## jUDDI Dev Guide

# A developer's guide to using jUDDI

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

and thanks to Darrin Mison

# 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` or ***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. jUDDI Architecture for UDDI v3 Project

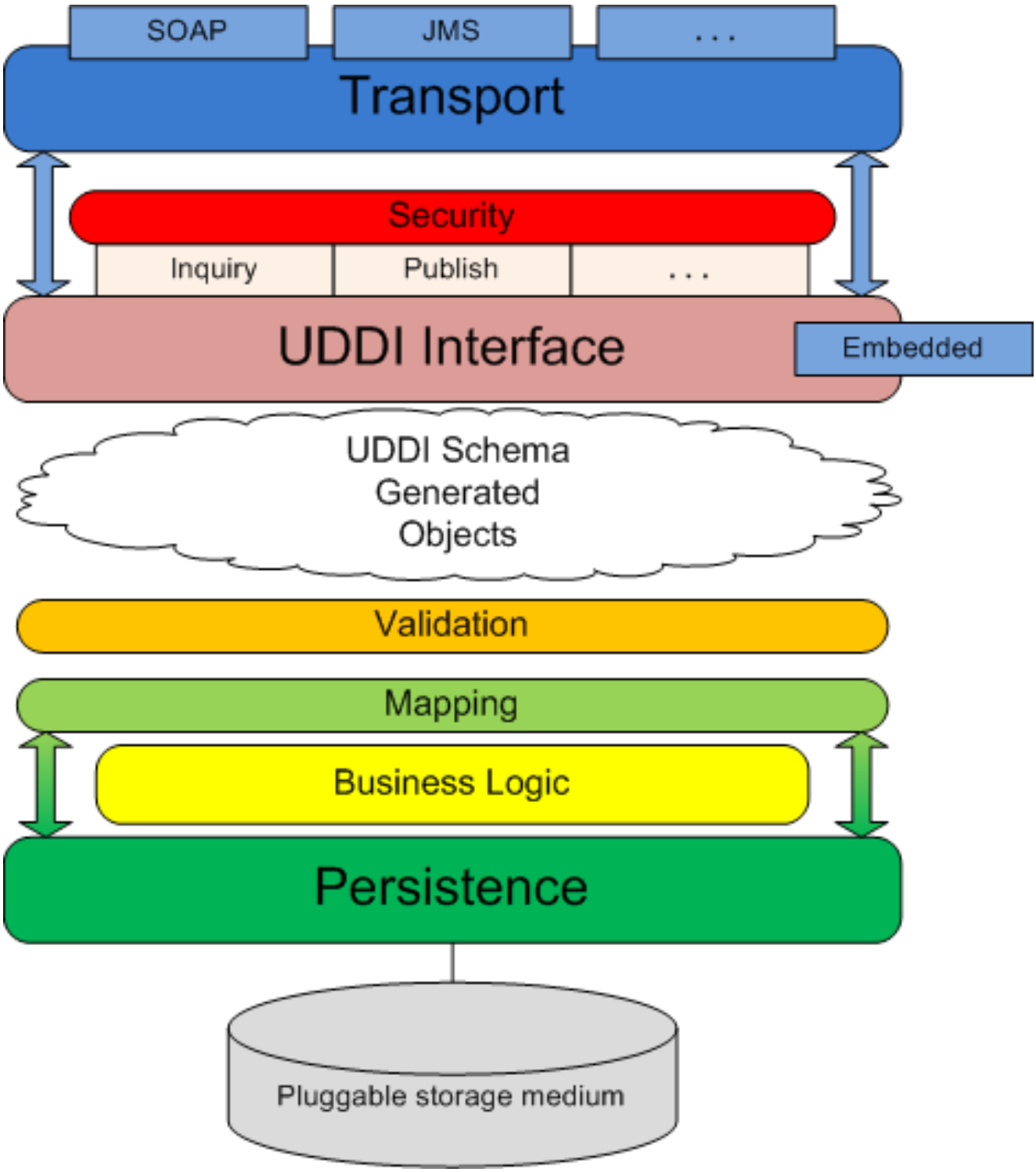


**Figure 1.1. jUDDI Architecture**

The diagram above shows the software layers and components employed in the jUDDI project implementation for UDDI v3. Here is a brief description of each item in the diagram and how they all work together to create the UDDI-compliant registry that is jUDDI:

1. Transport - the transport layer provides the means to receive and send out requests via a network or messaging service. The UDDI specification details an interface where XML messages are exchanged between client and server but is agnostic as to how those messages are relayed. By default, jUDDI uses Apache CXF to transport messages via SOAP over HTTP, however, the system is designed so other methods of transport can be easily plugged in (for example, JMS).

2. Security - security is provided by the UDDI specification and is based on policies defined in the specification. jUDDI implements all the mandatory policies and can be extended to support the optional policies. Chief among these policies is access control to the UDDI API exposed by jUDDI. jUDDI fully implements this policy, per the specification, which allows users to easily plug in their own third-party authentication framework.

3. UDDI Interface - the UDDI interface defines the methods set forth by the UDDI specification to interact with the registry. Within jUDDI, the interface classes are generated from the UDDI WSDL and they are implemented as POJOs. These classes are annotated with JAX-WS annotations allowing end-users to easily employ any suitable JAX-WS container to expose the interface.

4. In general, the interface implementation accepts incoming UDDI-based requests and ummarshals these requests to the appropriate schema object. This object is then served to the proceeding layers so the necessary logic can be performed to fulfill the request. After the request is fulfilled, this layer is responsible for marshalling the result and sending the response to the requesting party.

5. As the interface is implemented as POJOs, it can be accessed via an "embedded" mode. In this scenario, the methods of the implementation classes can be called directly. This allows users to embed jUDDI directly into their application without having to deploy a full-blown jUDDI server.

6. UDDI Schema Objects - The UDDI specification comes equipped with an XML schema for its many data structures. jUDDI employs XML-binding technology (JAXB) to generate objects from the schema (contained within the WSDL) that are then used as the arguments for the UDDI Interface layer. These objects needn't originate from XML – they can also be instantiated directly to make UDDI calls directly in java code.

7. Validation – the validation layer reads the schema object input from the UDDI interface layer and, based on rules defined in the specification, makes sure the input is valid for the given UDDI method. Failed validation results in an exception and an immediate return from the method call.

8. Mapping – the mapping layer is responsible for mapping the UDDI schema objects to the persistence layer model. For all intents and purposes, the mapping layer simply copies data from a schema object to the similar model object. This occurs in both directions, as input objects

must be mapped to the model to perform the necessary logic and results obtained from the call must be mapped back to the schema as output to the caller.

9. Business Logic - the business logic layer is responsible for performing all the business logic associated with the UDDI calls. The logic layer works with objects from the persistence layer and generally consists of querying the model based on user input.

10. Persistence - the persistence layer, as its name implies, is responsible for persisting registry data to a storage medium. To this end, a third-party persistence service that implements the Java Persistence API (Apache OpenJPA, Hibernate) is utilized to manage transactions with the storage medium and also to facilitate the plugging-in of various storage types. By default, jUDDI is packaged with Apache OpenJPA as the persistence provider and Apache Derby as the storage medium. This can easily be configured.

# Development Environment Setup

## 2.1. Prerequisites

To be able to build and run jUDDI you will need to have the following installed:

1. 1.5.X JDK
2. Maven 2.0.8

## 2.2. Building the Project

First, check out the jUDDI sources:

```
% svn co http://svn.apache.org/repos/asf/webservices/juddi/trunk
```

Then build the entire project using OpenJPA for persistance use:

```
% cd trunk
% mvn clean install -Dpersistence=openjpa
```

To use Hibernate change the persistence flag to hibernate. Optionally you can use a `settings.xml` to set your persistence choice on a permanent basis, so you don't have to provide the persistence variable every time you build. The default location of the `settings.xml` is in your `.m2` directory. An example file is checked into our source tree at `etc/.m2/settings.xml`.

## 2.3. Source Modules Overview

Within jUDDI source, there are the following modules:

1. uddi-ws: JAXWS stubs built from the WSDLs
2. uddi-tck: Test kit developed by jUDDI for testing UDDI v3 functionality. The TCK is not jUDDI specific and could be used to verify and validate other UDDI v3 implementations
3. juddi-core: the jUDDI jar containing the model, API, and core jUDDI functionality
4. juddiv3-war: a WAR module agnostic as to JAX-WS provider
5. juddi-cxf: a WAR module that uses CXF as the web service framework, chosen by default
6. juddi-tomcat: a module which builds a Tomcat bundle with juddi-cxf installed and Derby as a backend data base
7. juddi-console: a module which builds upon the juddi-tomcat module and adds a GWT-based administration console

8. uddi-client: a generic client library for communicating with a UDDI server

9. juddi-dist: a module used to produce shippable binary distributions

jUDDI v3 is set up to produce a number of different deliverables – a JAR, a WAR, and a Tomcat bundle. Depending on the scope of your application, or your interest in the project, you might want to use the Tomcat server bundle packaged with the Derby database and jUDDI, or you may just want to use the jUDDI JAR and make your own database and Web Service choices. jUDDI is set up so that it can support a range of environments.

## 2.4. Setting up Eclipse

The easiest way to setup jUDDI in eclipse is to use the m2eclipse plugin which can be found at http://m2eclipse.codehaus.org/update/. In order to run and debug the project unit tests, it is required that you install this plugin. After installing the plugin you should select:

1. “Enable Depency Management”

2. Then, “Enable Nested Modules”

3. Then, “Update Project Configuration”

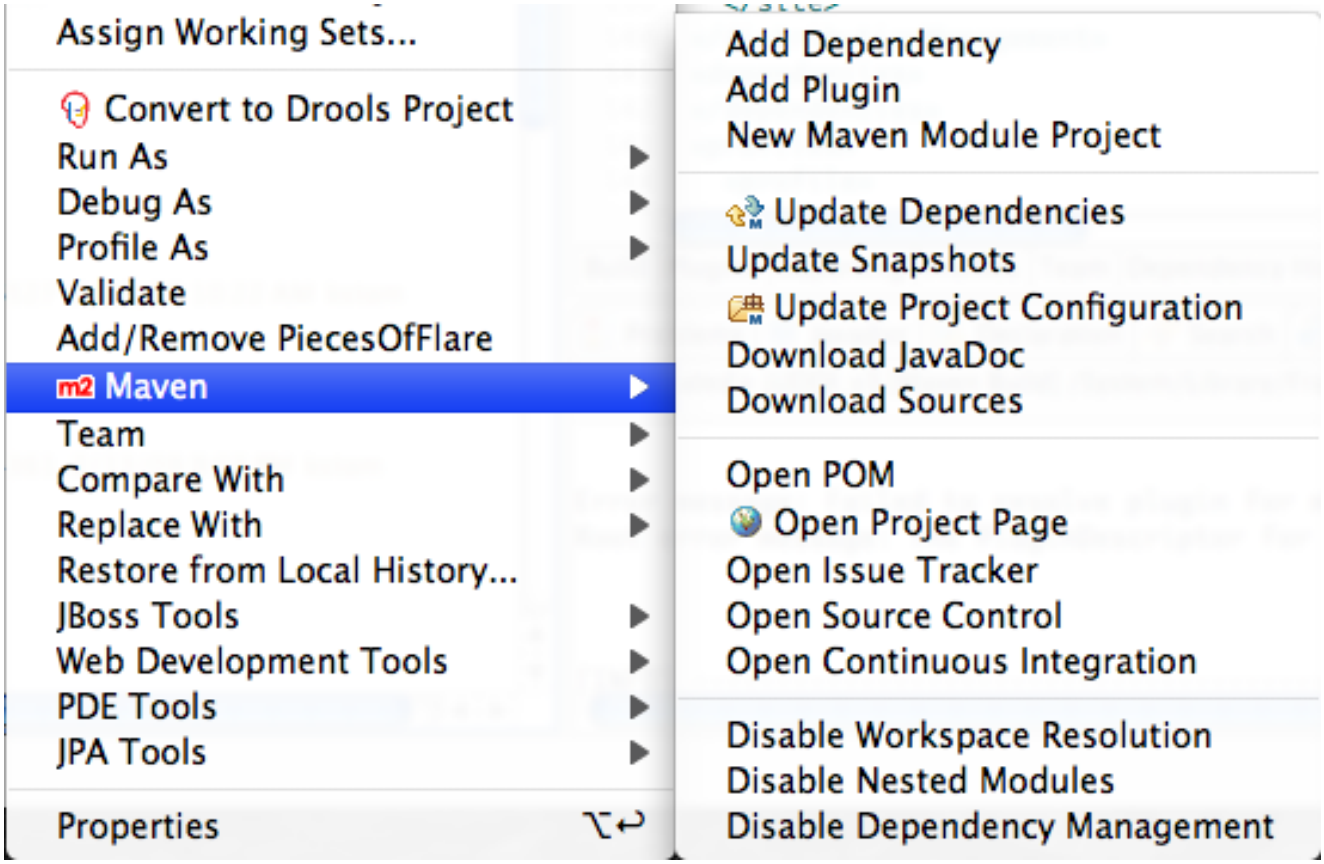


**Figure 2.1. Eclipse Maven Integration**

If you wish to change your persistence.xml for the purposes of testing, either change it and then build, or change `juddi-core/target/classes/META-INF/persistence.xml`. If you choose not to use the m2eclipse plugin you can setup your classpath by following these directions, but there are no guarantees that the unit tests will be debuggable within Eclipse. Choose “Eclipse” -> “Preferences” In the preference dialog, select “Java” -> “Build Path” -> “Classpath Variables” Add a new classpath variable :

```
Name: M2_REPO
```

```
Path : /[path-to-.m2]/.m2 (example : /home/tcunning/.m2)
```

```
% cd v3_trunk
```

```
% mvn eclipse:eclipse -Declipse.workspace=/[path-to-workspace]/workspace
```

Then within Eclipse, “Create New Project” and choose “Create from existing source” and choose the source folder that you just checked out from SVN.

## 2.5. Running a unittest from within Eclipse

To run one unittest from within eclipse simply right-click the unittest and select Debug As > Junit Test

### Figure 2.2. Eclipse Maven Integration

If you are using OpenJPA you have to make sure that the `openjpa-1.2.jar` is on the classpath and that for each unittest you specify the javaagent needed for the enhancement phase

```
-javaagent:/Users/kstam/.m2/repository/org/apache/openjpa/openjpa/1.2.0/
openjpa-1.2.0.jar
```

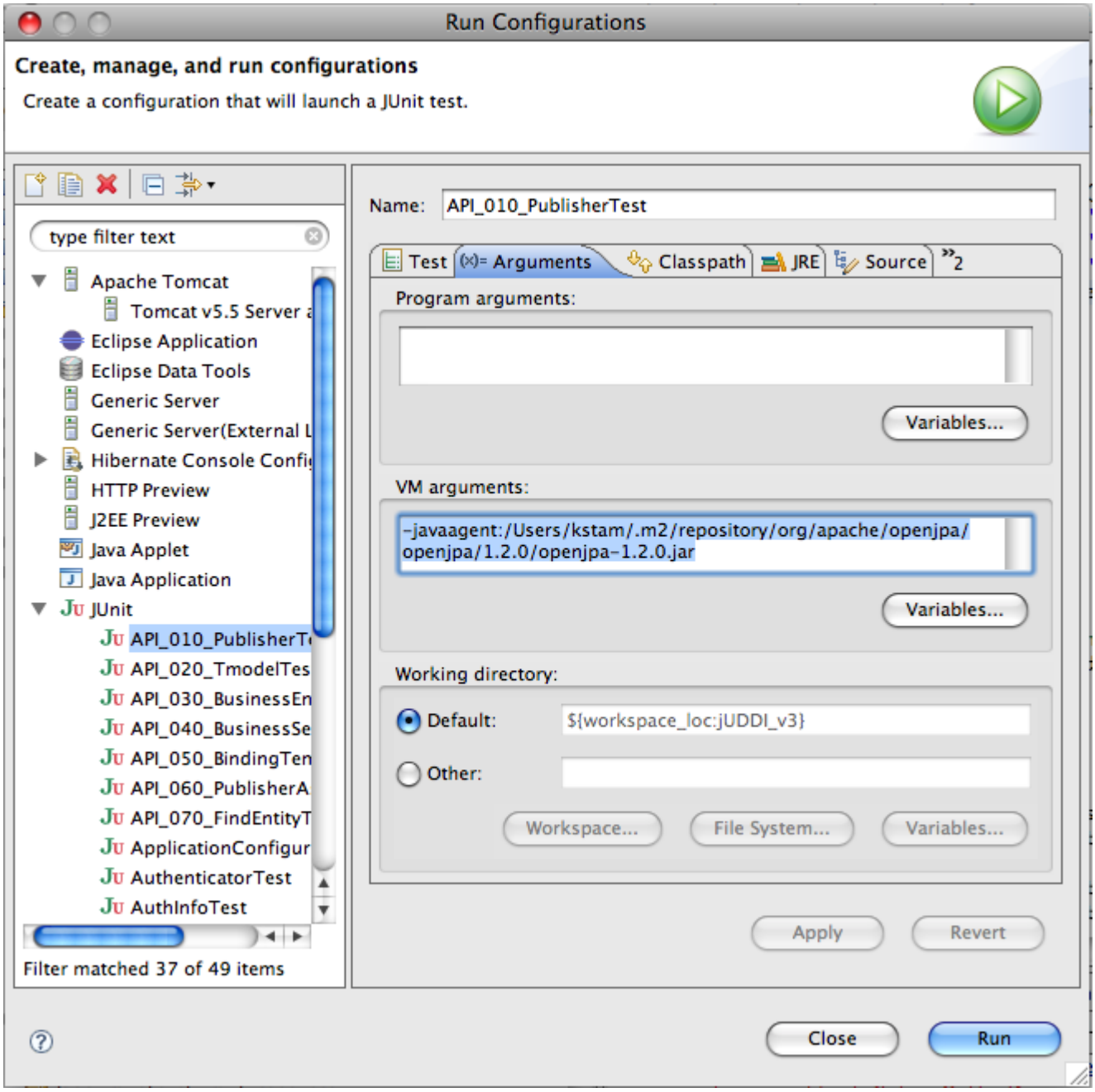


**Figure 2.3. Eclipse Maven Integration**

## 2.6. Building the JAR

The juddi-core module produces a JAR which contains the jUDDI source and a jUDDI persistence.xml configuration. jUDDI is currently setup so that you can choose between using either OpenJPA or Hibernate as your persistence framework. The juddi-core `pom.xml` contains two profiles, triggered on the "persistence" property.

OpenJPA

```
% cd juddi-core
% mvn clean install -Dpersistence=openjpa
```

Hibernate

```
% cd juddi-core
% mvn clean install -Dpersistence=hibernate
```

For juddi 3.0.0 and 3.0.1, the project built with Hibernate by default, but as of 3.0.2 the project now builds with openjpa as the default persistence layer. Two flavors of juddi-core are available as maven artifacts - juddi-core for hibernate usage and juddi-core-openjpa for use with OpenJPA.

## 2.7. Building the WAR

As with the JAR, you need to make a decision on what framework you would like to use when building the WAR. The project contains twoWAR modules – juddiv3-war, which produces a JAX-WS agnostic WAR, and juddi-cxf – which produces a WAR with CXF descriptors. The project would welcome any contribution of docs or descriptors for alternative JAX-WS providers.

## 2.8. Building the Tomcat Bundle

Tomcat bundle packages up one of the jUDDI WAR files, Apache Derby, and a few necessary configuration files and provides the user with a pre-configured jUDDI instance. By default, the WAR produced by the juddi-cxf module is used – the example below shown uses URLs and endpoints using the jUDDI CXF configuration. If you use the Axis 2 configuration, URLs and endpoints may differ.

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
```

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

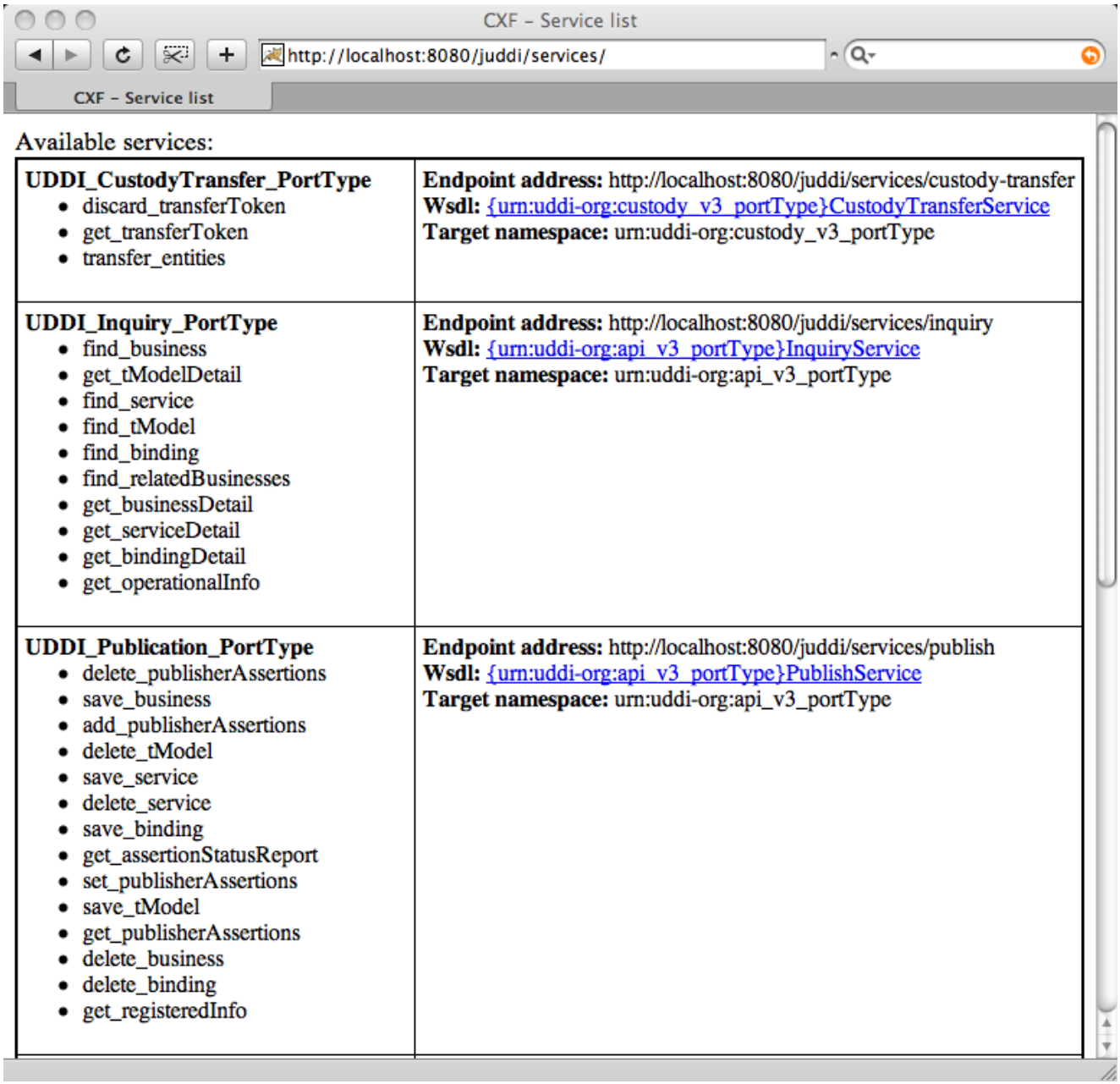


**Figure 2.4. jUDDI Service List**

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:

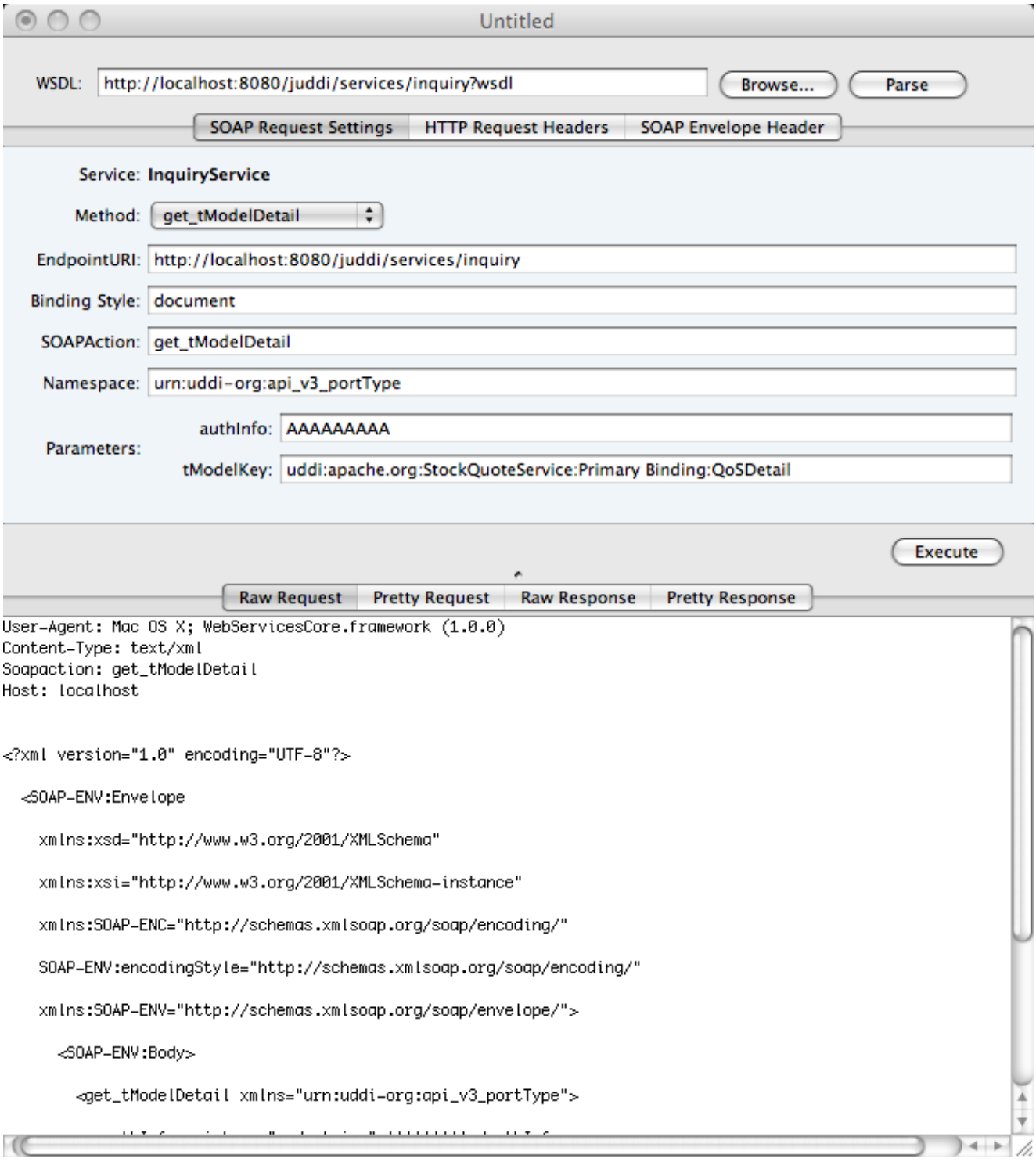


**Figure 2.5. SOAP Sample Request**

## 2.9. Running and Developing Tests

Currently the only unit tests are in juddi-core. We plan to add a suite of web service tests automated against the juddi-cargo module.

Running the tests:

```
% cd juddi-core
% mvn -Dpersistence=hibernate test
```

The tests are run through a maven-surefire-plugin within the juddi-core `pom.xml` :

```
      maven-surefire-plugin
      2.4.2

          src/test/resources/suite-init.xml,src/test/resources/suite-subscribe.xml,src/test/resources/
suite-clean.xml
```

The NUnit suite files listed here determine what tests are run with what data, and what order they are run in. `suite-init.xml` initializes the jUDDI database with data, `suite-subscribe.xml` runs a subscription test, and `suite-clean.xml` cleans the database and removes the test data.

To develop your own tests, please add another maven-surefire-plugin segment and the same ordering of XML files (`suite-init.xml`, your custom suite, and then `suite-clean.xml`).

# Release Process

## 3.1. Add your gpg key to KEYS

In the root of the project there is a KEYS file. Add your key to this file and check it into source control. Depending on your tool of choice you can use on of the following commands:

```
pgp -kxa <your name> and append it to this file.
(pgpk -ll <your name> && pgpk -xa <your name>) >> this file.
(gpg --list-sigs <your name>
    && gpg --armor --export <your name>) >> this file.
```

## 3.2. Release steps

`Environment: Apache Maven 2.2.1 (r801777; 2009-08-06 15:16:01-0400) Java version: 1.6.0_17`

1. Run mvn release:prepare, this will set all the right version numbers and create a tag in SVN.
2. Run mvn release:perform, will upload the release to a staging area in the Apache Nexus Repository.
3. Go into the juddi-dist, and run mvn clean install -Prelease
4. Upload the signed distribution artifacts to apache people.
5. Start a vote referencing the build artifacts, leave the vote open for 72 hrs.

On successful vote:

1 Release the staging artifacts in Nexus.

2 Copy the distribution artifacts to the mirror.

3 Update the website.

# 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 | | |