Beginning Java Web Services

by Henry Bequet

Chapter 7: Publishing and Discovering Web Services

Overview

In Chapter 5, we reviewed the Web Service Description Language (WSDL), which provided us with a standard way to describe web services in human and machine terms. As we mentioned in that chapter, having a description of the functionality and interface for our software is only half the battle when it comes to selling. We must have some well-known forum where software developers can **publish** their WSDL files, and where potential web service users can **discover** what is available. This situation is similar to what we face while running a bricks-and-mortar business – the business owner advertises in media such as the newspaper, the radio, or the Yellow Pages.

The Yellow Pages, or more generally the phonebook, is the model we are interested in when it comes to publishing web services. A key argument in favor of modeling the advertisement of web services after the phonebook is the fact that it provides more structure than other media. We will expand on this structure and its generalization to web services.

In this chapter we will see:

- Issues concerning web service publishing
- An overview of UDDI
- UDDI data structures
- Programming UDDI
- Some other publishing technologies: WS-Inspection and JAXR

We will see how the **Universal Description, Discovery, and Integration (UDDI)** industry standard extends the phonebook metaphor to provide a framework for the publication of web services. Armed with a good understanding of the concepts in UDDI, we will go from theory to practice:

- We will publish our `StockQuote` web service to publicly available UDDI registries
- We will also examine how a potential customer could find `StockQuote`

Let's start our journey in web service publishing by formalizing the problem that we are facing. A better definition of our problem will help us understand why UDDI is such a good solution.

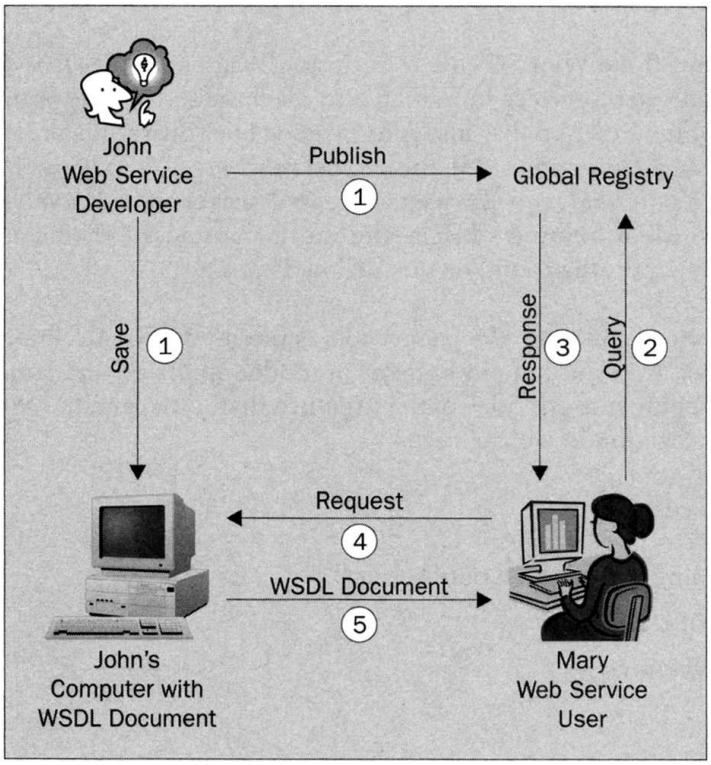
> **Note** In this chapter, we will use the terms registering and publishing interchangeably.

Web Service Publishing and Advertising

At first glance, the problem is simple – John, a software developer, has written a web service and the WSDL document that describes it. The question is – How can John reach Mary, another software developer and potential user of John's web service?

> **Note** Note that for the purpose of our discussion, a web service user is a software developer. We won't assume that someone has put a GUI on top of a web service to make it available to non-programmers.

If we assume that there is some public forum or global registry where John could advertise his web service, then the situation would reflect that depicted in the following diagram:



The steps are straightforward:

1. John publishes the web service to a global registry. At the same time, he saves the WSDL document describing his web service on his publicly available server.
2. Mary queries the global registry.
3. She gets a response from the global registry. When the response is positive, it contains at least one URL to a WSDL document. In this case, the URL points to John's machine.
4. Mary requests the WSDL document.
5. Mary receives the WSDL document.

We will refine the description of these steps shortly. Note that the diagram shown previously stops at the download of the WSDL file, which we will consider as the end of the discovery process for the remainder of our discussion.

We can see from the diagram that the global registry does not contain all the information. For instance the WSDL file, which we could compare to a brochure from a business, stays on John's computer. This is akin to a phonebook where we can get basic information on the purpose of the business, its phone number, and its address, but more detailed information such as a prospectus has to be retrieved from the business itself.

> **Note** The concept of a **directory** has been used in the software industry for quite some time now. For instance, the X.500 and the well-known **Lightweight Directory Access Protocol (LDAP)** are standards for the sharing of information without being repositories of all the information.

Let's spend a few more minutes discussing certain other issues regarding the global registry, which can be inferred from the previous diagram.

Universal

A phonebook must be usable by most people in the target audience. For instance, a phonebook in French published in Tuscaloosa, a city in Alabama, would have little chances of making any impact. To minimize the risk of irrelevancy, the global registry shown in the earlier diagram must be universal.

The fact that anybody who can read the local language can use a phonebook to find the name of a business, should translate into something like anybody with access to a computer can use the registry to find the name of a web service. In other words, any solution that will allow John to publish his web service must be accessible on most systems. For instance, a solution only available on an Apple Talk network using the Extended Binary-Coded Decimal Interchange Code (EBCDIC) would hardly qualify.

So if a universal web services registry should not use EBCDIC, what should it use? It should rely on standards that are accepted by a majority of companies (for example Microsoft, IBM, Sun, BEA) and implemented on a majority of platforms (for example Linux, Windows, Solaris, AIX). Standards such as HTTP, Unicode, XML, and SOAP would meet these two criteria.

Description

So far, we have agreed that our web service phonebook must be available on most platforms, but we still have not put any content in the phonebook. Simply stated, the registry must provide some form of description of web services. Ideally that description should meet the two requirements that we stipulated for WSDL in Chapter 5 – human readability and machine readability.

Discovery

When we open the phonebook, we can either look up businesses by name (White Pages) or by category (Yellow Pages), so the web service phonebook must also have some classification. As we mentioned in the previous paragraph, the taxonomy should be friendly to both humans and machines.

> **Note** Note that this issue of discovering components is not new to the software industry. Among other possibilities, the Common Object Request Broker Architecture (CORBA) supports a locator capability.

Integration

This last requirement is similar to the ubiquity of the registry that we mentioned previously, but is more concerned with the use of the web service. Once Mary has found that John's web service is a perfect match for her problem, it would be quite a disappointment to find out that she cannot use the software because it is limited to a platform or a programming language that she does not have access to. So our web service should be easy to integrate into existing applications, or at least be functional on many platforms.

UDDI Overview

UDDI and other technologies address the requirements that we presented in the previous section. We will be spending most of our time on UDDI, but we will also discuss some of the possible alternatives.

As web services are quite popular at the time of writing, there is a tendency to assume that everything about them is quite revolutionary. This is not necessarily so, especially when it comes to UDDI. We have already mentioned existing technologies like XML and HTTP that solve, at least partially, the problems tackled by UDDI. More precisely, we will see shortly that UDDI consists mainly of existing technologies and standards.

Let's start our discussion of UDDI with some historical perspective.

History of UDDI

UDDI grew out of the collaboration of three major industry players - IBM, Microsoft, and Ariba. IBM and Ariba have been active for a few years in the back-end services that make Business-to-Business (B2B) applications possible. IBM and Microsoft were among the companies that collaborated on the SOAP specification. Ariba and Microsoft have collaborated in the XML space for BixTalk and commerce XML (cXML). A notable absentee from the list is Sun Microsystems that embraced UDDI a couple of weeks after its announcement. Sun Microsystems is now a member of the UDDI community.

The UDDI organization runs a web site at http://www.uddi.org that serves both as a documentation site and as a registry of services available over the web. The community of companies that support UDDI is growing steadily and at the time of this writing, counts over 300 members.

During the summer of 2001, Ariba dropped out of the triumvirate, and now IBM and Microsoft run the site. However, the support of UDDI is not fading as other companies like HP, SAP, and more recently non-software companies such as NTT (Nipon Telephone & Telegraph) have joined the fray to operate their own UDDI registry.

The UDDI registry on http://www.uddi.org went live on May 2, 2001.

Core Technologies

To satisfy the goal of ubiquity that we discussed in the requirements phase, the architects of UDDI decided to rely as much as possible on existing open and interoperable standards. The major technologies that UDDI relies on are:

- TCP/IP, HTTP, and HTTPS
- XML, XML Schemas, and WSDL
- SOAP

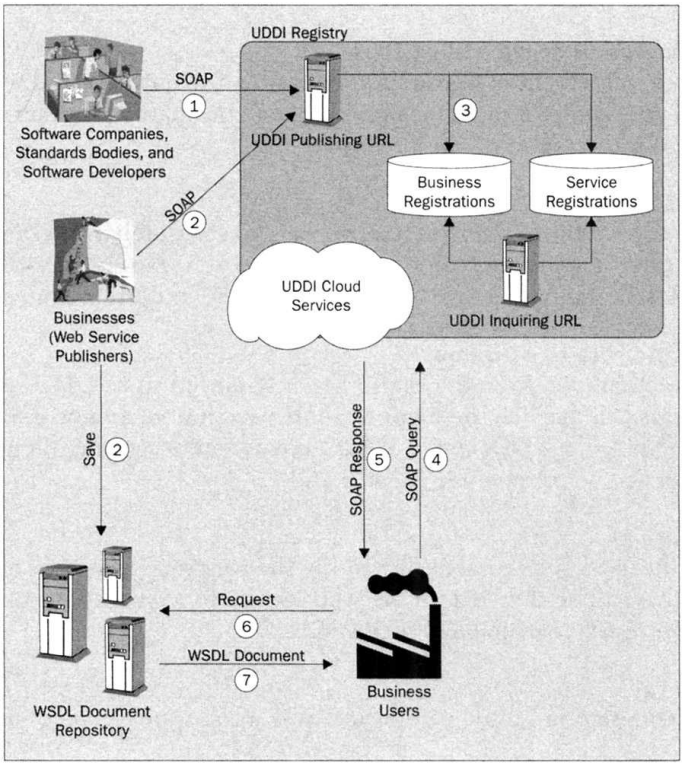
Some of us might argue that packaging open protocols under one umbrella is no technology, but in the mind of the UDDI partners, allowing providers and consumers of web services to keep their existing solutions was a key requirements, because the partners themselves have radically different solutions to common problems.

Methodology

Before diving into the methodology proposed by UDDI, let's talk about its applicability. UDDI is not exclusively reserved for web services. As

we will see shortly, the methodology, the model, and the taxonomies (discussed next) proposed by UDDI apply quite nicely to products other than web services.

Let's modify the earlier diagram that showed the interactions between John and Mary to explain the approach proposed by UDDI:



Fundamentally, nothing has changed between the earlier diagram of John and Mary and the one above. Again, the methodology described here is heavily geared towards web services described in WSDL, but other scenarios are also possible.

UDDI registries are linked to one another to form a global view of the available web services. This global view formed by the public UDDI registries is called the **UDDI cloud.**

The process presented in the above diagram consists of seven steps:

1. **Publish Web Service Types**
   In this first step, we separate the roles that were actually played by John in the earlier figure. The separation is based on splitting a web service into a web service type and a web service occurrence, in the same way that we would split an object between a class and the instance of a class.
   The type of service should span more than one company to promote reuse. Specifically, when two web services share the same WSDL interface definition, a consumer can easily switch from one to the other. This is a practical example of the benefits gained from the split of a WSDL document into two documents.

2. **Publish Web Service**
   It is the responsibility of businesses and software developers to publish web services according to the specifications published in Step 1. In the case of `StockQuote`, we will assume the roles of web service type publisher and web service publisher at the same time.
   It is important to point out that the publishing of web service types and their implementations is done using SOAP. In case we are worried about security, we will be glad to know that the integrity of the data in transit is protected via the UDDI account and the use of HTTPS.

3. **Register Business and Service Identifiers**
   This is another refinement of our earlier diagram – the business and the services are registered separately. A many-to-one relationship allows one business to register all the web services that it supports under the same business entity.

4. **Query a UDDI Registry (Discovery)**
   Based on the fact that the businesses are registered separately from the services that they provide, this querying process can be done by businesses or by services. Once again, the API to query the UDDI registry uses SOAP packets as the underlying transport mechanism.

5. **Retrieve Web Service Description**
   The information coming from the registry is only partial since UDDI is based on the directory model. Typically, things like the name, a short description, and business contacts can be retrieved directly from UDDI, but a WSDL document describing the web service interface and endpoints will be retrieved in Steps 6 and 7.

6. **WSDL Query**
   The URL of the WSDL document describing the web service is returned in Step 5. Commonly, this will be the URL of the web service followed by `?WSDL` as in http://myserver/Services/StockQuote?WSDL.

7. **WSDL Retrieval**
   The WSDL requested in Step 6 is returned over a transport protocol like HTTP

We will see shortly that Steps 1 to 3 constitute the publishing process which is modeled by the **publishing API** and that Steps 4 to 7 make up the querying process, modeled by the **inquiry API.**

Now that we have a better understanding of the high-level process, we can spend some time in discussing the details. We have already reviewed WSDL in detail, so we need to concentrate on the model used to store and query web services. Notice that the UDDI designers kept the phonebook metaphor in mind when defining their model.

The UDDI Data Model

In this section, we will have a look at how the data is archived in a UDDI registry. First, we will have a high-level overview and discuss the organization of the database. Then we will look at the model with finer granularity and review the data types. We will use this knowledge when writing code against a UDDI registry.

Organization

To store business and service registrations, a UDDI registry uses an organization similar to a phonebook and contains three distinct types of page, as follows:

- **White Pages**
  As in an everyday phonebook, the White Pages are indexed by the names of the businesses. They contain information like the business name, contact, and phone numbers. In addition, they contain data that we usually do not find in a phonebook, such as credit ratings.

- **Yellow Pages**
  These sort businesses by category. In other words, the Yellow Pages represent taxonomy. In the phonebook, the definition of the taxonomies is typically up to the phone company or the publisher, but this solution would not satisfy the requirement of universal support. Rather than inventing new taxonomies from scratch, the UDDI architects decided to support well-established classifications like the North American Industry Classification System (NAICS), which provides common industry definitions for Canada, Mexico, and the United States (http://www.census.gov/epcd/www/naics.html). There is also support for an UN-sponsored taxonomy (see http://www.uddi.org for details).

- **Green Pages**
  These contain the business and the technical descriptions of web services. When using WSDL, the technical information is contained within the WSDL document. We will see how this works in practice shortly.

Now that we have a grasp on the organization of the UDDI registry, let's zoom in and review the data structures.

Data Structures

The following UDDI data structures are used during the publication and discovery of web services:

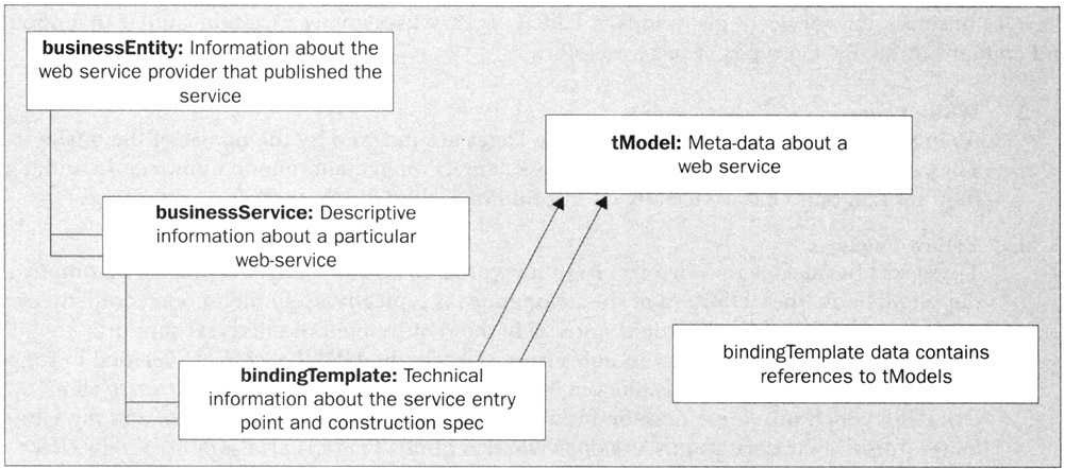
- `businessEntity`
  This contains general information about the service provider, and a collection of services (the `businessServices` element).

- `businessService`
  The `businessService` record is the essence of the Green Pages since it contains the business and technical descriptions of a web service. It is a container for a set of related web services. Examples of related web services are all the services for shipping or all the services for customer relationships management. The technical description of a web service is stored in a `bindingTemplate` container.

- `bindingTemplate`
  This is a container for a set of `tModels` (type of service), the actual technical description of the service. More precisely, a `bindingTemplate` is a container for `tModelInstanceInfo` records, which refer to the `tModel`.

- `tModel`
  This is metadata about the service; it is a *model* for a *type* of service. We will be looking at `tModel` in detail.

These data structures are represented in the following diagram:



Model

The `tModel` records contain three kinds of fields:

- The name of the service
- The description of the service
- A set of URL pointers to the actual specifications of the web service

The specification for a `tModel` defines three attributes and five elements as we can see in the following XML schema snippet (the full UDDI schema can be found at http://www.uddi.org/schema/uddi_1.xsd):

```
<element name="tModel">
  <type content="elementOnly">
    <group order="seq">
      <element ref="name" />
      <element ref="description" minOccurs="0" maxOccurs="*" />
      <element ref="overviewDoc" minOccurs="0" maxOccurs="1" />
      <element ref="identifierBag" minOccurs="0" maxOccurs="1" />
      <element ref="categoryBag" minOccurs="0" maxOccurs="1" />
    </group>
    <attribute name="tModelKey" minOccurs="1" type="string" />
    <attribute name="operator" type="string" />
    <attribute name="authorizedName" type="string" />
  </type>
</element>
```

The attributes of a `tModel` are:

- `tModelKey`
  The unique key assigned to the `tModel`. It is the only required attribute.
- `Operator`
  The name of the UDDI registry site operator. For example, IBM UDDI Test Registry.
- `AuthorizedName`
  The name of the person who registered the `tModel`. For example, Ron LeBossy.

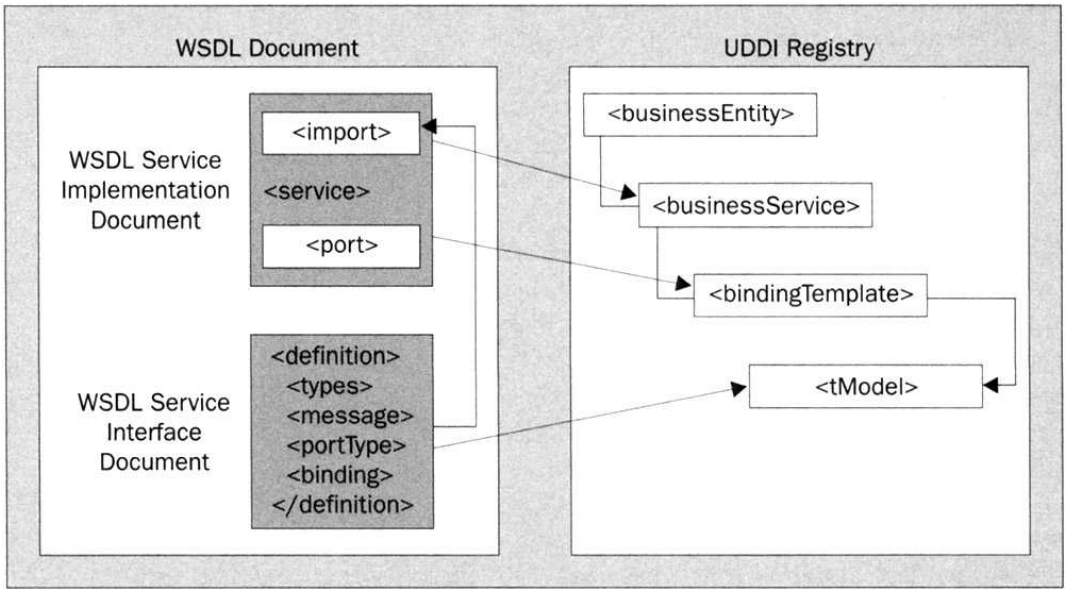
The elements of a `tModel` are:

- `name`
  A unique, human-readable identifier for the `tModel` record. The name is a required element.
- `description`
  A locale-aware description of the `tModel` record. The description is locale-aware because it carries a locale identifier, for example `en`.
- `overviewDoc`
  A container for remote descriptions and information. For instance, the overview URL contains a URL to a document that describes the web service. A recommended language to specify the description of a web service in a `tModel` is WSDL that we introduced in the previous chapter.
- `identifierBag`

An optional list of name-value pairs that can be used during search.

- `categoryBag`
  A container for one or more name-value pairs called key references. These allow for a flexible taxonomy of `tModels`. If WSDL is used to describe the service, then the `tModel` should be classified using the `uddi-org:types` taxonomy with the type `wsdlSpec`. We will use this model when registering the `StockQuote` service.

When using WSDL to describe a web service, the UDDI organization recommends a mapping between UDDI and WSDL data. This mapping is shown in the following diagram:



As we have just described, the `tModel` defines the type of web service, so it should come as no surprise that it is mapped to the definition of the web service – `<wsdl:types/>`, `<wsdl:message/>`, `<wsdl:portType/>`, and `<wsdl:binding/>`.

The `<wsdl:port/>`, which defines the URL of the web service finds its UDDI incarnation in the `<uddi:bindingTemplate/>` element. Finally, the WSDL implementation document corresponds to the `<uddi:businessService/>` element and its binding templates.

There are a couple of data structures that we have not discussed in detail because they are seldom used. For example, the `operationalInfo` structure includes data about the publishing of an entity, such as the creation time. The `publisherAssertion` structure provides a way to couple business entities stored in the registry.

This completes our UDDI overview. In the next section, we will go from theory to practice by looking at how John and Mary use UDDI as a facilitator of e-business.

Programming UDDI

We have seen in the UDDI overview that it provides four main capabilities. We can:

1. Register businesses
2. Register web service types
3. Register web services
4. Query for web services

We will start this section by publishing a business in the `RegisterBusiness` example and then we will publish a service along with its type in the `RegisterService` example. For the sake of brevity, we will combine the query of the UDDI registry with the registration.

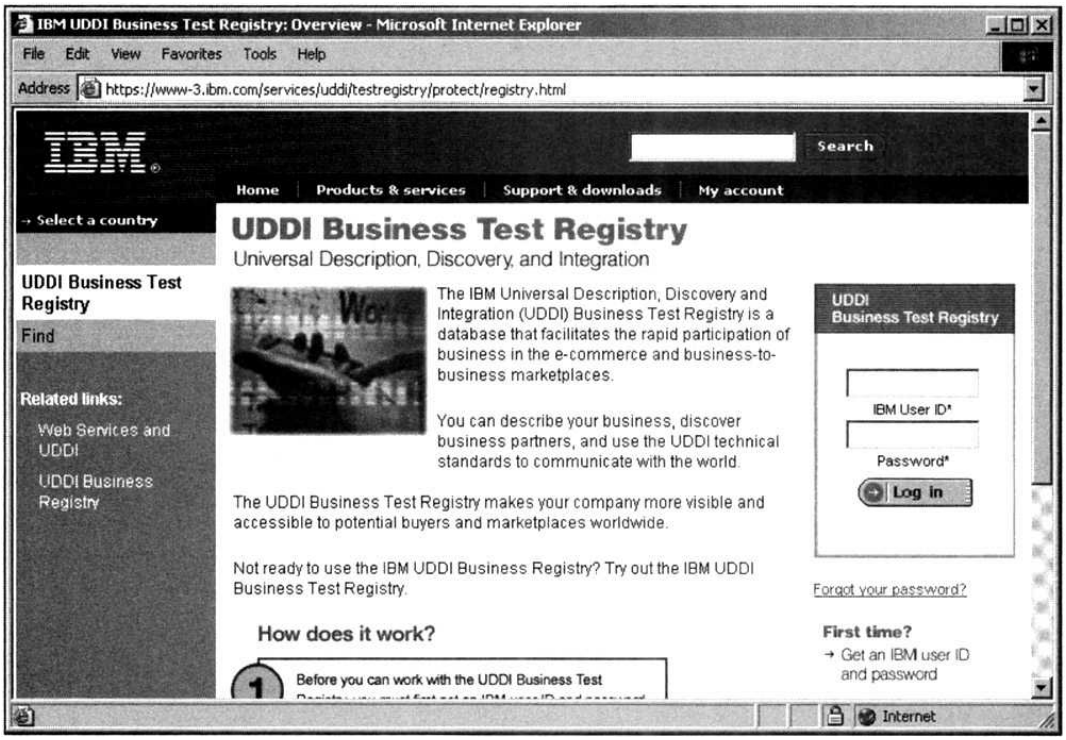
Note Note that the UDDI operators (IBM, Microsoft, HP) provide a GUI to both register and find web services according to the taxonomies that we have discussed in the UDDI overview. Before starting the next section, it might be a good idea to navigate to one of them and experiment with the concepts that we have discussed so far.

IBM maintains a test registry and a production registry:

- We can find the test UDDI registry at: https://www-3.ibm.com/services/uddi/testregistry/protect/ registry.html.
- The production UDDI registry can be found at: https://www-3.ibm.com/services/uddi/protect/registry. html.

We do not need a user name to query the registry, but we need to obtain a valid login in order to register our own business and services. To

obtain a user name and password, simply follow the `First Time` link as shown in the screenshot below. The process of getting an account is straightforward – enter a user name and password along with the contact information. Remember to keep the user name-password combination handy since we will be requiring it to run the `RegisterService` example:



This URL is to register new business and services (notice the HTTPS). To simply query the registry, follow the Find link above; this will take us to a URL that does not use HTTPS.

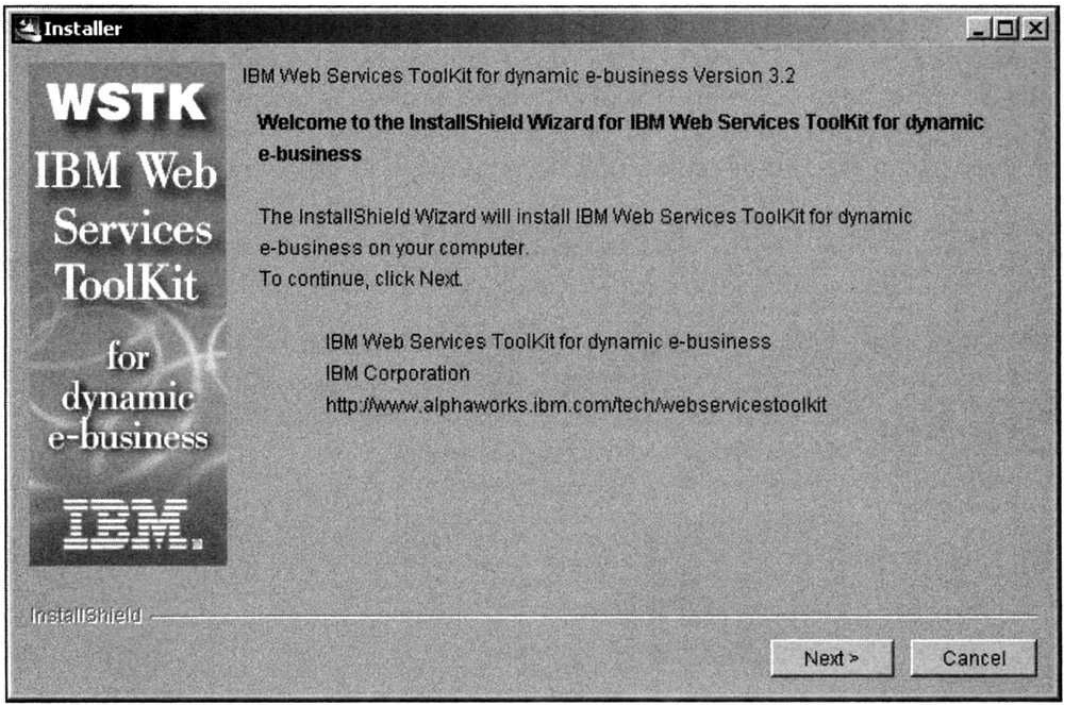
UDDI requests are in fact SOAP packets. So to program against a UDDI registry, one could put together SOAP packets using, for instance, the Axis client API, and call UDDI methods. There is an easier path – use the UDDI for Java API (UDDI4J API) from IBM that comes with the Web Service Toolkit (WSTK). The main benefit of the UDDI4J package is that it provides a hierarchical model of the UDDI registry, which is (mostly) isolated from the intricacies of the UDDI SOAP API.

Our next step will be to install the WSTK on our computer.

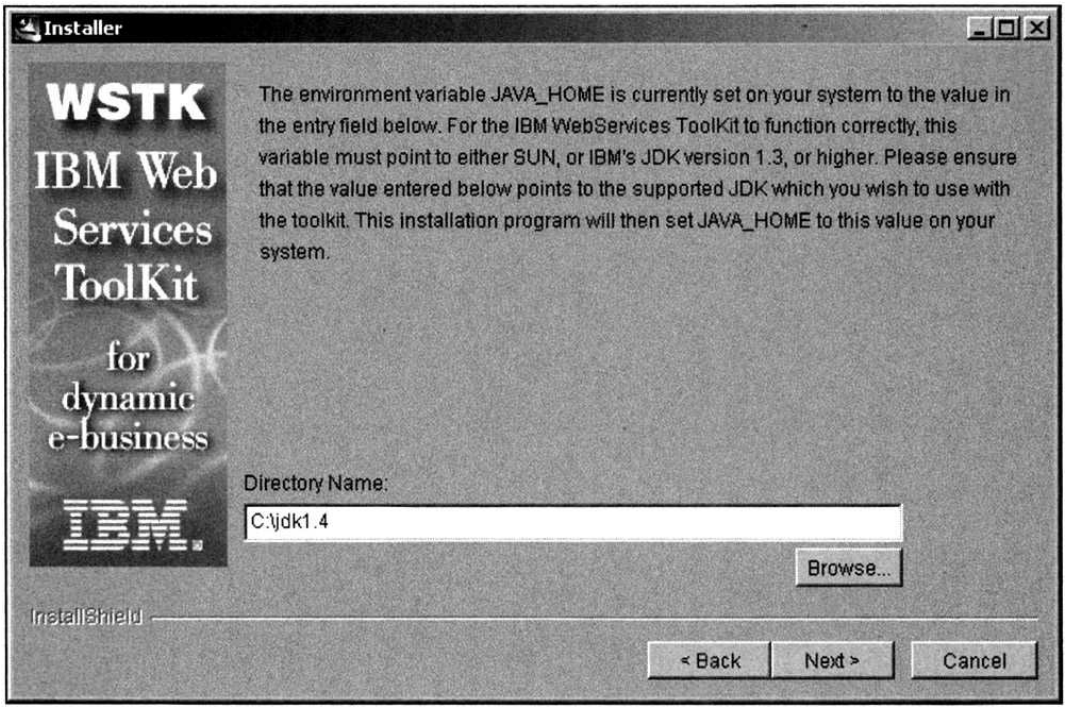
Web Services Toolkit (WSTK)

Download the latest version of WSTK from http://www.alphaworks.ibm.com/tech/webservicestoolkit.

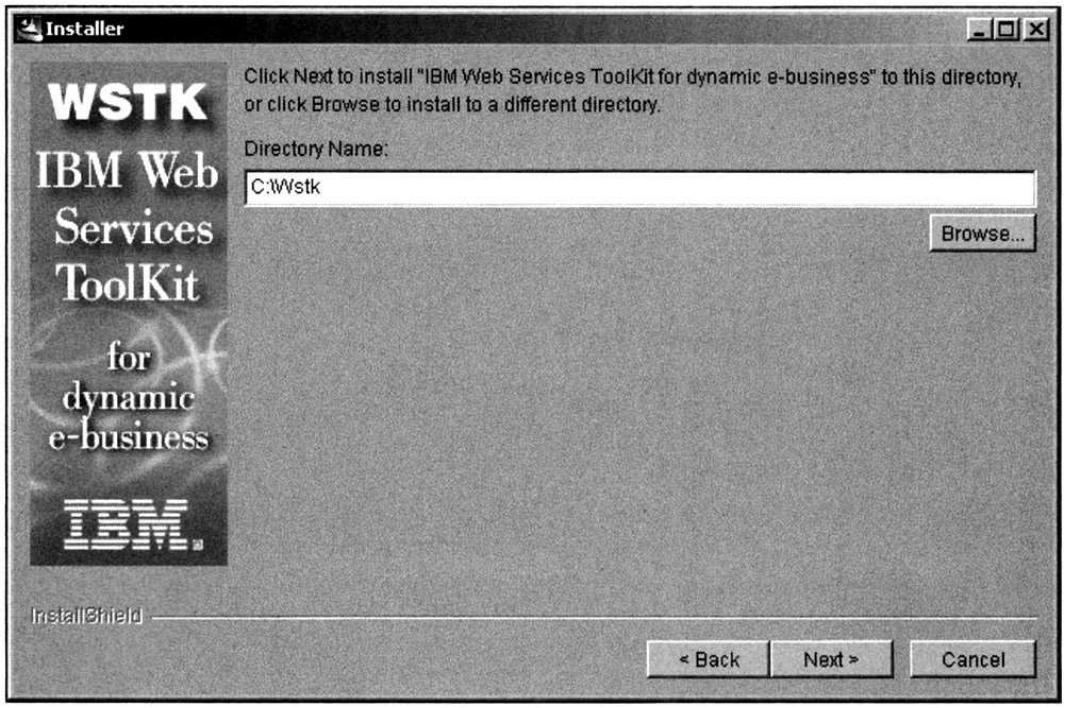
Once the installation file is downloaded to our machine, we can start the actual installation by double-clicking on it. Now, the first screen will appear. As we can see from the following screenshot, the current version at the time of this writing is 3.2:



WSTK 3.2 requires Java (Standard Edition) 1.3 or higher. The installation is not without a few detours, so let's take a few minutes to review it. When we click Next we see one screen for the unavoidable license agreement, followed by one to confirm the location of our JDK directory (in our case it's `c:\jdk1.4`):

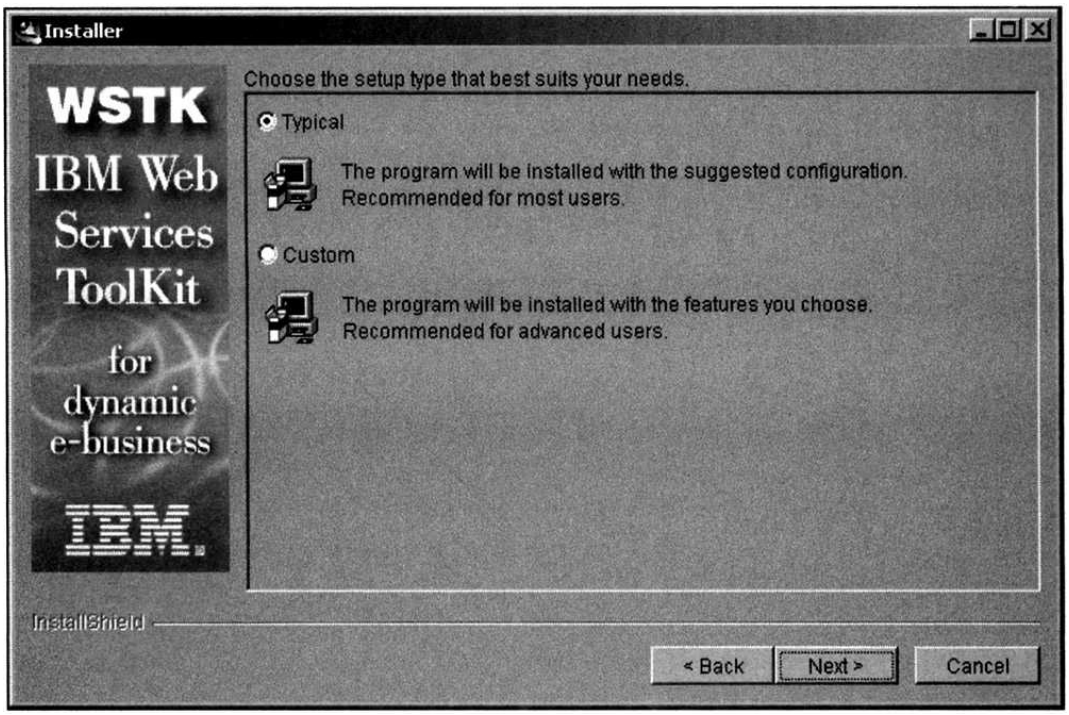


The next screen asks us for the WSTK installation directory. In our case, we will be installing it in the `C: \Wstk` directory:

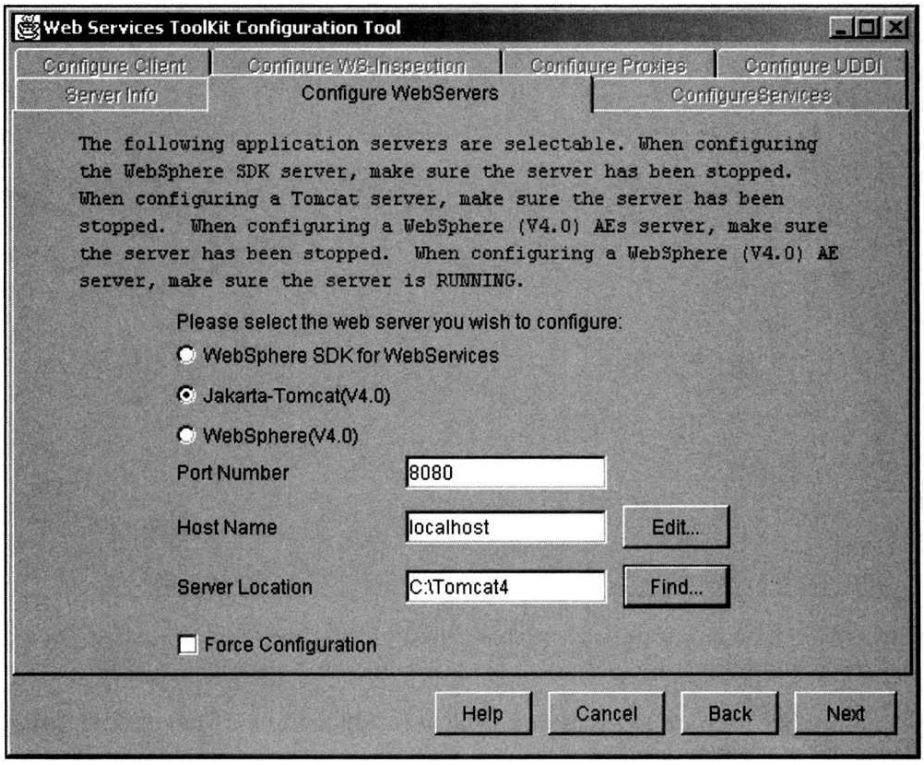


During the installation, an environment variable called `wstk_home` is also created; it contains the value of the WSTK installation directory.

Then it asks us whether we want to perform a `Typical Installation` or a `Custom Installation`. We will select the `Typical Installation`:



Things get slightly more demanding at the end of the installation process when the setup asks us to configure several components as shown in the following screenshot:



If we change our mind later about one of our choices regarding the configuration of WSTK, we can invoke the configuration tool by typing `%wstk_home%\bin\wstkconfig` (`wstk_home` is the directory where WSTK is installed).

The following components need to be configured (use `Next` and `Back` buttons to navigate between the different tabs):

- **Server Info**
  This tab provides information about the servers that we can use with WSTK. We can use WebSphere SDK, Jakarta Tomcat, or WebSphere (v 4.0).
- **Configure Web Servers**
  This option allows us to configure the web server for working with WSTK. We will be using Tomcat. Please refer to Chapter 3 for specifics on the Tomcat installation. For configuring Tomcat, we will use the parameters shown in the previous screenshot.
- **Configure Services**
  WSTK comes with several web services; however, we will not be using them in this book. So we can safely select the web services that we would like to try and unselect the others. However, beware that any web service that we select will be added to our `WebApps` directory, thereby making the process of loading our servlet engine (much) slower.
- **Configure WS-Inspection**
  Accept the defaults. We will have a short discussion on WS-Inspection at the end of this chapter.
- **Configure Proxies**
  If we are working through a proxy, we will have to configure WSTK for our proxy.
- **Configure UDDI**
  We need to enter the user name and password for the UDDI registry (IBM, Microsoft). We will assume that we have an account on the IBM Test Registry for the remainder of this chapter. Adapting our discussions to another registry like Microsoft's is straightforward.
- **Configure Client**
  We need to enter the host name and port numbers that will be used by clients. Since we are using Tomcat on the local machine, we need to enter `localhost` for the host name and `8080` for the port number.

Note The IBM and Microsoft UDDI registries are linked since they are part of the UDDI cloud that we mentioned earlier. So after a few hours, a business created in one shows up in the other. However, beware of interoperability issues. A business created in one UDDI registry cannot be edited in another. For instance, we cannot create a business in the IBM registry and add services to it using the Microsoft registry. We will get an `E_operatorMismatch` error with a description stating that the business is invalid.

Now that we have an account with a UDDI provider and the software packages to facilitate the use of UDDI, we can move on to our first example in which we will publish the web service that we developed in Chapter 3.

Publishing the jws Business

We know that the goal of publishing a web service is providing a service for someone or something. It is therefore not surprising to find out that the first step in registering a web service is actually to register a **business**. A new business can be registered via the GUI provided by most operators or it can be registered programmatically. We illustrate the latter in the `RegisterBusiness` example.

Try It Out: Publishing a Web Service

The `RegisterBusiness` class adds a business entity called `jws` to the IBM UDDI test registry.

1. Write the following into the `RegisterBusiness.java`:

```
package com.wrox.jws.stockquote;

import org.uddi4j.UDDIException;
import org.uddi4j.client. UDDIProxy;
import org.uddi4j.response. DispositionReport;
import org.uddi4j.datatype.business.BusinessEntity;
import org.uddi4j.datatype. Name;
import org.uddi4j.response.AuthToken;
import org.uddi4j.response.BusinessDetail;
import org.uddi4j.response.BusinessList;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.util.FindQualifier;
import org.uddi4j.util.FindQualifiers;
import java.util.Vector;

public class RegisterBusiness {</pr>
```

The `RegisterBusiness` class contains only one method, `main()`, in which we register a business and then query for its existence:

```
public static void main (String args[]) {
  String methodName = "com.wrox.jws.stockquote.Register';
  System.out.println (methodName + ": Starting...");

  // Enable https
  System.setProperty ("java.protocol.handler.pkgs",
    "com.ibm.net.ssl.internal.www.protocol");
  java.security.Security.addProvider (new com.ibm.jsse.JSSEProvider());
  // Construct a UDDIProxy object
  UDDIProxy proxy = new UDDIProxy();

  try {
    String bizName = "jws";
    String username = "hbequet";
    String password = "wroxpress";
    String inquiryURL =
      "http://www-3.ibm.com/services/uddi/testregistry/inquiryapi";
    String publishURL =
      "https://www-3.ibm.com/services/uddi/testregistry" +
      "/protect/publishapi";
    // Select the desired UDDI server node
    proxy.setInquiryURL(inquiryURL);
    proxy.setPublishURL(publishURL);

    // Set the transport to use Axis (default is Apache SOAP)
    System.setProperty("org.uddi4j.TransportClassName",
      "org.uddi4j.transport.ApacheAxisTransport");

    System.out.println(" Getting authorization tokens...");

    // Get an authorization token from the UDDI registry
    AuthToken token = proxy.get_authToken(username, password);
    // The minimal business entity contains a business name.
    Vector bizEntities = new Vector (1);
    BusinessEntity bizEntity = new BusinessEntity(" ", bizName);
    bizEntities.addElement (bizEntity);

    System.out.println(" Saving business...");

    // Save the business. We get an instance of BusinessDetail in return
    BusinessDetail bizDetail = proxy.save_business
      (token.getAuthInfoString(), bizEntities);
```

```
        System.out.println (" Business saved!");

        String bizKey = ( (BusinessEntity) (bizDetail.getBusinessEntityVector().
          elementAt (0))).getBusinessKey();

        System.out.println(" Business key: " + bizKey);
        System.out.println(" Verifying results...");
```

The next section of code queries UDDI for the business that we just registered. Typically, this step is not performed in a production application:

```
        // To check that everything went fine, we find the business that we
        // just added using its name (the only thing we have saved).
        Vector names = new Vector (1);
        names.add(new Name (bizName));

        BusinessList bizList = proxy.find_business (name, null, null,
          null, null, null, 0);
        Vector bizInfoVector =
          bizList.getBusinessInfos() .getBusinessInfoVector();
        System.out.println (" We found the following businesses:");

        for (int index = 0; index < bizInfoVector.size(); index++) {
          BusinessInfo bizInfo =
            (BusinessInfo)bizInfoVector.elementAt (index);
          System.out.print("    - " + bizInfo.getNameString());

          if (bizInfo.getBusinessKey() .equals (bizKey)) {
            System.out.print (" (the one we just added)");
          }
          System.out.println();
        }
```

The remainder of the code contains our exception handling:

```
      } catch (UDDIException uddiException) {
        DispositionReport report = uddiException.getDispositionReport();

        if (report != null) {
          System.err.println (methodName + ": Caught UDDIException!");
          System.err.println (" UDDIException faultCode:"
            + uddiException.getFaultCode()
            + "\n Operator:" + report.getOperator()
            + "\n Generic:" + report.getGeneric()
            + "\n Errno:" + report.getErrno()
            + "\n ErrCode:" + report.getErrCode()
            + "\n InfoText:" + report.getErrInfoText());
        }
      } catch (Exception exception) {
        System.err.println (methodName + ": Caught Exception!");
        exception.printStackTrace();
      }
      System.out.println(methodName + ": All done!");
    }
  }
```

Let's now build and test the `RegisterBusiness` class.

Building `RegisterBusiness` requires `uddi4j.jar, xmlParserAPIs.jar`, and `ibmjsse.jar` in our classpath. The `ibmjsse.jar` file contains the support for HTTPS.JSSE stands for Java Secure Socket Extension.

> **Note** Note that if we are using the JDK 1.4 or have another JSSEJAR file in our classpath, we can omit `ibmjsse.jar` from the classpath.

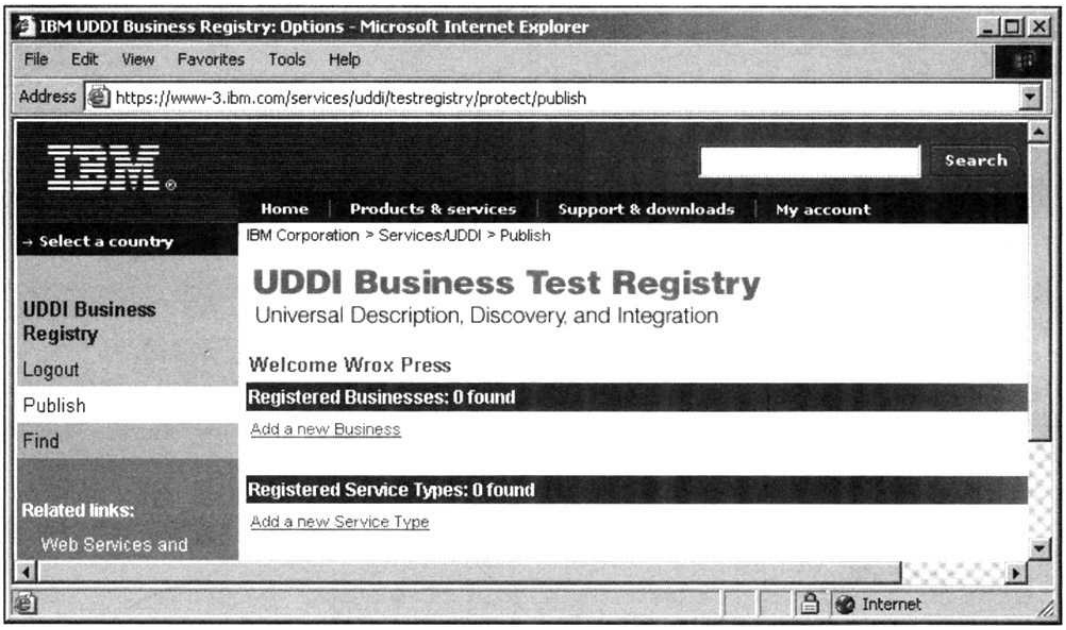
To compile the file `RegisterBusiness.java` file on Windows 2000, change to the `Register\src` directory and type the following commands (or their equivalent, in case of the Linux platform):

```
set classpath=%wstk_home%\uddi4j\lib\uddi4j.jar;%wstk_home%\lib\ibmjsse.jar;
              %jwsDirectory%\lib\xmlParserAPIs.jar;

javac -d ..\classes com\wrox\jws\stockquote\RegisterBusiness.java
```

These commands will produce a `RegisterBusiness.class` file in the `register\classes\com\wrox\jws\stockquote`

directory.

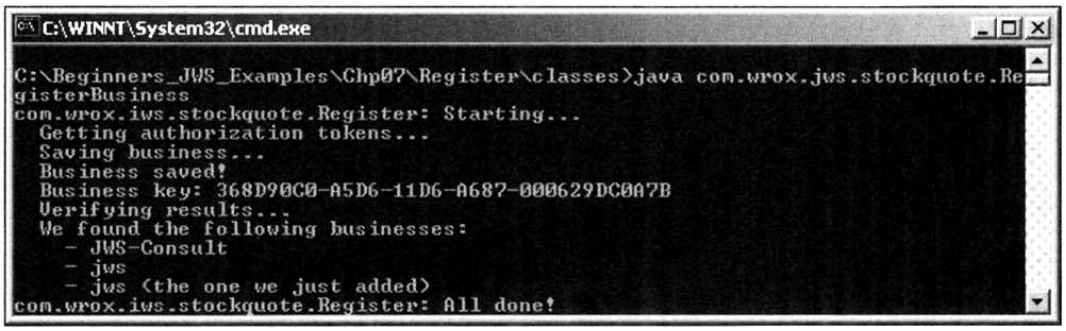
Before running the `RegisterBusiness` class, let's have a look at the IBM UDDI Test Registry. If we log on successfully, the page should look like the following, with no businesses and no services listed:



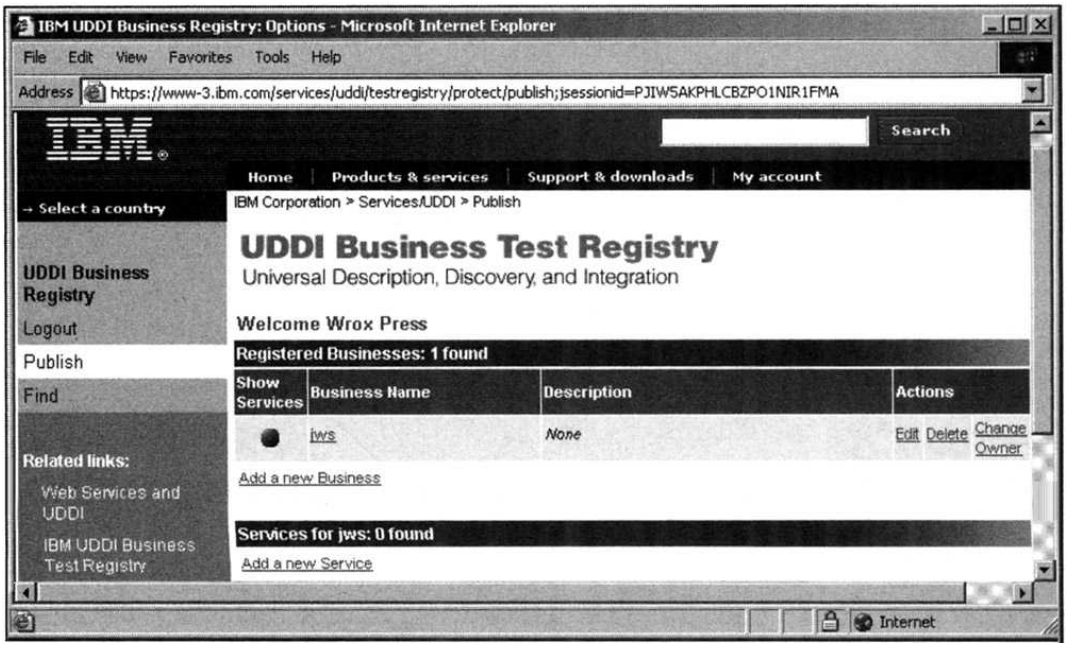
Now if we run the example with the following command (make sure that `RegisterBusiness.class`, `ibmjsse.jar` and `uddi4j.jar` are in our classpath and that we are in the `Register\classes` directory):

```
java com.wrox.jws.stockquote.RegisterBusiness
```

we will get the following result:



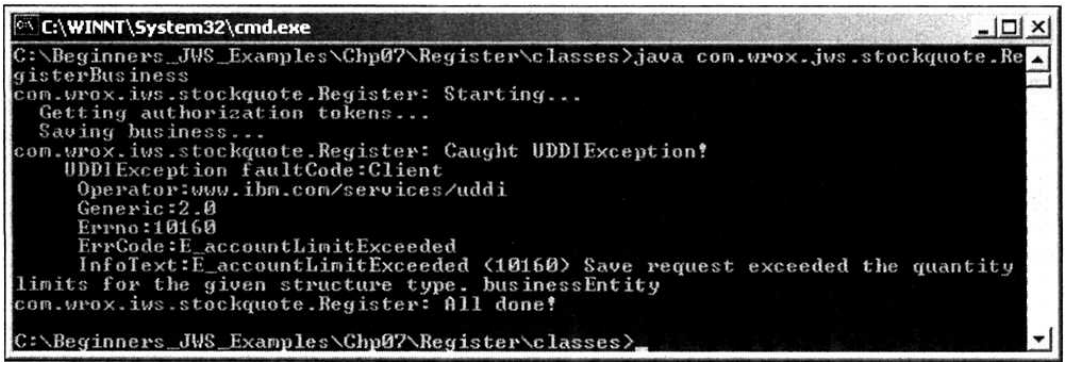
The refreshed browser page should show that the `jws` business has been registered successfully:



The business still has not registered any services, but we will address that shortcoming in the next section.

Note Note that the business key is a **Universal Unique Identifier** (UUID) – `2BA70F90-993A-11D6-9880-000629DC0A53`. A UUID is a 128-byte number generated with an algorithm that guarantees its uniqueness. Typical implementations rely on the address of the computer and the time of the day.

Bear in mind that if we try to run the `RegisterBusiness` class a second time against the IBM test registry, we will get an error since we can only register one business:



Let's now have a closer look at the implementation of the `RegisterBusiness` class.

How It Works

The `RegisterBusiness` class adds a business entity called `jws` to the IBM UDDI test registry. As we can see in the following listing, we make extensive use of the UDDI4J (`org.uddi4j`) package that comes with WSTK. The `uddi4j.jar` file can be found in the `uudi4j\lib` directory under the WSTK installation directory:

```
package com.wrox.jws.stockquote;

import org.uddi4j.UDDIException;
// Other imports omitted from this listing
```

We will discuss each class that we import from the UDDI4J package as we encounter them in the code:

```
public class RegisterBusiness {

  public static void main (String args[]) {
    String methodName = "com.wrox.jws.stockquote.Register";
```

We use the `methodName` variable to prefix our messages with the fully qualified `main()` method.

```
// Enable https
System.setProperty ("java.protocol.handler.pkgs",
  "com.ibm.net.ssl.internal.www.protocol");
java.security.Security.addProvider (new com.ibm.jsse.JSSEProvider());
```

We have seen earlier in this chapter that publishing to a UDDI directory entails the use of HTTPS for security reasons. If we don't enable HTTPS, the call to `setPublishURL()` below will fail with an `"unknown protocol"` error.

The next step is to create a UDDI proxy that will take care of generating the SOAP calls to UDDI and parsing the SOAP results from UDDI. Note that this proxy is different from the WSDL-generated proxy that we discussed in Chapter 5 since it is a generic proxy not tied to a specific web service:

```
// Construct a UDDIProxy object
UDDIProxy proxy = new UDDIProxy();

try {
  String bizName    = "jws";
  String username   = "hbequet";
  String password   = "wroxpress";
  String inquiryURL =
    "http://www-3.ibm.com/services/uddi/testregistry/inquiryapi";
  String publishURL =
    "https://www-3.ibm.com/services/uddi/testregistry/" +
    "protect/publishapi";
```

To run the example, we will need to update the user name and password to what we specified when we created the user account with the provider, as we discussed earlier. The `inquiryURL` is the URL used to query UDDI and the `publishURL` is the URL used to publish businesses and web services in UDDI.

Feel free to replace the IBM UDDI test registry URL by another provider. For instance, the following URLs are for the Microsoft UDDI registry (remove the `"test."` at the beginning of the URL to get to the production site):

- https://test.uddi.microsoft.com/inquire
- http://test.uddi.microsoft.com/publish

Once again, the publishing URL is protected through HTTPS. The next two lines of code simply tell the proxy which URL to use:

```
// Select the desired UDDI server node
proxy.setInquiryURL (inquiryURL);
proxy.setPublishURL (publishURL);
```

Now, we need to tell the UDDI4J package what SOAP client package to use when sending requests to the UDDI provider. Since we have been using Axis in most of this book, we will use it here as well:

```
// Set the transport to use Axis (default is Apache SOAP)
System.setProperty ("org.uddi4j.TransportClassName",
  "org.uddi4j.transport.ApacheAxisTransport");
```

The next statement requires a little bit of background information. The security model used by UDDI is based on user name and password arguments passed in every call. However, this implies that for every call, the server will have to perform credential validation. To lessen the burden on the server, the UDDI protocol contains the following compromise – the caller must obtain a security token from the server and pass that security token with every request to UDDI. To add an extra level of safety, the security token is only valid for a limited period of time. We will discuss other security models in detail later in the book, but for now, let's get a security token from the UDDI operator:

```
// Get an authorization token from the UDDI registry
AuthToken token = proxy.get_authToken(username, password);
```

In the following section of code, we create a `BusinessEntity` structure and submit it to the UDDI operator for registration. The `BusinessEntity` is a container for information about the business; in other words, it stores the White and Yellow Pages data.

Note that we have to use a vector since the API is geared toward the registration of several businesses with one call to minimize network traffic. A more detailed discussion of the data types defined by UDDI can be found at http://www.uddi.org/pubs/DataStructure-V2.00-Open-20010608. pdf:

```
// The minimal business entity contains a business name.
Vector bizEntities = new Vector (1);
BusinessEntity bizEntity = new BusinessEntity (" ", bizName);
bizEntities.addElement (bizEntity);

// Save the business. We get an instance of BusinessDetail in return
BusinessDetail bizDetail = proxy.save_business (
  token.getAuthInfoString(), bizEntities);
```

As we mentioned earlier, the security token must be passed with each call that requires authentication. It is worth pointing out that more

detailed information about the business such as the address, the phone number, and contact information can be added to the `BusinessEntity` structure.

The return value of `save_business()` is a `BusinessDetail` structure that contains most of the information saved in the registry. We use it in the next section of code to display the business key, a unique identifier assigned by the UDDI operator. Once again, notice the use of the vector that makes the API a little tedious:

```
String bizKey = ((BusinessEntity) (bizDetail.getBusinessEntityVector()
  elementAt (0))).getBusinessKey();
```

If we have got this far, then our business has been successfully registered. If something goes wrong, we are likely to get an `UDDIException` or an `IOException`. We handle all exceptions at the end of this method. The rest of the code goes back to the UDDI registry and verifies that our business actually got registered. In practice this is not necessary, but it is a good time for a short introduction to UDDI queries.

The code to query a UDDI registry is similar to the publishing code, except for two main differences – the inquiry URL differs from the publishing URL, and the inquiry URL does not require the use of HTTPS. As we can see below, to query UDDI, we simply pass the business name. We will discuss more complicated queries based on taxonomies shortly. Once again, notice the use of vectors:

```
// To check that everything went fine, we find the business that we
// just added using its name (the only thing we have saved).
Vector names = new Vector (1);
names.add(new Name (bizName));

BusinessList bizList = proxy.find_business (names, null, null, null,
  null, null, 0);
```

The last argument of the `find_business()` method specifies the maximum number of rows returned in the result (0 means no maximum). The other arguments specify more criteria like the URL or the taxonomy of the business. The return value of `find_business()` is a vector of `BusinessInfo` structures that we use to print the business name and key:

```
Vector bizInfoVector =
  bizList.getBusinessInfos() .getBusinessInfoVector();
System.out.println(" We found the following businesses:");
for (int index = 0; index < bizInfoVector.size(); index++) {
  BusinessInfo bizInfo =
    (BusinessInfo)bizInfoVector.elementAt (index);
  System.out.print (" - " + bizInfo.getNameString());

  if (bizInfo.getBusinessKey().equals (bizKey)) {
    System.out.print (" (the one we just added)");
  }
  System.out.println();
}
```

The last task on our list is to take care of errors. The `UDDIException` class gives us detailed information about what went wrong. As we might expect from an API that uses SOAP, UDDI returns errors in the form of SOAP packets. The error message contains a `<uddi:dispositionReport/>` element that is a container for a `<uddi:result/>` element, which holds the error code and the error info. The `DispositionReport` class models the error message as we can see in the exception handling block below:

```
    } catch (UDDIException uddiException) {
      DispositionReport report = uddiException.getDispositionReport();

      if (report != null) {
        System.err.println (methodName + ": Caught UDDIException!");
        System.err.println (" UDDIException faultCode:"
          + uddiException.getFaultCode()
          + "\n     Operator:" + report.getOperator()
          + "\n     Generic:"  + report.getGeneric()
          + "\n     Errno:"    + report.getErrno()
          + "\n     ErrCode:"  + report.getErrCode()
          + "\n     InfoText:" + report.getErrInfoText());
      }
    } catch (Exception exception) {
      System.err.println (methodName + ": Caught Exception!");
      exception.printStackTrace();
    }

    System.out.println (methodName + ": All done!");
  }
}
```

Let's now move to the next level and register the `StockQuote` service.

Publishing StockQuote

We will assume that we have created a business on the IBM Test Registry by running the `RegisterBusiness` example using our own UDDI account (that is with the user name and password that we obtained from the UDDI operator as previously).

Try It Out: Registering a Web Service

1. As we can see, the code of `RegisterService` has quite a few similarities to the `RegisterBusiness` class:

```
package com.wrox.jws.stockquote;

import org.uddi4j.UDDIException;
import org.uddi4j.client.UDDIProxy;
import org.uddi4j.response.DispositionReport;
import org.uddi4j.datatype.service.BusinessService;
import org.uddi4j.datatype.Name;
import org.uddi4j.datatype.binding.AccessPoint;
import org.uddi4j.datatype.binding.BindingTemplates;
import org.uddi4j.datatype.binding.BindingTemplate;
import org.uddi4j.datatype.binding.TModelInstanceDetails;
import org.uddi4j.datatype.binding.TModelInstanceInfo;
import org.uddi4j.response.AuthToken;
import org.uddi4j.response.ServiceDetail;
import org.uddi4j.response.ServiceList;
import org.uddi4j.response.ServiceInfos;
import org.uddi4j.response.ServiceInfo;
import org.uddi4j.response.BusinessList;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.util.CategoryBag;
import org.uddi4j.util.KeyedReference;
import java.util.Vector;

public class RegisterService {

  private static final String businessKey =
    "2BA70F90-993A-11D6-9880-000629DC0A53";
  private static final String bizName = "jws";

  // The following values are for the category of
  // WSDL-described web services
  private static final String keyName = "uddi-org:types";
  private static final String keyValue = "wsdlSpec";
  private static final String tModelKey =
    "uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4";

  // Beginning of the main() method omitted from this listing

      // Get an authorization token from the UDDI registry
      AuthToken token = proxy.get_authToken (username, password);

      // We create the StockQuote service
      String serviceName = "StockQuote";
      BindingTemplates bindingTemplates = new BindingTemplates();
      BindingTemplate bindingTemplate = new BindingTemplate();
      Vector bndVector = new Vector();
      Vector tVector = new Vector();

      // We need a tModel reference to a web service described with WSDL
      TModelInstanceDetails tModelInstanceDetails =
        new TModelInstanceDetails();
      TModelInstanceInfo tModelInstanceInfo = new TModelInstanceInfo();
      tModelInstanceInfo.setTModelKey (tModelKey);
      tVector.addElement (tModelInstanceInfo);
      tModelInstanceDetails.setTModelInstanceInfoVector (tVector);

      // The access point is through http
      AccessPoint accessPoint = new AccessPoint (
        "localhost/axis/servlet/AxisServlet", "http");
```

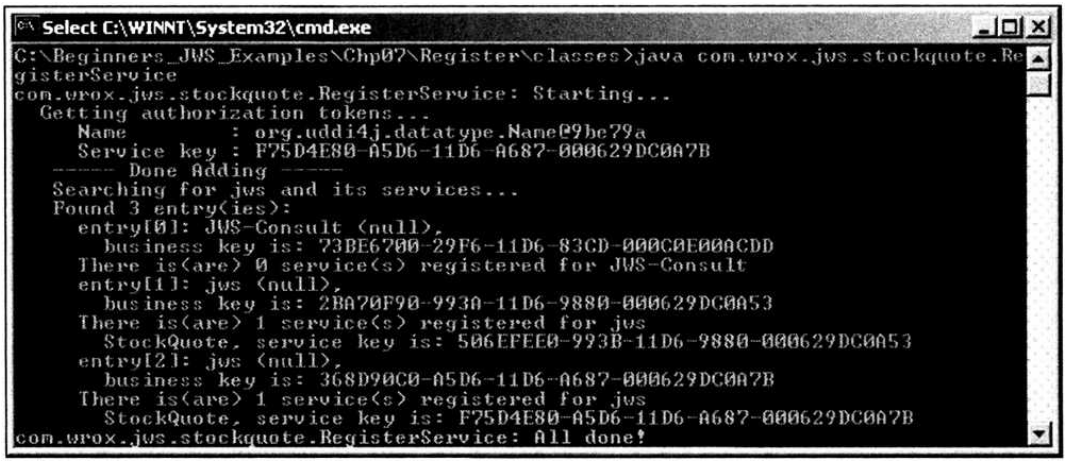
```
        bindingTemplate.setTModelInstanceDetails (tModelInstanceDetails);
        bindingTemplate.setAccessPoint (accessPoint);
        bindingTemplate.setBindingKey (" ");
        bindingTemplate.setDefaultDescriptionString ("Unsecure Stock quotes.");
        bndVector.addElement (bindingTemplate);
        bindingTemplates.setBindingTemplateVector (bndVector);

        BusinessService service = new BusinessService (" ");
        Vector sVector = new Vector(); // services
        Vector sVector = new Vector(); // service details

        // We add the binding templates to the business service
        service.setBindingTemplates (bindingTemplates);

        // We set the default service name (in English)
        service.setDefaultNameString ("StockQuote", "en");

        // Create a category bag with the WSDL tModel
        CategoryBag categoryBag = new CategoryBag();
        KeyedReference keyedReference = new KeyedReference (keyName,
          keyValue, tModelKey);
        categoryBag.add (keyedReference);

        // Save the service and its description in the UDDI registry
        service.setDefaultDescriptionString ("Stock quotes over the web");
        service.setBusinessKey (businessKey);
        service.setCategoryBag (categoryBag);
        sVector.addElement (service);

        ServiceDetail serviceDetail = proxy.save_service (
          token.getAuthInfoString(), sVector);

        // We print the service key of the service we just added
        sdVector = serviceDetail.getBusinessServiceVector();
        System.out.println ("    Name:"
          + ((BusinessService) sdVector.elementAt (0)).getDefaultName());
        System.out.println ("     Service key:"
          + ((BusinessService) sdVector.elementAt (0)).getServiceKey());

        System.out.println ("    ----- Done Adding -----");
        // Find jws
        System.out.println ("    searching for jws and its services...");
        Vector names = new Vector (1);
        names.add (new Name (bizName));
        BusinessList bizList = proxy.find_business (names, null, null, null,
          null, null, 0);
        Vector bizInfoVector =
          bizList.getBusinessInfos().getBusinessInfoVector();

        System.out.println ("Found" + bizInfoVector.size()
          + "entry (ies):");

        // List the businesses
        for (int ndx = 0; ndx < bizInfoVector.size(); ndx++) {
          BusinessInfo businessInfo =
            (BusinessInfo) bizInfoVector.elementAt (ndx);
          System.out.println ("     entry[" + ndx + "]: "
            +businessInfo.getNameString()+" ("
            +businessInfo.getDefaultDescriptionString()
            +"), \n       business key is:"
            +businessInfo.getBusinessKey());

          // For each business, we list the services
          ServiceInfos serviceInfos = businessInfo.getServiceInfos();
          Vector       siv          = serviceInfos.getServiceInfoVector();

          System.out.println ("    There is (are)"
            +siv.size()+"service (s) registered for"
            +businessInfo.getNameString());
```

```
            for (int svcNdx = 0; svcNdx < siv.size(); svcNdx++) {
              ServiceInfo serviceInfo = (ServiceInfo) siv.elementAt (svcNdx);
              System.out.println ("      "
                +serviceInfo.getNameString()+", service key is:"
                +serviceInfo.getServiceKey());
            }
          }
        }catch (UDDIException uddiException) {
          DispositionReport report = uddiException.getDispositionReport();
    // Remainder of code is similar to RegisterBusiness
```

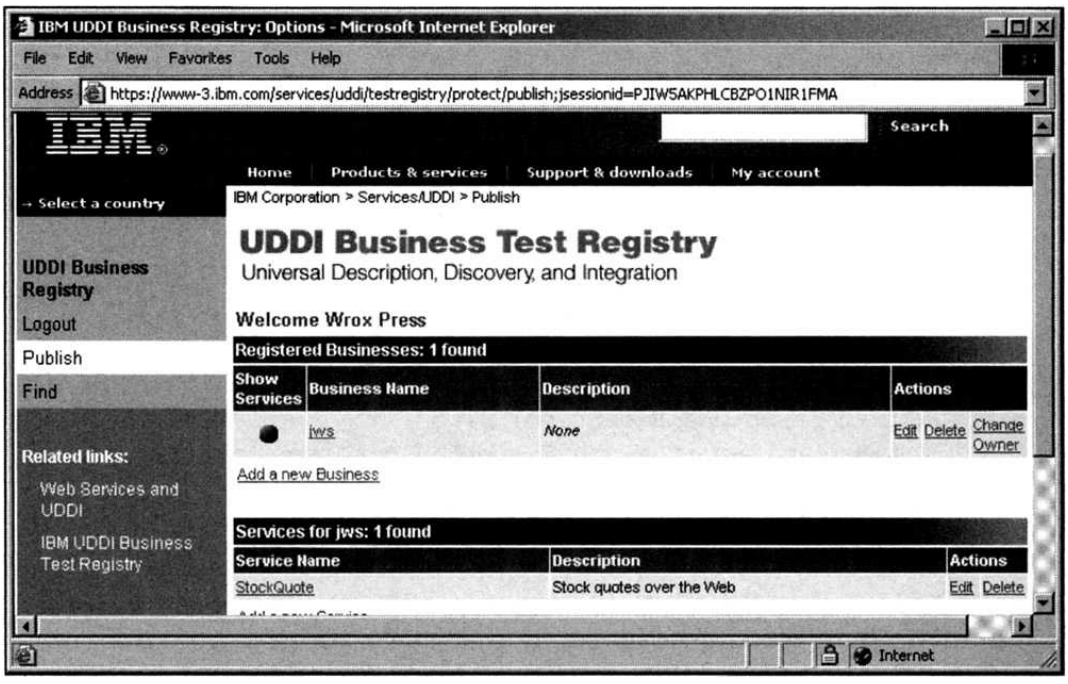
2. Let's now build and test the `RegisterService` class. Building and testing the `RegisterService` example is similar to what we did for `RegisterBusiness` (using the same classpath and assuming that our current directory is `src`):

```
javac -d ..\classes RegisterService.java

java com.wrox.jws.stockquote.RegisterService
```

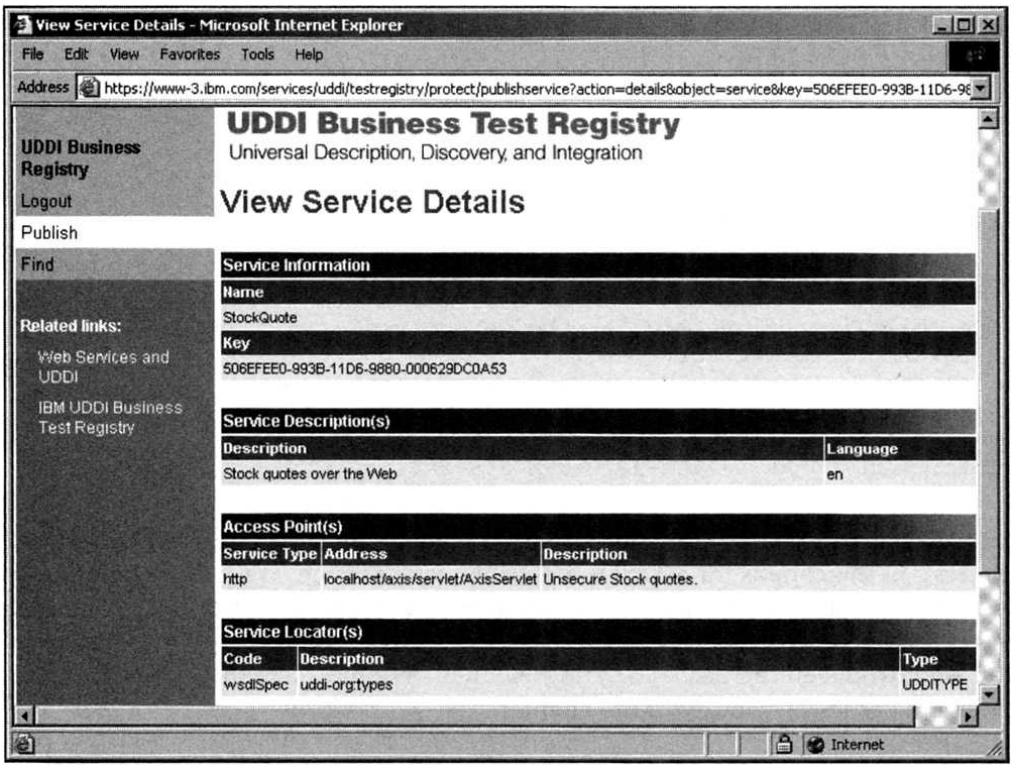
   After running this command, we will see:



3. We can double check that our service got added to the UDDI registry by visiting the web page as shown in the following screenshot:



4. If we drill down into the details of the `StockQuote` service (this can be done by clicking on the `StockQuote` service link), we will see the additional information that we have provided (access point, WSDL-described web service, and so on):



Let's now examine the code of the `RegisterService` class.

How It Works

The `RegisterService` example registers the `StockQuote` web service, but can easily be modified to register any web service. Similar to `RegisterBusiness`, we also query the UDDI registry for the service that we publish. We have mentioned previously that in UDDI, one organizes web services as part of a business, so we must obtain a business key for our business prior to registering one or more web services.

One possibility would be to cut and paste the code around the `find_business()` call that we reviewed in the previous example and take the business key out of the business info structure. For the sake of simplicity, we will hardcode the business key that we obtained from the previous registration – 368D90C0-A5D6-11D6-A687-000629DC0A7B.

When you are ready to run this example on your machine, be sure to replace the value above with the actual business key obtained while registering the business (the `RegisterBusiness` example above echoes the business key when it runs).

Let's now look the code of `RegisterService` in detail:

```
package com.wrox.jws.stockquote;

import org.uddi4j.UDDIException;
// Other import omitted from this listing
import java.util. Vector;
```

We will describe the new classes that we import from the UDDI4J package as we encounter them in the code. Again, we need to use a vector for most data types since the API is designed to minimize network roundtrips:

```
public class RegisterService {

  private static final String businessKey =
    "2BA70F90-993A-11D6-9880-000629DC0A53";
  private static final String bizName = "jws";

  // The following values are for the category of
  // WSDL-described web services
  private static final String keyName = "uddi-org:types";
  private static final String keyValue = "wsdlSpec";
  private static final String tModelKey =
    "uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4";
```

The business has been registered previously under the name `jws` with the business key listed above. The `tModelKey` in the code is for a

WSDL-described web service. We will register the service using this `tModel` and we will add it to the service category bag to put `StockQuote` in the taxonomy of web services described by WSDL. The key name and value constitute the keyed reference that we described earlier when we introduced the `categoryBag` element. We have already explained the relationship between these data types previously in the chapter.

The beginning of the `main()` method is mostly borrowed from the `RegisterBusiness` example (enabling HTTPS, setting up the transport, and so on). Things start to get different when we initialize the data structures for a service:

```
// Beginning of the main() method omitted from this listing
// Get an authorization token from the UDDI registry
AuthToken token = proxy.get_authToken(username, password);

// We create the StockQuote service
String serviceName = "StockQuote";
BindingTemplates bindingTemplates = new BindingTemplates();
BindingTemplate bindingTemplate = new BindingTemplate();
Vector bndVector = new Vector();
Vector tVector = new Vector();

// We need a tModel reference to a web service described with WSDL
TModelInstanceDetails tModelInstanceDetails =
    new TModelInstanceDetails();
TModelInstanceInfo tModelInstanceInfo = new TModelInstanceInfo();
tModelInstanceInfo.setTModelkey (tModelKey);
tVector.addElement(tModelInstanceInfo);
tModelInstanceDetails.setTModelInstanceInfoVector (tVector);
```

As we can see in the code, we use the WSDL `tModel` to define the type of the service. It is an acceptable solution since we expect the users of `StockQuote` to rely solely on WSDL to discover the service. An alternative is to define our own business service type (that is, our own `tModel`) and use it to register the web service. For specifics on how to create.your own `tModel`, please refer to the UDDI4J documentation.

In the next section of code, we define an access point and we initialize the binding template that will hold the data structures for the registration of `StockQuote`. The access point is simply the URL that customers can use to access our service. In this case, we simply register an HTTP access point that we qualify with the `unsecure` label since the data will travel unencrypted. We use the default port 80, but we can specify an alternative port such as 8081, `localhost:8081/axis/servlet/AxisServlet`:

```
// The access point is through http
AccessPoint accessPoint = new AccessPoint (
  "localhost/axis/servlet/AxisServlet", "http");

bindingTemplate.setTModelInstanceDetails (tModelInstanceDetails);
bindingTemplate.setAccessPoint (accessPoint);
bindingTemplate.setBindingKey ("");
bindingTemplate.setDefaultDescriptionString("Unsecure Stock quotes.");
bndVector.addElement (bindingTemplate);
bindingTemplates.setBindingTemplateVector (bndVector);
```

We now have enough information to create a business service structure. However, we still need to add a `categoryBag` to this service to include it in the taxonomy of WSDL-described web services.

The argument of the `BusinessService` constructor is the operator-assigned key for the business service. By passing an empty key, we tell the UDDI operator that we want a new service to be created. We can modify an existing service by passing a valid business service key:

```
BusinessService service = new BusinessService ("");
Vector sVector = new Vector(); // services
Vector sdVector = new Vector(); // service details

// We add the binding templates to the business service
service.setBindingTemplates (bindingTemplates);

// We set the default service name (in English)
service.setDefaultNameString("StockQuote", "en");

// Create a category bag with the WSDL tModel
CategoryBag categoryBag = new CategoryBag();
KeyedReference keyedReference = new KeyedReference(keyName,
  keyValue, tModelKey);
categoryBag.add(keyedReference);

// Save the service and its description in the UDDI registry
service.setDefaultDescriptionString("Stock quotes over the web");
```

```
service.setBusinessKey(businesskey);
service.setCategoryBag(categoryBag);
sVector.addElement(service);
```

If we look at the previous lines of code, we will see that we create a keyed reference to identify the `categoryBag` for the taxonomy. We then add that category bag to the service. Notice the initialization of the mandatory business key. The default description is optional.

Our service is now complete and ready for registration. Similar to `RegisterBusiness`, we use the `UDDIProxy` class to save the service. The returned value is a service detail class that contains the key generated by the UDDI operator among other values, as we can see in the following print statements:

```
ServiceDetail serviceDetail = proxy.save_service (
  token.getAuthInfoString(), sVector);

// We print the service key of the service we just added
sdVector = serviceDetail.getBusinessServiceVector();
System.out.println("     Name : "
  + ((BusinessService)sdVector.elementAt (0)).getDeafaultName());
System.out.println("     Service key : "
  + ((BusinessService)sdVector.elementAt(0)).getServicekey());
```

If we were not verifying the UDDI registration then this example would be complete.

Querying the UDDI registry for services is similar to `RegisterBusiness`; we run the query through the `proxy` object. In the following code snippet we query all the services associated to our business, but we can also directly query for a service via the `UDDIProxy.find_service()` method:

```
// Find jws
System.out.println("   Searching for jws and its services...");
Vector names = new Vector(1);
names.add(new Name(bizName));

BusinessList bizList = proxy.find_business(names, null, null, null,
  null, null, 0);
```

The `UDDIProxy.find_service()` method provides more sophisticated query capabilities than a simple name-based lookup. For instance, we can run a query that is based on taxonomies (category bags). In addition to WSDL-described services, we can query for standard taxonomies like the Universal Standard Products and Services Classification (UNSPSC), which is a hierarchical classification of products and services. For more information, we can go to the UNSPSC home page http://eccma.org/unspsc/.

For more specifics on UDDI query, check the UDDI4J documentation. But for now, let's go back to our example. As we just saw in the code, the list of businesses that match our criteria is assigned to `bizList`. We then use `bizList` to get a vector of `BusinessInfo` objects:

```
Vector bizInfoVector =
  bizList.getBusinessInfos().getBusinessInfoVector();
System.out.println(" Found " + bizInfoVector.size()
  + " entry(ies):");
```

Finally, we loop through the results to display the businesses and services that were returned by UDDI:

```
// List the businesses
for(int ndx = 0; ndx < bizInfoVector.size(); ndx++) {

  BusinessInfo businessInfo =
    (BusinessInfo)bizInfoVector.elementAt(ndx);
  System.out.println("     entry[" + ndx + "]: "
  + businessInfo.getNameString() + " ("
  + businessInfo.getDefaultDescriptionString()
  + "), \n       business key is: "
  + businessInfo.getBusinessKey());

// For each business, we list the services
ServiceInfos serviceInfos = businessInfo.getServiceInfos();
Vector siv = serviceInfos.getServiceInfoVector();

System.out.println("     There is (are) "
  + siv.size() + " service(s) registered for "
  + businessInfo.getNameString());

for(int svcNdx = 0; svcNdx < siv.size(); svcNdx++) {
  ServiceInfo serviceInfo = (ServiceInfo) siv.elementAt (svcNdx);
  System.out.println(" "
```

```
                + serviceInfo.getNameString() + ", service key is: "
                + serviceInfo.getServiceKey());
            }
          }
        } catch (UDDIException uddiException) {
```

The remainder of the code is identical to `RegisterBusiness` so we won't explain it again.

We have only scratched the surface of what can be done with UDDI and the UDDI4J package, but the examples that we reviewed provide us with a solid base for further learning.

Now that we have some practical experience with UDDI, it is time to take a step back and see what the technology can accomplish for us. If we recall the advertising problem that we described at the beginning of this chapter and compare it to what we have discussed about UDDI, we will see that, by and large, UDDI solved the issues that we identified, namely the following:

- **Universal**
  The UDDI cloud is accessible through HTTP and SOAP, which are available on most platforms in use today, and are open interoperable specifications.
- **Description**
  The UDDI registry provides both local (tModels) and remote descriptions (WSDL documents) of web services.
- **Discovery**
  The UDDI cloud provides a query mechanism by name and by taxonomies.
- **Integration**
  The web services registered in UDDI use standard protocols like HTTP and SOAP.

So, is it true to say that everything is rosy in web service land? Not quite. As we will discuss in the following section, UDDI will have to overcome some flaws if it is to become the first choice when it comes to publishing and advertising web services.

Other Publishing Technologies

According to a study conducted by SalCentral and WebServicesArchitect, close to 50% of the data stored in the production UDDI cloud is inaccurate. This is not surprising if we consider the fact that UDDI registration is not moderated; in other words anybody can put anything in the UDDI cloud. The details of the SalCentral/WebServicesArchitect research can be found at http://www.salcentral.com/uddi/default.asp.

To be convinced that a moderator is important, let's ask ourselves the following question – how accurate would the phone book be, if it were up to all the businesses and individuals to enter and keep their data up-to-date? Another analogy worth mentioning is the HTML publishing surge that we saw a few years ago – a lot of people wanted to publish their pages over the web, but very few wanted to maintain the accuracy of the data that they published. Even inside an enterprise, too often we see out-of-date departmental web sites.

Another shortcoming of UDDI is the lack of restricted communities; there is no equivalent in the UDDI cloud to our intranets and extranets.

In this section, we will take a short look at two potential solutions to these shortcomings. We will first look at setting up a private UDDI registry and then we will look at a specification that complements UDDI by supporting local publication of web services – the **Web Service Inspection Language (WSIL).**

Let's start with a discussion on setting up a private UDDI registry.

Private UDDI Registries

Setting up a private UDDI registry answers the weakness that we mentioned earlier:

- A private UDDI registry can easily be moderated and therefore limit the inaccuracies that inevitably creep into the database.
- A private UDDI registry, by definition, can be restricted to an intranet or an extranet.

However, these features come at a price – as its name implies, a private UDDI registry is not part of the cloud and can therefore not be used to gain new customers. It is only a valid medium to publish web services to our partners and co-workers. Another way to look at a private UDDI registry is to think of it as a corporate phonebook.

First, it is destined to be used by a specific, well-targeted audience and as such can contain proprietary information. Second, because the users usually know what they are looking for, a sophisticated taxonomy is not always required; only the largest companies have a phonebook with Yellow Pages.

One could argue that using UDDI is not required when the information is only going to be published to an intranet or an extranet. There is at least one good reason why the extra complexity and expense of a private UDDI registry over a homegrown solution (for example, a SQL or an LDAP database) are warranted. The reason is that UDDI is a standardized protocol, so if we use UDDI both internally and externally, we will be able to leverage our development for our private registry when we do go to the UDDI cloud.

When it comes to setting up a private UDDI registry, there are several solutions to choose from. One that is relatively easy to setup is the

lightweight UDDI server that comes with the WebSphere Software Development Kit for Web Services (WSDK).

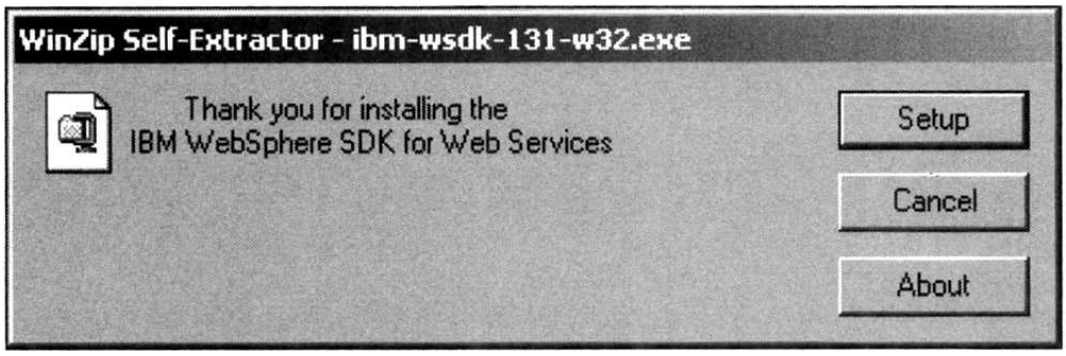
Try It Out: Setting up a Private UDDI Registry

Our first step in setting up a private registry is to download the WSDK.

1. We can download WSDK from http://www-106.ibm.com/developerworks/webservices/wsdk/ (follow the Download link on the right):

The download and setup process of WSDK will be reviewed in details in Chapter 10. For now, we are simply interested in getting a private UDDI registry up and running, so we will focus on that portion of WSDK.

After answering the obligatory registration questions, we will be prompted to download an `.exe` file for the Windows platform or a `.tar` file for Linux platform. At the time of this writing, the `.exe` for Windows is `ibm-wsdk-131-w32.exe`:

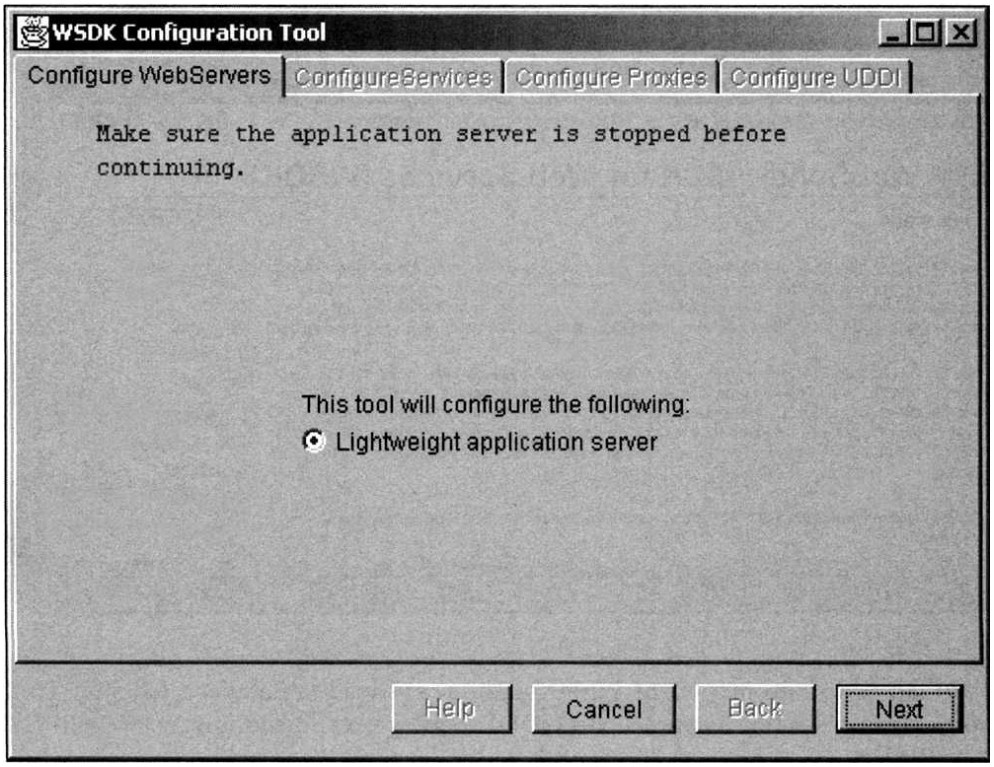


Click on Setup, enter the installation directory, (we will be installing it to `C:\WSDK`). We will assume that the installation directory is the `wsdk_home` shell variable for the purpose of this discussion. Note that we will need to restart our machine after the setup is finished.

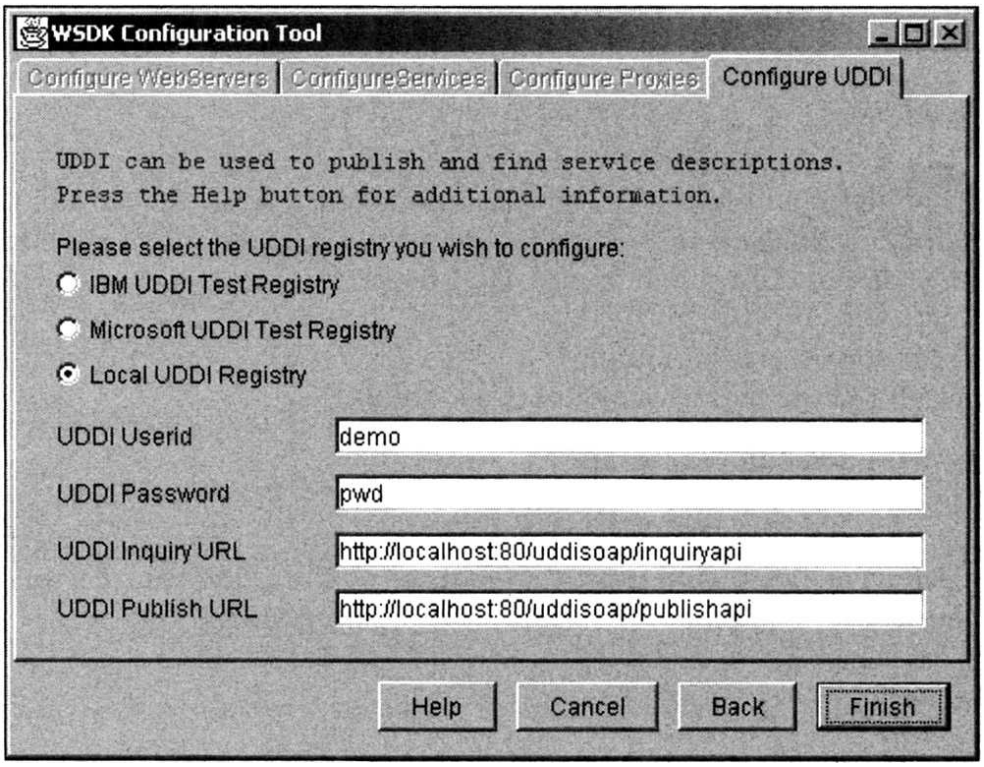
2. To enable the WebSphere UDDI implementation, we must run the WSDK configuration utility by typing the following command (or its equivalent for our installation):

```
%wsdk_home%\bin\wsdkconfig.bat
```

3. When the GUI comes up, select the default (Lightweight application server) as shown on the following screenshot:



Then click on the Next button until we get to the UDDI configuration tab:

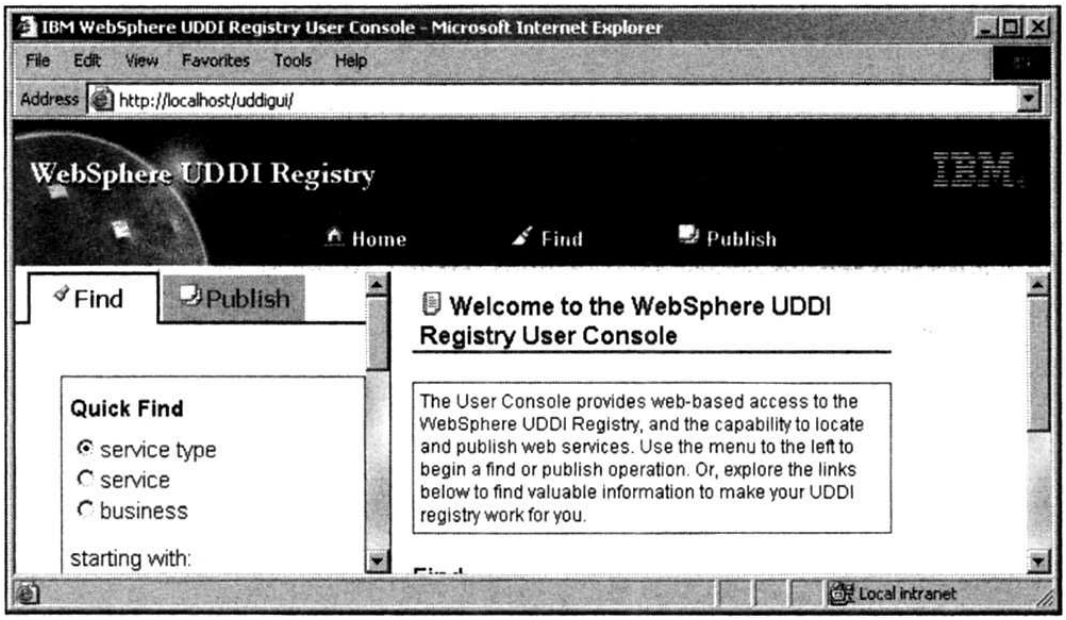


Once again, the defaults should suit us well. Click on Finish to set up the lightweight UDDI server (the process might take a few minutes).

4. To start the UDDI server, simply type the following command (or its equivalent for our installation) in a command (shell) window:
   `%wsdk_home%\WebSphere\bin\startServer`

   There is also a `stopServer` command to stop the UDDI server.

5. Once the server is running, navigate our browser to the following URL `http://localhost/uddigui`. Note that we are using the default port: 80. If everything goes as expected, we should see the following screen:



Note The WSDK UDDI registry comes with a default login. The user name is `demo`, and the password is `pwd`.

How It Works

This UDDI GUI is simpler than the IBM public site and is functional. We can run the `RegisterBusiness` and `RegisterService` examples against the following URLs (or their equivalent for our installation) to test our local UDDI registry:

- `http://localhost/uddisoap/inquiryapi`
  The inquiry API. Note that we can also use `inquiryAPI`.
- `http://localhost/uddisoap/publishapi`
  The publishing API. Note that we can also use `publishAPI`.

We will have to register the `jws` business prior to registering the `StockQuote` service, as we did for the public UDDI registry. The manual registration is straightforward:

1. Click on the `Publish` tab as shown in the previous screenshot
2. Enter the business name (`jws`)
3. Finally, click on the `Publish` now link

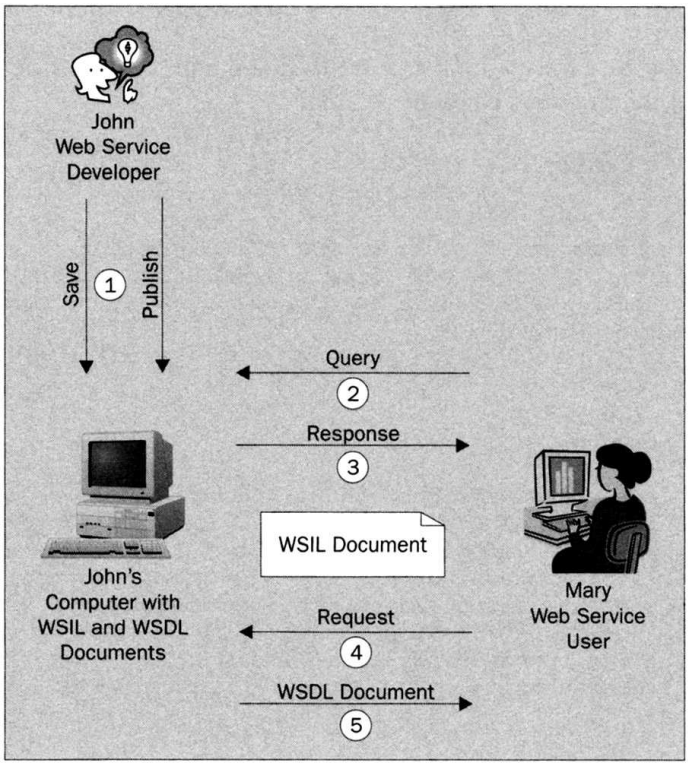
There is also an Advanced publish link that allows us to enter other information like an address, a contact, and so on.

We said earlier that a significant drawback of a private UDDI registry is that it does not participate in the UDDI cloud. There is a solution that bridges the public UDDI cloud and a private registry. This solution also comes with a price – a new XML dialect: the WSIL.

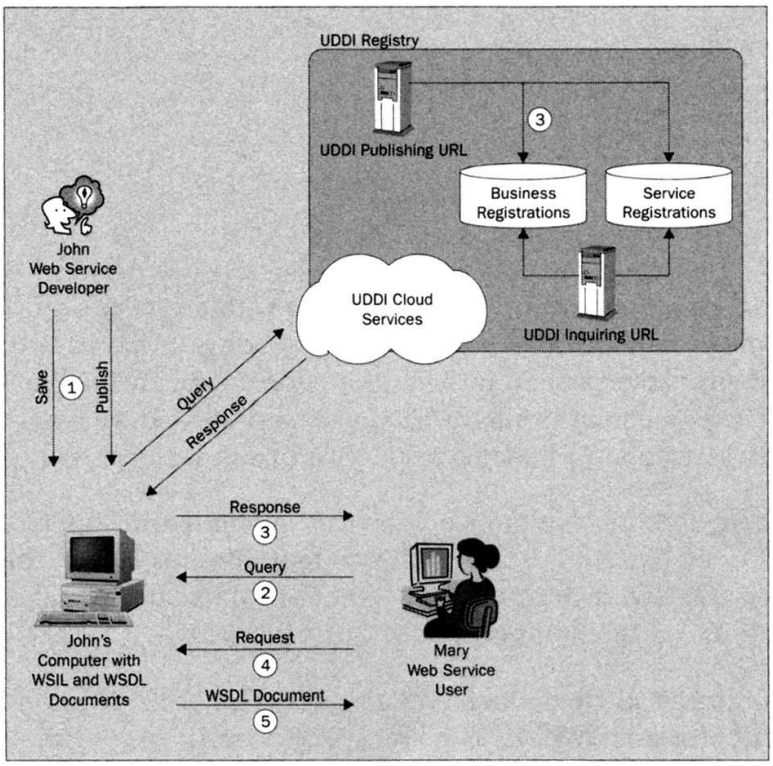
WS-Inspection

The motivation behind the **WS-Inspection** specification is simple – allow web service users to query a web server rather than a central registry. The WS-Inspection specification is a proprietary specification jointly proposed by IBM and Microsoft. At the time of this writing, the specification is still in the hands of both companies, but they have stated their intention of submitting the specification to a standards body. The text of the specification can be found at http://www-106.ibm.com/developerworks/webservices/library/ ws-wsilspec.html.

Going back to John and Mary, the following diagram illustrates the interactions between the web service publisher and user when they rely on WS-Inspection:



This diagram is slightly different from the first one we looked at in this chapter; here Mary queries John's web site for the WSIL document that contains the list of web services that John supports. This approach is actually complementary with UDDI, because nothing forbids John to publish in a WSIL document web services that are also available from the UDDI cloud as shown on the following figure:



In this case, John's computer is simply a gateway to a UDDI registry (public or private).

Without any more delays, let's have a look at a WSIL document for the `StockQuote` and `Market` web services that we used in the previous chapter:

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

```
<inspection
  xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  xmlns:wsilwsdl="http://schemas.xmlsoap.org/ws/2001/10/inspection/wsdl/"
  xmlns:wsiluddi="http://schemas.xmlsoap.org/ws/2001/10/inspection/uddi/"
  xmlns:uddi="urn:uddi-org:api">

  <service>
    <name xml:lang="en-US">
      StockQuoteService
    </name>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl"
      location="http://localhost/axis/services/StockQuote?wsdl">
      <wsilwsdl:reference endpointPresent="true">
        <wsilwsdl:implementedBinding
          xmlns:interface="http://stockquote.jws.wrox.com">
            interface:StockQuoteSoapBinding
        </wsilwsdl:implementedBinding>
      </wsilwsdl:reference>
    </description>
  </service>

  <service>
    <name xml:lang="en-US">MarketService</name>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl"
      location="http://localhost/axis/services/Market?wsdl">
      <wsilwsdl:reference endpointPresent="true">
        <wsilwsdl:implementedBinding
          xmlns:interface="http://stockquote.jws.wrox.com">
          interface:MarketSoapBinding
        </wsilwsdl:implementedBinding>
      </wsilwsdl:reference>
    </description>
  </service>

</inspection>
```

As we can see, the WSIL document is an XML document with an `<inspection>` root element. The namespace of the specification is http://schemas.xmlsoap.org/ws/2001/10/inspection, which we use as the default namespace in our example document. Inside the `<inspection>` element, there is a set of `<service>` elements, one for each web service that we want to publish. One can also have `<businessDescription/>` elements to publish businesses (we will see an example shortly).

In the example above, the definition of the web service is included in situ, but it is also possible to have a link to another WSIL document or a UDDI registry (more on this later). The `<name>` element is the name of the web service. The `<description>` element contains the actual description of the web service along with its location (for example, `StockQuote.wsdl`).

The `referencedNamespace` attribute is the namespace of the referenced document. Since in this case the web service is described using WSDL, its name space is http://schemas.xmlsoap.org/wsdl.

You will notice that the `<reference/>` element is not part of the default namespace; rather it is part of http://schemas.xmlsoap.org/ws/2001/10/inspection/wsdl, prefixed by `wsilwsdl` in this document. This methodology of supporting WSDL in WSIL through extensions is similar to what we saw with the support for SOAP inside WSDL documents.

This allows the specification to accommodate new description formalisms by simply adding a namespace rather than modifying the specification. In our example, the `<wsilwsdl:implementedBinding/>` element indicates which WSDL bindings are defined in the referenced WSDL document. The `<wsilwsdl:implementedBinding/>` element is optional.

You will also notice that our WSIL document declares the namespace for web services published in UDDI http://schemas.xmlsoap.org/ws/2001/10/inspection/uddi with the prefix `wsiluddi`. As a matter of fact, it is fairly easy to modify our WSIL document to reference a UDDI entry. Notice the use of the `<link>` element and the fact that we publish a business rather than a service.

```
<?xml version="1.0"?>

<inspection
  xmlns:wsiluddi="http://schemas.xmlsoap.org/ws/2001/10/inspection/uddi/"
  xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
  <link referencedNamespace="urn:uddi-org:api">
    <wsiluddi:businessDescription
      location="http://www.ibm.com/uddi/inquiryapi">
      <wsiluddi:businessKey>
        368D90C0-A5D6-11D6-A687-000629DC0A7B
```

```
          </wsiluddi:businessKey>
          <wsiluddi:discoveryURL useType="businessEntity">
            <http://www-3.ibm.com/services/uddi/testregistry/uddiget?
              businessKey=368D90C0-A5D6-11D6-A687-000629DC0A7B
          </wsiluddi:discoveryURL>
        </wsiluddi:businessDescription>
      </link>
    </inspection>
```

The UDDI-specific elements are:

- `<wsiluddi:businessDescription/>` element
  This contains a URL to the description of the business
- `<wsiluddi:businessKey/>` element
  This contains the UDDI business key
- `<wsiluddi:discoveryURL/>` element
  This contains the discovery URL along with the necessary argument (a UDDI business key)

Let's wrap up our review of the WSIL document above by saying that the `endPointPresent="true"` attribute simply states that the WSDL contains a WSDL `<service/>` element. An interface definition WSDL document would not contain a `<service/>` element, and as such would have `endPointPresent="false"` attribute.

Publishing the web services available on a given web site using WSIL is not enough because potential users must have a standard URL to go to. The same problem existed with WSDL and was solved by adding the `?WSDL` trailer to the web service URL as we saw in Chapter 5 (for example, http://localhost/axis/StockQuote?WSDL).

Here the convention must be web site-wide since the purpose of WSIL is to publish all the web services available on a given web site. The well-known WSIL URL is `inspection.wsil`. For instance, if our web services are at http://www.wroxpress.com/axis, then our WS-Inspection document should be at http://www.wroxpress.com/axis/inspection.wsil.

There are at least three ways to make the `inspection.wsil` URL available for all to browse:

- Hardcode a WSIL document
- Generate a WSIL document with the DOM
- Use the WSIL4J package available with the WSTK

Note At the time of this writing, WSIL4J (WSTK 3.2) does not automatically recognize the web services published by Axis, but according to the documentation, that feature is planned for a future release.

Solution 1 and 2 are straightforward and do not involve concepts specific to web services. Solution 3 is worth investigating since it has the potential of simplifying the creation and processing of WSIL documents.

The WSIL4J (`%wstk_home%\wsil4j`) package comes with two examples – one to read a local WSIL document and one to read a remote WSIL document. Alas, there is no example that creates a WSIL document, so we will go through one.

Try It Out: Generate a WSIL Document

The `Inspection` example provides an implementation of the `inspection.wsil` URL that produces the WSIL document that we just reviewed. The `Inspection` class is a servlet. If you are not familiar with servlet development do not worry; we will be discussing all the steps necessary to get the `Inspection` class working.

1. The code of the `Inspection` class is listed below:

```
package com.wrox.jws.stockquote;

import org.apache.wsil.WSILDocumentFactory;
import org.apache.wsil.WSILDocument;
import org.apache.wsil.WSILException;
import org.apache.wsil.Service;
import org.apache.wsil.ServiceName;
import org.apache.wsil.Description;
import org.apache.wsil.QName;

import org.apache.wsil.extension.wsdl.Reference;
import org.apache.wsil.extension.wsdl.ImplementedBinding;

import org.apache.wsil.impl.ServiceNameImpl;
import org.apache.wsil.impl.DescriptionImpl;
import org.apache.wsil.impl.extension.wsdl.WSDLExtensionBuilder;
```

```
import org.apache.wsil.impl.extension.wsdl.ReferenceImpl;
import org.apache.wsil.impl.extension.wsdl.ImplementedBindingImpl;
import org.apache.wsil.xml.XMLWriter;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Inspection extends HttpServlet {

  // Minimal doGet implementation.
  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    WSILDocument document = createDocument();
    XMLWriter writer = new XMLWriter();

    try {
      writer.writeDocument (document, res.getWriter());
      res.setContentType ("text/xml");
    } catch (WSILException wsilException) {
      throw new ServletException ("Cannot generate WSIL dcoument: "
        + wsilException);
    }
} // doGet

// Comments omitted from this listing
private void addService (WSILDocument document, String name,
  String location, String bindingUriNamespace, String bindingName) {

  Service stockQuoteService = document.createService();
  ServiceName stockQuoteServiceName = new ServiceNameImpl();
  Description stockQuoteDescription = new DescriptionImpl();

  // We add a <name/> element for the English language
  stockQuoteServiceName.setText (name);
  stockQuoteServiceName.setLang ("en-US");
  stockQuoteService.addServiceName (stockQuoteServiceName);

  // We add a <description/> element (location + referenced namespace)
  stockQuoteDescription.setLocation(location);
  stockQuoteDescription.setReferencedNamespace(
    "http://schemas.xmlsoap.org/wsdl");

  //We add a reference element to a SOAP binding:
  Reference refBinding = new ReferenceImpl();
  ImplementedBinding stockQuoteBinding = new ImplementedBindingImpl();

  refBinding.setEndopointPresent (Boolean.TRUE);
  stockQuoteBinding.setBindingName(
    new QName (bindingUriNamespace, bindingName));
  refBinding.addImplementedBinding (stockQuoteBinding);

  stockQuoteDescription.setExtensionElement(refBinding);
  stockQuoteService.addDescription(stockQuoteDescription);
  document.getInspection().addService(stockQuoteService);
}
// Create a WSIL document with two services: StockQuote and Market
private WSILDocument createDocument() throws WSILException {
  String methodName =
    "com.wrox.jws.stockquote.Inspection.createDocument";
  WSILDocument document = null;
```

```
    document = WSILDocumentFactory.newInstance().newDocument();
    addService(document, "StockQuoteService",
      "http://localhost/axis/services/StockQuote?wsdl",
      "http://stockquote.jws.wrox.com", "StockQuoteSoapBinding");

    addService(document, "MarketService",
      "http://localhost/axis/services/Market?wsdl",
      "http://stockquote.jws.wrox.com", "MarketSoapBinding");

    return document;
  }
```

The remainder of the code contains the `main()` method that is used as a test case.

2. Building the `Inspection` example requires that we include the following JAR files in our classpath:

   - `%jwsDirectory%\lib\xmlParserAPIs.jar`
     The Xerces parser is required since WSIL is an XML document
   - `%wstk_home%\uddi4j\lib\uddi4j.jar`
     `wstk_home` is where we installed the Web Service Toolkit
   - `%wstk_home%\wsil4j\lib\wsil4j.jar`
   - `%catalina_home%\common\lib\servlet.jar`
     `catalina_home` is where we installed Tomcat (Catalina)

3. Once you have built the class file (`Inspection.class`), make sure that you put it under `%axisDirectory%\webapps\axis\WEB-INF\classes`, where `axisDirectory` is your Axis installation directory. For instance, on Windows, type the following commands, assuming that the `src` directory is our current directory (it is best to stop Tomcat before building):

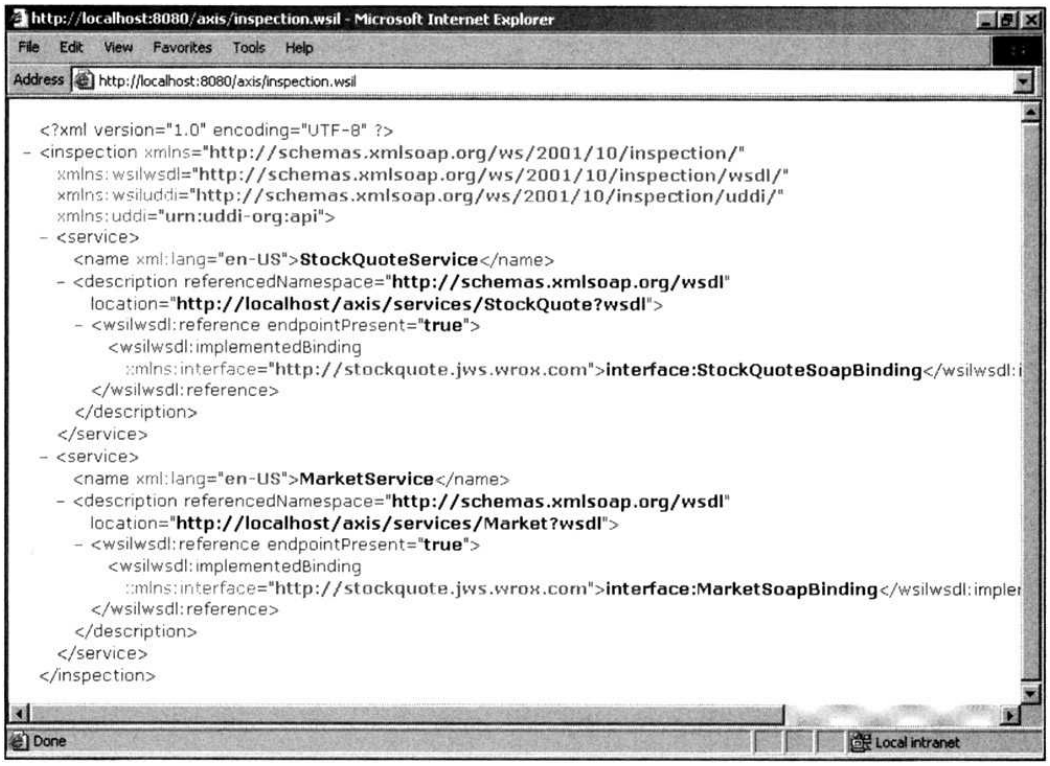
   ```
   javac -d %axisDirectory%\webapps\axis\WEB-INF\classes
    com\wrox\jws\stockquote\Inspection.java
   ```

   Important The order of the `<servlet/>` and `<servlet-mapping/>` elements is constrained by the DTD. So make sure that all the servlet definitions come before the servlet mappings in `web.xml`. If `web.xml` does not follow the constraints of its DTD we will get a validation error when Tomcat comes up.

4. Before being able to test our example, we need to define it as a servlet. To include `inspection.wsil` as a servlet, make sure that you add the following lines to the `web.xml` of your Axis installation (`%axisDirectory%\webapps\axis\WEB-INF\web.xml`):

   ```
   ...
   <servlet>
      <servlet-name>inspection</servlet-name>
      <servlet-class>com.wrox.jws.stockquote.Inspection</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>inspection</servlet-name>
      <url-pattern>inspection.wsil</url-pattern>
   </servlet-mapping>
   ...
   ```

5. Note that you might want to make a backup copy of the `web.xml` file before modifying it. Next, we will test the `Inspection` class. After starting Tomcat, you should be able to navigate to the `inspection.wsil` URL and see the following result:



Before proceeding with a review of the code, it is necessary to point out a few facts.

The WSIL4J package models the structure of a WSIL document by exposing a hierarchy with a model that is similar to the DOM API. For instance, if we look at the `<service/>` element, it is modeled with a `org.apache.wsil.Service` object and always contained inside a `org.apache.wsil.Inspection` element, which itself is part of a `org.apache.wsil.WSILDocument`. One creates a service with a call `WSILDocument.createService()`.

How it Works

Let's now have a closer look at the `Inspection` class:

```
package com.wrox.jws.stockquote;

import org.apache.wsil.WSILDocument Factory;
// Other imports removed from this listing
```

We will review the `org.apache.wsil` classes as we encounter them in the code. The `java.io` and `java.net` packages are simply used for the test code that reads the WSIL document from the `inspection.wsil` URL:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
```

The `javax.servlet` classes are used to implement a minimalist version of `doGet()`:

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Inspection extends HttpServlet {
  // Minimal doGet implementation.
  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    WSILDocument document = createDocument();
    XMLWriter    writer = new XMLWriter();

    try {
      writer.writeDocument (document, res.getWriter());
      res.setContentType ("text/xml");
```

```
        } catch (WSILException wsilException) {
          throw new ServletException ("Cannot generate WSIL document: "
            + wsilException);
        }
      } // doGet
```

As you can see above, the implementation of `doGet()` consists of the creation of the **WSIL** document followed by the writing of the document to the output of the servlet. The `XMLWriter` class that comes with the WSIL4J package is handy to write any WSIL document to a `java.io.OutputStream` or to a `java.io.Writer`.

The next method in the `Inspection` servlet is the `addService()` method, which adds a service to the `<inspection/>` element of the document passed as argument. The goal of `addService()` is to produce a document fragment like the following in the case of `StockQuote`:

```
<service>
 <name xml: lang="en-US">StockQuoteService</name>
 <description

    referencedNamespace="http://schemas.xmlsoap.org/wsdl"
    location="http://localhost/axis/services/StockQuote?wsdl">

  <wsilwsdl:reference endpointPresent="true">
   <wsilwsdl:implementedBinding
    xmlns:interface="http://stockquote.jws.wrox.com">
    interface:StockQuoteSoapBinding
   </wsilwsdl:implementedBinding>
  </wsilwsdl:reference>
 </description>
</service>
```

Here is the code for `addService()`:

```
private void addService (WSILDocument document, String name,
  String location, String bindingUriNamespace, String bindingName) {

  Service stockQuoteService = document.createService();
  ServiceName stockQuoteServiceName = new ServiceNameImpl();
  Description stockQuoteDescription = new DescriptionImpl();
```

Notice that we need the document to create a service, but that the elements inside the service can be created standalone and must be added through `add*` methods, as we can see below for the service name:

```
        // we add a <name/> element for the English language
        stockQuoteServiceName.setText (name);
        stockQuoteServiceName.setLang ("en-US")
        stockQuoteService.addServiceName (stockQuoteServiceName);
```

The `<description/>` is a little more complex since we want to include a binding for SOAP. First, we set the location and the namespace to WSDL:

```
        // We add a <description/> element (location + referenced namespace)
        stockQuoteDescription.setLocation (location);
        stockQuoteDescription.setReferencedNamespace(
          "http://schemas.xmlsoap.org/wsdl");
```

Next, we create the reference element and the binding:

```
        // We add a reference element to a SOAP binding:
        Reference refBinding = new ReferenceImpl();
        ImplementedBinding stockQuoteBinding = new ImplementedBindingImpl();

        refBinding.setEndpointPresent (Boolean.TRUE);
        stockQuoteBinding.setBindingName(
          new QName (bindingUriNamespace, bindingName));

        refBinding.addImplementedBinding (stockQuoteBinding);

        stockQuoteDescription.setExtensionElement (refBinding);
        stockQuoteService.addDescription (stockQuoteDescription);
```

As you can see, we add the `<description/>` element to the `<service/>` element. Finally, we must not forget to add the created service to the `<inspection/>` element:

```
            document.getInspection().addService (stockQuoteService);
        }
```

Thanks to the `addService()` method, adding a `<service/>` element for `StockQuote` and `Market` is relatively trivial:

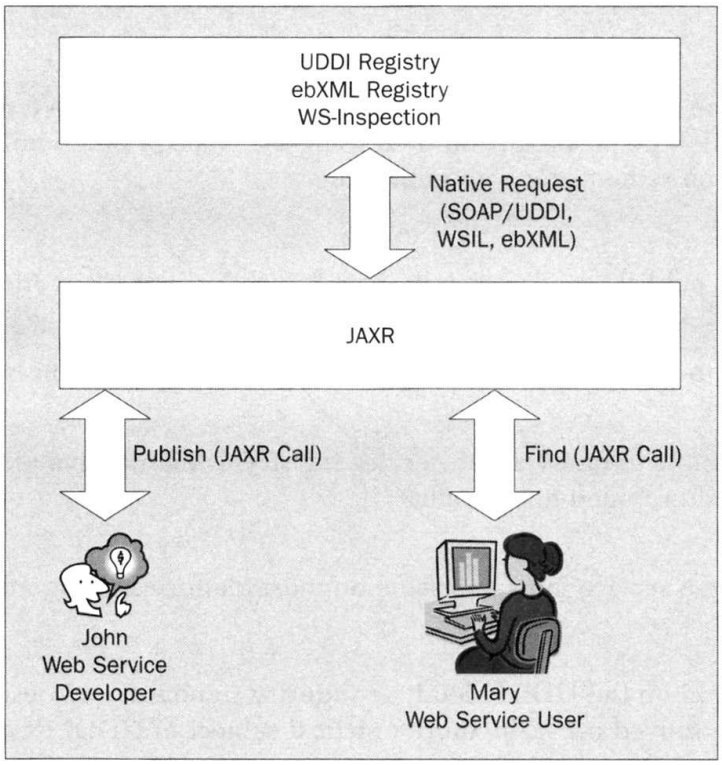
```
    // Create a WSIL document with two services: StockQuote and Market
    private WSILDocument createDocument() throws WSILException {

      String methodName = "com.wrox.jws.stockquote.Inspection.createDocument";
      WSILDocument document = null;
      document = WSILDocumentFactory.newInstance().newDocument();

      addService (document, "StockQuoteService",
        "http://localhost/axis/services/StockQuote?wsdl",
        "http://stockquote.jws.wrox.com", "StockQuoteSoapBinding");

      addService (document, "MarketService",
        "http://localhost/axis/services/Market?wsdl",
        "http://stockquote.jws.wrox.com", "MarketSoapBinding");

      return document;
  }
```

This concludes our detailed review of the `Inspection` class.

The WSIL example that we reviewed is fairly simple, but it should give us enough information to get started with our own project. We can enhance this example by adding links to other WSIL documents, or by adding links to UDDI registries (public or private).

Java for XML Registries

If there is a design that has been copied over and over, it is the two-tier design for database access first popularized by ODBC, and more recently by JDBC. The **Java API for XML Registries (JAXR)** is no exception, since it is a two-tier architecture to access data stored in XML. As you can see from the following figure, JAXR can be used in the case of John and Mary:



Note **Electronic business XML (ebXML)** is a specification sponsored by the United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) and by the Organization for the Advancement of Structured Information Standards (OASIS).

This architecture is also similar to the UDD14J package that we used in this chapter to access a UDDI registry. However, there are at least two significant differences:

- **UDDI4J was designed specifically for UDDI registries**

JAXR, on the other hand, provides an abstraction that can be used to access most XML registries.

- **UDDI4J was written by IBM and donated to open source**
  JAXR is the product of the Java Community Process (JCP). Whether or not the open process produced a better mousetrap is probably a matter of taste.

The JAXR specification can be downloaded from http://java.sun.com/xml/jaxr. At the time of this writing, the latest version of JAXR comes with the Java Web Service Developer Pack 1.0 (Java WSDP 1.0), which can be downloaded from http://java.sun.com/webservices/webservicespack.html.

Given the tendency of XML to generate new specifications, like dandelions on a spring lawn, a unifying technology like JAXR is likely to succeed in the long run. Once again, this is not unlike ODBC and JDBC that mostly isolate our database developments from the vendor-specific intricacies and peculiarities. Alas, at the time of this writing, only UDDI and ebXML registries are supported, so there is little JAXR can do to help us in the UDDI/WSIL world.

Summary

This chapter gave us the opportunity to review web service publishing. We started our discussion with a more formal definition of the problem, and then using the analogy of the ubiquitous phonebook, we identified four main requirements for web publishing:

- **Universal**
  A directory of published web services must be accessible to the widest possible audience
- **Description**
  A web service registry must provide some description of the published web services
- **Discovery**
  It must be possible to query a web service registry using intuitive searches like names, geography, and provided functionality
- **Integration**
  A published web service must be usable on most platforms, using most programming languages

We focused our attention on the UDDI cloud, an industry standard, and described why it satisfies our requirements. We then moved on to the more practical subject of UDDI development using the UDDI4J package with two examples: `RegisterBusiness` and `RegisterService`.

We wrapped up the chapter with a review of other web service publishing technologies than the UDDI cloud. We saw that setting up a private UDDI registry allowed us to service an intranet or an extranet. We also introduced the WS-Inspection specification and its Web Service Inspection Language (WSIL) as a way to bridge the gap between a local and a global XML registry. We concluded the chapter with a short introduction to the Java API for XML Registries (JAXR), which supports a standardized API to XML registries like UDDI and ebXML, but does not yet support WSIL.

In the next chapter, we'll look at how we can we can use messaging to send and receive web services asynchronously.