

# Java API for XML Registries

**T**HE Java API for XML Registries (JAXR) provides a uniform and standard Java API for accessing various kinds of XML registries.

After providing a brief overview of JAXR, this chapter describes how to implement a JAXR client to publish an organization and its web services to a registry and to query a registry to find organizations and services. Finally, it explains how to run the examples provided with this tutorial and offers links to more information on JAXR.

## Overview of JAXR

This section provides a brief overview of JAXR. It covers the following topics:

- What is a registry?
- What is JAXR?
- JAXR architecture

## What Is a Registry?

An XML *registry* is an infrastructure that enables the building, deployment, and discovery of web services. It is a neutral third party that facilitates dynamic and



loosely coupled business-to-business (B2B) interactions. A registry is available to organizations as a shared resource, often in the form of a web-based service.

Currently there are a variety of specifications for XML registries. These include

- The ebXML Registry and Repository standard, which is sponsored by the Organization for the Advancement of Structured Information Standards (OASIS) and the United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport (U.N./CEFACT)
- The Universal Description, Discovery, and Integration (UDDI) project, which is being developed by a vendor consortium

A *registry provider* is an implementation of a business registry that conforms to a specification for XML registries.

## What Is JAXR?

JAXR enables Java software programmers to use a single, easy-to-use abstraction API to access a variety of XML registries. A unified JAXR information model describes content and metadata within XML registries.

JAXR gives developers the ability to write registry client programs that are portable across various target registries. JAXR also enables value-added capabilities beyond those of the underlying registries.

The current version of the JAXR specification includes detailed bindings between the JAXR information model and both the ebXML Registry and the UDDI version 2 specifications. You can find the latest version of the specification at

<http://java.sun.com/xml/downloads/jaxr.html>

At this release of the Application Server, the JAXR provider implements the level 0 capability profile defined by the JAXR specification. This level allows access to both UDDI and ebXML registries at a basic level. At this release, the JAXR provider supports access only to UDDI version 2 registries.

Currently no public UDDI registries exist. However, you can use the Java WSDP Registry Server, a private UDDI version 2 registry that comes with release 1.5 of the Java Web Services Developer Pack (Java WSDP).

Service Registry, an ebXML registry and repository with a JAXR provider, is available as part of the Sun Java Enterprise System.

## JAXR Architecture

The high-level architecture of JAXR consists of the following parts:

- *A JAXR client*: This is a client program that uses the JAXR API to access a business registry via a JAXR provider.
- *A JAXR provider*: This is an implementation of the JAXR API that provides access to a specific registry provider or to a class of registry providers that are based on a common specification.

A JAXR provider implements two main packages:

- `javax.xml.registry`, which consists of the API interfaces and classes that define the registry access interface.
- `javax.xml.registry.infomodel`, which consists of interfaces that define the information model for JAXR. These interfaces define the types of objects that reside in a registry and how they relate to each other. The basic interface in this package is the `RegistryObject` interface. Its subinterfaces include `Organization`, `Service`, and `ServiceBinding`.

The most basic interfaces in the `javax.xml.registry` package are

- `Connection`. The `Connection` interface represents a client session with a registry provider. The client must create a connection with the JAXR provider in order to use a registry.
- `RegistryService`. The client obtains a `RegistryService` object from its connection. The `RegistryService` object in turn enables the client to obtain the interfaces it uses to access the registry.

The primary interfaces, also part of the `javax.xml.registry` package, are

- `BusinessQueryManager`, which allows the client to search a registry for information in accordance with the `javax.xml.registry.infomodel` interfaces. An optional interface, `DeclarativeQueryManager`, allows the client to use SQL syntax for queries. (The implementation of JAXR in the Application Server does not implement `DeclarativeQueryManager`.)
- `BusinessLifeCycleManager`, which allows the client to modify the information in a registry by either saving it (updating it) or deleting it.

When an error occurs, JAXR API methods throw a `JAXRException` or one of its subclasses.

Many methods in the JAXR API use a `Collection` object as an argument or a returned value. Using a `Collection` object allows operations on several registry objects at a time.

Figure 6–1 illustrates the architecture of JAXR. In the Application Server, a JAXR client uses the capability level 0 interfaces of the JAXR API to access the JAXR provider. The JAXR provider in turn accesses a registry. The Application Server supplies a JAXR provider for UDDI registries.

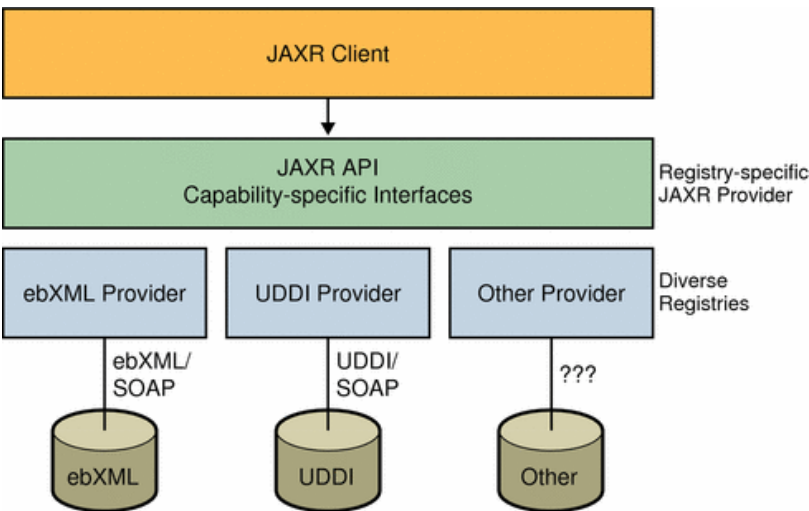


Figure 6–1 JAXR Architecture

# Implementing a JAXR Client

This section describes the basic steps to follow in order to implement a JAXR client that can perform queries and updates to a UDDI registry. A JAXR client is a client program that can access registries using the JAXR API. This section covers the following topics:

- Establishing a connection
- Querying a registry
- Managing registry data
- Using taxonomies in JAXR clients

This tutorial does not describe how to implement a JAXR provider. A JAXR provider provides an implementation of the JAXR specification that allows access to

an existing registry provider, such as a UDDI or ebXML registry. The implementation of JAXR in the Application Server itself is an example of a JAXR provider.

The Application Server provides JAXR in the form of a resource adapter using the Java EE Connector architecture. The resource adapter is in the directory `<JAVAEE_HOME>/lib/install/applications/jaxr-ra` ( `<JAVAEE_HOME>` is the directory where the Application Server is installed.)

This tutorial includes several client examples, which are described in Running the Client Examples (page 199), and a Java EE application example, described in Using JAXR Clients in Java EE Applications (page 206). The examples are in the directory `<INSTALL>/javaeetutorial5/examples/jaxr/` ( `<INSTALL>` is the directory where you installed the tutorial bundle.) Each example directory has a `build.xml` file (which refers to a `targets.xml` file) and a `build.properties` file in the directory `<INSTALL>/javaeetutorial5/examples/jaxr/common/`.

## Establishing a Connection

The first task a JAXR client must complete is to establish a connection to a registry. Establishing a connection involves the following tasks:

- Preliminaries: Getting access to a registry
- Creating or looking up a connection factory
- Creating a connection
- Setting connection properties
- Obtaining and using a `RegistryService` object

### Preliminaries: Getting Access to a Registry

To use the Java WSDP Registry Server, a private UDDI version 2 registry, you need to download and install Java WSDP 1.5 and then to install the Registry Server in the Application Server.

To download Java WSDP 1.5, perform these steps:

1. Go to the following URL:  
`http://java.sun.com/webservices/downloads/1.5/index.html`
2. Under Java Web Services Developer Pack v1.5, click Download.

3. On the Login page, click the Download link. (You do not have to log in.)
4. Select the Accept radio button to accept the license agreement.
5. Click the download arrow for your platform (Solaris or Windows).
6. Choose the directory where you will download Java WSDP.

Install Java WSDP as follows:

1. Go to the directory where you downloaded Java WSDP 1.5.
2. Run the Java WSDP installer. You can follow the instructions that are linked to from <http://java.sun.com/webservices/downloads/1.5/index.html>, although these instructions refer to a newer version of Java WSDP.
3. On the Select a Web Container page of the installer, select No Web Container.
4. Choose a directory where you will install Java WSDP.
5. Select either a Typical or a Custom installation. If you select Custom, remove the check marks from every checkbox you can except Java WSDP Registry Server. (You cannot remove the check marks from JAXB, JAXP, JAXR, or SAAJ; these technologies are required.)

After the installation completes, install the Registry Server in the Application Server as follows:

1. Stop the Application Server if it is running.
2. Copy the two WAR files in the directory `<JWSDP_HOME>/registry-server/webapps`, `RegistryServer.war` and `Xindice.war`, to the following directory:  
`<JAVAAE_HOME>/domains/domain1/autodeploy`
3. Start the Application Server.

Any user of a JAXR client can perform queries on a registry. To add data to the registry or to update registry data, however, a user must obtain permission from the registry to access it.

To add or update data in the Java WSDP Registry Server, you can use the default user name and password, `testuser` and `testuser`.

## Obtaining a Connection Factory

A client creates a connection from a connection factory. A JAXR provider can supply one or more preconfigured connection factories. Clients can obtain these factories by using resource injection.

At this release of the Application Server, JAXR supplies a connection factory through the JAXR RA, but you need to use a connector resource whose JNDI name is `eis/JAXR` to access this connection factory from a Java EE application. To inject this resource in a Java EE component, use code like the following:

```
import javax.annotation.Resource;.*;
import javax.xml.registry.ConnectionFactory;
...
@Resource(mappedName="eis/JAXR")
public ConnectionFactory factory;
```

Later in this chapter you will learn how to create this connector resource.

To use JAXR in a stand-alone client program, you must create an instance of the abstract class `ConnectionFactory` :

```
import javax.xml.registry.ConnectionFactory;
...
ConnectionFactory connFactory =
    ConnectionFactory.newInstance();
```

## Creating a Connection

To create a connection, a client first creates a set of properties that specify the URL or URLs of the registry or registries being accessed. For example, the following code provides the URLs of the query service and publishing service for a hypothetical registry. (There should be no line break in the strings.)

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
    "http://localhost:8080/RegistryServer/");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
    "http://localhost:8080/RegistryServer/");
```

With the Application Server implementation of JAXR, if the client is accessing a registry that is outside a firewall, it must also specify proxy host and port information for the network on which it is running. For queries it may need to specify



only the HTTP proxy host and port; for updates it must specify the HTTPS proxy host and port.

```
props.setProperty("com.sun.xml.registry.http.proxyHost",
    "myhost.mydomain");
props.setProperty("com.sun.xml.registry.http.proxyPort",
    "8080");
props.setProperty("com.sun.xml.registry.https.proxyHost",
    "myhost.mydomain");
props.setProperty("com.sun.xml.registry.https.proxyPort",
    "8080");
```

The client then sets the properties for the connection factory and creates the connection:

```
connFactory.setProperties(props);
Connection connection = connFactory.createConnection();
```

The `makeConnection` method in the sample programs shows the steps used to create a JAXR connection.

## Setting Connection Properties

The implementation of JAXR in the Application Server allows you to set a number of properties on a JAXR connection. Some of these are standard properties defined in the JAXR specification. Other properties are specific to the implementation of JAXR in the Application Server. Tables 6–1 and 6–2 list and describe these properties.

**Table 6–1** Standard JAXR Connection Properties

Property Name and Description	Data Type Default Value	
javax.xml.registry.queryManagerURL		
Specifies the URL of the query manager service within the target registry provider.	String	None
javax.xml.registry.lifeCycleManagerURL		
Specifies the URL of the life-cycle manager service within the target registry provider (for registry updates).	String	Same as the specified queryManagerURL value

Table 6–1 Standard JAXR Connection Properties (Continued)

Property Name and Description	Data Type	Default Value
<code>javax.xml.registry.semanticEquivalences</code>  Specifies semantic equivalences of concepts as one or more tuples of the ID values of two equivalent concepts separated by a comma. The tuples are separated by vertical bars: <code>id1,id2 id3,id4</code>	String	None
<code>javax.xml.registry.security.authenticationMethod</code>  Provides a hint to the JAXR provider on the authentication method to be used for authenticating with the registry provider.	String	None; UDDI_GET_AUTHTOKEN is the only supported value
<code>javax.xml.registry.uddi.maxRows</code>  The maximum number of rows to be returned by find operations. Specific to UDDI providers.	String	100
<code>javax.xml.registry.postalAddressScheme</code>  The ID of a <code>ClassificationScheme</code> to be used as the default postal address scheme. See <i>Specifying Postal Addresses</i> (page 197) for an example.	String	None

Table 6–2 Implementation-Specific JAXR Connection Properties

Property Name and Description	Data Type	Default Value
<code>com.sun.xml.registry.http.proxyHost</code>  Specifies the HTTP proxy host to be used for accessing external registries.	String	None
<code>com.sun.xml.registry.http.proxyPort</code>  Specifies the HTTP proxy port to be used for accessing external registries; usually 8080.	String	None

Table 6–2 Implementation-Specific JAXR Connection Properties (Continued)

Property Name and Description	Data Type	Default Value
com.sun.xml.registry.https.proxyHost	String	Same as HTTP proxy host value
Specifies the HTTPS proxy host to be used for accessing external registries.		
com.sun.xml.registry.https.proxyPort	String	Same as HTTP proxy port value
Specifies the HTTPS proxy port to be used for accessing external registries; usually 8080.		
com.sun.xml.registry.http.proxyUserName	String	None
Specifies the user name for the proxy host for HTTP proxy authentication, if one is required.		
com.sun.xml.registry.http.proxyPassword	String	None
Specifies the password for the proxy host for HTTP proxy authentication, if one is required.		
com.sun.xml.registry.useCache	Boolean, passed in as String	True
Tells the JAXR implementation to look for registry objects in the cache first and then to look in the registry if not found.		
com.sun.xml.registry.userTaxonomyFile-names	String	None
For details on setting this property, see Defining a Taxonomy (page 194).		

Obtaining and Using a RegistryService Object

After creating the connection, the client uses the connection to obtain a RegistryService object and then the interface or interfaces it will use:

```
RegistryService rs = connection.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();
BusinessLifeCycleManager blcm =
    rs.getBusinessLifeCycleManager();
```

Typically, a client obtains both a `BusinessQueryManager` object and a `BusinessLifeCycleManager` object from the `RegistryService` object. If it is using the registry for simple queries only, it may need to obtain only a `BusinessQueryManager` object.

## Querying a Registry

The simplest way for a client to use a registry is to query it for information about the organizations that have submitted data to it. The `BusinessQueryManager` interface supports a number of find methods that allow clients to search for data using the JAXR information model. Many of these methods return a `BulkResponse` (a collection of objects) that meets a set of criteria specified in the method arguments. The most useful of these methods are as follows:

- `findOrganizations`, which returns a list of organizations that meet the specified criteria—often a name pattern or a classification within a classification scheme
- `findServices`, which returns a set of services offered by a specified organization
- `findServiceBindings`, which returns the *service bindings* (information about how to access the service) that are supported by a specified service

The `JAXRQuery` program illustrates how to query a registry by organization name and display the data returned. The `JAXRQueryByNAICSClassification` and `JAXRQueryByWSDLClassification` programs illustrate how to query a registry using classifications. All JAXR providers support at least the following taxonomies for classifications:

- The North American Industry Classification System (NAICS). See <http://www.census.gov/epcd/www/naics.html> for details.
- The Universal Standard Products and Services Classification (UNSPSC). See <http://www.eccma.org/unspsc/> for details.
- The ISO 3166 country codes classification system maintained by the International Organization for Standardization (ISO). See <http://www.iso.org/iso/en/prods-services/iso3166ma/index.html> for details.

The following sections describe how to perform some common queries:

- Finding organizations by name
- Finding organizations by classification

- Finding services and service bindings

## Finding Organizations by Name

To search for organizations by name, you normally use a combination of find qualifiers (which affect sorting and pattern matching) and name patterns (which specify the strings to be searched). The `findOrganizations` method takes a collection of `findQualifier` objects as its first argument and takes a collection of `namePattern` objects as its second argument. The following fragment shows how to find all the organizations in the registry whose names begin with a specified string, `qString`, and sort them in alphabetical order.

```
// Define find qualifiers and name patterns
Collection<String> findQualifiers = new ArrayList<String>();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
Collection<String> namePatterns = new ArrayList<String>();
namePatterns.add(qString);

// Find orgs whose names begin with qString
BulkResponse response =
    bqm.findOrganizations(findQualifiers, namePatterns, null,
        null, null, null);
Collection orgs = response.getCollection();
```

The last four arguments to `findOrganizations` allow you to search using other criteria than the name: classifications, specification concepts, external identifiers, or external links. Finding Organizations by Classification (page 183) describes searching by classification and by specification concept. The other searches are less common and are not described in this tutorial.

A client can use percent signs ( `%` ) to specify that the query string can occur anywhere within the organization name. For example, the following code fragment performs a case-sensitive search for organizations whose names contain `qString` :

```
Collection<String> findQualifiers = new ArrayList<String>();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
Collection<String> namePatterns = new ArrayList<String>();
namePatterns.add("%" + qString + "%");

// Find orgs with names that contain qString
```

```
BulkResponse response =
    bqm.findOrganizations(findQualifiers, namePatterns, null,
        null, null, null);
Collection orgs = response.getCollection();
```

## Finding Organizations by Classification

To find organizations by classification, you establish the classification within a particular classification scheme and then specify the classification as an argument to the `findOrganizations` method.

The following code fragment finds all organizations that correspond to a particular classification within the NAICS taxonomy. (You can find the NAICS codes at <http://www.census.gov/epcd/naics/naicscod.txt>.) The NAICS taxonomy has a well-known universally unique identifier (UUID) that is defined by the UDDI specification. The `getRegistryObject` method finds an object based upon its key. (See *Creating an Organization*, page 186 for more information about keys)

```
String uuid_naics =
    "uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2";
ClassificationScheme cScheme =
    (ClassificationScheme) bqm.getRegistryObject(uuid_naics,
        LifecycleManager.CLASSIFICATION_SCHEME);
InternationalString sn = blcm.createInternationalString(
    "All Other Specialty Food Stores");
String sv = "445299";
Classification classification =
    blcm.createClassification(cScheme, sn, sv);
Collection<Classification> classifications =
    new ArrayList<Classification>();
classifications.add(classification);
BulkResponse response = bqm.findOrganizations(null, null,
    classifications, null, null, null);
Collection orgs = response.getCollection();
```

You can also use classifications to find organizations that offer services based on technical specifications that take the form of WSDL (Web Services Description Language) documents. In JAXR, a *concept* is used as a proxy to hold the information about a specification. The steps are a little more complicated than in the preceding example, because the client must first find the specification concepts and then find the organizations that use those concepts.

The following code fragment finds all the WSDL specification instances used within a given registry. You can see that the code is similar to the NAICS query

code except that it ends with a call to `findConcepts` instead of `findOrganizations` .

```
String schemeName = "uddi-org:types";
ClassificationScheme uddiOrgTypes =
    bqm.findClassificationSchemeByName(null, schemeName);

/*
 * Create a classification, specifying the scheme
 * and the taxonomy name and value defined for WSDL
 * documents by the UDDI specification.
 */
Classification wsdlSpecClassification =
    blcm.createClassification(uddiOrgTypes, "wsdlSpec",
        "wsdlSpec");

Collection<Classification> classifications =
    new ArrayList<Classification>();
classifications.add(wsdlSpecClassification);

// Find concepts
BulkResponse br = bqm.findConcepts(null, null,
    classifications, null, null);
```

To narrow the search, you could use other arguments of the `findConcepts` method (search qualifiers, names, external identifiers, or external links).

The next step is to go through the concepts, find the WSDL documents they correspond to, and display the organizations that use each document:

```
// Display information about the concepts found
Collection specConcepts = br.getCollection();
Iterator iter = specConcepts.iterator();
if (!iter.hasNext()) {
    System.out.println("No WSDL specification concepts found");
} else {
    while (iter.hasNext()) {
        Concept concept = (Concept) iter.next();

        String name = getName(concept);

        Collection links = concept.getExternalLinks();
        System.out.println("\nSpecification Concept:\n\tName: " +
            name + "\n\tKey: " + concept.getKey().getId() +
            "\n\tDescription: " + getDescription(concept));
        if (links.size() > 0) {
            ExternalLink link =
```

```

        (ExternalLink) links.iterator().next();
        System.out.println("\tURL of WSDL document: " +
            link.getExternalURI() + "");
    }

    // Find organizations that use this concept
    Collection<Concept> specConcepts1 =
        new ArrayList<Concept>();
    specConcepts1.add(concept);
    br = bqm.findOrganizations(null, null, null,
        specConcepts1, null, null);

    // Display information about organizations
    ...
}
}

```

If you find an organization that offers a service you wish to use, you can invoke the service using JAX-WS.

## Finding Services and Service Bindings

After a client has located an organization, it can find that organization's services and the service bindings associated with those services.

```

Iterator orgIter = orgs.iterator();
while (orgIter.hasNext()) {
    Organization org = (Organization) orgIter.next();
    Collection services = org.getServices();
    Iterator svcIter = services.iterator();
    while (svcIter.hasNext()) {
        Service svc = (Service) svcIter.next();
        Collection serviceBindings =
            svc.getServiceBindings();
        Iterator sbIter = serviceBindings.iterator();
        while (sbIter.hasNext()) {
            ServiceBinding sb =
                (ServiceBinding) sbIter.next();
        }
    }
}

```



## Managing Registry Data

If a client has authorization to do so, it can submit data to a registry, modify it, and remove it. It uses the `BusinessLifeCycleManager` interface to perform these tasks.

Registries usually allow a client to modify or remove data only if the data is being modified or removed by the same user who first submitted the data.

Managing registry data involves the following tasks:

- Getting authorization from the registry
- Creating an organization
- Adding classifications
- Adding services and service bindings to an organization
- Publishing an organization
- Publishing a specification concept
- Removing data from the registry

## Getting Authorization from the Registry

Before it can submit data, the client must send its user name and password to the registry in a set of *credentials*. The following code fragment shows how to do this.

```
String username = "testuser";
String password = "testuser";

// Get authorization from the registry
PasswordAuthentication passwdAuth =
    new PasswordAuthentication(username,
        password.toCharArray());

HashSet<PasswordAuthentication> creds =
    new HashSet<PasswordAuthentication>();
creds.add(passwdAuth);
connection.setCredentials(creds);
```

## Creating an Organization

The client creates the organization and populates it with data before publishing it.

An `Organization` object is one of the more complex data items in the JAXR API. It normally includes the following:

- A `Name` object.
- A `Description` object.
- A `Key` object, representing the ID by which the organization is known to the registry. This key is created by the registry, not by the user, and is returned after the organization is submitted to the registry.
- A `PrimaryContact` object, which is a `User` object that refers to an authorized user of the registry. A `User` object normally includes a `PersonName` object and collections of `TelephoneNumber`, `EmailAddress`, and `PostalAddress` objects.
- A collection of `Classification` objects.
- `Service` objects and their associated `ServiceBinding` objects.

For example, the following code fragment creates an organization and specifies its name, description, and primary contact. When a client creates an organization to be published to a UDDI registry, it does not include a key; the registry returns the new key when it accepts the newly created organization. The `blcm` object in the following code fragment is the `BusinessLifeCycleManager` object returned in *Obtaining and Using a RegistryService Object* (page 180). An `InternationalString` object is used for string values that may need to be localized.

```
// Create organization name and description
InternationalString s =
    blcm.createInternationalString("The Coffee Break");
Organization org = blcm.createOrganization(s);
s = blcm.createInternationalString("Purveyor of the " +
    "finest coffees. Established 1950");
org.setDescription(s);

// Create primary contact, set name
User primaryContact = blcm.createUser();
PersonName pName = blcm.createPersonName("Jane Doe");
primaryContact.setPersonName(pName);

// Set primary contact phone number
TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setNumber("(800) 555-1212");
Collection<TelephoneNumber> phoneNums =
    new ArrayList<TelephoneNumber>();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);
```

```
// Set primary contact email address
EmailAddress emailAddress =
    blcm.createEmailAddress("jane.doe@TheCoffeeBreak.com");
Collection<EmailAddress> emailAddresses =
    new ArrayList<EmailAddress>();
emailAddresses.add(emailAddress);
primaryContact.setEmailAddresses(emailAddresses);

// Set primary contact for organization
org.setPrimaryContact(primaryContact);
```

## Adding Classifications

Organizations commonly belong to one or more classifications based on one or more classification schemes (taxonomies). To establish a classification for an organization using a taxonomy, the client first locates the taxonomy it wants to use. It uses the `BusinessQueryManager` to find the taxonomy. The `findClassificationSchemeByName` method takes a set of `FindQualifier` objects as its first argument, but this argument can be null.

```
// Set classification scheme to NAICS
ClassificationScheme cScheme =
    bqm.findClassificationSchemeByName(null,
        "ntis-gov:naics:1997");
```

The client then creates a classification using the classification scheme and a concept (a taxonomy element) within the classification scheme. For example, the following code sets up a classification for the organization within the NAICS taxonomy. The second and third arguments of the `createClassification` method are the name and the value of the concept.

```
// Create and add classification
InternationalString sn =
    blcm.createInternationalString(
        "All Other Specialty Food Stores");
String sv = "445299";
Classification classification =
    blcm.createClassification(cScheme, sn, sv);
Collection<Classification> classifications =
    new ArrayList<Classification>();
classifications.add(classification);
org.addClassifications(classifications);
```

Services also use classifications, so you can use similar code to add a classification to a `Service` object.

## Adding Services and Service Bindings to an Organization

Most organizations add themselves to a registry in order to offer services, so the JAXR API has facilities to add services and service bindings to an organization.

Like an `Organization` object, a `Service` object has a name, a description, and a unique key that is generated by the registry when the service is registered. It may also have classifications associated with it.

A service also commonly has *service bindings*, which provide information about how to access the service. A `ServiceBinding` object normally has a description, an access URI, and a specification link, which provides the linkage between a service binding and a technical specification that describes how to use the service by using the service binding.

The following code fragment shows how to create a collection of services, add service bindings to a service, and then add the services to the organization. It specifies an access URI but not a specification link. Because the access URI is not real and because JAXR by default checks for the validity of any published URI, the binding sets its `validateURI` property to false.

```
// Create services and service
Collection<Service> services = new ArrayList<Service>();
InternationalString s =
    blcm.createInternationalString("My Service Name");
Service service = blcm.createService(s);
s = blcm.createInternationalString("My Service Description");
service.setDescription(is);

// Create service bindings
Collection<ServiceBinding> serviceBindings =
    new ArrayList<ServiceBinding>();
ServiceBinding binding = blcm.createServiceBinding();
s = blcm.createInternationalString("My Service Binding " +
    "Description");
binding.setDescription(is);
// allow us to publish a fictitious URI without an error
binding.setValidateURI(false);
binding.setAccessURI("http://TheCoffeeBreak.com:8080/sb/");
serviceBindings.add(binding);

// Add service bindings to service
service.addServiceBindings(serviceBindings);
```

```
// Add service to services, then add services to organization
services.add(service);
org.addServices(services);
```

## Publishing an Organization

The primary method a client uses to add or modify organization data is the `saveOrganizations` method, which creates one or more new organizations in a registry if they did not exist previously. If one of the organizations exists but some of the data have changed, the `saveOrganizations` method updates and replaces the data.

After a client populates an organization with the information it wants to make public, it saves the organization. The registry returns the key in its response, and the client retrieves it.

```
// Add organization and submit to registry
// Retrieve key if successful
Collection<Organization> orgs = new ArrayList<Organization>();
orgs.add(org);
BulkResponse response = blcm.saveOrganizations(orgs);
Collection exceptions = response.getException();
if (exceptions == null) {
    System.out.println("Organization saved");

    Collection keys = response.getCollection();
    Iterator keyIter = keys.iterator();
    if (keyIter.hasNext()) {
        Key orgKey = (Key) keyIter.next();
        String id = orgKey.getId();
        System.out.println("Organization key is " + id);
    }
}
```

## Publishing a Specification Concept

A service binding can have a technical specification that describes how to access the service. An example of such a specification is a WSDL document. To publish the location of a service's specification (if the specification is a WSDL document), you create a `Concept` object and then add the URL of the WSDL document to the `Concept` object as an `ExternalLink` object. The following code fragment shows how to create a concept for the WSDL document associated with the simple web service example in *Creating a Simple Web Service and Cli-*

ent with JAX-WS (page xvi). First, you call the `createConcept` method to create a concept named `HelloConcept`. After setting the description of the concept, you create an external link to the URL of the `Hello` service's WSDL document, and then add the external link to the concept.

```
Concept specConcept =
    blcm.createConcept(null, "HelloConcept", "");
InternationalString s =
    blcm.createInternationalString(
        "Concept for Hello Service");
specConcept.setDescription(s);
ExternalLink wsdlLink =
    blcm.createExternalLink(
        "http://localhost:8080/hello-jaxws/hello?WSDL",
        "Hello WSDL document");
specConcept.addExternalLink(wsdlLink);
```

Next, you classify the `Concept` object as a WSDL document. To do this for a UDDI registry, you search the registry for the well-known classification scheme `uddi-org:types`, using its key ID. (The UDDI term for a classification scheme is *tModel*.) Then you create a classification using the name and value `wsdlSpec`. Finally, you add the classification to the concept.

```
String uuid_types =
    "uuid:c1acf26d-9672-4404-9d70-39b756e62ab4";
ClassificationScheme uddiOrgTypes =
    (ClassificationScheme) bqm.getRegistryObject(uuid_types,
        LifecycleManager.CLASSIFICATION_SCHEME);

Classification wsdlSpecClassification =
    blcm.createClassification(uddiOrgTypes,
        "wsdlSpec", "wsdlSpec");
specConcept.addClassification(wsdlSpecClassification);
```

Finally, you save the concept using the `saveConcepts` method, similarly to the way you save an organization:

```
Collection<Concept> concepts = new ArrayList<Concept>();
concepts.add(specConcept);
BulkResponse concResponse = blcm.saveConcepts(concepts);
```

After you have published the concept, you normally add the concept for the WSDL document to a service binding. To do this, you can retrieve the key for the

concept from the response returned by the `saveConcepts` method; you use a code sequence very similar to that of finding the key for a saved organization.

```
String conceptKeyId = null;
Collection concExceptions = concResponse.getExceptions();
Key concKey = null;
if (concExceptions == null) {
    System.out.println("WSDL Specification Concept saved");

    Collection keys = concResponse.getCollection();
    Iterator keyIter = keys.iterator();
    if (keyIter.hasNext()) {
        concKey = (Key) keyIter.next();
        conceptKeyId = concKey.getId();
        System.out.println("Concept key is " + conceptKeyId);
    }
}
```

Then you can call the `getRegistryObject` method to retrieve the concept from the registry:

```
Concept specConcept =
    (Concept) bqm.getRegistryObject(conceptKeyId,
        LifeCycleManager.CONCEPT);
```

Next, you create a `SpecificationLink` object for the service binding and set the concept as the value of its `SpecificationObject`:

```
SpecificationLink specLink =
    blcm.createSpecificationLink();
specLink.setSpecificationObject(specConcept);
binding.addSpecificationLink(specLink);
```

Now when you publish the organization with its service and service bindings, you have also published a link to the WSDL document. Now the organization can be found via queries such as those described in *Finding Organizations by Classification* (page 183).

If the concept was published by someone else and you don't have access to the key, you can find it using its name and classification. The code looks very similar to the code used to search for a WSDL document in *Finding Organizations by*

Classification (page 183), except that you also create a collection of name patterns and include that in your search. Here is an example:

```
// Define name pattern
Collection namePatterns = new ArrayList();
namePatterns.add("HelloConcept");

BulkResponse br = bqm.findConcepts(null, namePatterns,
    classifications, null, null);
```

## Removing Data from the Registry

A registry allows you to remove from it any data that you have submitted to it. You use the key returned by the registry as an argument to one of the `BusinessLifeCycleManager` delete methods: `deleteOrganizations`, `deleteServices`, `deleteServiceBindings`, `deleteConcepts`, and others.

The `JAXRDelete` sample program deletes the organization created by the `JAXR-Publish` program. It deletes the organization that corresponds to a specified key string and then displays the key again so that the user can confirm that it has deleted the correct one.

```
String id = key.getId();
System.out.println("Deleting organization with id " + id);
Collection<Key> keys = new ArrayList<Key>();
keys.add(key);
BulkResponse response = blcm.deleteOrganizations(keys);
Collection exceptions = response.getException();
if (exceptions == null) {
    System.out.println("Organization deleted");
    Collection retKeys = response.getCollection();
    Iterator keyIter = retKeys.iterator();
    Key orgKey = null;
    if (keyIter.hasNext()) {
        orgKey = (Key) keyIter.next();
        id = orgKey.getId();
        System.out.println("Organization key was " + id);
    }
}
```

A client can use a similar mechanism to delete concepts, services, and service bindings.



## Using Taxonomies in JAXR Clients

In the JAXR API, a taxonomy is represented by a `ClassificationScheme` object. This section describes how to use the implementation of JAXR in the Application Server to perform these tasks:

- To define your own taxonomies
- To specify postal addresses for an organization

### Defining a Taxonomy

The JAXR specification requires that a JAXR provider be able to add user-defined taxonomies for use by JAXR clients. The mechanisms clients use to add and administer these taxonomies are implementation-specific.

The implementation of JAXR in the Application Server uses a simple file-based approach to provide taxonomies to the JAXR client. These files are read at run-time, when the JAXR provider starts up.

The taxonomy structure for the Application Server is defined by the JAXR Pre-defined Concepts DTD, which is declared both in the file `jaxrconcepts.dtd` and, in XML schema form, in the file `jaxrconcepts.xsd`. The file `jaxrconcepts.xml` contains the taxonomies for the implementation of JAXR in the Application Server. All these files are contained in the `<JAVAEE_HOME>/lib/appserv-ws.jar` file. This JAR file also includes files that define the well-known taxonomies used by the implementation of JAXR in the Application Server: `naics.xml`, `iso3166.xml`, and `unspsc.xml`.

The entries in the `jaxrconcepts.xml` file look like this:

```
<PredefinedConcepts>
  <JAXRClassificationScheme id="schId" name="schName">
    <JAXRConcept id="schId / conCode" name="conName"
      parent="parentId" code="conCode">
    </JAXRConcept>
    ...
  </JAXRClassificationScheme>
</PredefinedConcepts>
```

The taxonomy structure is a containment-based structure. The element `PredefinedConcepts` is the root of the structure and must be present. The `Pre-JAXR-ClassificationScheme` element is the parent of the structure, and the

JAXRConcept elements are children and grandchildren. A JAXRConcept element may have children, but it is not required to do so.

In all element definitions, attribute order and case are significant.

To add a user-defined taxonomy, follow these steps.

1. Publish the JAXRClassificationScheme element for the taxonomy as a ClassificationScheme object in the registry that you will be accessing. To publish a ClassificationScheme object, you must set its name. You also give the scheme a classification within a known classification scheme such as uddi-org:types. In the following code fragment, the name is the first argument of the LifecycleManager.createClassificationScheme method call.

```

InternationalString sn =
    blcm.createInternationalString("MyScheme");
InternationalString sd = blcm.createInternationalString(
    "A Classification Scheme");
ClassificationScheme postalScheme =
    blcm.createClassificationScheme(sn, sd);
String uuid_types =
    "uuid:c1acf26d-9672-4404-9d70-39b756e62ab4";
ClassificationScheme uddiOrgTypes =
    (ClassificationScheme) bqm.getRegistryObject(uuid_types,
    LifecycleManager.CLASSIFICATION_SCHEME);
if (uddiOrgTypes != null) {
    Classification classification =
        blcm.createClassification(uddiOrgTypes,
        "postalAddress", "postalAddress");
    postalScheme.addClassification(classification);
    InternationalString ld =
        blcm.createInternationalString("My Scheme");
    ExternalLink externalLink =
        blcm.createExternalLink(
            "http://www.mycom.com/myscheme.xml", ld);
    postalScheme.addExternalLink(externalLink);
    Collection<ClassificationScheme> schemes =
        new ArrayList<ClassificationScheme>();
    schemes.add(cScheme);
    BulkResponse br =
        blcm.saveClassificationSchemes(schemes);
}

```

The BulkResponse object returned by the saveClassificationSchemes method contains the key for the classification scheme, which you need to retrieve:

```
if (br.getStatus() == JAXRResponse.STATUS_SUCCESS) {
    System.out.println("Saved ClassificationScheme");
    Collection schemeKeys = br.getCollection();
    Iterator keysIter = schemeKeys.iterator();
    while (keysIter.hasNext()) {
        Key key = (Key) keysIter.next();
        System.out.println("The postalScheme key is " +
            key.getId());
        System.out.println("Use this key as the scheme" +
            " uuid in the taxonomy file");
    }
}
```

2. In an XML file, define a taxonomy structure that is compliant with the JAXR Predefined Concepts DTD. Enter the `ClassificationScheme` element in your taxonomy XML file by specifying the returned key ID value as the `id` attribute and the name as the `name` attribute. For the foregoing code fragment, for example, the opening tag for the `JAXRClassificationScheme` element looks something like this (all on one line):

```
<JAXRClassificationScheme
id="uuid:nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnnn"
name="MyScheme">
```

The `ClassificationScheme id` must be a universally unique identifier (UUID).

3. Enter each `JAXRConcept` element in your taxonomy XML file by specifying the following four attributes, in this order:
- a. `id` is the `JAXRClassificationScheme id` value, followed by a `/` separator, followed by the code of the `JAXRConcept` element.
  - b. `name` is the name of the `JAXRConcept` element.
  - c. `parent` is the immediate parent `id` (either the `ClassificationScheme id` or that of the parent `JAXRConcept`).
  - d. `code` is the `JAXRConcept` element code value.

The first `JAXRConcept` element in the `naics.xml` file looks like this (all on one line):

```
<JAXRConcept
id="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2/11"
name="Agriculture, Forestry, Fishing and Hunting"
parent="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
code="11"></JAXRConcept>
```

4. To add the user-defined taxonomy structure to the JAXR provider, specify the connection property `com.sun.xml.registry.userTaxonomyFileNames` in your client program. You set the property as follows:

```
props.setProperty
("com.sun.xml.registry.userTaxonomyFileNames",
 "c:\mydir\xxx.xml;c:\mydir\xxx2.xml");
```

Use the vertical bar ( | ) as a separator if you specify more than one file name.

## Specifying Postal Addresses

The JAXR specification defines a postal address as a structured interface with attributes for street, city, country, and so on. The UDDI specification, on the other hand, defines a postal address as a free-form collection of address lines, each of which can also be assigned a meaning. To map the JAXR `PostalAddress` format to a known UDDI address format, you specify the UDDI format as a `ClassificationScheme` object and then specify the semantic equivalences between the concepts in the UDDI format classification scheme and the comments in the JAXR `PostalAddress` classification scheme. The JAXR `PostalAddress` classification scheme is provided by the implementation of JAXR in the Application Server.

In the JAXR API, a `PostalAddress` object has the fields `streetNumber`, `street`, `city`, `state`, `postalCode`, and `country`. In the implementation of JAXR in the Application Server, these are predefined concepts in the `jaxrconcepts.xml` file, within the `ClassificationScheme` named `PostalAddressAttributes`.

To specify the mapping between the JAXR postal address format and another format, you set two connection properties:

- The `javax.xml.registry.postalAddressScheme` property, which specifies a postal address classification scheme for the connection
- The `javax.xml.registry.semanticEquivalences` property, which specifies the semantic equivalences between the JAXR format and the other format

For example, suppose you want to use a scheme named `MyPostalAddressScheme`, which you published to a registry with the UUID `uuid:f7922839-f1f7-9228-c97d-ce0b4594736c`.

```
<JAXRClassificationScheme id="uuid:f7922839-f1f7-9228-c97d-
ce0b4594736c" name="MyPostalAddressScheme">
```

First, you specify the postal address scheme using the `id` value from the `JAXR-ClassificationScheme` element (the UUID). Case does not matter:

```
props.setProperty("javax.xml.registry.postalAddressScheme",
    "uuid:f7922839-f1f7-9228-c97d-ce0b4594736c");
```

Next, you specify the mapping from the `id` of each `JAXRConcept` element in the default JAXR postal address scheme to the `id` of its counterpart in the scheme you published:

```
props.setProperty("javax.xml.registry.semanticEquivalences",
    "urn:uuid:PostalAddressAttributes/StreetNumber," +
    "uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/" +
    "StreetAddressNumber|" +
    "urn:uuid:PostalAddressAttributes/Street," +
    "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/" +
    "StreetAddress|" +
    "urn:uuid:PostalAddressAttributes/City," +
    "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/City|" +
    "urn:uuid:PostalAddressAttributes/State," +
    "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/State|" +
    "urn:uuid:PostalAddressAttributes/PostalCode," +
    "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/ZipCode|" +
    "urn:uuid:PostalAddressAttributes/Country," +
    "urn:uuid:f7922839-f1f7-9228-c97d-ce0b4594736c/Country");
```

After you create the connection using these properties, you can create a postal address and assign it to the primary contact of the organization before you publish the organization:

```
String streetNumber = "99";
String street = "Imaginary Ave. Suite 33";
String city = "Imaginary City";
String state = "NY";
String country = "USA";
String postalCode = "00000";
String type = "";
PostalAddress postAddr =
    blcm.createPostalAddress(streetNumber, street, city, state,
        country, postalCode, type);
Collection<PostalAddress> postalAddresses =
    new ArrayList<PostalAddress>();
postalAddresses.add(postAddr);
primaryContact.setPostalAddresses(postalAddresses);
```

If the postal address scheme and semantic equivalences for the query are the same as those specified for the publication, a JAXR query can then retrieve the postal address using `PostalAddress` methods. To retrieve postal addresses when you do not know what postal address scheme was used to publish them, you can retrieve them as a collection of `Slot` objects. The `JAXRQueryPostal.java` sample program shows how to do this.

In general, you can create a user-defined postal address taxonomy for any `PostalAddress` `tModels` that use the well-known categorization in the `uddi-org:types` taxonomy, which has the `tModel UUID` `uuid:c1acf26d-9672-4404-9d70-39b756e62ab4` with a value of `postalAddress`. You can retrieve the `tModel overviewDoc`, which points to the technical detail for the specification of the scheme, where the taxonomy structure definition can be found. (The JAXR equivalent of an `overviewDoc` is an `ExternalLink`.)

## Running the Client Examples

The simple client programs provided with this tutorial can be run from the command line. You can modify them to suit your needs. They allow you to specify the Java WSDP Registry Server for queries and updates. (To install the Registry Server, follow the instructions in Preliminaries: Getting Access to a Registry (page 175).

The examples, in the `<INSTALL>/javaeetutorial5/examples/jaxr/simple/src/` directory, are as follows:

- `JAXRQuery.java` shows how to search a registry for organizations.
- `JAXRQueryByNAICSClassification.java` shows how to search a registry using a common classification scheme.
- `JAXRQueryByWSDLClassification.java` shows how to search a registry for web services that describe themselves by means of a WSDL document.
- `JAXRPublish.java` shows how to publish an organization to a registry.
- `JAXRDelete.java` shows how to remove an organization from a registry.
- `JAXRSaveClassificationScheme.java` shows how to publish a classification scheme (specifically, a postal address scheme) to a registry.
- `JAXRPublishPostal.java` shows how to publish an organization with a postal address for its primary contact.
- `JAXRQueryPostal.java` shows how to retrieve postal address data from an organization.

- JAXRDeleteScheme.java shows how to delete a classification scheme from a registry.
- JAXRPublishConcept.java shows how to publish a concept for a WSDL document.
- JAXRPublishHelloOrg.java shows how to publish an organization with a service binding that refers to a WSDL document.
- JAXRDeleteConcept.java shows how to delete a concept.
- JAXRGetMyObjects.java lists all the objects that you own in a registry.

The `<INSTALL>/javaetutorial5/examples/jaxr/simple/` directory also contains the following:

- A `build.xml` file for the examples
- A `JAXRExamples.properties` file, in the `src` subdirectory, that supplies string values used by the sample programs
- A file called `postalconcepts.xml` that serves as the taxonomy file for the postal address examples

## Before You Compile the Examples

Before you compile the examples, edit the file `<INSTALL>/javaetutorial5/examples/jaxr/simple/src/JAXRExamples.properties` as follows.

1. If the Application Server where you installed the Registry Server is running on a system other than your own or if it is using a nondefault HTTP port, change the following lines:

```
query.url=http://localhost:8080/RegistryServer/
publish.url=http://localhost:8080/RegistryServer/
...
link.uri=http://localhost:8080/hello-jaxws/hello?WSDL
...
wsdlorg.svcbnd.uri=http://localhost:8080/hello-jaxws/hello
```

Specify the fully qualified host name instead of `localhost`, or change `8080` to the correct value for your system.

2. (Optional) Edit the following lines, which contain empty strings for the proxy hosts, to specify your own proxy settings. The proxy host is the system on your network through which you access the Internet; you usually specify it in your Internet browser settings.

```
## HTTP and HTTPS proxy host and port
http.proxyHost=
```

```
http.proxyPort=8080
https.proxyHost=
https.proxyPort=8080
```

The proxy ports have the value 8080, which is the usual one; change this string if your proxy uses a different port.

Your entries usually follow this pattern:

```
http.proxyHost=proxyhost.mydomain
http.proxyPort=8080
https.proxyHost=proxyhost.mydomain
https.proxyPort=8080
```

You need to specify a proxy only if you want to specify an external link or service binding that is outside your firewall.

- 3. Feel free to change any of the organization data in the remainder of the file. This data is used by the publishing and postal address examples. Try to make the organization names unusual so that queries will return relatively few results.

You can edit the `src/JAXRExamples.properties` file at any time. The `asant` targets that run the client examples will use the latest version of the file.

**Note:** Before you compile any of the examples, follow the preliminary setup instructions in Building the Examples (page xxxiii).

## Compiling the Examples

To compile the programs, go to the `<INSTALL >/javaetutorial5/examples/jaxr/simple/` directory. A `build.xml` file allows you to use the following command to compile all the examples:

```
asant
```

This command uses the default target, `build`, which performs the compilation. The `asant` tool creates a subdirectory called `build`.

## Running the Examples

You must start the Application Server in order to run the examples against the Registry Server.



## Running the JAXRPublish Example

To run the `JAXRPublish` program, use the `run-publish` target with no command-line arguments:

```
asant run-publish
```

The program output displays the string value of the key of the new organization.

After you run the `JAXRPublish` program but before you run `JAXRDelete`, you can run `JAXRQuery` to look up the organization you published.

## Running the JAXRQuery Example

To run the `JAXRQuery` example, use the `asant` target `run-query`. Specify a `query-string` argument on the command line to search the registry for organizations whose names contain that string. For example, the following command line searches for organizations whose names contain the string `"coffee"` (searching is not case-sensitive):

```
asant -Dquery-string=coffee run-query
```

## Running the JAXRQueryByNAICSClassification Example

After you run the `JAXRPublish` program, you can also run the `JAXRQueryByNAICSClassification` example, which looks for organizations that use the All Other Specialty Food Stores classification, the same one used for the organization created by `JAXRPublish`. To do so, use the `asant` target `run-query-naics`:

```
asant run-query-naics
```

## Running the JAXRDelete Example

To run the `JAXRDelete` program, specify the key string displayed by the `JAXRPublish` program as input to the `run-delete` target:

```
asant -Dkey-string=keyString run-delete
```

## Publishing a Classification Scheme

To publish organizations with postal addresses, you must first publish a classification scheme for the postal address.

To run the `JAXRSaveClassificationScheme` program, use the target `run-save-scheme` :

```
asant run-save-scheme
```

The program returns a UUID string, which you will use in the next section.

## Running the Postal Address Examples

Before you run the postal address examples, perform these steps:

1. Open the file `src/postalconcepts.xml` in an editor.
2. Wherever you see the string `uuid-from-save`, replace it with the UUID string returned by the `run-save-scheme` target (including the `uuid:` prefix).

For a given registry, you only need to publish the classification scheme and edit `postalconcepts.xml` once. After you perform those steps, you can run the `JAXRPublishPostal` and `JAXRQueryPostal` programs multiple times.

1. Run the `JAXRPublishPostal` program. Specify the string you entered in the `postalconcepts.xml` file, including the `uuid:` prefix, as input to the `run-publish-postal` target:

```
asant -Duuid-string=uuidstring run-publish-postal
```

The *uuidstring* would look something like this:

```
uuid:938d9ccd-a74a-4c7e-864a-e6e2c6822519
```

The program output displays the string value of the key of the new organization.

2. Run the `JAXRQueryPostal` program. The `run-query-postal` target specifies the `postalconcepts.xml` file in a `<sysproperty>` tag.

As input to the `run-query-postal` target, specify both a `query-string` argument and a `uuid-string` argument on the command line to search the registry for the organization published by the `run-publish-postal` target:

```
asant -Dquery-string=coffee  
-Duuid-string=uuidstring run-query-postal
```

The postal address for the primary contact will appear correctly with the JAXR `PostalAddress` methods. Any postal addresses found that use other postal address schemes will appear as `Slot` lines.

If you want to delete the organization you published, follow the instructions in Running the JAXRDelete Example (page 202).

## Deleting a Classification Scheme

To delete the classification scheme you published after you have finished using it, run the `JAXRDeleteScheme` program using the `run-delete-scheme` target:

```
asant -Duuid-string=uuidstring run-delete-scheme
```

## Publishing a Concept for a WSDL Document

To publish the location of the WSDL document for the JAX-WS `Hello` service, first deploy the service to the Application Server as described in Creating a Simple Web Service and Client with JAX-WS (page xvi).

Then run the `JAXRPublishConcept` program using the `run-publish-concept` target:

```
asant run-publish-concept
```

The program output displays the UUID string of the new specification concept, which is named `HelloConcept`. You will use this string in the next section.

After you run the `JAXRPublishConcept` program, you can run `JAXRPublishHelloOrg` to publish an organization that uses this concept.

## Publishing an Organization with a WSDL Document in Its Service Binding

To run the `JAXRPublishHelloOrg` example, use the `asant` target `run-publish-hello-org`. Specify the string returned from `JAXRPublishConcept` (including the `uuid:` prefix) as input to this target:

```
asant -Duuid-string=uuidstring run-publish-hello-org
```

The *uuidstring* would look something like this:

```
uuid:10945f5c-f2e1-0945-2f07-5897ebcfaa35
```

The program output displays the string value of the key of the new organization, which is named Hello Organization.

After you publish the organization, run the `JAXRQueryByWSDLClassification` example to search for it. To delete it, run `JAXRDelete`.

## Running the JAXRQueryByWSDLClassification Example

To run the `JAXRQueryByWSDLClassification` example, use the `asant` target `run-query-wsdl`. Specify a `query-string` argument on the command line to search the registry for specification concepts whose names contain that string. For example, the following command line searches for concepts whose names contain the string "helloconcept" (searching is not case-sensitive):

```
asant -Dquery-string=helloconcept run-query-wsdl
```

This example finds the concept and organization you published.

## Deleting a Concept

To run the `JAXRDeleteConcept` program, specify the UUID string displayed by the `JAXRPublishConcept` program as input to the `run-delete-concept` target:

```
asant -Duuid-string=uuidString run-delete-concept
```

Do not delete the concept until after you have deleted any organizations that refer to it.

## Getting a List of Your Registry Objects

To get a list of the objects you own in the registry—organizations, classification schemes, and concepts—run the `JAXRGetMyObjects` program by using the `run-get-objects` target:

```
asant run-get-objects
```

## Other Targets

To remove the `build` directory and class files, use the command

```
asant clean
```

To obtain a syntax reminder for the targets, use the command

```
asant -projecthelp
```

## Using JAXR Clients in Java EE Applications

You can create Java EE applications that use JAXR clients to access registries. This section explains how to write, compile, package, deploy, and run a Java EE application that uses JAXR to publish an organization to a registry and then query the registry for that organization. The application in this section uses two components: an application client and a stateless session bean.

The section covers the following topics:

- Coding the application client: `MyAppClient.java`
- Coding the `PubQuery` session bean
- Compiling the source files
- Starting the Application Server
- Creating JAXR resources
- Creating and packaging the application
- Deploying the application
- Running the application client

You will find the source files for this section in the directory `<INSTALL>/javaeetutorial5/examples/jaxr/clientsession`. Path names in this section are relative to this directory.

The following directory contains a built version of this application:

```
<INSTALL>/javaeetutorial5/examples/jaxr/provided-ears
```

## Coding the Application Client: MyAppClient.java

The application client class, `src/MyAppClient.java`, accesses the `PubQuery` enterprise bean's remote interface. The program calls the bean's two business methods, `executePublish` and `executeQuery`.

## Coding the PubQuery Session Bean

The `PubQuery` bean is a stateless session bean that has two business methods. The bean uses remote interfaces rather than local interfaces because it is accessed from the application client.

The remote interface, `src/PubQueryRemote.java`, declares two business methods: `executePublish` and `executeQuery`. The bean class, `src/PubQueryBean.java`, implements the `executePublish` and `executeQuery` methods and their helper methods `getName`, `getDescription`, and `getKey`. These methods are very similar to the methods of the same name in the simple examples `JAXRQuery.java` and `JAXRPublish.java`. The `executePublish` method uses information in the file `PubQueryBeanExample.properties` to create an organization named The Coffee Enterprise Bean Break. The `executeQuery` method uses the organization name, specified in the application client code, to locate this organization.

The bean class injects a `ConnectionFactory` resource. It implements a `@PostConstruct` method named `makeConnection`, which uses the `ConnectionFactory` to create the `Connection`. Finally, a `@PreDestroy` method named `endConnection` closes the `Connection`.

## Editing the Properties File

Before you compile the application, edit the `PubQueryBeanExamples.properties` file in the same way you edited the `JAXRExamples.properties` file to run the simple examples. Feel free to change any of the organization data in the file.

## Compiling the Source Files

To compile the application source files, go to the directory `<INSTALL>/javaeetutorial5/examples/jaxr/clientsession`. Use the following command:

```
asant build
```

The `build` target places the properties file and the class files in the `build` directory.

## Starting the Application Server

To run this example, you need to start the Application Server. Follow the instructions in Starting and Stopping the Application Server (page 28).

## Creating JAXR Resources

To use JAXR in a Java EE application that uses the Application Server, you need to access the JAXR resource adapter (see Implementing a JAXR Client, page 174) through a connector connection pool and a connector resource. You can create these resources in the Admin Console.

If you have not done so, start the Admin Console as described in Starting the Admin Console (page 29).

To create the connector connection pool, perform the following steps:

1. In the tree component, expand the Resources node, then expand the Connectors node.
2. Click Connector Connection Pools.
3. Click New.
4. On the General Settings page:
  - a. Type `jaxr-pool` in the Name field.
  - b. Choose `jaxr-ra` from the Resource Adapter drop-down list.
  - c. Choose `com.sun.connector.jaxr.JaxrConnectionFactory` (the only choice) from the Connection Definition drop-down list
  - d. Click Next.
5. On the next page, click Finish.

To create the connector resource, perform the following steps:

1. Under the Connectors node, click Connector Resources.
2. Click New. The Create Connector Resource page appears.
3. In the JNDI Name field, type `eis/JAXR`.
4. Choose `jaxr-pool` from the Pool Name drop-down list.
5. Click OK.

If you are in a hurry, you can create these objects by executing the following command (from the directory `<INSTALL>/javaeetutorial5/examples/jaxr/clientsession`):

```
asant create-resource
```

## Packaging the Application

The `build.xml` file in the `clientsession` directory defines Ant targets that package the `clientsession` application. To package the application, use the following command:

```
asant pack-ear
```

The `pack-ear` target depends on the `pack-client` and `pack-ejb` targets, which in turn depend on the `build` target.

The `pack-client` target creates a JAR file that contains the client class file, a manifest file, and the `PubQueryBeanExample.properties` file.

The `pack-ejb` target packages the session bean. It creates a JAR file that contains the bean class files, a manifest file, and the `PubQueryBeanExample.properties` file.

The `pack-ear` target packages the two JAR files along with an `application.xml` file. It creates a file named `clientsession.ear` in the `clientsession` directory.



## Deploying the Application

The `build.xml` file in the `clientsession` directory defines an Ant target that deploys the `clientsession.ear` file and returns a client JAR file. Use the following command:

```
asant deploy-ear
```

This command deploys the application and returns a JAR file named `client-sessionClient.jar` in the `clientsession` directory.

## Running the Application Client

To run the client, use the following command:

```
appclient -client clientsessionClient.jar
```

The program output in the terminal window looks like this:

```
To view the bean output,  
check <install_dir>/domains/domain1/logs/server.log.
```

In the server log, you will find the output from the `executePublish` and `executeQuery` methods, wrapped in logging information.

After you run the example, use the following command to undeploy the application:

```
asant undeploy-ear
```

You can use the `run-delete` target in the `simple` directory to delete the organization that was published.

## Further Information

For more information about JAXR, registries, and web services, see the following:

- Java Specification Request (JSR) 93: JAXR 1.0:  
<http://jcp.org/jsr/detail/093.jsp>
- JAXR home page:

<http://java.sun.com/xml/jaxr/>

- Universal Description, Discovery and Integration (UDDI) project:

<http://www.uddi.org/>

- ebXML:

<http://www.ebxml.org/>

- Service Registry (ebXML Registry/Repository):

<http://www.sun.com/products/soa/registry/>

- Open Source JAXR Provider for ebXML Registries:

<http://ebxmlrr.sourceforge.net/jaxr/>

- Java Platform, Enterprise Edition:

<http://java.sun.com/javaee/>

- Java Technology and XML:

<http://java.sun.com/xml/>

- Java Technology and Web Services:

<http://java.sun.com/webservices/>



