RFC 206 - Asynchronous Services

Draft

37 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

0.2 Trademarks

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 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

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 asynchronous

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 ~~Proxy~~Mediator – A ~~proxy~~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

~~Always~~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 Future.

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 ~~proxy~~mediator are used to register the asynchronous executions that should occur.

5.1 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.

5.2 Async ~~Proxies~~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 ~~proxy~~mediator for~~to~~ the real service.

```
<T> T createAsyncMediator(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.createAsyncMediator(ref);
    ...
}
```

5.2.1 Generating a Mediator

When creating the Async ~~Proxy~~Mediator object the Async Service should attempt to load all of the classes listed in the objectC~~c~~lass 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 resulting proxy should implement all of the interfaces that can be loaded by the client bundle, and be defined using the client bundle's classloader~~

The async service must then generate a mediator object 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.

5.2.2 ~~Class Proxying~~Generating a Mediator for Concrete Classes

If a service is registered advertising one or more concrete class types then generating a mediator~~Proxying Java interfaces can be easily achieved using the java.lang.reflect.Proxy class, however proxying class types~~ requires more complex handling. If an Async service implementation supports generating mediator objects for concrete classes ~~proxying~~ then it should ~~declare~~register ~~the~~a service property with the name org.osgi.service.async.~~proxy~~mediate.classes=true. If concrete classes ~~proxying is~~are supported then the ~~proxy~~mediator object created by the Async service should also inherit from the ~~lowest~~most specialised ~~subclass~~concrete type listed in the target service's objectclass property. There are three reasons why the Async service may not be able to ~~proxy~~mediate a class type:

1. The ~~lowest sub-~~most specialised type is final
2. The ~~lowest sub-~~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 ~~proxying~~mediator.

5.2.3 Async ~~Proxy~~Mediator return types

When invoked the Async ~~proxy~~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.

5.2.4 Thread safety

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

5.3 Building simple tasks

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

5.3.1 Establishing a context

To begin an asynchronous task the client invokes a method on the asynchronous ~~proxy~~mediator. The client then passes the result of that invocation to the Async service to establish the asynchronous invocation as the “current context”. The context 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.createAsyncMediator(ref);
    AsyncBuilder<Boolean> builder = asyncService
           .build(mediator.contains(“badEntry”));
      ...
}
```

At this point the Asynchronous task is not complete, and the asynchronous invocation will not have started. Establishing the context has two purposes:

- To register the method call made on the ~~proxy~~mediator as an asynchronous invocation in this task.
- To establish return type information for listener and completion methods

If there is an error establishing a context, for example a client calls the build method without having invoked the async ~~proxy~~mediator, then the Async service should throw an `AsyncException`.

Once a context has been established, clients can continue to build the asynchronous task in one of several ways:

5.3.2 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 ~~proxy~~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(ref);
    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(ref);
        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.

5.3.3 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.
- ~~Always~~Completion Callbacks – these are always called after the asynchronous invocation has completed. ~~Always~~Completion Callbacks will be made after any success or failure callbacks have been made. Implements the `org.osgi.service.async.`~~`Always`~~`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 ~~`always`~~`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.

5.3.4 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, using an `org.osgi.service.async.VoidMethodCall`. This interface defines a single abstract method callback inside which the client can make the void method call. In Java 8 this can be simplified to a lambda expression that makes the void method call. Another option for the client is to break the fluency of the builder, and to call the void method as a separate statement.

Example – Anonymous Inner class:

```
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(new VoidMethodCall() {
       public void invokeVoid() { mediator.clear(); }
    }).launch();
}
```

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(ref);
```

```
        mediator.clear()
        asyncService.build((Void) null).launch();
    }
```

Example – Lambda 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(ref);
        asyncService.build(() -> mediator.clear()).launch();
    }
```

SuccessCallbacks for void methods are passed null as the return argument from the method.

5.4 Building more complex tasks

Asynchronous tasks may consist of multiple distinct asynchronous invocations. The builder returned by the Async service (`org.osgi.service.async.AsyncBuilder`) therefore supports establishing a new asynchronous invocation context after any necessary callbacks have been registered:

5.4.1 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 ~~`always`~~`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();
    }
```

5.4.2 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();
}
```

5.4.3 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();
    }
```

5.4.4 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'
    ...
}
```

5.4.5 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 ~~always~~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'
    ...
}
```

5.5 Execution Failures

There are a variety of reasons that Asynchronous invocations may fail. In any of these cases the asynchronous invocation should fail with an `org.osgi.service.async.AsyncException`. This exception should be passed to any failure callbacks, and should be the cause of any `ExceutionException` thrown by `Future#get()`

- If the client bundle's bundle context becomes invalid before looking up the target service
- If the target service becomes unavailable before making the asynchronous invocation, or returns null on lookup
- 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 a previous dependent invocation from a `then()` or `afterAll()` call failed by throwing an exception

If the target service is successfully invoked, but the method call throws an exception, then this should be passed to the failure callback without wrapping it. It should also be the cause of any `ExceutionException` thrown by `Future#get()`

5.6 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 `registerCallbacks(SuccessCallback, FailureCallback)` method, holding on to the `Cancellable` returned by the invocation
3. On the same thread make the desired asynchronous invocation on the target service.
4. The `AsyncDelegate` should begin asynchronously executing the method, and return a garbage value to the caller. When asynchronous execution completes the `AsyncDelegate` should invoke the relevant callback method, depending on whether invocation returned normally, or threw an exception.
5. If at any point the `Cancellable` is cancelled then the `AsyncDelegate` should make a best-effort attempt to stop asynchronous execution of the task. If the asynchronous invocation from step 3 has not yet been made then the `AsyncDelegate` should discard the registered callbacks and return to “normal” operation.

Example

```
List service = ctx.getService(listRef);
if(service instanceof AsyncDelegate) {
     Cancellable c = ((AsyncDelegate)service)
               .registerCallbacks(new MySuccess(), new MyFailure());
     service.contains(“badEntry”);
}
// Callbacks will occur on completion of the asynchronous work
// Work can be cancelled using c.cancel();
```

6 Data Transfer Objects

It is unclear whether Asynchronous Services would benefit from DTOs

7 Javadoc

OSGi Javadoc

12/4/13 11:49 AM

Package org.osgi.service.async

Interface Summary		Page
Async	The Asynchronous Execution Service, used to access an `AsyncBuilder` for creating asynchronous jobs and "asynchronous mediators" for use with the builder.	*21*
AsyncBuilder	A Builder for asynchronous executions.	*23*
AsyncCompleter		*27*
CompletionCallback	This callback is passed to `AsyncBuilder` by users when they wish to be notified about the completion of their tasks.	*30*
FailureCallback	This callback is passed to `AsyncBuilder` by users when they wish to be notified about failures in their execution.	*31*
SuccessCallback	This callback is passed to `AsyncBuilder` by users when they wishe to be notified about successful completion of their tasks.	*32*
VoidMethodCall	This interface is used with `Async` and `AsyncBuilder` when making asynchronous calls of void methods.	*33*

Exception Summary		Page
AsyncException	This Exception is passed to a `FailureCallback` or returned as the cause of an `ExecutionException` from `Future.get()` when there was a problem starting the Asynchronous task.	*29*

Interface Async

org.osgi.service.async

`public interface` **`Async`**

The Asynchronous Execution Service, used to access an `AsyncBuilder` for creating asynchronous jobs and "asynchronous mediators" for use with the builder. Typical usage:

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

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

MyService asyncMediator = async.createAsyncMediator(ref);

Future<BigInteger> result = async.build(asyncMediator.getSumOverAllValues())
                                      .asPromise();
```

Calls can be made in parallel, chained, and callbacks can be made. For more information about this usage see AsyncBuilder

---

Method Summary		Page
`AsyncBuilder<Void>`	**`build`**`(VoidMethodCall call)` Create an `AsyncBuilder` for building asynchronous executions.	22
`AsyncBuilder<T>`	**`build`**`(T call)` Create an `AsyncBuilder` for building asynchronous executions.	21
`T`	**`createAsyncMediator`**`(<any> ref)` Create an asynchronous mediator for this service reference.	21

Method Detail

createAsyncMediator

`T` **`createAsyncMediator`**`(<any> ref)`

Create an asynchronous mediator for this service reference. The mediator will implement all of the interfaces declared in the `org.osgi.framework.Constants.OBJECTCLASS` property of the `org.osgi.framework.ServiceReference`. Method calls made on this mediator will not invoke the real service, and will not block.

Returns:
An asynchronous mediator for use with the AsyncBuilder returned by this service

---

build

```
AsyncBuilder<T> build(T call)
              throws IllegalStateException
```

Create an `AsyncBuilder` for building asynchronous executions.

Parameters:
`call` - the return value of a method call made on an async mediator created by `createAsyncMediator(ServiceReference)`

Returns:
throws IllegalStateException if no calls have been made to an async mediator by the current thread

Throws:

`IllegalStateException`

---

build

```
AsyncBuilder<Void> build(VoidMethodCall call)
                   throws IllegalStateException
```

Create an `AsyncBuilder` for building asynchronous executions. Use this method when you wish to asynchronously call a void method.

Parameters:

`call` - A `VoidMethodCall` that invokes a void method on the async mediator object

Returns:

throws IllegalStateException if after calling `VoidMethodCall.invokeVoid()` no calls have been made to an async mediator by the current thread

Throws:

`IllegalStateException`

Interface AsyncBuilder

org.osgi.service.async

Type Parameters:

`T` - The current return type of the Asychronous task.

All Superinterfaces:

AsyncCompleter<T>

```
public interface AsyncBuilder
extends AsyncCompleter<T>
```

A Builder for asynchronous executions. It allows multiple tasks to be executed in parallel, or sequentially, or a mixture of both

---

Method Summary		Page
`AsyncBuilder<Void>`	`afterAll(VoidMethodCall nextCall)` Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of all of the previous calls.	26
`AsyncBuilder<V>`	`afterAll(V nextCall)` Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of all of the previous calls.	25
`AsyncCompleter<Void>`	`andFinally()` Calling this method indicates to the framework that no new steps will be added to this task.	23
`AsyncCompleter<T>`	`onCompletion(CompletionCallback always)` Register a callback that should be made when the current step of this task has completed.	24
`AsyncBuilder<T>`	`onFailure(FailureCallback onFailure)` Register a callback that should be made if the current step of the asynchronous execution fails to complete successfully.	24
`AsyncBuilder<T>`	`onSuccess(SuccessCallback<? super T> onSuccess)` Register a callback that should be made if the current step of the asynchronous execution completes successfully.	24
`AsyncBuilder<Void>`	`parallel(VoidMethodCall parallelCall)` Indicate that another asynchronous service call should be made, and that it may execute in parallel with the current call.	25
`AsyncBuilder<V>`	`parallel(V parallelCall)` Indicate that another asynchronous service call should be made, and that it may execute in parallel with the current call.	24
`AsyncBuilder<Void>`	`then(VoidMethodCall parallelCall)` Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of the current call.	25
`AsyncBuilder<V>`	`then(V nextCall)` Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of the current call.	25

Methods inherited from interface org.osgi.service.async.AsyncCompleter
`asPromise, asPromises, launch`

Method Detail

andFinally

```
AsyncCompleter<Void> andFinally()
```

Calling this method indicates to the framework that no new steps will be added to this task. Calling this method changes the behaviour of subsequently added success and failure callbacks, which will now apply to the aggregate task as a whole.

Returns:
An AsyncCompleter that can be used to register callbacks for the entire asynchronous task

---

onSuccess

`AsyncBuilder<T> onSuccess(SuccessCallback<? super T> onSuccess)`

Register a callback that should be made if the current step of the asynchronous execution completes successfully.

Specified by:
`onSuccess` in interface `AsyncCompleter`

Parameters:
`onSuccess` - a callback that will be called with the value returned by this task if it has successfully completed

---

onFailure

`AsyncBuilder<T> onFailure(FailureCallback onFailure)`

Register a callback that should be made if the current step of the asynchronous execution fails to complete successfully.

Specified by:
`onFailure` in interface `AsyncCompleter`

Parameters:
`onFailure` - a callback that will be called if this task fails to complete successfully

---

onCompletion

`AsyncCompleter<T> onCompletion(CompletionCallback always)`

Register a callback that should be made when the current step of this task has completed.

Specified by:
`onCompletion` in interface `AsyncCompleter`

Parameters:
`always` - a callback that will be called when all parts of this task have completed

---

parallel

`AsyncBuilder<V> parallel(V parallelCall)`

Indicate that another asynchronous service call should be made, and that it may execute in parallel with the current call. Whether or not this call does execute in parallel is determined by the number and availability of threads in the underlying implementation. Note that this call establishes a new AsyncBuilder context, meaning that any subsequent calls to `onFailure(FailureCallback)`, and related methods will be for the new execution result.

Returns:
An AsyncBuilder representing the new context

---

parallel

`AsyncBuilder<Void> **parallel**(VoidMethodCall parallelCall)`

Indicate that another asynchronous service call should be made, and that it may execute in parallel with the current call. Whether or not this call does execute in parallel is determined by the number and availability of threads in the underlying implementation. Note that this call establishes a new AsyncBuilder context, meaning that any subsequent calls to `onFailure(FailureCallback)`, and related methods will be for the new execution result.

Returns:
An AsyncBuilder representing the new context

---

then

`AsyncBuilder<V> **then**(V nextCall)`

Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of the current call. Note that execution of the new task is only blocked by the current task, other parallel executions may not have completed when the new task starts. If you wish to wait for all parallel tasks to complete then you should use the `afterAll(Object)` method. Note that this call establishes a new AsyncBuilder context, meaning that any subsequent calls to `onFailure(FailureCallback)`, and related methods will be for the new execution result.

Returns:
An AsyncBuilder representing the new context

---

then

`AsyncBuilder<Void> **then**(VoidMethodCall parallelCall)`

Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of the current call. Note that execution of the new task is only blocked by the current task, other parallel executions may not have completed when the new task starts. If you wish to wait for all parallel tasks to complete then you should use the `afterAll(VoidMethodCall)` method. Note that this call establishes a new AsyncBuilder context, meaning that any subsequent calls to `onFailure(FailureCallback)`, and related methods will be for the new execution result.

Returns:
An AsyncBuilder representing the new context

---

afterAll

`AsyncBuilder<V> **afterAll**(V nextCall)`

Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of all of the previous calls. have completed. Note that execution of the new task is blocked by all previous tasks Note that this call establishes a new AsyncBuilder context, meaning that any subsequent calls to `onFailure(FailureCallback)`, and related methods will be for the new execution result.

Returns:
An AsyncBuilder representing the new context

---

afterAll

`AsyncBuilder<Void> afterAll(VoidMethodCall nextCall)`

Indicate that another asynchronous service call should be made, and that it may only begin to execute after the completion of all of the previous calls. have completed. Note that execution of the new task is blocked by all previous tasks Note that this call establishes a new AsyncBuilder context, meaning that any subsequent calls to `onFailure(FailureCallback)`, and related methods will be for the new execution result.

Returns:

An AsyncBuilder representing the new context

Interface AsyncCompleter

org.osgi.service.async

All Known Subinterfaces:

AsyncBuilder

`public interface AsyncCompleter`

Method Summary		Page
`Future<T>`	`asPromise()` Complete building this asynchronous task, returning a Future that can be used to obtain the result of the asynchronous execution.	27
`List<Future<?>>`	`asPromises()` Complete building this asynchronous task, returning a list of Futures that can be used to obtain the results of the component parts of this asynchronous execution.	27
`void`	`launch()` Complete building this asynchronous task, enabling it to be run.	27
`AsyncCompleter<T>`	`onCompletion(CompletionCallback always)` Register a callback that should be made when all steps of this task have completed.	28
`AsyncCompleter<T>`	`onFailure(FailureCallback onFailure)` Register a callback that should be made if any step of the aggregate task fails to complete successfully.	28
`AsyncCompleter<T>`	`onSuccess(SuccessCallback<? super T> onSuccess)` Register a callback that should be made if all steps of this aggregate task complete successfully	28

Method Detail

launch

`void launch()`

Complete building this asynchronous task, enabling it to be run.

---

asPromise

`Future<T> asPromise()`

Complete building this asynchronous task, returning a Future that can be used to obtain the result of the asynchronous execution. If there were multiple parts to this execution then this Future will only represent the state of the final task. When it `Future.isDone()` it does not necessarily indicate that other parallel tasks have completed.

Returns:

The result of the final asynchronous invocation in this task as a Future

---

asPromises

`List<Future<?>> asPromises()`

Complete building this asynchronous task, returning a list of Futures that can be used to obtain the results of the component parts of this asynchronous execution.

Returns:
A List containing promises for each asynchronous invocation in the asynchronous task, in the order they were declared when building the task.

---

onSuccess

`AsyncCompleter<T> onSuccess(SuccessCallback<? super T> onSuccess)`

Register a callback that should be made if all steps of this aggregate task complete successfully

Parameters:
`onSuccess` - a callback that will be called with the value `null` if all parts of this task successfully complete

---

onFailure

`AsyncCompleter<T> onFailure(FailureCallback onFailure)`

Register a callback that should be made if any step of the aggregate task fails to complete successfully.

Parameters:
`onFailure` - a callback that will be called when any part of this task fails to complete successfully

---

onCompletion

`AsyncCompleter<T> onCompletion(CompletionCallback always)`

Register a callback that should be made when all steps of this task have completed.

Parameters:
`always` - a callback that will be called when all parts of this task have completed

Class AsyncException

org.osgi.service.async

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └org.osgi.service.async.AsyncException
```

All Implemented Interfaces:

Serializable

```
public class AsyncException
extends RuntimeException
```

This Exception is passed to a `FailureCallback` or returned as the cause of an `ExecutionException` from `Future.get()` when there was a problem starting the Asynchronous task. This may be because the backing service was unregistered, or because the Asynchronous implementation was unable to accept any more work. This Exception should not be used to wrap Exceptions thrown by the service execution. These should be given directly to the `FailureCallback` or set as the cause of an `ExecutionException` when using `Future.get()`

Constructor Summary	Page
AsyncException()	29
AsyncException(String message)	29
AsyncException(String message, Throwable cause)	29
AsyncException(Throwable cause)	29

Constructor Detail

AsyncException

```
public AsyncException()
```

AsyncException

```
public AsyncException(String message,
                      Throwable cause)
```

AsyncException

```
public AsyncException(String message)
```

AsyncException

```
public AsyncException(Throwable cause)
```

Interface CompletionCallback

org.osgi.service.async

`public interface` **`CompletionCallback`**

This callback is passed to `AsyncBuilder` by users when they wish to be notified about the completion of their tasks. It will be called when the task completes, regardless of whether it was successful or unsuccessful.

Method Summary		Page
`void`	**`complete`**`()` Called by the Asynchronous Services runtime to notify that an asynchronous call completed.	30

Method Detail

complete

`void` **`complete`**`()`

Called by the Asynchronous Services runtime to notify that an asynchronous call completed. The task may, or may not, have been successful.

Interface FailureCallback

org.osgi.service.async

`public interface FailureCallback`

This callback is passed to `AsyncBuilder` by users when they wish to be notified about failures in their execution.

Method Summary		Page
void	**failure**(Throwable t) Called by the Asynchronous Services runtime to notify that an asynchronous call failed to complete normally.	31

Method Detail

failure

`void failure(Throwable t)`

Called by the Asynchronous Services runtime to notify that an asynchronous call failed to complete normally. This may mean that the task could not be called at all, or that it threw an Exception while running.

Interface SuccessCallback

org.osgi.service.async

`public interface **SuccessCallback**`

This callback is passed to `AsyncBuilder` by users when they wishe to be notified about successful completion of their tasks.

Method Summary		Page
`void`	`success(T returnValue)` Called by the Asynchronous Services runtime to notify that an asynchronous call completed normally.	32

Method Detail

success

`void success(T returnValue)`

Called by the Asynchronous Services runtime to notify that an asynchronous call completed normally.

Parameters:

`returnValue` - The value returned by the asynchronous call. This will be null for void method executions.

Interface VoidMethodCall

org.osgi.service.async

```
public interface VoidMethodCall
```

This interface is used with `Async` and `AsyncBuilder` when making asynchronous calls of void methods. Usage example:

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

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

MyService asyncMediator = async.createAsyncMediator(ref);

async.build(asyncMediator.sendConfirmationEmail(customerEmail, orderNumber).launch();
```

Method Summary		Page
void	**invokeVoid**() This method is invoked by the Async service to determine which method should be asynchronously invoked	33

Method Detail

invokeVoid

```
void invokeVoid()
```

This method is invoked by the Async service to determine which method should be asynchronously invoked

Package org.osgi.service.async.spi

Interface Summary		Page
AsyncDelegate	This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently.	*35*
Cancellable		*36*

Interface AsyncDelegate

org.osgi.service.async.spi

`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
Cancellable	**registerCallbacks**(SuccessCallback<?> success, FailureCallback failure) This method can be used by the Async service to optimize Asynchronous execution.	35

Method Detail

registerCallbacks

```
Cancellable registerCallbacks(SuccessCallback<?> success,
                              FailureCallback failure)
```

This method can be used by the Async service to optimize Asynchronous execution. When called, the `AsyncDelegate` should begin tracking the thread that invoked this method, and store away the supplied callbacks. The next invocation made on this service by the tracked thread should be executed asynchronously (immediately returning a garbage value if necessary). When the asynchronous invocation completes the `AsyncDelegate` should call the relevant callback

Returns:

an object that may be used to 'cancel' the asynchronous execution. Cancellation is a best-effort operation, and may not halt the execution of a running operation.

Interface Cancellable

org.osgi.service.async.spi

```
public interface Cancellable
```

Method Summary		Page
void	`cancel()` Attempt to cancel this asynchronous execution.	36

Method Detail

cancel

```
void cancel()
```

Attempt to cancel this asynchronous execution. This may have no effect.

8 Considered Alternatives

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 ~~proxy~~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