APS Tools Library

User Guide

Version: 0.9.2

Author: Tommy Svensson

Table of Contents

1 APSToolsLib

This is a library of utilities including a service tracker that beats the (beep) out of the default one including exception rather than null response, timeout specification, getting a proxied service implementation that automatically uses the tracker, allocating a service, calling it, and deallocating it again. This makes it trivially easy to handle a service being restarted or redeployed. It also includes a logger utility that will lookup the standard log service and log to that if found.

This bundle provides no services. It just makes all its packages public. Every bundle included in APS makes use of APSToolsLib so it must be deployed for things to work.

Please note that this bundle has no dependencies! That is, it can be used as is without requireing any other APS bundle.

1.1 APSServiceTracker

This does the same thing as the standard service tracker included with OSGi, but does it better with more options and flexibility. One of the differences between this tracker and the OSGi one is that this throws an *APSNoServiceAvailableException* if the service is not available. Personally I think this is easier to work with than having to check for a null result.

There are several variants of constructors, but here is an example of one of the most used ones within the APS services:

```
    APSServiceTracker<Service> tracker = new APSServiceTracker<Service>(context,
Service.class, ”20 seconds”);
    tracker.start();
```

Note that the third argument, which is a timeout can also be specified as an int in which case it is always in miliseconds. The string variant supports the a second word of ”sec[onds]” and ”min[utes]” which indicates the type of the first numeric value. ”forever” means just that and requires just one word. Any other second words than those will be treated as milliseconds. The APSServiceTracker also has a set of constants for the timeout string value:

```
    public static final String SHORT_TIMEOUT = "3 seconds";
    public static final String MEDIUM_TIMEOUT = "30 seconds";
    public static final String LARGE_TIMEOUT = "2 minutes";
    public static final String VERY_LARGE_TIMEOUT = "5 minutes";
    public static final String HUGE_LARGE_TIMEOUT = "10 minutes";
    public static final String NO_TIMEOUT = "forever";
```

On bundle stop you should do:

```
    tracker.stop(context);
```

So that the tracker unregisters itself from receiving bundle/service events.

1.1.1 Services and active service

The tracker tracks all instances of the service being tracked. It however have the notion of an active service. The active service is the service instance that will be returned by allocateService() (which is internally used by all other access methods also). On startup it will be the first service instance received. It will keep tracking other instances comming in, but as long as the active service does not go away it will be the one used. If the active service goes away then the the one that is at the beginning of the list of the other tracked instances will become active. If that list is empty

there will be no active, which will trigger a wait for a service to become available again if allocateService() is called.

1.1.2 Providing a logger

You can provide an APSLogger (see further down about APSLogger) to the tracker:

```
    tracker.setLogger(apsLogger);
```

When available the tracker will log to this.

1.1.3 Tracker as a wrapped service

The tracker can be used as a wrapped service:

```
    Service service = tracker.getWrappedService();
```

This gives you a proxied *service* instance that gets the real service, calls it, releases it and return the result. This handles transparently if a service has been restarted or one instance of the service has gone away and another came available. It will wait for the specified timeout for a service to become available and if that does not happen the *APSNoServiceAvailableException* will be thrown. This is of course a runtime exception which makes the service wrapping possible without loosing the possibility to handle the case where the service is not available.

1.1.4 Using the tracker in a similar way to the OSGi standard tracker

To get a service instance you do:

```
    Service service = tracker.allocateService();
```

Note that if the tracker has a timeout set then this call will wait for the service to become available if it is currently not available until an instance becomes available or the timeout time is reached. It will throw *APSNoServiceAvailableException* on failure in any case.

When done with the service do:

```
    tracker.releaseService();
```

1.1.5 Accessing a service by tracker callback

There are a few variants to get a service instance by callback. When the callbacks are used the actual service instance will only be allocated during the callback and then released again.

1.1.5.1 onServiceAvailable

This will result in a callback when any instance of the service becomes available. If there is more than one service instance published then there will be a callback for each.

```
        tracker.onServiceAvailable(new OnServiceAvailable<Service>() {
            @Override
            public void onServiceAvailable(Service service, ServiceReference
serviceReference) throws Exception {
                // Do something.
            }
        });
```

1.1.5.2 onServiceLeaving

This will result in a callback when any instance of the service goes away. If there is more than one service instance published the there will be a callback for each instance leaving.

```
        onServiceLeaving(new OnServiceLeaving<Service>() {

            @Override
            public void onServiceLeaving(ServiceReference service, Class serviceAPI)
throws Exception {
                // Handle the service leaving.
            }
        });
```

Note that since the service is already gone by this time you don't get the service instance, only its reference and the class representing its API. In most cases both of these parameters are irellevant.

1.1.5.3 onActiveServiceAvailable

This does the same thing as onServiceAvailable() but only for the active service. It uses the same *OnServiceAvailable* interface.

1.1.5.4 onActiveServiceLeaving

This does the same thing as onServiceLeaving() but for the active service. It uses the same *OnServiceLeaving* interface.

1.1.5.5 withService

Runs the specified callback providing it with a service to use. This will wait for a service to become available if a timeout has been provided for the tracker.

Don't use this in an activator start() method! onActiveServiceAvailable() and onActiveServiceLeaving() are safe in a start() method, this is not!

```
        tracker.withService(new WithService<Service>() {
            @Override
            public void withService(Service service, Object... args) throws Exception {
                // do something here.
            }
        }, arg1, arg2);
```

If you don't have any arguments this will also work:

```
        tracker.withService(new WithService<Service>() {
            @Override
            public void withService(Service service) throws Exception {
                // do something here
            }
        });
```

1.1.5.6 withServiceIfAvailable

This does the same as withService(...) but without waiting for a service to become available. If the service is not available at the time of the call the callback will not be called. No exception is thrown by this!

1.1.5.7 withAllAvailableServices

This is used exactly the same way as withService(...), but the callback will be done for each tracked service instance, not only the active.

1.2 APSLogger

This provides logging functionality. The no args constructor will log to System.out by default. The OutputStream constructor will logg to the specified output stream by default.

The APSLogger can be used by just creating an instance and then start using the info(...), error(...), etc methods. But

in that case it will only log to System.out or the provided OutputStream. If you however do this:

```
    APSLogger logger = new APSLogger();
    logger.start(context);
```

then the logger will try to get hold of the standard OSGi LogService and if that is available log to that. If the log service is not available it will fallback to the OutputStream.

If you call the `setServiceRefrence(serviceRef);` method on the logger then information about that service will be provied with each log.

1.3 APSActivator

This is a BundleActivator implementation that uses annotations to register services and inject tracked services. Any bundle can use this activator by just importing the *se.natusoft.osgi.aps.tools* package.

This is actually a trivial not very large class that just scans the bundle for files ending in *.class*, removes the *.class* from the path and translates the separator char to '.' and then passes it to bundle.getClass(...) to get the Class instance for it. After that it can inspect the bundles all classes for annotations and act on them. Most methods are protected making it easy to subclass this class and expand on its functionality.

The following annotations are available:

@APSOSGiServiceProvider - This should be specified on a class that implements a service interface and should be registered as an OSGi service. *Please note* that the first declared implemented interface is used as service interface!

```
    public @interface OSGiProperty {
        String name();
        String value();
    }

    public @interface APSOSGiServiceProvider {

        /** Extra properties to register the service with. */
        OSGiProperty[] properties() default {};
    }
```

@APSOSGiService - This should be specified on a field having a type of a service interface to have a service of that type injected, and continuously tracked. Any call to the service will throw an APSNoServiceAvailableException (runtime) if no service has become available before the specified timeout. It is also possible to have APSServiceTracker as field type in which case the underlying configured tracker will be injected instead.

```
    public @interface APSOSGiService {

        /** The timeout for a service to become available. Defaults to 30 seconds. */
        String timeout() default "30 seconds";

        /** Any additional search criteria. Should start with '(' and end with ')'.
Defaults to none. */
        String additionalSearchCriteria() default "";
    }
```

@APSInject - This will have an instance injected. There will be a unique instance for each name specified with the default name of "default" being used in none is specified. There are 2 field types handled specially: BundleContext and APSLogger. A BundleContext field will get the bundles context injected. For an APSLogger instance the 'loggingFor' annotation property can be specified. Please note that any other type must have a default constructor to be instantiated and injected!

```
    public @interface APSInject {
        /**
         * The name of the instance to inject. If the same is used in multiple classes
the same instance will
         * be injected.
         */
        String name() default "default";

        /**
         * A label indicating who is logging. If not specified the bundle name will be
used. This is only
         * relevant if the injected type is APSLogger.
         */
        String loggingFor() default "";
    }
```

@APSBundleStart - This should be used on a method and will be called on bundle start. The method should take no arguments. If you need a BundleContext just inject it with @APSInejct. The use of this annotation is only needed for things not supported by this activator. Please note that a method annotated with this annotation can be static (in which case the class it belongs to will not be instantiaded -- due to this!). You can provide this annotation on as many methods in as many classes as you want. They will all be called (in the order classes are discovered in the bundle).

```
    public @interface APSBundleStart {}
```

@APSBundleStop - This should be used on a method and will be called on bundle stop. The method should take no arguments. This should probably be used if @APSBundleStart is used. Please note that a method annotated with this annotation can be static!

```
    public @interface APSBundleStop {}
```

1.4 APSContextWrapper

This provides a static wrap(...) method:

```
    Service providedService = APSContextWrapper.wrap(serviceProvider, Service.class);
```

where *serviceProvider* is an instance of a class that implements *Service*. The resulting instance is a java.lang.reflect.Proxy implementation of *Service* that ensures that the *serviceProvider* ClassLoader is the context class loader during each call to all service methods that are annotated with @APSRunInBundlesContext annotation in *Service*. The wrapped instance can then be registered as the OSGi service provider.

Normally the threads context class loader is the original service callers context class loader. For a web application it would be the web containers context class loader. If a service needs its own bundles class loader during its execution then this wrapper can be used.

1.5 ID generators

There is one interface:

```
    /**
     * This is a generic interface for representing IDs.
     */
    public interface ID extends Comparable<ID> {

        /**
         * Creates a new unique ID.
         *
         * @return A newly created ID.
         */
        public ID newID();
```

```
        /**
         * Tests for equality.
         *
         * @param obj The object to compare with.
         *
         * @return true if equal, false otherwise.
         */
        @Override
        public boolean equals(Object obj);

        /**
         * @return The hash code.
         */
        @Override
        public int hashCode();

    }
```

that have 2 implementations:

- IntID - Produces int ids.
- UUID - Produces java.util.UUID Ids.

1.6 Javadoc

The javadoc for this can be found at http://apidoc.natusoft.se/APSToolsLib/.