Multitenancy Guide

JOnAS Team ()

- Jun 2012 -

Table of Contents

Chapter 1. Introduction

1.1. Multitenancy

JOnAS hosts and deploys applications written in Java. However, an application can not be natively deployed more than once in an instance of JOnAS. If necessary, the application will be deployed on another server instance because there is a risk of collision. One solution of this problem is multitenancy. This new feature provides the ability to deploy the same application multiple times on a single instance of JOnAS without prior configuration.

1.2. Shared application server

Tenants will run on the same application server (JOnAS) each with an instance of the application. Thus, in one instance of JOnAS, there will be many instances of an application than tenants using it. This multitenancy level is not without impact on the JOnAS application server because it will make changes in order to deploy the same application multiple times for multiple tenants in ensuring customization of resources and security among tenants.

According to JavaEE7 specifications, for each tenant, an instance of the application is deployed.

Chapter 2. Tenant context

Each tenant is identified by a tenant identifier following the pattern `T<id>` when `id` is numeric. This identifier is defined in web descriptor, application descriptor or even addon descriptor with the hierarchy :

```
Addon > EAR > WAR
```

It was necessary to define a default identifier (`defaultTenantId = T0`) for applications that do not give a specific tenant-id. it is simply used to enforce the policy where each instance of an application is linked to an identifier of tenant, and will not be used for customizing data (no changes will be made).

The identifier of the tenant must be present when deploying the application but also during its execution. In fact, during deployment, several services operate to save the settings and application data in the JOnAS environment. These services have a dependence on the multitenant service and will use it for customizing data. Therefore, it is necessary that the information "tenant-id" is constantly present throughout the duration of the deployment.

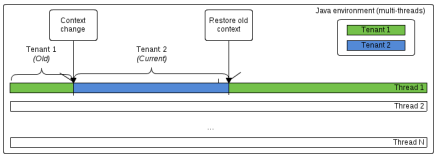
The tenant context is composed of :

- TenantId : tenant identifier
- InstanceName

To access the context of the current tenant, use :

```
TenantCurrent.getCurrent().getTenantContext();
```

Tenant-id is stored in a variable associated to the ThreadLocal. When running the application, an HTTP filter is set up, it sets all contexts associated with the thread, including the context of tenant, before the server responds to the client request.




```
// Save the current context
old = TenantCurrent.getCurrent().getTenantContext();
TenantCurrent.getCurrent().setTenantContext(this.ctx);
```

Next, execute the request. And finally, restore the old context :

```
// Restore the old context
TenantCurrent.getCurrent().setTenantContext(old);
```

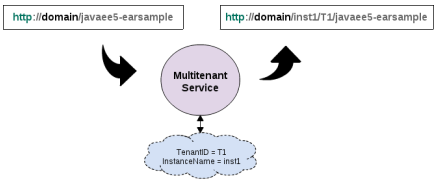
This filter is created by calling the multitenant service. The valve is set in Tomcat7Service as follows :

```
// For the tenantId
Filter tenantIdHttpFilter = null;
String tenantId = null;
if (getMultitenantService() != null){
  // get an instance of the filtre
  tenantId = super.getTenantId(war.getWarDeployable());
  tenantIdHttpFilter = getMultitenantService().getTenantIdFilter(tenantId);
  // needs to add this filter on the context
  jStdCtx.addValve(new FilterValveWrapper(tenantIdHttpFilter));
}
```

Chapter 3. Customization

3.1. Context root customization

Context root is defined in the web descriptor of the application. This context must be unique for each tenant. However, because tenants are instances of the same application, context root is the same for all. During deployment, context root of each instance is prefixed by the instance name and the tenant-id.



This customization is done during the deployment of the webapp.

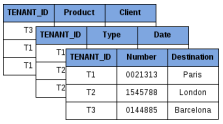
```
protected String updateContextRoot(String contextRoot, IDeployable deployable) {
  String tenantId = getTenantId(deployable);
  String instanceName = multitenantService.getInstanceNameFromContext();

  if (instanceName != null) {
    contextRoot = instanceName + "/" + contextRoot;
  }

  if (tenantId != null) {
    contextRoot = tenantId + "/" + contextRoot;
  }
  return contextRoot;
}
```

3.2. Data isolation in database

3.2.1. Shared database and shared schema



3.2.2. Propagation of the tenantId to eclipselink

To persist the tenantId in database, we have to set the `eclipselink.tenant-id` property in persistence.xml file. To automatize the propagation of the tenantId to eclipselink we need to add this property automatically when the application is added. Then, we will use the method :

```
// This property will propagate the tenantId to eclipselink
// This value will be added to entities tables
String tenantIdProperty = "eclipselink.tenant-id";
String tenantIdValue = tenantId;
persistenceUnitManager.setProperty(tenantIdProperty, tenantIdValue);
```

3.2.3. EJB Entities configured as multitenant

Entities must be configured as multitenant to enable adding tenant-id in the database. For that, we have to add `@Multitenant` annotation in each class but we need to do that automatically (when multitenant service is activated). A solution is to use a Session Customizer (cf http://wiki.eclipse.org/

Customizing_the_EclipseLink_Application_(ELUG)[1]). It is a simple class with only one method (customize) and take one parameter (Session session). In this method, we will set all entity classes as multitenant as follows :

```
public void customize(Session session) throws Exception {
  Map<Class, ClassDescriptor> descs = session.getDescriptors();
  // For each entity class ...
  for(Map.Entry<Class, ClassDescriptor> desc : descs.entrySet()){
    // Create a multitenant policy (Single table)
    SingleTableMultitenantPolicy policy = new SingleTableMultitenantPolicy(desc.getValue());
    // Tell that column descriminator is TENANT_ID (it will be added in the database)
    policy.addTenantDiscriminatorField("eclipselink.tenant-id", new
 DatabaseField("TENANT_ID"));
    // Add this policy in class derscriptor
    desc.getValue().setMultitenantPolicy(policy);
  }
}
```

Then, during the deployment of the application, an eclipselink property is set to use this session customizer :

```
// This property will configure entities as multitenant
// It is the equivalent of @Multitenant
String sessionCustomizerProperty = "eclipselink.session.customizer";
String sessionCustomizerClass =
 "org.ow2.easybeans.persistence.eclipselink.MultitenantEntitiesSessionCustomizer";
persistenceUnitManager.setProperty(sessionCustomizerProperty, sessionCustomizerClass);
```

Because tenants share the same database and the same tables, it is important to ensure that a tenant does not drop an create tables. For that, verify if the `drop-and-create-tables` eclipselink property is not set. Otherwise, change this property to `create-tables` only :

```
// If eclipselink was enabled to drop and create tables
// change this property to only create tables
String createTablesProperty = "eclipselink.ddl-generation";
String dropAndCreateTablesValue = "drop-and-create-tables";
String createTablesValue = "create-tables";
Map<String, String> properties = persistenceUnitManager.getProperty(createTablesProperty);
for (Map.Entry<String, String> property : properties.entrySet()){
  if (property.getValue().equals(dropAndCreateTablesValue)) {
    logger.warn("This tenant was enabled to drop and create tables. Eclipselink property is
 changed to only create tables");
    persistenceUnitManager.setProperty(createTablesProperty, createTablesValue,
 property.getKey());
  }
}
```

3.3. JNDI names customization

When an application is deployed in multitenant mode, we take the risk of having a conflict between bound names of each tenant. A solution is to add a prefix before each name. This prefix is the tenantId of the tenant which names are related.

3.3.1. Naming strategy

During the deployment, a name is prefixed the syntax : `T<id>/name` when `T<id>` is the application's tenant-id. The naming strategy is set by:

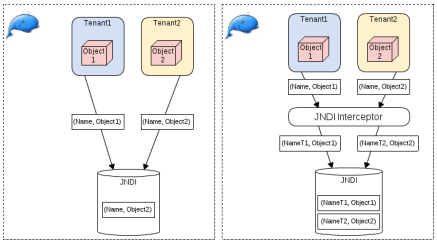
```
newNamingStrategies.add(ejb3Service.getNamingStrategy(prefix, oldNamingStrategy));
```

Example: MyInitializerBean will be T1/MyInitializerBean. In addition, as for versioning service, a virtual JNDI binding is made. It will remove the prefix and rebind the old name to the same object. Then, we will have 2 names (MyInitializeBean and T1/MyInitializerBean) linked to the same object.

[1] http://wiki.eclipse.org/Customizing_the_EclipseLink_Application_(ELUG)

3.3.2. JNDI interceptor

To customize JNDI names bound by the application by adding the tenant-id as prefix, an interceptor is set.



This interceptor (`org.ow2.jonas.lib.tenant.interceptor.jndi.JNDITenantIdInterceptor`) is an implementation of `org.ow2.carol.jndi.intercept.ContextInterceptor` and is registered in Carol Interceptor Manager when Multitenant service is activated. Then, all JNDI calls are intercepted.

```
// Add tenantId JNDI interceptor
jndiTenantIdInterceptor = new JNDITenantIdInterceptor(JNDI_SEPARATOR);
SingletonInterceptorManager.getInterceptorManager().registerContextInterceptor(jndiTenantIdInterceptor);
```

Two operations are made by this interceptor :

- Distinguish calls from multitenant application to JNDI
- Prefix these names by adding the tenant-id

3.4. MBeans customization

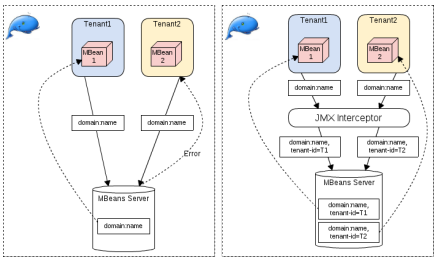
When we deploy a same application two times for two different tenants, the problem is that application's MBeans will have the same identifier which will create a case of conflict. To avoid it, a solution is to add an attribute in the MBean's ObjectName named tenantId :

```
Domaine:name=MBeanName;tenantId=T1
```

To do that, we need to intercept all MBeanServer methods call since majority of these methods use the ObjectName. A solution is to set a proxy of the principal MBeanServer (which is returned by `ManagementFactory.getPlatformMBeanServer()`)

3.4.1. JMX interceptor

To customize MBeans, a JMX interceptor is set to add a `tenant-id` property to the MBean ObjectName.



This interceptor (`org.ow2.jonas.lib.tenant.interceptor.jmx.JMXTenantIdInterceptor`) implements `org.ow2.jonas.jmx.Interceptor` and is added to the InvocationHandler (`org.ow2.jonas.jmx.internal.interceptor.InvocationHandlerImpl`) by multitenant service :

```
// Add tenantId JMX interceptor
jmxTenantIdInterceptor = new JMXTenantIdInterceptor(tenantIdAttributeName,
 allowToAccessPlatformMBeans);
jmxService.addInterceptor(jmxTenantIdInterceptor);
```

and will be called before querying the MBeanServer.

3.4.2. Customized MBeanServerBuilder

In order to create a "proxified" MBeanServer, a new class `org.ow2.jonas.services.bootstrap.mbeanbuilder.JOnASMBeanServerBuilder` which extends `javax.management.MBeanServerBuilder` is used and set as a system property :

```
// MBeanServerBuilder
System.setProperty("javax.management.builder.initial",
 "org.ow2.jonas.services.bootstrap.mbeanbuilder.JOnASMBeanServerBuilder");
```

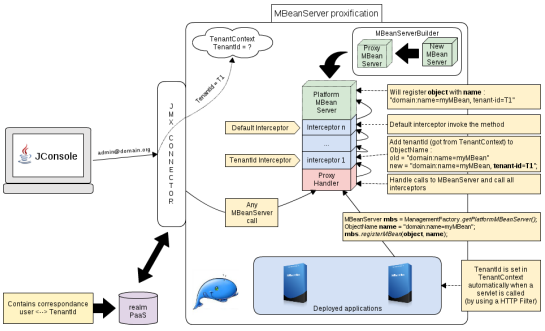
Then, the first MBeanServer created in the platform is a proxy with a default interceptor. This default interceptor is always the last called and will call the MBeanServer method originally invoked (before interception).

```
// Create real MBeanServer with outerProxy
MBeanServer origin = super.newMBeanServer(defaultDomain, outerProxy, delegate);

// Create handler for MBeanServer proxy and add
// the default interceptor
InvocationHandlerImpl invocationHandler = new InvocationHandlerImpl();
invocationHandler.addInterceptor(new MBeanServerDelegateInterceptor(origin));

// Create the MBeanServer proxy
MBeanServer proxy = (MBeanServer) Proxy.newProxyInstance(origin.getClass().getClassLoader(),
                                                         new Class<?>[]{MBeanServer.class},
                                                         invocationHandler);
```

It is possible to add as many interceptors that it is desired and they will be called one by one.



Using a customized MBeanServerBuilder can be problematic. In fact, as described in JONAS-867[2], if the system property `com.sun.management.jmxremote` is set before JOnAS startup, this has the effect of creating some MBeans and then initialize the MBeanServer. However, the system property `javax.management.builder.initial` which is set when JOnAS starts and define the class which is used to build the platform MBeanServer, this one is present in `org.ow2.jonas.services.bootstrap.mbeanbuilder.JOnASMBeanServerBuilder` and is not known by the classloader at this step. If the system property `com.sun.management.jmxremote` is not set, this error should not appear.

3.5. Tenants administration isolation

All administrators are defined in a special realm. Tenants administration isolation is done by defining two profiles :

[2] http://jira.ow2.org/browse/JONAS-867

3.5.1. Tenant administrator profile

This profile is define by the `super-admin` role :

```
<role name="super-admin" description="Role allowing access to all tenants" />
```

and will be assigned to a unique user in the platform :

```
<user name="superAdmin" password="admin" roles="superAdmin"/>
```

This user will have a full access to all MBeans when connecting on a administration tool such as JConsole.

3.5.2. Super administrator profile

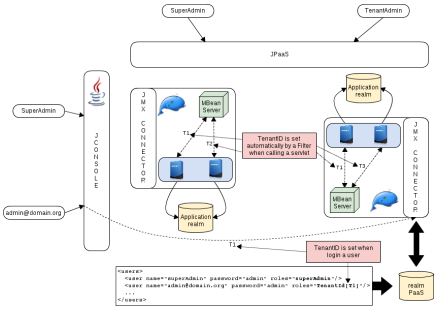
For each tenant, a specific role is defined according to the pattern `TenantId[T<id>]` :

```
<role name="TenantId[T1]" description="Role allowing access to tenant T1 information" />
```

and will be assigned to a unique user in the group of user belonging to the tenant.

```
<user name="admin@client" password="admin" roles="TenantId[T1]"/>
```

This user will have access to MBeans which contain the property `tenant-id=T1`.



This isolation is made when JMX Security is activated. For that, set JMX Security to true in jonas.properties :

```
jonas.service.jmx.secured    true
```

Then, change authentication method and parameter to :

```

jonas.service.jmx.authentication.method    jmx.remote.x.login.config
jonas.service.jmx.authentication.parameter    jaas-jmx
```

Next, activate TenantIdLoginModule for jaas-jmx in jaas.config :

```

jaas-jmx {
  // Use LoginModule for JMX authentication
  org.ow2.jonas.security.auth.spi.JResourceLoginModule required
  resourceName="memrlm_1"
  ;

  // Use the login module to add tenantId in TenantContext
  org.ow2.jonas.lib.tenant.loginmodule.TenantIdLoginModule required;
};
```

During tenant administrator login, TenantIdLoginModule will propagate tenantContext in order to filter MBeans by the `tenant-id` property presents in the ObjectName. Only MBeans which their objectName contains the same tenant-id than the connected user will appear.

3.6. Logs customization

JOnAS use Monolog[3] for logging. Monolog is a very static project and it was necessary to make it extensible for logging other information than those predefined (as date, classname, etc). One solution is to write an interface in monolog `org.objectweb.util.monolog.api.LogInfo`:

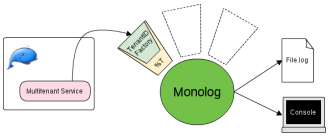
```
package org.objectweb.util.monolog.api;

/**
* This interface allows to add an extension to Monolog
* @author Mohammed Boukada
*/
public interface LogInfo {

  /**
  * Gets the info value
  * @return info value
  */
  String getValue ();
}
```

that will be implemented by JOnAS services.

In this case, this interface is implemented by multitenant service and provides the tenant-id of the current tenant.



An ipojo component is defined in `modules/libraries/externals/monolog` and is responsible of registration of monolog's extensions. When an implementation of this interface is registered in OSGi platform :

```
<provides specifications="org.objectweb.util.monolog.api.LogInfo">
  <property field="pattern" name="pattern" type="java.lang.Character"/>
</provides>
```

This component will add the extension to monolog :

```
<component classname="org.ow2.jonas.monolog.MonologExtension"
           immediate="false"
           name="MonologExtension">

  <requires optional="true"
            specification="org.objectweb.util.monolog.api.LogInfo"
            aggregate="true"
            proxy="false"
            nullable="false">
    <callback type="bind" method="addExtension" />
    <callback type="unbind" method="removeExtension" />
  </requires>

  <!-- LifeCycle Callbacks -->
  <callback method="start" transition="validate" />
  <callback method="stop" transition="invalidate" />

</component>
```

MonologExtension class contains method which are called as callback when a service implementing LogInfo interface is registered:

```
/**
* Add an extension to Monolog
```

[3] http://monolog.ow2.org/

```
 * @param logInfoProvider
 */
public void addExtension(final LogInfo logInfoProvider, ServiceReference ref) {
  Character pattern = (Character) ref.getProperty("pattern");
  Monolog.monologFactory.addLogInfo(pattern, logInfoProvider);
  logger.info("Extension ''{0}'' was added by ''{1}'' to Monolog", pattern,
 logInfoProvider.getClass().getName());
}

/**
* Remove an extension from Monolog
*/
public void removeExtension(ServiceReference ref) {
  Character pattern = (Character) ref.getProperty("pattern");
  Monolog.monologFactory.removeLogInfo(pattern);
  logger.info("Extension ''{0}'' was removerd from Monolog.", pattern);
}
```

To use monolog extension, you need to make a dependency on :

```
  <dependency>
    <groupId>org.ow2.monolog</groupId>
    <artifactId>monolog</artifactId>
    <version>2.2.1-SNAPSHOT</version>
  </dependency>
```

or any later version.

For seeing tenant-id in log messages, add `%T` to the wanted handler (tty, logf, ...) in trace.properties. Example :

```
handler.tty.pattern  %T %d : %O{1}.%M : %m%n
```

In this example, tenant-id will be added at the beginning of the log message. If tenant-id is not set in tenantContext or its value is `T0` (which is default tenant-id) then nothing will be printed.