APS Tools Library

User Guide

1.0.0

Tommy Svensson

APSToolsLib

This is a library of utilities including a service tracker that is far better than the default one, including exception rather than null response on timeout, 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, otherwise just log to stdout.

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. It however requires APSOSGiTestTools to build, but that is only a test dependency.

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. I also think that trying to keep bundles and services up are better than pulling them down as soon as one depencency goes away for a short while, for example due to redeploy of newer version. This is why APSServiceTracker takes a timeout and waits for a service to come back before failing.

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.

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 the active service 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.

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.

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.

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();
```

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.

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.
    }
});
```

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.

onActiveServiceAvailable

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

onActiveServiceLeaving

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

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
    }
});
```

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!

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.

onTimeout (since 0.9.3)

This allows for a callback when the tracker times out waiting for a service. This callback will be called just before the *APSNoServiceAvailableException* is about to be thrown.

```
tracker.onTimeout(new OnTimeout() {
    @Override
    public void onTimeout() {
        // do something here
    }
});
```

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.

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 rather trivial class that just scans the bundle for classes and inspects all classes for annotations and act on them. Most methods are protected making it easy to subclass this class and expand on its functionality.

Please note that it does *class.getDeclaredFields()* and *class.getDeclaredMethods()*! This means that it will only see the bottom class of an inheritance hiearchy!

The following annotations are available:

@OSGiServiceProvider - 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 unless you specify serviceAPIs={Svc.class, ...}.

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

public @interface OSGiServiceInstance {

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

    /**
     * The service API to register instance with. If not specified the first
     * implemented interface will be used.
     */
    Class[] serviceAPIs() default {};
}
```

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

    /**
     * The service API to register instance with. If not specified the first
     * implemented interface will be used.
     */
    Class[] serviceAPIs() default {};

    /**
     * This can be used as an alternative to properties() and also supports
     * several instances.
     */
    OSGiServiceInstance[] instances() default {};

    /**
     * An alternative to providing static information. This class will be
     * instantiated if specified and provideServiceInstancesSetup() will
     * be called to provide implemented service APIs, service properties,
     * and a service instance. In this last, it differs from
     * instanceFactoryClass() since that does not provide an instance.
     * This allows for more easy configuration of each instance.
     */
    Class<? extends APSActivatorServiceSetupProvider>
        serviceSetupProvider()
        default APSActivatorServiceSetupProvider.class;

    /**
     * This can be used as an alternative and will instantiate the
     * specified factory class which will deliver one set of
     * Properties per instance.
     */
    Class<? extends APSActivator.InstanceFactory> instanceFactoryClass()
        default APSActivator.InstanceFactory.class;

    /**
     * If true this service will be started in a separate thread.
     * This means the bundle start will continue in parallel and
     * that any failures in startup will be logged, but will
     * not stop the bundle from being started. If this is true
     * it wins over required service dependencies of the service
     * class. Specifying this as true allows you to do things that
     * cannot be done in a bunde activator start method, like
     * calling a service tracked by APSServiceTracker, without
     * causing a deadlock.
     */
    boolean threadStart() default false;
}
```

Do note that for the *serviceSetupProvider()* another solution is to use the *@BundleStart* (see below) and just create instances of your service and register them with the BundleContext. But if you use *@OSGiServiceProvider* to instantiate and register other "one instance" services, then using *serviceSetupProvider()* would look a bit more consistent.

@**APSExternalizabe**, @**APSRemoteService** - These 2 annotations are synonyms and have no properties. They should be used on a service implementation class. When either of these are specified the "aps-externalizable=true" property will be set when the service is registered with the OSGi container. The APSExternalProtocolExtender will react on this property and make the service externally accessible.

@**OSGiService** - 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.

If *required=true* is specified and this field is in a class annotated with *@OSGiServiceProvider* then the class will not be registered as a service until the service dependency is actually available, and will also be unregistered if the tracker for the service does a timeout waiting for a service to become

available. It will then be reregistered again when the dependent service becomes available again. Please note that unlike iPOJO the bundle is never stopped on dependent service unavailability, only the actual service is unregistered as an OSGi service. A bundle might have more than one service registered and when a dependency that is only required by one service goes away the other service is still available.

```
public @interface OSGiService {

    /**
     * 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 "";

    /**
     * This should specify a Class implementing
     * APSActivatorSearchCriteriaProvider. If specified it will
     * be used instead of additionalSearchCriteria() by
     * instantiating the Class and calling its method to get
     * a search criteria back. This allows for search criteria
     * coming from configuration, which a static annotation String
     * does not.
     */
    Class<? extends APSActivatorSearchCriteriaProvider>
        searchCriteriaProvider()
        default APSActivatorSearchCriteriaProvider.class;

    /**
     * If set to true the service using this service will not
     * be registered until the service becomes available.
     */
    boolean required() default false;
}
```

@**Managed** - This will have an instance managed and injected. There will be a unique instance for each name specified with the default name of "default" being used if 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 Managed {

    /**
     * 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 "";
}
```

@**ExecutorSvc** - This should always be used in conjunction with @Managed! This also assumes that the annotated field is of type ExecutorService or ScheduledExecutorService. This annotation provides some configuration for the ExecutorService that will be injected.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface ExecutorSvc {
```

```
        enum ExecutorType {
            FixedSize,
            WorkStealing,
            Single,
            Cached,
            Scheduled,
            SingleScheduled
        }

        /** This is loosely the number of concurrent threads. */
        int parallelism() default 10;

        /** The type of ExecutorService wanted. */
        ExecutorType type() default ExecutorType.FixedSize;

        /** If true the created ExecutorService will be wrapped with a delegate that disallows
        configuration. */
        boolean unConfigurable() default false;
    }
```

@**Schedule** - Schedules a Runnable using a ScheduledExecutionService. Indifferent from @ExecutorSvc this does not require an @Managed also, but do work with @Managed if that is used to inject an instance of Runnable to be scheduled. @Schedule is handled after all injections have been done.

```
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.FIELD)
    public @interface Schedule {

        /**
         * The defined executor service to schedule this on. This should be the name of it. If left
        blank an internal
         * ScheduledExecutorService will be used.
         */
        String on() default "";

        /** The amount of time to wait for the (first) execution. */
        long delay();

        /** If specified how long to wait between runs. */
        long repeat();

        /** The time unit used for the above values. Defaults to seconds. */
        TimeUnit timeUnit() default TimeUnit.SECONDS;

        /** Possibility to affect the size of the thread pool when such is created internally for
        this (on="..." not provided!). */
        int poolSize() default 2;
    }
```

@**BundleStart** - 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 *@Managed*. 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). 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 BundleStart {

        /**
         * If true the start method will run in a new thread.
         * Any failures in this case will not fail
         * the bundle startup, but will be logged.
         */
        boolean thread() default false;
    }
```

@**BundleStop** - 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 *@BundleStart* is used. Please note that a

method annotated with this annotation can be static!

```
public @interface BundleStop {}
```

Usage as BundleActivator

The *APSActivator* class has 2 constructors. The default constructor without arguments are used for BundleActivator usage. In this case you just specify this class as your bundles activator, and then use the annotations described above. Thats it!

Other Usage

Since the activator usage will manage and create instances of all annotated classes this will not always work in all situations. One example is web applications where the web container is responsible for creating servlets. If you specifiy APSActivator as an activator for a WAB bundle and then use the annotations in a servlet then APSActivator will have a managed instance of the servlet, but it will not be the same instance as the web contatiner will run.

Therefore APSActivator has another constructor that takes a vararg of instances: `public APSActivator(Object... instances)`. There is also a `public void addManagedInstance(Object instance)` method. These allow you to add an already existing instance to be managed by APSActivator. In addition to the provided existing instances it will still scan the bundle for classes to manage. It will however not double manage any class for which an existing instance of has already been provided. Any annotated class for which existing instances has not been provided will be instantiated by APSActivator.

Please note that if you create an instance of APSActivator in a servlet and provide the servlet instance to it and start it (you still need to do *start(BundleContext)* and *stop(BundleContext)* when used this way!), then you need to catch the close of the servlet and do *stop* then.

There are 2 support classes:

- [APSVaadinWebTools]: APSVaadinOSGiApplication - This is subclassed by your Vaading application.
- [APSWebTools]: APSOSGiSupport - You create an instance of this in a servlet and let your servlet implement the *APSOSGiSupportCallbacks* interface which is then passed to the constructor of APSOSGiSupport.

Both of these creates and manages an APSActivator internally and catches shutdown to take it down. They also provide other utilities like providing the BundleContext. See *APSWebTools* for more information.

APSActivatorPlugin

Any implementing classes of this interface can be specified in META-INF/services/se.natusoft.osgi.aps.tools.APSActivatorPlugin file, one per line. These are loaded by java.util.ServiceLoader. The implementation can be provided by another bundle which should then export the relevant packages which can then be imported in the using bundle.

The APSActivatorPlugin API looks like this:

```
public interface APSActivatorPlugin {

    interface ActivatorInteraction {
        void addManagedInstance(Object instance, Class forClass);
    }

    void analyseBundleClass(ActivatorInteraction activatorInteraction, Class bundleClass);
}
```

Be warned that this is currently very untested! No APS code uses this yet.

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.

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.

Javadoc

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