# UDDI4J: Matchmaking for Web services
**Interacting with a UDDI server**

Doug Tidwell
Web Services Evangelist, IBM
January 2001

As part of its continued commitment to Web services, IBM has released UDDI4J, an open-source Java implementation of the Universal Discovery, Description, and Integration protocol (UDDI). In this article, we'll discuss the basics of UDDI, the Java API to UDDI, and how you can use this technology to start building, testing, and deploying your own Web services.

**Contents:**

The central idea behind the Web services revolution is that the Web will be populated with an assortment of small pieces of code, all of which can be published, found, and invoked across the Web. One key technology for the service-based Web is SOAP, the Simple Object Access Protocol. Based on XML, SOAP allows an application to interact with remote applications. That's all well and good, but how do we find those applications in the first place? That's where UDDI comes in.

UDDI provides three basic functions, popularly known as publish, find, and bind:

- Publish: How the provider of a Web service registers itself.
- Find: How an application finds a particular Web service.
- Bind: How an application connects to, and interacts with, a Web service after it's been found.

A UDDI registry contains three kinds of information, described in terms of telephone directories:

- White pages: Information such as the name, address, telephone number, and other contact information of a given business.
- Yellow pages: Information that categorizes businesses. This is based on existing (non-electronic) standards (see Resources).
- Green pages: Technical information about the Web services provided by a given business.

When I am working with a UDDI registry, there are four information types that are important to me:

- Business information: Contained in a `BusinessEntity` object, which in turn contains information about services, categories, contacts, URLs, and other things necessary to interact with a given business.
- Service information: Describes a group of Web services. These are contained in a `BusinessService` object.
- Binding information: The technical details necessary to invoke a Web service. This includes URLs, information about method names, argument types, and so on. The UDDI4J `BindingTemplate` object represents this data.
- Information about specifications for services: This is metadata about the various specifications implemented by a given Web service. These are called tModels in the UDDI specification; the UDDI4J `TModel` object represents this data.

In this article I will show how to write basic code for manipulating objects in a UDDI registry. This UDDI registry is available in the open-source UDDI4J package (see Resources) that you can download from the IBM alphaWorks site.

## What's in the UDDI4J package
UDDI4J contains an implementation of the client side of UDDI (everything your application needs to publish, find, and bind a Web service). It also includes the source of the code, the complete JavaDoc documentation, and three sample applications. I will go over the `UDDIProxy` class, the most important class in the package, and cover the three sample applications.

### The UDDIProxy class
This class is the main object you'll use in your UDDI applications. It has all the methods you need to connect to a UDDI

registry, execute a query, and process the results. In the sample applications I discuss here, the `UDDIProxy` object is what we use to interact with the registry. Listing 1 shows how to create such an object and point it at the registry:

**Listing 1: Code fragment for creating a UDDIProxy**

```
UDDIProxy proxy = new UDDIProxy();

proxy.setInquiryURL("http://www-3.ibm.com/services/uddi/ testregistry/inquiryapi");

proxy.setPublishURL("https://www-3.ibm.com/services/uddi/
testregistry/protect/publishapi");
```

This code creates the object, then points the proxy to the IBM UDDI registry. Notice that the protocol in the `setPublishURL` method is *https*, the UDDI standard defines that anyone should be able to query a registry, but only applications and users with the proper access can modify registry information.

**Other classes**
Other classes supplied with UDDI4J provide access to all of the information available in the UDDI registry. For example, the `BusinessList` object allows you to access some number of `BusinessInfo` objects. For a given `BusinessInfo` object, you can access the `ServiceInfo` objects that describe the services it provides. The classes in UDDI4J map to the various XML elements and APIs described in the UDDI Programmer's API Specification and the UDDI XML Structure Reference (see Resources).

**Sample application 1: Find a business in the UDDI registry**
This application, `FindBusinessExample.java`, connects to a UDDI registry and attempts to find businesses that meet a certain criteria. In this sample, we're looking for all businesses that begin with the letter "S":

```
BusinessList bl = proxy.find_business("S", null, 0);
```

All versions of the `find_business` method take three parameters: the first is the search parameter, the second is a `FindQualifiers` object (`null` in this example), and the third is the number of matches to return (`0` means return all matches). There are several different kinds of search parameters, the simplest of which is a search string. More advanced search parameters include collections of business identifiers, business categories, URLs, and `tModels`. The method returns a `BusinessList` object, a collection that can be used to find specific information about all the businesses that match the search parameter.

Once we have the list of businesses that match our search criteria, this code iterates through the list and prints the names of all the businesses that match:

**Listing 2: Code fragment for printing a list of matching business objects**

```
Vector businessInfoVector  = bl.getBusinessInfos().getBusinessInfoVector();

for (int i = 0; i < businessInfoVector.size(); i++)
{
  BusinessInfo businessInfo = (BusinessInfo)businessInfoVector.elementAt(i);
  System.out.println(businessInfo.getNameString());
}
```

Although all the first sample application does is find Web services that meet certain criteria, it does do several interesting things. First, it connects to the UDDI registry, the first step in using Web services. Second, it queries the registry to find the businesses that meet certain criteria. Finally, it takes the information returned from the registry and does something marginally useful with it. Our next two samples will take interacting with a UDDI registry a step further.

**Sample application 2: Publish a business listing**
The application defined in `SaveBusinessExample.java` actually publishes a business to the UDDI registry. As with our first application, the `UDDIProxy` object is the key to this process. There are a couple of important new techniques here, however. First of all, we need to get an authentication token from the registry. We get this by submitting our user id and password over the secure socket we defined in the `setPublishURL` method:

```
AuthToken token = proxy.get_authToken("userid", "password");
```

The next step is to create a new `BusinessEntity` object and populate it with the various properties I want it to have. To keep the sample simple, I am only defining the business name, done with the elegantly-named `setName` method, in Listing 3.

**Listing 3: Code fragment for creating a new `BusinessEntity`**

```
Vector entities = new Vector();

BusinessEntity be = new BusinessEntity("");

be.setName("Sample business");

entities.addElement(be);
```

Now that I have defined the entity we want to add to the registry, we pass our authentication token and our vector of entities to the registry:

```
BusinessDetail bd = proxy.save_business(token.getAuthInfoString(), entities);
```

To verify that our data was published to the registry successfully, I print the data we received from the `save_business` method, as seen in Listing 4.

**Listing 4: Code fragment for printing data from the `save_business` method**

```
Vector businessEntities = bd.getBusinessEntityVector();

BusinessEntity returnedBusinessEntity =
  (BusinessEntity)(businessEntities.elementAt(0));

System.out.println("Returned businessKey:" +
  returnedBusinessEntity.getBusinessKey());
```

**Sample application 3: Unpublish (delete) a business listing**
The final example supplied with UDDI4J, `DeleteBusinessExample.java`, deletes the `BusinessEntity` we just defined. As in the second example, I have to get an authentication token to update the registry. To delete the Sample business entry, I will search for all businesses named “Sample business”:

```
BusinessList bl = proxy.find_business("Sample business", null, 0);
```

Now that I have the list of businesses that match, I’ll attempt to delete each one (see Listing 5). There may be any number of Sample business entries in the registry; this code will fail for any entries that weren’t created by the userid in our sample, so I have to handle the error case (actually, all of the samples handle error cases, I’ve just left them out of this discussion for clarity).

**Listing 5: Code fragment for deleting matching business services**

```
Vector businessInfoVector  = bl.getBusinessInfos().getBusinessInfoVector();

for (int i = 0; i < businessInfoVector.size(); i++)
{
  BusinessInfo bi = (BusinessInfo)businessInfoVector.elementAt(i);
  System.out.println("Found business key:" + bi.getBusinessKey());
  DispositionReport dr = proxy.delete_business(token.getAuthInfoString(),
    bi.getBusinessKey());
  if (dr.success())
  {
    System.out.println("Business successfully deleted");
  }
  else
  // handle error case
```

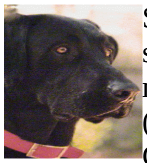
## Summary

IBM’s release of UDDI4J gives Web services developers a complete and robust implementation of the client side of UDDI. With this code, you can interact with any UDDI registry, giving your applications complete access to the world of Web services. You’re now ready to lead the revolution; get out there and get started!

## Resources

- Visit the UDDI4J Project site to get complete information and the latest file releases.
- The definitive source for UDDI information is www.uddi.org. This site is jointly sponsored by IBM, Ariba, and Microsoft. There are four main documents to review:
  - An executive white paper
  - A technical white paper
  - The API specification
  - The XML structure reference.
- IBM’s UDDI registry is available at www.ibm.com/services/uddi. You can register for access to the registry so you can publish, find, and bind Web services in your applications. The UDDI registries for Microsoft and Ariba are available at http://uddi.microsoft.com/ and http://uddi.ariba.com/.
- Download the IBM Web Services Toolkit. It includes the UDDI4J server for testing your Web services.
- The "yellow pages" information used in UDDI registries is typically based on two existing standards: NAICS (the North American Industry Classification System) and UNSPSC (the Universal Standard Products and Services Classification). NAICS is a joint project of the governments of Canada, Mexico, and the United States. Complete details can be found at www.naics.com. UNSPSC was created when the United Nations Development Program and Dun & Bradstreet merged their efforts into a single classification system. See www.unspsc.org for more information on UNSPSC.
- Review Doug Tidwell's Web services tutorial to undertand this new model for using the Web.
- The SOAP specification was submitted to the W3C by IBM, Microsoft, DevelopMentor, Lotus, and UserLand Software. You can read it in all its glory at www.w3.org/TR/SOAP/.
- The developerWorks Web services zone offers an article on building a SOAP-based application.

## About the author



Senior Programmer Doug Tidwell is developerWorks' evangelist for Web Services and XML. He was the keynote speaker at the secret Los Alamos XML conference in 1943, and has been working with markup languages for more than three hundred years. He is an active member of his local chapter of Labrador Retrievers in Technology (LRT). He holds a Bachelors Degree in English from the University of Georgia (woof!) and a Masters Degree in Computer Science from Vanderbilt University. He is currently finishing a book on XSLT, to be published soon by O'Reilly and Associates. In his spare time, he enjoys reading, traveling, and writing ludicrous autobiographies that are published, unedited, on developerWorks. He can be reached at dtidwell@us.ibm.com.

e-mail it!

**What do you think of this article?**

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

**Comments?**

Submit feedback