RFC 226 - Scheduling

Draft

32 Pages

Abstract

10 point Arial Centered.

This RFC defines a time service that can provide time aspects like delays, timeouts, and periodic scheduling with the option to have cron like scheduling using the whiteboard pattern. The solution takes advantage of the OSGi Promise API and the date-time API introduced in Java 8.

Copyright © OSGi Alliance 2016

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

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

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

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

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

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

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

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

0.2 Trademarks

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

0.3 Feedback

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

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 9.1.

`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	May 2016	David Bosschaert – initial version based on RFP 166
Design info	May 2016	Peter Kriens

1 Introduction

This RFC originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that needs to be solved.

"*The only reason for time is so that everything doesn't happen at once.*" as Albert Einstein said. Though we are trying to prove Einstein wrong by adding more and more cores to our CPUs it is clear that time is an uncomfortable citizen in our software world. Simple concepts like before-after relations are surprisingly hard to work with, trying to schedule a number

Copyright © OSGi Alliance 2016
All Rights Reserved

of actions in time, or just trying to work with dates and time in the light of the myriad of calendars and timezones further highlight the complexity of time. The primary reason of this complexity is that software has no built-in concept of time. We do not have variables that depreciate or easy primitives to spread out an operation over time (at least after we abandoned the delay loop).

The result is that there are a myriad of APIs in Java to handle delays, timeouts, and periodic scheduling. This RFP analyzes what there currently is and what is missing from an OSGi perspective.

2 Application Domain

2.1.1 Direction of Control

Time aspects of software are highly related with the *direction of control* between collaborating *actors*, where an actor is a piece of code. If an actor gets called it has no control over at what time it gets control. If the actor has control, it is expensive to delay with delay loop so the control must be temporarily handed over. Threads can be used for delaying inline because timed waits can be performed that look identical to a delay loop in the code but now the CPU could be busy with more useful work then executing NOPs.

A program that uses this threaded model is the easiest way to read and write programs since the *flow* of the code is the most concise possible in our current languages. However, the model does not scale well since threads are still expensive resources and the synchronous model requires one thread for each continuous flow. That said, a popular language like Erlang evolves around hundreds of thousands of (lightweight) threads that process messages by waiting for a message. The state of the actor is partly maintained by where the actor waits for the next message and the stack.

2.1.2 Push & Pull

A *producer* is an actor that has objects that it wants to convey to a *client*. Software allows us (surprisingly) only two options. The producer *pushes* the objects to the client or the client *pulls* the objects from the producer, the difference: time. For example, the producer can write (push) bytes to an Output Stream implemented by the client. Alternatively, the producer can implement an Input Stream and wait until the client reads (pulls) a byte. Though the same bytes are transferred, this direction of control has dramatic consequences for how we write our software as well as the runtime timing.

The more common software model in Java has been clients that pull since this model is closer to procedural languages, especially when they do not have lambdas. That is, we have iterators in Java. The client pulling allows the client to maintain the state on the call stack and procedural languages have fine grained highly optimized facilities for this model. For example, the Collections in Java support iterators that are based on the pull model: the client calls `next()` to get the next object.

2.1.3 Event Based Models

The increased complexity and distribution of applications (where the the flow timing becomes much more significant) is making the client-is-pushed model more popular despite its increased complexity. Since the client gets called whenever objects are available there is less waiting and it becomes easier to orchestrate multiple flows.

Java 8 significantly changed this landscape by introducing *lambdas*. Lambdas are objects that capture context and code, allowing the code to be passed around as an argument. Lambdas reduce the conceptual cost of using callback mechanisms because lambdas are lexically dramatically smaller then corresponding (albeit virtually

Copyright © OSGi Alliance 2016
All Rights Reserved

identical in utility) inner classes. Java 8 also makes it easier to use the context of the lambda invocation by removing the need to mark all scoped variables as final (although it falls short of full closures).

Having lambdas made it possible to provide many push based clients in existing APIs. The `forEach` method on collections pushes the members of the collection to a lambda. Another example is the new Java 8 Stream library. It an interesting mix of a basic pull API combined with a push model that leverages lambdas. Streams are based on iterators, thus are pull, but the streams push the elements of the stream onto lambdas.

Though new Java 8 API uses a push model because that is very effective with lambda's, they did not allow an event based model where objects are not already present. For distributed applications the *event* model is more attractive where the asynchronously arriving events are pushed to the client.

2.1.4 Promises & Java RX

In Enterprise Edition Release 6, the OSGi Alliance introduced a Java *Promise*, based on the Javascript promises. A Promise removes any synchronicity between the producer and the client. The Promise represents a value that will arrive in the future or it will signal an error. Promises make it easier to sequence a number of steps in time without blocking. That is, a a function can define a multi-step sequence of actions, keep its state local to the function, handle errors that occur in any of the actions centrally, and still return immediately. That is, a Promise acts as a broker between a producer and a client to remove the synchronicity between them.

It is common that sequence of steps can have time outs (execute before) or require intermediate delays (do not execute before). Timeouts and delays can be implemented with intermediate promises that use a *scheduler*.

Promises are about a single return value. However, many problems require a (continuous) stream of events. Netflix promotes an open source project RXJava that provides push based stream model. The model uses a similar API as the Java 8 Stream API to process the objects but behind the scenes uses an event model to push objects into it.

2.1.5 Timers

Java has had a java.util.Timer class since the 1.0. Clients can create a Timer object, which creates a background thread, and then the client can register *tasks* with a *schedule*, a specification of when to *invoke* the task. A task is some object that can be executed, in this case a Timer Task. The tasks are then efficiently *scheduled* and invoked when they expire. An invocation is done in the scheduler thread. Long running tasks can therefore postpone the invocation of other tasks if previous tasks take a long time or the system is busy.

Tasks are *scheduled*. A schedule could be a *delay*. In that case the task was executed after the given *duration*. A duration is fixed length of time, for example 2 hours. Durations in the Timer are specified in a long representing milliseconds. A task could also be scheduled at a certain *instance time*, using a Date object. Instance time is defined in Java 8 as a number of nanoseconds from the *epoch*. The epoch is a fixed moment in time Jan 1 1970 0:0 UTC.

A schedule could also be *periodic*. A periodic schedule executes the Timer Task continuously with an *interval*. The interval is the duration between two invocations. Since Timer tasks take time to execute and there can be other causes for upholding the Timer Scheduler the actual interval can *skewed* and thus be longer than the given interval. Periodic schedules can provide an initial delay or a specific date for *initial scheduling*.

Periodic schedules can also be scheduled at a *fixed rate*. A fixed rate periodic schedule ensures that the interval is not skewed. Each invocation is scheduled at a multiple interval duration from the initial scheduling instance time.

Timer tasks could also be *canceled* by the clients. Since a task can be canceled at any moment in time, there is a potential race condition between the execution of the task and the cancelation. This in general requires the task to use locks or atomic booleans to verify executing against an expired context. Since a scheduled task has very little cost and the task can still be executed in an expired context a cancelation is not always opportune.

Since the Timer is a Java class, it includes the maintenance methods together with the collaboration methods. In general, each actor in an application tends to create their own timer and timers are generally not shared.

Copyright © OSGi Alliance 2016 All Rights Reserved

2.1.6 Java Util Concurrent

Java 5 introduced the java.util.concurrent package that provided a number of services to run tasks in a background thread and/or schedule at a given date-time or delay. The Scheduled Executor Service specifies a service that has the same facilities as the Timer class. However, durations are specified with a 2 parameters: a magnitude and a Time Unit and did not allow the initial scheduling to be scheduled at a date-time. It did provide background scheduling and any task invocations were executed away from the scheduler thread.

The Scheduled Executor Service is an interface. This interface contains the scheduling methods as well as life cycle and maintenance methods. The Executors class provides a number of standard implementations.

2.1.7 Quartz

The Timer and Scheduled Executor Service are limited to *instance* time. However, many tasks require scheduling based on dates and times. For example, some tasks must be executed every third Wednesday of the week or at 5 AM Sunday morning. The Linux cron jobs used a specific format to specify such a scheduling by creating a *mask* for seconds, minutes, hours, day of the week, day of the month, month, and year. When the mask matches, the given task is executed.

In the Java world, this facility was mainly provided by the Quartz library. It allows the specification of a cron like schedule and then executes predefined tasks. Tasks in this context are specified by their class name, they are then loaded and instantiated when needed.

2.1.8 Java 8 Date and Time

After having been shown the way by Joda Time Java 8 finally has a mature date and time library. The library provides now abstractions for local times and dates (i.e. 4 o'clock) that can be mapped to a specific instance time later when the day and location are known.

One relevant aspect of this library is the Temporal Adjuster interface. The Temporal Adjuster is a strategy for adjusting a *temporal* object. A temporal object is an instance time, a local date or time, or any other object that understands an actual time. *Adjusters* exist to externalize the adjusting. For example, an adjuster that sets the next date-time avoiding weekends, or one that sets the date to the last day of the month.

Practically, Temporal Adjusters can be used to abstract the concept of the delay, date, or periodic interval in the Timer/Scheduled Executor Service but they also encompass the cron mask.

2.2 Terminology + Abbreviations

3 Problem Description

Currently time is quite chaotic in Java. It is not clear what API to use to schedule tasks and for relatively simple task like cron scheduling it is necessary to escape to a significant library like Quartz.

- A major problem is that it is hard to share schedulers. One of the possible solutions would be to registers a Timer or Scheduled Executor Service in the service registry and then share this object between all components. However, both APIs include the life cycle and maintenance methods on their interfaces. In an OSGi environment, life cycle and interface must be clearly separated from the collaboration API.

Copyright © OSGi Alliance 2016
All Rights Reserved

- The existing Java Timer and Scheduled Executor Service also do not work with the very advanced Java 8 Date Time API. The Timer is limited to milliseconds or Date and the ScheduledExecutorService has the awkward kludge called TimeUnit.
- The Promise API lacks any time awareness.

Therefore, the solution this RFP seeks is a service that provides time support in the OSGi way leveraging Java 8 and our Promise API. This service should make it easy to handle timeouts, delays, and periodic schedules with very flexible interval specifications through a shared service.

4 Requirements

4.1.1 Scheduler

- S0010 – Provide a way to execute a lambda in another thread with an optional schedule
- S0020 – A schedule must at least support:
  - A delay
  - A cron/Quartz like syntax
  - An interval with an initial delay
  - An temporal object
  - Temporal Adjuster based
- S0060 – Allow the scheduled lambda to throw Exceptions. Periodic tasks must always continue to be executed even if they throw Exceptions.
- S0070 – It must be possible to specify intervals so that they are at a fixed rate.
- S0080 – Provide a Temporal Adjuster implementation for the Quartz cron syntax
- S0090 – Provide a Temporal Adjuster implementation for an interval based on durations.
- S0100 – Periodic schedules must be able to provide a push model for the objects the lambda returns on each invocation. This will allow periodic schedules to act as a timed event producer.
- S0110 – For a given registration and scheduled instance time the lambda must be executed only once.
- S0120 – Lambdas that take too much time to execute (at least 10 secs) must be black listed. This should be configurable.
- S0130 – Exceptions in lambda execution must be logged
- S0140 – It must be possible to stop a periodic schedule
- S0150 – One time schedules should not be canceable since this creates complex race conditions.

4.1.2 Promises

- P0010 – Wrap a promise so that it will fail if it is not resolved before a given delay or instance time.
- P0020 – Wrap a promise so that it will delay after it is resolved. Failures are reported immediately.

Copyright © OSGi Alliance 2016
All Rights Reserved

- P0030 – Wrap a promise so that it will not resolve before a given instance time. Failures are reported immediately.
- P0040 – Lambdas that are scheduled once must be able to return a value through a promise.

4.1.3 Whiteboard

- W0020 – Provide a whiteboard service model to schedule a service
- W0030 – The syntax for the whiteboard schedule must be Quartz cron syntax
- W0040 – Tasks must be repeated a configurable number of times (minimally 3) if they fail with an exception.
- W0050 – A Task must only be executed once for a given scheduled instance time.

5 Technical Solution

Quite often you need to run a task every hour, every third Wednesday, or on January one on odd years. From an OSGi perspective the design is quite straightforward, just register a service, specify the cron expression as a property and wait to be called back. The service diagram is as follows:

The API consists of the following interfaces:

- *Scheduler* – General service provider interface to access the different timing methods with promises and lambdas.
- *CronJob<W>* – A worker that needs to be activated at a given time that is defined by a cron specification.
- *RunnableWithException* – A Runnable that can throw checked exceptions. This interface enables the lambdas to throw exceptions.
- *CancellablePromise* – A Promise with a cancel method. Canceling means that the returned promise is failed, it does not imply that the underlying activity is canceled.
- *CancelException* – Singleton exception that is used to fail a CancelablePromise when it is canceled.
- *TimeoutException* – Singleton exception that is used to fail a CancelablePromise when it times out.
- *ShutdownException* – Singleton exception that is used to fail a CancelablePromise when the scheduler shuts down or corresponding Scheduler service is unget.
- *RequireSchedulerImplementation* – Annotation to require a scheduler when there is no direct service dependency

Copyright © OSGi Alliance 2016
All Rights Reserved

5.1 Cron Schedules

Cron schedules use the whiteboard pattern. To get called at a specific time a service must register a CronJob service with a service property with the cron specification. For example:

```
@Component(
    property = CronJob.CRON + "=1-30/2 * * * * ?"
  )
  public class CronComponent implements CronJob
    @Override
    public void run(Object data) throws Exception {
      System.out.println("Cron Component");
    }
  }
```

The CRON expression has 7 fields:

- Second in minute
- Minute in hour
- Hour in day
- Day of month
- Month in year
- Day of week
- Year (optional)

5.1.1 Cron Expression

The expression is matched against the actual time. Every time the time matches, the corresponding action is executed. For matching, each field can have quite a complex syntax. The simplest is if it is a wildcard, the asterisk ('*'). A wildcard always matches. The second form is just a number, this just matches at that position. For example, 0 12 * * * * * will match noon every day. You can also specify a range like 0-30. To repeat you can add a slash and the step value like 0/5 which repeats every fifth second (if used as the second field). And last but not least you can add additional syntax after a comma (,) like 1,5,15,45.

It is also possible to use predefined names:

`@yearly (or @annually)`	Run once a year at midnight on the morning of January
`@monthly`	Run once a month at midnight on the morning of the first day of the month
`@weekly`	Run once a week at midnight on Sunday morning
`@daily`	Run once a day at midnight
`@hourly`	Run once an hour at the beginning of the hour
`@reboot`	Run at startup

The scheduler uses the syntax made popular in Java by Quartz[3].. There are cron simulators to test these expressions since they can become quite complex.

The cron expression must be the last line in a multiline string. The expression can be preceded by comments and/or environment variables. Environment variables are property assignments. Comments and empty lines are ignored.

 All Rights Reserved

```
cron     ::=  ( env | comment | empty )* cron-expression
env      ::=  KEY '=' VALUE '\n'
comment  ::=  '#' any '\n'
empty    ::= '\n'
cron-expression ::= <see [3]> '\n'?
```

All tokens can be preceded by whitespace.

5.1.2 Environment

A Cron Job service can specify a DTO, interface, or any type that can be converted from a Map<String,String> with the Converter specification. The environment properties are put in a map and then converted to the generic type of the actual CronJob<T>. This makes it possible to have run methods that take proper Java types.

```
public interface CData {
   int port();
   String host();
}
public void run( CData data ) {
  …
}}
```

5.2 Delays & Schedules & Timers

Java 8 and OSGi Release 6 provide lambdas and promises. Since there are default methods in Java 8, the API is build around a few primitive methods with a number of default convenience methods.

5.2.1 Delays

The Scheduler service has a number of overloaded after methods to schedule calling a lambda after a certain amount of time:

```
scheduler.after( () →
  System.out.println("fire!"), 100 ); // ms
```

The following overloads are available:

- CancellablePromise<Instant> after(long ms);
- CancellablePromise<Instant> after(Duration d);
- <T> CancellablePromise<T> after(Callable<T> call, Duration d);
- <T> CancellablePromise<T> after(RunnableWithException r, long ms)

5.3 At a Given Time

It is also possible to call at a specific moment in time:

```
LocalDateTime localDateTime =
LocalDateTime.parse("2017-01-13T09:54:42.820Z", ISODATE);
```

Copyright © OSGi Alliance 2016 All Rights Reserved

```
ZonedDateTime zonedDateTime =
localDateTime.atZone(ZoneId.of("UTC"));
Instant instant = zonedDateTime.toInstant();
scheduler.at( () → System.out.println("fire!"), instant );
```

The following overloaded variations of the at method are available:

- CancellablePromise<Instant> at(long epochTime);
- <T> CancellablePromise<T> at(Callable<T> callable, long epochTime);
- <T> CancellablePromise<T> at(Callable<T> callable, Instant instant)
- CancellablePromise<Void> at(RunnableWithException r, Instant instant)
- CancellablePromise<Void> at(RunnableWithException r, long epochMilli)

5.4 Promises

Promises cannot be canceled nor have a timeout. This can be quite awkward in many asynchronous situations. The Scheduler therefore adds timeouts and cancelation to Promises. It defines the CancelablePromise:

```
public interface CancellablePromise<T> extends Promise<T> {
  boolean cancel();
}
```

The cancel method returns a boolean to indicate if the cancel was successful or if the Cancellable Promise was already done because it was resolved or failed before.

A Cancellable Promise wraps an existing promise and then provides a cancel method.When canceled, the Cancellable Promise will fail with a singleton CancelException. Additionally, the Scheduler service can also use this mechanism to provide a timeout:

```
void foo( Promise p ) {
    CancellablePromise cp = scheduler.before( p,
      Duration.ofMinutes(5) );
    cp.then( this::start ).
       then( this::secondStage ).
       then( this::thirdStage, this::failure );
 }
```

When the Cancellable Promise times out it will fail with a singleton TimeoutException.

Such Cancellable Promise now has 4 reasons to become done. It can be resolved, failed, canceled or timeout.

5.5 Periodic Schedules

An example of a schedule is:

```
Closeable rampUp = scheduler.schedule(  this::tick, 10, 20, 40, 80, 100 );
```

A schedule takes a list of delays. It will repeat the last delay until the returned Closeable is closed. The Closeable can be closed multiple times. If at the time of the closing the lambda is running then its thread must be interrupted.

Copyright © OSGi Alliance 2016
All Rights Reserved

Relative schedules must be scheduled relative to the start of the schedule. That is, repeated delays must be calculated from the start time of the schedule and not after the callback has finished. If a schedule finds that a period has already passed it must skip that callback and log a WARNING.

It is also possible to schedule via a Cron expression. The following example ticks every second second in the first half of each minute.

```
Closeable cron = scheduler.schedule(
  this::cronTick, "0-30/2 * * * * *" );
```

The following overloads exist:

- Closeable schedule(RunnableWithException r, String cronExpression) throws Exception;
- <T> Closeable schedule(Class<T> type, CronJob<T> r, String cronExpression) throws Exception;
- Closeable schedule(RunnableWithException r, long first, long... ms) throws Exception
- Closeable schedule(RunnableWithException r, Duration first, Duration... duration) throws Exception

5.6 Callbacks

All callback methods must return in a reasonable time. An implementation may interrupt a callback when it takes too much time to return.

How much at minimum? Or shall we kill this?

5.7 Lifecycle

The following events can influence the life cycle of any scheduled callback:

- The Scheduler service is unget
- The Scheduler service is unregistered
- The Closeable of a schedule is closed

In a dynamic system like OSGi all these events can happen in any order and closely together. When these events happen an outstanding callback can become orphaned. Where orphaned is a situation where either the Scheduler service or the bundle that got the Scheduler service is no longer active.

There are then two cases:

- The callback is not running
- The callback is running

In the case the callback is running the Scheduler service must interrupt the callback. If the method does not return in reasonable amount of time then the Scheduler may use alternative means to reclaim the thread. In that case it must log a WARNING.

5.8 Exceptions

Any callback can throw exceptions. The Scheduler must log these exceptions as warnings.

Copyright © OSGi Alliance 2016
All Rights Reserved

5.9 Example Implementation OSGi enRoute

An implementation of this API can be found in the OSGi enRoute Bundles[4]. project. There is also an example project with a GUI[5]. that can be used to exercise the Scheduler

6 Data Transfer Objects

No data transfer objects are provided

7 Javadoc

Copyright © OSGi Alliance 2016
All Rights Reserved

OSGi Javadoc

5/19/16 8:38 PM

Copyright © OSGi Alliance 2016
All Rights Reserved

Package osgi.enroute.scheduler.api

Interface Summary		Page
CancellablePromise	A Cancellable Promise is a Promise that can be canceled.	18
CronJob	The software utility Cron is a time-based job scheduler in Unix-like computer operating systems.	19
RequireSchedulerImplementation	Require an implementation for the this specification	21
Scheduler	A Scheduler service provides timed semantics to Promises.	22
Scheduler.RunnableWithException	Convenience interface that is a Runnable but allows exceptions	28
SchedulerConstants	Specification of the JSONRPC API	29

Class Summary		Page
ShutdownException	Singleton exception thrown when a scheduler is shutdown and kills the tasks	30
TimeoutException	The singleton exception thrown when a task times out	31

Exception Summary		Page
CancelException	This is a singleton exception instance to mark a canceled promise.	17

Class CancelException

osgi.enroute.scheduler.api

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
              └osgi.enroute.scheduler.api.CancelException
```

All Implemented Interfaces:

Serializable

```
public class CancelException
extends RuntimeException
```

This is a singleton exception instance to mark a canceled promise.

Field Summary		Page
static	**SINGLETON** The singleton Cancel Exception instance	17

Field Detail

SINGLETON

```
public static SINGLETON
```

The singleton Cancel Exception instance

Interface CancellablePromise

osgi.enroute.scheduler.api

All Superinterfaces:
org.osgi.util.promise.Promise

```
public interface CancellablePromise
extends org.osgi.util.promise.Promise
```

A Cancellable Promise is a Promise that can be canceled. This will fail it with the `CancelException`, which is a singleton.

Method Summary		Page
boolean	**cancel**() Cancel this promise (fail it with `CancelException`, which is a singleton).	18

Methods inherited from interface org.osgi.util.promise.Promise
fallbackTo, filter, flatMap, getFailure, getValue, isDone, map, onResolve, recover, recoverWith, then, then

Method Detail

cancel

```
boolean cancel()
```

Cancel this promise (fail it with `CancelException`, which is a singleton).

Returns:
true if this promise is canceled or false if it already was canceled.

Copyright © OSGi Alliance 2016 All Rights Reserved

Interface CronJob

osgi.enroute.scheduler.api

`public interface CronJob`

The software utility Cron is a time-based job scheduler in Unix-like computer operating systems. People who set up and maintain software environments use cron to schedule jobs (commands or shell scripts) to run periodically at fixed times, dates, or intervals. It typically automates system maintenance or administration—though its general-purpose nature makes it useful for things like connecting to the Internet and downloading email at regular intervals. [1] The name cron comes from the Greek word for time, χρόνος chronos.

The Unix Cron defines a syntax that is used by the Cron service. A user should register a Cron service with the `CRON` property. The value is according to the {link http://en.wikipedia.org/wiki/Cron}.

```
 * * * * * * *
 | │ │ │ │ │ |
 | │ │ │ │ │ └ year (optional)
 | │ │ │ │ └── day of week from Monday (1) to Sunday (7).
 | │ │ │ └──── month (1 - 12) from January (1) to December (12).
 | │ │ └────── day of month (1 - 31)
 | │ └──────── hour (0 - 23)
 | └────────── min (0 - 59)
 └──────────── sec (0-59)

 Field name   mandatory   Values             Special characters
 Seconds       Yes         0-59               * / , -
 Minutes      Yes         0-59            * / , -
 Hours      Yes         0-23            * / , -
 Day of month Yes         1-31             * / , - ? L W
 Month      Yes         1-12 or JAN-DEC     * / , -
 Day of week Yes         1-7 or MON-SUN     * / , - ? L #
 Year         No          1970-2099        * / , -
```

Asterisk (*)

The asterisk indicates that the cron expression matches for all values of the field. E.g., using an asterisk in the 4th field (month) indicates every month.

Slash (/)

Slashes describe increments of ranges. For example 3-59/15 in the 1st field (minutes) indicate the third minute of the hour and every 15 minutes thereafter. The form "*\/..." is equivalent to the form "first-last/...", that is, an increment over the largest possible range of the field.

Comma (,)

Commas are used to separate items of a list. For example, using "MON,WED,FRI" in the 5th field (day of week) means Mondays, Wednesdays and Fridays. Hyphen (-) Hyphens define ranges. For example, 2000-2010 indicates every year between 2000 and 2010 AD, inclusive.

Additionally, you can use some fixed formats:

Copyright © OSGi Alliance 2016 All Rights Reserved

```
 @yearly (or @annually) Run once a year at midnight on the morning of Januar
y 1 0 0 1 1 *
 @monthly Run once a month at midnight on the morning of the first day of th
e month 0 0 1 * *
 @weekly Run once a week at midnight on Sunday morning 0 0 * * 0
 @daily Run once a day at midnight 0 0 * * *
 @hourly Run once an hour at the beginning of the hour 0 * * * *
 @reboot Run at startup @reboot (at service registration time)
```

Please not that for the constants we follow the Java 8 Date & Time constants. Major difference is the day number. In Quartz this is 0-6 for SAT-SUN while here it is 1-7 for MON-SUN.

Field Summary		*Page*
	CRON The service property that specifies the cron schedule.	20

Method Summary		*Page*
`void`	**run**(Object data) Run a cron job.	20

Field Detail

CRON

`public static final CRON = "cron"`

The service property that specifies the cron schedule. The type is String+.

Method Detail

run

```
void run(Object data)
         throws Exception
```

Run a cron job.

Parameters:

`data` - The data for the job

Throws:

`Exception`

Copyright © OSGi Alliance 2016 All Rights Reserved

Interface RequireSchedulerImplementation

osgi.enroute.scheduler.api

All Superinterfaces:
Annotation

```
public interface RequireSchedulerImplementation
extends Annotation
```

Require an implementation for the this specification

Copyright © OSGi Alliance 2016 All Rights Reserved

Interface Scheduler

osgi.enroute.scheduler.api

```
public interface Scheduler
```

A Scheduler service provides timed semantics to Promises. A Scheduler can delay a promise, it can resolve a promise at a certain time, or it can provide a timeout to a promise.

This scheduler has a millisecond resolution.

Nested Class Summary		Page
static interface	**Scheduler.RunnableWithException** Convenience interface that is a Runnable but allows exceptions	28

Method Summary		Page
CancellablePromise	**after**(Duration d) Convenience method to use a Duration instead of a millisecond delay.	26
CancellablePromise	**after**(Callable call, Duration d) Convenience method to to a duration and a callable.	27
CancellablePromise	**after**(Callable call, long ms) Return a promise that resolves after delaying ms with the result of the call that is executed after the delay.	23
CancellablePromise	**after**(long ms) Return a promise that will resolve after the given number of milliseconds.	23
CancellablePromise	**after**(Scheduler.RunnableWithException r, long ms) Convenience method to to a duration and a RunnableWithException.	27
CancellablePromise	**at**(Instant instant) Convenience method to use an Instant.	25
CancellablePromise	**at**(Callable callable, Instant instant) Convenience method to use an instant and a Callable.	26
CancellablePromise	**at**(Callable callable, long epochTime) Return a promise that resolves at the given epochTime with the result of the call.	24
CancellablePromise	**at**(long epochTime) Return a promise that resolves at the given epochTime	23
CancellablePromise	**at**(Scheduler.RunnableWithException r, Instant instant) Convenience method to use an instant and a RunnableWithException.	26
CancellablePromise	**at**(Scheduler.RunnableWithException r, long epochMilli) Convenience method to use an instant and a RunnableWithException.	26
CancellablePromise	**before**(org.osgi.util.promise.Promise promise, long timeout) Return a cancellable promise that fails with a TimeoutException when the given promise is not resolved before the given timeout.	25

Copyright © OSGi Alliance 2016 All Rights Reserved

Closeable	**schedule**(Class type, CronJob r, String cronExpression) Schedule a runnable to be executed for the give cron expression (See CronJob).	24
Closeable	**schedule**(Scheduler.RunnableWithException r, String cronExpression) Schedule a runnable to be executed for the give cron expression (See CronJob).	24
Closeable	**schedule**(Scheduler.RunnableWithException r, Duration first, Duration[] duration) Convenience method to use durations instead of milliseconds.	27
Closeable	**schedule**(Scheduler.RunnableWithException r, long first, long[] ms) Schedule a runnable to be executed in a loop.	25

Method Detail

after

`CancellablePromise` **`after`**`(long ms)`

Return a promise that will resolve after the given number of milliseconds. This promise can be canceled.

Parameters:
`ms` - Number of milliseconds to delay
Returns:
A cancellable Promise

---

after

```
CancellablePromise after(Callable call,
                         long ms)
```

Return a promise that resolves after delaying ms with the result of the call that is executed after the delay.

Parameters:
`call` - provides the result
`ms` - Number of ms to delay
Returns:
A cancellable Promise

---

at

`CancellablePromise` **`at`**`(long epochTime)`

Return a promise that resolves at the given epochTime

Parameters:
`epochTime` - The Java (System.currentMillis) time
Returns:
A cancellable Promise

Copyright © OSGi Alliance 2016 All Rights Reserved

at

```
CancellablePromise at(Callable callable,
                      long epochTime)
```

Return a promise that resolves at the given epochTime with the result of the call.

Parameters:
`callable` - provides the result
`epochTime` - The Java (System.currentMillis) time

Returns:
A cancellable Promise

schedule

```
Closeable schedule(Scheduler.RunnableWithException r,
                   String cronExpression)
            throws Exception
```

Schedule a runnable to be executed for the give cron expression (See `CronJob`). Every time when the cronExpression matches the current time, the runnable will be run. The method returns a closeable that can be used to stop scheduling. This variation does not take an environment object.

Parameters:
`r` - The Runnable to run
`cronExpression` - A Cron Expression

Returns:
A closeable to terminate the schedule

Throws:
`Exception`

schedule

```
Closeable schedule(Class type,
                   CronJob r,
                   String cronExpression)
            throws Exception
```

Schedule a runnable to be executed for the give cron expression (See `CronJob`). Every time when the cronExpression matches the current time, the runnable will be run. The method returns a closeable that can be used to stop scheduling. The run metjod of r takes an environment object. An environment object is a custom interface where the names of the methods are the keys in the properties (see `osgi.enroute.dto.api.DTOs`).

Parameters:
`type` - The data type of the parameter for the cron job
`r` - The Runnable to run
`cronExpression` - A Cron Expression

Returns:
A closeable to terminate the schedule

Throws:
`Exception`

Copyright © OSGi Alliance 2016 All Rights Reserved

schedule

```
Closeable schedule(Scheduler.RunnableWithException r,
                   long first,
                   long[] ms)
            throws Exception
```

Schedule a runnable to be executed in a loop. The first time the first is as delay, later the values in ms are used sequentially. If no more values are present, the last value is re-used. The method returns a closeable that can be used to stop scheduling. This is a fixed rate scheduler. That is, a base time is established when this method is called and subsequent firings are always calculated relative to this start time.

Parameters:

`r` - The Runnable to run
`first` - The first time to use
`ms` - The subsequent times to use.

Returns:

A closeable to terminate the schedule

Throws:

`Exception`

---

before

```
CancellablePromise before(org.osgi.util.promise.Promise promise,
                          long timeout)
```

Return a cancellable promise that fails with a `TimeoutException` when the given promise is not resolved before the given timeout. If the given promise fails or is resolved before the timeout then the returned promise will be treated accordingly. The cancelation does not influence the final result of the given promise since a Promise can only be failed or resolved by its creator.

If the timeout is in the past then the promised will be resolved immediately

Parameters:

`promise` - The promise to base the returned promise on
`timeout` - The number of milliseconds to wait.

Returns:

A cancellable Promise

---

at

```
CancellablePromise at(Instant instant)
```

Convenience method to use an Instant. See `at(long)`

Parameters:

`instant` - The instant for the time

Returns:

a cancellable promise that is resolved when the instant has been reached

---

Copyright © OSGi Alliance 2016 All Rights Reserved

at

```
CancellablePromise at(Callable callable,
                      Instant instant)
```

Convenience method to use an instant and a Callable. See `at(Callable, long)`.

Parameters:

`callable` - will be called when instant is reached

Returns:

a cancellable promise that is resolved when the instant has been reached and the callable has been called

at

```
CancellablePromise at(Scheduler.RunnableWithException r,
                      Instant instant)
```

Convenience method to use an instant and a RunnableWithException. See `at(RunnableWithException, long)`.

Parameters:

`r` - the runnable with exception to call when instant has been reached

`instant` - the time to run r

Returns:

A cancellable promise

at

```
CancellablePromise at(Scheduler.RunnableWithException r,
                      long epochMilli)
```

Convenience method to use an instant and a RunnableWithException. See `at(RunnableWithException, long)`.

Parameters:

`r` - the runnable with exception to call when instant has been reached

`epochMilli` - the time at which to run in epoch time

Returns:

A cancellable promise

after

```
CancellablePromise after(Duration d)
```

Convenience method to use a Duration instead of a millisecond delay. See `after(long)`.

Parameters:

`d` - the duration to wait

Returns:

A cancellable promise

Copyright © OSGi Alliance 2016 All Rights Reserved

after

```
CancellablePromise after(Callable call,
                         Duration d)
```

Convenience method to to a duration and a callable. See `after(Callable, long)`.

Parameters:

`call` - the callable to call

`d` - the duration to wait

Returns:

A cancellable promise

after

```
CancellablePromise after(Scheduler.RunnableWithException r,
                         long ms)
```

Convenience method to to a duration and a RunnableWithException. See `after(Callable, long)`.

Parameters:

`r` - the runnable with exception to call when instant has been reached

`ms` - the time to wait in milliseconds

Returns:

A cancellable promise

schedule

```
Closeable schedule(Scheduler.RunnableWithException r,
                   Duration first,
                   Duration[] duration)
            throws Exception
```

Convenience method to use durations instead of milliseconds. See `schedule(RunnableWithException, long, long...)`

Parameters:

`r` - the runnable to run after each duration

`first` - the first duration

`duration` - subsequent durations

Returns:

A cancellable promise

Throws:

`Exception`

Copyright © OSGi Alliance 2016 All Rights Reserved

Interface Scheduler.RunnableWithException

osgi.enroute.scheduler.api

Enclosing class:
Scheduler

```
public static interface Scheduler.RunnableWithException
```

Convenience interface that is a Runnable but allows exceptions

---

Method Summary		Page
void	run()	28

Method Detail

run

```
void run()
         throws Exception
```

Throws:
Exception

Copyright © OSGi Alliance 2016 All Rights Reserved

Interface SchedulerConstants

osgi.enroute.scheduler.api

`public interface` **`SchedulerConstants`**

Specification of the JSONRPC API

Field Summary		Page
	SCHEDULER_SPECIFICATION_NAME Name	29
	SCHEDULER_SPECIFICATION_VERSION Version	29

Field Detail

SCHEDULER_SPECIFICATION_NAME

`public static final` **`SCHEDULER_SPECIFICATION_NAME`** `= "osgi.enroute.scheduler"`

Name

SCHEDULER_SPECIFICATION_VERSION

`public static final` **`SCHEDULER_SPECIFICATION_VERSION`** `= "1.0.0"`

Version

Copyright © OSGi Alliance 2016 All Rights Reserved

Class ShutdownException

osgi.enroute.scheduler.api

```
java.lang.Object
  └java.lang.Throwable
      └osgi.enroute.scheduler.api.ShutdownException
```

All Implemented Interfaces:

Serializable

```
public class ShutdownException
extends Throwable
```

Singleton exception thrown when a scheduler is shutdown and kills the tasks

---

Field Summary		Page
static	**SINGLETON** The singleton exception	30

Field Detail

SINGLETON

`public static SINGLETON`

The singleton exception

Copyright © OSGi Alliance 2016 All Rights Reserved

Class TimeoutException

osgi.enroute.scheduler.api

```
java.lang.Object
  └java.lang.Throwable
      └osgi.enroute.scheduler.api.TimeoutException
```

All Implemented Interfaces:

Serializable

```
public class TimeoutException
extends Throwable
```

The singleton exception thrown when a task times out

Field Summary		Page
static	**SINGLETON** The singleton timeout exception	31

Field Detail

SINGLETON

```
public static SINGLETON
```

The singleton timeout 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 Security Considerations

The Scheduler must run the callbacks in an environment that does not restrict their permission. This generally means the Scheduler must have All Permission.

The usage of the scheduler can be restricted with Service Permission.

9 Document Support

9.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. http://www.quartz-scheduler.org/documentation/quartz-2.1.x/tutorials/crontrigger.html

[4]. https://github.com/osgi/osgi.enroute.bundles/tree/master/osgi.enroute.scheduler.simple.provider

[5]. https://github.com/osgi/osgi.enroute.examples/tree/master/osgi.enroute.examples.scheduler.application

9.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St Drezery
Voice	+33633982260
e-mail	Peter.Kriens@aQute.biz

9.3 Acronyms and Abbreviations

9.4 End of Document

Copyright © OSGi Alliance 2016 All Rights Reserved