JNDI

These instructions apply to releases of jetty-6.1.12rc3 and above, and of jetty-7.0.0.pre4 and above. For prior releases, these instructions remain applicable except for the "scoping" parameter of naming entries, which is not present, leaving the format as:

```
<New class=type of naming entry>
  <Arg>name to bind as</Arg>
  <Arg>the object to bind</Arg>
</New>
```

Using JNDI Resources with Jetty

Jetty supports `java:comp/env` lookups in webapps. This is an optional feature, and as such some setup needs to be done. We'll show you how to do it, although we have provided jetty configuration files that already have JNDI enabled to help you get going faster. More on that later.

Firstly, to enable JNDI for a web application, you need to configure the WebAppContext to parse the web.xml file and perform the `java:comp/env` linkages. The class that does this is `org.mortbay.jetty.plus.webapp.Configuration`, and we specify its name in the list of configurations to be applied to the webapp when we define the `org.mortbay.jetty.webapp.WebAppContext` for it. The following example enables naming services for the `xyz WebAppContext`:

```
<Array id="plusConfig"
type="java.lang.String">

<Item>org.mortbay.jetty.webapp.WebInfConfigu
ration</Item>

<Item>org.mortbay.jetty.plus.webapp.EnvConfi
guration</Item>

<Item>org.mortbay.jetty.plus.webapp.Configur
ation</Item>

<Item>org.mortbay.jetty.webapp.JettyWebXmlCo
nfiguration</Item>

<Item>org.mortbay.jetty.webapp.TagLibConfigu
ration</Item>
</Array>
  ...
  <New id="xyzWebAppContext"
class="org.mortbay.jetty.webapp.WebAppContex
t">
              ...
    <Set name="ConfigurationClasses"><Ref
id="plusConfig"/></Set>
              ...
  </New>
```

Or, more conveniently, you can specify that these configurations are applied to all webapps deployed by the WebAppDeployer :

```
<Call name="addLifeCycle">
      <Arg>
        <New
class="org.mortbay.jetty.deployer.WebAppDepl
oyer">
          <Set name="contexts"><Ref
id="Contexts"/></Set>
          <Set
name="webAppDir"><SystemProperty
name="jetty.home"
default="."/>/webapps</Set>
          <Set
name="parentLoaderPriority">false</Set>
          <Set name="extract">true</Set>
          <Set
name="allowDuplicates">false</Set>
          <Set
name="defaultsDescriptor"><SystemProperty
name="jetty.home"
default="."/>/etc/webdefault.xml</Set>
          <Set
name="configurationClasses"><Ref
id="plusConfig"/></Set>
        </New>
      </Arg>
</Call>
```

Hint

To save you some time, we have included the `etc/jetty-plus.xml` configuration file that configures a WebAppDeployer to deploy all webapps in the `webapps-plus` directory with JNDI. You can modify this as desired, or merge it with your etc/jetty.xml file to lessen the number of config files on the command line. See here for examples of putting multiple xml files on the jetty run line.

You may now configure naming resources that can be referenced in a web.xml file and accessed from within the `java:comp/env` naming environment of the webapp during execution. Specifically, you may configure support for the following web.xml elements:

```
<env-entry/>
<resource-ref/>
<resource-env-ref/>
```

Configuring env-entrys shows you how to set up overrides for `<env-entry>` elements in web.xml. Configuring resource-refs and resource-env-refs discusses how to configure support resources such as `javax.sql.DataSource`.

Furthermore, it is possible to plug a JTA `javax.transaction.UserTransaction` implementation into Jetty so that webapps can lookup `java:comp/UserTransaction` to obtain a distributed transaction manager. See Configuring XA Transactions.

You can define your naming resources with 3 scopes:

1. jvm scope - the name is unique within the jvm
2. server scope - the name is unique to the Server instance
3. webapp scope - the name is unique to the WebAppContext instance

The section Global or scoped to a webapp explains what scoping is, and shows you how to use it. Essentially, scoping ensures that JNDI bindings from one webapp do not interfere with the JNDI bindings of another - unless of course you wish them to.

Before we go any further, lets take a look at what kind of things can be bound into JNDI with Jetty.

What can be bound and general overview

There are 4 types of objects that can be bound into Jetty's JNDI:

- an ordinary POJO instance
- a java.naming.Reference instance
- an object instance that implements the java.naming.Referenceable interface
- a linkage between a name as referenced in web.xml and as referenced in the environment

The binding for all of these object types generally follows the same pattern:

```
<New class=type of naming entry>
<Arg>scope</Arg>
<Arg>name to bind as</Arg>
<Arg>the object to bind</Arg>
</New>
```

The `type of naming entry` can be:

- `"org.mortbay.jetty.plus.naming.EnvEntry"` for <env-entry>s
- `"org.mortbay.naming.plus.Resource"` for all other type of resources
- `"org.mortbay.plus.naming.Transaction"` for a JTA manager. We'll take a closer look at this in the Configuring XA Transactions section.
- `"org.mortbay.plus.naming.Link"` for link between a web.xml resource name and a NamingEntry. See Configuring Links for more info.

There are 3 places in which you can define naming entries:

1. jetty.xml
2. WEB-INF/jetty-env.xml
3. context xml file

Naming entries defined in a jetty.xml file will generally be scoped at either the jvm level or the Server level. Naming entries in a jetty-env.xml file will generally be scoped to the webapp in which the file resides, although you are able to enter jvm or Server scopes if you wish, that is not really recommended. In most cases you will define all naming entries that you want visible to a particular Server instance, or to the jvm as a whole in a jetty.xml file. Entries in a context xml file will generally be scoped at the level of the webapp to which it applies, although once again, you can supply a less strict scoping level of Server or jvm if you want.

Configuring env-entrys

Sometimes it is useful to be able to pass configuration information to a webapp at runtime that either cannot be or is not convenient to be coded into a web.xml `<env-entry>`. In this case, you can use `org.mortbay.jetty.plus.naming.EnvEntry` and even configure them to override an entry of the same name in web.xml.

```
<New
class="org.mortbay.jetty.plus.naming.EnvEntry">
  <Arg></Arg>
  <Arg>mySpecialValue</Arg>
  <Arg type="java.lang.Integer">4000</Arg>
  <Arg type="boolean">true</Arg>
</New>
```

This example will define a virtual `<env-entry>` called `mySpecialValue` with value `4000` that is unique within the

whole jvm. It will be put into JNDI at `java:comp/env/mySpecialValue` for *every* webapp deployed. Moreover, the boolean argument indicates that this value should override an `env-entry` of the same name in web.xml. If you don't want to override, then omit this argument or set it to `false`.

See Global or scoped to a webapp for more information on other scopes.

Note that the Servlet Specification only allows the following types of object to be bound to an `env-entry`:

- java.lang.String
- java.lang.Integer
- java.lang.Float
- java.lang.Double
- java.lang.Long
- java.lang.Short
- java.lang.Character
- java.lang.Byte
- java.lang.Boolean

However, Jetty is a little more flexible and will also allow you to bind custom POJOs, javax.naming.References and javax.naming.Referenceables. Be aware if you take advantage of this feature that your web application will **not be portable**.

To use the `EnvEntry` configured above, use code in your servlet/filter/etc such as:

```
import javax.naming.InitialContext;

InitialContext ic = new InitialContext();
Integer mySpecialValue =
(Integer)ic.lookup("java:comp/env/mySpecialV
alue");
```

Configuring resource-refs and resource-env-refs

Any type of resource that you want to refer to in a web.xml file as a `<resource-ref>` or `<resource-env-ref>` c an be configured using the `org.mortbay.naming.plus.Resource` type of naming entry. You provide the scope, the name of the object (relative to `java:comp/env`) and a POJO instance or a javax.naming.Reference instance or javax.naming.Referenceable instance.

The J2EE Specification recommends that DataSources are stored in `java:comp/env/jdbc`, JMS connection factories under `java:comp/env/jms`, JavaMail connection factories under `java:comp/env/mail` and URL connection factories under `java:comp/env/url`. For example:

Resource Type	Name in jetty.xml	Environment Lookup
javax.sql.DataSource	jdbc/myDB	java:comp/env/jdbc/myDB

javax.jms.QueueConnectionFactory	jms/myQueue	java:comp/env/jms/myQueue
javax.mail.Session	mail/myMailService	java:comp/env/mail/myMailService

Configuring DataSources

Lets look at an example of configuring a javax.sql.DataSource. Jetty can use any DataSource implementation available on it's classpath. In our example, we'll use a DataSource from the Derby relational database, but you can use any implementation of a `javax.sql.DataSource`. In this example, we'll configure it as scoped to a webapp with the id of 'wac':

```
<Configure id='wac'
class="org.mortbay.jetty.webapp.WebAppContex
t">
...
<New id="myds"
class="org.mortbay.jetty.plus.naming.Resourc
e">
  <Arg><Ref id="wac"/></Arg>
  <Arg>jdbc/myds</Arg>
  <Arg>
    <New
class="org.apache.derby.jdbc.EmbeddedDataSou
rce">
      <Set name="DatabaseName">test</Set>
      <Set
name="createDatabase">create</Set>
    </New>
  </Arg>
</New>
</Configure>
```

The above would create an instance of `org.apache.derby.jdbc.EmbeddedDataSource`, call the two setter methods `setDatabaseName("test");` and `setCreateDatabase("create");` and bind it into the JNDI scope for the webapp. If you have the appropriate <resource-ref> setup in your web.xml, then it will be available from

application lookups as `java:comp/env/jdbc/myds`.

To lookup your DataSource in your servlet/filter/etc do:

```
import javax.naming.InitialContext;
import javax.sql.DataSource;

InitialContext ic = new InitialContext();
DataSource myDS =
(DataSource)ic.lookup("java:comp/env/jdbc/myds");
```

Careful!

When configuring Resources, you need to ensure that the type of object you configure matches the type of object you expect to lookup in `java:comp/env`. For database connection factories, this means that the object you register as a Resource **must** implement the `javax.sql.DataSource` interface.

There are more examples of DataSources for various databases here.

Configuring JMS Queues, Topics and ConnectionFactories

Jetty is able to bind any implementation of the JMS destinations and connection factories. You just need to ensure the implementation jars are available on Jetty's classpath.

Here's an example of binding an ActiveMQ in-JVM connection factory:

```
<Configure id='wac'
class="org.mortbay.jetty.webapp.WebAppContex
t">
...
<New id="cf"
class="org.mortbay.jetty.plus.naming.Resourc
e">
  <Arg><Ref id='wac'/></Arg>
  <Arg>jms/connectionFactory</Arg>
  <Arg>
    <New
class="org.apache.activemq.ActiveMQConnectio
nFactory">

<Arg>vm://localhost?broker.persistent=false<
/Arg>
    </New>
  </Arg>
</New>
</Configure>
```

There is more information about ActiveMQ and Jetty here.

Configuring Mail

Jetty also provides infrastructure for providing access to javax.mail.Sessions from within an application:

```
<Configure id='wac'
class="org.mortbay.jetty.webapp.WebAppContex
t">
...
```

```
<New id="mail"
class="org.mortbay.jetty.plus.naming.Resourc
e">
     <Arg><Ref id="wac"/></Arg>
     <Arg>mail/Session</Arg>
     <Arg>
       <New
class="org.mortbay.naming.factories.MailSess
ionReference">
         <Set name="user">fred</Set>
         <Set
name="password">OBF:1xmk1w261z0f1w1c1xmq</Se
t>
         <Set name="properties">
           <New
class="java.util.Properties">
             <Put
name="mail.smtp.host">XXX</Put>
             <Put
name="mail.from">me@me</Put>
             <Put
name="mail.debug">true</Put>
           </New>
          </Set>
       </New>
```

```
      </Arg>
</New>
</Configure>
```

The setup above creates an instance of the `org.mortbay.naming.factories.MailSessionReference` class , calls it's setter methods `setUser("fred");`, `setPassword("OBF:1xmk1w261z0f1w1c1xmq");` to set up the authentication for the mail system, then populates a set of Properties, setting them on the MailSessionReference instance. The result of this is that an application can lookup `java:comp/env/mail/Session` at runtime and obtain access to a `javax.mail.Session` that has the necessary configuration to permit it to send email via SMTP.

> ✅ You can set the password to be plain text, or use Jetty's password obfuscation mechanism to make the config file more secure from prying eyes. Note that the other Jetty encryption mechanisms of MD5 and Crypt cannot be used as the original password cannot be recovered, which is necessary for the mail system.

We will be adding more examples of configuring database datasources (eg using XAPool and DBCP) and jms connection factories, so check back regularly. Contributions are also welcome.

Configuring XA Transactions

If you want to be able to perform distributed transactions with your resources, you will need a transaction manager that supports the JTA interfaces that you can lookup as `java:comp/UserTransaction` in your webapp. Jetty does not ship with one, rather you may plug in the one of your preference. You can configure the one of your choice using the `org.mortbay.jetty.plus.naming.Transaction` object in a jetty config file. In the following example, we will configure the Atomikos transaction manager:

```
<New id="tx"
class="org.mortbay.jetty.plus.naming.Transac
tion">
  <Arg>
    <New
class="com.atomikos.icatch.jta.J2eeUserTrans
action"/>
  </Arg>
</New>
```

Hint

In order to use the Atomikos transaction manager, you will need to download it and install it. There are instructions here on how to configure it for jetty6.

See also the instructions for how to configure JOTM. Contributions of instructions for other transaction managers are welcome.

Configuring Links

Usually, the name you configure for your NamingEntry should be the same as the name you refer to it as in you web.xml. For example:

```
In a context xml file:

<Configure id='wac'
class="org.mortbay.jetty.webapp.WebAppContex
t">
...
<New id="myds"
class="org.mortbay.jetty.plus.naming.Resourc
e">
  <Arg><Ref id="wac"/></Arg>
  <Arg>jdbc/mydatasource</Arg>
  <Arg>
    <New
class="org.apache.derby.jdbc.EmbeddedDataSou
rce">
      <Set name="DatabaseName">test</Set>
      <Set
name="createDatabase">create</Set>
    </New>
  </Arg>
</New>
```

```
</Configure>

and in web.xml:

  <resource-ref>

<res-ref-name>jdbc/mydatasource</res-ref-nam
e>

<res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <injection-target>

<injection-target-class>com.acme.JNDITest</i
njection-target-class>

<injection-target-name>myDatasource</injecti
```

```
on-target-name>
    </injection-target>
  </resource-ref>
```

If you wish, you can refer to it in web.xml by a different name, and link it to the name in your org.mortbay.jetty.plus.naming.Resource by using an org.mortbay.jetty.plus.naming.Link type of NamingEntry. For the example above, we could refer to the `jdbc/mydatasource` resource as {{jdbc/mydatasource1}} by doing:

```
In a context xml file:

<Configure id='wac'
class="org.mortbay.jetty.webapp.WebAppContex
t">
...
<New id="myds"
class="org.mortbay.jetty.plus.naming.Resourc
e">
  <Arg><Ref id="wac"/></Arg>
  <Arg>jdbc/mydatasource</Arg>
  <Arg>
    <New
class="org.apache.derby.jdbc.EmbeddedDataSou
rce">
      <Set name="DatabaseName">test</Set>
      <Set
name="createDatabase">create</Set>
    </New>
  </Arg>
</New>
</Configure>
```

in a jetty-env.xml file:

```
  <New id="map1"
class="org.mortbay.jetty.plus.naming.Link">
    <Arg><Ref id='wac'/></Arg>
    <Arg>jdbc/mydatasource1</Arg> <!-- name
in web.xml -->
    <Arg>jdbc/mydatasource</Arg>  <!-- name
in container environment -->
  </New>
```

and in web.xml:

```
  <resource-ref>

<res-ref-name>jdbc/mydatasource1</res-ref-na
me>

<res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <injection-target>

<injection-target-class>com.acme.JNDITest</i
njection-target-class>

<injection-target-name>myDatasource</injecti
```

```
on-target-name>
    </injection-target>
  </resource-ref>
```

This can be useful when you cannot change web.xml but need to link it to a resource in your deployment environment.

Global or scoped to a webapp

As we said before, you can control the visibility of your JNDI naming entries within your jvm, Server and WebAppContext instances. Naming entries at the *jvm scope* are visible by any application code, and are available to be bound to java:comp/env. Naming entries at the *Server scope* will not interfere with entries of the same name in a different Server instance, and are avilable to be bound to java:comp/env of any webapps deployed to that Server instance. Finally, the most specific scope are entries at the *webapp scope*. These are only available to be bound to java:comp/env of the webapp in which it is defined.

The scope is controlled by the 1st parameter to the NamingEntry.

The jvm scope is represented by a null parameter:

```
<New id="cf"
class="org.mortbay.jetty.plus.naming.Resourc
e">
  <Arg></Arg>
  <Arg>jms/connectionFactory</Arg>
  <Arg>
    <New
class="org.apache.activemq.ActiveMQConnectio
nFactory">

<Arg>vm://localhost?broker.persistent=false<
/Arg>
    </New>
  </Arg>
</New>
```

The Server scope is represented by referencing the related Server object:

```
<Configure id="Server"
class="org.mortbay.jetty.Server">
...
<New id="cf"
class="org.mortbay.jetty.plus.naming.Resourc
e">
  <Arg><Ref id="Server"/></Arg>
  <Arg>jms/connectionFactory</Arg>
  <Arg>
    <New
class="org.apache.activemq.ActiveMQConnectio
nFactory">

<Arg>vm://localhost?broker.persistent=false<
/Arg>
    </New>
  </Arg>
</New>
</Configure>
```

The webapp scope is represented by referencing the related WebAppContext object:

```
<Configure id='wac'
class="org.mortbay.jetty.webapp.WebAppContex
t">
...
<New id="cf"
class="org.mortbay.jetty.plus.naming.Resourc
e">
  <Arg><Ref id='wac'/></Arg>
  <Arg>jms/connectionFactory</Arg>
  <Arg>
    <New
class="org.apache.activemq.ActiveMQConnectio
nFactory">

<Arg>vm://localhost?broker.persistent=false<
/Arg>
    </New>
  </Arg>
</New>
</Configure>
```

As you can see, the most natural config files in which to declare naming entries of each scope are:

- jetty.xml - jvm or Server scope
- WEB-INF/jetty-env.xml or a context xml file - webapp scope

Demo Web Application

There is a demonstration webapp which sets up examples of all of the JNDI resources we've discussed so far.

In order to run this demonstration, you will need to download the transaction manager of your choice and Derby . At the time of writing, the webapp has been tested with both JOTM and with Atomikos transaction managers.

Building the Demo

As the demo webapp is not pre-built with the distribution, you first have to build it. It is located in `examples/test-jndi-webapp`. There is a `README.txt` file in there which explains how to build it, and how to add support for different transaction managers.

- run "mvn clean install" to build it
- then edit contexts/test-jndi.xml and uncomment one of the transaction manager setups
- then edit contexts/test-jndi.d/WEB-INF/jetty-env.xml and uncomment one of the transaction manager setups
- copy a derby.jar to the jetty lib/ directory, as well as copy all the necessary jars for the flavour of transaction manager you are using. There are instructions for some of the popular transaction managers on the wiki at J NDI

You run the demo like so:

For jetty 6.x:

```
java -jar start.jar
```

For jetty 7.x:

```
java -DOPTIONS=plus,ext,default -jar start.jar
```

The URL for the demonstration is at:

```
http://localhost:8080/test-jndi
```

Adding Support for a Different Transaction Manager

1. Edit the src/etc/templates/filter.properties file and add a new set of token and replacement strings following the pattern established for ATOMIKOS and JOTM.
2. Edit the src/etc/templates/jetty-env.xml file and add configuration for new transaction manager following the pattern established for the other transaction managers.
3. Edit the src/etc/templates/jetty-test-jndi.xml file and add configuration for the new transaction manager following the pattern established for the other transaction managers.