Classloading in Tuscany SCA Java

Classloading in Tuscany

⚠ **Note posted by Rajini**

Tuscany's use of classloaders doesn't seem to be well-defined, even though the concept of a runtime classLoader and contribution classloaders should have made it easy to isolate these namespaces. All Tuscany samples and tests are currently run with a single application classloader containing absolutely everything including Tuscany, all the jar files Tuscany requires and all the Java contribution jars/folders.

I am running Tuscany as a bundle inside an OSGi runtime, and came across some issues with classloading when multiple classloaders are used. Even though these can be fixed for OSGi without changes to the non-OSGi related code in Tuscany, it will be good to fix these properly in Tuscany since these issues affect anyone who wants to use Tuscany without a great big classloader hierarchy containing Tuscany, all its dependencies and all contributions.

Here is my understanding of classloaders used by Tuscany:

1) Tuscany Runtime classloader
This is the classloader used to load Tuscany itself. It should be able to locate all Tuscany jar files and all its dependent jar files (eg. axis2). There are many locations in Tuscany which explicitly read the classloader used to load one Tuscany class in order to load another Tuscany class. For example, many of the files defined in META-INF/services and the classes they refer to are loaded in this way.

2) Contribution classloaders
Referred to as application classloader in the DefaultSCADomain, a separate classloader can (in theory) be used to refer to the contributions - in the case of DefaultSCADomain, this is a single classloader associated with a single contribution.

There is currently no concept of a contribution classloader for each contribution (except for OSGi contributions), and all resolution of classes from contributions is done using a single classloader. This does not reflect the SCA specification, which requires contribution imports/exports to be explicitly specified - requiring multiple classloaders to enforce the import/export definition during class resolution.

3) Thread context classloader
Tuscany sets thread context classloader only in one class - HotUpdatableSCADomain (which sets Context classloader to a single URLClassLoader containing all the contribution jar files, with the original context classloader as parent).

Tuscany uses the thread context classloader in around 30 different locations. Around half of these have FIXMEs saying that the classloader should be passed in. Many of these uses of the context classloader are trying to obtain the runtime classloader. And at least one is trying to obtain the contribution classloader.

Many libraries which are used by Tuscany require the thread context classloader to be able to load the library classes. Most of these use the factory pattern and dynamically load implementation classes to create instances using the context classloader (eg. javax.xml.datatype.DataTypeFactory.
newInstance()). The context classloader should in these cases to be able to find classes in Tuscany's dependent jar files, and hence the classloader used to load these dependent jars should be
on the thread context classloader hierarchy.

Axis2 libraries (and possibly others) use the thread context classloader to load Tuscany classes. Axis2 needs to find Tuscany's message receivers org.
apache.tuscany.sca.binding.ws.axis2.Axis2ServiceInOutSyncMessageReceiver and org.apache.tuscany.sca.binding.ws.axis2.
Axis2ServiceInMessageReceiver, and
these are loaded by Axis2 using the context classloader. So Tuscany runtimeClassloader should be on the thread context classloader hierarchy.

4) Java Application classloader
Tuscany doesn't directly rely on the application classloader (the classloader corresponding to CLASSPATH). But since Tuscany doesn't set the thread context classloader, for any application that doesn't explicitly set thread context classloader, there is an indirect dependency on the application classloader, and hence CLASSPATH should contain everything required by the thread context classloader (which at the moment includes the entire world).

So we have the following requirements on classloaders:

1. Tuscany runtime classloader should find Tuscany and all its dependencies (essentially tuscany-sca-manifest.jar).
2. Contributions are not required to be on the classpath, nor do they have to be visible to the thread context classloader. One classloader per contribution is required to enforce SCA contribution import/export semantics.
3. Thread context classloader should be able to see Tuscany's dependencies and also Tuscany classes - so this can be the same as (or a child of) the Tuscany runtime classloader.
4. Any code using Tuscany should either have Tuscany runtime and its dependencies on the CLASSPATH, or should set the thread context classloader for its threads using Tuscany. All Tuscany tests rely on CLASSPATH, while Tuscany can be run inside OSGi without CLASSPATH, by setting the context classloader.

I would like to propose the following fixes to Tuscany classloading (my aim is to implement neater isolation without too much impact on the code):

1) Fix contribution classloading to enforce SCA contribution specification by using one classloader per contribution. Remove the requirement for contributions to be present in the classpath - Tuscany runtime classloader, thread context classloader and the application classloader should not be required to see the contribution classes.

2) Remove unnecessary usages of thread context classloader in Tuscany where the classloader that is required is the Tuscany Runtime classloader. It should be possible in many cases to determine the classloader directly without having to pass classloaders around. This is already done in some places, so this is just making the code consistent.

3) There are many uses of the thread context classloader where it is not very clear which classloader is actually required. I would suggest that as far as possible, Tuscany should avoid using the thread context classloader, and in places where it cannot be avoided without major changes, the code should be commented to show which classloader is required.

As an example (the FIXME is from the existing code)

```
public JAXBElement<?> create(ElementInfo element, TransformationContext context) {
    .....
    ClassLoader classLoader = context != null ? context.getClassLoader() : null;
    if (classLoader == null) {
        //FIXME Understand why we need this, the classloader should be passed in
        classLoader = Thread.currentThread().getContextClassLoader();
    }
    Class<?> factoryClass = Class.forName(factoryClassName, true, classLoader);
    ....
}
```

Given that Tuscany and its dependencies need to be visible from the thread context classloader anyway because other libraries need them there, this change is to merely to clean up Tuscany classloading and provide more clarity. It does not aim to remove the external dependencies on the thread context classloader.

4) It will be good to ensure that all uses of the Tuscany runtime classloader (eg. to load META-INF/services) continue to work even if each Tuscany module was loaded by a different classloader (ie, Tuscany runtime classloader is a hierarchy of module classloaders rather than a single one). This is not strictly necessary at the moment, but it is necessary if we want to package Tuscany modules as separate OSGi bundles.

5) Document Tuscany's requirements on classloaders (CLASSPATH and thread context classloader).

I would appreciate your feedback, both regarding whether there is better way to fix this, and on areas which I have missed out or misunderstood. I will be happy to provide a patch if this is an acceptable solution.

"Second note from Rajini

There are two sets of classloading in Tuscany that we need to look at and these can be handled independently of each other.
1) Classloading architecture for SCA contributions
2) Classloading architecture for SCA runtime modules

In both cases, there are two ways of improving modularity in Tuscany
a) Use separate classloaders per module to provide isolation, using a classloader architecture similar to OSGi, but without actually running in an OSGi runtime.
b) Use OSGi bundles to provide module isolation and versioning, where the versioning support would rely on an OSGi runtime.

For both 1) and 2), we could support a) and/or b).

Tuscany already supports 1b), and the proposed fixes will support 1a).

Tuscany does not implement 2a) or 2b), and if done properly, the implementation would required code changes that are pervasive across Tuscany. So the question is - do we really want to run Tuscany modules as separate OSGi bundles (or isolate modules using separate classloaders)? IMHO, it is only worth the hassle, if Tuscany modules are properly versioned and dynamically replaceable - ie, run as bundles in an OSGi runtime (2b). The proposed fixes do not implement 2a) and provide a partial solution for 2b) based on OSGi manifest files to minimize code changes to Tuscany.

1) Classloading architecture for SCA contributions

At the moment, OSGi bundle contributions can be used in Tuscany which provide modularity and versioning with the help of an OSGi runtime. Plain Jar contributions or folders use a single classloader at the moment and the proposed fixes will introduce contribution classloaders to isolate contributions.

SCA contributions can specify dependencies in terms of import/export statements in the same way as OSGi bundles. If you consider Java contributions, SCA contribution dependencies are a subset of OSGi bundle dependencies, because SCA does not support versioning or any of the other attributes that can be associated with importing packages in OSGi. So SCA contribution classloading will be a simpler version of OSGi bundle classloading. The code for supporting SCA import/export is already in place in Tuscany- it just doesn't get used because the thread context classloader is currently used to resolve classes.

2) Classloading architecture for SCA runtime modules

At the moment, Tuscany uses a single classloader for the runtime and its dependencies, and even though Tuscany runtime architecture uses extensible modules with reasonably well defined dependencies, module isolation is not achieved because of the use of a single classloader. Tuscany also has dependencies on many libraries which rely on the thread context classloader.

Even though we are looking at running Tuscany in an OSGi runtime to support distributed-OSGi, at the moment we are still assuming that by default Tuscany would be run without an OSGi runtime. Hence the proposed fixes aim to minimize Tuscany changes required to implement 2b). Tuscany's use of thread context classloader will be removed wherever possible, but the requirement on thread context classloader will not be removed altogether since it is used by libraries that Tuscany depends on.

There are two issues with modularizing Tuscany runtime using OSGi or OSGi-style classloading. Thread context classloaders and OSGi dont go very well together. And the extension module architecture used by Tuscany where a core module is extended through the use of extension modules ends up in a classloader hierarchy which is the reverse of what is required - the core bundle does not have visibility of the classes from the extension bundles.

There is an OSGi RFP which addresses classloading enhancements for OSGi including those required for applications relying on thread context classloaders. Possible solutions to implement 2b) now include the use of DynamicImport-Package, Eclipse buddy policy, Spring-OSGi Extender Model or OSGi services.

If we want to implement 2a) and 2b) properly at some point, it might make sense to provide a solution that uses a utility library to implement all classloading (which can then do something different based on whether Tuscany is running inside an OSGi runtime or outside). But the changes required for this will be pervasive across Tuscany code.

A note from Sebastien

How about going step by step and:
1. try to bootstrap the tuscany runtime with two classloaders: CL1 application code, CL2 runtime
2. extend to CL1 application code, CL2 Tuscany and SCA APIs, CL3 runtime
3. split the runtime in multiple CLs

and on a separate path:
1. try to bootstrap the tuscany runtime with two classloaders: CL1 application code, CL2 runtime
2. split the application code in multiple CLs

We could create integration tests for these configurations (not necessarily using OSGi, as these can be built with just plain classloaders IMO), and it would help us identify bad classloader usages, fix them, and detect+prevent classloader issues over time.

Thoughts?

–
Jean-Sebastien

"A note from Simon Nash"

I think Raymond's graph of dependencies was helpful in laying out the visbility relationships. There's also a counterpoint for what things should NOT be visible. In the following, "see" means that it's statically referenceable using the same classloader.

1. Application code shouldn't be able to see non-imported contributions, Tuscany SPIs, or Tuscany runtime code.

2. Tuscany APIs shouldn't be able to see anything else.

3. Tuscany SPIs shouldn't be able to see Tuscany runtime code or application code.

4. Tuscany runtime code shouldn't be able to see Tuscany runtime code in other modules, or application code.

5. 3rd party code (not written with knowledge of Tuscany) shouldn't be able to see Tuscany runtime code, Tuscany SPIs, Tuscany APIs, or application code.