RFC 224: Resolver Service Updates

Draft

23 Pages

Abstract

Updates to the Resolver Service for Release 7.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

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

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

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

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

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

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

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

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

0.2 Trademarks

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

0.3 Feedback

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

0.4 Table of Contents

0.5 Terminology and Document Conventions

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

`Source code is shown in this typeface.`

0.6 Revision History

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

Revision	Date	Comments
Initial	04/29/16	Initial Draft Thomas Watson, IBM

1 Introduction

This RFC collects a numbers of requested enhancements to the Resolver Service that were suggested after Release 6 design work was completed.

2 Application Domain

The Resolver Service was first released as part of Release 5. From the Version 1.0 spec:

> Resolving transitive dependencies is a non-trivial process that requires careful design to achieve the required performance since the underlying problem is NP-complete. OSGi frameworks have always included such resolvers but these were built into the frameworks. They were not usable outside the framework for tooling, for example automatically finding the dependencies of a bundle that needs to be installed.

The Resolver Service provides the base for provisioning, build and diagnostic toolings. The Resolver service could also be used by a Framework implementation for resolving bundles at runtime, but the 1.0 version of the Resolver service did not focus on this use case. As a result the specification is lacking some basic runtime functionality needed in order to implement the Core Framework specification.

The Equinox and Felix Core Framework implementations both provide the Resolver Service implementation and use the Resolver implementation internally to resolve bundles at runtime. In both cases some additional implementation specific functionality was added to allow the resolver to be used for runtime bundle resolution.

3 Problem Description

3.1 Resolve Related Resources

When resolving bundles at runtime Framework implementations have several different approaches they can take to resolving the set of installed bundles. Below are the approaches taken by the Equinox and Felix Framework implementations

3.1.1 Resolve All Bundles At Once

With this approach the framework implementation of the ResolveContext `getOptionalResources()` places every existing bundle in the INSTALLED state as part of the collection of optional resources. This is guaranteed to pull in every bundle into the resolve operation and resolve as many as possible in one go.

The advantages of this approach is that it gives the resolver algorithm the greatest flexibility to pick a capability provider that will result in a consistent class space. The other advantage is that it will allow the greatest number of fragments to attach to their hosts that are being resolved at the same time.

The disadvantage of this approach is that will more choices presented to the resolver the more likely the resolve algorithm is going to explode trying to find a valid solution. This may result in situations where a resolution operation appears to endlessly loop, crash with an out of memory exception, or stack overflow exception.

3.1.2 Resolve One Root Bundle At a Time

With this approach the framework implementation of the ResolveContext `getOptionalResources()` or `getMandatoryResources()` places a single bundle in the INSTALLED state as part of the collection resources to resolve. The main point is the resolve operation starts with a single root resource to resolve. The resolver then looks at the root resource's requirements and calls the ResolveContext to find providers which may pull in additional resources to resolve.

The advantage of this approach is that it limits the resolver's choice of possible providers to the tree required to resolve the single root resource. This may help reduce the possible explosion of the resolver algorithm when finding a valid solution for a consistent class space.

The disadvantage of this approach is that limiting the choices of the resolver by locking in provider selections of earlier root resources may make it impossible to resolve other bundles later. Another disadvantage is that fragment resources will only get pulled into the resolution if they have capabilities that are directly depended on by other bundles being resolved at the same time. This may leave a fragment unattached while its host bundle becomes resolved. Later the fragment bundle will not be allowed to resolve because its host is already resolved.

3.1.2.1 Pull in Related Resources

In order to allow a framework implementation to resolve bundles using the single root bundle approach the resolver needs the ability to pull in related resources when resolving a resource. For example, when resolving a host bundle at runtime it is mostly likely desired to pull in as many applicable fragments as possible to the resolve operation. The Felix Resolver implementation extended the ResolveContext with a implementation specific type which added a new method

```
public Collection<Resource> getOndemandResources(Resource host);
```

Each time the resolver attempts to resolve a resource it asks the ResolveContext for additional “on demand” resources which are to be added to the current resolve operation as if they were part of the original optional resources for the resolve operation.

3.2 Resolve Dynamic Package Imports

At runtime the framework must support dynamic package resolution. With the current Resolver API this is possible, but very cumbersome. It would involve treating the resolved bundle with the dynamic import as if it was unresolved and attempting to resolve it again. The resolve context would then have to ensure all non-dynamic requirements got re-wired to the same capabilities they already are wired to and then have an extra requirement that is used to represent the current package being requested for dynamic resolution.

This approach requires a lot of extra work during runtime class loading. The Felix resolver implementation adds a new implementation specific method that allows dynamic resolution of a requirement for an already resolved resource. The following method is available:

```
public Map<Resource, List<Wire>> resolve(
  ResolveContext rc, Resource host, Requirement dynamicReq,
  List<Capability> matches)
```

This method allows the resolution of the dynamicReq that is associated with a host resource to establish new wires from the host resource. This method also allows additional resources to get pulled into the resolve operation in order to resolve the dynamicReq.

3.3 Allow the ResolveContext to Cancel a Resolve Operation

The resolution problem is an np-complete problem. Depending on the resolver algorithm it is possible to introduce problem sets that will result in an apparent endless-loop, out of memory, or stack overflow. It would be useful if the ResolveContext could provide some heuristics to force the resolver to cancel the existing resolution operation that is “taking forever” or “too much memory” etc.

For example, a framework implementation may decide to first try to resolve a large set of bundles all at once because that gives the resolver the greatest level of flexibility to find a consist class space. But after a certain amount of time or memory usage the framework may want to force the resolution operation to quit. At that point the framework may decide to take a more manageable approach by resolving only a single root bundle at a time.

4 Requirements

4.1 ResolveContext

- S0010 – A ResolveContext must be able to add related resources to a resolve operation in response to a specific resource being pulled into a resolve operation.
- S0020 – When related resources are added to a resolve operation, the resolve operation must treat the related resource as optional unless the related resource is already considered mandatory.
- S0030 – A ResolveContext must be able to add related resources that already have existing wirings. For example, to attach a resolved fragment to another host resource.
- S0040 – A ResolveContext must be able to add any type of related resource to the resolve operation. For example, related resources are not restricted to the osgi.fragment type.
- S0070 – A ResolveContext must be able to cancel an ongoing resolve operation

4.2 Resolver

- S0100 – A Resolver must be able to dynamically wire a requirement for an existing wiring of a host resource.
- S0110 – A Resolver must only support dynamic resolution of requirements that use the osgi.wiring.package namespace
- S0120 – A Resolver must not dynamically wire to a package capability when the host wiring is already wired to a package capability with the same package name
- S0130 – A Resolver must not dynamically wire to a package capability when the host wiring is already providing a package capability with the same package name

- S0140 – A Resolver must be able to resolve additional resources in order to resolve a dynamic requirement.

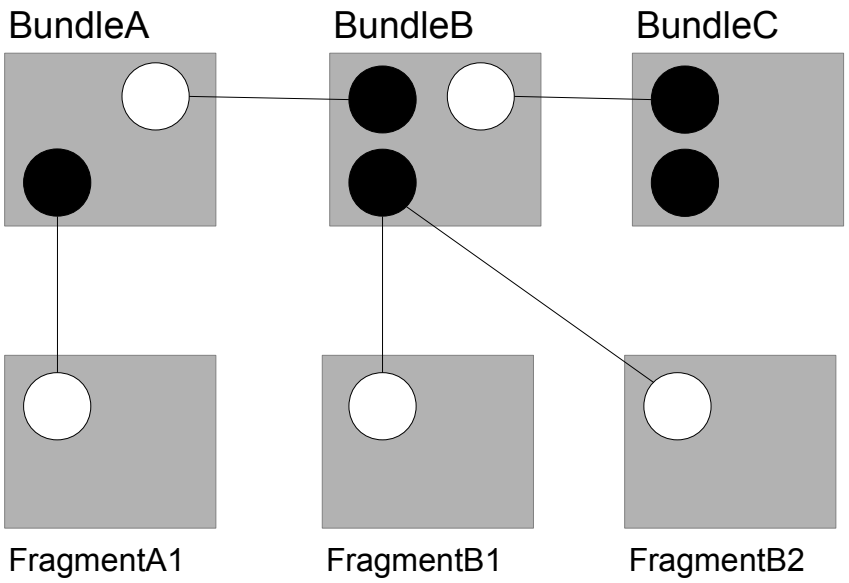
5 Technical Solution

5.1 Resolve Related Resources

In order to resolve related resources the ResolveContext is extended to include the following method:

```
public Collection<Resource> findRelatedResources(Resource resource) {
            return Collections.emptyList();
}
```

This method is used by the resolver to find resources that are related to the given resource. The Resolver implementation will call this method for each resource that is pulled into a resolve operation. Use the following set of bundles as an example:



A resolve operation starts with a collection of mandatory and optional resources to resolve according to the ResolveContext These are considered the root resources of the resolve operation. In the diagram above a

resolve operation is started with a single root resource BundleA. BundleA has a single requirement the pulls in resource BundleB as providing a matching capability. BundleB has a single requirement that pulls in resource BundleC as a providing matching capability. Using the 1.0 version of the Resolver Service implementation will result in a wiring map being returned with only the resources BundleA, BundleB, and BundleC as resolved with a new set of wires. All fragment resources FragmentA1, FragmentB1 and FragmentB2 would be left out of the resolution results.

With the new `findRelatedResources(Resource resource)` method the resolver will call the ResolveContext for each resource that is pulled into the resolve operation. This includes the root resources as well as the related resources returned by `findRelatedResources(Resource resource)`. In the example above the ResolveContext has the option to return FragmentA1 as a related resource for for BundleA and FragmentB1 and FragmentB2 as related resources for BunldeB. This results in all three fragments being included in the resolver

TODO: Do we really want to call back the the ResolveContext for each resource the resolve context adds as related resources, or should we require the ResolveContext to do that work as part of the original call to findRelatedResources?

Related resources that are already considered mandatory or optional resources have no effect on the final resolution result. Related resources that are not already considered mandatory or optional resources are added to the resolve operation as if they are optional resources. Failing to resolve such related resources does not result in a ResolutionException

Related resources may already have existing wirings. If the resource with an existing wiring does not have the osgi.fragment type then the additional resource has no effect on the final resolution result. If the resource with and existing wiring has the osgi.fragment type then the fragment resource is allowed to attach to new host osgi.bundle type resources if they provide the matching osgi.wiring.host capability for the fragment.

A ResolveContext is free to return any type of resource as a related resource, the types are not restricted to the osgi.fragment type.

5.2 Resolving Dynamic Package Imports

In order to resolve osgi.wiring.package requirements with the resolution directive of “dynamic” the Resolver interface is extended with the following method:

```
public Map<Resource,List<Wire>> dynamicResolve(
  ResolveContext context,
  Wiring wiring,
  Requirement requirement)
    throws ResolutionException;
```

The requirement must use the osgi.wiring.package namespace and have a resolution directive of “dynamic” and be hosted by the provided wiring. If the requirement has a cardinality directive of “single” then the wiring provided must not have an existing wire that uses the provided requirement.

The resolver uses the requirement to call findProviders on the resolve context in order to find matching capabilities. In order for a matching capability to be considered as valid for a dynamic wire it must satisfy the following rules:

1. The capability must use the osgi.wiring.package namespace

2. The wiring must not provide an osgi.wiring.package capability that has the same package name as the matching capability. In other words, the wiring must not already export the package.

3. The wiring must not have a required wire that wires to an osgi.wiring.package capability that has the same package name as the matching capability.

If a matching capability is not considered valid then it is ignored by the resolver.

TODO: A ResolveContext likely can do these checks more efficiently than the resolver. The resolver will have to iterate over each provided package capability as well as required package (and required bundle) to determine if the package is exported or imported. In a framework we will already know that the package is not imported/required because dynamic package resolution is a last resort operation after we have already checked the imports/requires and local class-path. Similarly a framework is not even supposed to do dynamic resolution if a wiring already exports a package. I would prefer we state these as recommended rules in order to get correct resolution behavior. Similar to how the resolver trusts that findCapabilities will only return capabilities that really match a requirement.

At this point in the dynamic resolution process the resolution operation continues on as a normal resolution process. The only difference is that the root resource is the already resolved resource with the wiring that is hosting the dynamic package import requirement. The resources providing the matching capabilities are resolved as in a normal resolution operation.

5.3 Canceling a Resolution Process

The resolution problem is considered an np-complete problem. Depending on the resolution problem set (the set resources being resolved) and the resolution algorithm the resolution process may cause the resolution algorithm to take up large amounts of resources and/or appear to never complete. Given enough time and resources the resolution operation may be able to complete. The resolve context may want to put a limit on the amount of time or resources used during a resolution operation. In order to allow a resolve context to cancel a current resolve process the following method is added:

```
public boolean isCancelled() {
            return false;
}
```

The resolver is obligated to call this method periodically while it is determining a valid solution for resolution. For example, as the resolver is determining the complete set of possible solutions and iterating over them to find one that provides a consistent class space. If this method returns true then the resolver must cancel the current resolution operation and result in a ResolutionException being thrown by the resolver resolve method.

6 Data Transfer Objects

No new DTOs required for this RFC

7 Javadoc

OSGi Javadoc

5/3/16 11:03 AM

Package Summary		Page
org.osgi.service.resolver	Resolver Service Package Version 1.1.	13

Package org.osgi.service.resolver

```
@org.osgi.annotation.versioning.Version(value="1.1")
```

Resolver Service Package Version 1.1.

See:

Description

Interface Summary		Page
HostedCapability	A capability hosted by a resource.	14
Resolver	A resolver service resolves the specified resources in the context supplied by the caller.	21

Class Summary		Page
ResolveContext	A resolve context provides resources, options and constraints to the potential solution of a `resolve` operation.	17

Exception Summary		Page
ResolutionException	Indicates failure to resolve a set of requirements.	15

Package org.osgi.service.resolver Description

Resolver Service Package Version 1.1.

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="[1.1,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.resolver; version="[1.1,1.2)"
```

Interface HostedCapability

org.osgi.service.resolver

All Superinterfaces:

org.osgi.resource.Capability

```
@org.osgi.annotation.versioning.ProviderType
public interface HostedCapability
extends org.osgi.resource.Capability
```

A capability hosted by a resource.

A HostedCapability is a Capability where the `getResource()` method returns a Resource that hosts this Capability instead of declaring it. This is necessary for cases where the declaring Resource of a Capability does not match the runtime state. For example, this is the case for fragments attached to a host. Most fragment declared capabilities and requirements become hosted by the host resource. Since a fragment can attach to multiple hosts, a single capability can actually be hosted multiple times.

ThreadSafe

Method Summary		Page
org.osgi.resource.Capability	**getDeclaredCapability**() Return the Capability hosted by the Resource.	14
org.osgi.resource.Resource	**getResource**() Return the Resource that hosts this Capability.	14

Methods inherited from interface org.osgi.resource.Capability
equals, getAttributes, getDirectives, getNamespace, hashCode

Method Detail

getResource

```
org.osgi.resource.Resource getResource()
```

Return the Resource that hosts this Capability.

Specified by:

getResource in interface org.osgi.resource.Capability

Returns:

The Resource that hosts this Capability.

---

getDeclaredCapability

```
org.osgi.resource.Capability getDeclaredCapability()
```

Return the Capability hosted by the Resource.

Returns:

The Capability hosted by the Resource.

Class ResolutionException

org.osgi.service.resolver

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

All Implemented Interfaces:

Serializable

```
public class ResolutionException
extends Exception
```

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 extend this class to provide extra state information about the reason for the resolution failure.

Constructor Summary	Page
ResolutionException(String message) Create a ResolutionException with the specified message.	16
ResolutionException(String message, Throwable cause, Collection<org.osgi.resource.Requirement> unresolvedRequirements) Create a ResolutionException with the specified message, cause and unresolved requirements.	15
ResolutionException(Throwable cause) Create a ResolutionException with the specified cause.	16

Method Summary		Page
Collection<org.osgi.resource.Requirement>	**getUnresolvedRequirements**() Return the unresolved requirements, if any, for this exception.	16

Constructor Detail

ResolutionException

```
public ResolutionException(String message,
                           Throwable cause,
                           Collection<org.osgi.resource.Requirement> unresolvedRequirements)
```

Create a ResolutionException with the specified message, cause and unresolved requirements.

Parameters:

message - The message.
cause - The cause of this exception.

`unresolvedRequirements` - The unresolved mandatory requirements from mandatory resources or `null` if no unresolved requirements information is provided.

ResolutionException

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

Create a `ResolutionException` with the specified message.

Parameters:

`message` - The message.

ResolutionException

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

Create a `ResolutionException` with the specified cause.

Parameters:

`cause` - The cause of this exception.

Method Detail

getUnresolvedRequirements

```
public Collection<org.osgi.resource.Requirement> getUnresolvedRequirements()
```

Return the unresolved requirements, if any, for this exception.

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

Returns:

A collection of the unresolved requirements for this exception. The returned collection may be empty if no unresolved requirements information is available.

Class ResolveContext

org.osgi.service.resolver

```
java.lang.Object
  └ org.osgi.service.resolver.ResolveContext
```

```
@org.osgi.annotation.versioning.ConsumerType
abstract public class ResolveContext
extends Object
```

A resolve context provides resources, options and constraints to the potential solution of a `resolve` operation.

Resolve Contexts:

- Specify the mandatory and optional resources to resolve. The mandatory and optional resources must be consistent and correct. For example, they must not violate the singleton policy of the implementer.
- Provide `capabilities` that the Resolver can use to satisfy `requirements` via the `findProviders(Requirement)` method
- Constrain solutions via the `getWirings()` method. A wiring consists of a map of existing `resources` to `wiring`.
- Filter requirements that are part of a resolve operation via the `isEffective(Requirement)`.

A resolver may call the methods on the resolve context any number of times during a resolve operation using any thread. Implementors should ensure that this class is properly thread safe.

Except for `insertHostedCapability(List, HostedCapability)` and `isCancelled()`, the resolve context methods must be *idempotent*. This means that resources must have constant capabilities and requirements and the resolve context must return a consistent set of capabilities, wires and effective requirements.

ThreadSafe

Constructor Summary	Page
ResolveContext()	18

Method Summary		Page
abstract List<org.osgi.resource.Capability>	**findProviders**(org.osgi.resource.Requirement requirement) Find Capabilities that match the given Requirement.	18
Collection<org.osgi.resource.Resource>	**findRelatedResources**(org.osgi.resource.Resource resource) Find resources that are related to the given resource.	20
Collection<org.osgi.resource.Resource>	**getMandatoryResources**() Return the resources that must be resolved for this resolve context.	18
Collection<org.osgi.resource.Resource>	**getOptionalResources**() Return the resources that the resolver should attempt to resolve for this resolve context.	18
abstract Map<org.osgi.resource.Resource,org.osgi.resource.Wiring>	**getWirings**() Returns the wirings for existing resolved resources.	19

abstract int	**insertHostedCapability**(List<org.osgi.resource.Capability> capabilities, HostedCapability hostedCapability) Add a HostedCapability to the list of capabilities returned from findProviders(Requirement).	19
boolean	**isCancelled**() Returns true if the currently running resolve operation was cancelled before it completed normally.	20
abstract boolean	**isEffective**(org.osgi.resource.Requirement requirement) Test if a given requirement should be wired in the resolve operation.	19

Constructor Detail

ResolveContext

```
public ResolveContext()
```

Method Detail

getMandatoryResources

```
public Collection<org.osgi.resource.Resource> getMandatoryResources()
```

Return the resources that must be resolved for this resolve context.

The default implementation returns an empty collection.

Returns:
A collection of the resources that must be resolved for this resolve context. May be empty if there are no mandatory resources. The returned collection may be unmodifiable.

---

getOptionalResources

```
public Collection<org.osgi.resource.Resource> getOptionalResources()
```

Return the resources that the resolver should attempt to resolve for this resolve context. Inability to resolve one of the specified resources will not result in a resolution exception.

The default implementation returns an empty collection.

Returns:
A collection of the resources that the resolver should attempt to resolve for this resolve context. May be empty if there are no optional resources. The returned collection may be unmodifiable.

---

findProviders

```
public abstract List<org.osgi.resource.Capability> findProviders(org.osgi.resource.Requirement requirement)
```

Find Capabilities that match the given Requirement.

The returned list contains org.osgi.resource.Capability objects where the Resource must be the declared Resource of the Capability. The Resolver can then add additional HostedCapability objects with the insertHostedCapability(List, HostedCapability) method when it, for example, attaches

fragments. Those HostedCapability objects will then use the host's Resource which likely differs from the declared Resource of the corresponding Capability.

The returned list is in priority order such that the Capabilities with a lower index have a preference over those with a higher index. The resolver must use the insertHostedCapability(List, HostedCapability) method to add additional Capabilities to maintain priority order. In general, this is necessary when the Resolver uses Capabilities declared in a Resource but that must originate from an attached host.

Each returned Capability must match the given Requirement. This means that the filter in the Requirement must match as well as any namespace specific directives. For example, the mandatory attributes for the osgi.wiring.package namespace.

Parameters:
requirement - The requirement that a resolver is attempting to satisfy. Must not be null.
Returns:
A list of org.osgi.resource.Capability objects that match the specified requirement.

---

insertHostedCapability

```
public abstract int insertHostedCapability(List<org.osgi.resource.Capability> capabilities,
                                           HostedCapability hostedCapability)
```

Add a HostedCapability to the list of capabilities returned from findProviders(Requirement).

This method is used by the Resolver to add Capabilities that are hosted by another Resource to the list of Capabilities returned from findProviders(Requirement). This function is necessary to allow fragments to attach to hosts, thereby changing the origin of a Capability. This method must insert the specified HostedCapability in a place that makes the list maintain the preference order. It must return the index in the list of the inserted HostedCapability.

Parameters:
capabilities - The list returned from findProviders(Requirement). Must not be null.
hostedCapability - The HostedCapability to insert in the specified list. Must not be null.
Returns:
The index in the list of the inserted HostedCapability.

---

isEffective

```
public abstract boolean isEffective(org.osgi.resource.Requirement requirement)
```

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

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

Parameters:
requirement - The Requirement to test. Must not be null.
Returns:
true if the requirement should be considered as part of the resolve operation.

---

getWirings

```
public abstract Map<org.osgi.resource.Resource,org.osgi.resource.Wiring> getWirings()
```

Returns the wirings for existing resolved resources.

For example, if this resolve context is for an OSGi framework, then the result would contain all the currently resolved bundles with each bundle's current wiring.

Multiple calls to this method for this resolve context must return the same result.

Returns:
The wirings for existing resolved resources. The returned map is unmodifiable.

---

findRelatedResources

```
public Collection<org.osgi.resource.Resource> findRelatedResources(org.osgi.resource.Resource resource)
```

Find resources that are related to the given resource.

The resolver attempts to resolve related resources during the current resolve operation. Failing to resolve one of the related resources will not result in a resolution exception unless the related resource is also a `mandatory` resource.

The resolve context is asked to return related resources for each resource that is pulled into a resolve operation. For example, a fragment can be considered a related resource for a host bundle. When a host is being resolved the resolve context will be asked if any related resources should be added to the resolve operation. The resolve context may decide that the potential fragments of the host should be resolved along with the host.

Parameters:
`resource` - The Resource that a resolver is attempting to find related resources for. Must not be `null`.
Returns:
A collection of the resources that the resolver should attempt to resolve for this resolve context. May be empty if there are no related resources. The returned collection may be unmodifiable.
Since:
1.1

---

isCancelled

```
public boolean isCancelled()
```

Returns `true` if the currently running resolve operation was cancelled before it completed normally.

The resolver is obligated to call this method periodically while it is determining a valid solution for resolution. For example, as the resolver is iterating over the set of possible solutions trying to find a consistent class space. If this method returns `true` then the resolver must cancel the current resolution operation and throw a `ResolutionException`.

This method allows a resolve context to cancel a long running resolution operation that appears to be running endlessly or at risk of running out of resources. The resolve context may then decide to give up on resolution or attempt to try resolving again with a smaller set of resources which may allow the resolution operation to complete normally.

Returns:
`true` if the currently running resolve operation was cancelled before it completed normally

Interface Resolver

org.osgi.service.resolver

```
@org.osgi.annotation.versioning.ProviderType
public interface Resolver
```

A resolver service resolves the specified resources in the context supplied by the caller.

ThreadSafe

Method Summary		Page
Map<org.osgi.resource.Resource,List<org.osgi.resource.Wire>>	**dynamicResolve**(ResolveContext context, org.osgi.resource.Wiring wiring, org.osgi.resource.Requirement requirement) Resolves a given requirement dynamically for the given wiring using the given resolve context and return any new resources and wires to the caller.	22
Map<org.osgi.resource.Resource,List<org.osgi.resource.Wire>>	**resolve**(ResolveContext context) Resolve the specified resolve context and return any new resources and wires to the caller.	21

Method Detail

resolve

```
Map<org.osgi.resource.Resource,List<org.osgi.resource.Wire>> resolve(ResolveContext context)
                                                              throws ResolutionException
```

Resolve the specified resolve context 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 throwing 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 that resource.

The resolve method returns the delta between the start state defined by ResolveContext.getWirings() and the end resolved state. That is, only new resources and wires are included.

The behavior of the resolver is not defined if the specified resolve context supplies inconsistent information.

Parameters:

`context` - The resolve context for the resolve operation. Must not be `null`.

Returns:

The new resources and wires required to satisfy the specified resolve context. The returned map is the property of the caller and can be modified by the caller.

Throws:

ResolutionException - If the resolution cannot be satisfied.

---

dynamicResolve

```
Map<org.osgi.resource.Resource,List<org.osgi.resource.Wire>> dynamicResolve(ResolveContext context,
                                                                            org.osgi.resource.Wiring wiring,
                                                                            org.osgi.resource.Requirement requirement)
                                                                     throws ResolutionException
```

Resolves a given requirement dynamically for the given wiring using the given resolve context and return any new resources and wires to the caller.

The requirement must be a `requirement` of the wiring and must use the `package` namespace.

The resolve context is not asked for `mandatory` resources or for `optional` resources. The resolve context is asked to `find providers` for the given requirement. The matching `package` capabilities returned by the resolve context must not have a `osgi.wiring.package` attribute equal to a `package` capability already `wired to` by the wiring or equal a `package` capability `provided` by the wiring. The resolve context may be requested to `find providers` for other requirements in order to resolve the resources that provide the matching capabilities to the given requirement.

If the requirement `cardinality` is not `multiple` then no new wire must be created if the `wires` of the wiring already contain a wire that uses the `requirement`

This operation may resolve additional resources in order to resolve the dynamic requirement. The returned map will contain entries for each resource that got resolved in addition to the specified wiring `resource`. The wire list for the wiring resource will only contain wires which are for the dynamic requirement.

Parameters:

`context` - The resolve context for the resolve operation. Must not be `null`.
`wiring` - The wiring with the dynamic `requirement`. Must not be `null`.
`requirement` - The dynamic requirement. Must not be `null`.

Returns:

The new resources and wires required to satisfy the specified dynamic requirement. The returned map is the property of the caller and can be modified by the caller. If no new wires were created the an empty map is returned.

Throws:

`ResolutionException`

8 Considered Alternatives

9 Security Considerations

No security concerns for this RFC

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

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	
Company	
Address	
Voice	
e-mail	

10.3 Acronyms and Abbreviations

10.4 End of Document