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

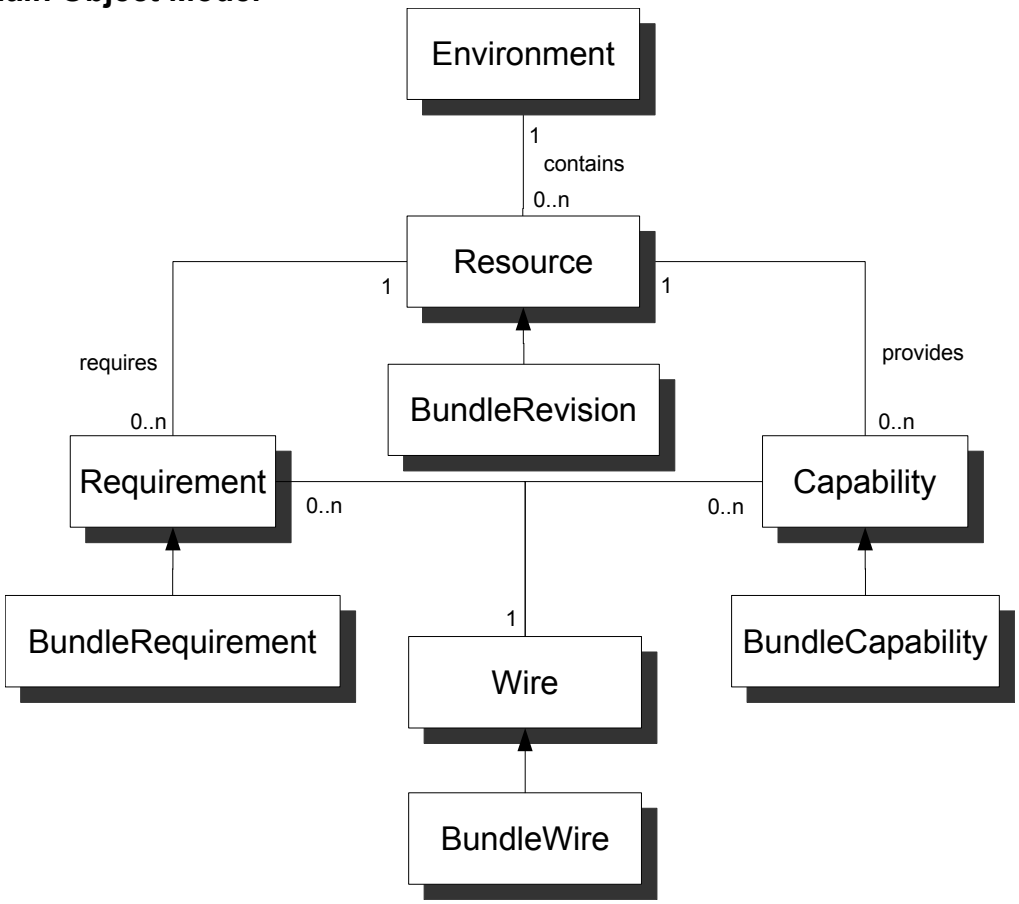
- The repository must be able to scale to ten thousand bundles
- Compliant with other OSGi services
- Easy to use
- It must be possible to implement a repository with a simple file. That is, a server must not be required

5 Technical Solution

5.1 Entities

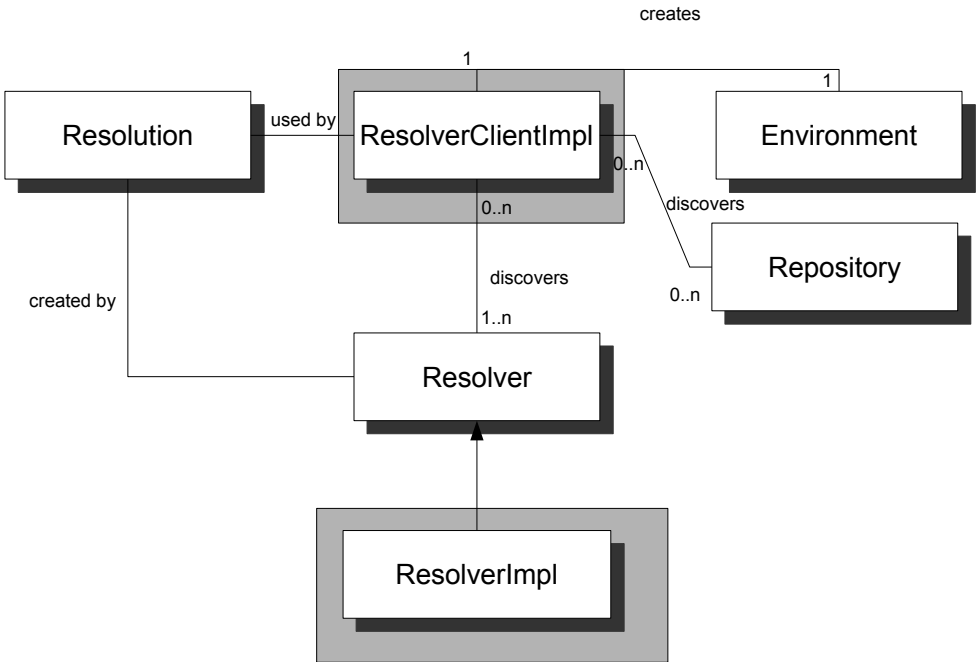
- Attribute – A map of key value properties that define the idenitity of a Capability
- Directive – A map of key value instructions that direct a Resolver in the processing of Requirements and Capabilities
- Capability – A namespaced set of attributes and directives
- Requirement – An assertion on a 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

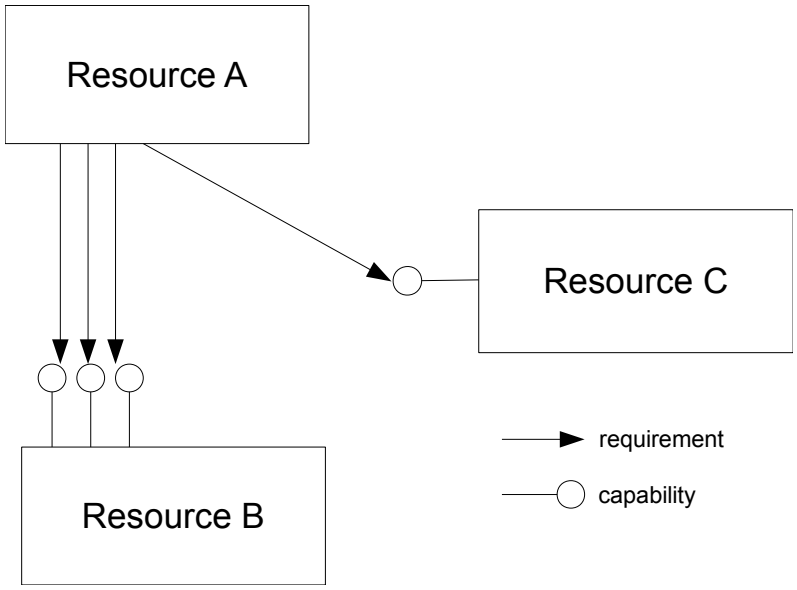
- A bundle,
- A certificate
- A configuration file

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

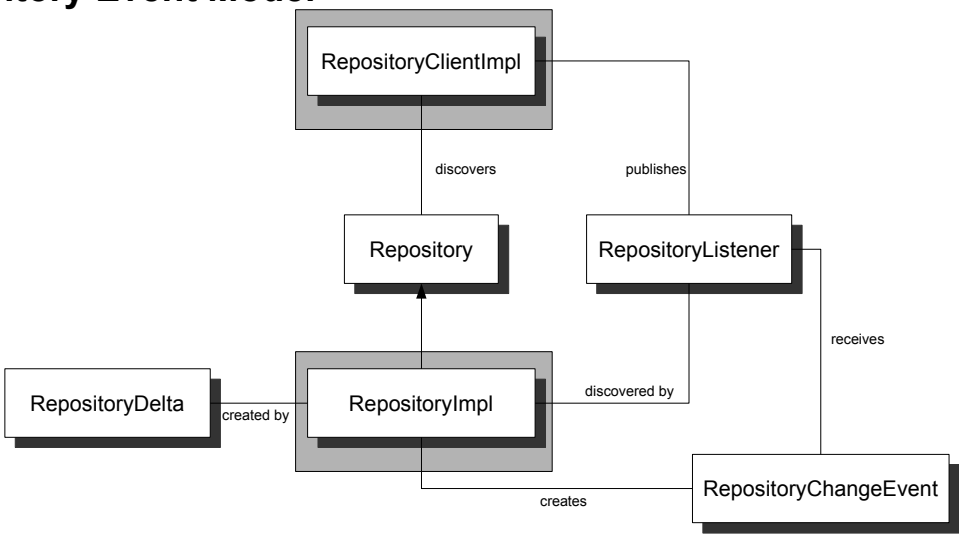
The value of the repository would be greatly enhanced if we would support a licensing model. Currently, certain bundles require the authentication so they can not be directly downloaded. This makes OBR like solutions for these bundles impossible in a standard way.

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