

jUDDI 3.0 - Developer Guide

Developer Guide

ASF-JUDDI-DEVGUIDE-15/09/09

Contents

Table of Contents

Contents.....	2	Dev Environment Setup.....	5
About This Guide.....	3	Introduction.....	5
What This Guide Contains.....	3	Building the project.....	5
Audience.....	3	Setting up Eclipse.....	7
Prerequisites.....	3	Building the JAR.....	8
Organization.....	3	Building the WAR.....	9
Documentation Conventions.....	3	Building the Tomcat Bundle.....	10
Additional Documentation.....	4	Running and Developing tests.....	12
Contacting Us.....	4	Index.....	13



About This Guide

What This Guide Contains

The Developer Guide document describes

Audience

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

Prerequisites

None.

Organization

This guide contains the following chapters:

- **Chapter 1, Dev Environment Setup**
- **Chapter 2, Testing**
- **Chapter 3,**

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	<p>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:</p> <pre>persistPolicy (Never OnTimer OnUpdate NoMoreOftenThan)</pre>
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-dev@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>

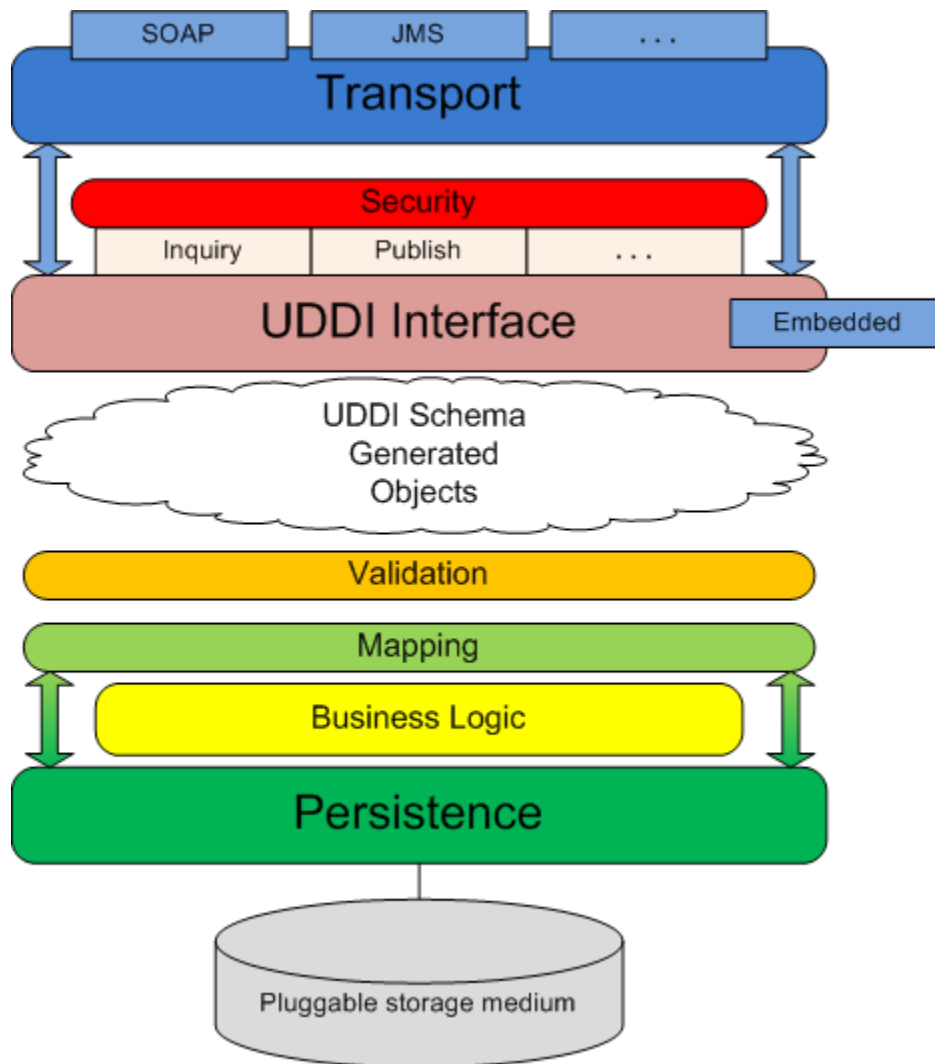
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,

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.

jUDDI Architecture

jUDDI Architecture for UDDI v3 Project



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:

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).

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.

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.

In general, the interface implementation accepts incoming UDDI-based requests and unmarshals 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.

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.

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.

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.

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.

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.

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.

Dev Environment Setup

Prerequisites

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

- 1.5.0_x JDK and
- Maven 2.0.8.

Building the project

First, check out the jUDDI sources:

```
% svn co http://svn.apache.org/repos/asf/webservices/juddi/branches/v3\_trunk
```

Then build the entire project using OpenJPA for persistence use:

```
% cd v3_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*.

Source modules overview

Within jUDDI source, there are the following modules:

- uddi-ws* - JAXWS stubs built from the WSDLs.
- uddi-tck* – Test kit developed by jUDDI for testing UDDIv3 functionality. This TCK is none jUDDI specific and could be used with other UDDI implementations.
- juddi-core* - the jUDDI jar containing the model, api and core jUDDI functionality
- juddi-cxf* - a WAR module that uses CXF as the web service framework, chosen by default
- juddi-axis* - a WAR module that uses Axis 2 as the web service framework, this is an alternate to using CXF
- juddi-tomcat* - a module which builds a Tomcat bundle with juddi-cxf installed and derby as a backend database
- juddi-cargo* - a module to help test jUDDI web services (currently no tests included)
- uddi-client* – a generic client library for communicating to a UDDI server.

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.

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:

- “Enable Dependency Management”
- Then, “Enable Nested Modules”
- Then, “Update Project Configuration”

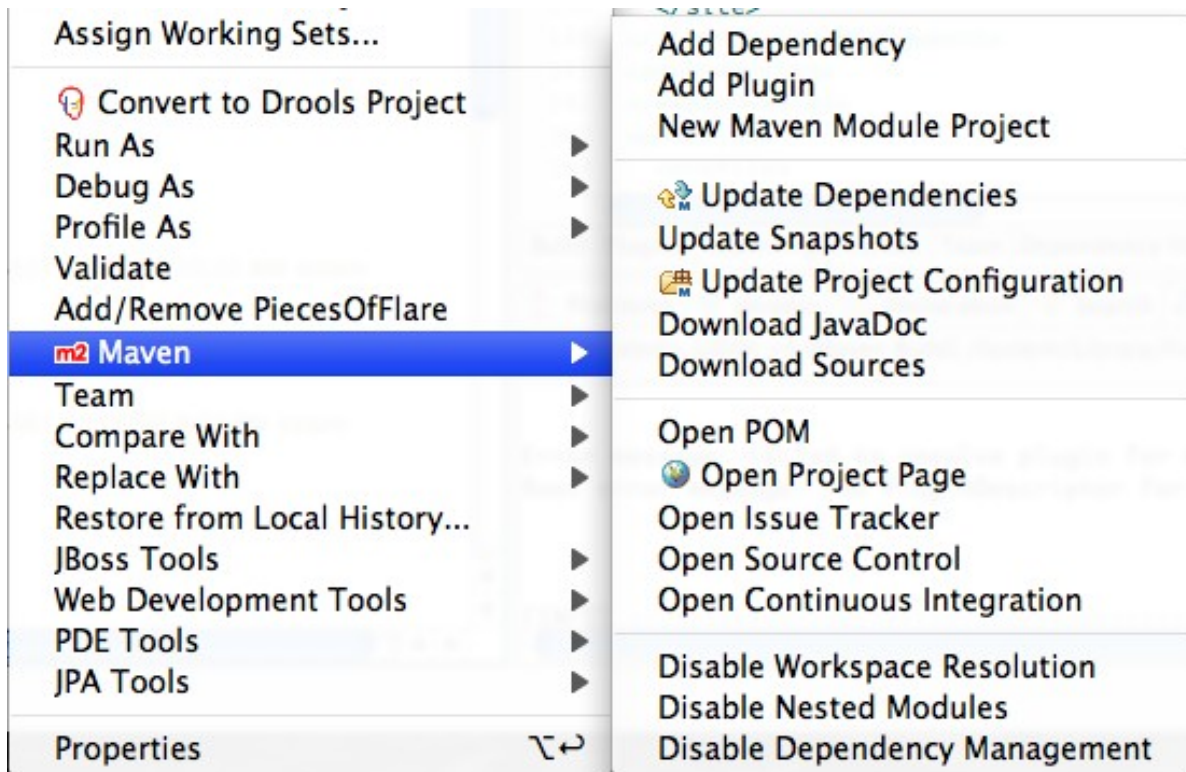


Figure 2.1 Enable m2eclipse

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.

Running one unittest from within Eclipse

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

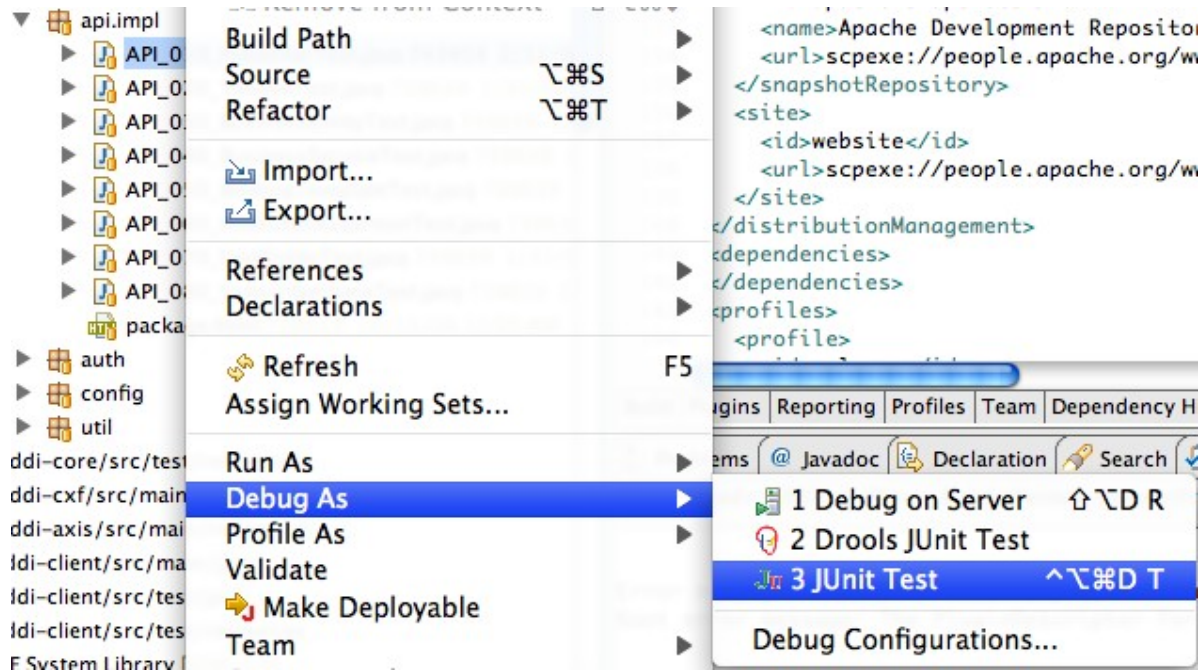


Figure 2.2 Running a unittests from within eclipse.

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
```

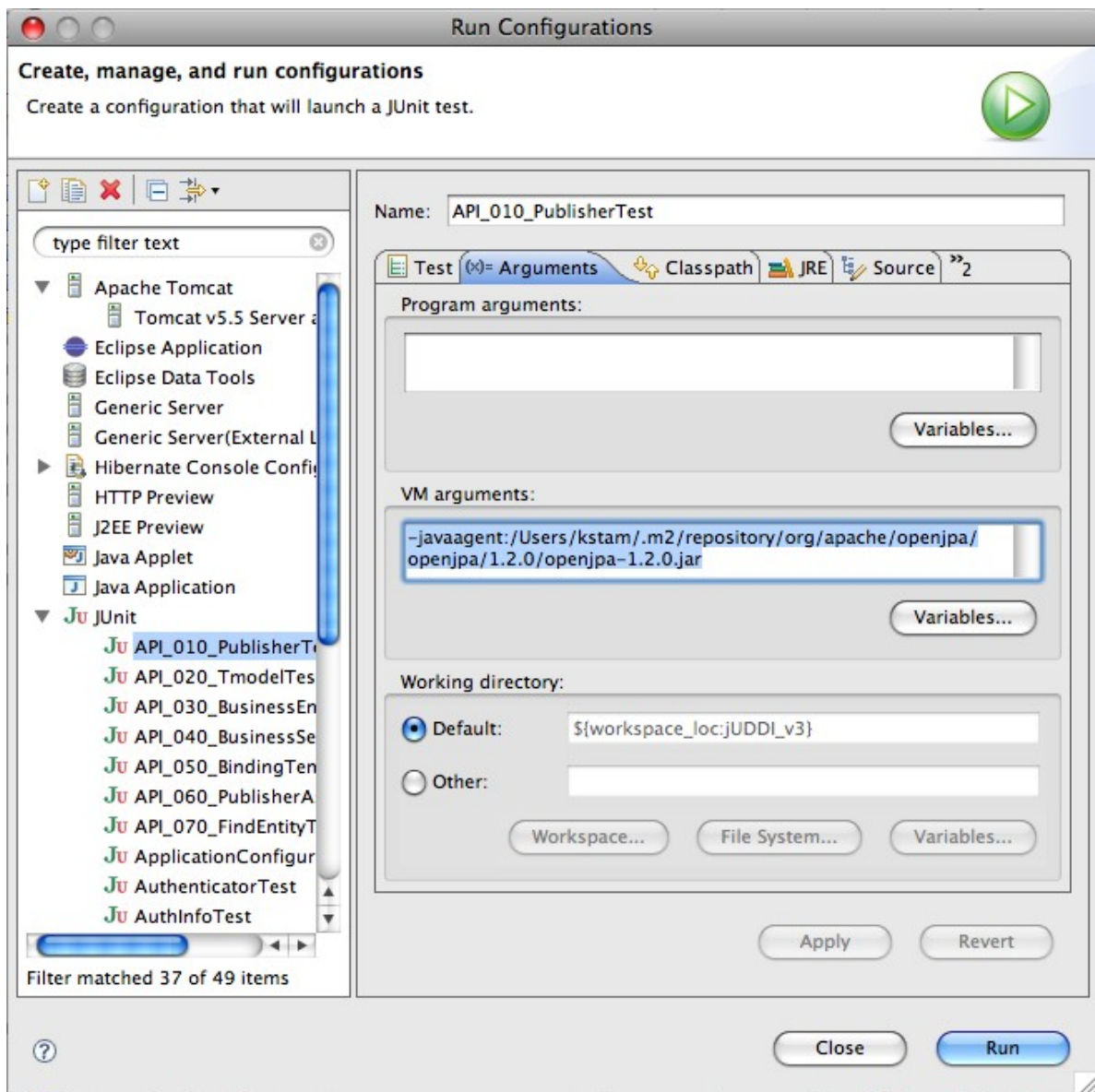


Figure 2.3. Specifying the javaagent needed by OpenJPA during the enhancement phase.

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
```

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 `juddi-cxf` module builds a war that uses CXF as a web service framework, and the `juddi-axis` target builds a war that uses Axis 2 as a web service framework.

There are currently some issues with the Axis 2 web service framework in `juddi-axis`.

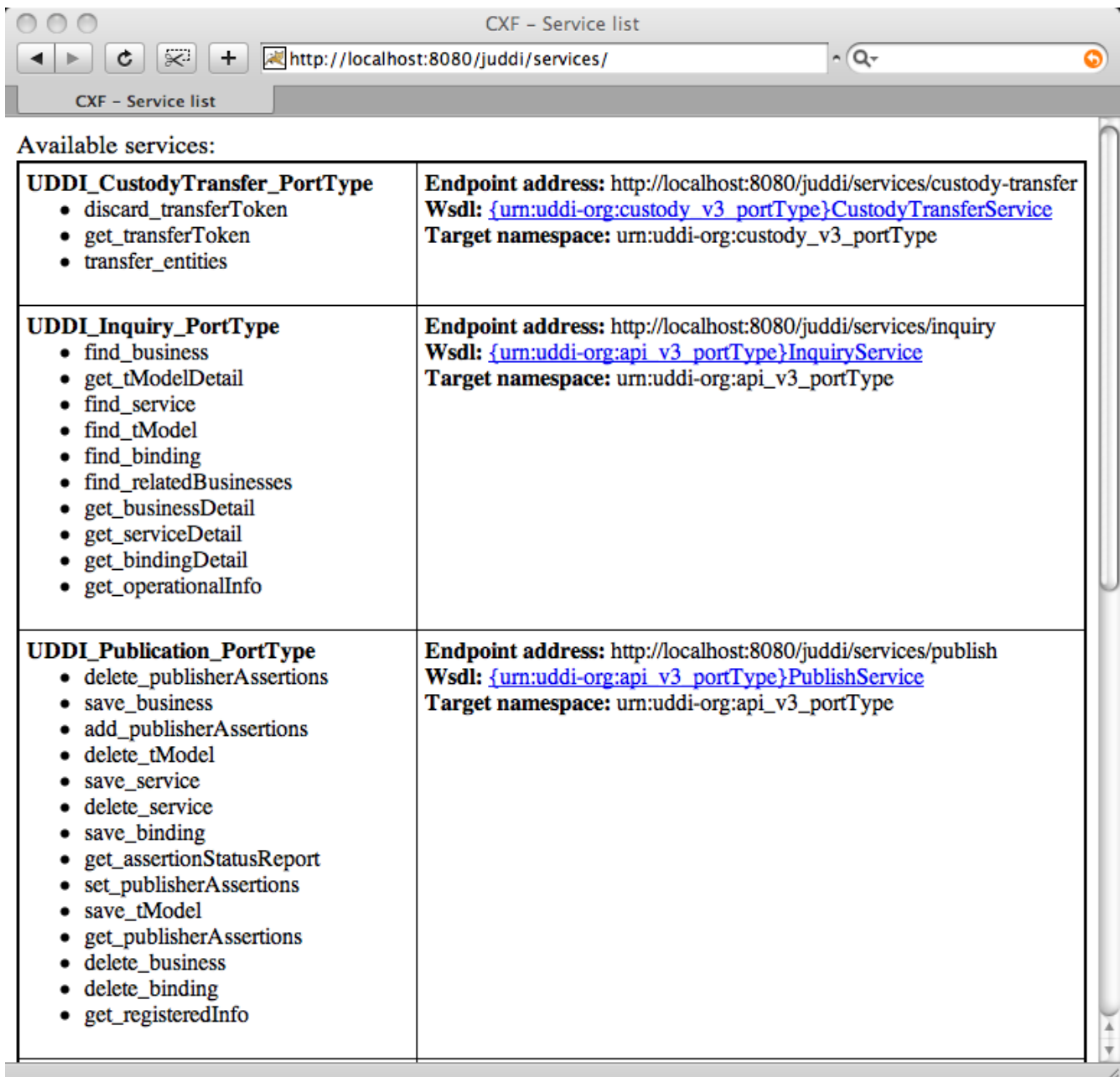
Building the Tomcat Bundle

The jUDDI 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/juddi3/services>

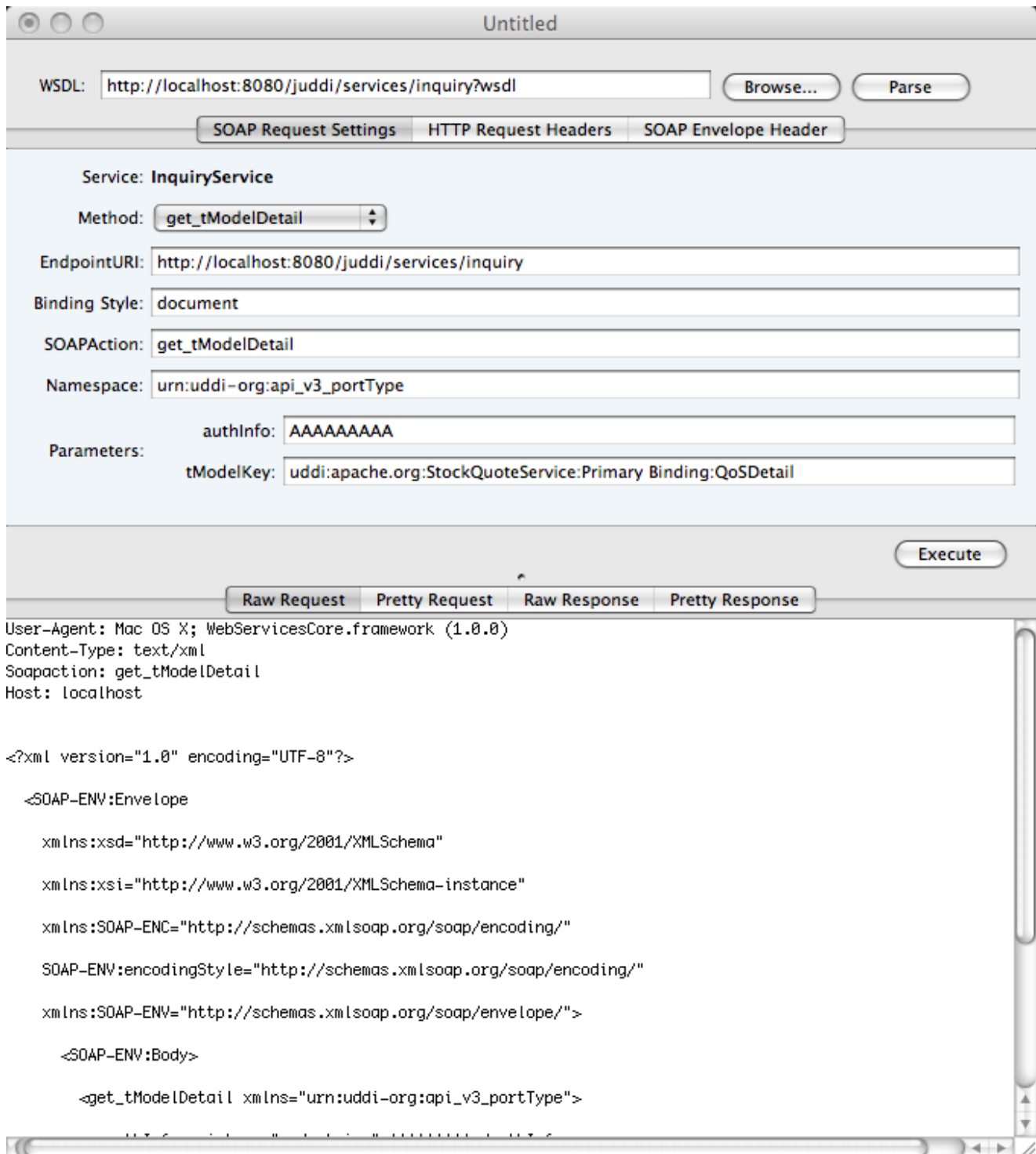


The screenshot shows a web browser window titled "CXF - Service list" with the address bar containing "http://localhost:8080/juddi/services/". The page content is as follows:

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. Using any SOAP client, you should be able to send some sample requests to jUDDI to test:



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 :

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.4.2</version>
  <configuration>
    <suiteXmlFiles>
      <suiteXmlFile>src/test/resources/suite-
init.xml,src/test/resources/suite-subscribe.xml,src/test/resources/suite-
clean.xml</suiteXmlFile>
    </suiteXmlFiles>
  </configuration>
</plugin>
```

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

Add your gpg key to KEYS

Release steps

Environment:

Maven version: 2.0.8

Java version: 1.5.0_16

OS name: "mac os x" version: "10.5.6" arch: "i386" Family: "unix"

1. Run mvn clean install -Prelease -Dpersistence=hibernate.
2. Copy the release artifacts to people apache.org:/public_html/releases/juddi-3.0.0.alpha
3. Start a vote referencing the build artifacts, leave the vote open for 72 hrs.

On successfull vote:

4. Create a Tag in SVN for the release
5. Run mvn clean deploy -Prelease -Dpersistence=hibernate
6. Update the website



Index
