Understanding web services specifications, Part 3: Uniform Description and Discovery Interface (UDDI)

Nicholas Chase

July 21, 2006

The current emphasis on Service-Oriented Architectures (SOA) has put the spotlight on web services, but it's easy to get lost in all the information being bandied about. This series gives you the straight story on all of the major web service specifications, starting with Simple Object Access Protocol (SOAP) and working down to WS Business Process Execution Language (WS-BPEL). This tutorial, Part 3 in the series, explains the basic concepts of the Uniform Description and Discovery Interface (UDDI), and how to access it using a Java application.

View more content in this series

Before you start

This tutorial explains the Uniform Description and Discovery Interface, or UDDI. UDDI specifies a registry of web services and other corporate information and is intended to provide a way to discover new web services to use, and under ideal circumstances, automate their use.

This tutorial assumes that you are familiar with the general concepts of web services, as well as SOAP. Knowledge of WSDL is also helpful, but the tutorial does review the general concepts.

To follow along with the code in the last section, you should be familiar with Java, but the concepts, like the rest of the tutorial, apply to any programming language.

About this series

This tutorial series teaches the basic concepts of web services by following the exploits of the fictional newspaper, The Daily Moon, as the staff uses web services to create a workflow system to increase productivity in the midst of much change.

Part 1 explained the basic concepts behind web services and showed how to use SOAP, the specification that underlies most of what is to come, connecting the classifieds department with the Content Management System.

Part 2 takes things a step further, explaining how to use Web Services Description Language (WSDL) to define the messages produced at expected by web service, enabling the team to more easily create services and the clients that connect to them.
Part 3 finds the team with a number of services in place and a desire to locate them easily. In response, Universal Description, Discovery and Integration (UDDI) provides a searchable registry of available services at a way to publicize their own services to others.

Parts 4 and 5, WS-Security and WS-Policy, finds the team with a goal of securing the paper’s services and the changes the teams need to make in order to access those newly secured services.

Interoperability is the key word in Part 6, as services from several different implementations must be accessed from a single system. Part six covers the requirements and tests involved in WS-I certification.

Finally, Part 7 shows how to use Business Process Execution Language (WS-BPEL) to create complex applications from individual services.

Now let’s look at what this tutorial covers in a bit more detail.

**About this tutorial**

**Part 1 of this series** introduced you to web services and emphasized the importance of SOAP. We did this through the exploits of a Classifieds department of the fictional Daily Moon newspaper. In **Part 2 of this series**, the Classifieds Department decided to create a web service of their own, documenting its use with Web Services Description Language (WSDL). Here in Part 3, the paper's publisher has decided to go all out on web services, and has mandated the use of a UDDI registry to keep track of all available services.

In the course of this tutorial, you will learn:

- What UDDI is
- What UDDI is for
- The structure of UDDI data
- How to effectively represent WSDL using UDDI
- Operations you can perform on UDDI data
- Interacting with a UDDI registry using Java

**Prerequisites**

In order to follow along with the code for this tutorial, you will need to have the following software available:

Apache Geronimo or another UDDI implementation. In order to follow along with the code, you will need access to a UDDI registry. To that end, it isn't important what type of registry you use, as UDDI is fundamentally a web service and should be accessible from any platform or language, to any platform or language. That said, the Apache Geronimo application server, which is also the basis for IBM's WebSphere Community Edition, and which we have been using in Parts 1 and 2 of this series, has Apache's UDDI registry implementation, jUDDI, pre-installed and configured as part of a standard install. You can download [Apache Geronimo](https://geronimo.apache.org/) from Apache.org. For more information on installing Geronimo, see **Part 1** of this series.
Overview
Let's take a look at where we are so far.

The story so far
This series follows the staff of the fictional Daily Moon newspaper as it moves many of its everyday operations to a web services based system. In Part 1, the Classifieds Department learned about SOAP by interacting with the Content Management System, and in Part 2, they created their own service, and defined it using Web Services Description Language (WSDL). Now Rudy, the paper's publisher, is so impressed with the way web services are working within his organization that he wants to see them used more often, even to the point of interaction with other organizations.

To that end, he asks two members of the IT staff, Gene and Francis, to create a registry of web services, which he hopes will enable other departments to more easily find what they need to move their own operations over to web services.

The pair decides the best way to do that will be to implement a Uniform Description and Discovery Interface (UDDI) registry, which will enable them to not only store information on the paper's own services, it can also interact with registries at other newspapers.

WSDL refresher
Part of the process of registering a service with a UDDI registry involves manipulating the WSDL definition, so before moving on to the new project, Gene reviews the basic concepts of WSDL by going back over the files created in his last project. See Listing 1.

Listing 1. The WSDL file
```xml
<wsdl:definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsd1/
    xmlns:tns="http://www.daily-moon.com/classifieds"
```
xmlns:ns1="http://org.apache.axis2/xsd"
targetNamespace="http://www.daily-moon.com/classifieds">

<wsdl:types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://org.apache.axis2/xsd"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
    <xs:element name="createNewAdRequest">
      <xs:complexType>
        <xs:sequence>
          <xs:element type="xs:string" name="content" />
          <xs:element type="xs:string" name="endDate" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="createNewAdResponse">
      <xs:complexType>
        <xs:sequence>
          <xs:element type="xs:int" name="newAdId" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</wsdl:types>

<wsdl:message name="createNewAdRequestMessage">
  <wsdl:part name="part1" element="ns1:createNewAdRequest" />
</wsdl:message>

<wsdl:message name="createNewAdResponseMessage">
  <wsdl:part name="part1" element="ns1:createNewAdResponse" />
</wsdl:message>

<wsdl:message name="getExistingAdsResponseMessage">
  ...
</wsdl:message>

<wsdl:portType name="ClassifiedServicePortType">
  <wsdl:operation name="createNewAd">
    <wsdl:input message="tns:createNewAdRequestMessage" />
    <wsdl:output message="tns:createNewAdResponseMessage" />
  </wsdl:operation>

  <wsdl:operation name="finalizeIssue">...
  </wsdl:operation>

  <wsdl:operation name="editExistingAd">...
  </wsdl:operation>

  <wsdl:operation name="getExistingAds">...
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="ClassifiedServiceBinding"
  type="tns:ClassifiedServicePortType">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />

  <wsdl:operation name="createNewAd">
    <soap:operation soapAction="createNewAd" style="document" />
    <wsdl:input>
      <soap:body use="literal"
        namespace="http://ws.apache.org/axis2" />
    </wsdl:input>
  </wsdl:operation>
</wsdl:binding>
We've removed some pieces this document for space reasons, but let's review the general concepts.

Starting at the bottom, we have the actual service, which is available at a specific location, and implements the `ClassifiedServiceBinding` interface. That interface is represented as a binding, specifying operations defined in the `ClassifiedServicePortType` portType. The `portType` defines input and output messages using elements defined in the types section.

An XML client can read this information and use it to access the web service. (Or more accurately, a web service client generation tool can read this information and generate a client that can access the web service.)

**Where we go from here**
Over the course of this tutorial, Gene and Francis will break down the *Daily Moon's* information into UDDI-friendly structures, including business information and service information. They'll use it to enter the paper's information and services into a UDDI registry using UDDI4J. They will also work with the private registry of another newspaper, the *Daily Star*, to find a service that provides ongoing sports scores.

Now let's look at what UDDI is all about.

**Introducing UDDI**
UDDI is basically a web service, but it's so different from SOAP and WSDL that it's good to understand a little background before we start.

**Why UDDI?**
When all applications are local, it's pretty easy to find what you want. However, when you have a distributed system such as web services, you don't have the benefit of having a central repository. Distributed systems are also prone to change. This is the world into which UDDI came into being. It was intended to serve two purposes. In its initial incarnation, it was conceived as a sort of
"Universal Business Registry". The idea what is that businesses could search for partners using one of three methods:

- "White pages": The white pages were like the white pages of the phonebook, in which one could look up information about a company. For example, if you knew the name of the company, you could find out where it was, how to contact them, and even who to contact within the organization.
- "Yellow Pages": The yellow pages, again, were like the yellow pages of the phonebook, in which you could look up businesses based on classification. UDDI specified various taxonomies companies could use to classify themselves. For example, if you were looking for sporting goods, you could look for companies with a North American Industry Classification System (NAICS) code of 339920.
- "Green pages": No, there are no green pages in your phonebook, but the idea was that companies could use this method of searching to find trading partners had implemented a particular service. For example, you might do a search for companies that implements a distance calculation using zip codes.

UDDI was also conceived as a way to keep distributed applications running in the long term. The idea was that you would cache information on how to access a particular service, and if your client broke, the application would automatically go back to the registry and check to see if the information had changed. If it had, you could simply make the changes in your application (automatically, in an ideal world) and retry your request.

**UDDI in the real world**

When Francis went looking for the Universal Business Registry (UBR), he discovered that it no longer existed. In fact, UBR had been discontinued by its corporate sponsors when UDDI version 3.0 was released. Why? Well, no matter how good the UBR idea was, the reality was that virtually nobody used it. The reason is twofold; first, there was very little useful information in the UBR -- unless, that is, you are looking for a stock ticker service, in which case you could find quite an abundance. The second reason is that companies, as a rule, do not do business with organizations with which they are not familiar. Very few organizations, if any, are going to do an automated search for trading partners, and then open up their systems to them without a great bit of due diligence. So the automated search and invocation model didn't really take off.

It was also hampered by the fact that UDDI was, in some ways, ahead of its time. Designed to handle all types of services, and not just SOAP over HTTP, it did not immediately lend itself to the use of WSDL definitions, at least in the view of those trying to use it.

That's not to say, however, that UDDI is not being used at all. Instead, it is actually being used as more of a "private registry." Companies create their own registries, and enter their own services. UDDI can also be used and is a type of business registry internally, in the way LDAP since then has traditionally been used.

What's more, as we move past a "web services only" way of thinking into more of a Service Oriented Architecture (SOA) mentality, this ability to handle more than just SOAP may just prove itself in the long run.
The UDDI data structure

The information contained in a UDDI registry consists of five different types:

- **businessEntity**, or the actual corporate organization. This may mean overall organization, or it may be an affiliate or a subdivision.
- **publisherAssertion**, or the relationship between various **businessEntities**. PublisherAssertions must be claimed by both parties in order to be valid (so I can't claim to be a subdivision of another company) unless both entities are responsible to the publisher, or unless both entities are entered into the registry by the same user account.
- **bindingTemplate**, which is essentially the specification of an interface for service. It can be implemented by multiple **businessServices**.
- **businessService**, or a service provided by a business. Now, that may seem simple, but in the world of UDDI, that does not necessarily mean it is a web service. For example, you can actually specify your business' telephone support service (meaning the actual phone number the user dials, and everything that comes with it) as a UDDI service. Of course, you wouldn't have a dummy WSDL document, but it would be a service provided by your business.
- **tModels**, or metadata models. Investigating UDDI, Francis comes to the conclusion that the tModel is perhaps the biggest reason why UDDI did not take off as expected. As a registry of services, you would expect to find a way to directly specify the interface for a service, as WSDL does. But, as stated, UDDI was not ever intended to be exclusively about web services, and was designed with much more flexibility. tModels do help point to XML documents, as we'll see later on, but they are in fact meant as a general way to specify information about something, via service, a business, or anything else.

Those are the basic data structures of UDDI. Now let's look at the ways in which UDDI says we can interact with them.

The UDDI API

UDDI has a reputation for being overly complex, but at its core it is about the five datatypes described above, combined with four operations: **find**, **get**, **save**, and **delete**. In other words, the UDDI specification notes the following things you can do with data:

- **find_xx**: These methods, such as **find_businessEntity**, **find_businessService**, and so on, provide a way for you to search for a record in the UDDI registry. These methods return the key that identifies the object.
- **get_xx**: Once you have the unique key identifying an object, you can use the **get_xx** methods, such as **get_businessService** and so on, to retrieve the actual object itself. For example, **get_businessService** returns the entire **businessService** object, from which you gain any information that you need.
- **save_xx**: As you probably have guessed, these methods add information to the database, or they alter information that is already in the database.
- **delete_xx**: These methods, such as **delete_bindingTemplate**, **delete_tModel**, and so on, take the unique key of the object as a parameter, and remove it from the database. The actual behavior of these methods depends on what you are deleting. For example, because **tModels**
are frequently referenced by other objects in the database, they can’t actually be deleted; they are instead hidden.

Virtually all of UDDI rests on these 20 intersection between five objects (businessService, bindingTemplate, publisherAssertion, businessEntity, and tModel) and four actions (find, get, save, and delete).

Now let’s see what the structure actually looks like.

The UDDI data structure
To start the process of adding the company to a UDDI registry, Francis defined the information to enter in to the registry.

businessEntity
The first step is to enter the newspaper itself into the registry. To do this, Francis creates a businessEntity object. This object contains information about the business, both to identify it and to provide information about it. For example, the businessEntity object includes one more names, (which enables you to, say, create a name element for each language in which the company is represented), contact information (Francis is told to add Pat Mooney as the main contact), services associated with the entity (which Gene will and later) and information to identify and categorize the company.

Although most people will never interact with the UDDI Registry in this way, the fact is that UDDI is essentially a SOAP application; all of the operations that you execute happen as a result of SOAP messages being passed back and forth. What this means is that the businessEntity Francis is creating ultimately gets represented as an XML element. See Listing 2.

Listing 2. The businessEntity element

```
<businessEntity businessKey="{uuid}" operator="(optional)"
authorizedName="(optional)">
<discoveryURLs>
<discoveryURL></discoveryURL>
<discoveryURL></discoveryURL>
</discoveryURLs>
<name>The Daily Moon</name>
<name lang="fr-ca">La Lune Quotidienne</name>
<description></description>
<contacts>
<contact>
<description></description>
<personName>Pat Mooney</personName>
<phone></phone>
<email>patm@daily-moon.com</email>
<address useType="optional, billing, etc." sortCode="optional"
tModelKey="optional">
<addressLine></addressLine>
</address>
</contact>
...</contact>
</contacts>
</businessEntity>
```
A `businessEntity` record does not have to be complicated. In fact, only a single name and a unique identifier to act as the `businessKey` are required. However, for the sake of completeness, I've included the complete structure, minus some snipped sections we'll talk about in a moment. Empty elements should be removed in a production environment.

Let's look at some of the information here.

Starting at the top, `businessKey`, again, is a unique identifier. The operator and `authorizedName` attributes are handled internally by the registry; you don't have to touch them, but they represent the entity that runs the registry and the person or account who added the record.

The `discoveryURLs` are added to the registry every time a `businessEntity` gets saved to the registry. These URLs are meant to be a location to which the user can go for more information about the `businessEntity`.

Notice that Francis had added an additional name, representing the name of the newspaper in French, where the paper is trying to drum up business. Each of these entries carries the `lang` attribute specifying the code for the language in question. You can supply alternate names from common abbreviations, for example, IBM versus International Business Machines.

The optional description that comes next is simply a text string. In fact, most of the major elements in UDDI have an optional description.

Next, Francis can add more contacts. For each contact he can specify as much or as little information as necessary. The `personName` is typically supplied, but other information such as phone number, e-mail and so on are supplied based on the discretion of those involved, as all information in the UDDI Registry is publicly available to all users. Contacts may also have an address, with the `useType` specifying the type of address. For example, in the case of a contact, it may be home versus work and so on. In the case of a company, the `useType` might be main office, distribution center, and so on.

The `businessServices` element lists any services that have been associated with the `businessEntity`. Because Gene hasn't added any services yet, this element will typically be empty. The last two items, `identifierBag` and `categoryBag`, provide additional information about the company itself, and are used in searches. We will examine each of them in a moment.
**publisherAssertion**

A publisherAssertion is a statement about the relationship of one businessEntity to another. For example, Francis knows that one of the goals of this project is to use a service provided in the Daily Star, in which the Daily Star provides sports scores for the Daily Moon. That makes the Daily Star a content provider for the Daily Moon. Francis can represent that as shown in Listing 3.

**Listing 3. The publisherAssertion**

```xml
<publisherAssertion>
  <fromKey>(businessKey for Daily Star)</fromKey>
  <toKey>(businessKey for Daily Moon)</toKey>
  <keyedReference tModelKey="904B0009-DS3A-11DA-B055-858A1DA9D79"
                  keyValue="CONTENTPROVIDER" />
</publisherAssertion>
```

The fromKey and toKey elements represent the unique IDs for the two entities involved. In this case, that is the Daily Moon, and the Daily Star. The Daily Star is a content provider for the Daily Moon, so the keys are placed accordingly. The keyedReference will be seen again and again. In this case, we are saying that the reference itself is a content provider. The keyValue is the only attribute required for a keyedReference, but it's not unusual also to include a keyName for readability, and a tModelKey to identify a particular type of reference. In this case, the tModelKey represents the namespace of associations between newspapers as opposed to internal representations. We'll talk more about tModels in a minute.

**businessService**

Next, Francis needs to enter information about any actual services the Daily Moon is going to provide. A businessService object contains two major pieces of information in addition to the required unique serviceKey and the name. They are the bindingTemplates and any categories to which the service belongs. See Listing 4.

**Listing 4. The businessService**

```xml
<businessService serviceKey="required" businessKey="optional">
  <name></name>
  <description></description>
  <bindingTemplates>
    <bindingTemplate serviceKey="optional" bindingKey="required">
    ...
    </bindingTemplate>
    <bindingTemplates>
    ...
  </categoryBag>
</businessService>
```

**BindingTemplate**

Tmodelinstanceedetails is an indication of what applications or specifications it's compatible with. See Listing 5.

**Listing 5. The bindingTemplate**

```xml
(bindingTemplate serviceKey="optional"
```
A bindingTemplate is what defines both where the service can be found and what it does, which doesn't sound very complicated. However, both of these functions take on new significance in the context of UDDI. For example, to determine where a service can be accessed, you can either provide an accessPoint, which represents a simple URL, or any other appropriate data, such as a phone number for a helpline, or a hostingRedirector.

**Taxonomies**

When we classify information, it helps to be able to agree on what to call the classifications. These agreed-upon classifications are called taxonomies.

For example, the North American Industry Classification System (NAICS) distinguishes between "Sporting goods rental," "Sporting goods stores," and "Sporting goods stores, used," assigning each a unique code. This way, when you encounter a code of 532292, you know you're talking about a place to rent sporting goods, as opposed to a place to buy them.

UDDI operators add a number of taxonomies to the system as a matter of course, making them available for your use without you having to register tModels for them. For example, the jUDDI to create a tModel for uddi-org:general_keywords with a tModel key of A035A07C-F362-44DD-8F95-E2B134BF43B4, so you can use it without defining it yourself.
The `hostingRedirector` is used when the binding for a service changes. For example, if the *Daily Moon* were to set up its classified ads service and allow either newspapers to submit ads using the `createNewAdd()` call, it would need to either make sure that the service never changed, or provide a way to tell clients it has changed. That's where the `hostingRedirector` comes in. In that case, Francis (or actually, Gene) would create a new `bindingTemplate`, representing the new information, and enter that `bindingTemplate`'s `bindingKey` into the `bindingKey` attribute of the `hostingRedirector` element.

Defining what the service does can be even more difficult. The reason for this is twofold. First, because we are not necessarily talking about a SOAP service, you can't simply provide a WSDL file. Second, because this is meant to be an automated process, you need to provide this information in such a way that is unambiguous.

The result is that the `bindingTemplate` includes a `tModelInstanceDetails` element, which contains one or more `tModelInstanceDetail` elements. Each of those includes one or more `tModelInstanceInfo` elements, which point to a particular `tModel`. Again, we'll talk about `tModels` in a moment. The important thing to understand, however, is that be `tModelInstanceInfo` element can contain an `instanceDetails` element, which itself contains an `overviewDoc`, which includes the `overviewURL`, a URL for a document that actually defines the service. (Yes, that is where you specify the WSDL file.)

**TModels (including `tModel` of `tModels`)**

Now let's talk about `tModels`.

`tModels` actually serve two purposes. The first, as we have briefly seen in the discussion of `publisherAssertions`, is that of "namespace indicator". In other words, a `tModel` can provide a way to distinguish between types of information that can be easily confused. One example that is often given is that of tax codes, or other easily confusable numbers. For example, a `keyedReference` that includes the following doesn't seem to provide all of the information, because these two tax IDs belong to different countries. In both countries, they're called a tax ID, but programmatically, we must have a way to distinguish between them, as shown in **Listing 6**.

**Listing 6. Tax IDs from different countries**

```xml
<keyedReference tModelKey="" keyName="taxid"
keyValue="11111111" />
<keyedReference tModelKey="" keyName="taxid"
keyValue="22222222" />
```

That's where a `tModel` comes in. We can create a key model for "US tax code", and another for "Mexico tax code", and so on. Once we create those, we can use their `tModel` keys to further qualifying these `keyedReferences`. See **Listing 7**.
The other use of the tModel is as a specification identifier. For example, Francis knows that Gene is going to be creating entries for the ClassifiedService the pair worked on last month. This service represents a specific interface, and as such, it can be identified using a tModel. The tModel might look something like that shown in Listing 8.

Listing 8. The tModel

As with every other object we've looked at, the tModel has a unique key, which is its only required information. However, in this case, we would also want to specify the overviewURL, which will mean the URL for the interface's WSDL file. (We'll create that in the next section.)

Again, tModels can be identified using an identifierBag, and categorized using a categoryBag. Let's look at how that works.

identifierBag

Because UDDI is intended to be flexible, there is no specific way to identify a businessEntity. UDDI does not mandate identifying a business by its D-U-N-S number, or by its tax ID number, or by any other particular identification method. Instead, it provides a container into which you can place as many identifiers as necessary, as you see in Listing9.
Listing 9. The identifierBag
<identifierBag><keyedReference
tModelKey="4064C064-6D14-4F35-8953-9652166475A9"
   keyName="DUNS Number" keyValue="55555555" />
   <keyedReference tModelKey="" keyName="US Tax ID" keyValue="111111111" />
</identifierBag>

Each keyedReference you add in an identifierBag represents another opportunity for users to find
the object in question in a search.

categoryBag
The categoryBag works like the identifierBag, in that it contains one or more keyedReferences.
See Listing 10.

Listing 10. The categoryBag
<categoryBag>
   <keyedReference tModelKey="C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
      keyName="optional" keyValue="511110" />
</categoryBag>

In this case, Francis has created a categoryBag that shows that the Daily Moon is a newspaper
branch office. Now let's look at using all of these objects.

Mapping WSDL to UDDI
Before Gene can write any code to send all of this information to the UDDI registry, he has to make
one important connection between their current services and UDDI. He has to adapt them to the
UDDI notion of services, interfaces, and binding templates.

The service versus the interface
The UDDI vision of a service, as you may recall, consists of three parts: the interface, defined
using tModels; the businessService; and the bindingTemplate, which ties the two of them
together. So Gene's first step is to figure out how the WSDL file that defines the SOAP service can
fit into this structure.

His first step is to break down the WSDL. Fortunately, this is fairly simple. The interface consists of
all of the pieces of the WSDL file except for the actual implementation itself. See Listing 11.

Listing 11. The WSDL interface
<wsdl:definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:tns="http://ws.apache.org/axis2"
xmlns:axis2="http://ws.apache.org/axis2"
xmlns:ns1="http://org.apache.axis2/xsd"
targetNamespace="http://ws.apache.org/axis2">
   <wsdl:types>
      <xs:schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://org.apache.axis2/xsd"
         elementFormDefault="unqualified"
         attributeFormDefault="unqualified">
   </wsdl:types>
</wsdl:definitions>
This includes the type definitions (the XML schema), any messages, the portType, and even the binding, because the binding is not specific to a particular implementation.

The implementation consists of the actual service element. See Listing 12.

**Listing 12. The service implementation WSDL**

```xml
<wsdl:definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                   xmlns:wsd1="http://schemas.xmlsoap.org/wsdl/"
                   xmlns:tns="http://www.daily-moon.com/ClassifiedService-interface"/>
```
Notice that the implementation file imports the interface file. In other words, this file is the complete definition of the service.

Now let's look at how that corresponds to our UDDI objects.

**Mapping the interface to a tModel**
Gene first has to map the interface to an appropriate tModel. See Listing 13.

**Listing 13. The tModel**

```xml
<tModel tModelKey="66999A50-D5F4-11DA-9A50-FA44D6AD622A">
  <name>http://www.daily-moon.com/ClassifiedService-interface</name>
  <description>Interface for the Daily Moon Classified Department web application</description>
  <overviewDoc>
    <description>WSDL interface document</description>
    <overviewURL>http://www.daily-moon.com/ClassifiedService-interface.wsdl</overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="C1ACF26D-9672-4404-9D70-39B756E62AB4" keyName="uddi-org:types" keyValue="wsdlSpec" />
    <keyedReference tModelKey="C0B9FE13-179F-413D-8A5B-5004DB8E5BB2" keyName="ntis-gov:naics:1997" keyValue="511110" />
  </categoryBag>
</tModel>
```

Starting at the bottom, notice the categoryBag. Gene has added two keyedReferences. The first specifies that we are in fact dealing with a WSDL file. That particular tModelKey has been predefined to represent the uddi-org:types namespace. All WSDL files must be specified in this way. The second reference is tied to the ntis-gov:naics:1997 namespace, so it would be easy for
Gene's users to find this definition among all of the WSDL files that represent services having to do with newspapers. Moving up, we see the overviewDoc. In addition to the optional description, it specifies the actual URL at which we can find the WSDL file.

Finally, the name of the tModel should match the target namespace of the WSDL file.

Mapping the implementation to a binding template
Gene next has to map the implementation itself to a bindingTemplate. See Listing 14.

Listing 14. The bindingTemplate
<bindingTemplate serviceKey="" bindingKey="">
  <accessPoint>
    http://www.daily-moon.com:8080/axis2/services/ClassifiedService
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceDetail>
      <tModelInstanceInfo
        tModelKey="66999A50-D5F4-11DA-9A50-FA44D6AD622A">
        <instanceDetails>
          <description>The canonical implementation of the Daily Moon's classified service.</description>
          <overviewDoc>
            http://www.daily-moon.com/ClassifiedService-impl.wsdl
          </overviewDoc>
        </instanceDetails>
      </tModelInstanceInfo>
    </tModelInstanceDetail>
  </tModelInstanceDetails>
</bindingTemplate>

Here we see that Gene creates a new bindingTemplate, setting the accessPoint as the URL for the actual service. The key modelInstanceDetails links the interface tModel (via the tModelKey) to the implementation document (specified in the overviewDoc). Now he just has to reference it from the service.

Referencing the bindingTemplate from the businessService and the businessEntity
The final step is to put it all together and reference the bindingTemplate from the businessService, and the businessService from the businessEntity. See Listing 15.

Listing 15. The complete businessEntity
<businessEntity businessKey="1A3DB888-D5F4-11DA-B880-F94D3591C691">
  <name>The Daily Moon</name>
  <name lang="fr-ca">La Lune Quotidienne</name>
  <contacts>
    <contact>
      <personName>Pat Mooney</personName>
      <email>patm@daily-moon.com</email>
    </contact>
  </contacts>
</businessEntity>
In other words, the service is now complete: Gene now has a **businessEntity** that includes a service.
Finding and using information

Now that all of the information is in, at least theoretically, Gene needs to look at the process for finding it again.

How UDDI searches work: finding a business

Searching in UDDI involves two steps. First, you find the key of the object you want, and then you get information on that object. To that end, in order for Gene to find a business, he needs to create a `find_business` request, as shown in **Listing 16**.

**Listing 16. Finding a business by category**

```xml
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
  <categoryBag>
    <keyedReference tModelKey="C0B9FE13-179F-413D-8A5B-5004DB8E5BB2" keyName="optional" keyValue="511110" />
  </categoryBag>
</find_business>
```

This request will find all businesses that match the categories referenced in the `categoryBag`. Searches on the `categoryBag` value are treated as an "AND" search. In other words, any item that match is at least one of the categories a to match. You can also specify an `identifierBag` search in this way. Gene isn't really interested in searching by category, however. For the moment, he just wants to see whether or not the business is in the registry, so instead he'd rather search by name, like in **Listing 17**.

**Listing 17. Finding a business by name**

```xml
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
  <name>daily</name>
</find_business>
```

Note that a name search is treated as though the name you're looking for is followed by a wildcard. In other words, the search here is the same as doing a search for "daily*", so both the *Daily Moon* and the *Daily Star*, and any other business that starts with "daily" will show up. These searches are also case-insensitive.

What a search finds

The `find_business` request to returns a `businessList` See **Listing 18**.

**Listing 18. The businesses returned by a search**

```xml
<businessList generic="2.0" operator="uddi.sourceOperator" truncated="true" xmlns="urn:uddi-org:api_v2">
  <businessInfos>
    <businessInfo businessKey="1A3DB880-D5F4-11DA-B880-F94D3591C691">
      <name>The Daily Moon</name>
      <serviceInfos>
        ...
      </serviceInfos>
    </businessInfo>
    ...
  </businessInfos>
</businessList>
```
The `businessList` includes one `businessInfo` element for each `businessEntity` that satisfies the search. Each of those `serviceInfo` elements includes a `businessKey` for the actual `businessEntity`. It also includes an abbreviated version of the information that would be listed in the `businessEntity`, so that you can get an idea of whether this is the business you're looking for.

Once you have the appropriate `businessKey`, you need to get more information on the `businessEntity`.

**Once you have the key**

To get more information on a particular `businessEntity` based on the `businessKey`, you need to execute a `get_businessDetail` request. See **Listing 19**.

**Listing 19. Getting business information**

```
<get_businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
  <businessKey>1A3DB880-D5F4-11DA-B880-F94D3591C691</businessKey>
</get_businessDetail>
```

You can list as many `businessKeys` in a `get_businessDetail` request as you like. Each will have its own `businessDetail` element in the results. See **Listing 20**.

**Listing 20. Returned business details**

```
<businessDetail generic="2.0" operator="uddi.sourceOperator" truncated="false" xmlns="urn:uddi-org:api_v2">
  <businessEntity businessKey="1A3DB880-D5F4-11DA-B880-F94D3591C691">
    <name>The Daily Moon</name>
    <name lang="fr-ca">La Lune Quotidienne</name>
    <contacts>...
      <keyedReference
        tModelKey="C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
        keyName="optional" keyValue="511110" />
    </categoryBag>
  </businessEntity>
</businessDetail>
```

Each `businessDetail` element includes the full `businessEntity` of the business involved.
Changing how search works

While this works, there are times that you want to change the behavior of the search. For example, perhaps you want to look for an exact match rather than a wildcard match, or you want a case-sensitive search. You can change the behavior of the search using `findQualifiers`. See Listing 21.

**Listing 21. Using `findQualifiers`**

```xml
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
    <findQualifiers>
        <findQualifier>sortByNameAsc</findQualifier>
    </findQualifiers>
    <name>daily</name>
</find_business>
```

In this case, the search will proceed as usual, but the results will be returned sorted by name. You can add as many `findQualifier` elements as you like. Among those that are available for use are `exactNameMatch`, `caseSensitiveMatch`, `sortByNameDesc`, `sortByDateAsc`, and `sortByDateDesc`, or `LikeKeys` (which tells the registry to treat keys in a `categoryBag` or `identifierBag` as an "OR" condition if they share the same `tModelKey`), and `combineCategoryBags` (which treats all of the `categoryBags` of a `businessEntity`, whether they belong to the `businessEntity`, the `businessService`, or the `bindingTemplate`, as if they belong to the `businessService`).

Finding a type of service

Gene also wants to make sure that users can find any services based on a particular interface. For example, this query will locate `bindingTemplates` that you use the `ClassifiedService` interface. See Listing 22.

**Listing 22. Finding a `bindingTemplate`**

```xml
<find_binding generic="2.0" xmlns="urn:uddi-org:api_v2">
    <tModelBag>
        <tModelKey>66999A50-D5F4-11DA-9A50-FA44D6AD622A</tModelKey>
    </tModelBag>
</find_binding>
```

This query returns a `serviceList`. See Listing 23.

**Listing 23. The returned services**

```xml
<serviceList generic="2.0" operator="uddi.sourceOperator" xmlns="urn:uddi-org:api_v2">
    <serviceInfos>
        <serviceInfo
            serviceKey="064B4170-D5F5-11DA-8170-A74C17FA61A7"
            businessKey="1A3DB880-D5F4-11DA-B880-F94D3591C691">
            <name>ClassifiedService</name>
        </serviceInfo>
    </serviceInfos>
</serviceList>
```

As in the case to `find_business`, in which you then had to go back and request information on the `businessEntity`, you would then specifically request information on the servers based on the `serviceKey`. See Listing 24.
Listing 24. Getting service details

```xml
<get_serviceDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
  <serviceKey>064B4170-D5F5-11DA-8170-A74C17FA61A7</serviceKey>
</get_serviceDetail>
```

This query returns a serviceDetail for each serviceKey specified. See Listing 25.

Listing 25. The returned service details

```xml
<serviceDetail generic="2.0" operator="uddi.sourceOperator"
    xmlns="urn:uddi-org:api_v2">
  <businessService
    serviceKey="064B4170-D5F5-11DA-8170-A74C17FA61A7"
    businessKey="1A3DB880-D5F4-11DA-B880-F94D3591C691">
    <name>ClassifiedService</name>
    <bindingTemplates>
      <bindingTemplate
        bindingKey="904BD800-D53A-11DA-B055-850A1DA99D79">
        <accessPoint>
          http://www.daily-moon.com:8080/axis2/services/ClassifiedService
        </accessPoint>
        ...
        <keyedReference
          tModelKey="C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
          keyName="ntis-gov:naics:1997" keyValue="511110" />
      </bindingTemplate>
    </bindingTemplates>
  </businessService>
  ...
</serviceDetail>
```

Now that Gene knows what he wants to do, he's ready to start coding.

Working with UDDI programmatically

Gene's final job is to set up a group of proof-of-concept applications to show how other groups can build applications based around the UDDI registry.

Setting up

The first step in working with a UDDI registry is the registry software itself, but Gene is pleasantly surprised to discover that because the paper is using Apache Geronimo as its Web application server, that step is a ready taken care of; Geronimo comes with Apache jUDDI preinstalled.

That done, Gene has to decide how he is actually going to access the registry. Here, he has two choices. The first choose is to do it directly, using a UDDI based APIs such as the one included in jUDDI or UDDI4J. The second is to use a Java API for XML Registries (JAXR) implementation. JAXR is intended to access any registry, with UDDI and ebXML support being its main goals.

Gene decides to go straight to the source, and downloads UDDI4J. UDDI4J, in turn, is dependent on some of the libraries distributed with Apache Axis, so he downloads that, too. (Note that even though previous parts of this project have used Axis2, UDDI4J is dependent on the libraries distributed with Axis 1.)
Setup is fairly straightforward, requiring him only to make sure that all of the necessary UDDI4J and Axis *.jar files are on the CLASSPATH. See Listing 26.

Listing 26. Setting the CLASSPATH set

```
CLASSPATH=.;<UDDI4J_HOME>/lib/uddi4j.jar;<AXIS_HOME>/lib/axis.jar;
<AXIS_HOME>/lib/saj.jar;<AXIS_HOME>/lib/jaxrpc.jar;<AXIS_HOME>/lib/log4j-1.2.8.jar;<AXIS_HOME>/lib/commons-logging-1.0.4.jar;
<AXIS_HOME>/lib/commons-discovery-0.2.jar;
```

Finally, Gene needs a username and password for connecting to the registry. In this respect, every registry is different, so we won't go into details. The default installation of UDDI in Apache Geronimo uses a username and password of "juddi".

Connect to the registry

The first step in any application is to connect to the registry. See Listing 27.

Listing 27. Connecting to the registry

```
import org.uddi4j.client.UDDIProxy;
import org.uddi4j.response.AuthToken;
import org.uddi4j.transport.TransportFactory;

public class CreateNewBusiness {
    public static void main (String args[]){
        String inquiryURL = "http://localhost:8080/juddi/inquiry";
        String publishURL = "http://localhost:8080/juddi/publish";
        String userId = "juddi";
        String credential = "juddi";

        String transportClass = "org.uddi4j.transport.ApacheAxisTransport";
        System.setProperty(TransportFactory.PROPERTY_NAME, transportClass);

        UDDIProxy proxy = new UDDIProxy();
        try {
            proxy.setInquiryURL(inquiryURL);
            proxy.setPublishURL(publishURL);

            AuthToken token = proxy.get_authToken(userId, credential);
            System.out.println("Security authToken:" +
                                     token.getAuthInfoString());
        } catch(Exception e ) {
            e.printStackTrace();
        }
    }
}
```

Starting at the top, Gene sets the various values he'll need throughout the application. Specifically, Gene sets the class name for the TransportFactory. UDDI4J can use any number of appropriate classes, such as the one that comes with Apache Axis, as you see here, the one that comes with the old Apache SOAP package, or anything similar.

Understanding web services specifications, Part 3: Uniform Description and Discovery Interface (UDDI)
Finally, he sets the `inquiryURL` and `publishURL` values for the registry -- they will typically be different, in part because publishing is usually accomplished using SSL -- and request an authorization token.

This token is important because it is how each request indicates that it has been authenticated to the server. Because each registry handles things differently, it is not practical (or secure) to keep sending the username and password with each request.

When Gene runs the application, he sees a result of:

```
Security authToken:authToken:505A7DE0-D897-11DA-A5BF-ADFF17B378CF
```

### Create a business entity

Next Gene wants to create the business entity. See Listing 28.

#### Listing 28. Creating the `businessEntity`

```java
import org.uddi4j.client.UDDIProxy;
import org.uddi4j.response.AuthToken;
import org.uddi4j.transport.TransportFactory;
import org.uddi4j.datatype.business.BusinessEntity;
import org.uddi4j.response.BusinessDetail;
import org.uddi4j.datatype.business.Contact;
import org.uddi4j.datatype.business.Contacts;
import org.uddi4j.util.KeyedReference;
import org.uddi4j.util.IdentifierBag;
import org.uddi4j.util.CategoryBag;
import org.uddi4j.datatype.tmodel.TModel;
import java.util.Vector;

public class CreateNewBusiness {
    public static void main (String args[]){
        String inquiryURL = "http://localhost:8080/juddi/inquiry";
        String publishURL = "http://localhost:8080/juddi/publish";
        String userId = "juddi";
        String credential = "juddi";
        String businessName = "The Daily Moon";
        String alt_businessName = "Daily Moon";
        String contact_personName = "Pat Moonie";
        String contact_phone = "212-555-1212";
        String identifier_homepage = "http://www.daily-moon.com";
        String category_NAICS = "511110";
        String transportClass = "org.uddi4j.transport.ApacheAxisTransport";
        System.setProperty(TransportFactory.PROPERTY_NAME, transportClass);

        UDDIProxy proxy = new UDDIProxy();
        try {
            proxy.setInquiryURL(inquiryURL);
            proxy.setPublishURL(publishURL);

            AuthToken token = proxy.get_authToken(userId, credential);
            System.out.println("Security authToken:" +
                token.getAuthToken());
```

```
The process of putting together a businessEntity basically consists of building elements just as they are in the XML, with the "plurals" (such as contacts, tModelInfos and so on) being handled as vectors.

In this case, Gene builds the contact vector, the identifierBag, and the categoryBag, and adds them to the BusinessEntity object. Object itself gets added to a Vector, which is said to these save_business() method, along with the authorization token.

The result is a BusinessDetail object that contains one or more BusinessEntity objects as they exist in the database. In this case, Gene submitted the BusinessEntity without a value for the businessKey, which lets the registry know to create a new object, and thus a new key. By retrieving the first BusinessEntity from the vector, Gene can then obtain the new businessKey.

When he runs the application, he gets the result shown in Listing 29:
Listing 29. Obtaining a new businessKey

Security authToken: authToken: 1D05DB90-D899-11DA-A5BF-D3513DE466CA
The new businessKey: 1D359E20-D899-11DA-A5BF-F9EAF6D4E3F7

Create a service interface  tModel

With the business in place, Gene next sets about adding the service interface. See Listing 30.

Listing 30. Creating a tModel

```java
import org.uddi4j.client.UDDIProxy;
import org.uddi4j.response.AuthToken;
import org.uddi4j.transport.TransportFactory;
import java.util.Vector;
import org.uddi4j.datatype.tmodel.TModel;
import org.uddi4j.response.TModelDetail;
import org.uddi4j.datatype.OverviewURL;
import org.uddi4j.datatype.OverviewDoc;
import org.uddi4j.util.CategoryBag;
import org.uddi4j.util.KeyedReference;

public class CreateNewInterface{

    public static void main (String args[]){
        String inquiryURL = "http://localhost:8080/juddi/inquiry";
        String publishURL = "http://localhost:8080/juddi/publish";
        String userId = "juddi";
        String credential = "juddi";

        String transportClass =
            "org.uddi4j.transport.ApacheAxisTransport";
        System.setProperty(TransportFactory.PROPERTY_NAME,
            transportClass);

        UDDIProxy proxy = new UDDIProxy();

        try {
            proxy.setInquiryURL(inquiryURL);
            proxy.setPublishURL(publishURL);

            AuthToken token = proxy.get_authToken(userId, credential);
            System.out.println("Security authToken:" +
                token.getAuthInfoString());

            Vector tModels = new Vector();
            TModel tModel = new TModel("",
                "http://www.daily-moon.com/classifieds-interface");
            tModel.setDefaultDescriptionString(
                "Interface for the Daily Moon Classified " +
                "Department web application");

            OverviewDoc overviewDoc = new OverviewDoc();
            overviewDoc.setDefaultDescriptionString(
                "WSDL interface document");
            OverviewURL overviewURL = new OverviewURL("http://www.nicholaschase.com/ClassifiedsService-interface.wsdl");
            overviewDoc.setOverviewURL(overviewURL);
            tModel.setOverviewDoc(overviewDoc);

            KeyedReference wsdlNotation =
                new KeyedReference(TModel.TYPES_TMODEL_KEY,
                    "wsdlSpec",
                    "http://www.daily-moon.com/classifieds-interface.wSDL");
            tModel.addOverviewURL(overviewURL);

            tModels.add(tModel);
            proxy.addTModelDetail(tModels);
        }
```
As before, Gene sets out to model the XML element in Java code. He starts by creating the TModel, again with no key to signify that he's creating a new object, and sets the description. He then creates the OverviewDoc, complete with description and OverviewURL. Next, he creates the KeyedReferences for the CategoryBag. Note that he's referencing specific tModels, so he provides the tModelKey for each of them; this value is optional. Finally, he saves the TModel, once again retrieving the object from the returned value.

Running the application gives Gene the result shown in Listing 31:

Listing 31. Obtaining a new tModel key

```java
TModel Saved: http://www.daily-moon.com/classifieds-interface
TModel Key : uuid:332F67D0-D8A1-11DA-A5BF-DF15F533BFA1

Create a service
Creating the service itself follows the same basic procedure. See Listing 32.

Listing 32. Creating the service

```java
import org.uddi4j.client.UDDIProxy;
import org.uddi4j.response.AuthToken;
import org.uddi4j.transport.TransportFactory;
import java.util.Vector;
import org.uddi4j.datatype.business.BusinessEntity;
import org.uddi4j.datatype.service.BusinessService;
import org.uddi4j.datatype.binding.BindingTemplate;
import org.uddi4j.datatype.binding.BindingTemplates;
import org.uddi4j.util.CategoryBag;
import org.uddi4j.util.KeyedReference;
import org.uddi4j.datatype.binding.AccessPoint;
import org.uddi4j.datatype.binding.TModelInstanceDetails;
```
import org.uddi4j.datatype.binding.TModelInstanceInfo;
import org.uddi4j.datatype.binding.InstanceDetails;
import org.uddi4j.datatype.OverviewDoc;
import org.uddi4j.datatype.OverviewURL;
import org.uddi4j.datatype.tmodel.TModel;
import org.uddi4j.response.ServiceDetail;

public class CreateNewService{

    public static void main (String args[]){
        String inquiryURL = "http://localhost:8080/juddi/inquiry";
        String publishURL = "http://localhost:8080/juddi/publish";
        String userId = "juddi";
        String credential = "juddi";
        String businessKey = "1D359E20-D899-11DA-A5BF-F9EAF6D4E3F7";
        String transportClass = "org.uddi4j.transport.ApacheAxisTransport";
        System.setProperty(TransportFactory.PROPERTY_NAME,
            transportClass);
        UDDIProxy proxy = new UDDIProxy();
        try {
            proxy.setInquiryURL(inquiryURL);
            proxy.setPublishURL(publishURL);
            AuthToken token = proxy.get_authToken(userId, credential);
            System.out.println("Security authToken:" +
                token.getAuthInfoString());
            BusinessService businessService = new BusinessService("");
            businessService.setDefaultNameString("ClassifiedService",null);
            businessService.setBusinessKey(businessKey);
            BindingTemplate bindingTemplate = new BindingTemplate();
            AccessPoint accessPoint = new AccessPoint("http://www.daily-moon.com:8080/axis2/services/ClassifiedService",
                "http");
            bindingTemplate.setAccessPoint(accessPoint);
            TModelInstanceDetails tModelDetails =
                new TModelInstanceDetails();
            TModelInstanceInfo tModelInstanceInfo =
                new TModelInstanceInfo("332F67D0-D8A1-11DA-A5BF-DF15F533BFA1");
            InstanceDetails instanceDetails = new InstanceDetails();
            OverviewDoc overviewDoc = new OverviewDoc();
            OverviewURL overviewURL = new OverviewURL("http://www.nicholaschase.com/ClassifiedService-impl.wsdl");
            overviewDoc.setOverviewURL(overviewURL);
            instanceDetails.setOverviewDoc(overviewDoc);
            tModelInstanceInfo.setInstanceDetails(instanceDetails);
            tModelDetails.add(tModelInstanceInfo);
            bindingTemplate.setTModelInstanceDetails(tModelDetails);
            BindingTemplates bindingTemplates = new BindingTemplates();
            bindingTemplates.add(bindingTemplate);
            businessService.setBindingTemplates(bindingTemplates);
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}
KeyedReference wsdlNotation =
    new KeyedReference(TModel.TYPES_TMODEL_KEY,
        "wsdlSpec",
        "C1ACF26D-9672-4404-9D70-39B756E62A04");
KeyedReference typeNotation =
        "511110",
        "C0B9FE13-179F-413D-8A5B-5084DB85B2");
CategoryBag catBag = new CategoryBag();
catBag.add(wsdlNotation);
catBag.add(typeNotation);
businessService.setCategoryBag(catBag);
Vector services = new Vector();
services.addElement(businessService);
ServiceDetail serviceDetail = proxy.save_service(
    token.getAuthInfoString(),services);
Vector businessServices =
    serviceDetail.getBusinessServiceVector();
BusinessService businessServiceReturned =
    (BusinessService)(businessServices.elementAt(0));
String serviceKey = businessServiceReturned.getServiceKey();
System.out.println("The Name: "+
    businessServiceReturned.getDefaultNameString());
System.out.println("The ServiceKey: "+ serviceKey);
} catch(Exception e ) {
    e.printStackTrace();
}
}

By now Gene found the pattern familiar; create the objects as they exist in the XML document. Create the BusinessService, add the businessKey. Create the BindingTemplate, making sure to reference the tModelKey created when he created the interface tModel and the URL for the actual WSDL file. Create the categoryBag, add it, and save the service. Retrieve the service and its name and key. Running the application gave Gene the output shown in Listing 33.

Listing 33. Retrieving the service name key

Security authToken:authToken:E4203390-D8A5-11DA-A5BF-8F6C6EDFEBD
The Name: ClassifiedService
The ServiceKey: E43F2D40-D8A5-11DA-A5BF-880A3C83523B

Find a business

Next Gene turns his eye towards putting those searches into play. He starts with a business search, trying to make sure that the newspaper has been entered into the registry, and that it hasn't been duplicated. See Listing 34.

Listing 34. Finding a business

import org.uddi4j.client.UDDIProxy;
import org.uddi4j.response.AuthToken;
import org.uddi4j.transport.TransportFactory;
import java.util.Vector;
import org.uddi4j.datatype.Name;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.response.BusinessList;
import org.uddi4j.util.FindQualifier;
import org.uddi4j.util.FindQualifiers;

public class FindBusiness{

    public static void main (String args[]){
        String inquiryURL = "http://localhost:8080/juddi/inquiry";
        String publishURL = "http://localhost:8080/juddi/publish";
        String userId = "juddi";
        String credential = "juddi";

        String transportClass = "org.uddi4j.transport.ApacheAxisTransport";
        System.setProperty(TransportFactory.PROPERTY_NAME, transportClass);

        UDDIProxy proxy = new UDDIProxy();
        try {
            proxy.setInquiryURL(inquiryURL);
            proxy.setPublishURL(publishURL);

            AuthToken token = proxy.get_authToken(userId, credential);
            System.out.println("Security authToken:" +
                    token.getAuthInfoString());

            String businessToFind = "The Daily Moon";
            Vector names = new Vector();
            names.add(new Name(businessToFind));

            FindQualifiers findQualifiers = new FindQualifiers();
            Vector qualifier = new Vector();
            qualifier.add(new FindQualifier("exactNameMatch"));
            findQualifiers.setFindQualifierVector(qualifier);
            BusinessList businessList =
                    proxy.find_business(names, null, null,
                    null,null,findQualifiers,5);

            Vector businessInfoVector  =
                    businessList.getBusinessInfos().getBusinessInfoVector();
            for( int i = 0; i < businessInfoVector.size(); i++ ){
                BusinessInfo businessInfo =
                        (BusinessInfo)businessInfoVector.elementAt(i);
                System.out.println(businessInfo.getDefaultNameString() +
                        " : " +
                        businessInfo.getBusinessKey());
            }
        } catch(Exception e ) {
            e.printStackTrace();
        }
    }
}

Gene's looking for a particular name value, so that's the type of search he builds. He creates a Vector of Name objects with a single item and creates a FindQualifier to specify that he's looking only for an exact match. Finally, he feeds the request to the find_business() method, which takes as parameters the names, any discoveryURLs, an IdentifierBag, a CategoryBag, a TModelBag, any FindQualifiers, and the maximum number of results to return.
The result is a BusinessList, through which Gene iterates to see all of its values in case the information has been duplicated. For each one, he prints the name and the businessKey as you see in Listing 35.

### Listing 35. Business name and the businessKey

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>The Daily Moon:</td>
<td>1D359E20-D899-11DA-A5BF-F9EAF6D4E3F7</td>
</tr>
<tr>
<td>The Daily Moon:</td>
<td>1A3DB880-D5F4-11DA-B880-F94D3591C691</td>
</tr>
</tbody>
</table>

In this case, he can see that the information has been duplicated, so he can feed the extra businessKey to the UDDIProxy object's delete_business() method.

### Find a service

Next, Gene wants to make sure the ClassifiedService can be found based on its category. See Listing 36.

### Listing 36. Finding a type of service

```java
import org.uddi4j.client.UDDIProxy;
import org.uddi4j.response.AuthToken;
import org.uddi4j.transport.TransportFactory;
import java.util.Vector;
import org.uddi4j.util.CategoryBag;
import org.uddi4j.util.KeyedReference;
import org.uddi4j.datatype.tmodel.TModel;
import org.uddi4j.datatype.binding.BindingTemplate;
import org.uddi4j.datatype.binding.TModelInstanceInfo;
import org.uddi4j.datatype.binding.TModelInstanceDetails;
import org.uddi4j.datatype.binding.AccessPoint;
import org.uddi4j.datatype.binding.InstanceDetails;
import org.uddi4j.datatype.service.BusinessService;
import org.uddi4j.response.ServiceDetail;
import org.uddi4j.response.ServiceInfo;
import org.uddi4j.response.ServiceList;

class FindTypeOfService {
    public static void main (String args[]){
        String inquiryURL = "http://localhost:8080/juddi/inquiry";
        String publishURL = "http://localhost:8080/juddi/publish";
        String userId = "juddi";
        String credential = "juddi";
        String transportClass = "org.uddi4j.transport.ApacheAxisTransport";
        System.setProperty(TransportFactory.PROPERTY_NAME, transportClass);

        UDDIProxy proxy = new UDDIProxy();

        try {
            proxy.setInquiryURL(inquiryURL);
            proxy.setPublishURL(publishURL);

            AuthToken token = proxy.get_authToken(userId, credential);
            System.out.println("Security authToken:" +
                              token.getAuthInfoString());

            KeyedReference wsdlNotation =
                new KeyedReference(TModel.TYPES_TMODEL_KEY,
                                  TModel.TYPES_TMODEL_KEY,
                                  //...
Gene's first step is to create a CategoryBag that enables him to search for all WSDL specified services having to do with newspapers. The result of that search is a ServiceList, through which he iterates to get more information. A ServiceInfo object represents each service in the result, and he loops through each of them, displaying their name and serviceKey.

If, however, Gene wants more details, such as the URL for the WSDL file, he has to use the serviceKey to request the ServiceDetail for the service. The ServiceDetail has zero or more BindingTemplates associated with it, and if he finds one, he works his way down through the
TModelInstanceDetails object to the InstanceDetails object, which includes the OverviewDoc, and thus the OverviewURL. The result shows not just the name and serviceKey of the resulting service, but also the URL where the user can find more information. See Listing 37.

Listing 37. The final results

```plaintext
Security authToken:authToken:59FE37C0-D8AF-11DA-A5BF-A79333DB92F9
Service name: ClassifiedService
Service key: E43F2D40-D8A5-11DA-A5BF-880A3C83523B
Document located at
   http://www.nicholaschase.com/ClassifiedService-impl.wsdl
```

Summary

A distributed environment requires a centralized repository so users and developers know what's available and where to find it, and web services are no exception. This tutorial explained the use of the Uniform Description and Discovery Interface (UDDI), which provides a registry of information about available businesses and services.

In this tutorial, you learned:

- What UDDI is and why it's important
- How to structure UDDI data
- How to distinguish between SOAP interfaces and implementations
- How to find data in a UDDI registry
- How to interact with a UDDI registry using Java

The next parts of this series will deal with increasing security needs, examining WS-Security and WS-Policy.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>uddicode.zip</td>
<td>6KB</td>
</tr>
</tbody>
</table>
Related topics

- Read the other tutorials in this series.
- Visit the UDDI site.
- You can find specifications for all three versions, including XML schema documents.
- For information about different methods of using WSDL with UDDI registries, see these articles in the developerWorks series "Understanding WSDL in a UDDI registry:"
  - Part 1: How to publish and find WSDL service descriptions (September 2001)
  - Part 2: Usage scenarios in publishing and finding WSDL service descriptions (September 2001)
- Read more about UDDI4J: UDDI4J: Matchmaking for web services.
- Download Apache Geronimo.
- To create your own UDDI registry, download Apache jUDDI.
- Download UDDI4J.
- Download Apache Axis 1.3.
- Download Java 1.4.2. Geronimo will not run properly with Java 1.5.
- UDDI4Py is a Python package that allows the sending of requests to and processing of responses from the UDDI Version 2 APIs.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)