139 Asynchronous Service Specification

Version 1.0

139.1 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. In many cases asynchronous programs can easier to write for high throughput systems.

The purpose of the Asynchronous Service is to bridge the gap between existing, primarily synchronous, services in the OSGi service registry, and asynchronous programming. The Asynchronous Service therefore provides a way to invoke arbitrary OSGi services asynchronously, providing results and failure notifications through the OSGi Promise API *****TODO REF*****

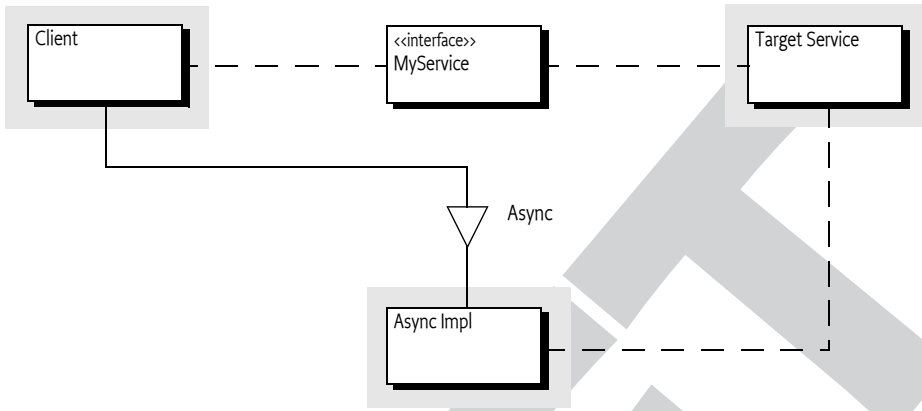
139.1.1 Essentials

- *Asynchronous Invocation* - A single method call that is to be executed without blocking the requesting thread.
- *Client* - Application code that wishes to invoke one or more OSGi services asynchronously.
- *Async Service* - The OSGi service representing the Asynchronous Services implementation. Used by the client to make one or more *Asynchronous Invocations*.
- *Async Mediator* - A mediator object created by the *Async Service* which represents the target service. Used by the *Client* to register asynchronous invocations.
- *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.

139.1.2 Entities

- *Async Service* - A service that can create Async Mediators and run Asynchronous Invocations.
- *Target Service* - A service that is to be called asynchronously by the client.
- *Client* - The code that makes asynchronous invocations using the Async Service
- *Promise* - An OSGi promise, representing the result of the Asynchronous Invocation.

Figure 139.1 *Class and Service overview*



139.2 Usage

This section is an introduction in the usage of the Async Service. It is not the formal specification, the normative part starts at *Async Service* on page 658. This section leaves out some of the details for clarity.

139.2.1 Synopsis

The Async service provides a mechanism for a client to *asynchronously* invoke methods on a target service. The service may be aware of the asynchronous nature of the call and actively participate in it, or be unaware and execute normally. In either case the client's thread will not block, and will continue executing its next instructions. Clients are notified of the completion of their task, and whether it was successful or not, through the use of the OSGi Promise API.

Each asynchronous invocation is registered by the client making a method call on an *Async Mediator*, and then started by making a call to the Async service that created the mediator. This call returns a Promise that will eventually be resolved with the return value from the asynchronous invocation.

An Async Mediator can be created by the client, either from an Object, or directly from a Service Reference. Using a Service Reference has the advantage that the mediator will track the underlying service. This means that if the service is unregistered before the asynchronous call begins then the Promise will resolve with a failure, rather than continuing using an invalid service object.

139.2.2 Making Asynchronus Invocations

The general pattern for a client is to obtain the Async service, and a ServiceReference for the target service. The client then creates an Async Mediator for the target service, invokes a method on the mediator, then starts the asynchronous call. This is demonstrated in the following 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"));
    ...
}
```

This example demonstrates how simply clients can make asynchronous calls using the Async service. The eventual result can be obtained from the promise using one of the relevant callbacks.

One important thing to note is that whilst the call to asyncService.call(...) causes the asynchronous invocation to begin, the actual execution of the underlying task may be queued until a thread is available to run it. If the service has been unregistered before the execution actually begins then the promise will be resolved with a ServiceException. The type of the ServiceException will be ASYNC ***** TODO REF core *****

139.2.3 Asynchronous invocations of void methods

The return value of the mediator method call is used to provide type information to the Async service. This, however, does not work for void methods that have no return value. In this case the client can either pass an arbitrary object to the call method, or use the zero argument version of the call method. In either case the returned promise will eventually resolve with a value of null. This is demonstrated in ???

```
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);
    mediator.clear();
    Promise<Void> promise = asyncService
           .call();
    ...
}
```

139.2.4 Multi Threading

By their very definition asychronous tasks do not run inline, and typically they will not run on the same thread as the caller. This is not, however, a guarantee. A valid implementation of the Async service may have only one worker thread, which may be the thread currently running in the client code. Asynchronous invocations also have the same threading model as the OSGi Promise API. This means that callbacks may run on arbitrary threads, which may, or may not, be the same as the client thread, or the thread which executed the asynchronous work.

It is important for multi-threaded clients to note that calls to the mediator and async service must occur on the same thread. For example it is not supported to invoke a mediator using one thread, and then to begin the asynchronous invocation by calling one of the Async.call(...) methods on a different thread.

139.3 Async Service

The Async service is the primary interaction point between a client and the Async Service 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 mechanisms, either directly using the OSGi framework API, or using dependency injection.

The Async service is used to:

- Create async mediators
- Begin asynchronous invocations
- Obtain Promise objects representing the result of the asynchronous invocation

139.3.1 Using the Async service

The first action that a client wishing to make an asynchronous invocation must take is to create an async mediator using one of the mediate methods. Once created the client invokes the method that should be run asynchronously, supplying the arguments that should be used. This call records the invocation, but does not start the asynchronous task. The Asynchronous task begins when the client makes invokes one of the call methods on the Async service. The call methods must return a ~~valid~~ Promise representing ~~the state of~~ the asynchronous invocation. This must resolve with the value returned by the asynchronous invocation, or fail with the failure thrown by the asynchronous invocation.

139.3.2 Asynchronous failures

There are a variety of reasons that asynchronous invocations may be started correctly by the client, but then fail without running the asynchronous task. In any of these cases the Promise representing the asynchronous invocation must fail with an org.osgi.framework.ServiceException. This ServiceException must be initialised with a type of ASYNC (***** Ref core R6 *****).

The following list of scenarios is not exhaustive, but indicates failure scenarios that must result in a ServiceException

- If the client is using a service reference backed mediator and the client bundle's bundle context becomes invalid before looking up the target service.
- If the client is using a service reference backed mediator and the service is unregistered before making the asynchronous invocation.
- If the client is using a service reference backed mediator and the service lookup returns null
- 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 using the recorded arguments.

139.3.3 Thread safety and instance sharing

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. Whilst the async service is able to be used across multiple threads, if a client wishes to make an asynchronous invocation then the call to the mediator and the call to `async.call(...)` must occur on the same thread. The returned Promise may then be shared between threads if required.

It is expected, although not required, that the Async service implementation will use a Service Factory to create customized implementations for each client bundle. This simplifies the tracking of the relevant client bundle context to use when performing service lookups on the client bundle's behalf. Clients should therefore not share instances of the Async service with other bundles. Instead both bundles should obtain their own instances from the service registry.

139.4 The Async Mediator

Async Mediators are dynamically created objects that have the same type or interface as the object being mediated, and are used to record method invocations and arguments. Mediators may be created either from a `ServiceReference` or from a service object. The actions and overall result are similar for both `mediate(...)` methods, with the primary difference being the manner in which the types to be mediated are determined.

139.4.1 Determining the mediated types for a `ServiceReference`

Mediator objects for service references are lazy, and creating one should not result in a lookup of the service object. Therefore, when creating the Async Mediator object the Async Service must not introspect the service object, but instead it should attempt to load all of the types 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 successfully loaded classes are then used when creating the mediator object. If the set of successfully loaded classes is null then an IllegalArgumentException must be thrown.

139.4.2 Determining the mediated types for a service object

When mediating a service object there is no way to be lazy, therefore the service object can be introspected to determine its type. In this case the class to be mediated is the class returned by calls to `serviceObject.getClass()`.

139.4.3 Building the mediator object

Once the set of types to be mediated has been determined then the set must be filtered into java interface types and java class types. The generated mediator object must extend the most specialised class type, and implement all of the provided interfaces. If there is no class type to be extended then the mediated object should extend `java.lang.Object`.

There are three reasons why the Async service may not be able to mediate a class type:

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

If any of these constraints are violated then the Async service should fall back to the next most specialised type, creating an interface-only mediator if necessary.

139.4.4 Async mediator behaviours

When invoked the Async mediator should record the method call, and its arguments, and then return rapidly (i.e. it should not perform blocking operations). The values returned by the mediator object are opaque, and 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. If the mediated method call has a return type, specifically it is non-void, then this object must be passed to the the async service's call method when beginning the asynchronous invocation

139.4.5 Thread safety and instance sharing

Async mediators, unlike instances of the Async service, are not required to be thread safe. Clients should not share mediator objects with other bundles, or accross threads. Also, if a client wishes to make an asynchronous invocation then the call to the mediator and the call to async.call(...) must occur on the same thread. The returned Promise may then be shared between threads if required.

Async mediators created from ServiceReference objects are lazy, and may remain directly associated with the service reference and client bundle after creation. Clients should therefore not share mediator objects with other bundles. Instead both bundles should create their own mediators.

139.5 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 may advertise this to clients and to the Async Service by implementing org.osgi.service.async.AsyncDelegate. This interface can be used by the asynchronous services implementation, or by the client directly, to make an asynchronous call on the service.

When making an asynchronous invocation the async service must check to see whether the target service implements AsyncDelegate. If the target service does implement AsyncDelegate then the async service must delegate the asynchronous call using the async method.

If the call to the AsyncDelegate returns a Promise, then the Promise returned by the async service must be resolved with that Promise. If the call to the AsyncDelegate returns null then the async service must continue with the asynchronous invocation as if the target service did not implement AsyncDelegate. If the call to the AsyncDelegate throws an Exception then this must be used to fail the promise returned by the async service.

Because the AsyncDelegate behaviour is transparently handled by the async service, clients of the async service do not need to know whether the target service implements AsyncDelegate or not, their usage pattern can remain unchanged.

139.6 Security

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. For example access to the Async service can be controlled using ServicePermission[Async, GET].

139.7 org.osgi.service.async

Asynchronous Services Package Version 1.0.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

139.7.1 Summary

- `Async` - The Asynchronous Execution Service.
- `AsyncDelegate` - This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently.

139.7.2 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.

Provider Type Consumers of this API must not implement this type

139.7.2.1 public Promise<R> call(R r)

Type Arguments <R>

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

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());
```

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

139.7.2.2 public 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 preferrable 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

139.7.2.3 public T mediate(T target)

Type Arguments <T>

target The service object

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 call() 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());
```

Returns A mediator for the service object

139.7.2.4 public T mediate(ServiceReference<T> target)

Type Arguments <T>

target The service object

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 call() 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 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());
```

Returns A mediator for the service object

139.7.3 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.

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

m the method that should be asynchronously executed

args the arguments that should be used to invoke the method

□ 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.

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

139.8 References

[1] *OSGi Core Release 6*
http://www.osgi.org/Specifications/HomePage