Resource Configuration

1.0 Introduction

Each application has a set of resources and services that it depends on (database storage, queueing, mail, etc.) and which need to be made available to it when it is deployed. Such resources may be scoped to the application instance, or may be shareable. To facilitate ease-of-use, it is desirable to externalize such resource dependencies as “resource definition” metadata in order to enable the deployment process to be further automated. Resource definitions allow an application to be deployed into the cloud with more minimal administrative configuration, and can also serve as a “template” for modifications and/or extensions to meet the deployment requirements of additional tenants. When an application is to be deployed to support multiple tenants, resource definitions will need to be modified to be tenant specific or incremented with tenant-specific metadata.

Application-specific resources can be scoped to any of the following Java EE namespaces as appropriate. Namespaces are assumed to be tenant specific.

- java:global: global namespace for all applications in the application server instance
- java:app: namespace shared by all components and modules in an application
- java:module: namespace shared by all components in a Java EE module
- java:comp: per component namespace

Java EE 6 provides “DataSource Resource Definition” metadata[1] to support the ability of an application to define its dependencies on one or more DataSources in its environment. Once defined, a DataSource resource may be referenced by a component using the lookup element of the Resource annotation or the resource-ref deployment descriptor element.

To support the deployment of Java EE 7 cloud-enabled applications, including multitenant applications, this ability needs to be extended to other resource types.

2.0 Examples

The following examples illustrate usage scenarios that we wish to accommodate.

1. See [1], Section 5.17, “DataSource Resource Definition”.

Example 1:

Application1 requires use of a database. It needs to issue only standard SQL, and doesn't have any dependencies on a particular database implementation, except that it must be able to support the use of JTA distributed transactions. Application1 specifies a JNDI name in the java:app namespace, and the Data-Source class name as the interface javax.sql.XADataSource. Upon deployment, the PaaS platform provisions a database for the application, and makes the Data-Source object available at the specified JNDI location. The DataSource object is otherwise configured using the defaults of the PaaS platform.

Example 2:

Application2 needs an Oracle database. It specifies a JNDI name in the java:app namespace and specifies the class-name as oracle.jdbc.pool.OracleData-Source. Application2 specifies QoS information in terms of minimum, maximum, and initial connection pool sizes. The platform provisions an Oracle database for the application and the DataSource object is made available at the specified JNDI location. The connection pool is configured as requested, subject to any limits imposed by the platform instance.

Example 3:

The Deployer of Application3 has worked with the PaaS System Administrator to provision a MySQL database. In addition to specifying the JNDI name and the class name com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource, the application specifies the particular server and port at which this database is accessed as well as other properties to be used for the configuration of the connection pool.

Example 4:

Application4 uses JMS for internal communication via a JMS queue which will be linked to an MDB. Because JMS is a required service that must be provided by the Java EE environment, Application4 specifies the JNDI names for the queue and connection factory used to send to it, and specifies the type of the administered object as javax.jms.Queue. The platform creates the required objects, including the "physical" queue used in the JMS broker implementation that is included as part of the platform environment.

Example 5:

Two related applications wish to communicate using JMS queues. These applications are to be deployed in the same application server instance. Application5A will be a queue sender to one queue and a queue receiver of the other. Application5B will be a queue receiver of the former queue and a queue sender to the latter. Both applications need access to JMS connection factories that will enable them to send/receive from these destinations.

Application5A specifies that it requires that a JMS queue connection factory and the two queues. Application5B does likewise. Applications 5A and 5B each use the same global JNDI name to refer to the queue that Application 5A will send to and Application 5B will receive from. Likewise, they will use a second common JNDI name to specify the queue that Application 5A will receive from and Application 5B will send to.

Example 6:

Two related applications wish to communicate using JMS queues. These applications will be deployed in <u>separate</u> application server instances. Application6A will be a queue sender to one queue and a queue receiver of the other. Application6B will be a queue receiver of the former queue and a queue sender to the latter. Both applications need access to JMS connection factories that will enable them to send/receive from these destinations.

Application6A specifies that it requires that a JMS queue connection factory and the two queues. Application6B likewise specifies that it requires a connection factory and access to the two queues. While the two queues are common to applications 6A and 6B at the level of the JMS broker, the queues managed at the level of the application server instances are distinct. Applications 6A and 6B each use the queue names defined within the JMS broker to identify the sharing of physical resources.

Example 7:

Application7, used by the tenant ExtraServices Inc., requires JavaMail as part of its work flow. It specifies the JNDI location at which the JavaMail Session object is to be made available and specifies that this should be configured to use the from address CRMService@ExtraServices.com.

3.0 Discussion of the Use Cases

There are two main use cases that need to be addressed. Some applications will require cases that are intermediate in specificity between these two to be enabled:

- An application requires an instance of a resource. Its needs are general in that it requires a resource with certain properties, but it does not require a particular instance of the resource. It expects the resource to be provisioned and configured for it by the PaaS platform instance.
- An application requires a particular instance of a resource, with specific configuration properties.
  - This resource may already exist. For example, it may previously have been created and configured by the Deployer or System Administrator.
  - Alternatively, the application requires that the resource be created for it and configured with the specified properties.

In Example 1 above, the application developer requires that a database be provided and a DataSource be made available at a particular JNDI location. The application's needs in terms of SQL are very generic and are expected to be satisfied by a wide range of databases. Such a resource dependency might be expressed using a generalization of the existing datasource XML element as follows:

```
<data-source>
   <name>java:app/jdbc/myDB</name>
   <class-name>javax.sql.XADataSource</class-name>
   <isolation-level>TRANSACTION_READ_COMMITTED</isolation-level>
</data-source>
```

The class-name is used to specify a generic XADataSource resource rather than specifying a vendor's implementation class and the JNDI name specifies that this DataSource will be an application-scoped resource. The platform will be responsible for provisioning a database in its environment (presumably the database that the container regards as its "default") and for mapping the DataSource. The configuration of the DataSource other than for isolation-level will be made according to the platform's defaults.

[Open Issue: The examples here are illustrated using a strawman XML format. We would also expect to consider annotations, with XML overrides.]

In Examples 2 and 3, the application requirements are more specific.

Application2 requires an instance of an Oracle database, but expects that this database will be provisioned for it and a DataSource be configured and made available with the specified properties. The resource dependency of Application 2 might be expressed as follows:

```
<data-source>
  <name>java:app/jdbc/myOracleDB</name>
  <class-name>oracle.jdbc.pool.OracleDataSource</class-name>
  <initial-pool-size>5</initial-pool-size>
  <max-pool-size>25</max-pool-size>
  <min-pool-size>10</min-pool-size>
</data-source>
```

Application3 requires that it be linked to a specific database that has already been provisioned through administrative action. The resource dependency of Application3 might be expressed as follows:

```
<data-source>
  <name>java:app/jdbc/mySQLDB</name>
  <class-name>
        com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource
  </class-name>
  <server-name>dbServer</server-name>
  <port-number>6689</port-number>
  <database-name>myDB</database-name>
  <isolation-level>TRANSACTION_READ_COMMITTED</isolation-level>
  <initial-pool-size>5</initial-pool-size>
  <max-pool-size>25</max-pool-size>
  <min-pool-size>10</min-pool-size>
  <max-statements>30</max-statements>
</data-source>
```

In Example 4, the application requires that a ConnectionFactory and a Queue be provisioned for it.

The resource dependencies of Application4 might be expressed as follows:

```
<jms-resource>
  <name>java:app/myJMSConnectionFactory</name>
</jms-resource>

<jms-admin-object>
  <name>java:app/myQueue</name>
  <resource-type>javax.jms.Queue</resource-type>
<jms-admin-object>
```

The type of the JMS resource object does not need to be specified, as it defaults to javax.jms.ConnectionFactory. No further information needs to be specified, as Application4 will use the default JMS broker and resource adapter provided by the platform, using the default configuration.

The platform causes the required queue to be created in the JMS broker, and the provider-specific queue name (the name of the "physical" queue) is generated by the platform. The platform creates the configured JMSConnectionFactory and Queue objects and binds them to the requested JNDI locations.

In Examples 5 and 6, two applications are communicating via JMS. Each application requires that two queues and a QueueConnectionFactory be available to it. These requirements can be fulfilled by the JMS broker implementation that is available by default in the platform environment.

In Example 5, the two applications that need to communicate are deployed in the same application server instance. In Example 6, they are deployed in different instances.

In Example 5, the two applications can use global JNDI names to specify the queues that are used for communication. They do not need to know or specify physical destination names.

Application5A might specify its requirements as follows

```
<jms-resource>
  <name>java:app/myJMSConnectionFactory</name>
  <resource-type>javax.jms.QueueConnectionFactory</resource-type>
</jms-resource>

<jms-admin-object>
  <name>java:global/queue1</name>
  <resource-type>javax.jms.Queue</resource-type>
<jms-admin-object>
```

```
<jms-admin-object>
  <name>java:global/queue2</name>
  <resource-type>javax.jms.Queue</resource-type>
<jms-admin-object>
```

Application5B specifies its requirements as follows, using the global JNDI names of those queues to insure linkage with Application5A:

```
<jms-resource>
  <name>java:app/myJMSCF</name>
  <resource-type>javax.jms.QueueConnectionFactory</resource-type>
</jms-resource>

<jms-admin-object>
  <name>java:global/queue1</name>
  <resource-type>javax.jms.Queue</resource-type>
<jms-admin-object>

<jms-admin-object>
  <name>java:global/queue2</name>
  <resource-type>javax.jms.Queue</resource-type>
<jms-admin-object>
```

In Example 6, the two applications are deployed to different application server instances. In this case, they need to use the same JMS infrastructure, and they need to use names within that infrastructure to indicate the shared objects that are to be used for commuication.

Application6A might specify its requirements as follows:

```
<jms-resource>
  <name>java:app/myJMSConnectionFactory</name>
  <resource-type>javax.jms.QueueConnectionFactory</resource-type>
</jms-resource>

<jms-admin-object>
  <name>java:app/myQueueToSendTo</name>
  <resource-type>javax.jms.Queue</resource-type>
  <resource-name>queue123</resource-name>
<jms-admin-object>

<jms-admin-object>
  <name>java:app/myQueueToReceiveFrom</name>
  <resource-type>javax.jms.Queue</resource-type>
  <resource-name>queue124</resource-name>
<jms-admin-object>
```

Application6A is deployed first. The names that is specifies are used by the platform in creating the queues in the JMS server.

Application6B specifies its requirements as follows, using the same queue names to insure linkage with Application6A:

```
<jms-resource>
  <name>java:app/myJMSCF</name>
  <resource-type>javax.jms.QueueConnectionFactory</resource-type>
</jms-resource>
```

```
<jms-admin-object>
  <name>java:app/receiveFromQueue</name>
  <resource-type>javax.jms.Queue</resource-type>
  <resource-name>queue123</resource-name>
<jms-admin-object>
```

```
<jms-admin-object>
  <name>java:app/sendToQueue</name>
  <resource-type>javax.jms.Queue</resource-type>
  <resource-name>queue124</resource-name>
<jms-admin-object>
```

In Example 7, the platform includes a Mail service provider, which is used by Application7. Application7 uses the default smtp transport protocol to send mail. It specifies that mail sent by it list the particular company email address in the from field.

Application7 might specify its requirements as follows:

```
<mail-resource>
  <name>java:app/mailSession</name>
  <from>CRMService@ExtraServices.com</from>
<mail-resource>
```

4.0 DataSource Resources

The DataSource resource definition annotation and XML element defined in Java EE 6 provide for the specification of the following:

- description: description of the datasource
- name: the jndi name of the data source being defined (required)
- class name: fully qualified class name of the DataSource, XADataSource or ConnectionPoolDataSource implementation class (e.g., com.example.VendorDataSource) (required)
- server name: database server name
- port number: port where a server is listening for requests
- database name: name of a database on a server
- url: JDBC URL If the url property is specified along with other standard DataSource properties such as server name, database name and port number, the more specific properties will take precedence and url will be ignored.
- user: User name to use for connection authentication.
- password: Password to use for connection authentication.
- property: JDBC DataSource property. This may be a vendor-specific property or a less commonly used DataSource property. Any number of properties may be specified.

- login timeout: Sets the maximum time in seconds that this data source will wait while attempting to connect to a database.
- transactional: Set to false if connections should not participate in transactions.
- isolation level: Isolation level for connections.
- initial pool size: Number of connections that should be created when a connection pool is initialized.
- max pool size: Maximum number of connections that should be concurrently-allocated for a connection pool.
- min pool size: Minimum number of connections that should be concurrently allocated for a connection pool.
- max idle time: The number of seconds that a physical connection should remain unused in the pool before the connection is closed for a connection pool.
- max statements: The total number of statements that a connection pool should keep open.

We propose that the interfaces javax.sql.DataSource and javax.sql.XADataSource be valid as values specified for the class-name attribute. If XADataSource is specified, an XADataSource is required to be made available.

If a database has not been previously provisioned, the PaaS platform is responsible for provisioning it and configuring the data source object to reference it. If a database has previously been provisioned for the application (e.g., by administrative action), it is the responsibility of the application to specify the server and port at which the database is to be accessed.

A platform instance is permitted to impose restrictions upon pool sizes, timeouts, and max statements subject to its implementation limits and quality of service guarantees. If an isolation level is specified, the platform instance must satisfy the request or provide a higher level of isolation.

Open Issue: What happens upon failure to satisfy the requested resources? Does the application fail to deploy or is there run-time failure?

5.0 JMS Connection Factory Resources

We propose to define annotations and XML descriptor elements for JMS resource definition metadata along the lines of the DataSource definition metadata described above. The following JMS resource definition attributes are proposed for configuration of JMS connection factory resources. Both annotations and XML elements should be defined to capture this information.

- description: description of the resource
- name: the jndi name of the resource being defined (required)
- resource-adapter-name: resource adapter name
- resource-type: JMS connection factory, e.g., javax.jms.QueueConnection-Factory. If not specified, this defaults to javax.jms.ConnectionFactory.
- user: User name to use for connection authentication.
- password: Password to use for connection authentication.
- clientId: JMS client identifier to be associated with a Connection
- max-pool-size: Maximum number of connections that should be concurrently allocated for a connection pool.
- property: resource property. This may be a vendor-specific property. Any number of properties may be specified.

Open Issue: Should any of the following attributes be standardized, or should they be left as properties? :

- connection-timeout: Sets the maximum time in seconds to wait while attempting to connect to a resource.
- transactional: [Do we need any way to distinguish XA from non-XA if transactional?]
- initial-pool-size: Number of connections that should be created when a connection pool is initialized.
- min-pool-size: Minimum number of connections that should be concurrently allocated for a connection pool.
- max-idle-time: The number of seconds that a physical connection should remain unused in the pool before the connection is closed for a connection pool.
- Other??

6.0 JMS Destination Resources (Queues and Topics)

We propose the following attributes for the configuration of JMS administered object resources. These would be captured in administered-object-specific annotations and XML elements.

- description: description of the administered object
- name: JNDI name for the administered object (required)
- resource-type: fully qualified type of the administered object (required), e.g., javax.jms.Queue
- resource-adapter: name of the resource adapter

- resource-name: name of the administered object
- property: configuration properties. Any number of properties may be specified. These may be vendor specific.

7.0 Mail Resources (javax.mail.Session resources)

- description: description of the mail resource
- name: JNDI name for the resource (required)
- store-protocol: storage protocol service.   [Note that the platform only requires support for sending mail.]
- store-protocol-class: service provider implementation class for storage. [Ditto.]
- transport-protocol: transport protocol service, for sending messages.
- transport-protocol-class: service provider implementation class for transport.
- host: mail server host name
- user: mail server user name
- from: email address the mail server uses to indicate the message sender
- property: configuration properties. Any number of properties may be specified. These may be vendor specific.

8.0 Other Connector and/or Custom Resources

Resource adapters for these are might be packaged with the application for example.

- description: description of the resource
- jndi-name: JNDI name for the resource (required)
- res-type: fully qualified type of the resource
- property: configuration properties. Any number of properties may be specified. These may be vendor specific
- Other??

9.0 Open Issues

A number of the examples (e.g., Example 4) illustrate the creation of objects upon deployment. Should this creation be limited to cloud environments? For example, should we key it off a cloud.xml descriptor?

What format should the specification of an application's resource configuration dependencies take, particularly when the application is intended for multitenant use? For example, should they be externalized in a separate services.xml file? Should we support annotations as well, with XML overriding?

Should such a descriptor also include the specification of QoS information as well as service dependencies? If so, what additional QoS metadata should we standardize on (e.g., scaling information such as minimum and maximum number of server instances)?

Semantics of JNDI global names. We have been assuming that the JNDI namespace is per tenant. If application server instances are shared by different tenants, it is not intended that these resources are shared by those tenants by default. However, there are some resources (e.g., databases) that might potentially be shared by tenants. What means should we use to indicate sharing of resources?

We will need a means to specify which attributes of the resource definition *must* be modified by a tenant, which attributes of the resource definition *must not* be modified when the application is deployed for a tenant, and which attributes of the resource definition *may* be modified by a tenant.

10.0 References

[1] Java Platform, Enterprise Edition Specification, v6, http://jcp.org/en/jsr/detail?id=316