



RFC 206 - Asynchronous Services

Draft

40 Pages

Abstract

10 point Arial Centered.

The OSGi service registry is used by bundles to collaborate using loosely coupled services, registered with one or more public interfaces that can be called. OSGi services are, like most Java objects, normally designed to be called synchronously. There are, however, often significant advantages that can be realized by clients when they execute one or more parts of their operation asynchronously. This RFC provides a generic mechanism that allows existing OSGi services with a synchronous API to be invoked asynchronously without requiring them to be modified.

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 Document Information.....	2
0.1 License.....	2
0.2 Trademarks.....	3
0.3 Feedback.....	3
0.4 Table of Contents.....	3
0.5 Terminology and Document Conventions.....	4
0.6 Revision History.....	4
1 Introduction.....	5
2 Application Domain.....	5
2.1 Asynchronous programming models.....	6
2.2 Mixture of programming models.....	6
2.3 Terminology + Abbreviations.....	6
3 Problem Description.....	7
3.1 Asynchronous Services.....	7
4 Requirements.....	7
5 Technical Solution.....	8
5.1 The Async Service.....	8
5.2 Async Proxies.....	8
5.2.1 Class Proxying.....	9
5.2.2 Async Proxy return types.....	9

5.2.3 Thread safety.....	9
5.3 Building simple tasks.....	9
5.3.1 Establishing a context.....	10
5.3.2 Completing tasks.....	10
5.3.3 Registering Callbacks.....	11
5.3.4 Establishing context for Void methods.....	12
5.4 Building more complex tasks.....	14
5.4.1 Running in Parallel.....	14
5.4.2 Running sequentially.....	14
5.4.3 Waiting for previous invocations.....	15
5.4.4 Receiving multiple promises.....	15
5.4.5 Registering aggregate callbacks.....	16
5.5 Execution Failures.....	17
5.6 Delegating to asynchronous implementations.....	17
6 Data Transfer Objects.....	18
7 Javadoc.....	18
8 Considered Alternatives.....	35
9 Security Considerations.....	35
10 Document Support.....	36
10.1 References.....	36
10.2 Author's Address.....	36
10.3 Acronyms and Abbreviations.....	36
10.4 End of Document.....	36

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 10.1.

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	08/11/13	Tim Ward – Initial version of RFC
0.2	15/11/13	Virtual F2F comments – update numbering of requirements to match Final RFP version; use DS in example code to demonstrate good practice.

Revision	Date	Comments
0.3	04/12/13	EEG Call comments. Rename AlwaysCallback to CompletionCallback (including associated methods). Rename Async Proxy to Async mediator, Clarify the mechanism for generating mediators for concrete types
0.4	09/01/14	Introduce the Promise API. Remove async task builder in favour of promises
<u>0.5</u>	<u>03/02/14</u>	<u>Post Austin F2F – some minor changes for the Promises API, make all Async services support concrete type mediation.</u>

1 Introduction

OSGi Bundles collaborate using loosely coupled services registered in the OSGi service registry. This is a powerful and flexible model, and allows for the dynamic replacement of services at runtime. OSGi services are therefore a very common interaction pattern within OSGi.

As with most Java APIs and Objects, OSGi services are primarily synchronous in operation. This has several benefits; synchronous APIs are typically easier to write and to use than asynchronous ones; synchronous APIs provide immediate feedback; synchronous implementations typically have a less complex threading model.

Asynchronous APIs, however, have different advantages. Asynchronous APIs can reduce bottlenecks by encouraging more effective use of parallelism, improving the responsiveness of the application. This intent of this RFC is to allow clients to get the benefits of asynchronous invocation, even when the Service API has been written in a synchronous way.

2 Application Domain

This section explores various aspects of adding support for asynchronous execution. Asynchronous execution typically is achieved via the introduction of a queuing mechanism for “tasks” which are pulled in and executed by one or more Threads. In the case of remote invocations, the task queue is often on the remote machine, allowing the request to be sent and executed without occupying a local Thread. These mechanisms are often also used to handle events, for example the OSGi Event Admin Service provides an asynchronous communication model.

Synchronous invocations are typically easier to program, but once a client makes a request, either local or remote, the client is blocked waiting for execution to complete and return control to the client. While asynchronous execution may be more complex to program, it offers many benefits and advantages.

For example, synchronous remote invocations depend on the availability of the network during request execution. If a client or server fails during the execution of a request, the request typically has to be resubmitted. This may not be a problem for some applications, where it's easy to re-create the request input. But for other applications, such as an ATM, gas pump, or electronic funds transfer, it may not be easy to recapture the input data and create another request message, and asynchronous protocols meet the requirement better. Even when it is possible to recreate a request message, it is not always easy to know at which point the server failed – i.e. whether or not an update was performed as a result of executing the request, and if so, whether performing the update a second time might cause data inconsistency. And in this case asynchronous protocols can also offer some advantages.

Synchronous invocations operate on a first-come, first-served scheduling mechanism (i.e. the computer has to process requests as they are made by the caller). This means that it's not easy to treat some invocations with higher priority than others, although this is a common application requirement (for example, a bank wants to process the outstanding \$1M deposits ahead of the \$10 deposits near the end of the banking day). As they have work queues, asynchronous processing engines can process work in an arbitrary order if they choose.

2.1 Asynchronous programming models

A variety of asynchronous programming models and frameworks are successfully used in enterprise applications today, including ExecutorServices, Async EJBs, Async Servlets, Node.js, store-and-forward, pub-sub, and broadcast/multicast to name a few. These programming models assume that a task is visible to a program using one or more asynchronous submission mechanisms (for example, JMS) and that the program is responsible for explicitly creating or retrieving a response using the API and then may act upon it in a way that is visible to another program using the same API.

For example, a store and forward system has one program submitting a message to a queue using a SEND or SUBMIT command, and another program retrieving the message from a queue using a RECEIVE or DEQUEUE command from the asynchronous programming model API. The sending program is responsible for packing, or serializing, the message, and the receiving program is responsible for unpacking, or de-serializing the message. (Some APIs define a wire format while others do not.)

In each case, management utilities are required to configure the capabilities of the asynchronous implementation being used so that they are able to reject work when overloaded, make best use of the resources available, and to identify, report and resolve any errors that may occur.

2.2 Mixture of programming models

Many enterprise applications require both synchronous and asynchronous execution models for different types of IT functions. For a reserved ticket purchase, for example, it may be necessary to synchronize the database update with the reply to the user to indicate the ticket was purchased, since only one person can have a given seat. For a book purchase, however, it may be sufficient to reply to the user that the order was received, and that it would be fulfilled later. Some of the fulfillment operations for a book order might also use synchronous communications, for example to debit inventory while packing the order for shipment.

2.3 Terminology + Abbreviations

Client – Application code that wishes to call one or more OSGi services asynchronously

Target Service – A service that is to be called asynchronously by the client

Async Service – The OSGi service representing the Asynchronous Services implementation. Used by the client.

Async Mediator – A mediator object representing the target service, created by the Async Service

Success Callback – A callback made when an asynchronous invocation exits with a normal return value

Failure Callback – A callback made when an asynchronous invocation exits by throwing an exception

Completion Callback – A callback that is made when an asynchronous invocation exits, regardless of how it exits.

Asynchronous Invocation – A single method call that is to be executed without blocking the requesting thread.

Asynchronous Task – An aggregate of one or more asynchronous invocations. The invocations that make up a task may run in parallel, or sequentially, or a mixture of both.

3 Problem Description

The current OSGi programming model for communications among components and bundles is based on the OSGi service interface, which implies a synchronous semantic (i.e. the client invokes on the interface and waits for the reply), and language objects as parameters. These characteristics are typical of local invocations and distributed RPC and meet many requirements, but we want to extend these capabilities to support asynchronous invocation.

3.1 Asynchronous Services

We propose that the EEG evaluate options for specifying Asynchronous invocation of services – specifically the ability for a client to issue an invocation on a service interface without waiting for completion, and relying on a later notification or polling to check completion and retrieve results. For illustration, a low-level equivalent of such a framework is provided in J2SE by the Future interface. Other technologies (such as CORBA) provide asynchronous 'one-way' support on their remote interfaces. There are significant design considerations involved in selecting whether this may be defined within the “OSGi Services” architecture, and/or “Blueprint”, and/or Remote Services; and how a particular choice of solution relates to all three architectures.

4 Requirements

AS01 – The solution **MUST** provide a standard client-side API for making asynchronous invocations on existing, synchronous, OSGi services, where the invocation returns quickly and a return value can be obtained later.

AS02 – The solution **MUST** allow transparent delegation to services that are already implemented in an asynchronous fashion, therefore servicing the asynchronous requests through their own implementations.

AS03 – The solution **MAY** provide a synchronous client-side API to services which are implemented in an asynchronous fashion.

AS04 – The solution **MUST** allow for one-way (fire and forget) asynchronous services.

AS05 – The solution **MUST** support Promises, where invocations can be made that later return a value

AS06 – The solution **SHOULD** support callbacks when asynchronous executions complete, both successfully and unsuccessfully

AS07 – The solution **MUST** be applicable to both local OSGi Services as well as Remote OSGi Services.

AS08 – The solution **MUST** be fully backwards compatible with existing OSGi Service and Service Registry usage.

AS09 – The solution **SHOULD** be sympathetic to Java 8's lambda support, meaning callbacks should follow the Single Abstract Method principle where possible.

AS10 – The solution **MUST** define a mechanism that allows service providers to advertise an asynchronous mode of operation if they support it.

5 Technical Solution

In order for a client to make asynchronous invocations on a service there are several necessary steps. First it is necessary to identify the service to be invoked, which we shall refer to as the target service. In the absence of any further support the client would then need to create a Runnable or Callable that invoked the target service, and then execute this using an Executor or by starting a new Thread.

Rather than having each client managing its own asynchronous Executor an Async Service can manage the execution of the asynchronous invocation. This requires the Async service to track the invocations made on the target service and to asynchronously service them. To support requirements AS05 and AS06 the Async service also needs to provide a mechanism to register callbacks, and to return a Promise.

This requirement is similar to the requirements that mocking frameworks such as Mockito have. They track invocations on proxy objects so that the invocations can later be checked, or so that specific invocations can be configured to return particular values. The Async Service uses a similar pattern, where invocations on a mediator are used to register the asynchronous executions that should occur.

5.1 OSGi Promises

One of the fundamental pieces of an asynchronous programming model is the mechanism by which clients retrieve the result of the asynchronous work. Since Java 5 there has been a `java.util.concurrent.Future` interface available in the core Java runtime, which means that it is the de-facto Promise API in Java. Futures have some limitations however, in that they have no mechanism for registering callbacks. This shortcoming will be addressed in Java 8 with the introduction of `java.util.concurrent.CompletableFuture`, however this is also unsuitable for use in an OSGi specification. OSGi therefore requires its own Promise API to support Asynchronous Services. As the API is generally applicable to asynchronous programming it is anticipated that it may be used more widely. As such it should be able to be used independently of Asynchronous Services. It would also be advantageous for OSGi Promises to be usable outside of an OSGi framework, therefore the Promises API should avoid having any dependency on other OSGi APIs, such as the core framework package.

5.1.1 The Promise

The primary interface for OSGi Promises is the `org.osgi.util.promise.Promise<T>`. There are several important things that a Promise must be able to achieve:

1. It must be possible to determine whether an asynchronous execution has completed, and to get the result of the asynchronous execution, and to retrieve the failure if the execution completed exceptionally. This can be achieved using the `isDone()` `getValue()` and `getError()` methods:
 - `public boolean isDone();` is used to determine whether a Promise has resolved. If the promise has resolved then this method returns true, otherwise it returns false. This method should not block, If the method returns true then the Promise should not block in subsequent calls to `get()` and `getError()`
 - `public T getValue();` is used to retrieve the result of the asynchronous task. It blocks until the promise is resolved, or until the calling thread is interrupted. If the thread is interrupted then `InterruptedException` is thrown. If the promise resolves successfully then the result is returned. If the promise resolves with a failure then the failure is wrapped in an `InvocationTargetException` and thrown to the client.
 - `public Throwable getError();` is used to retrieve the result of a failed execution. It blocks until the promise is resolved, or until the calling thread is interrupted. If the thread is interrupted then `InterruptedException` is thrown. If the promise resolves successfully then `null` is returned. If the promise resolves with a failure then the exception that was thrown is returned to the client.
2. It must be possible to register callbacks with the promise that are called after the promise is resolved. These callbacks need to be able to access the result of the execution. It must also be possible to delay the calling of the callbacks, even if the Promise is already resolved. This is achieved through the use of the two `then(...)` methods and, `onresolve()`, ~~`hold()`~~, ~~and `launch()`~~:
 - `public Promise<R> then(Success<R, ? super T> success)` and `public Promise<R> then(Success<R, ? super T> success, Failure<? Super T> failure)` are the mechanism by which clients can register callbacks. Success callbacks are called after a promise has resolved with a value, Failure callbacks are called after a promise with a failure. In both cases there is a “happens before” relationship, which means that the promise will always return true for calls to `isDone()` and will not block when retrieving a value. The

thread that is used to execute the callbacks is undefined and implementation dependent. If there are multiple callbacks defined for a single promise then they may be called concurrently on separate threads, or sequentially on a single thread. Client calls to `then(...)` return another Promise. This is known as “chaining”, and the chained promise resolves in one of four ways:

1. The parent promise resolves with a failure. In this case the chained promise resolves immediately with the same failure.
 2. The parent promise resolves successfully, but the Success callback throws an exception. In this case the chained promise resolves immediately with the exception thrown by the callback.
 3. The parent promise resolves successfully, and the Success callback is `null`, or returns `null`. In this case the chained promise resolves immediately with a successful value of `null`.
 4. The parent promise resolves successfully, and the Success callback returns a Promise. In this case the chained promise resolves when the returned promise resolves. The resolution of both promises must be the same value.
- `public void onresolve(Runnable toCall);` registers a runnable to be called when the promise resolves, either successfully or unsuccessfully. This has the same “happens before” behaviour as the callbacks from the `then()` methods, and also runs on an undefined thread. It does not make sense for the client to pass `null` to this method. If this situation occurs then the Promise should throw a `NullPointerException`
 - ~~`public Promise<T> hold();` is used to prevent the promise from executing its callbacks when it resolves. Instead the callbacks will be called when the promise has resolved and the `launch()` method has been called. The value returned by `hold()` is this, meaning that it is suitable for fluent style usage.~~
3. It must be possible to cancel Promises that are no longer meaningful. Sometimes in asynchronous systems the originator of a promise no longer needs the result of an asynchronous execution. In these cases it can be very useful to cancel a Promise such that it resolves immediately. This can be achieved and queried through the `isCancelled()` and `cancel()` methods:
 - `public boolean isCancelled();` is used to determine whether a Promise was cancelled. If the promise was resolved before it was cancelled, or no calls to `cancel()` have been made, then this method returns false, otherwise it returns true. This method should not block. If this method returns true then calls to `isDone()` must also return true.
 - `public boolean cancel();` is used to cancel a promise. If the promise was resolved normally before the call to `cancel` then this method is a no-op, and must return false. If the promise has already been cancelled then this method must return true. If the promise not yet been resolved then it must be resolved immediately with a `CancelledPromiseException`, the promise must be marked as cancelled, and true must be returned. When a promise is cancelled the backing asynchronous task may also be cancelled, but this is implementation dependent.

5.1.2 The Deferred class

Promises may be implemented in many different ways, however there is significant value in having a simple implementation available as part of the promise API. The `org.osgi.util.promise.Deferred` class provides a simple implementation that allows implementations to easily provide basic promise behaviour.

The `Deferred` object is instantiated by the code that wishes to perform the asynchronous work. It then uses the `getPromise()` method to obtain a promise to return to the client. When the asynchronous work has completed the asynchronous code then calls either the `resolve()` method or the `fail()` method on the deferred object.

~~In some cases it can be useful to create a promise representing a value that has already been calculated (for example in a Success callback). The `Deferred` class offers a static method `newResolvedPromise(T object)` to simplify this case.~~

5.1.3 The Promises utility class

There are some common usage patterns involving promises that can benefit from helper utility methods. These methods are defined as static methods of the `org.osgi.util.promise.Promises` class.

Parallel promises

One common use case is that a client wishes to be notified when a group of promises running in parallel are all complete. This can be achieved using the `Promise<Object[]> parallel(Promise<?>... promises)` or `Promise<T[]> parallel(T[] array, Promise<? Extends T>... promises)` method. These methods take a number of promises, and return a promise that resolves once all of the promises passed to the method have resolved. The value of the aggregate promise is an array containing all of the values returned by the promises passed in, in the same order. If any of the promises fail then the aggregate promise will resolve with a `Promises.ParallelException`. This exception provides access to the success and failure values returned by the underlying promises.

Direct promises

~~In some cases it can be useful to create a promise representing a value that has already been calculated (for example in a Success callback). The `Promises` class offers a static method `newResolvedPromise(T object)` to simplify this case.~~

Shareable promises

~~Promises are shareable values that are thread safe, however in some cases it is not appropriate to allow third-party code the ability to cancel your unresolved promise. To help in this situation the `Promises` class offers a static method `shareableCopyOf(Promise<T> promise)` which returns a `Promise` that can be safely passed to other code. The returned promise will resolve when the provided promise resolves, and will resolve in the same way, however calls to `cancel()` will not cancel the original promise.~~

5.2 The Async Service

The Async service is the primary interaction point between a client and the Async Services implementation. An Async Services implementation must expose a service implementing the `org.osgi.service.async.Async` interface.

Clients obtain an instance of the Async Service using the normal OSGi service registry mechanism, either directly using the OSGi framework API, or using dependency injection.

Implementations of the Async service must be thread safe. They should be safe to use simultaneously across multiple clients and from multiple threads within the same client.

5.3 Async Mediators

When a client has chosen a target service, it can use the Async service to make an asynchronous invocation. The first step is to use the Async Service to create a mediator for the real service.

```
<T> T mediate(ServiceReference<T> ref);
```

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.mediate(ref);
    ...
}
```

5.3.1 Generating a Mediator

When creating the Async Mediator object the Async Service should attempt to load all of the classes listed in the objectClass property of the service reference using the client bundle. Any ClassNotFoundException thrown when attempting to load these classes should be ignored. The loaded classes should then be divided into Java interfaces and concrete classes.

The async service must then generate a mediator object. If the service only advertises Java interfaces in its objectClass property then the mediator object must implement ~~that implements~~ all of the Java interfaces that could be loaded by the client bundle. The mediator class must be defined using the client bundle's classloader. This can easily be achieved using the java.lang.reflect.Proxy class.

5.3.2 Generating a Mediator for Concrete Classes

If a service is registered advertising one or more concrete class types then generating a mediator requires more complex handling. ~~If an Async service implementation supports generating mediator objects for concrete classes then it should register a service property with the name org.osgi.service.async.mediate=classes. If concrete classes are supported then~~ In this case the mediator object created by the Async service should also inherit from the most specialised concrete type listed in the target service's objectclass property that can be loaded using the client bundle's ClassLoader. There are three reasons why the Async service may not be able to mediate a class type:

1. The most specialised type is final
2. The most specialised type has no zero-argument constructor
3. One or more public methods present in the type hierarchy (other than those declared by java.lang.Object) are final.

If any of these rules are violated then the Async service should fall back to creating an interface-only mediator.

5.3.3 Async Mediator return types

When invoked the Async mediator should return rapidly (i.e. it should not perform blocking operations). The client should not attempt to interpret the returned value. The value may be null (or null-like in the case of primitives) or contain implementation specific information.

5.3.4 Thread safety

Whilst the Async Service itself must be thread safe, async mediator objects may not be. Clients should avoid sharing async mediator objects between threads if they wish to be portable between implementations.

5.3.5 Mediating other objects

The Async service also allows mediators to be created for arbitrary objects (rather than for Service References) using the method:

```
<T> T mediate(T object);
```

In this case ~~the generated object should implement all of the interfaces implemented by the object, and if concrete class proxying is supported then it should inherit from the type of the supplied object. The mediator generation behaviour is otherwise the same as for a mediator mediating created using a service reference which lists the entire Java class and interface hierarchy for the supplied object.~~

5.4 Building simple tasks

Once a mediator has been created it can be used, in conjunction with the Async service, to run an asynchronous task.

5.4.1 Running an asynchronous task

To begin an asynchronous task the client invokes a method on the asynchronous mediator. The client then passes the result of that invocation to the Async service to begin the asynchronous invocation. The execution has an associated type, which is the return type of the asynchronous invocation (or the associated wrapper type for primitives and void).

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.mediate(ref);
    Promise<Boolean> promise = asyncService
        .call(mediator.contains("badEntry"));
    ...
}
```

At this point the Asynchronous task is running, and may have already completed. It can be interacted with using the Promise returned by the call method.

Once a context has been established, clients can continue to start other asynchronous tasks, or to chain Promises with the one returned by the async service.

5.4.2 Establishing context for Void methods

In Java void methods have no return value, and therefore cannot return anything. This means that void methods cannot establish context in the same way that other methods do. Void methods therefore need to be declared in a different way, the client calls the void method as a separate statement and then either:

- passes **null** to the async service.

or

- calls the no-args `call()` method

Example – Out of Line expression:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.createAsyncMediator(listRef);
    mediator.clear();
    asyncService.call((Void) null);
}
```

Promises for void methods are resolved with null when they are successful.

5.5 Building more complex tasks

Asynchronous tasks may consist of multiple distinct asynchronous invocations. The promise api allows tasks to be chained, or to wait for a set of tasks to have completed.

5.5.1

5.6 Execution Failures

There are a variety of reasons that Asynchronous invocations may fail. In any of these cases the asynchronous invocation should resolve with an `org.osgi.service.framework.ServiceException`, with a type of ASYNC (see OSGi core R6). This exception should be passed to any failure callbacks as normal

- If the client bundle's bundle context becomes invalid before looking up the target service. This does not apply if the mediated object was not created with a `ServiceReference`.
- If the target service becomes unavailable before making the asynchronous invocation, or returns null on lookup. This does not apply if the mediated object was not created with a `ServiceReference`.
- If the Async service is unable to accept new work, for example it is in the process of being shut down.
- If the target service is unable to be invoked with the supplied arguments (this indicates a missing uses constraint)

If the target service is successfully invoked, but the method call throws an exception, then this should be used to resolve the promise without wrapping it.

5.7 Delegating to asynchronous implementations

Some service APIs are already asynchronous in operation, and others are partly asynchronous, in that some methods run asynchronously and others do not. There are also services which have a synchronous API, but could run asynchronously because they are a proxy to another service. A good example of this kind of service is a remote service. Remote services are local views of a remote endpoint, and depending upon the implementation of the endpoint it may be possible to make the remote call asynchronously, optimizing the thread usage of any local asynchronous call.

Services that already have some level of asynchronous support can advertise this by implementing the `org.osgi.service.async.spi.AsyncDelegate` interface. This can be used by the asynchronous services implementation, or by the client directly, to indicate that a call made on the service should be processed asynchronously. The `AsyncDelegate` can be used as follows:

1. Cast the object to `AsyncDelegate`
2. Invoke the `async(Method, Object[])` method, holding on to the `Promise` returned by the invocation
 - If the returned promise is null then this service does not support asynchronous invocation of that method, and should be treated as a normal synchronous service.
 - If the async call throws an exception then the promise should immediately be resolved with that exception.
3. The `AsyncDelegate` will begin asynchronously executing the method, when the returned promise resolves it should be used to resolve the promise returned to the client by the async service.

6 Data Transfer Objects

It is unclear whether Asynchronous Services would benefit from DTOs

7 Javadoc

OSGi Javadoc

2/3/14 11:25 AM

Package Summary		Page
org.osgi.service.async		17
org.osgi.util.promise		22

Package **org.osgi.service.async**

Interface Summary		Page
<i>Async</i>	The Asynchronous Execution Service.	18
<i>AsyncDelegate</i>	This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently.	21

Interface Async

org.osgi.service.async

public interface **Async**

The Asynchronous Execution Service. This can be used to make asynchronous invocations on OSGi services and objects through the use of a mediator object.

Typical usage:

```
Async async = ctx.getService(asyncRef);

ServiceReference<MyService> ref = ctx.getServiceReference(MyService.class);

MyService asyncMediator = async.mediate(ref);

Promise<BigInteger> result = async.call(asyncMediator.getSumOverAllValues());
```

The [Promise](#) API allows callbacks to be made when asynchronous tasks complete, and can be used to chain Promises. Multiple asynchronous tasks can be started concurrently, and will run in parallel if the Async service has threads available.

Method Summary		Page
Promise<?>	call () This method launches the last method call registered by a mediated object as an asynchronous task.	19
Promise<R>	call (R r) This method launches the last method call registered by a mediated object as an asynchronous task.	19
T	mediate (<any> target) Create a mediator for the given service.	19
T	mediate (T target) Create a mediator for the given object.	18

Method Detail

mediate

T **mediate**(T target)

Create a mediator for the given object. The mediator is a generated object that registers the method calls made against it. The registered method calls can then be run asynchronously using either the [call\(Object\)](#) or [hold\(Object\)](#) method.

The values returned by method calls made on a mediated object should be ignored.

Normal usage:

```
I i = async.mediate(s);
Promise<String> p = async.call(i.foo());
```

Specified by:

[mediate](#) in interface [Async](#)

Parameters:

target - The service object

Returns:

A mediator for the service object

mediate

T **mediate**(<any> target)

Create a mediator for the given service. The mediator is a generated object that registers the method calls made against it. The registered method calls can then be run asynchronously using either the [call\(Object\)](#) or [hold\(Object\)](#) method.

The values returned by method calls made on a mediated object should be ignored.

This method differs from [mediate\(Object\)](#) in that it can track the availability of the backing service. This is recommended as the preferred option for mediating OSGi services as asynchronous tasks may not start executing until some time after they are requested. Tracking the validity of the `org.osgi.framework.ServiceReference` for the service ensures that these tasks do not proceed with an invalid object.

Normal usage:

```
I i = async.mediate(s);
Promise<String> p = async.call(i.foo());
```

Specified by:

[mediate](#) in interface [Async](#)

Parameters:

target - The service object

Returns:

A mediator for the service object

call

[Promise](#)<R> **call**(R r)

This method launches the last method call registered by a mediated object as an asynchronous task. The result of the task can be obtained using the returned promise

Typically the parameter for this method will be supplied inline like this:

```
I i = async.mediate(s);
Promise<String> p = async.call(i.foo());
```

Parameters:

r - the return value of the mediated call, used for type information

Returns:

a Promise which can be used to retrieve the result of the asynchronous execution

call

[Promise](#)<?> **call**()

This method launches the last method call registered by a mediated object as an asynchronous task. The result of the task can be obtained using the returned promise

Generally it is preferable to use [call\(Object\)](#) like this:

```
I i = async.mediate(s);
Promise<String> p = async.call(i.foo());
```

However this pattern does not work for void methods. Void methods can therefore be handled like this:

```
I i = async.mediate(s);  
i.voidMethod()  
Promise<?> p = async.call();
```

Returns:

a Promise which can be used to retrieve the result of the asynchronous execution

Interface AsyncDelegate

org.osgi.service.async

public interface **AsyncDelegate**

This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently. This may mean that the service has access to its own thread pool, or that it can delegate work to a remote node, or act in some other way to reduce the load on the Asynchronous Services implementation when making an asynchronous call.

Method Summary		Page
Promise <?>	async (Method m, Object[] args) This method can be used by the Async service to optimize Asynchronous execution.	21

Method Detail

async

```
Promise<?> async(Method m,  
                    Object[] args)  
    throws Exception
```

This method can be used by the Async service to optimize Asynchronous execution. When called, the [AsyncDelegate](#) should execute the supplied method using the supplied arguments asynchronously, returning a promise that can be used to access the result. If the method cannot be executed asynchronously by the delegate then it should return `null`.

Parameters:

- `m` - the method that should be asynchronously executed
- `args` - the arguments that should be used to invoke the method

Returns:

A promise representing the asynchronous result, or `null` if this method cannot be asynchronously invoked.

Throws:

Exception

Package org.osgi.util.promise

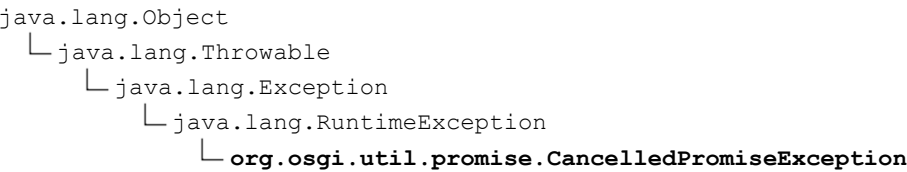
Interface Summary		Page
Failure	A Callback that is made when the promise resolves with a failure.	26
Promise	A Promise represents a future value, it handles the interaction to do asynchronous processing.	28
Success	A Callback that is made when the promise resolves successfully.	34

Class Summary		Page
Deferred	A Deferred is the manager of a Promise .	24
Promises	Utilities for promises.	32

Exception Summary		Page
CancelledPromiseException	This exception is not normally thrown.	23
ParallelException	Exception thrown for Parallel execution.	27

Class CancelledPromiseException

[org.osgi.util.promise](#)



All Implemented Interfaces:
Serializable

```
public class CancelledPromiseException
extends RuntimeException
```

This exception is not normally thrown. Typically it is used to resolve a [Promise](#) in the event that it is cancelled by the client before it is resolved normally.

Constructor Summary	Page
CancelledPromiseException (String message)	23

Constructor Detail

CancelledPromiseException

```
public CancelledPromiseException(String message)
```

Class Deferred

[org.osgi.util.promise](#)

```
java.lang.Object
└─ org.osgi.util.promise.Deferred
```

Type Parameters:
T - The associated value type

```
public class Deferred
extends Object
```

A [Deferred](#) is the manager of a [Promise](#). The Promise is used by the clients but the code that actually does the async work must use the [Deferred](#) to resolve the [Promise](#) with a value or an error.

Constructor Summary	Page
Deferred () Create a new Deferred result that can be resolved using resolve(Object)	24

Method Summary	Page
void fail (Throwable t) Resolve the promise with a failure.	25
Promise<T> getPromise () Return the promise associated with this resolver.	24
void resolve (T value) Resolve this promise with a value.	24

Constructor Detail

Deferred

```
public Deferred()

Create a new Deferred result that can be resolved using resolve\(Object\)
```

Method Detail

getPromise

```
public Promise<T> getPromise()

Return the promise associated with this resolver.
```

resolve

```
public void resolve(T value)

Resolve this promise with a value.

Parameters:
    value - the value to resolve with.
```


fail

```
public void fail(Throwable t)
```

Resolve the promise with a failure.

Parameters:

`t` - the failure

Interface Failure

[org.osgi.util.promise](#)

Type Parameters:

Value - The type of the promise expected by this callback

public interface **Failure**

A Callback that is made when the promise resolves with a failure.

Method Summary		Page
void	fail (Promise <? extends Value > promise) Called when the promise was resolved with an error.	26

Method Detail

fail

void **fail**([Promise](#)<? extends [Value](#)> promise)
throws Exception

Called when the promise was resolved with an error.

Parameters:

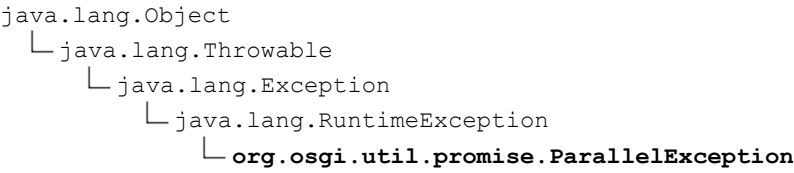
promise - the promise this callback was registered on

Throws:

Exception

Class ParallelException

[org.osgi.util.promise](#)



All Implemented Interfaces:
Serializable

```
public class ParallelException
extends RuntimeException
```

Exception thrown for Parallel execution. Provides access to all the exceptions thrown.

Constructor Summary	Pag e
ParallelException (Object[] values, Throwable[] exceptions)	27

Method Summary	Pag e
Throwable[] getExceptions ()	27
Object[] getValues ()	27

Constructor Detail

ParallelException

```
public ParallelException(Object[] values,
                        Throwable[] exceptions)
```

Method Detail

getExceptions

```
public Throwable[] getExceptions()
```

getValues

```
public Object[] getValues()
```

Interface Promise

[org.osgi.util.promise](#)

Type Parameters:

T - The result type associated with this promise

```
public interface Promise
```

A Promise represents a future value, it handles the interaction to do asynchronous processing. Promises can be created with a [Deferred](#). The Promise is used by the caller of an asynchronous function to get the result or handle the errors. It can either get a callback when the Promise is resolved with a value or an error, or it can be used in chaining. In chaining callbacks are provided that receive the resolved promise, and a new promise is generated that resolves when the callback resolves

Both onresolve and chaining (then) can be repeated any number of times, even long after the value has been resolved.

Example onresolve usage:

```
final Promise<String> foo = foo();
foo.onresolve(new Runnable() {
    public void run() {
        System.out.println(foo.get());
    }
});
```

Example chaining usage:

```
Success doubler = new Success

() {
    public Promise{@code} call(Promise{@code} p) {
        return Deferred.getDirectPromise(p.get()+p.get());
    }
};
final Promise foo = foo().then(doubler).then(doubler);
foo.( new Runnable() { public void run() {
    System.out.println( foo.get() );
} });
```

Method Summary		Pag e
boolean	cancel () Cancel this promise, and resolve it immediately with a CancelledPromiseException .	31
Throwable	getError () Waits if necessary for the asynchronous computation to complete, and then retrieves the error that occurred.	30
T	getValue () Waits if necessary for the computation to complete, and then retrieves its result.	30
boolean	isCancelled () Returns <code>true</code> if this task was cancelled before it resolved, or <code>false</code> if the task has not been cancelled.	31
boolean	isDone () Returns <code>true</code> if this task has completed.	30
void	onResolve (Runnable done) Register a <code>Runnable</code> that will be called when this promise has been resolved with either an error or a value.	30

Promise <R>	then (Success <R, ? super T > success)	29
	Chain promise calls so they are executed in sequence.	
Promise <R>	then (Success <R, ? super T > success, Failure <? super T > failure)	29
	Chain promise calls so they are executed in sequence.	

Method Detail

then

```
Promise<R> then (Success<R, ? super T> success,  
                Failure<? super T> failure)
```

Chain promise calls so they are executed in sequence. This promise will call one of the given callbacks when it gets resolved with either a value or an error.

The return value from this method is a new promise that will get resolved with an error if this promise is resolved with an error, or when the success callback throws an error. Otherwise the returned promise will get resolved with the same object as the promise returned by the success callback.

Note that callbacks may be called immediately on the thread that calls [then\(Success\)](#), immediately on a different thread, or at some point in the future after this promise resolves.

There is a defined "happens before" relationship in promise resolution which means that the promise is always resolved before the callback occurs, i.e. [isDone\(\)](#) will return `true` and [getValue\(\)](#) will not block.

Parameters:

`success` - Callback when this promise is successfully resolved. May be `null` if no success behaviour is required.

`failure` - Callback when this promise is resolved with an error. May be `null` if no failure behaviour is required.

Returns:

A new promise that will get resolved when the success method successfully returns or when there is a failure

then

```
Promise<R> then (Success<R, ? super T> success)
```

Chain promise calls so they are executed in sequence. This promise will call the supplied callback when it gets resolved with a value.

The return value from this method is a new promise that will get resolved with an error if this promise is resolved with an error, or when the success callback throws an error. Otherwise the returned promise will get resolved with the same object as the promise returned by the success callback.

Note that callbacks may be called immediately on the thread that calls [then\(Success\)](#), immediately on a different thread, or at some point in the future after this promise resolves.

There is a defined "happens before" relationship in promise resolution which means that the promise is always resolved before the callback occurs, i.e. [isDone\(\)](#) will return `true` and [getValue\(\)](#) will not block.

Parameters:

`success` - Callback when this promise is successfully resolved. May be `null` if no success behaviour is required.

Returns:

A new promise that will get resolved when the success method successfully returns or when there is an error

onResolve

```
void onResolve(Runnable done)
    throws NullPointerException
```

Register a `Runnable` that will be called when this promise has been resolved with either an error or a value. This method can be used at any time.

Parameters:

`done` - the `Runnable` called when this promise is resolved.

Throws:

`NullPointerException` - if the supplied `Runnable` is `null`

isDone

```
boolean isDone()
```

Returns `true` if this task has completed. Completion may be due to normal termination, an exception, or cancellation -- in all of these cases, this method will return `true`.

Returns:

`true` if this task completed

getValue

```
T getValue()
    throws InvocationTargetException,
           InterruptedException
```

Waits if necessary for the computation to complete, and then retrieves its result. If the promise resolves with an error then calls to this method will throw an `InvocationTargetException` wrapping the `Exception` that resolved the promise

Returns:

the computed result

Throws:

`InvocationTargetException` - if the promise was resolved with an error

`InterruptedException` - if the current thread was interrupted while waiting

getError

```
Throwable getError()
    throws InterruptedException
```

Waits if necessary for the asynchronous computation to complete, and then retrieves the error that occurred. If no error happens then this method will return `null`.

Returns:

the error that occurred when calculating the return value for this promise, or `null` if no error occurred

Throws:

`InterruptedException` - if the current thread was interrupted while waiting

isCancelled

`boolean isCancelled()`

Returns `true` if this task was cancelled before it resolved, or `false` if the task has not been cancelled.

Returns:

`true` if this task completed by being cancelled

cancel

`boolean cancel()`

Cancel this promise, and resolve it immediately with a [CancelledPromiseException](#). This will also resolve any chained promises, and is useful for indicating that the background work is no longer useful or meaningful.

When cancelling a promise there is an inherent race condition with the thread that would normally resolve the promise. The return value from this method is identical to the value that would be returned by calling [isCancelled\(\)](#) immediately after calling [cancel\(\)](#) on the promise. If this method returns false then it means that the caller lost the race and the promise resolved with a real value or error.

Note that this method gives few guarantees about the state of any background work after cancellation. If this promise is a chained promise then the background work must continue normally so that the original promise, and any other chained promises, can complete normally. If this promise is the "root" promise then the exact behaviour of [cancel\(\)](#) is implementation dependent. The background work may be halted, interrupted, or left to continue normally.

Returns:

`true` if this task completed by being cancelled

Class Promises

[org.osgi.util.promise](#)

```
java.lang.Object
└─ org.osgi.util.promise.Promises
```

```
public class Promises
    extends Object
```

Utilities for promises.

Constructor Summary

[Promises](#) ()**Page**

32

Method Summary

		Page
static Promise <T>	newResolvedPromise (T value) Create a promise that is already resolved with the supplied value	33
static Promise <Object>	parallel (Promise <?>... promises) Run all the given promises in parallel.	32
static Promise <T>	parallel (T[] array, Promise <? extends T>... promises) Run all the given promises in parallel.	33
static Promise <T>	shareableCopyOf (Promise <T> promise) Create a copy of a promise that can safely be shared with other clients.	33

Constructor Detail

Promises

```
public Promises()
```

Method Detail

parallel

```
@SafeVarargs
public static Promise<Object> parallel(Promise<?>... promises)
    throws Exception
```

Run all the given promises in parallel. This will return a new promise that will be resolved when all the given promises have been resolved.

Parameters:

`promises` - the array of promises

Returns:

a new promise

Throws:

`Exception`

parallel

```
@SafeVarargs
public static Promise<T> parallel(T[] array,
                                Promise<? extends T>... promises)
    throws Exception
```

Run all the given promises in parallel. This will return a new promise that will be resolved when all the given promises have been resolved.

Parameters:

`array` - the array that the return values from the promises are to be stored in. If the array is not big enough then a new array of the same element type will be used.
`promises` - the array of promises

Returns:

a new promise that will resolve when all of the supplied promises resolve.

Throws:

Exception

newResolvedPromise

```
public static Promise<T> newResolvedPromise(T value)
```

Create a promise that is already resolved with the supplied value

shareableCopyOf

```
public static Promise<T> shareableCopyOf(Promise<T> promise)
```

Create a copy of a promise that can safely be shared with other clients.

This method is useful for when a promise needs to be shared with a client that may call [Promise.cancel\(\)](#) on promises that it does not own.

Parameters:

`promise` - A new promise that will resolve when the supplied promise resolves.

Interface Success

org.osgi.util.promise

Type Parameters:

- Return - The type of the chained promise created by this callback
- Value - The type of the promise expected by this callback

public interface **Success**

A Callback that is made when the promise resolves successfully.

Method Summary		Page
Promise<Return>	call (Promise <? extends Value > promise) Called when the promise that this callback was registered with completed successfully.	34

Method Detail

call

[Promise](#)<[Return](#)> **call** ([Promise](#)<? extends [Value](#)> promise)
throws Exception

Called when the promise that this callback was registered with completed successfully. To simplify the remainder of this description we will refer to the [Promise](#) returned by [call\(Promise\)](#) as 'p' and the [Promise](#) returned by [Promise.then\(Success\)](#) as 'q'. If the return value 'p' is `null` then the chained promise 'q' will resolve immediately with a successful value of `null`. If 'p' is not null then the chained promise 'q' will resolve when the promise 'p' resolves.

Parameters:

promise - the [Promise](#) that resolved successfully

Returns:

A new promise in the chain, or null if the chain should resolve immediately.

Throws:

Exception

Java API documentation generated with [DocFlex/Doclet](#) v1.5.6

DocFlex/Doclet is both a multi-format Javadoc doclet and a free edition of [DocFlex/Javadoc](#). If you need to customize your Javadoc without writing a full-blown doclet from scratch, DocFlex/Javadoc may be the only tool able to help you! Find out more at www.docflex.com

8 Considered Alternatives

Initially this RFC used a builder model to create aggregate tasks. With the addition of the Promise API this is no longer necessary.

8.1.1 Completing tasks

Once a client has established a context then they can “complete” building the task using either the `launch()` method or `asPromise()` method. The `launch()` method is used for “Fire and Forget” style invocations, while the `asPromise()` method returns a `java.util.concurrent.Future`, typed appropriately for the return type of the asynchronous invocation.

Once one of the “completion” methods has been invoked the asynchronous task should begin executing. The real service object should be obtained by the Async Service implementation by using the client's `BundleContext` to call `getService()` on the `ServiceReference` used to create the async mediator.

Example – Fire and Forget:

```
private Async asyncService;

private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.createAsyncMediator(listRef);
    asyncService.build(mediator.contains("badEntry")).launch();
}
```

Example – With a promise:

```
private Async asyncService;

private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.createAsyncMediator(listRef);
    Future<Boolean> futureResult = asyncService
        .build(mediator.contains("badEntry")).asPromise();
    ...
}
```

The `Future` returned by the `asPromise()` method must obey the Java contracts for Futures. It must:

- be thread safe,
- Implement a cancel method that should make a best-effort to cancel or prevent the execution of the asynchronous invocation.
- provide a blocking get methods which returns the result of the execution
- throw an `ExecutionException` from the get methods, wrapping the real failure, if the asynchronous invocation threw an `Exception`
- throw a `CancellationException` from the get methods if the `Future` was cancelled

- Provide methods for determining whether a Future is finished, and whether it has been cancelled.

8.1.2 Registering Callbacks

Having established a context, a client may register callbacks with the asynchronous services implementation.

There are three kinds of callback:

- **Success Callbacks** – these are called with the result of the asynchronous invocation, if it returned normally. Implements the `org.osgi.service.async.SuccessCallback` interface.
- **Failure Callbacks** – these are called with the exception thrown by the asynchronous invocation, if it returned exceptionally. Implements the `org.osgi.service.async.FailureCallback` interface.
- **Completion Callbacks** – these are always called after the asynchronous invocation has completed. Completion Callbacks will be made after any success or failure callbacks have been made. Implements the `org.osgi.service.async.CompletionCallback` interface.

All three callback interfaces follow the Single Abstract Method principle, which means that they are able to be substituted for lambda expressions in Java 8.

Callbacks can be registered against the asynchronous invocation represented by the current context using the `onSuccess()`, `onFailure()` and `onCompletion()` methods:

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.createAsyncMediator(ref);
    asyncService.build(mediator.remove("badEntry"))
        .onSuccess(new MySuccessCallback())
        .onFailure(new MyFailureCallback())
        .onCompletion(new MyCompletionCallback()).launch();
}
```

Callbacks may also be used in conjunction with the `asPromise()` method.

8.2 More complex tasks

8.2.1 Running in Parallel

Most commonly asynchronous invocations are used to run multiple tasks in parallel. This can be achieved using the `parallel()` method to establish a new context. Once a new context has been established any calls to `onSuccess()`, `onFailure()` or `onCompletion()` will be applied to the new context.

Any parallel asynchronous invocations in an Asynchronous task are eligible to be run in parallel with the preceding asynchronous invocation. Note that this does not mean that the tasks will definitely run in parallel, for example there may be insufficient worker threads available to run additional tasks.

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
```

```
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.createAsyncMediator(ref);
    asyncService.build(mediator.contains("badEntry"))
        .onSuccess(new MySuccessCallback()) //Applies to "badEntry"
        .parallel(mediator.contains("goodEntry"))
        .onSuccess(new AnotherSuccessCallback()) // Applies to "goodEntry"
        .launch();
}
```

8.2.2 Running sequentially

Sometimes asynchronous invocations have an implicit ordering requirement. In this case a single task can be created that only starts running an invocation after the previous invocation has successfully completed. This is accomplished by using the `then()` method to establish a new context. This works in the same way as the `parallel()` method, but the new asynchronous invocation is only eligible to run after the previous task returns

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.createAsyncMediator(ref);
    asyncService.build(mediator.remove("badEntry"))
        .then(mediator.contains("badEntry"))
        .onSuccess(new AnotherSuccessCallback()) // Applies to contains()
        .launch();
}
```

8.2.3 Waiting for previous invocations

Sometimes it is not enough to wait for a single asynchronous invocation to complete, and instead you need to wait for a group of tasks to complete before continuing. In this case the `afterAll()` method establishes a new asynchronous invocation context that is only eligible to execute after all previous asynchronous invocations have completed.

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
```

```
List mediator = asyncService.createAsyncMediator(ref);
asyncService.build(mediator.contains("badEntry"))
    .onSuccess(new MySuccessCallback())
    .parallel(mediator.contains("goodEntry"))
    .onSuccess(new AnotherSuccessCallback())
    .afterAll(mediator.addAll(Arrays.asList("goodEntry", "badEntry")))
    .launch();
}
```

8.2.4 Receiving multiple promises

Some clients that issue multi-invocation asynchronous tasks will wish to consume the results of their asynchronous invocations using the `Future` API. Therefore the `org.osgi.service.async.AsyncCompleter` interface defines an `asPromises()` method that completes the asynchronous task, returning a `List` of `Future` objects, representing the completions of each asynchronous invocation in the task. The order of the objects in the `List` is the same as the order in which the client created the asynchronous invocation contexts that make up the task.

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}

public synchronized void doStuff() {
    List mediator = asyncService.createAsyncMediator(ref);
    List<Future<?>> promises = asyncService
        .build(mediator.contains("badEntry"))
        .parallel(mediator.contains("goodEntry"))
        .parallel(mediator.contains("anotherEntry"))
        .asPromises(); //List is in the order 'bad, good, another'
    ...
}
```

8.2.5 Registering aggregate callbacks

It can be useful for clients to receive notifications about the overall progress of an asynchronous task, rather than about its individual elements. To support this use case the `AsynchronousBuilder` interface declares the `andFinally()` method, which establishes a special context. This context represents the entire asynchronous task, not just a single asynchronous invocation within the task. This means that any callbacks registered using `onSuccess()` or `onCompletion()` will be called when the entire task has completed. Success callbacks will be called with a null argument. Failure callbacks registered with `onFailure()` will receive callbacks immediately when any of the asynchronous invocations that make up the asynchronous task fail. This means that the failure callback may be called multiple times, possibly concurrently.

It is supported to use aggregate callbacks as well as callbacks on individual asynchronous invocations.

Example:

```
private Async asyncService;
private ServiceReference<List> listRef;

@Reference
synchronized void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = List.class)
synchronized void setList(ServiceReference<List> list) {
    listRef = list;
}
```

```
}  
public synchronized void doStuff() {  
    List mediator = asyncService.createAsyncMediator(ref);  
    List<Future<?>> promises = asyncService.build(mediator.contains("badEntry"))  
        .parallel(mediator.contains("goodEntry"))  
        .parallel(mediator.contains("anotherEntry"))  
        .andFinally().onCompletion(new AllChecksFinished())  
        .asPromises(); //List is in the order 'bad, good, another'  
    ...  
}
```

9 Security Considerations

Asynchronous Services implementations must be careful to avoid elevating the privileges of client bundles when calling services asynchronously. This means that the implementation must:

- Use the client bundle to load interfaces when generating the asynchronous mediator. This prevents clients from gaining access to interfaces they would not normally be permitted to import.
- Use the client's bundle context when retrieving the target service. This prevents the client bundle from being able to make calls on a service object that they would normally be forbidden from obtaining.

Further security considerations can be addressed using normal OSGi security rules, access to the Async service can be controlled using `ServicePermission[Async, GET]`.

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

10.2 Author's Address

Name	Tim Ward
Company	Paremus Ltd
Address	
Voice	
e-mail	tim.ward@paremus.com

10.3 Acronyms and Abbreviations

10.4 End of Document