Starting and stopping a container

Table of Contents

This tutorial demonstrates how to use the Cargo Maven2 plugin to automatically start/stop a container (possibly deploying some deployables to it as it starts).

- Configuration
- Configuring an Embedded Jetty Container
- Setting the JAVA_HOME for the container
- Automatic deployment of project's artifact (for J2EE projects)
- Automatically executing and stopping the container when running mvn install
- Adding JARs to the container's classpath
- Adding JDBC DataSources to the container's JNDI tree
- Adding POJO or JavaMail Resources to the container's JNDI tree

> **Like to learn by example? Take a look at our archetypes**
>
> We have several Maven2 Archetypes that contain sample Maven2/Maven3 projects with different use cases for the CARGO plugin, we would really recommend that you check them out. For more details, read here: Maven2 Archetypes.

Configuration

Example of a minimalist configuration:

```
[...]
  <build>
    <plugins>
      <plugin>

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
[...]
```

Yes, you've read it right, there's no `<configuration>` element in this example! When you use this setup the

Cargo m2 plugin will use a Jetty container by default. You can start the container with `mvn cargo:start` and stop it with `mvn cargo:stop`.

> **Wait after the container has started**
>
> If you want Cargo to start the container and "wait" (i.e., do not stop it right afterwards, so that you can for example to manual tests), please run the following command:
>
> ```
> mvn cargo:run
> ```
>
> Cargo will then start the container and show the following message: `Press Ctrl-C to stop the container....`

Example of a lightweight configuration:

```
[...]
<configuration>

  <!-- Container configuration -->
  <container>
    <containerId>tomcat5x</containerId>

<home>c:/apps/jakarta-tomcat-5.0.30</home>
  </container>

  <!-- Configuration to use with the
container -->
  <configuration>

<home>${project.build.directory}/tomcat5x</h
ome>
  </configuration>

</configuration>
[...]
```

This minimal configuration allows you to configure a default Tomcat 5.x standalone configuration (when the configuration type is not defined as above, the plugin will use a standalone configuration by default) in `${project.basedir}/target/resin`.

Example of a full-fledged m2 configuration:

```
[...]
<configuration>

  <!-- Container configuration -->
```

```
  <container>
    <containerId>tomcat6x</containerId>
    <zipUrlInstaller>

<url>http://archive.apache.org/dist/tomcat/t
omcat-6/v6.0.32/bin/apache-tomcat-6.0.32.zip
</url>
    </zipUrlInstaller>
    <!--
      Instead of downloading the container,
you can also reuse an
      existing installation by settings its
directory:

      <home>c:/apps/tomcat-6.0.32</home>
      -->

<output>${project.build.directory}/tomcat6x/
container.log</output>
    <append>false</append>

<log>${project.build.directory}/tomcat6x/car
go.log</log>
  </container>

  <!-- Configuration to use with the
container or the deployer -->
  <configuration>
    <type>standalone</type>

<home>${project.build.directory}/tomcat6x</h
```

```
ome>
    <properties>

<cargo.servlet.port>8080</cargo.servlet.port
>
      <cargo.logging>high</cargo.logging>
    </properties>
  </configuration>
  <deployables>
    <deployable>
      <groupId>war group id</groupId>
      <artifactId>war artifact
id</artifactId>
      <type>war</type>
      <properties>
        <context>optional root
context</context>
      </properties>
    </deployable>
    <deployable>
      <groupId>ear group id</groupId>
      <artifactId>ear artifact
id</artifactId>
      <type>ear</type>
    </deployable>
    [...]
```

```
  </deployables>
 </configuration>
[...]
```

This example shows the usage of a standalone configuration for configuring Tomcat 6.x. Note that it's possible to define deployables in the `<configuration>` element and they'll be deployed before the container starts (this is what we call static deployment). We have also defined some configuration properties to tell Cargo to configure Tomcat 6.x to start on port 8080 and to output highly verbose logs (useful for debugging).

If you have a container that is already installed and configured, say with other deployables already in there, you may want to use an existing configuration. This done by specifying `<type>existing</type>`. In that case you won't be able to control the configuration from Cargo (like port to use, logging levels, etc) as it'll be defined externally.

> ✅ **Timeouts**
>
> The CARGO container definiton as well as the various application ping options in CARGO have a `timeout` option (which is extremely useful in real-life applications), the default timeout being set to 2 minutes in most cases. Please check the Maven2 Plugin Reference Guide as well as the documentation about Container Timeout for details.

Configuring an Embedded Jetty Container

The most basic container you can use in cargo is an Embedded Jetty 4x/5x/6x Container. The following maven 2 plugin definition will configure an embedded Jetty 6x container. Note the use of the `embedded` type element to specify an embedded container.

```
[...]
<plugin>
 <groupId>org.codehaus.cargo</groupId>
 <artifactId>cargo-maven2-plugin</artifactId
>
 <configuration>
   <container>
     <containerId>jetty6x</containerId>
     <type>embedded</type>
   </container>
 </configuration>
</plugin>
[...]
```

Note: If you wish to use Jetty 5.x, you don't have to specify `<containerId>` nor `<type>` in the `<container>` element because Jetty 5x is Cargo's default container.

Setting the JAVA_HOME

You can set an alternate JAVA_HOME for non-embedded containers. This will allow you to run a container that uses JDK 1.4 even if your build runs on Java 5. Here's an example of how to do this.

```
[...]
<plugin>
 <groupId>org.codehaus.cargo</groupId>
 <artifactId>cargo-maven2-plugin</artifactId
>
 <configuration>
   [...]
   <properties>

<cargo.java.home>/usr/package/j2dsk1.4.2_11<
/cargo.java.home>
   </properties>
   [...]
 </configuration>
</plugin>
[...]
```

Automatic deployment of project's artifact (for J2EE projects)

If your project is a J2EE project (i.e. of type `<packaging>war</packaging>`, `<packaging>ear</packaging>`, `<packaging>ejb</packaging>` or `<packaging>uberwar</packaging>`) and you use the Cargo m2 plugin on that project then the generated artifact will be automatically added to the list of deployables to deploy. You can control the location of the artifact by using the `<location>` element (it defaults to `${project.build.directory}/${project.build.finalName}.${project.packaging}`). In addition if you want to wait for the deployment to be finished you can specify a `<pingURL>` (none is used by default). Here's an example:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
    <deployables>
      <deployable>

<location>${project.build.directory}/${proje
ct.build.finalName}.${project.packaging}</lo
cation>

<pingURL>http://localhost:port/mycontext/ind
ex.html</pingURL>
      </deployable>
    </deployables>
    [...]
```

Automatically executing and stopping the container when running mvn install

You might to automate the execution of `cargo:start` and `cargo:stop`. Of course you can bind those goals to any lifecycle phase (see Maven2's introduction to the lifecycle tutorial). Here's a common example used to start and the stop the container for doing integration tests:

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <executions>
    <execution>
      <id>start-container</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>start</goal>
      </goals>
    </execution>
    <execution>
      <id>stop-container</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    [Cargo plugin configuration goes in
here]
  </configuration>
</plugin>
```

In this example we've bound the execution of the `cargo:start` goal to the `pre-integration-test` phase and the `cargo:stop` goal to the `post-integration-test` phase. You'd run your tests in the `integration-test` p hase where you'll be assured that your container is already up and running. Running `mvn install` will execute everything automatically. For a full-fledged example and source code of an integration test with Cargo, see chapter 4 of the Better Builds with Maven book.

Adding JARs to the container's classpath

There are cases when you want to add extra JARs to your container's classpath, in order to share them between webapps for example. In order to add them, the first step is to define them as standard Maven2 dependencies. Then you should add a `<dependencies>` element under the `<container>` element and one `<dependency>` element for each JAR to add. For example, to add the Spring 2.0 JAR to your container's classpath you would write:

```
[...]
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
    <version>2.0</version>
  </dependency>
</dependencies>
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    <container>
      [...]
      <dependencies>
        <dependency>

<groupId>org.springframework</groupId>
          <artifactId>spring</artifactId>
        </dependency>
      </dependencies>
    </container>
    [...]
  </configuration>
</plugin>
```

Adding JDBC DataSources to the container's JNDI tree

Cargo supports configuration of one or more javax.sql.DataSource (DataSource) objects or arbitrary (Resource) objects into the JNDI tree of your container. Not all containers support this. Here are the ones that do.

DataSource configuration is the preferred means to configure a javax.sql.DataSource even if when you can technically do the same with a Resource. There are two major reasons for this:

1. Every configuration that supports Resources also support DataSources, but not visa versa. In other words, you will not be as portable, if you hard-code to a container's Resource implementation.
2. Resources are tricky and vary between vendors. Why add more work for yourself?

The next part of this topic will show how to configure DataSources in a portable way.

DataSource Configuration Examples

One DataSource:

Just add a property that starts with cargo.datasource.datasource. Note that you will probably need to add a classpath dependency for the driver.

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
    <properties>
      <cargo.datasource.datasource.derby>

cargo.datasource.driver=org.apache.derby.jdb
c.EmbeddedDriver|

cargo.datasource.url=jdbc:derby:derbyDB;crea
te=true|
        cargo.datasource.jndi=jdbc/CargoDS|
        cargo.datasource.username=APP|

cargo.datasource.password=nonemptypassword
```

```
      </cargo.datasource.datasource.derby>
    </properties>
    [...]
    <container>
      [...]
      <dependencies>
        <dependency>

<groupId>org.apache.derby</groupId>
          <artifactId>derby</artifactId>
        </dependency>
      </dependencies>
```

```
    </container>
  </configuration>
</plugin>
```

Multiple DataSources

To have multiple DataSources bound to JNDI, simply add more properties that start with cargo.datasource.datasource.

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
    <properties>
      <cargo.datasource.datasource.derby>

cargo.datasource.driver=org.apache.derby.jdb
c.EmbeddedDriver|

cargo.datasource.url=jdbc:derby:derbyDB;crea
te=true|
        cargo.datasource.jndi=jdbc/CargoDS|
        cargo.datasource.username=foo|

cargo.datasource.password=foopassword
      </cargo.datasource.datasource.derby>
      <cargo.datasource.datasource.derby2>

cargo.datasource.driver=org.apache.derby.jdb
```

```
c.EmbeddedDriver|

cargo.datasource.url=jdbc:derby:derbyDB2;cre
ate=true|
        cargo.datasource.jndi=jdbc/CargoDS2|
        cargo.datasource.username=bar|

cargo.datasource.password=barpassword
      </cargo.datasource.datasource.derby2>
    </properties>
```

```
    [...]
  </configuration>
</plugin>
```

Transactional DataSource

If you are not using an XADataSource, some containers still support transactions (ex. oc4j, weblogic). To do this, add the property cargo.datasource.transactionsupport.

- Note that if you specify this on a container that does not support transactions, you will get a CargoException during the configuration phase.
- valid values are LOCAL_TRANSACTION or XA_TRANSACTION. Note that XA_TRANSACTION is emulated in this case.

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
    <properties>
      <cargo.datasource.datasource.derbytx>
        cargo.datasource.driver=org.apache.derby.jdbc.EmbeddedDriver|
        cargo.datasource.url=jdbc:derby:derbyDB;create=true|
        cargo.datasource.jndi=jdbc/CargoDS|
        cargo.datasource.username=foo|
        cargo.datasource.password=foopassword|
        cargo.datasource.transactionsupport=LOCAL_TRANSACTION
      </cargo.datasource.datasource.derbytx>
    </properties>
    [...]
  </configuration>
</plugin>
```

XA Transactional DataSource

If you want to use real XA transactions, do the following:

1. set cargo.datasource.type=javax.sql.XADataSource
2. set cargo.datasource.driver=a valid implementation of javax.sql.XADataSource
3. set cargo.datasource.properties=a semi-colon delimited list of properties for that class

- Note that if you specify this on a container that does not support XADataSource configuration, you will get a CargoException during the configuration phase.

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
    <properties>
      <cargo.datasource.datasource.derbytx>
        cargo.datasource.type=javax.sql.XADataSource|

cargo.datasource.driver=org.apache.derby.jdbc.EmbeddedXADataSource|
        cargo.datasource.jndi=jdbc/CargoDS|
        cargo.datasource.username=foo|
        cargo.datasource.password=foopassword|

cargo.datasource.properties=createDatabase=create;databaseName=derbyDB
      </cargo.datasource.datasource.derbytx>
    </properties>
    [...]
  </configuration>
</plugin>
```

Adding POJO Resources to the container's JNDI tree

Cargo supports configuration of arbitrary (Resource) objects into the JNDI tree of your container. Not all containers support this. Here are the ones that do. Resources are configured when you add properties that start with cargo.resource.resource. Note that you will probably need to add a classpath dependency for the class and its interface.

Resource Configuration Examples

JavaMail Session:

- Resin and Tomcat require you supply the mail and activation jars externally due to Sun license requirements. The below example shows this.

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
```

```
    <properties>
      <cargo.resource.resource.mail>
        cargo.resource.name=mail/Session|

cargo.resource.type=javax.mail.Session|

cargo.resource.parameters=mail.smtp.host=loc
alhost;mail.smtp.connectiontimeout=10000
      </cargo.resource.resource.mail>
    </properties>
    [...]
    <container>
      [...]
      <dependencies>
        <dependency>

<groupId>javax.activation</groupId>

<artifactId>activation</artifactId>
        </dependency>
        <dependency>
          <groupId>javax.mail</groupId>
          <artifactId>mail</artifactId>
        </dependency>
      </dependencies>
    </container>
  </configuration>
</plugin>
...
<dependencies>
  <dependency>
```

```
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1</version>
  </dependency>
  <dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
```

```
    <version>1.4</version>
  </dependency>
</dependencies>
```

ConnectionPoolDataSource:

If you use a framework that does its own database connection pooling, you may need to configure a native ConnectionPoolDataSource. Here's how:

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
    <properties>
      <cargo.resource.resource.nativederby>

cargo.resource.name=resource/nativeCPDataSou
rce|

cargo.resource.type=javax.sql.ConnectionPool
DataSource|

cargo.resource.class=org.apache.derby.jdbc.E
mbeddedConnectionPoolDataSource|

cargo.resource.parameters=createDatabase=cre
ate;databaseName=derbyDB;user=APP;password=n
onemptypassword
      </cargo.resource.resource.nativederby>
```

```
    </properties>
    [...]
    <container>
      [...]
      <dependencies>
        <dependency>

<groupId>org.apache.derby</groupId>
          <artifactId>derby</artifactId>
        </dependency>
      </dependencies>
```

```
    </container>
  </configuration>
</plugin>
```

Multiple Resources

To have multiple POJO Resources bound to JNDI, simply add more properties that start with cargo.resource.resource.

```
[...]
<plugin>
  <groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>
  <configuration>
    [...]
    <properties>
      <cargo.resource.resource.apple>
        cargo.resource.name=resource/apple|
        cargo.resource.type=my.app.Fruit|

cargo.resource.class=my.app.AppleImpl|

cargo.resource.parameters=color=red;texture=
crunchy
      </cargo.resource.resource.apple>
      <cargo.resource.resource.orange>
        cargo.resource.name=resource/orange|
        cargo.resource.type=my.app.Fruit|

cargo.resource.class=my.app.OrangeImpl|

cargo.resource.parameters=color=orange;textu
re=rindy
      </cargo.resource.resource.orange>
    </properties>
    [...]
  </configuration>
</plugin>
```