJ Hargrave
2nd draft	27 July 2011	Added javadoc for Version and VersionRange classes. BJ Hargrave
3rd draft	5 August 2011	Updated from CPEG call: Updated requirements. Javdoc improvements. Added intersection method per Glyn and toFilterString method per Tom. BJ Hargrave

1 Introduction

The OSGi Core specification defines a Version[3] and Version Range[4] syntax. The OSGi Core Specification also defines a Version class[5]. This RFC proposed to update the Version and Version Range specifications to expand version qualifiers to support a range of so-called “snapshot” qualifiers which sort below all other qualifiers. This RFC also proposed to update the Version class accordingly and to introduce a VersionRange class.

2 Application Domain

Versions are used in many places in the OSGi specification. Capabilities have versions and Requirements can specify ranges of the versions of Capabilities that will satisfy them. Further to this, the specification defines Semantic Versioning[6] which further outlines the importance of versioning in the OSGi specifications.

A version consists of 4 components: 3 integer components specifying the major, minor and micro versions and the String qualifier of the version. In the external (String) form of a version, all the components are separated by a dot ('.'). For example: “1.2.3.qualifier”.

A version range consists of a single version or two versions specified with interval notation[7]. When a single version is specified, it is the left-closed end of the interval with Infinity at the right-open end of the interval.

3 Problem Description

During development of a bundle (or package), there is a period of time where the bundle has not been declared final. That is, the bundle has a planned version number once final, but that version number is not practically “consumed” until the bundle has been declared final. However, during development of the bundle, it must have a version number. This version number must be larger than the version number of the previous final version of the bundle but less than the version number intended for the bundle once final.

There are several usage patterns for version numbers which have emerged to deal with this problem. For example, some use an odd/even version number for the minor version to differentiate between development versions and final (release) versions. Some also place the build timestamp in the qualifier to distinguish all built versions of a bundle, but there is no clear marking which is the final version so dependents cannot mandate a final version.

4 Requirements

Requirements for the technical solution:

VERSION1 – There must be a total ordering over all version numbers.

VERSION2 – Version numbers must be able to be identified as pre-release (development).

VERSION3 – A version number identified as pre-release must support ~~multiple~~a range of qualifiers like version numbers identified as release.

VERSION4 – A version number identified as pre-release must sort before that version number when not identified as pre-release.

VERSION5 – Existing users of version numbers must continue to behave as before. That is, a set of bundles using Core 4.3 versioning, must behave identically when deployed on a framework supporting this new versioning proposal.

VERSION6 – If a version range having an endpoint specified without a qualifier (e.g. “[1.2.3,2.0.0)”) would include a release version with a qualifier of the empty string (e.g. “1.2.3”), then the version range must also include that version when identified as pre-release.

VERSION7 – If a version range having an endpoint specified without a qualifier (e.g. “[1.2.3,2.0.0)”) would exclude a release version with a qualifier of the empty string (e.g. “2.0.0”), then the version range must also exclude that version when identified as pre-release.

VERSION8 – When specifying versions in a version range, it must be possible to specify a version identified as a release version with an empty qualifier. Such a version is a so-called “mid-point” version in that it is between all pre-release qualifiers and all (non-empty) release qualifiers.

5 Technical Solution

5.1 Version

The version specification[3] is updated to allow a second class of qualifiers. The use of a *pre-release* qualifier identifies a version as pre-release. The dash ('-') separator is added to the syntax to separate the numerical portion of the version from the pre-release qualifier. Since the dash separator is not allowed in the current syntax, there is no issue with parsing existing version strings. The qualifiers of all existing versions will be separated with a dot ('.') and therefore do not identify the version as pre-release.

So a version will now have two classes of qualifiers: pre-release and release. Within each class of qualifier, normal `String.compareTo` sorting order applies. All pre-release qualifiers sort lower than all release qualifiers. A version string without a qualifier separator ('.' or '-') is defined to have a release qualifier of the empty string (“”). This is consistent with current version specification.

Furthermore, since we now have two classes of qualifiers, there is now a pre-release qualifier of the empty string and a release qualifier of the empty string. So version strings must allow the specification of each type of empty string qualifier. So the version syntax must allow an empty qualifier to be specified.

The version syntax is updated to:

```
version       ::= major( '.' minor ( '.' micro ( ( '.' | '-' ) qualifier )? )? )?
major         ::= number    // See 1.3.2
minor         ::= number
micro         ::= number
qualifier     ::= ( alphanum | '_' | '-' )*
```

The `version` production is updated to allow dash as a qualifier separator, '.' => ('.'|'-'), and the `qualifier` production is updated to allow the empty string, + => *.

Some examples:

“1.2.3” is a version with a major of 1, a minor of 2, a micro of 3 and a release qualifier of the empty string.

“1.2.3.” is equal to “1.2.3”. Both have the same numerical components and a release qualifier of the empty string. (The version string ending in a qualifier separator ('.' or '-') is now valid.)

“1.2.3.” is less than “1.2.3.x”. Both have the same numerical components but the release qualifier of the empty string is less than the release qualifier of “x”.

“1.2.3-x” is less than “1.2.3.”. Both have the same numerical components but the first version has a pre-release qualifier and the second has a release qualifier. And all pre-release qualifiers sort lower than any release qualifier.

“1.2.3-” is less than “1.2.3-x”. Both have the same numerical components but the pre-release qualifier of the empty string is less than the pre-release qualifier of “x”.

1.2.3- < 1.2.3-x < 1.2.3. < 1.2.3.x

We thus have a total ordering over all versions that is backwards compatible with the existing version specification.

5.2 Version Range

Given the two classes of qualifiers and the total ordering over all version, a small change is needed to the version range specification[4].

While no change is needed to the version range syntax, since it depends upon the version syntax which was updated above, we need to specify rules for whether a version is included in the version range when an endpoint in the version range does not specify a qualifier (either pre-release or release). If an endpoint of the version range does not specify a qualifier, then qualifier used by the endpoint will vary based upon whether the endpoint is left or right and open (exclusive) or closed (inclusive). This is necessary to provide meaningful and useful behavior for development time.

The follow two rules must be met:

1. If a version range having an endpoint specified without a qualifier (e.g. “[1.2.3,2.0.0)”) would include a version with a release qualifier of the empty string (e.g. “1.2.3”), then the version range must also include that version when identified as pre-release (e.g. “1.2.3-x”).

2. If a version range having an endpoint specified without a qualifier (e.g. “[1.2.3,2.0.0)”) would exclude a version with a release qualifier of the empty string (e.g. “2.0.0”), then the version range must also exclude that version when identified as pre-release (e.g. “2.0.0-x”).

There are four cases of endpoints with unspecified qualifiers to consider: left-closed, left-open, right-closed, right-open.

In the examples below, the capital letter (e.g. L) represents a version with specified major, minor and micro components but with no specified qualifier (e.g. “1.2.3”). L < R.

[L,R) – left-closed, right-open. This is the most common use of version range.

[L,R] – left-closed, right-closed. This is is generally only used when X = Y to indicate an exact version.

(L,R) – left-open, right-open. This is generally never used.

(L,R] – left-open, right-closed. This is generally never used.

For a left-closed endpoint (“[L,”)with an unspecified qualifier, the pre-release qualifier of empty string (“L-”) must be used by the version range. This allows version L with any qualifier to be included in the range. This facilitates development by including pre-release L versions.

For right-open endpoint (“,R)”) with an unspecified qualifier, the pre-release qualifier of empty string (“R-”) must be used by the version range. This excludes version R with any qualifier from being included in the range. This facilitates development by excluding all R versions.

For a left-open endpoint (“(L,”)with an unspecified qualifier, the release qualifier of empty string (“L.”) must be used by the version range. This allows version L with any non-empty release qualifier to be included in the range. All pre-release L versions and L with a release qualifier of the empty string are excluded.

For right-closed endpoint (“,R]”) with an unspecified qualifier, the release qualifier of empty string (“R.”) must be used by the version range. This allows version R with any pre-release qualifier and the release qualifier of the empty string to be included in the range. All release L versions with a non-empty release qualifier are excluded.

So, in summary:

[L,R) => [L-,R-) – All L versions are included. No R versions are included. This is the most common use of version ranges.

[L,R] => [L-,R.) – All L versions are included. No R versions with non-empty release qualifiers are included. This is generally only used when L=R for exact versions. So, for the exact version of X, [X,X] => [X-,X.] includes all pre-release X qualifiers and the release qualifier of the empty string.

(L,R) => [L.,R-) – L versions with non-empty release qualifiers are included. No R versions are included.

(L,R] => [L.,R.) – L versions with non-empty release qualifiers are included. No R versions with non-empty release qualifiers are included.

Of course, to avoid these version range endpoint unspecified qualifier rules, the developer can always specify a qualifier on the endpoints, even if it is the empty string.

These changes provide the same behavior as today when only release versions are used and add useful behavior for development with pre-release versions.

6 Javadoc

Javadoc for the Version and VersionRange classes. The Version class has been updated to support the new pre-release qualifiers and the new VersionRange class is introduced.

OSGi Javadoc

8/5/11 5:12 PM

Package Summary		Page
org.osgi.framework	Framework Package Version 1.7.	9

Package org.osgi.framework

Framework Package Version 1.7.

See:
Description

Class Summary		Page
Version	Version identifier for capabilities such as bundles and packages.	10
VersionRange	Version range.	16

Package org.osgi.framework Description

Framework Package Version 1.7.

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

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.framework; version="[1.7,2.0)"
```

Class Version

org.osgi.framework

```
java.lang.Object
  └ org.osgi.framework.Version
```

All Implemented Interfaces:
Comparable<Version>

```
public class Version
extends Object
implements Comparable<Version>
```

Version identifier for capabilities such as bundles and packages.

Version identifiers have four components.

1. Major version. A non-negative integer.
2. Minor version. A non-negative integer.
3. Micro version. A non-negative integer.
4. Qualifier. A text string. See `Version(String)` for the format of the qualifier string.

Versions can also be identified as release versions or pre-release versions. Given the same numerical components, the qualifiers of all pre-release version sort lower than the qualifiers of release versions. In the external format, `String`, of a version, release versions use `"."` to separate the numerical components from the qualifier and pre-release versions use `"-"` to separate the numerical components from the qualifier.

`Version` objects are immutable.

Since:
1.3
Version:
$Id: 81642bc6af927534df4fbfcd0bb7962b181ca0b1 $
Immutable

Field Summary	Page
static Version **emptyVersion** The empty version "0.0.0".	11

Constructor Summary	Page
Version(int major, int minor, int micro) Creates a release version identifier from the specified numerical components.	11
Version(int major, int minor, int micro, String qualifier) Creates a release version identifier from the specified components.	11
Version(int major, int minor, int micro, String qualifier, boolean release) Creates a version identifier from the specified components.	12
Version(String version) Creates a version identifier from the specified string.	12

Method Summary	Page
int **compareTo**(Version other) Compares this `Version` object to another `Version`.	14
boolean **equals**(Object object) Compares this `Version` object to another object.	14

int	**getMajor**() Returns the major component of this version identifier.	*13*
int	**getMicro**() Returns the micro component of this version identifier.	*13*
int	**getMinor**() Returns the minor component of this version identifier.	*13*
String	**getQualifier**() Returns the qualifier component of this version identifier.	*13*
int	**hashCode**() Returns a hash code value for the object.	*14*
boolean	**isReleaseVersion**() Returns `true` if the version is a release version and `false` if the version is a pre-release version.	*13*
static Version	**parseVersion**(String version) Parses a version identifier from the specified string.	*12*
String	**toString**() Returns the string representation of this version identifier.	*14*

Field Detail

emptyVersion

```
public static final Version emptyVersion
```

The empty version "0.0.0".

Constructor Detail

Version

```
public Version(int major,
               int minor,
               int micro)
```

Creates a release version identifier from the specified numerical components.

The qualifier is set to the empty string and the version is a release version.

Parameters:
`major` - Major component of the version identifier.
`minor` - Minor component of the version identifier.
`micro` - Micro component of the version identifier.
Throws:
`IllegalArgumentException` - If the numerical components are negative.

Version

```
public Version(int major,
               int minor,
               int micro,
               String qualifier)
```

Creates a release version identifier from the specified components.

The version is a release version.

Parameters:
`major` - Major component of the version identifier.
`minor` - Minor component of the version identifier.
`micro` - Micro component of the version identifier.
`qualifier` - Qualifier component of the version identifier. If `null` is specified, then the qualifier will be set to the empty string.
Throws:
`IllegalArgumentException` - If the numerical components are negative or the qualifier string is invalid.

Version

```
public Version(int major,
               int minor,
               int micro,
               String qualifier,
               boolean release)
```

Creates a version identifier from the specified components.

Parameters:
`major` - Major component of the version identifier.
`minor` - Minor component of the version identifier.
`micro` - Micro component of the version identifier.
`qualifier` - Qualifier component of the version identifier. If `null` is specified, then the qualifier will be set to the empty string.
`release` - `true` if a release version or `false` if a pre-release version.
Throws:
`IllegalArgumentException` - If the numerical components are negative or the qualifier string is invalid.
Since:
1.7

Version

```
public Version(String version)
```

Creates a version identifier from the specified string.

Version string grammar:

```
version ::= major('.'minor('.'micro(('.'|'-')qualifier)?)?)?
major ::= digit+
minor ::= digit+
micro ::= digit+
qualifier ::= (alpha|digit|'_'|'-')*
digit ::= [0..9]
alpha ::= [a..zA..Z]
```

Parameters:
`version` - String representation of the version identifier. There must be no whitespace in the argument.
Throws:
`IllegalArgumentException` - If `version` is improperly formatted.

Method Detail

parseVersion

```
public static Version parseVersion(String version)
```

Parses a version identifier from the specified string.

See `Version(String)` for the format of the version string.

Parameters:
`version` - String representation of the version identifier. Leading and trailing whitespace will be ignored.
Returns:
A `Version` object representing the version identifier. If `version` is `null` or the empty string then `emptyVersion` will be returned.
Throws:
`IllegalArgumentException` - If `version` is improperly formatted.

---

getMajor

```
public int getMajor()
```

Returns the major component of this version identifier.

Returns:
The major component.

---

getMinor

```
public int getMinor()
```

Returns the minor component of this version identifier.

Returns:
The minor component.

---

getMicro

```
public int getMicro()
```

Returns the micro component of this version identifier.

Returns:
The micro component.

---

getQualifier

```
public String getQualifier()
```

Returns the qualifier component of this version identifier.

Returns:
The qualifier component.

---

isReleaseVersion

```
public boolean isReleaseVersion()
```

Returns `true` if the version is a release version and `false` if the version is a pre-release version.

Returns:
`true` if the version is a release version and `false` if the version is a pre-release version.
Since:
1.7

---

toString

```
public String toString()
```

Returns the string representation of this version identifier.

The format of the version string will be `major.minor.micro.qualifier` if it is a `release version` or `major.minor.micro-qualifier` if the version is a pre-release version.

Overrides:
`toString` in class `Object`
Returns:
The string representation of this version identifier.

---

hashCode

```
public int hashCode()
```

Returns a hash code value for the object.

Overrides:
`hashCode` in class `Object`
Returns:
An integer which is a hash code value for this object.

---

equals

```
public boolean equals(Object object)
```

Compares this `Version` object to another object.

A version is considered to be **equal to** another version if the major, minor and micro components are equal, the qualifier component is equal (using `String.equals`) and both versions are release or pre-release.

Overrides:
`equals` in class `Object`
Parameters:
`object` - The `Version` object to be compared.
Returns:
`true` if `object` is a `Version` and is equal to this object; `false` otherwise.

---

compareTo

```
public int compareTo(Version other)
```

Compares this `Version` object to another `Version`.

A version is considered to be **less than** another version if its major component is less than the other version's major component, or the major components are equal and its minor component is less than the other version's minor component, or the major and minor components are equal and its micro component is less than the other version's micro component, or the major, minor and micro components are equal and it's a pre-release version and the other version is a release version, or the major, minor and micro

components are equal and both versions are release or pre-release and it's qualifier component is less than the other version's qualifier component (using `String.compareTo`).

A version is considered to be **equal to** another version if the major, minor and micro components are equal, both versions are release or pre-release and the qualifier components are equal (using `String.compareTo`).

Specified by:

`compareTo` in interface `Comparable`

Parameters:

`other` - The `Version` object to be compared.

Returns:

A negative integer, zero, or a positive integer if this version is less than, equal to, or greater than the specified `Version` object.

Throws:

`ClassCastException` - If the specified object is not a `Version` object.

Class VersionRange

org.osgi.framework

```
java.lang.Object
  └ org.osgi.framework.VersionRange
```

```
public class VersionRange
extends Object
```

Version range. A version range is an interval describing a set of `versions`.

A range has a left (lower) endpoint and a right (upper) endpoint. Each endpoint can be open (excluded from the set) or closed (included in the set).

`VersionRange` objects are immutable.

Since:
1.7
Version:
$Id: cf78e34820ea562bd56c2bad7aeb5c9995730da2 $
Immutable

Field Summary		Page
static char	**LEFT_CLOSED** The left endpoint is closed and is included in the range.	17
static char	**LEFT_OPEN** The left endpoint is open and is excluded from the range.	17
static char	**RIGHT_CLOSED** The right endpoint is closed and is included in the range.	17
static char	**RIGHT_OPEN** The right endpoint is open and is excluded from the range.	17

Constructor Summary	Page
VersionRange(char leftType, Version leftEndpoint, Version rightEndpoint, char rightType) Creates a version range from the specified versions.	17
VersionRange(String range) Creates a version range from the specified string.	18

Method Summary		Page
boolean	**equals**(Object object) Compares this `VersionRange` object to another object.	20
Version	**getLeft**() Returns the left endpoint of this version range.	18
char	**getLeftType**() Returns the type of the left endpoint of this version range.	18
Version	**getRight**() Returns the right endpoint of this version range.	18
char	**getRightType**() Returns the type of the right endpoint of this version range.	19
int	**hashCode**() Returns a hash code value for the object.	20

boolean	**includes**(Version version) Returns whether this version range includes the specified version.	19
VersionRange	**intersection**(VersionRange... ranges) Returns the intersection of this version range with the specified version ranges.	19
boolean	**isEmpty**() Returns whether this version range is empty.	19
String	**toFilterString**(String attributeName) Returns the filter string for this version range using the specified attribute name.	20
String	**toString**() Returns the string representation of this version range.	19

Field Detail

LEFT_OPEN

```
public static final char LEFT_OPEN = 40
```

The left endpoint is open and is excluded from the range.

The value of LEFT_OPEN is '('.

LEFT_CLOSED

```
public static final char LEFT_CLOSED = 91
```

The left endpoint is closed and is included in the range.

The value of LEFT_CLOSED is '['.

RIGHT_OPEN

```
public static final char RIGHT_OPEN = 41
```

The right endpoint is open and is excluded from the range.

The value of RIGHT_OPEN is ')'.

RIGHT_CLOSED

```
public static final char RIGHT_CLOSED = 93
```

The right endpoint is closed and is included in the range.

The value of RIGHT_CLOSED is ']'.

Constructor Detail

VersionRange

```
public VersionRange(char leftType,
                    Version leftEndpoint,
                    Version rightEndpoint,
                    char rightType)
```

Creates a version range from the specified versions.

Parameters:
`leftType` - Must be either `LEFT_CLOSED` or `LEFT_OPEN` .
`leftEndpoint` - Left endpoint of range. Must not be `null`.
`rightEndpoint` - Right endpoint of range. May be `null` to indicate the right endpoint is *Infinity*.
`rightType` - Must be either `RIGHT_CLOSED` or `RIGHT_OPEN`.
Throws:
`IllegalArgumentException` - If the arguments are invalid.

VersionRange

```
public VersionRange(String range)
```

Creates a version range from the specified string.

Version range string grammar:

```
range ::= interval | atleast
interval ::= ( '[' | '(' ) left ',' right ( ']' | ')' )
left ::= version
right ::= version
atleast ::= version
```

Parameters:
`range` - String representation of the version range. The versions in the range must contain no whitespace. Other whitespace in the range string is ignored.
Throws:
`IllegalArgumentException` - If `range` is improperly formatted.

Method Detail

getLeft

```
public Version getLeft()
```

Returns the left endpoint of this version range.

Returns:
The left endpoint.

---

getRight

```
public Version getRight()
```

Returns the right endpoint of this version range.

Returns:
The right endpoint. May be `null` which indicates the right endpoint is *Infinity*.

---

getLeftType

```
public char getLeftType()
```

Returns the type of the left endpoint of this version range.

Returns:
LEFT_CLOSED if the left endpoint is closed or LEFT_OPEN if the left endpoint is open.

---

getRightType

```
public char getRightType()
```

Returns the type of the right endpoint of this version range.

Returns:
RIGHT_CLOSED if the right endpoint is closed or RIGHT_OPEN if the right endpoint is open.

---

includes

```
public boolean includes(Version version)
```

Returns whether this version range includes the specified version.

Parameters:
`version` - The version to test for inclusion in this version range.
Returns:
`true` if the specified version is included in this version range; `false` otherwise.

---

intersection

```
public VersionRange intersection(VersionRange... ranges)
```

Returns the intersection of this version range with the specified version ranges.

Parameters:
`ranges` - The version ranges to intersect with this version range.
Returns:
A version range representing the intersection of this version range and the specified version ranges. If no version ranges are specified, then this version range is returned.

---

isEmpty

```
public boolean isEmpty()
```

Returns whether this version range is empty. A version range is empty if the set of versions defined by the interval is empty.

Returns:
`true` if this version range is empty; `false` otherwise.

---

toString

```
public String toString()
```

Returns the string representation of this version range.

The format of the version range string will be a version string if the right end point is *Infinity* (`null`) or an interval string.

Overrides:
`toString` in class `Object`
Returns:
The string representation of this version range.

---

hashCode

```
public int hashCode()
```

Returns a hash code value for the object.

Overrides:
`hashCode` in class `Object`
Returns:
An integer which is a hash code value for this object.

---

equals

```
public boolean equals(Object object)
```

Compares this `VersionRange` object to another object.

A version range is considered to be **equal to** another version range if both the endpoints and their types are equal or if both version ranges are `empty`.

Overrides:
`equals` in class `Object`
Parameters:
`object` - The `VersionRange` object to be compared.
Returns:
`true` if `object` is a `VersionRange` and is equal to this object; `false` otherwise.

---

toFilterString

```
public String toFilterString(String attributeName)
```

Returns the filter string for this version range using the specified attribute name.

Parameters:
`attributeName` - The attribute name to use in the returned filter string.
Returns:
A filter string for this version range using the specified attribute name.
Throws:
`IllegalArgumentException` - If the specified attribute name is not a valid attribute name.
See Also:
"Core Specification, Filters, for a description of the filter string syntax."

---

7 Considered Alternatives

None.

8 Security Considerations

None.

9 Document Support

9.1 References

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

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

[3] Version specification. Section 3.2.5 in the Core 4.3 Specification.

[4] Version Range specification. Section 3.2.6 in the Core 4.3 Specification.

[5] Version class specification. Section 9.1.30 in the Core 4.3 Specification.

[6] Semantic Versioning. Section 3.7.3 in the Core 4.3 Specification.

[7] Interval Notation. http://en.wikipedia.org/wiki/Interval_%28mathematics%29

9.2 Author's Address

Name	BJ Hargrave
Company	IBM Corporation
e-mail	hargrave@us.ibm.com

9.3 Acronyms and Abbreviations

9.4 End of Document

RFC 176 Declarative Services 1.2

Draft

8 Pages

Abstract

Update Declarative Services to 1.2 with some features requested via bug reports.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 8.1.

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

0.3 Revision History

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

Revision	Date	Comments
Initial	6 Sep 2011	Initial draft.
2nd draft	9 Sep 2011	Addressed comments from Tom Watson.
3rd draft	16 Sep 2011	Fixed typo. Renamed abstemious policy option to reluctant. Removed localization of component properties.

1 Introduction

The current published version of Declarative Services is 1.1. Since that version was published, a set of feature requests has been submitted as bug reports. This RFC will propose several minor enhancements to the DS spec.

2 Application Domain

The Declarative Services model uses a declarative model for publishing, finding and binding to OSGi services. This model simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies. This minimizes the amount of code a programmer has to write; it also allows service components to be loaded only when they are needed.

3 Problem Description

3.1 Bug 1956

A component may access the service properties of a bound service which are passed to the bind method. A bound service may have its service properties changed. DS provides no way for the component to be notified of service property changes for a bound service (assuming the property changes do not cause the service to become unbound).

3.2 Bug 1960

A component may use a static policy to access referenced services. This simplifies the implementation of the component since it does not have to handle the dynamism of services. However, sometimes a component may want to track the latest set of target services yet not be written to handle service dynamics. The component wants to be reactivated each time the set of target services changes. DS provides no way for a static reference to always track the latest set of target services.

3.3 Bug 1963

With a unary cardinality, a static or dynamic reference is bound to the highest ranked service. However, if a higher ranked service is registered after that binding, the newer, higher ranked service will not replace the bound service. DS provides no way for a unary reference to always track the highest ranked service.

4 Requirements

DS2 – A component must be able to be notified when the service properties of a bound service are changed (if the change does not cause the bound service to become unbound.)

DS3 – A component must be able to use a static reference to track the latest set of target services.

DS4 – A component must be able to use a unary reference to track the highest ranked target service.

5 Technical Solution

5.1 Service Properties Update

To support service property update notification, a new `updated` attribute is added to the `reference` element. The method named by the updated attribute is called when the service properties of a bound service are changed. If the service properties change causes the service to no longer be a target service, then the service is unbound and the updated method is not called.

The updated method uses the same method prototypes as the bind and unbind methods. If an updated method throws an exception, SCR must log an error message containing the exception with the Log Service, if present and the service will remain bound and the component will not be deactivated.

Adding a new attribute will require updating the version of the XML schema.

5.2 Greedy Policy Option

Both the `static` and `dynamic` reference policies are *reluctant* (non-greedy). When a new target service become available, dynamic multiple references will react to that service and bind to it. But static references and unary references will ignore the new target service.

A new `policy-option` attribute is added to the `reference` element. This will allow a `greedy` policy option to be specified instead of the default `reluctant` policy option.

The following table describes the default behavior (also DS 1.1 behavior) for an activated component when a new target service becomes available. The behavior is also obtained when the reluctant policy option is specified.

`policy-option="reluctant"`	`static`	`dynamic`
0..1	Ignore new target service.	If no service is currently bound, bind new target service. Otherwise, ignore new target service.
1..1	Ignore new target service.	Ignore new target service.
0..n	Ignore new target service.	Bind new target service (same for both policy options.)
1..n	Ignore new target service.	Bind new target service (same for both policy options.)

When the new greedy policy option is specified, the following new behavior for an activated component is obtained when a new target service becomes available.

policy-option=”greedy”	static	dynamic
0..1	If no service is currently bound or the new target service is higher ranked than the currently bound service, reactivate to bind new target service.	If no service is currently bound, bind new target service. Otherwise, if the new target service is higher ranked than the currently bound service, unbind the currently bound service and bind the new target service.
1..1	If the new target service is higher ranked than the currently bound service, reactivate to bind new target service.	If the new target service is higher ranked than the currently bound service, unbind the currently bound service and bind the new target service.
0..n	Reactivate to bind new target service and existing bound services, if any.	Bind new target service (same for both policy options.)
1..n	Reactivate to bind new target service and existing bound services.	Bind new target service (same for both policy options.)

Adding a new attribute will require updating the version of the XML schema.

6 Considered Alternatives

6.1 Abstemious

Renamed the policy option abstemious to reluctant.

6.2 Localization

After discussion in CPEG, we decided localizing component properties is not useful in any practical way. Requirement DS1 was removed.

DS1 – Localization of component properties must be supported.

> In the 1.1 DS spec, the properties element is defined with a single required attribute: `entry`. The value of the entry attribute specifies the path of a properties file entry in the bundle. For example,
>
> ```
> <properties entry="OSGI-INF/vendor.properties" />
> ```
>
> This path include the extension, if any, of the entry. To support a localization scheme for component properties like *3.11 Localization* in the core spec, the properties element really needs to specify the basename, without extension, of the "family" of entries. I don't think it is wise to require SCR to parse off the file extension and attempt to insert locale names. It would be much cleaner to define a new attribute to specify the basename and indicate the properties element wishes to participate in localization. For example,

```
<properties basename="OSGI-INF/vendor" />
```

Properties file entries, like service component descriptions, can be located in the bundle or its attached fragments. `Bundle.findEntries` can be used to load the properties file entries.

Adding a new attribute will require updating the version of the XML schema.

7 Security Considerations

These changes do not alter the security considerations of the DS spec.

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

Name	BJ Hargrave
Company	IBM
e-mail	hargrave@us.ibm.com

8.3 Acronyms and Abbreviations

DS – Declarative Services

8.4 End of Document

The OSGi Alliance and its members specify, create, advance, and promote wide industry adoption of an open delivery and management platform for application services in home, commercial buildings, automotive and industrial environments. The OSGi Alliance serves as the focal point for a collaborative ecosystem of service providers, developers, manufacturers, and consumers. The OSGi specifications define a standardized, component oriented, computing environment for networked services. OSGi technology is currently being delivered in products and services shipping from several Fortune 100 companies. The OSGi Alliance's horizontal software integration platform is ideal for both vertical and cross-industry business models within home, vehicle, mobile and industrial environments. As an independent non-profit corporation, the OSGi Alliance also provides for the fair and uniform creation and distribution of relevant intellectual property – including specifications, reference implementations, and test suites – to all its members.

HOW TO REACH US:

OSGi Alliance
Bishop Ranch 6
2400 Camino Ramon, Suite 375
San Ramon, CA 94583 USA

Phone: +1.925.275.6625
E-mail: marketinginfo@osgi.org
Web: http://www.osgi.org