

jUDDI 3.0

User Guide

ASF-JUDDI-USRGUIDE-02/10/09

Contents

Table of Contents

Contents.....	2	Introduction.....	5
		Using the JAR.....	6
		Using the WAR files.....	7
		Using the Tomcat Bundle.....	8
		Using jUDDI as Web Service.....	9
		Using jUDDI with your application.....	11
About This Guide.....	3	Authentication.....	12
What This Guide Contains.....	3	Introduction.....	12
Audience.....	3	JUDDI Authentication.....	13
Prerequisites.....	3	XMLDocAuthentication.....	13
Organization.....	3	CryptedXMLDocAuthentication.....	13
Documentation Conventions.....	3	JBoss Authentication.....	14
Additional Documentation.....	4		
Contacting Us.....	4	Index.....	15
Setup.....	5		

About This Guide

What This Guide Contains

The User Guide document describes use of jUDDI – installation and setup.

Audience

This guide is most relevant to engineers who are responsible for setting up jUDDI 3.0 installations.

Prerequisites

None.

Organization

This guide contains the following chapters:

- **Chapter 1, Setup**
- **Chapter 2, Authentication**

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
<i>Italic</i>	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
Code	Text that represents programming code.
Function Function	A path to a function or dialog box within an interface. For example, “Select File Open.” indicates that you should select the Open function from the File menu.
() and	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: <code>persistPolicy (Never OnTimer OnUpdate NoMoreOftenThan)</code>
Note:	A note highlights important supplemental information.
Caution:	A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Table 1 Formatting Conventions

Additional Documentation

None on the subject.

Contacting Us

Email: juddi-user@ws.apache.org

License

Copyright 2001-2009 The Apache Software Foundation.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and
limitations under the License.

UDDI Registry

Introduction to UDDI

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/> [UDDI].

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.

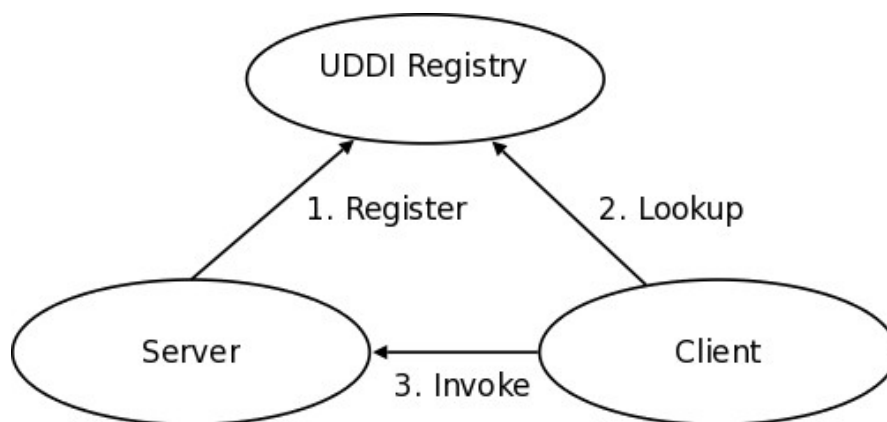


Figure 1.1 Invocation pattern using the UDDI Registry.

A business publishes services to the UDDI registry. A client looks up the service in the registry and receives servicebinding 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

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.

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 backend or persistence layer, or configure your own Webservice provider with the provided WSDL files and classes.

Using the WAR files

As with the JAR, you need to make a decision on what framework you would like to use when building the WAR. There will eventually be two WAR files shipped – one using CXF and one using Axis 2. For the alpha release, only CXF has been tested thoroughly.

Simple copy the WAR to the deploy folder of your server (this release has been tested under Apache Tomcat 5.5.23), start your server, and follow the directions under “using jUDDI as a Web Service”.

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, the Hibernate is used as the persistence layer and CXF is used as a Web Service framework.

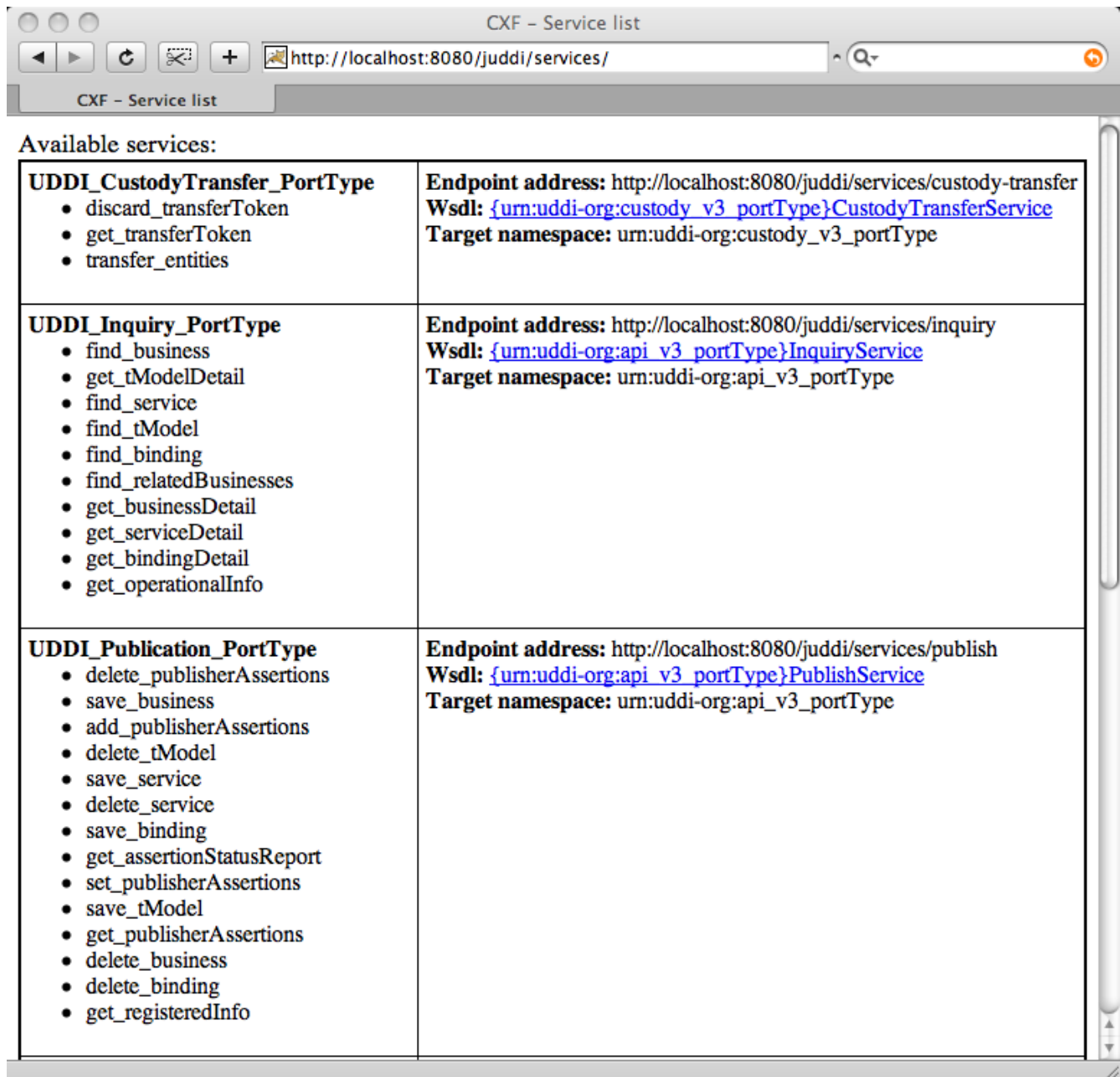
To get started using the Tomcat bundle, unzip the juddi-tomcat-bundle.zip, and start Tomcat :

```
% 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.

Using jUDDI as Web Service

Browse to <http://localhost:8080/juddiv3/services>



Available services:	
UDDI_CustodyTransfer_PortType <ul style="list-style-type: none">• discard_transferToken• get_transferToken• transfer_entities	Endpoint address: http://localhost:8080/juddi/services/custody-transfer WSDL: {urn:uddi-org:custody_v3_portType}CustodyTransferService Target namespace: urn:uddi-org:custody_v3_portType
UDDI_Inquiry_PortType <ul style="list-style-type: none">• find_business• get_tModelDetail• find_service• find_tModel• find_binding• find_relatedBusinesses• get_businessDetail• get_serviceDetail• get_bindingDetail• get_operationalInfo	Endpoint address: http://localhost:8080/juddi/services/inquiry WSDL: {urn:uddi-org:api_v3_portType}InquiryService Target namespace: urn:uddi-org:api_v3_portType
UDDI_Publication_PortType <ul style="list-style-type: none">• delete_publisherAssertions• save_business• add_publisherAssertions• delete_tModel• save_service• delete_service• save_binding• get_assertionStatusReport• set_publisherAssertions• save_tModel• get_publisherAssertions• delete_business• delete_binding• get_registeredInfo	Endpoint address: http://localhost:8080/juddi/services/publish WSDL: {urn:uddi-org:api_v3_portType}PublishService Target namespace: urn:uddi-org:api_v3_portType

The services page shows you the available endpoints and methods available. You should be able to send some sample requests to jUDDI to test:

Using any SOAP client, you

Untitled

WSDL:

Service: **InquiryService**

Method:

EndpointURI:

Binding Style:

SOAPAction:

Namespace:

Parameters:

authInfo:

tModelKey:

User-Agent: Mac OS X; WebServicesCore.framework (1.0.0)
Content-Type: text/xml
Soapaction: get_tModelDetail
Host: localhost

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

<SOAP-ENV:Envelope

  xmlns:xsd="http://www.w3.org/2001/XMLSchema"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"

  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

  <SOAP-ENV:Body>

    <get_tModelDetail xmlns="urn:uddi-org:api_v3_portType">
```

Using jUDDI with your application

As of the Alpha release, two of the UDDI v3 APIs should be active within jUDDI : inquiry and publish.

Authentication

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.v3.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.

JUDDI Authentication

The default authentication mechanism provided by jUDDI is the `JUDDIAAuthenticator`. The *authenticate* phase of the `JUDDIAAuthenticator` 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.

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.v3.auth.JUDDIAAuthentication
```

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.v3.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`.

CryptedXMLDocAuthentication

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

```
juddi.auth = org.apache.juddi.v3.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`.

JBoss Authentication

Finally it is 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 its 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.v3.auth.JBossAuthenticator
juddi.securityDomain=java:/jaas/other
```

The `juddi.auth` property plugs the `JbossAuthenticator` class into the juddi the Authenticator framework. The `juddi.securityDomain`, 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 `$JBOSS_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

Derby Out-of-the-box

By default jUDDI uses an embedded Derby database. This allows us to build a download-able distribution that works out-of-the-box, without having to do any database setup work. We recommend switching to a 'real' 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>

```

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/CATALINE/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 then 'root'.

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 standardjbosscmp-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!

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

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 predefined `tModels`.

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 then 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 then 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.

Tokens in the Seed Data

You may have noticed the tokens in the root_BusinessEntity.xml file (`${juddi.server.name}` and `${juddi.server.port}`). The value of these tokens can set in the `juddiv3.properties` file, and the user can create his/her own tokens.

Custom 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 then

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

The `juddiv3.war` ships with example data for an “Affiliate A” registry, with a root publisher call “`affa`”. To use this Seed Data, in the `juddiv3.war/WEB-INF/classes/`, rename the directory `RENAME2ACTIVATE_juddi_custom_install_data` to `juddi_custom_install_data` and in the `juddiv3.properties` file set

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

before you start jUDDI.

Using the UDDI-Client

Introduction

The jUDDI project includes a UDDI-Client (uddi-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.

Configuration

The UDDI client has a configuration file called uddi.properties. 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.properties file looks like

```
#### JAXWS Transport #####
uddi.proxy.transport           =org.apache.juddi.v3.client.transport.JAXWSTransport
uddi.custody.transfer.url      =http://localhost:8080/juddi/services/custody-transfer?wsdl
uddi.inquiry.url               =http://localhost:8080/juddi/services/inquiry?wsdl
uddi.publish.url               =http://localhost:8080/juddi/services/publish?wsdl
uddi.security.url              =http://localhost:8080/juddi/services/security?wsdl
uddi.subscription.url          =http://localhost:8080/juddi/services/subscription?wsdl
uddi.subscription.listener.url =http://localhost:8080/juddi/services/subscription-
listener?wsdl

#### RMI Transport #####
#uddi.proxy.transport          =org.apache.juddi.v3.client.transport.RMITransport
#java.naming.factory.initial   =org.jnp.interfaces.NamingContextFactory
#java.naming.factory.url.pkgs  =org.jboss.naming
#java.naming.provider.url       =jnp://localhost:1099
#uddi.custody.transfer.url      =/juddi/UDDICustodyTransferService
#uddi.inquiry.url               =/juddi/UDDIInquiryService
#uddi.publish.url               =/juddi/UDDIPublicationService
#uddi.security.url              =/juddi/UDDISecurityService
#uddi.subscription.url          =/juddi/UDDISubscriptionService
#uddi.subscription.listener.url =/juddi/UDDISubscriptionListenerService

#### InVM Transport #####
#uddi.proxy.transport          =org.apache.juddi.v3.client.transport.InVMTransport
#uddi.custody.transfer.url      =org.apache.juddi.api.impl.UDDICustodyTransferImpl
#uddi.inquiry.url               =org.apache.juddi.api.impl.UDDIInquiryImpl
#uddi.publish.url               =org.apache.juddi.api.impl.UDDIPublicationImpl
#uddi.security.url              =org.apache.juddi.api.impl.UDDISecurityImpl
#uddi.subscription.url          =org.apache.juddi.api.impl.UDDISubscriptionImpl
#uddi.subscription.listener.url =org.apache.juddi.api.impl.UDDISubscriptionListenerImpl
```

JAX-WS Transport

Using the settings in the uddi.properties 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.

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 juddi.war/WEB-INF/classes/juddi.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.

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.

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.

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 the chapter “Authentication” to learn more about this subject.

Another place to look for sample code on how to use the uddi-client check the juddi-console module.

Subscription

Introduction

Subscriptions can be used to receive notifications of changes to certain registry entities such as business, services and binding templates. 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.

Subscription Listener

Part of the UDDI v3 subscription model is that it allows the user to be informed of changes to subscribed objects. jUDDI provides a subscription-listener webservice which should serve as an example for any Subscription Listener service that a developer will implement – it logs and displays any notifications. A developer will probably want to implement the SubscriptionListener on their own and provide logic for what happens when a change to a subscribed object.

In addition, the juddi-console provides a utility that allows you to monitor subscription notifications. If you register <http://localhost:8080/subscription-listener/services/listener?wsdl> as a Binding and then use the bindingKey and your accessPoint, then notifications will be sent to the console.



Administration

Introduction

General Stuff about administration.

Changing the listener port

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

- edit `conf/server.xml` and change the port within the `<Connector>` element
- edit `webapps/uddi-portlets/WEB-INF/classes/uddi.properties` and change the port numbers within the endpoint URLs
- edit `webapps/juddi/WEB-INF/classes/juddi.properties` and change the port number
- edit `pluto/WEB-INF/classes/server.xml` and change the port within the `<Connector>` element

References

[UDDI] uddi.xml.org – Online community for the Universal Description, Discovery, and Intergration OASIS Standard. <http://uddi.xml.org>

[COM-CONFIG] Apache Commons Configuration Project <http://commons.apache.org/configuration/>

Index
