

jUDDI User Guide

A guide to using jUDDI

by Tom Cunningham, Kurt Stam, Jeff Faath, and The jUDDI Community

and thanks to Darrin Mison

Preface	v
1. Document Conventions	v
1.1. Typographic Conventions	v
1.2. Pull-quote Conventions	vii
1.3. Notes and Warnings	vii
2. We Need Feedback!	viii
1. UDDI Registry	1
1.1. Introduction	1
1.2. UDDI Registry	1
2. Getting Started	3
2.1. What Should I Download?	3
2.2. Using the JAR	3
2.3. Using the WAR File	3
2.4. Using the Tomcat Bundle	3
2.5. Using jUDDI Web Services	4
3. Authentication	7
3.1. Introduction	7
3.2. jUDDI Authentication	8
3.3. XMLDocAuthentication	8
3.4. CryptedXMLDocAuthentication	9
3.5. LDAP Authentication	9
3.6. JBoss Authentication	10
4. Database Setup	11
4.1. Derby Out-of-the-Box	11
4.2. Switch to MySQL	13
4.3. Switch to Postgres	14
4.4. Switch to Oracle	15
4.5. Switch to HSQL	15
4.6. Switch to <other db>	17
5. Root Seed Data	19
5.1. Introduction	19
5.2. Seed Data Files	19
5.3. Token in the Seed Data	22
5.4. Customer Seed Data	22
6. jUDDI Configuration	23
6.1. Introduction	23
6.2. Authentication	23
6.3. Startup	23
6.4. Queries	25
6.5. Proxy Settings	26
6.6. KeyGeneration	27
6.7. Subscription	27
6.8. Transfer	28
7. Using the jUDDI-Client	29

7.1. Introduction	29
7.2. Configuration	29
7.3. JAX-WS Transport	30
7.4. RMI Transport	30
7.5. InVM Transport	31
7.6. Dependencies	32
7.7. Sample Code	32
8. UDDI Annotations	35
8.1. Introduction	35
8.2. UDDIService Annotation	35
8.3. UDDIServiceBinding Annotation	36
8.4. WebService Example	36
8.5. CategoryBag Attribute	37
9. Simple Publishing Using the jUDDI API	39
9.1. UDDI Data Model	39
9.2. jUDDI Additions to the Model	40
9.3. UDDI and jUDDI API	41
9.4. Getting Started	42
9.4.1. Simple Publishing Example	42
9.5. Conclusion	45
10. Subscription	47
10.1. Introduction	47
10.2. Two node example setup: Sales and Marketing	47
10.3. Deploy the HelloSales Service	51
10.4. Configure a user to create Subscriptions	53
10.5. Synchronous Notifications	55
11. Administration	61
11.1. Introduction	61
11.2. Changing the Listener Port	61
11.3. Changing the Oracle Sequence name	61
11.4. Persistence	67
12. Deploying to JBoss 6.0.0.GA	69
12.1. Introduction	69
12.2. Add juddiv3.war	69
12.3. Change web.xml	69
12.4. Configure Datasource	69
13. Deploying to Glassfish 2.1.1	71
13.1. Introduction	71
13.2. Glassfish jars	71
13.3. Configure the JUDDI datasource	71
13.4. Add juddiv3-cxf.war	72
13.5. Run juddi	73
A. Revision History	75

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts* [<https://fedorahosted.org/liberation-fonts/>] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F1** to switch to the first virtual terminal. Press **Ctrl+Alt+F7** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in `Mono-spaced Bold`. For example:

File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

Mono-spaced Bold Italic Of Proportional Bold Italic

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type `ssh username@domain.name` at a shell prompt. If the remote machine is `example.com` and your username on that machine is john, type `ssh john@example.com`.

The `mount -o remount file-system` command remounts the named file system. For example, to remount the `/home` file system, the command is `mount -o remount /home`.

To see the version of a currently installed package, use the `rpm -q package` command. It will return a result as follows: `package-version-release`.

Note the words in bold italics above `username`, `domain.name`, `file-system`, `package`, `version` and `release`. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as

a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules (MPMs)*. Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object      ref  = iniCtx.lookup("EchoBean");
        EchoHome    home = (EchoHome) ref;
        Echo        echo = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

A Note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.



Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you!

For any issues you find, or improvements you have, please sign up for a JIRA account at <https://issues.apache.org/jira/secure/Dashboard.jspa> and file a bug under the "jUDDI" component.

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

UDDI Registry

1.1. Introduction

The Universal Description, Discovery and Integration (UDDI) protocol is one of the major building blocks required for successful Web services. UDDI creates a standard interoperable platform that enables companies and applications to quickly, easily, and dynamically find and use Web services over the Internet. UDDI also allows operational registries to be maintained for different purposes in different contexts. UDDI is a cross-industry effort driven by major platform and software providers, as well as marketplace operators and e-business leaders within the OASIS standards consortium. UDDI has gone through 3 revisions and the latest version is 3.0.2. Additional information regarding UDDI can be found at <http://uddi.xml.org>.

1.2. UDDI Registry

The UDDI Registry implements the UDDI specification. UDDI is a Web-based distributed directory that enables businesses to list themselves on the Internet and discover each other, similar to a traditional phone book's yellow and white pages. The UDDI registry is both a white pages business directory and a technical specifications library. The Registry is designed to store information about Businesses and Services and it holds references to detailed documentation.

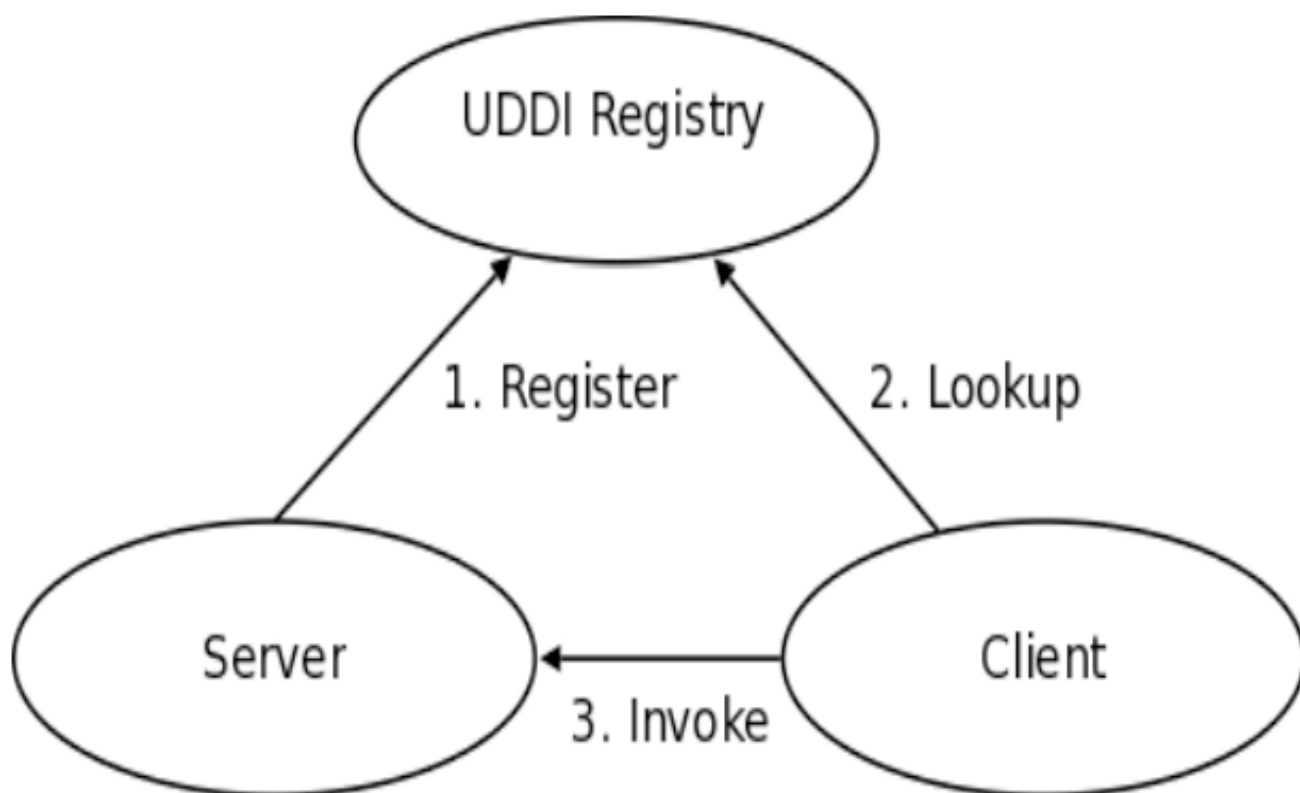


Figure 1.1. Invocation Pattern using the UDDI Registry

A business publishes services to the UDDI registry. A client looks up the service in the registry and receives service binding information. The client then uses the binding information to invoke the service. The UDDI APIs are SOAP based for interoperability reasons. The UDDI v3 specification defines 9 APIs:

1. `UDDI_Security_PortType`, defines the API to obtain a security token. With a valid security token a publisher can publish to the registry. A security token can be used for the entire session.
2. `UDDI_Publication_PortType`, defines the API to publish business and service information to the UDDI registry.
3. `UDDI_Inquiry_PortType`, defines the API to query the UDDI registry. Typically this API does not require a security token.
4. `UDDI_CustodyTransfer_PortType`, this API can be used to transfer the custody of a business from one UDDI node to another.
5. `UDDI_Subscription_PortType`, defines the API to register for updates on a particular business of service.
6. `UDDI_SubscriptionListener_PortType`, defines the API a client must implement to receive subscription notifications from a UDDI node.
7. `UDDI_Replication_PortType`, defines the API to replicate registry data between UDDI nodes.
8. `UDDI_ValueSetValidation_PortType`, by nodes to allow external providers of value set validation. Web services to assess whether `keyedReferences` or `keyedReferenceGroups` are valid.
9. `UDDI_ValueSetCaching_PortType`, UDDI nodes may perform validation of publisher references themselves using the cached values obtained from such a Web service.

Getting Started

2.1. What Should I Download?

The jUDDI server deploys as a WebARchive (war) named `juddiv3.war`. Within jUDDI, there are three downloadable files (`juddi-core.jar`, `juddi.war`, and `juddi-tomcat.zip`). You should determine which one to use depending on what level of integration you want with your application and your platform / server choices.

jUDDI also ships with client side code, the `juddi-client.jar`. The jUDDI server depends on the `juddi-client.jar` in situations where one server communicates to another server. In this setup one server acts as a client to the other server. The `juddi-client`.

2.2. Using the JAR

The `juddi-core` module produces a JAR which contains the jUDDI source and a jUDDI `persistence.xml` configuration. jUDDI's persistence is being actively tested with both OpenJPA and with Hibernate. If you are going to use only the JAR, you would need to directly insert objects into jUDDI through the database back end or persistence layer, or configure your own Web Service provider with the provided WSDL files and classes.

2.3. Using the WAR File

As with the JAR, you need to make a decision on what framework you would like to use when building the WAR. jUDDI's architecture supports any JAX-WS compliant WS stack (Axis, CXF, etc). The jUDDI 3.0.GA release ships with CXF in the Tomcat bundle, but any docs or descriptors to support other WS stacks would be welcome contributions. Simply copy the WAR to the deploy folder of your server (this release has been tested under Apache Tomcat 6.0.20), start your server, and follow the directions under "using jUDDI as a Web Service".

2.4. Using the Tomcat Bundle

The jUDDI Tomcat bundle packages up the jUDDI WAR, Apache Derby, and a few necessary configuration files and provides the user with a pre-configured jUDDI instance. By default, Hibernate is used as the persistence layer and CXF is used as a Web Service framework. To get started using the Tomcat bundle, unzip the `juddi-tomcat-bundle.zip`, and start Tomcat :

```
% cd apache-tomcat-6.0.20/bin
% ./startup.sh
```

It is suggested that you use JDK 1.6 with the Tomcat 6 bundle. On Mac OS X you can either change your `JAVA_HOME` settings or use `/Applications/Utilities/Java Preferences.app` to change your current JDK.

Once the server is up and running can make sure the root data was properly installed by browsing to <http://localhost:8080/juddiv3>

You should see the screen show in *Figure 2.1, "jUDDI Welcome Page"*.

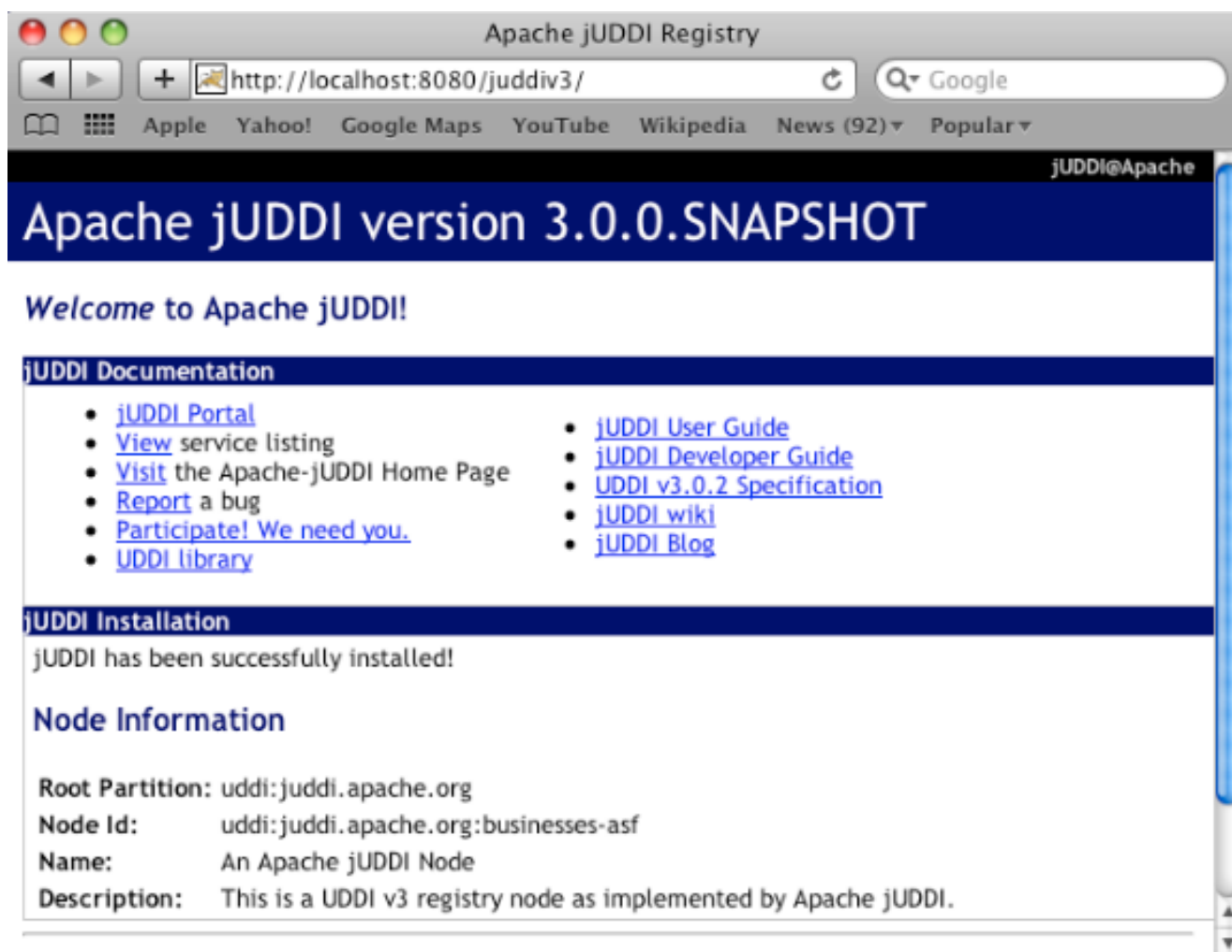


Figure 2.1. jUDDI Welcome Page

2.5. Using jUDDI Web Services

Once the jUDDI server is started, you can inspect the UDDI WebService API by browsing to <http://localhost:8080/juddiv3/services>

You should see an overview of all the Services and their WSDLs.

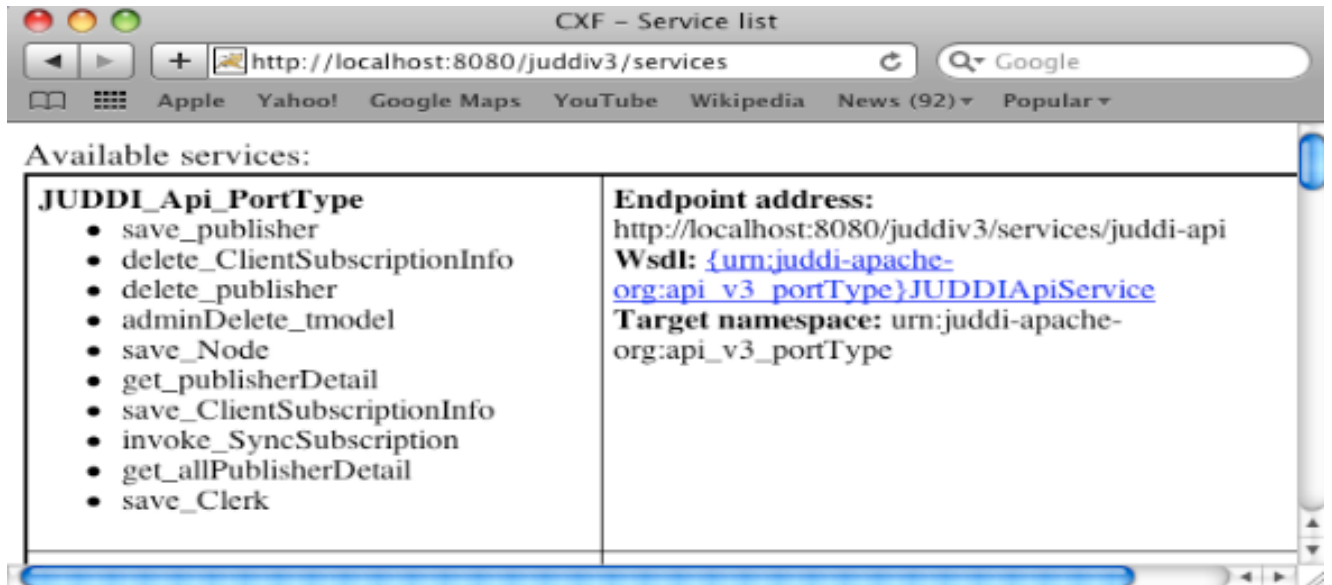


Figure 2.2. UDDI Services Overview

The services page shows you the available endpoints and methods available. Using any SOAP client, you should be able to send some sample requests to jUDDI to test:

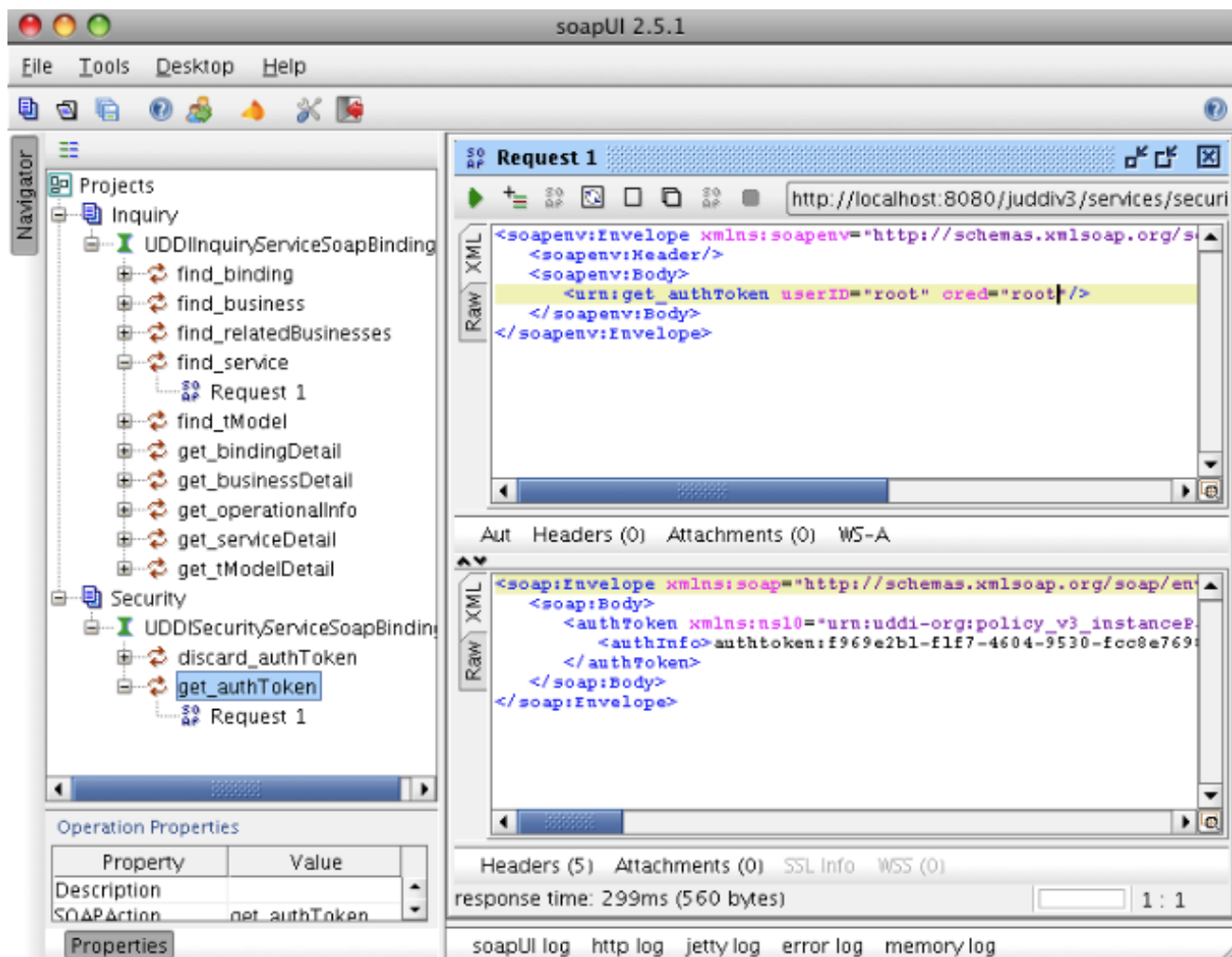


Figure 2.3. Getting an authToken using SoapUI

Authentication

3.1. Introduction

In order to enforce proper write access to jUDDI, each request to jUDDI needs a valid `authToken`. Note that read access is not restricted and therefore queries into the registries are not restricted.

To obtain a valid `authToken` a `getAuthToken()` request must be made, where a `GetAuthToken` object is passed. On the `GetAuthToken` object a `userid` and `credential` (password) needs to be set.

```
org.uddi.api_v3.GetAuthToken ga = new org.uddi.api_v3.GetAuthToken();
ga.setUserID(pubId);
ga.setCred("");

org.uddi.api_v3.AuthToken token = securityService.getAuthToken(ga);
```

The property `juddi.auth` in the `juddi.properties` configuration file can be used to configure how jUDDI is going to check the credentials passed in on the `GetAuthToken` request. By default jUDDI uses the `JUDDIAuthenticator` implementation. You can provide your own authentication implementation or use any of the ones mention below. The implementation needs to implement the `org.apache.juddi.auth.Authenticator` interface, and `juddi.auth` property should refer to the implementation class.

There are two phases involved in Authentication. The `authenticate` phase and the `identify` phase. Both of these phases are represented by a method in the `Authenticator` interface.

The `authenticate` phase occurs during the `GetAuthToken` request as described above. The goal of this phase is to turn a user id and credentials into a valid publisher id. The publisher id (referred to as the “authorized name” in UDDI terminology) is the value that assigns ownership within UDDI. Whenever a new entity is created, it must be tagged with ownership by the authorized name of the publisher. The value of the publisher id can be completely transparent to jUDDI – the only requirement is that one exists to assign to new entities. Thus, the `authenticate` phase must return a non-null publisher id. Upon completion of the `GetAuthToken` request, an authentication token is issued to the caller.

In subsequent calls to the UDDI API that require authentication, the token issued from the `GetAuthToken` request must be provided. This leads to the next phase of jUDDI authentication – the `identify` phase.

The `identify` phase is responsible for turning the authentication token (or the publisher id associated with that authentication token) into a valid `UddiEntityPublisher` object. The `UddiEntityPublisher` object contains all the properties necessary to handle ownership of UDDI entities. Thus, the token (or publisher id) is used to “identify” the publisher.

The two phases provide compliance with the UDDI authentication structure and grant flexibility for users that wish to provide their own authentication mechanism. Handling of credentials and publisher properties can be done entirely outside of jUDDI. However, jUDDI provides the Publisher entity, which is a sub-class of `UddiEntityPublisher`, to persist publisher properties within jUDDI. This is used in the default authentication and is the subject of the next section.

3.2. jUDDI Authentication

The default authentication mechanism provided by jUDDI is the `JUDDIAuthenticator`. The authenticate phase of the `JUDDIAuthenticator` simply checks to see if the user id passed in has an associated record in the Publisher table. No credentials checks are made. If, during authentication, the publisher does not exist, it the publisher is added on the fly.



Warning

Do not use jUDDI authentication in production.

The identify phase uses the publisher id to retrieve the Publisher record and return it. All necessary publisher properties are populated as Publisher inherits from `UddiEntityPublisher`.

```
juddi.auth = org.apache.juddi.auth.JUDDIAuthentication
```

3.3. XMLDocAuthentication

The `XMLDocAuthentication` implementation needs a XML file on the classpath. The `juddi.properties` file would need to look like

```
juddi.auth = org.apache.juddi.auth.XMLDocAuthentication
juddi.usersfile = juddi-users.xml
```

where the name of the XML can be provided but it defaults to `juddi-users.xml`, and the content of the file would look something like

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<juddi-users>
  <user userid="anou_mana" password="password" />
  <user userid="bozo" password="clown" />
  <user userid="sviens" password="password" />
</juddi-users>
```


The authenticate phase checks that the user id and password match a value in the XML file. The identify phase simply uses the user id to populate a new `UddiEntityPublisher`.

3.4. CryptedXMLDocAuthentication

The `CryptedXMLDocAuthentication` implementation is similar to the `XMLDocAuthentication` implementation, but the passwords are encrypted

```

juddi.auth = org.apache.juddi.auth.CryptedXMLDocAuthentication
juddi.usersfile = juddi-users-encrypted.xml
juddi.cryptor = org.apache.juddi.cryptor.DefaultCryptor

```

where the name user credential file is `juddi-users-encrypted.xml`, and the content of the file would look something like

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<juddi-users>
  <user userid="anou_mana" password="+j/kXkZJftwTFTBH6Cf6IQ==" />
  <user userid="bozo" password="Na2Ait+2aW0=" />
  <user userid="sviens" password="+j/kXkZJftwTFTBH6Cf6IQ==" />
</juddi-users>

```

The `DefaultCryptor` implementation uses `BEWithMD5AndDES` and `Base64` to encrypt the passwords. Note that the code in the `AuthenticatorTest` can be used to learn more about how to use this `Authenticator` implementation. You can plugin your own encryption algorithm by implementing the `org.apache.juddi.cryptor.Cryptor` interface and referencing your implementation class in the `juddi.cryptor` property.

The authenticate phase checks that the user id and password match a value in the XML file. The identify phase simply uses the user id to populate a new `UddiEntityPublisher`.

3.5. LDAP Authentication

`LdapSimpleAuthenticator` provides a way of authenticating users using Ldap simple authentication. It is fairly rudimentary and more LDAP integration is planned in the future, but this class allows you to authenticate a user based on an LDAP principal, provided that the principal and the juddi publisher ID are the same.

To use this class you must add the following properties to the `juddi.properties` file:

```

juddi.auth=org.apache.juddi.auth.LdapSimpleAuthenticator
juddi.auth.url=ldap://localhost:389

```

The `juddi.auth.url` property configures the `LdapSimpleAuthenticator` class so that it knows where the LDAP server resides. Future work is planned in this area to use the LDAP uid rather than the LDAP principal as the default publisher id.

3.6. JBoss Authentication

Finally is it possible to hook up to third party credential stores. If for example jUDDI is deployed to the JBoss Application server it is possible to hook up to it's authentication machinery. The `JBossAuthenticator` class is provided in the `docs/examples/auth` directory. This class enables jUDDI deployments on JBoss use a server security domain to authenticate users.

To use this class you must add the following properties to the `juddi.properties` file:

```
juddi.auth=org.apache.juddi.auth.JBossAuthenticator
juddi.securityDomain=java:/jaas/other
```

The `juddi.auth` property plugs the `JBossAuthenticator` class into the jUDDI the Authenticator framework. The `juddi.security.domain`, configures the `JBossAuthenticator` class where it can lookup the application server's security domain, which it will use to perform the authentication. Note that JBoss creates one security domain for each application policy element on the `$JBASS_HOME/server/default/conf/login-config.xml` file, which gets bound to the server JNDI tree with name `java:/jaas/<application-policy-name>`. If a lookup refers to a non existent application policy it defaults to a policy named `other`.

Database Setup

4.1. Derby Out-of-the-Box

By default jUDDI uses an embedded Derby database. This allows us to build a downloadable distribution that works out-of-the-box, without having to do any database setup work. We recommend switching to an enterprise-level database before going to production. JUDDI uses the Java Persistence API (JPA) in the back end and we've tested with both OpenJPA and Hibernate. To configure which JPA provider you want to use, you will need to edit the configuration in the `persistence.xml`. This file can be found in the `juddi.war/WEB-INF/classes/META-INF/persistence.xml`

For Hibernate the content of this file looks like

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="juddiDatabase" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:comp/env/jdbc/JuddiDS</jta-data-source>
    <!-- entity classes -->
    <class>org.apache.juddi.model.Address</class>
    <class>org.apache.juddi.model.AddressLine</class>
    <class>org.apache.juddi.model.AuthToken</class>
    <class>org.apache.juddi.model.BindingCategoryBag</class>
    <class>org.apache.juddi.model.BindingDescr</class>
    <class>org.apache.juddi.model.BindingTemplate</class>
    <class>org.apache.juddi.model.BusinessCategoryBag</class>
    <class>org.apache.juddi.model.BusinessDescr</class>
    <class>org.apache.juddi.model.BusinessEntity</class>
    <class>org.apache.juddi.model.BusinessIdentifier</class>
    <class>org.apache.juddi.model.BusinessName</class>
    <class>org.apache.juddi.model.BusinessService</class>
    <class>org.apache.juddi.model.CategoryBag</class>
    <class>org.apache.juddi.model.Contact</class>
    <class>org.apache.juddi.model.ContactDescr</class>
    <class>org.apache.juddi.model.DiscoveryUrl</class>
    <class>org.apache.juddi.model.Email</class>
    <class>org.apache.juddi.model.InstanceDetailsDescr</class>
    <class>org.apache.juddi.model.InstanceDetailsDocDescr</class>
```

```
<class>org.apache.juddi.model.KeyedReference</class>
<class>org.apache.juddi.model.KeyedReferenceGroup</class>
<class>org.apache.juddi.model.OverviewDoc</class>
<class>org.apache.juddi.model.OverviewDocDescr</class>
<class>org.apache.juddi.model.PersonName</class>
<class>org.apache.juddi.model.Phone</class>
<class>org.apache.juddi.model.Publisher</class>
<class>org.apache.juddi.model.PublisherAssertion</class>
<class>org.apache.juddi.model.PublisherAssertionId</class>
<class>org.apache.juddi.model.ServiceCategoryBag</class>
<class>org.apache.juddi.model.ServiceDescr</class>
<class>org.apache.juddi.model.ServiceName</class>
<class>org.apache.juddi.model.ServiceProjection</class>
<class>org.apache.juddi.model.Subscription</class>
<class>org.apache.juddi.model.SubscriptionChunkToken</class>
<class>org.apache.juddi.model.SubscriptionMatch</class>
<class>org.apache.juddi.model.Tmodel</class>
<class>org.apache.juddi.model.TmodelCategoryBag</class>
<class>org.apache.juddi.model.TmodelDescr</class>
<class>org.apache.juddi.model.TmodelIdentifier</class>
<class>org.apache.juddi.model.TmodelInstanceInfo</class>
<class>org.apache.juddi.model.TmodelInstanceInfoDescr</class>
<class>org.apache.juddi.model.TransferToken</class>
<class>org.apache.juddi.model.TransferTokenKey</class>
<class>org.apache.juddi.model.UddiEntity</class>
<class>org.apache.juddi.model.UddiEntityPublisher</class>

<properties>
  <property name="hibernate.archive.autodetection" value="class"/>
  <property name="hibernate.hbm2ddl.auto" value="update"/>
  <property name="hibernate.show_sql" value="false"/>
  <property name="hibernate.dialect"
    value="org.hibernate.dialect.DerbyDialect"/>
</properties>
</persistence-unit>
</persistence>
```

The `persistence.xml` reference a datasource "java:comp/env/jdbc/JuddiDS". Datasource deployment is Application Server specific. If you are using Tomcat, then the datasource is defined in `juddi/META-INF/context.xml` which by default looks like

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
```

```

<WatchedResource>WEB-INF/web.xml</WatchedResource>
<!-- -->
<Resource name="jdbc/JuddiDS" auth="Container"
  type="javax.sql.DataSource" username="" password=""
  driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
  url="jdbc:derby:juddi-derby-test-db;create=true"
  maxActive="8"
  />
</Context>

```

4.2. Switch to MySQL

To switch over to MySQL you need to add the mysql driver (i.e. The `mysql-connector-java-5.1.6.jar`) to the classpath and you will need to edit the `persistence.xml`

```
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
```

and the `datasource`. For tomcat you the `context.xml` should look something like

```

<?xml version="1.0" encoding="UTF-8"?>
<Context>
  <WatchedResource>WEB-INF/web.xml</WatchedResource>
  <Resource name="jdbc/JuddiDS" auth="Container"
    type="javax.sql.DataSource" username="root" password=""
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/juddiv3"
    maxActive="8"/>
</Context>

```



Warning

Tomcat copies the `context.xml` to `conf/CATALINA/localhost/juddiv3.xml`, and if you update the `context.xml` it may not update this copy. You should simply delete the `juddiv3.xml` file after updating the `context.xml`.

To create a MySQL database name `juddiv3` use

```
mysql> create database juddiv3
```

and finally you probably want to switch to a user which is a bit less potent than `'root'`.

4.3. Switch to Postgres

This was written from a JBoss - jUDDI perspective. Non-JBoss-users may have to tweak this a little bit, but for the most part, the files and information needed is here.

Logged in as postgres user, access psql:

```
# psql

postgres=# CREATE USER juddi with PASSWORD 'password';
postgres=# CREATE DATABASE juddi;
postgres=# GRANT ALL PRIVILEGES ON DATABASE juddi to juddi;
```

Note, for this example, my database is called juddi, as is the user who has full privileges to the database. The user 'juddi' has a password set to 'password'.

```
<datasources>
  <local-tx-datasource>
    <jndi-name>JuddiDS</jndi-name>
    <connection-url>jdbc:postgresql://localhost:5432/juddi</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>juddi</user-name>
    <password>password</password>
    <!-- sql to call when connection is created. Can be anything,
    select 1 is valid for PostgreSQL
    <new-connection-sql>select 1</new-connection-sql>
    -->
    <!-- sql to call on an existing pooled connection when it is obtained
    from pool. Can be anything, select 1 is valid for PostgreSQL
    <check-valid-connection-sql>select 1</check-valid-connection-sql>
    -->
    <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml -->
    <metadata>
      <type-mapping>PostgreSQL 8.0</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

In `persistence.xml`, reference the correct JNDI name of the datasource and remove the derby Dialect and add in the postgresql Dialect:

```
<jta-data-source>java:comp/env/jdbc/JuddiDS</jta-data-source>
.....
```

```
<property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
```

Be sure to have `postgresql-8.3-604.jdbc4.jar` in the `lib` folder!

4.4. Switch to Oracle

To switch over to Oracle you need to add the oracle driver (i.e. `theClasses12.jar`) to the classpath and you will need to edit the `persistence.xml`

```
<property name="hibernate.dialect" value="org.hibernate.dialect.Oracle10gDialect"/>
```

To create a Oracle database name `juddiv3` with the ultimate in minimalism use

```
sqlplus> create database juddiv3;
```

then you probably want to switch to a user which is a bit less potent then 'root' and set the appropriate password.



Warning

Tomcat copies the `context.xml` to `conf/CATALINA/localhost/juddiv3.xml`, and if you update the `context.xml` it may not update this copy. You should simply delete the `juddiv3.xml` file after updating the `context.xml`.

Please check the [Section 11.3, "Changing the Oracle Sequence name"](#) if you want to change the Oracle Sequence name.

4.5. Switch to HSQL

First make sure you have a running `hsqldb`. For a standalone server startup use:

```
java -cp hsqldb.jar org.hsqldb.server.Server --port 1747 --database.0 file:juddi --dbname.0 juddi
```

Next, connect the client manager to this instance using:

Chapter 4. Database Setup

```
java -classpath hsqldb.jar org.hsqldb.util.DatabaseManagerSwing --driver org.hsqldb.jdbcDriver
--url jdbc:hsqldb:hsqldb://localhost:1747/juddi -user sa
```

and create the juddi user:

```
CREATE USER JUDDI PASSWORD "password" ADMIN;
CREATE SCHEMA JUDDI AUTHORIZATION JUDDI;
SET DATABASE DEFAULT INITIAL SCHEMA JUDDI;
ALTER USER juddi set initial schema juddi;
```

From now on, one can connect as JUDDI user to that database and the database is now ready to go.

To switch over to HSQL you need to add the hsql driver (i.e. The `hsqldb.jar`) to the classpath and you will need to edit the `persistence.xml`

```
<property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
```

and the datasource. For tomcat you the `context.xml` should look something like

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
  <WatchedResource>WEB-INF/web.xml</WatchedResource>
  <!-- HSQL data source -->
  <Resource name="jdbc/JuddiDS" auth="Container"
    type="javax.sql.DataSource" username="JUDDI" password="password"
    driverClassName="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:hsqldb://localhost:1747/juddi"
    maxActive="8"
  />
</Context>
```




Warning

Tomcat copies the `context.xml` to `conf/CATALINA/localhost/juddiv3.xml`, and if you update the `context.xml` it may not update this copy. You should simply delete the `juddiv3.xml` file after updating the `context.xml`.

4.6. Switch to <other db>

If you use another database, please document, and send us what you had to change to make it work and we will include it here.

Root Seed Data

5.1. Introduction

As of UDDI v3, each registry need to have a "root" publisher. The root publisher is the owner of the UDDI services (inquiry, publication, etc). There can only be one root publisher per node. JUDDI ships some default seed data for the root account. The default data can be found in the `juddi-core-3.x.jar`, under `juddi_install_data/`. By default jUDDI installs two Publishers: "root" and "uddi". Root owns the root partition, and uddi owns all the other seed data such as pre-defined tModels.

5.2. Seed Data Files

For each publisher there are four seed data files that will be read the first time you start jUDDI:

```
<publisher>_Publisher.xml
<publisher>_tModelKeyGen.xml
<publisher>_BusinessEntity.xml
<publisher>_tModels.xml
```

For example the content of the `root_Publisher.xml` looks like

```
<publisher xmlns="urn:juddi-apache-org:api_v3" authorizedName="root">
  <publisherName>root publisher</publisherName>
  <isAdmin>true</isAdmin>
</publisher>
```

Each publisher should have its own key generator schema so that custom generated keys cannot end up being identical to keys generated by other publishers. It is therefor that the each publisher need to define their own KenGenerator tModel. The tModel Key Generator is defined in the file `root_tModelKeyGen.xml` and the content of this file is

```
<tModel tModelKey="uddi:juddi.apache.org:keygenerator" xmlns="urn:uddi-org:api_v3">
  <name>uddi-org:keyGenerator</name>
  <description>Root domain key generator</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#keyGen
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="uddi:uddi.org:categorization:types"
```

```
    keyName="uddi-org:types:keyGenerator"
    keyValue="keyGenerator" />
</categoryBag>
</tModel>
```

This means that the legal format of keys used by the root publisher need to be in the form `uddi:juddi.apache.org:<text-of-choice>`. The use of other types of format will lead to an 'illegal key' error. The root publisher can only own one KeyGenerator while any other publisher can own more than one KeyGenerator. KeyGenerators should not be shared unless there is a good reason to do so. If you want to see your publisher with more than just the one KeyGenerator tModel, you can use the `<publisher>_tModels.xml` file.

Finally, in the `<publisher>_BusinessEntity.xml` file can be used to setup Business and Service data. In the `root_BusinessEntity.xml` we specified the ASF Business, and the UDDI services; Inquiry, Publish, etc.:

```
<businessEntity xmlns="urn:uddi-org:api_v3"
  xmlns:xml="http://www.w3.org/XML/1998/namespace"
  businessKey="uddi:juddi.apache.org:businesses-asf">
  <!-- Change the name field to represent the name of your registry -->
  <name xml:lang="en">An Apache jUDDI Node</name>
  <!-- Change the description field to provided
  a brief description of your registry -->
  <description xml:lang="en">
    This is a UDDI v3 registry node as implemented by Apache jUDDI.
  </description>
  <discoveryURLs>
  <!-- This discovery URL should point to the home installation URL of jUDDI -->
  <discoveryURL useType="home">
    http://{juddi.server.name}:{juddi.server.port}/juddiv3
  </discoveryURL>
</discoveryURLs>
<categoryBag>
  <keyedReference tModelKey="uddi:uddi.org:categorization:nodes" keyValue="node" />
</categoryBag>

<businessServices>
<!-- As mentioned above, you may want to provide user-defined keys for
these (and the services/bindingTemplates below. Services that you
don't intend to support should be removed entirely -->
  <businessService serviceKey="uddi:juddi.apache.org:services-inquiry"
    businessKey="uddi:juddi.apache.org:businesses-asf">
    <name xml:lang="en">UDDI Inquiry Service</name>
```

```

<description xml:lang="en">Web Service supporting UDDI Inquiry API</description>
<bindingTemplates>
  <bindingTemplate bindingKey="uddi:juddi.apache.org:servicebindings-inquiry-ws"
    serviceKey="uddi:juddi.apache.org:services-inquiry">
    <description>UDDI Inquiry API V3</description>
    <!-- This should be changed to the WSDL URL of the inquiry API.
    An access point inside a bindingTemplate will be found for every service
    in this file. They all must point to their API's WSDL URL -->
    <accessPoint useType="wsdlDeployment">
      http://${juddi.server.name}:${juddi.server.port}/juddiv3/services/inquiry?wsdl
    </accessPoint>
    <tModelInstanceDetails>
      <tModelInstanceInfo tModelKey="uddi:uddi.org:v3_inquiry">
        <instanceDetails>
          <instanceParms>
            <![CDATA[
              <?xml version="1.0" encoding="utf-8" ?>
              <UDDIinstanceParmsContainer
                xmlns="urn:uddi-org:policy_v3_instanceParms">
                <defaultSortOrder>
                  uddi:uddi.org:sortorder:binarysort
                </defaultSortOrder>
              </UDDIinstanceParmsContainer>
            ]]>
          </instanceParms>
        </instanceDetails>
      </tModelInstanceInfo>
    </tModelInstanceDetails>
    <categoryBag>
      <keyedReference keyName="uddi-org:types:wsdl" keyValue="wsdlDeployment"
        tModelKey="uddi:uddi.org:categorization:types"/>
    </categoryBag>
  </bindingTemplate>
</bindingTemplates>
</businessService>
<businessService serviceKey="uddi:juddi.apache.org:services-publish"
  businessKey="uddi:juddi.apache.org:businesses-asf">
  <name xml:lang="en">UDDI Publish Service</name>
  .....
</businessService>
</businessServices>
</businessEntity>

```

Note that the seeding process only kicks off if no publishers exist in the database. So this will only work with a clean database, unless you set `juddi.seed.always` to true. Then it will re-apply all files with the exception of the root data files. Note that this can lead to losing data that was added to entities that are re-seeded, since data is not merged.

5.3. Token in the Seed Data

You may have noticed the tokens in the `root_BusinessEntity.xml` file (`${juddi.server.baseurl}`). The value of this tokens can set in the `juddiv3.properties` file. The value substitution takes place at runtime, so that different nodes can do the substitution with their own value if needed.

5.4. Customer Seed Data

In your deployment you probably do not want to use the Seed Data shipped with the default jUDDI install. The easiest way to overwrite this data is to add it to a directory call `juddi_custom_install_data` in the `juddiv3.war/WEB-INF/classes/` directory. That way you don't have to modify the `juddi-core-3.x.jar`. Additionally if your root publisher is not called "root" you will need to set the `juddi.root.publisher` property in the `juddiv3.properties` file to something other than

```
juddi.root.publisher=root
```

The `juddiv3.war` ships with two example data directory. One for the Sales Affiliate, and one for the Marketing Affiliate. To use the Sales Seed Data, in the `juddiv3.war/WEB-INF/classes/`, rename the directory

```
mv RENAME4Sales_juddi_custom_install_data juddi_custom_install_data
```

before you start jUDDI the first time. It will then use this data to populate the database. If you want to rerun you can trash the database it created and restart tomcat. Don't forget to set the tokens in the `juddiv3.properties` file.

jUDDI_Configuration

6.1. Introduction

jUDDI will look for a `juddiv3.properties` file on the root of the classpath. In the `juddiv3.war` you can find it in `juddiv3.war/WEB_INF/classes/juddiv3.properties`

6.2. Authentication

```
# Specifies whether the inquiry API requires authentication
juddi.authenticate.Inquiry=false
```

This flag determines whether authentication (the presence of a `getAuthToken`) is required on queries invoking the Inquiry API. By default, jUDDI sets this to false for ease of use.

```
# jUDDI Authentication module to use
juddi.authenticator = org.apache.juddi.v3.auth.JUDDIAuthenticator
```

The jUDDI authenticator class to use. See Chapter 3 of the Userguide for the choices provided.

6.3. Startup

```
# The ${juddi.server.baseurl} token can be referenced in accessPoints and will be resolved at
runtime.
juddi.server.baseurl=http://localhost:8080
```

Token that can be accessed in `accessPointURLs` and resolved at runtime.

```
#
juddi.root.publisher=root
```

The username for the jUDDI root publisher. This is usually just set to "root".

```
#
juddi.seed.always=false
```

Chapter 6. jUDDI_Configuration

Forces seeding of the jUDDI data. This will re-apply all files with the exception of the root data files. Note that this can lead to losing data that was added to the entities that are re-seeded, since data is not merged.

```
#  
juddi.load.install.data=false
```

This property allows you to cancel loading of the jUDDI install data.

```
# Default locale  
juddi.locale=en_US
```

The default local to use.

```
# Name of the persistence unit to use (the default, "juddiDatabase" refers to the unit compiled  
into the juddi library)  
juddi.persistenceunit.name=juddiDatabase
```

The persistence name for the jUDDI database that is specified in the `persistence.xml` file.

```
# Check-the-time-stamp-on-this-file Interval in milli seconds  
juddi.configuration.reload.delay=2000
```

The time in milliseconds in which `juddiv3.properties` is polled for changes.

```
# These two tokens are referenced in the install data. Note that you  
# can use any tokens, and that their values can be set here or as  
# System parameters.  
juddi.server.name=macdaddy  
juddi.server.port=8080
```

The server name and port number of the server.

```
#The UDDI Operator Contact Email Address  
juddi.operatorEmailAddress=admin@juddi.org
```


The jUDDI operator email address.

6.4. Queries

```
# The maximum number of UDDI artifacts allowed  
# per publisher. A value of '-1' indicates any  
# number of artifacts is valid (These values can be  
# overridden at the individual publisher level).  
juddi.maxBindingsPerService=10
```

The maximum number of bindings that can be specified per service.

```
# The maximum number of UDDI artifacts allowed  
# per publisher. A value of '-1' indicates any  
# number of artifacts is valid (These values can be  
# overridden at the individual publisher level).  
juddi.maxBusinessesPerPublisher=25
```

The maximum number of businesses that can be registered per publisher.

```
# The maximum number of "IN" clause parameters. Some RDMBS limit the number of  
# parameters allowed in a SQL "IN" clause.  
juddi.maxInClause=1000
```

The maximum number of parameters within an IN clause.

```
# The maximum name size and maximum number  
# of name elements allows in several of the  
# FindXxxx and SaveXxxx UDDI functions.  
juddi.maxNameElementsAllowed=5
```

Maximum number of name elements allowed in a jUDDI query.

```
# The maximum name size and maximum number  
# of name elements allows in several of the  
# FindXxxx and SaveXxxx UDDI functions.
```

```
juddi.maxNameLength=255
```

Maximum name size within a jUDDI query.

```
# The maximum number of rows returned in a find_* operation. Each call can set  
# this independently, but this property defines a global maximum.  
juddi.maxRows=1000
```

Maximum number of rows within a response.

```
# The maximum number of UDDI artifacts allowed  
# per publisher. A value of '-1' indicates any  
# number of artifacts is valid (These values can be  
# overridden at the individual publisher level).  
juddi.maxServicesPerBusiness=20
```

Maximum number of services in a business.

```
# The maximum number of UDDI artifacts allowed  
# per publisher. A value of '-1' indicates any  
# number of artifacts is valid (These values can be  
# overridden at the individual publisher level).  
juddi.maxTModelsPerPublisher=100
```

Maximum number of TModels a publisher can create.

6.5. Proxy Settings

```
#only used by RMITransport  
#juddi.proxy.factory.initial =org.jnp.interfaces.NamingContextFactory  
#juddi.proxy.provider.url =jnp://localhost:1099  
#juddi.proxy.factory.url.pkg =org.jboss.naming
```

This is the upper boundary set by the registry. Between the user defined endDate of a Subscription and this value, the registry will pick the earliest date.

6.6. KeyGeneration

```
# jUDDI Cryptor implementation to use  
juddi.cryptor = org.apache.juddi.cryptor.DefaultCryptor
```

Cryptor implementation that jUDDI will use.

```
# jUDDI Key Generator to use  
juddi.keygenerator=org.apache.juddi.keygen.KeyGenerator
```

Key generator implementation that jUDDI will use.

```
# jUDDI UUIDGen implementation to use  
juddi.uuidgen = org.apache.juddi.uuidgen.DefaultUUIDGen
```

UUID generation implementation that jUDDI will use.

6.7. Subscription

```
# Minutes before a "chunked" subscription call expires  
juddi.subscription.chunkexpiration.minutes=5
```

This is the expiration time of a subscription “chunk”.

```
#  
# Days before a subscription expires  
juddi.subscription.expiration.days=30
```

This is the upper boundary set by the registry. Between the user defined endDate of a Subscription and this value, the registry will pick the earliest date.

```
# Specifies the interval at which the notification timer triggers  
juddi.notification.interval=3000000
```

Specifies the interval at which the notification timer triggers.

```
# Specifies the amount of time to wait before the notification timer initially fires  
juddi.notification.start.buffer=20000
```

Specifies the amount of time to wait before the notification timer initially fires.

6.8. Transfer

```
# Days before a transfer request expires  
juddi.transfer.expiration.days=3
```

Days before a transfer request expires.

Using the jUDDI-Client

7.1. Introduction

The jUDDI project includes a jUDDI-Client (`juddi-client-3.0.0.jar`) which can be used to connect to the Registry. The client uses the UDDI v3 API so it should be able to connect to any UDDI v3 compliant registry, however we have only tested it with jUDDIv3. It may be useful to take a look at the unit-tests in the `jUDDIv3-uddi-client` module to see how the client can be used.

7.2. Configuration

The UDDI client has a configuration file called `uddi.xml`. In this file you can set the type “Transport” used by the client to talk to the registry. The client tries to locate this file on the classpath and uses Apache Commons Configuration [COM-CONFIG] to read it. By default the `uddi.xml` file looks like

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<uddi>
  <reloadDelay>5000</reloadDelay>
  <manager name="example-manager">
    <nodes>
      <node>
        <name>default</name>
        <description>Main jUDDI node</description>
        <properties>
          <property name="serverName" value="localhost"/>
          <property name="serverPort" value="8080"/>
          <property name="keyDomain" value="juddi.apache.org"/>
          <property name="department" value="businesses" />
        </properties>
        <proxyTransport>
          org.apache.juddi.v3.client.transport.InVMTransport
        </proxyTransport>
        <custodyTransferUrl>
          org.apache.juddi.api.impl.UDDICustodyTransferImpl
        </custodyTransferUrl>
        <inquiryUrl>org.apache.juddi.api.impl.UDDIInquiryImpl</inquiryUrl>
        <publishUrl>org.apache.juddi.api.impl.UDDIPublicationImpl</publishUrl>
        <securityUrl>org.apache.juddi.api.impl.UDDISecurityImpl</securityUrl>
        <subscriptionUrl>org.apache.juddi.api.impl.UDDISubscriptionImpl
        </subscriptionUrl>
        <subscriptionListenerUrl>
          org.apache.juddi.api.impl.UDDISubscriptionListenerImpl
        </subscriptionListenerUrl>
      </node>
    </nodes>
  </manager>
</uddi>
```

```
    <juddiApiUrl>org.apache.juddi.api.impl.JUDDIApiImpl</juddiApiUrl>
  </node>
</nodes>
</manager>
</uddi>
```

7.3. JAX-WS Transport

Using the settings in the `uddi.xml` file from above, the client will use JAX-WS to communicate with the (remote) registry server. This means that the client needs to have access to a JAX-WS compliant WS stack (such as CXF, Axis2 or JBossWS). Make sure to point the JAXWS URLs to where the UDDI client can find the WSDL documents.

```
<!-- JAX-WS Transport -->
<proxyTransport>org.apache.juddi.v3.client.transport.JAXWSTransport</proxyTransport>
  <custodyTransferUrl>
    http://${serverName}:${serverPort}/juddiv3/services/custody-transfer?wsdl
  </custodyTransferUrl>
  <inquiryUrl>
    http://${serverName}:${serverPort}/juddiv3/services/inquiry?wsdl
  </inquiryUrl>
  <publishUrl>
    http://${serverName}:${serverPort}/juddiv3/services/publish?wsdl
  </publishUrl>
  <securityUrl>
    http://${serverName}:${serverPort}/juddiv3/services/security?wsdl
  </securityUrl>
  <subscriptionUrl>
    http://${serverName}:${serverPort}/juddiv3/services/subscription?wsdl
  </subscriptionUrl>
  <subscriptionListenerUrl>
    http://${serverName}:${serverPort}/juddiv3/services/subscription-listener?wsdl
  </subscriptionListenerUrl>
  <juddiApiUrl>
    http://${serverName}:${serverPort}/juddiv3/services/juddi-api?wsdl
  </juddiApiUrl>
```

7.4. RMI Transport

If jUDDIv3 is deployed to an Application Server it is possible to register the UDDI Services as RMI services. If this is desired you need to edit the `juddiv3.war/WEB-INF/classes/juddiv3.properties` file, on the server. Add the following setting

```
juddi.jndi.registration=true
```

Now at deployment time, the RMI based UDDI services are bound into the Global JNDI namespace.

```
juddi (class: org.jnp.interfaces.NamingContext)
```

- UDDIPublicationService (class: org.apache.juddi.rmi.UDDIPublicationService)
- UDDICustodyTransferService (class: org.apache.juddi.rmi.UDDICustodyTransferService)
- UDDISubscriptionListenerService (class: org.apache.juddi.rmi.UDDISubscriptionListenerService)
- UDDISecurityService (class: org.apache.juddi.rmi.UDDISecurityService)
- UDDISubscriptionService (class: org.apache.juddi.rmi.UDDISubscriptionService)
- UDDIInquiryService (class: org.apache.juddi.rmi.UDDIInquiryService)

Next, on the client side you need to comment out the JAXWS section in the `uddi.properties` file and use the RMI Transport section instead. Optionally you can set the `java.naming.*` properties. In this case we specified setting for connecting to jUDDIv3 deployed to a JBoss Application Server. If you like you can set the `java.naming.*` properties in a `jndi.properties` file, or as System parameters.

7.5. InVM Transport

If you choose to use InVM Transport this means that the jUDDIv3 server is running in the same VM as you client. If you are deploying to `juddi.war` the server will be started by the `org.apache.juddi.RegistryServlet`, but if you are running outside any container, you are responsible for starting and stopping the `org.apache.juddi.Registry` Service yourself. Make sure to call

```
Registry.start()
```

before making any calls to the Registry, and when you are done using the Registry (on shutdown) call

```
Registry.stop()
```

so the Registry can release any resources it may be holding. To use InVM Transport uncomment this section in the `uddi.properties` while commenting out the JAXWS and RMI Transport sections.

7.6. Dependencies

The UDDI client depends on `uddi-ws-3.0.0.jar`, `commons-configuration-1.5.jar`, `commons-collection-3.2.1.jar` and `log4j-1.2.13.jar`, plus

- libraries for JAXB if you are not using JDK5.
- JAXWS client libraries when using JAXWS transport (like CXF).
- RMI and JNDI client libraries when using RMI Transport.

7.7. Sample Code

Sample code on how to use the UDDI client can be found in the `uddi-client` module on the jUDDIv3 project. Usually the first thing you want to is to make a call to the Registry to obtain an Authentication Token. The following code is taken from the unit tests in this module.

```
public void testAuthToken() {
    try {
        String clazz = ClientConfig.getConfiguration().getString(
            Property.UDDI_PROXY_TRANSPORT,Property.DEFAULT_UDDI_PROXY_TRANSPORT);
        Class<?> transportClass = Loader.loadClass(clazz);
        if (transportClass!=null) {
            Transport transport = (Transport) transportClass.newInstance();
            UDDISecurityPortType securityService = transport.getSecurityService();
            GetAuthToken getAuthToken = new GetAuthToken();
            getAuthToken.setUserID("root");
            getAuthToken.setCred("");
            AuthToken authToken = securityService.getAuthToken(getAuthToken);
            System.out.println(authToken.getAuthInfo());
            Assert.assertNotNull(authToken);
        } else {
            Assert.fail();
        }
    } catch (Exception e) {
        e.printStackTrace();
        Assert.fail();
    }
}
```


Make sure that the publisher, in this case “root” is an existing publisher in the Registry and that you are supplying the correct credential to get a successful response. If needed check [Chapter 3, Authentication](#) to learn more about this subject.

Another place to look for sample code is the `docs/examples/helloworld` directory. Alternatively you can use annotations.

UDDI Annotations

8.1. Introduction

Conventionally Services (`BusinessService`) and their EndPoints (`BindingTemplates`) are registered to a UDDI Registry using a GUI, where an admin user manually adds the necessary info. This process tends to make the data in the Registry rather static and the data can grow stale over time. To make the data in the UDDI more dynamic it makes sense to register and EndPoint (`BindingTemplate`) when it comes online, which is when it gets deployed. The UDDI annotations are designed to just that: register a Service when it get deployed to an Application Server. There are two annotations: `UDDIService`, and `UDDIServiceBinding`. You need to use both annotations to register an EndPoint. Upon undeployment of the Service, the EndPoint will be de-registered from the UDDI. The Service information stays in the UDDI. It makes sense to leave the Service level information in the Registry since this reflects that the Service is there, however there is no EndPoint at the moment ("Check back later"). It is a manual process to remove the Service information. The annotations use the `juddi-client` library which means that they can be used to register to any UDDIv3 registry.

8.2. UDDIService Annotation

The `UDDIService` annotation is used to register a service under an already existing business in the Registry. The annotation should be added at the class level of the java class.

Table 8.1. UDDIService attributes

attribute	description	required
<code>serviceName</code>	The name of the service, by default the clerk will use the one name specified in the <code>WebService</code> annotation	no
<code>description</code>	Human readable description of the service	yes
<code>serviceKey</code>	UDDI v3 Key of the Service	yes
<code>businessKey</code>	UDDI v3 Key of the Business that should own this Service. The business should exist in the registry at time of registration	yes
<code>lang</code>	Language locale which will be used for the name and description, defaults to "en" if omitted	no

attribute	description	required
categoryBag	Definition of a CategoryBag, see below for details	no

8.3. UDDIServiceBinding Annotation

The UDDIServiceBinding annotation is used to register a BindingTemplate to the UDDI registry. This annotation cannot be used by itself. It needs to go along side a UDDIService annotation.

Table 8.2. UDDIServiceBinding attributes

attribute	description	required
bindingKey	UDDI v3 Key of the ServiceBinding	yes
description	Human readable description of the service	yes
accessPointType	UDDI v3 AccessPointType, defaults to wsdlDeployment if omitted	no
accessPoint	Endpoint reference	yes
lang	Language locale which will be used for the name and description, defaults to "en" if omitted	no
tModelKeys	Comma-separated list of tModelKeys key references	no
categoryBag	Definition of a CategoryBag, see below for further details	no

8.4. WebService Example

The annotations can be used on any class that defines a service. Here they are added to a WebService, a POJO with a JAX-WS 'WebService' annotation.

```
package org.apache.juddi.samples;

import javax.jws.WebService;
import org.apache.juddi.v3.annotations.UDDIService;
import org.apache.juddi.v3.annotations.UDDIServiceBinding;

@UDDIService(
    businessKey="uddi:myBusinessKey",
```

```

serviceKey="uddi:myServiceKey",
description = "Hello World test service")
@UDDIServiceBinding(
bindingKey="uddi:myServiceBindingKey",
description="WSDL endpoint for the helloWorld Service. This service is used for "
+ "testing the jUDDI annotation functionality",
accessPointType="wsdlDeployment",
accessPoint="http://localhost:8080/juddiv3-samples/services/helloworld?wsdl")
@WebService(
endpointInterface = "org.apache.juddi.samples.HelloWorld",
serviceName = "HelloWorld")

public class HelloWorldImpl implements HelloWorld {
public String sayHi(String text) {
System.out.println("sayHi called");
return "Hello " + text;
}
}

```

On deployment of this WebService, the juddi-client code will scan this class for UDDI annotations and take care of the registration process. The configuration file `uddi.xml` of the juddi-client is described in the chapter [Chapter 7, Using the jUDDI-Client](#). In the clerk section you need to reference the Service class 'org.apache.juddi.samples.HelloWorldImpl':

```

<clerk name="BobCratchit" node="default" publisher="sales" password="sales">
  <class>org.apache.juddi.samples.HelloWorldImpl</class>
</clerk>

```

which means that Bob is using the node connection setting of the node with name "default", and that he will be using the "sales" publisher, for which the password is "sales". There is some analogy here as to how datasources are defined.

8.5. CategoryBag Attribute

The CategoryBag attribute allows you to reference tModels. For example the following categoryBag

```

<categoryBag>
  <keyedReference tModelKey="uddi:uddi.org:categorization:types"

```

```
keyName="uddi-org:types:wsdl" keyValue="wsdlDeployment" />
<keyedReference tModelKey="uddi:uddi.org:categorization:types"
keyName="uddi-org:types:wsdl2" keyValue="wsdlDeployment2" />
</categoryBag>
```

can be put in like

```
categoryBag="keyedReference=keyName=uddi-org:types:wsdl;keyValue=wsdlDeployment;" +
    "tModelKey=uddi:uddi.org:categorization:types," +
    "keyedReference=keyName=uddi-org:types:wsdl2;keyValue=wsdlDeployment2;" +
    "tModelKey=uddi:uddi.org:categorization:types2",
```

Simple Publishing Using the jUDDI API

One of the most common requests we get on the message board is “How do I publish a service using jUDDI?” This question holds a wide berth, as it can result anywhere from not understanding the UDDI data model, to confusion around how jUDDI is set up, to the order of steps required to publish artifacts in the registry, to general use of the API – and everything in between. This article will attempt to answer this “loaded” question and, while not going into too much detail, will hopefully clear some of the confusion about publishing into the jUDDI registry.

9.1. UDDI Data Model

Before you begin publishing artifacts, you need to know exactly how to break down your data into the UDDI model. This topic is covered extensively in the specification, particularly in section 3, so I only want to gloss over some for details. Readers interested in more extensive coverage should most definitely take a look at the UDDI specification.

Below is a great diagram of the UDDI data model (taken directly from the specification):

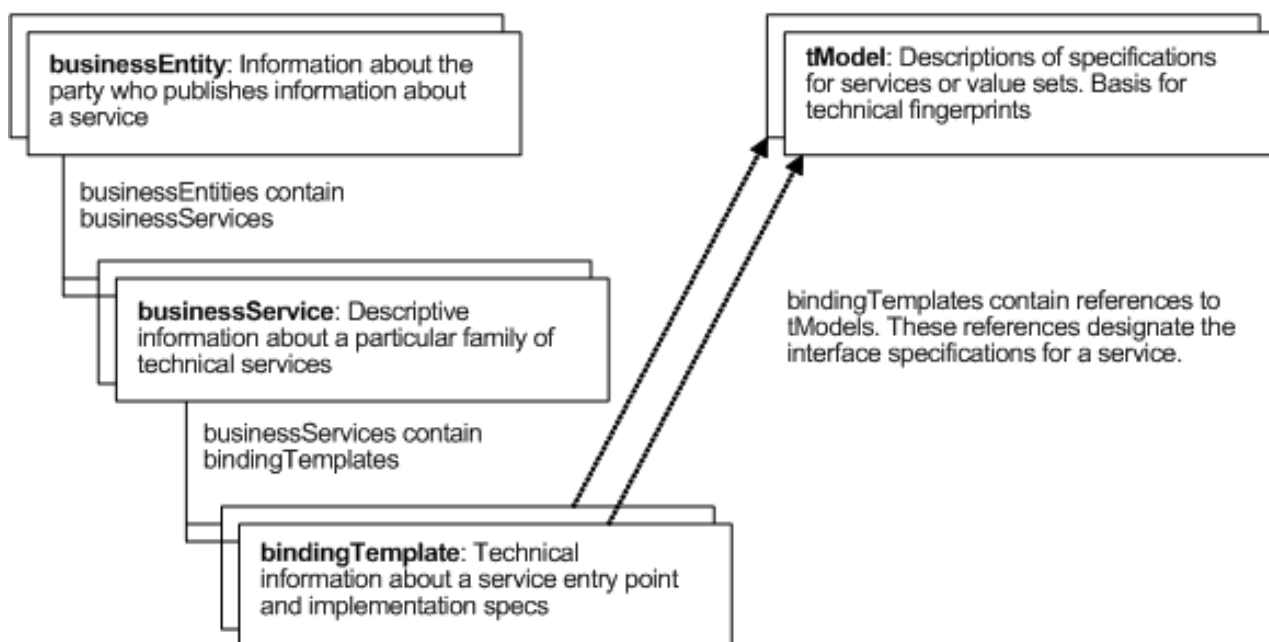


Figure 9.1. UDDI Core Data Structures

As you can see, data is organized into a hierarchical pattern. Business Entities are at the top of the pyramid, they contain Business Services and those services in turn contain Binding Templates. TModels (or technical models) are a catch-all structure that can do anything from categorize one of the main entities, describe the technical details of a binding (ex. protocols, transports, etc), to

registering a key partition. TModels won't be covered too much in this article as I want to focus on the three main UDDI entities.

The hierarchy defined in the diagram is self-explanatory. You must first have a Business Entity before you can publish any services. And you must have a Business Service before you can publish a Binding Template. There is no getting around this structure; this is the way UDDI works.

Business Entities describe the organizational unit responsible for the services it publishes. It generally consist of a description and contact information. How one chooses to use the Business Entity is really dependent on the particular case. If you're one small company, you will likely just have one Business Entity. If you are a larger company with multiple departments, you may want to have a Business Entity per department. (The question may arise if you can have one uber-Business Entity and multiple child Business Entities representing the departments. The answer is yes, you can relate Business Entities using Publisher Assertions, but that is beyond the scope of this article.)

Business Services are the cogs of the SOA landscape. They represent units of functionality that are consumed by clients. In UDDI, there's not much to a service structure; mainly descriptive information like name, description and categories. The meat of the technical details about the service is contained in its child Binding Templates.

Binding Templates, as mentioned above, give the details about the technical specification of the service. This can be as simple as just providing the service's access point, to providing the location of the service WSDL to more complicated scenarios to breaking down the technical details of the WSDL (when used in concert with tModels). Once again, getting into these scenarios is beyond the scope of this article but may be the subject of future articles.

9.2. jUDDI Additions to the Model

Out of the box, jUDDI provides some additional structure to the data model described in the specification. Primarily, this is the concept of the Publisher.

The UDDI specification talks about ownership of the entities that are published within the registry, but makes no mention about how ownership should be handled. Basically, it is left up to the particular implementation to decide how to handle "users" that have publishing rights in the registry.

Enter the jUDDI Publisher. The Publisher is essentially an out-of-the-box implementation of an identity management system. Per the specification, before assets can be published into the registry, a "publisher" must authenticate with the registry by retrieving an authorization token. This authorization token is then attached to future publish calls to assign ownership to the published entities.

jUDDI's Publisher concept is really quite simple, particularly when using the default authentication. You can save a Publisher to the registry using jUDDI's custom API and then use that Publisher to publish your assets into the registry. jUDDI allows for integration into your own identity management system, circumventing the Publisher entirely if desired. This is discussed in more

detail in the documentation, but for purposes of this article, we will be using the simple out-of-the-box Publisher solution.

One quick note: ownership is essentially assigned to a given registry entity by using its “authorizedName” field. The “authorizedName” field is defined in the specification in the operationalInfo structure which keeps track of operational info for each entity.

9.3. UDDI and jUDDI API

Knowing the UDDI data model is all well and good. But to truly interact with the registry, you need to know how the UDDI API is structured and how jUDDI implements this API. The UDDI API is covered in great detail in chapter 5 of the specification but will be summarized here.

UDDI divides their API into several “sets” – each representing a specific area of functionality. The API sets are listed below:

- Inquiry – deals with querying the registry to return details on entities within
- Publication – handles publishing entities into the registry
- Security – open-ended specification that handles authentication
- Custody and Ownership Transfer – deals with transferring ownership and custody of entities
- Subscription – allows clients to retrieve information on entities in a timely manner using a subscription format
- Subscription Listener – client API that accepts subscription results
- Value Set (Validation and Caching)– validates keyed reference values (not implemented by jUDDI)
- Replication – deals with federation of data between registry nodes (not implemented by jUDDI)

The most commonly used APIs are the Inquiry, Publication and Security APIs. These APIs provide the standard functions for interacting with the registry.

The jUDDI server implements each of these API sets as a JAX-WS compliant web service and each method defined in the API set is simply a method in the corresponding web service. The client module provided by jUDDI uses a “transport” class that defines how the call is to be made. The default transport uses JAX-WS but there are several alternative ways to make calls to the API. Please refer to the documentation for more information.

One final note, jUDDI defines its own API set. This API set contains methods that deal with handling Publishers as well as other useful maintenance functions (mostly related to jUDDI’s subscription model). This API set is obviously proprietary to jUDDI and therefore doesn’t conform to the UDDI specification.

9.4. Getting Started

Now that we've covered the basics of the data model and API sets, it's time to get started with the publishing sample. The first thing that must happen is to get the jUDDI server up and running. Please refer to this [article](http://apachejuddi.blogspot.com/2010/02/getting-started-with-juddi-v3.html) [http://apachejuddi.blogspot.com/2010/02/getting-started-with-juddi-v3.html] that explains how to start the jUDDI server.

9.4.1. Simple Publishing Example

We will now go over the "simple-publish" example found in the documentation. This sample expands upon the HelloWorld example in that after retrieving an authentication token, a Publisher, BusinessEntity and BusinessService are published to the registry.

The sample consists of only one class: SimplePublish. Let's start by taking a look at the constructor:

```
public SimplePublish() {
    try {
        String clazz = UDDIClientContainer.getUDDIClerkManager(null).
            getClientConfig().getUDDINode("default").getProxyTransport();
        Class<?> transportClass = ClassUtil.forName(clazz, Transport.class);
        if (transportClass!=null) {
            Transport transport = (Transport) transportClass.
                getConstructor(String.class).newInstance("default");

            security = transport.getUDDISecurityService();
            juddiApi = transport.getJUDDIApiService();
            publish = transport.getUDDIPublishService();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The constructor uses the jUDDI client API to retrieve the transport from the default node. You can refer to the documentation if you're confused about how clerks and nodes work. Suffice it to say, we are simply retrieving the default client transport class which is designed to make UDDI calls out using JAX-WS web services.

Once the transport is instantiated, we grab the three API sets we need for this demo: 1) the Security API set so we can get authorization tokens, 2) the proprietary jUDDI API set so we can save a Publisher and 3) the Publication API set so we can actually publish entities to the registry.

All the magic happens in the publish method. We will look at that next.

Here are the first few lines of the publish method:

```
// Setting up the values to get an authentication token for the 'root' user ('root' user
// has admin privileges and can save other publishers).
GetAuthToken getAuthTokenRoot = new GetAuthToken();
getAuthTokenRoot.setUserID("root");
getAuthTokenRoot.setCred("");

// Making API call that retrieves the authentication token for the 'root' user.
AuthToken rootAuthToken = security.getAuthToken(getAuthTokenRoot);
System.out.println ("root AUTHTOKEN = " + rootAuthToken.getAuthInfo());
```

This code simply gets the authorization token for the 'root' user. The 'root' user (or publisher) is automatically installed in every jUDDI instance and acts as the "administrator" for jUDDI API calls. Additionally, the 'root' user is the owning publisher for all the initial services installed with jUDDI. You may be wondering what those "initial services" are. Well, since the UDDI API sets are all implemented as web services by jUDDI, every jUDDI node actually registers those services inside itself. This is done per the specification.

Let's get back to the code. Now that we have root authorization, we can add a publisher:

```
// Creating a new publisher that we will use to publish our entities to.
Publisher p = new Publisher();
p.setAuthorizedName("my-publisher");
p.setPublisherName("My Publisher");

// Adding the publisher to the "save" structure, using the 'root' user authentication info and
// saving away.
SavePublisher sp = new SavePublisher();
sp.getPublisher().add(p);
sp.setAuthInfo(rootAuthToken.getAuthInfo());
juddiApi.savePublisher(sp);
```

Here we've simply used the jUDDI API to save a publisher with authorized name "my-publisher". Notice how the authorization token for the 'root' user is used. Next, we need to get the authorization token for this new publisher:

```
// Our publisher is now saved, so now we want to retrieve its authentication token
GetAuthToken getAuthTokenMyPub = new GetAuthToken();
getAuthTokenMyPub.setUserID("my-publisher");
getAuthTokenMyPub.setCred("");
AuthToken myPubAuthToken = security.getAuthToken(getAuthTokenMyPub);
System.out.println ("myPub AUTHTOKEN = " + myPubAuthToken.getAuthInfo());
```

This is pretty straightforward. You'll note that no credentials have been set on both authorization calls. This is because we're using the default authenticator which doesn't require credentials. We have our authorization token for our new publisher, now we can simply publish away:

```
// Creating the parent business entity that will contain our service.
BusinessEntity myBusEntity = new BusinessEntity();
Name myBusName = new Name();
myBusName.setValue("My Business");
myBusEntity.getName().add(myBusName);

// Adding the business entity to the "save" structure, using our publisher's authentication info
// and saving away.
SaveBusiness sb = new SaveBusiness();
sb.getBusinessEntity().add(myBusEntity);
sb.setAuthInfo(myPubAuthToken.getAuthInfo());
BusinessDetail bd = publish.saveBusiness(sb);
String myBusKey = bd.getBusinessEntity().get(0).getBusinessKey();
System.out.println("myBusiness key: " + myBusKey);

// Creating a service to save. Only adding the minimum data: the parent business key
retrieved
//from saving the business above and a single name.
BusinessService myService = new BusinessService();
myService.setBusinessKey(myBusKey);
Name myServName = new Name();
myServName.setValue("My Service");
myService.getName().add(myServName);
// Add binding templates, etc...

// Adding the service to the "save" structure, using our publisher's authentication info and
// saving away.
SaveService ss = new SaveService();
```

```
ss.getBusinessService().add(myService);
ss.setAuthInfo(myPubAuthToken.getAuthInfo());
ServiceDetail sd = publish.saveService(ss);
String myServKey = sd.getBusinessService().get(0).getServiceKey();
System.out.println("myService key: " + myServKey);
```

To summarize, here we have created and saved a BusinessEntity and then created and saved a BusinessService. We've just added the bare minimum data to each entity (and in fact, have not added any BindingTemplates to the service). Obviously, you would want to fill out each structure with greater information, particularly with services. However, this is beyond the scope of this article, which aims to simply show you how to programmatically publish entities.

There are a couple important notes regarding the use of entity keys. Version 3 of the specification allows for publishers to create their own keys but also instructs implementers to have a default method. Here we have gone with the default implementation by leaving each entity's "key" field blank in the save call. jUDDI's default key generator simply takes the node's partition and appends a GUID. In a default installation, it will look something like this:

```
uddi:juddi.apache.org:<GUID>
```

You can, of course, customize all of this, but that is left for another article. The second important point is that when the BusinessService is saved, I've had to explicitly set its parent business key (retrieved from previous call saving the business). This is a necessary step when the service is saved in an independent call like this. Otherwise you would get an error because jUDDI won't know where to find the parent entity. I could have added this service to the BusinessEntity's service collection and saved it with the call to saveBusiness. In that scenario I would not have to set the parent business key.

9.5. Conclusion

That does it for this article. Hopefully I managed to clear some of the confusion around the open-ended question, "How do I publish a service using jUDDI?".

Subscription

10.1. Introduction

Subscriptions come to play in a multi-registry setup. Within your company you may have the need to run with more than one UDDI, let's say one for each department, where you limit access to the systems in each department to just their own UDDI node. However you may want to share some services cross departments. The subscription API can help you cross registering those services and keeping them up to date by sending out notifications as the registry information in the parent UDDI changes.

There are two type of subscriptions:

asynchronous

Save a subscription, and receive updates on a certain schedule.

synchronous

Save a subscription and invoke the `get_Subscription` and get a synchronous reply.

The notification can be executed in a synchronous and an asynchronous way. The asynchronous way requires a listener service to be installed on the node to which the notifications should be sent.

10.2. Two node example setup: Sales and Marketing

In this example we are setting up a node for 'sales' and a node for 'marketing'. For this you need to deploy jUDDI to two different services, then you need to do the following setup:

Procedure 10.1. Setup Node 1: Sales

1. Create `juddi_custom_install_data`.

```
cd juddiv3/WEB-INF/classes
mv RENAME4SALES_juddi_custom_install_data juddi_custom_install_data
```

2. edit: `webapps/juddiv3/WEB-INF/classes/juddiv3.properties` and set the following property values where 'sales' is the DNS name of your server.

```
juddi.server.name=sales
juddi.server.port=8080
```

3. Start the server (tomcat), which will load the UDDI seed data (since this is the first time you're starting jUDDI, see [Chapter 5, Root Seed Data](#))

```
bin/startup.sh
```

4. Open your browser to <http://sales:8080/juddiv3>. You should see:

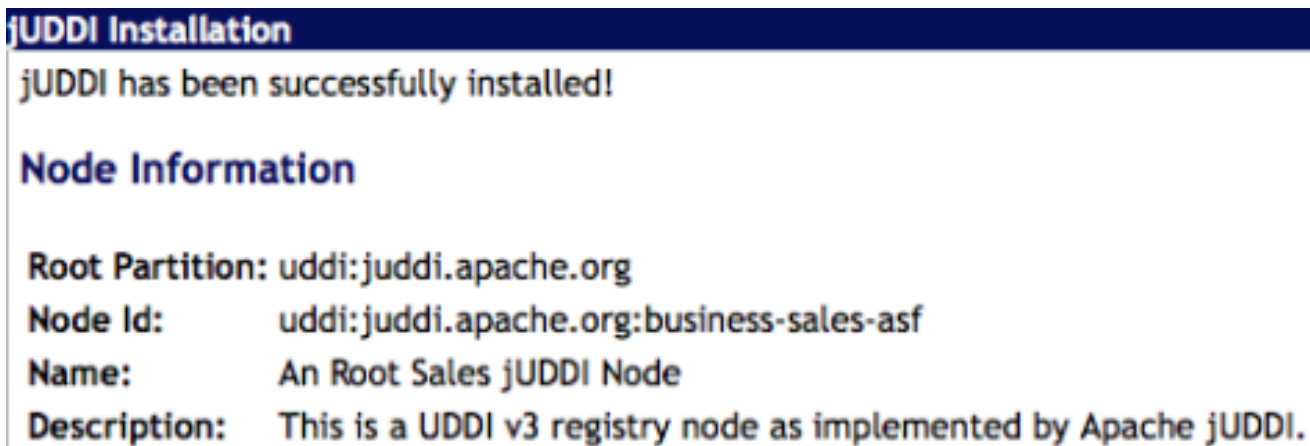


Figure 10.1. Sales Node Installation

Procedure 10.2. Setup Node 2: Marketing

1. Create `juddi_custom_install_data`.

```
cd juddiv3/WEB-INF/classes
mv RENAME4MARKETING_juddi_custom_install_data juddi_custom_install_data
```

2. edit: `webapps/juddiv3/WEB-INF/classes/juddiv3.properties` and set the following property values where 'marketing' is the DNS name of your server.

```
juddi.server.name=marketing
juddi.server.port=8080
```

3. Start the server (tomcat), which will load the UDDI seed data (since this is the first time you're starting jUDDI, see [Chapter 5, Root Seed Data](#))

```
bin/startup.sh
```

4. Open your browser to <http://marketing:8080/juddiv3>. You should see:

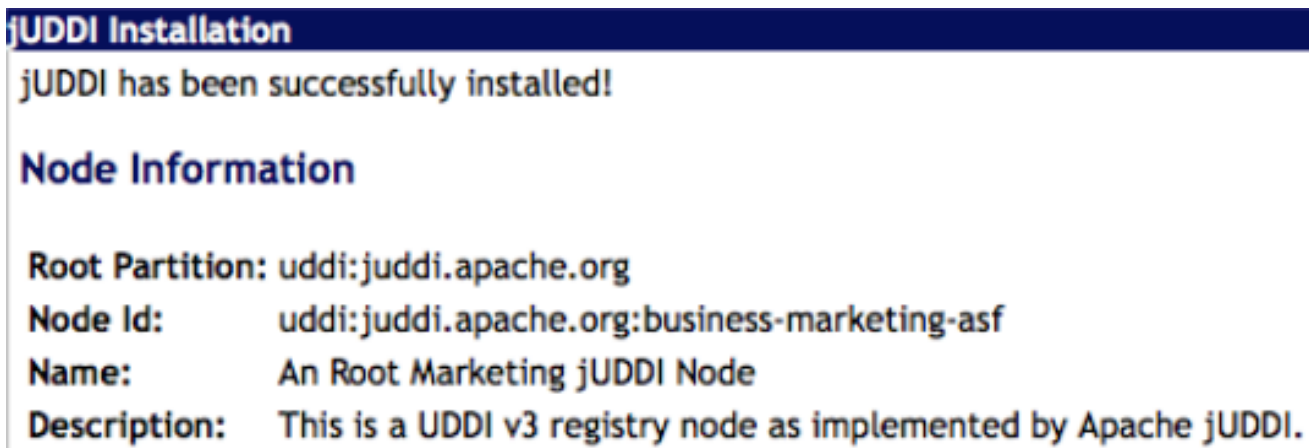


Figure 10.2. Marketing Node Installation

Note that we kept the root partition the same as sales and marketing are in the same company, however the Node Id and Name are different and reflect that this node is in 'sales' or 'marketing'.

Finally you will need to replace the sales server's `uddi-portlets.war/WEB-INF/classes/META-INF/uddi.xml` with `uddi-portlets.war/WEB-INF/classes/META-INF/uddi.xml.sales`. Then, edit the `uddi-portlets.war/WEB-INF/classes/META-INF/uddi.xml` and set the following properties:

```
<name>default</name>
<properties>
  <property name="serverName" value="sales"/>
  <property name="serverPort" value="8080"/>
  <property name="rmiPort" value="1099"/>
</properties>
```

Log into the sales portal: <http://sales:8080/pluto> with username/password: sales/sales.



Figure 10.3. Sales Services

Before logging into the marketing portal, replace marketing's `uddi-portlet.war/WEB-INF/classes/META-INF/uddi.xml` with `udd-portlet.war/WEB-INF/classes/META-INF/uddi.xml.marketing`. Then you will need to edit the `uddi-portlet.war/WEB-INF/classes/META-INF/uddi.xml` and set the following properties:

```
<name>default</name>
<properties>
  <property name="serverName" value="marketing"/>
  <property name="serverPort" value="8080"/>
  <property name="rmiPort" value="1099"/>
</properties>
```

Now log into the marketing portal <http://marketing:8080/pluto> with username/password: marketing/marketing. In the browser for the marketing node we should now see:

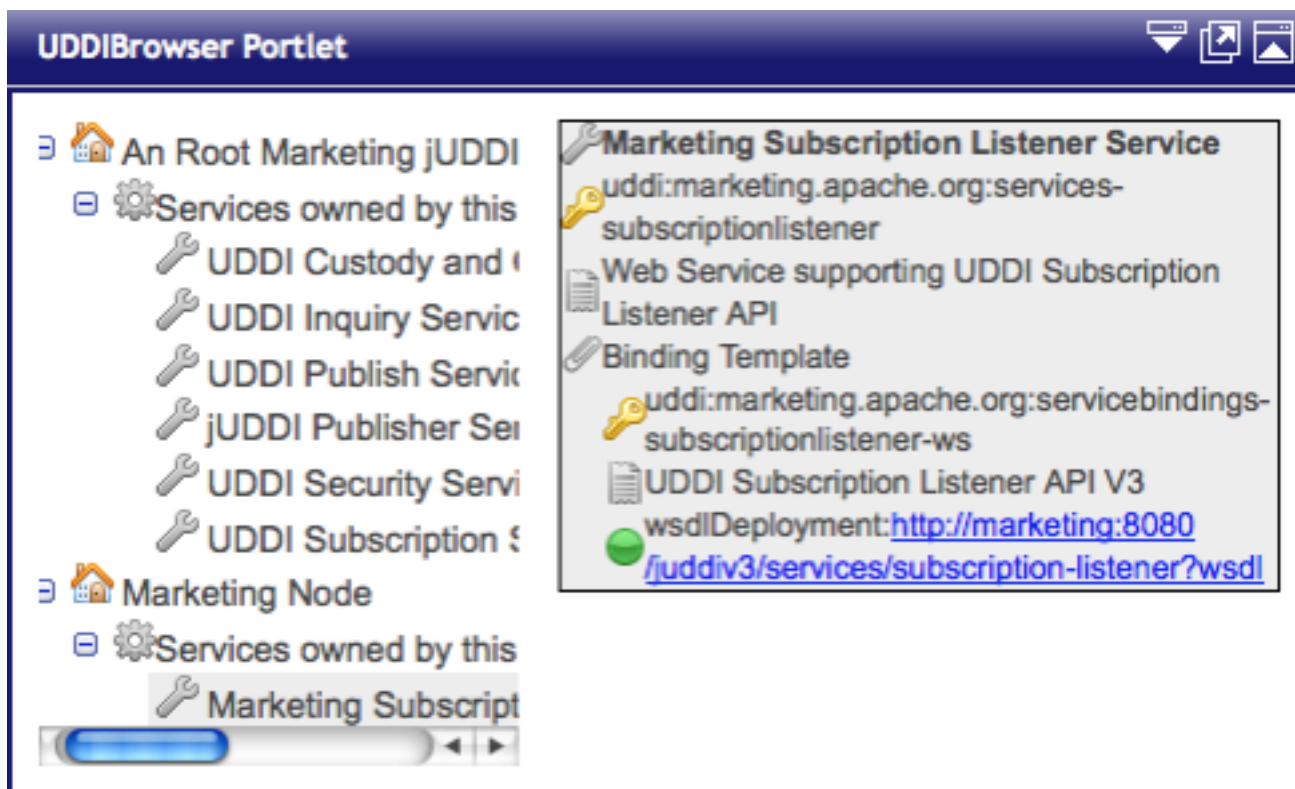


Figure 10.4. Marketing Services

Note that the subscriptionlistener is owned by the Marketing Node business (and not the Root Marketing Node). The Marketing Node Business is managed by the marketing publisher.

10.3. Deploy the HelloSales Service

The sales department developed a service called HelloSales. The HelloSales service is provided in the `juddiv3-samples.war`, and it is annotated so that it will auto-register. Before deploying the war, edit the `juddiv3-samples.war/WEB-INF/classes/META-INF/uddi.xml` file to set some property values to 'sales'.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<uddi>
  <reloadDelay>5000</reloadDelay>
  <manager name="example-manager">
    <nodes>
      <node>
        <name>default</name>
        <description>Sales jUDDI node</description>
        <properties>
          <property name="serverName" value="sales"/>
          <property name="serverPort" value="8080"/>
        </properties>
      </node>
    </nodes>
  </manager>
</uddi>
```

```
    <property name="keyDomain" value="sales.apache.org"/>
    <property name="department" value="sales" />
</properties>
<proxyTransport>
    org.apache.juddi.v3.client.transport.InVMTransport
</proxyTransport>
<custodyTransferUrl>
    org.apache.juddi.api.impl.UDDICustodyTransferImpl
</custodyTransferUrl>
<inquiryUrl>org.apache.juddi.api.impl.UDDIInquiryImpl</inquiryUrl>
<publishUrl>org.apache.juddi.api.impl.UDDIPublicationImpl</publishUrl>
<securityUrl>org.apache.juddi.api.impl.UDDISecurityImpl</securityUrl>
<subscriptionUrl>
    org.apache.juddi.api.impl.UDDISubscriptionImpl
</subscriptionUrl>
<subscriptionListenerUrl>
    org.apache.juddi.api.impl.UDDISubscriptionListenerImpl
</subscriptionListenerUrl>
<juddiApiUrl>org.apache.juddi.api.impl.JUDDIApiImpl</juddiApiUrl>
</node>
</nodes>
</manager>
</uddi>
```

Now deploy the `juddiv3-samples.war` to the sales registry node, by building the `juddiv3-samples.war` and deploying. The HelloWorld service should deploy

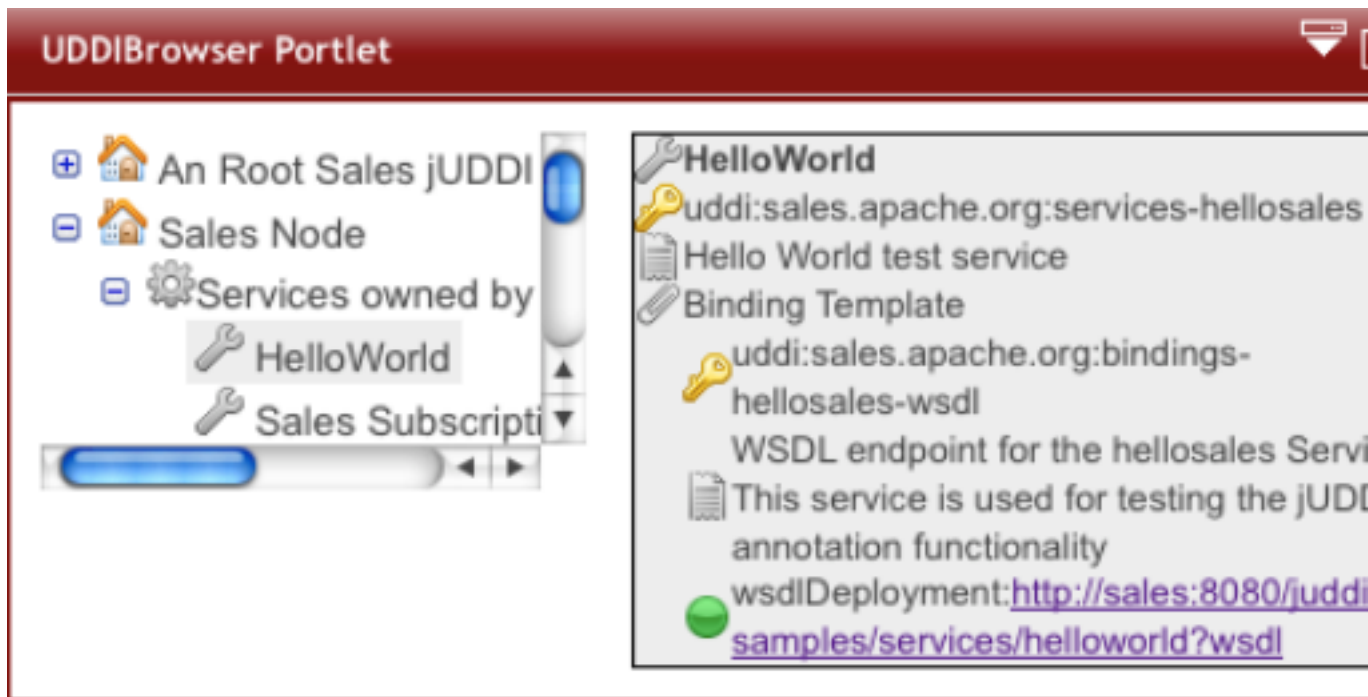


Figure 10.5. Registration by Annotation, deploying the `juddi-samples.war` to the sales Node

On the Marketing UDDI we'd like to subscribe to the HelloWorld service, in the Sales UDDI Node. As mentioned before there are two ways to do this; synchronously and asynchronously.

10.4. Configure a user to create Subscriptions

For a user to create and save subscriptions the publisher needs to have a valid login to both the sales and the marketing node. Also if the marketing publisher is going to create registry objects in the marketing node, the marketing publisher needs to own the sales keygenerator tModel. Check the `marketing_*.xml` files in the root seed data of both the marketing and sales node, if you want to learn more about this. It is important to understand that the 'marketing' publisher in the marketing registry owns the following tModels:

```
<save_tModel xmlns="urn:uddi-org:api_v3">

  <tModel tModelKey="uddi:marketing.apache.org:keygenerator" xmlns="urn:uddi-org:api_v3">
    <name>marketing-apache-org:keyGenerator</name>
    <description>Marketing domain key generator</description>
    <overviewDoc>
      <overviewURL useType="text">
        http://uddi.org/pubs/uddi_v3.htm#keyGen
      </overviewURL>
    </overviewDoc>
  </tModel>
</save_tModel>
```

```
<categoryBag>
  <keyedReference tModelKey="uddi:uddi.org:categorization:types"
    keyName="uddi-org:types:keyGenerator"
    keyValue="keyGenerator" />
</categoryBag>
</tModel>

<tModel tModelKey="uddi:marketing.apache.org:subscription:keygenerator"
  xmlns="urn:uddi-org:api_v3">
  <name>marketing-apache-org:subscription:keyGenerator</name>
  <description>Marketing Subscriptions domain key generator</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#keyGen
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="uddi:uddi.org:categorization:types"
      keyName="uddi-org:types:keyGenerator"
      keyValue="keyGenerator" />
  </categoryBag>
</tModel>

<tModel tModelKey="uddi:sales.apache.org:keygenerator" xmlns="urn:uddi-org:api_v3">
  <name>sales-apache-org:keyGenerator</name>
  <description>Sales Root domain key generator</description>
  <overviewDoc>
    <overviewURL useType="text">
      http://uddi.org/pubs/uddi_v3.htm#keyGen
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="uddi:uddi.org:categorization:types"
      keyName="uddi-org:types:keyGenerator"
      keyValue="keyGenerator" />
  </categoryBag>
</tModel>
</save_tModel>
```

If we are going to use the marketing publisher to subscribe to updates in the sales registry, then we need to provide this publisher with two clerks in the `uddi.xml` of the `uddi-portlet.war`.

```
<clerks registerOnStartup="false">
```

```

<clerk name="MarketingCratchit" node="default"
  publisher="marketing" password="marketing"/>

<clerk name="SalesCratchit" node="sales-ws"
  publisher="marketing" password="marketing"/>
<!-- optional
<xregister>
  <servicebinding
    entityKey="uddi:marketing.apache.org:servicebindings-subscriptionlistener-ws"
    fromClerk="MarketingCratchit" toClerk="SalesCratchit"/>
  </xregister>
-->
</clerks>

```

Here we created two clerks for this publisher called 'MarketingCratchit' and 'SalesCratchit'. This will allow the publisher to check the existing subscriptions owned by this publisher in each of the two systems.

10.5. Synchronous Notifications

While being logged in as the marketing publisher on the marketing portal, we should see the following when selecting the UDDISubscription Portlet.

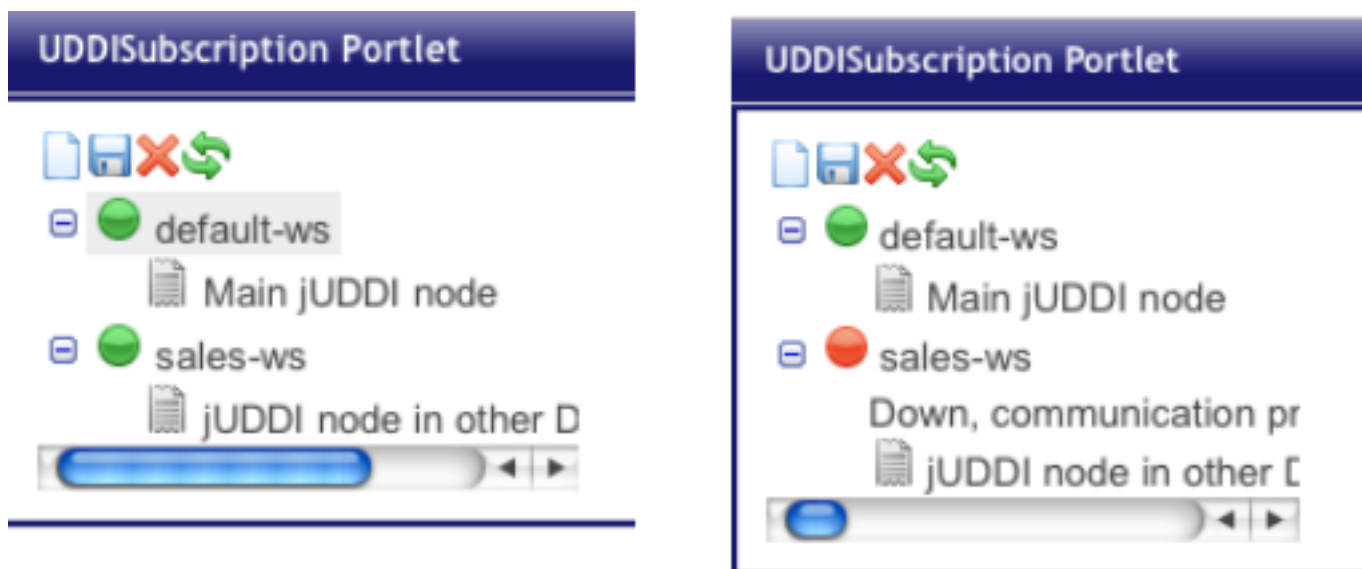


Figure 10.6. Subscriptions. In (a) both nodes are up while in (b) the sales node is down

When both nodes came up green you can click on the 'new subscription' icon in the toolbar. Since we are going to use this subscription synchronously only the Binding Key and Notification Interval

should be left blank, as shown in *Figure 10.7, “Create a New Subscription”*. Click the save icon to save the subscription.

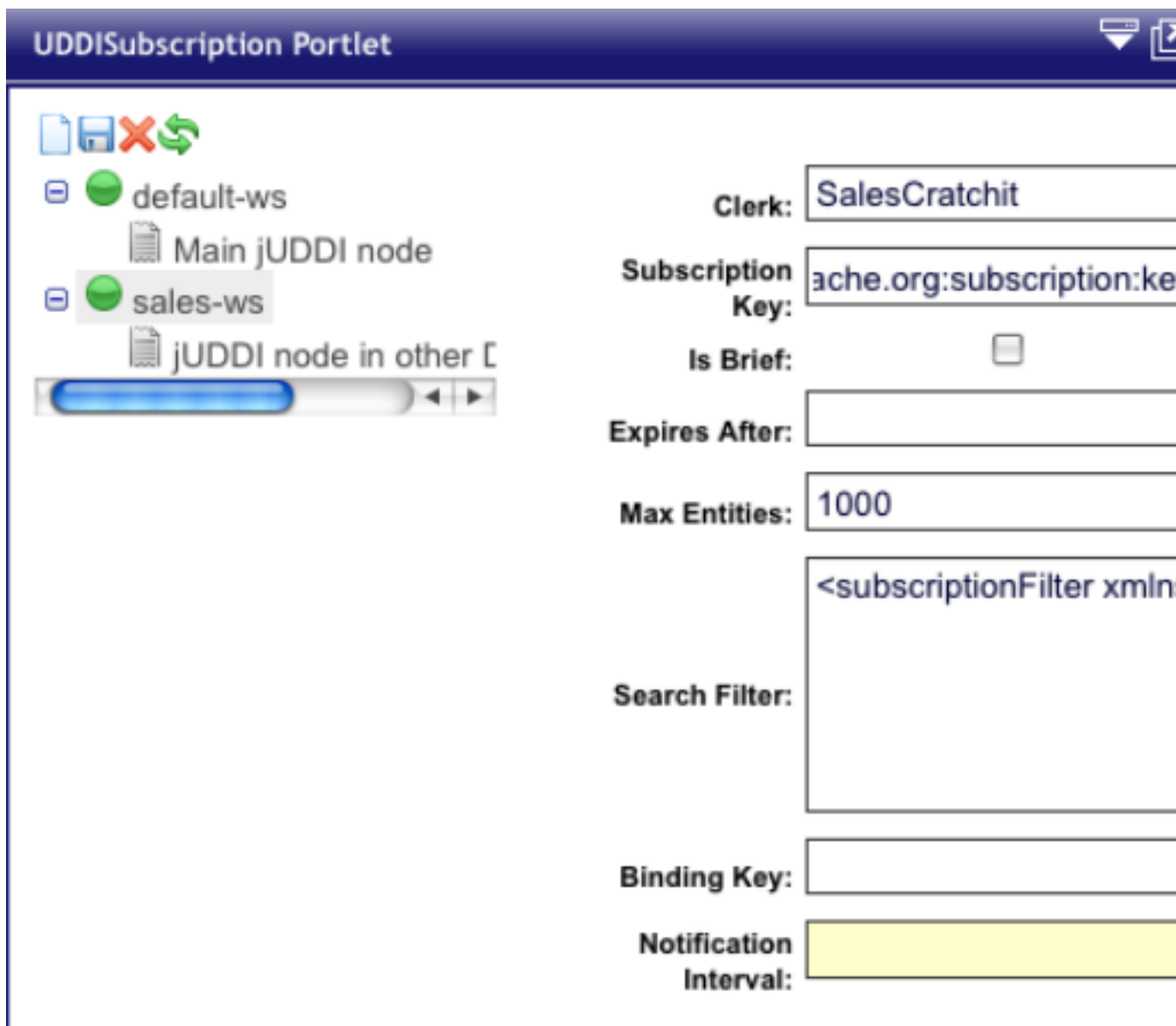


Figure 10.7. Create a New Subscription

Make sure that the subscription Key uses the convention of the keyGenerator of the marketing publisher. You should see the orange subscription icon appear under the “sales-ws” UDDI node.



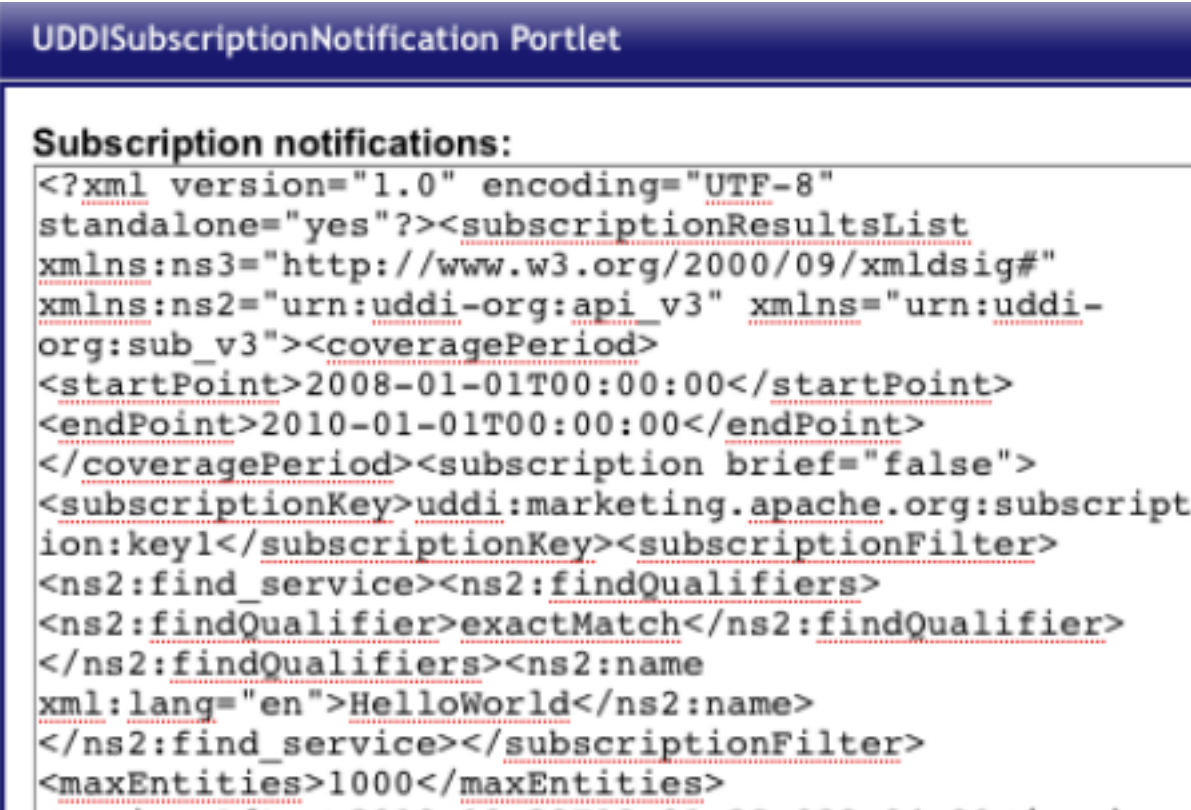
Figure 10.8. A Newly Saved Subscription

To invoke a synchronous subscription, click the icon with the green arrows. This will give you the opportunity to set the coverage period.



Figure 10.9. Set the Coverage Period

Click the green arrows icon again to invoke the synchronous subscription request. The example finder request will go out to the sales node and look for updates on the HelloWorld service. The raw XML response will be posted in the UDDISubscriptionNotification Portlet.

The image shows a screenshot of a web application window titled "UDDISubscriptionNotification Portlet". The window has a dark blue header bar with the title and some navigation icons on the right. Below the header, the main content area displays the text "Subscription notifications:" followed by a large block of raw XML code. The XML code is displayed in a monospaced font and contains various XML tags and attributes, including namespaces, coverage periods, subscription keys, and filters. The code is as follows:

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><subscriptionResultsList
xmlns:ns3="http://www.w3.org/2000/09/xmlsig#"
xmlns:ns2="urn:uddi-org:api_v3" xmlns="urn:uddi-
org:sub_v3"><coveragePeriod>
<startPoint>2008-01-01T00:00:00</startPoint>
<endPoint>2010-01-01T00:00:00</endPoint>
</coveragePeriod><subscription brief="false">
<subscriptionKey>uddi:marketing.apache.org:subscript
ion:key1</subscriptionKey><subscriptionFilter>
<ns2:find_service><ns2:findQualifiers>
<ns2:findQualifier>exactMatch</ns2:findQualifier>
</ns2:findQualifiers><ns2:name
xml:lang="en">HelloWorld</ns2:name>
</ns2:find_service></subscriptionFilter>
<maxEntities>1000</maxEntities>
```

Figure 10.10. The Raw XML response of the synchronous Subscription request

The response will also be consumed by the marketing node. The marketing node will import the HelloWorld subscription information, as well as the sales business. So after a successful sync you should now see three businesses in the Browser Portlet of the marketing node, see [Figure 10.11](#), “The registry info of the HelloWorld Service information was imported by the subscription mechanism.”



Figure 10.11. The registry info of the HelloWorld Service information was imported by the subscription mechanism.

Administration

11.1. Introduction

General Stuff about administration.

11.2. Changing the Listener Port

If you want to change the port Tomcat listens on to something non-standard (something other than 8080):

jUDDI Server

1. edit `conf/server.xml` and change the port within the `<Connector>` element
2. edit `webapps/juddiv3/WEB-INF/classes/juddiv3.properties` and change the port number

jUDDI Portal

1. edit `webapps/uddi-portlets/WEB-INF/classes/META-INF/uddi.xml` and change the port numbers within the endpoint URLs
2. edit `pluto/WEB-INF/classes/server.xml` and change the port within the `<Connector>` element

11.3. Changing the Oracle Sequence name

If you are using Hibernate as a persistence layer for jUDDI, then Oracle will generate a default sequence for you ("HIBERNATE_SEQUENCE"). If you are using hibernate elsewhere, you may wish to change the sequence name so that you do not share this sequence with any other applications. If other applications try to manually create the default hibernate sequence, you may even run into situations where you find conflicts or a race condition.

The easiest way to handle this is to create an `orm.xml` file and place it within the classpath in a META-INF directory, which will override the jUDDI persistence annotations and will allow you to specify a specific sequence name for use with jUDDI. The following `orm.xml` specifies a "juddi_sequence" sequence to be used with jUDDI.

```
<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/
  persistence/orm_1_0.xsd"
  version="1.0">
```

```
<sequence-generator name="juddi_sequence" sequence-name="juddi_sequence"/>

<entity class="org.apache.juddi.model.Address">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.AddressLine">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.BindingDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.BusinessDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.BusinessIdentifier">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.BusinessName">
  <attributes>
```

```
<id name="id">
  <generated-value generator="juddi_sequence" strategy="AUTO"/>
</id>
</attributes>
</entity>

<entity class="org.apache.juddi.model.CategoryBag">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.Contact">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.ContactDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.DiscoveryUri">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.Email">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
```

```
</entity>

<entity class="org.apache.juddi.model.InstanceDetailsDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.InstanceDetailsDocDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.KeyedReference">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.KeyedReferenceGroup">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.OverviewDoc">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.OverviewDocDescr">
  <attributes>
```



```
<id name="id">
  <generated-value generator="juddi_sequence" strategy="AUTO"/>
</id>
</attributes>
</entity>

<entity class="org.apache.juddi.model.PersonName">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.Phone">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.ServiceDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.ServiceName">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.SubscriptionMatch">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
```

```
</entity>

<entity class="org.apache.juddi.model.TmodelDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.TmodelIdentifier">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.TmodelInstanceInfo">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.TmodelInstanceInfoDescr">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.TransferTokenKey">
  <attributes>
    <id name="id">
      <generated-value generator="juddi_sequence" strategy="AUTO"/>
    </id>
  </attributes>
</entity>

<entity class="org.apache.juddi.model.BindingTemplate">
  <attributes>
```

```
<basic name="accessPointUrl">
  <column name="access_point_url" length="4000"/>
</basic>
</attributes>
</entity>
</entity-mappings>
```

11.4. Persistence

jUDDI supports both OpenJPA and Hibernate as persistence providers. If you are embedding jUDDI, it is important to note that there are two JARs provided through maven. If you will be using Hibernate, please use the juddi-core JAR, if you are using OpenJPA, use juddi-core-openjpa.

The difference between these JARs is that the persistence classes within juddi-core-openjpa have been enhanced (http://people.apache.org/~mprudhom/openjpa/site/openjpa-project/manual/ref_guide_pc_enhance.html). Unfortunately, the Hibernate classloader does not deal well with these enhanced classes, so it is important to note not to use the juddi-core-openjpa JAR with Hibernate.

Deploying to JBoss 6.0.0.GA

12.1. Introduction

This section describes how to deploy juddi to JBoss 6.0.0.GA.

First, download jboss-6.0.0.GA - the zip or tar.gz bundle may be found at <http://www.jboss.org/jbossas/downloads/>. Download the bundle and uncompress it.

12.2. Add juddiv3.war

Copy juddiv3.war to server/default/deploy and unpack it.

Insert `jboss-web.xml` into the `juddiv3.war/WEB-INF` directory , should look like the following :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE jboss-web PUBLIC
"-//JBoss//DTD Web Application 2.3V2//EN"
"http://www.jboss.org/j2ee/dtd/jboss-web_3_2.dtd">

<jboss-web>

  <resource-ref>
    <res-ref-name>jdbc/JuddiDS</res-ref-name>
    <jndi-name>java:JuddiDS</jndi-name>
  </resource-ref>
  <depends>jboss.jdbc:datasource=JuddiDS,service=metadata</depends>

</jboss-web>
```

12.3. Change web.xml

Replace the `WEB-INF/web.xml` with the `jbossws-native-web.xml` within `docs/examples/appserver`.

12.4. Configure Datasource

The first step for configuring a datasource is to copy your JDBC driver into the classpath. Copy your JDBC driver into `${jboss.home.dir}/server/${configuration}/lib`, where `configuration` is the profile you wish to start with (default, all, etc.). Example :

```
cp mysql-connector-java-5.0.8-bin.jar /opt/jboss-5.1.0.GA/server/default/lib
```

Next, configure a JBoss datasource file for your db. Listed below is an example datasource for MySQL :

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>JuddiDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/juddiv3</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name>
    <password></password>
    <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter</exception-sorter-class-
name>

    <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml (optional) -->
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Next, make a few changes to the `juddiv3.war/classes/META-INF/persistence.xml`. Change the "hibernate.dialect" property to match the database you have chosen for persistence. For MySQL, change the value of `hibernate.dialect` to "org.hibernate.dialect.MySQLDialect". A full list of dialects available can be found in the hibernate documentation (https://www.hibernate.org/hib_docs/v3/api/org/hibernate/dialect/package-summary.html). Next, change the `<jta-data-source>` tags so that it reads `<non-jta-data-source>`, and change the value from `java:comp/env/jdbc/JuddiDS` to `java:/JuddiDS`.

Deploying to Glassfish 2.1.1

13.1. Introduction

This section describes how to deploy juddi to Glassfish 2.1.1. These instructions will use CXF as a webservice framework.

First, download the glassfish-v2.1.1 installer JAR. Once downloaded, install using the JAR and then run the ant setup script :

```
java -jar glassfish-installer-v2.1.1-b31g-linux.jar
cd glassfish
ant -f setup.xml
```

13.2. Glassfish jars

Copy the following JARs into domains/domain1/lib/ext. Note that for the purposes of this example, we have copied the MySQL driver to domains/domain1/lib/ext :

```
antlr-2.7.6.jar
cglib-nodep-2.1_3.jar
commons-collections-3.2.1.jar
commons-logging-1.1.jar
dom4j-1.6.1.jar
hibernate-3.2.5.ga.jar
hibernate-annotations-3.3.0.ga.jar
hibernate-commons-annotations-3.0.0.ga.jar
hibernate-entitymanager-3.3.1.ga.jar
hibernate-validator-3.0.0.ga.jar
javassist-3.3.ga.jar
jboss-common-core-2.0.4.GA.jar
jta-1.0.1B.jar
mysql-connector-java-5.0.8-bin.jar
persistence-api-1.0.jar
```

13.3. Configure the JUDDI datasource

First, using the asadmin administration tool, import the following file :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//Sun Microsystems Inc.//DTD Application Server 9.0 Domain//
EN" "*<install directory>/lib/dtds/sun-resources_1_3.dtd*">
<resources>
<jdbc-connection-pool          name="mysql-pool"          datasource-
classname="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"      res-
type="javax.sql.DataSource">
<property name="user" value="juddi"/>
<property name="password" value="juddi"/>
<property name="url" value="jdbc:mysql://localhost:3306/juddiv3"/>
</jdbc-connection-pool>
<jdbc-resource  enabled="true"  jndi-name="jdbc/mysql-resource"  object-type="user"  pool-
name="mysql-pool"/>
</resources>
```

```
asadmin add-resources resource.xml
```

Then use the Glassfish administration console to create a "jdbc/juddiDB" JDBC datasource resource based on the mysql-pool Connection Pool.

13.4. Add juddiv3-cxf.war

Unzip the juddiv3-cxf WAR into `domains/domain1/autodeploy/juddiv3.war` .

Add a `sun-web.xml` file into `juddiv3.war/WEB-INF`. Make sure that the JNDI references matches the JNDI location you configured in the Glassfish administration console.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
Application Server 9.0 Servlet 2.5//EN"
'http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd'>
<sun-web-app>
<resource-ref>
<res-ref-name>jdbc/juddiDB</res-ref-name>
<jndi-name>jdbc/juddiDB</jndi-name>
</resource-ref>
</sun-web-app>
```


Next, make a few changes to `juddiv3.war/WEB-INF/classes/META-INF/persistence.xml`. Change the "hibernate.dialect" property to match the database that you have chosen for persistence. For MySQL, change the value of `hibernate.dialect` to "org.hibernate.dialect.MySQLDialect". A full list of dialects available can be found in the hibernate documentation (https://www.hibernate.org/hib_docs/v3/api/org/hibernate/dialect/package-summary.html). Next, change the `<jta-data-source>` change the value from `java:comp/env/jdbc/JuddiDS` to `java:comp/env/jdbc/JuddiDB`.

13.5. Run juddi

Start up the server :

```
cd bin
asadmin start-domain domain1
```

Once the server is deployed, browse to <http://localhost:8080/juddiv3>

Appendix A. Revision History

Revision History

Revision 1.1 Thu Jan 07 2010

TomCunningham<tcunning@apache.org>

Translated Dev Guide to docbook

Revision 1.0 Mon Nov 16 2009

DarrinMison<dmison@redhat.com>

Created from community jUDDI Guide

