

ClassMan

Introduction

The ClassMan toolkit integrated into Loom is a set of utility classes that enable ClassLoader hierarchies to be constructed from xml configurations. The toolkit supports hierarchial ClassLoaders and ClassLoaders defined by directed graphs via the use of "Join" ClassLoaders that can have multiple parents. This results in construction of a ClassLoader lattice.

Each non-Join ClassLoader can be defined in terms of;

- Entries: URLs designating either a directory or a file
- FileSets: Sets of files defined in a manner similar to Ants Filesets.
- Extensions: Definitions of Extensions, aka "Optional Packages".

Each ClassLoader also has a name and a parent. The parent is the name of the parent ClassLoader. Usually the parent ClassLoaders are one of the predefined ClassLoaders. The predefined are passed into the ClassMan toolkit from external application code.

The predefined ClassLoaders are generally named according to a pattern that places the "**at start and end of name. ie `**myPredefinedClassLoader`**". Loom predefines the following classloaders.

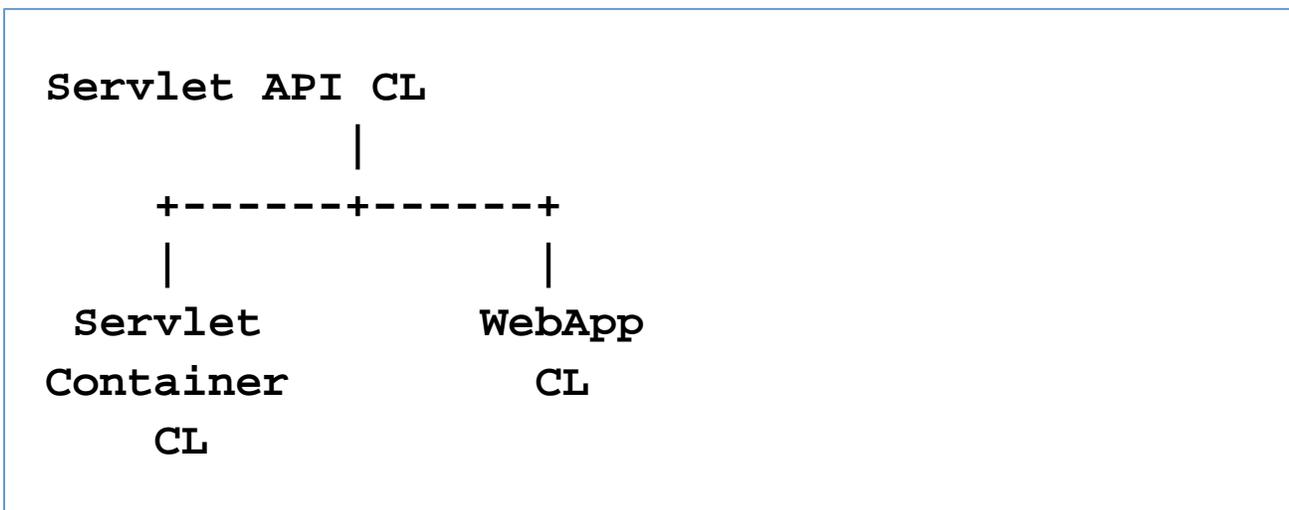
- **system**: The System ClassLoader.
- **common**: Common between container and application code.
- **shared**: Shared between all application code. Note: This is currently equivelent to **common**.

The commented [DTD](#) describes the descriptor format explicitly.

Loom Sample

Let us also assume that we want to host a servlet container (like Catalina, Jo! or Jetty) in Loom. The servlet specification requires that the servlets are capable of "seeing" the servlet API but recomends strongly that no servlet should be able to access any container specific classes.

To satisfy this requirement we decided to place the Servlet API classes in a parent ClassLoader to the Containers ClassLoader and each Web Applications ClassLoader. ie



This way, both the Container and the WebApp ClassLoaders will load the Servlet API from the same ClassLoader.

Unfortunately, in our case Loom already assembles the Servlet Container CL by default and does not give us the opportunity to construct the Servlet API CL as a parent ClassLoader. Luckily we can override this using the ClassMan toolkit using the following configuration file.

```
<classloaders default="container"
version="1.0">

  <!-- needed to run under earlier JVMs that
do not include JNDI -->
  <classloader name="jndi-api"
parent="*system*">
    <entry
location="sar:SAR-INF/ext/jndi.jar"/>
  </classloader>

  <!--
    The actual Servlet API classLoader. Note
that this does not specify
    a physical location but instead defines
an extension. This allows
    the container to search for the library
that best satisfies this
    extension. Usually all the extensions
are stored in a central directory
    and Loom will search through the jars in
central to find the servlet
    jar. This allows several applications to
share the same jar.
  -->
  <classloader name="servlet-api"
parent="*system*">
    <extension>
```

```
<name>javax.servlet</name>
```

```
<specification-version>2.3</specification-version>
```

```
<vendor-id>org.apache.jakarta</vendor-id>
```

```
<vendor-version>1.2.3.4</vendor-version>
```

```
</extension>
```

```
</classloader>
```

```
<!--
```

This is a special ClassLoader that merges two other

ClassLoaders together. When you try to load a class from

this ClassLoader, the ClassLoader will first try to load

the class from servlet-api ClassLoader and then try to

load the class from the jndi-api ClassLoader. This works

fine if the ClassLoaders define disjoint sets of classes.

ie No class should be loadable from both the servlet-api

ClassLoader and the jndi-api ClassLoader (with the exception

of Classes Loaded from System ClassLoader).

```
-->
```

```
<join name="webapp-common">
  <classloader-ref name="servlet-api"/>
  <classloader-ref name="jndi-api"/>
</join>
```

```
<!--
```

This classloader is needed to join the Loom API

and the Servlet API into one ClassLoader. This is needed

because the container is built using Loom APIs

but needs to share the Servlet APIs with the WebApps.

```
-->
```

```
<join name="container-base">
  <classloader-ref name="webapp-common"/>
  <classloader-ref name="*common*" />
</join>
```

```
<!--
```

This classloader is the one used to actually load the

Servlet Container. We know this as it is specified as the

default ClassLoader in <classloaders/> element.

```
-->
```

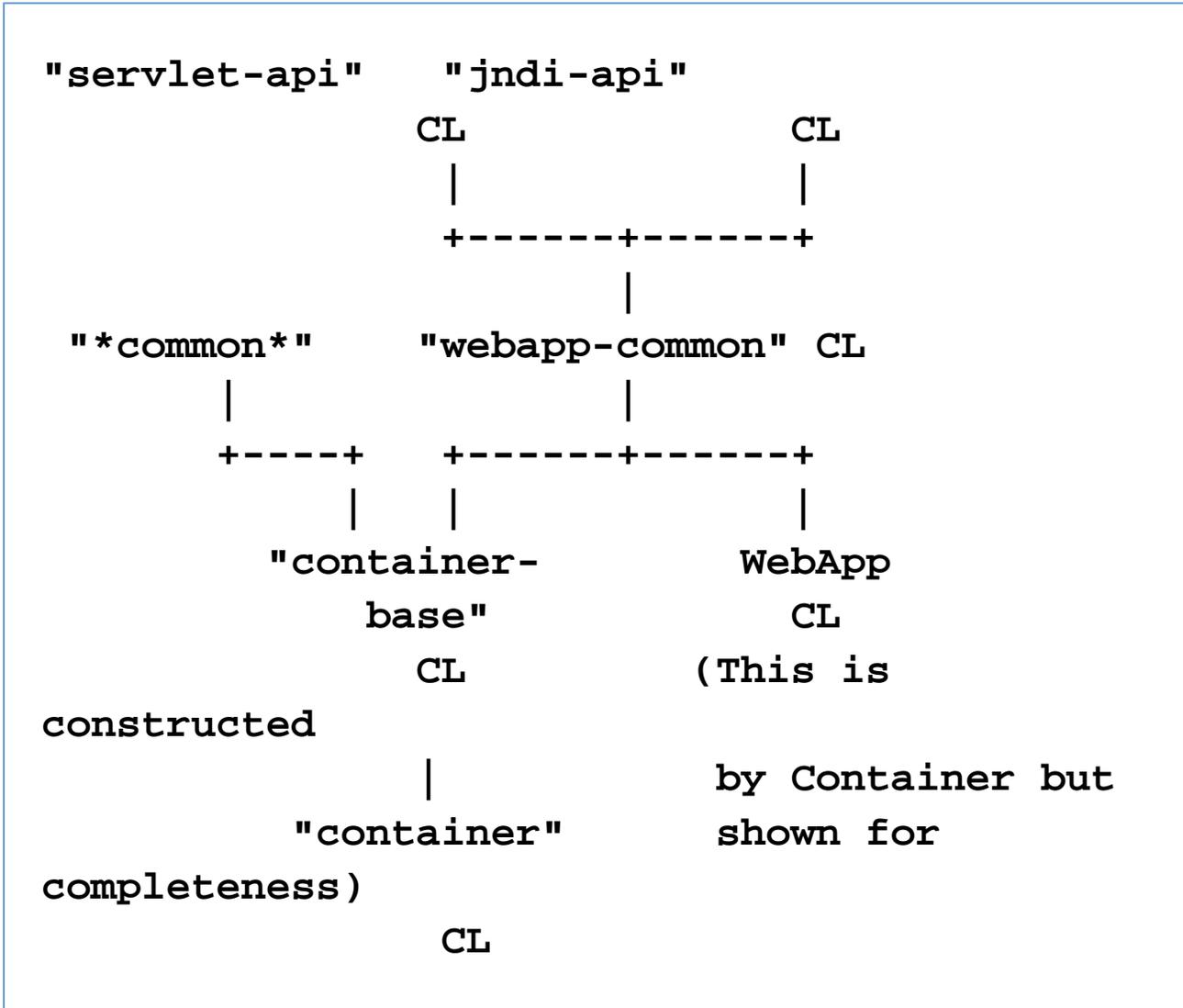
```
<classloader name="container"
parent="container-base">
  <entry location="sar:SAR-INF/classes/" />
```

```
<fileset dir="sar:SAR-INF/lib/">  
  <include name="*.jar"/>  
</fileset>
```

```
</classloader>
```

```
</classloaders>
```

The first thing you notice about this is that the ClassLoader hierarchy is much more complicated. In fact the diagram now looks like;



In reality we could have merged "servlet-api" and "jndi-api" into "webapp-common" but we separated them for illustration purposes.

The above demonstrates one of the most complex examples that you are likely to come across. This arose because there was multiple "containers" hosted in same ClassLoader hierarchy. The Servlet API specification requires that implementation classes not be visible to API clients. The Loom API specification requires the same thing.