Module 2 Content

What is the JDBC API?

We all are aware of two basic facts:

- Practically every mission-critical information system relies on data stored in some sort of a database.
- Business systems usually use relational databases.

I am sure that all of you have a decent fundamental knowledge about main principles behind database theory such as:

- what is database and what is database schema;
- what is relational data model;
- what is a database table and its structural elements, a column and a row;
- what is SQL as standard language used to create, manipulate, examine, and manage relational databases.

A program must connect to a database before sending SQL commands. Each database vendor has a different interface as well as different extensions of SQL.

Open Database Connectivity protocol (ODBC) was created to standardize a mechanism accessing a database so that a program code will not depend on database connectivity specifics and will instead become transparent to it. It [ODBC] provides a consistent interface for communicating with a database and for accessing database metadata. Database vendors typically provide specific ODBC drivers to their particular database management system.

ODBC drivers are C or C++ programs. They are not platform portable nor do they have underlying support libraries or network drivers. So for Java, as a pure platform natural language, ODBC drivers are a very good match. That's why Sun Microsystems and other Java community vendors came up with Java technology as an analogy of ODBC. It is called **J**ava **D**ata**b**ase **C**onnectivity protocol or JDBC™ for short.

Java Database Connectivity (JDBC™) is a programming interface that lets developers using the Java programming language to gain access to a wide variety of databases and other data sources, either directly or through middleware.

There are two high-levels goals for the JDBC API:

1. Adherence to the following common database standards:
    - the ability to pass raw SQL statements directly to the data source;
    - JDBC driver support of the ANSI SQL92 Entry Level standard;
    - JDBC architecture that supports the functionality of common database bridges such as ODBC.
2. Keeping the API simple by:
    - providing specific APIs for standard database activities;
    - leveraging the style and benefits of the Java core classes as much as possible;
    - creating methods that map to specific functionality rather than relying on a small API with multiple meanings based on parameter value;
    - using strong typing whenever possible to provide compile-time checking.

The JDBC is a critical programming interface enabling Java with database connectivity for passing queries or data and for receiving the results of processing. These data facilities provide multi-user concurrency and transaction control, high throughput, and recovery capabilities.

The JDBC API defines a standard way of accessing a relational database from Java applications regardless of where the application is running or where the database is. In cases of web application, we can access databases using CGI programs running on web server machines. However, the Java solution is more robust, maintainable, easier to develop, and more secure. The JDBC API allows programmers to update multiple data items with a single command or even access multiple database servers within a single transaction. It also lets programmers reuse database connections - a process known as connection pooling, which makes the database access process both more effective and scalable.

The JDBC API is a "thin" object wrapper of SQL and query responses. It was developed based on ODBC. In fact, Sun supplies a bridge between them so that a program can access any data source accessible through an ODBC driver. The JDBC driver must perform data type mapping between JDBC and database types, and interpret SQL escape sequences.

Consider the following typical scenarios for using JDBC based on application architecture:

- Standalone applications.
- Applets communicating with a Web Server.
- Applications and applets communicating with a database through the JDBC/ODBC gateway.
- Applications accessing remote resources using mechanisms such as the Java RMI.

An application can have either a two-tier (with an application and the JDBC driver residing on the same machine) or a three-tier (with a client accessing an application component running as a Java Servlet on the web server machine accessing the database resource on behalf of client) architecture.

History of JDBC Specification

JDBC specification was and is an important component of overarching Java EE specification and Java EE programming model.

Here is a short snap shot of the JDBC specification and API evolution (extracted from the latest JDBC 4.1 specification).

> The JDBC API is a mature technology, having first been specified in January 1997. In its initial release, the JDBC API focused on providing a basic call-level interface to SQL databases. The JDBC 2.1 specification and the 2.0 Optional Package specification then broadened the scope of the API to include support for more advanced applications and for the features required by application servers to manage use of the JDBC API on behalf of their applications. The JDBC 3.0 specification operated with the stated goal to round out the API by filling in smaller areas of missing functionality.
> With JDBC 4.1, the goals are two fold:
> - First, to improve the Ease-of-Development experience for all developers working with SQL in the Java platform.
> - Secondly, to provide a range of enterprise level features to expose JDBC to a richer set of tools and APIs to manage JDBC resources.

Here is a link to the JDBC 4.1 specification: http://jcp.org/aboutJava/communityprocess/final/jsr221/

Four Types of JDBC Drivers

There are four types of JDBC driver categories based on how they are implemented:

Type 1 JDBC-ODBC bridge

Type 2 Native-API partly Java driver

Type 3 Open net-protocol all Java driver

Type 4 Proprietary-net-protocol all Java driver

Type 1 JDBC-ODBC bridge

It provides database access via one or more ODBC drivers. It requires that an ODBC driver be installed on a client machine. It provides no host redirection for database queries. The basic flow of processes involved is as follows:

> Client –> JDBC driver –> ODBC driver –> Database
> *JDBC driver makes a native call to ODBC driver.*

Sun's Microsystems supplies the JDBC-ODBC bridge driver (type 1) called **sun.jdbc.odbc.JdbcOdbcDriver** packaged with the Java SDK.

Type 2 Native-API partly Java driver

It provides no host redirection for database queries. It does not use ODBC, but directly communicates to a vendor-specific driver or database API. The basic flow of processes involved is as follows:

> Client –> JDBC driver –> Vendor API –> Database

Type 3 Open net-protocol all Java driver

It is capable of forwarding a database request to a remote data source. It is not vendor-specific and can interface to multiple database types. The net server gateway component runs on a remote machine and may or may not use an ODBC driver to access the database. But this process is completely transparent to the client.

Type 4 Proprietary-net-protocol all Java driver

It is an all Java implementation of the JDBC native database driver. The database and the JDBC application perform direct Java-to-Java calls using this driver type. Drivers can operate on any platform and can be downloaded from a server.

For currently available JDBC drivers see JDBC Data Access API: http://developers.sun.com/product/jdbc/drivers. The JDBC API is contained in the package called **java.sql**.

Creating a Simple JDBC Program

Below are the basic steps to set up your Java code to work with a database.

1. Load a JDBC driver

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// if you use a JDBC-ODBC bridge
```

or

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
// for Apache Derby database driver
```

or

```
Class.forName("jdbc.driverXYZ");
// if the driver you use has a name driverXYZ
// Driver vendor will provide a name for their driver
```

You can also load the JDBC driver by specifying the property **jdbc.drivers**, for example:

```
java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver myDatabaseProgram
```

2. Establish a connection to a database by specifying the database's URL

```
Connection con = DriverManager.getConnection (url, "myLogin", "myPassword");
```

or

```
Connection con = DriverManager.getConnection(url, properties);
```

If the JDBC-ODBC bridge driver is used, the JDBC URL will start with **jdbc:odbc:**. The rest of the URL is generally your data source name or database system. In place of " myLogin " you put the name you use to log in to the DBMS; in place of " myPassword " you put your password for the DBMS. The following are examples of valid URLs: **jdbc:odbc:myDataSource** or **jdbc:derby:myDB**.

For details on the URL syntax, always look in a given JDBC driver and database documentation.

The connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS.

JDBC 2.0 introduced the DataSource interface and the Java Naming and Directory Interface (JNDI) technology for naming and location services. JNDI can be used to hide details of a database connection such as a JDBC driver class, user name, password, and connection URI. All you have to do with JNDI to create a database connection is to specify a resource name that corresponds to an entry in a database or naming service. You will then receive the information necessary to establish a connection with a database. You will learn specifics of the JNDI API later in this session. You will also have a chance to work with the JNDI and DataSource object in the homework assignment.

3. Construct an SQL, or callable statement object, and send queries

```
try
{
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("Select emp_name from Employee");
}
catch (SQLException sqlexp)
```

```
{
process the exception
}
```

Only one ResultSet per Statement can be open at any point in time. Therefore, if the reading of one ResultSet is interleaved with the reading of another, each must have been generated by different Statement objects.

The **executeQuery()** method is used for Select statement whereas the **executeUpdate()** method is used for update statements.

Starting from JDBC 2.0 specification, there is an executeBatch() method for executing multiple statements.

4. Optionally retrieve metadata

We can use the **java.sql.DatabaseMetaData** and **java.sql.ResultSetMetaData** classes to get information about meta-data. It makes our program more dynamic and configurable.

5. Process query results--the transaction completes when the results have all been retrieved

Executing a Statement usually generates a ResultSet object. Public abstract interface ResultSet provides access to a table of data. A ResultSet maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The **next()** method moves the cursor to the next row.

```
while (rs.next())
{
...
rs.getString("First_Name");
// or
rs.getString(1);
...
// process data
}
```

In JDBC 2.0 there is a method called **previous ()** to move the cursor to the previous row in the result set.

6. Close the connection

For comprehensive coverage of JDBC API features, both enhancements and extensions, I would like to refer you to the JDBC Tutorial and Reference at
http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html .

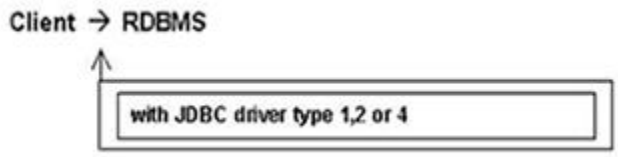
Study the JDBC API feature, specifically the scrollable result set, the updateable result set, and the Data Sources. They play an important role and have common usage in the application development process.

For JDBC 3.0 features and enhancements, please use the JDBC API Tutorial and Reference, 3rd Edition, see http://my.safaribooksonline.com/book/programming/java/0321173848

WebLogic Database Configuration

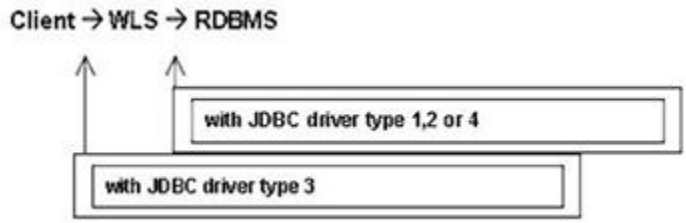
The WebLogic Server allows you to configure a database connection in several ways:

1. Two-tier



If the application has two-tier architecture the client program gets direct connection to the database using a type 1, 2, or 4 JDBC driver. In this scenario, if your client is a Java application running outside of the WebLogic Server container, you do not use the WebLogic Server so you do not even have to start it. However, you might have a Java client being implemented as the Java EE component, perhaps a Java Servlet, a JSP, or an EJB, that is running inside the WebLogic Server container. In this case you do need to start the WLS instance, not for the purpose of getting database connectivity service, but for providing run-time service for this client component.

2. Multi-tier



If the application has a multi-tier architecture the client program connects to one of the WebLogic servers using a type 3 JDBC driver and then the WebLogic server gets a direct connection to the database using a type 1, 2 or 4 JDBC driver.

Working with WLS Admin Console

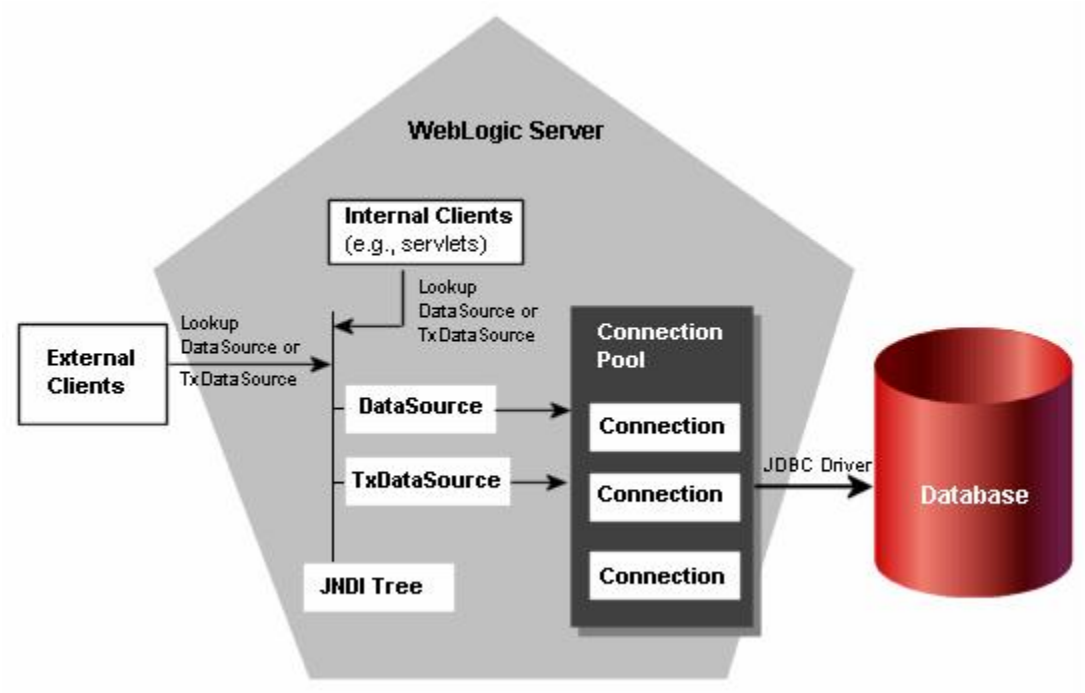
The WebLogic Server provides a variety of options for database access using the Java Database connectivity (JDBC) specification. These options include two-tier JDBC drivers called WebLogic jDrivers for Oracle, Microsoft SQL Server, and Informix database management systems (DBMS), and multi-tier drivers that work with the WebLogic Server as an intermediary between an application and the DBMS.

Multi-tier drivers use WebLogic Server to access connection pools that provide ready-to-use pools of connections to your DBMS. Because these database connections are already established when the connection pool starts up the overhead of establishing database connections is eliminated. Client and server-side applications can utilize connections from a connection pool through a DataSource on the JNDI tree (the preferred method) or by using a WebLogic wrapper driver. When finished with a connection, applications return the connection to the connection pool. This technique improves the overall performance of the application. However, to achieve the best performance level a DB connection pool has to be tuned by adjusting parameters such as the initial number of connections, maximum number of connections, shrinking capabilities, etc.

Connection pools require a two-tier JDBC driver to make the connection from the WebLogic Server to the DBMS. This two-tier driver can be one of the WebLogic jDrivers or a third-party JDBC driver.

For database access from **server-side applications**, such as HTTP Servlets, we have to use a DataSource from the Java Naming and Directory Interface (JNDI) tree or use the WebLogic Pool driver. For distributed transactions we always use a TxDataSource from the JNDI tree. For transactions distributed across multiple servers within a single WebLogic domain with one database instance, we use a TxDataSource from the JNDI tree or use the JTS driver.

For accessing DB connection pools from **client-side applications**, Oracle recommends using a DataSource from the JNDI tree to access database connections rather than the RMI driver. Oracle offers the RMI driver for client-side JDBC. The RMI driver provides a standards-based approach using the Java Enterprise Edition (Java EE) specifications. The WebLogic RMI driver is a Type 3 JDBC driver that uses RMI and a DataSource object to create database connections.



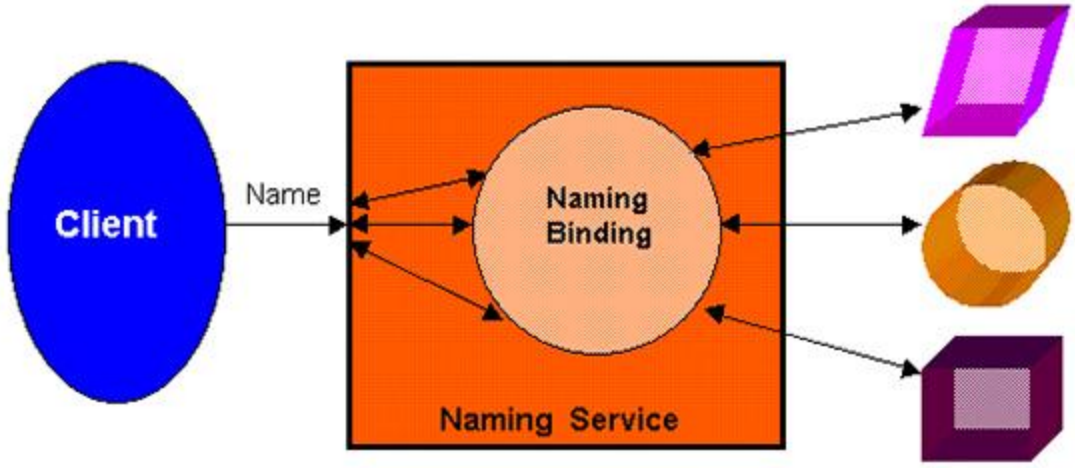
This image is adopted from Oracle WLS documentation.

For more information please use the WebLogic Server documentation listed in the "Links and References" section. Reviewing and using WebLogic documentation will also help you with your homework assignment.

JNDI as a Naming and Directory Services Interface

Problem: Differences in Naming Service

Naming and directory services are the ability of an application to locate an object or service that may be anywhere on the network.



A generic naming service

A naming service maintains a set of **bindings**. Bindings relate names to objects. All objects in a naming system are named in the same way (that is, they subscribe to the same **naming convention**). Clients use the naming service to locate objects by name.

There are a number of existing naming services that follow this pattern, but differ in the details:

- **COS (Common Object Services) Naming:** The naming service for CORBA applications; allows applications to store and access references to CORBA objects.
- **DNS (Domain Name System):** The Internet's naming service; maps people-friendly names into computer-friendly IP (Internet Protocol) addresses in dotted-quad notation (207.69.175.36). Interestingly, DNS is a distributed naming service.
- **LDAP (Lightweight Directory Access Protocol):** Developed by the University of Michigan; as its name implies, it is a lightweight version of DAP (Directory Access Protocol), which in turn is part of X.500, a standard for network directory services. Currently, over 40 companies endorse LDAP.
- **NIS (Network Information System) and NIS+:** Network naming services developed by Sun Microsystems. Both allow users to access files and applications on any host with a single ID and password.

They all have something in common; they:

- provide the ability to bind names to objects and to look up objects by name;
- do not store objects directly but instead store references to objects. As an illustration, consider DNS: the address 207.69.175.36 is a reference to a computer's location on the Internet, not the computer itself.

Unfortunately they all have their proprietary naming format or naming conventions. For example, here are naming conventions for files and their locations defined within different naming systems:

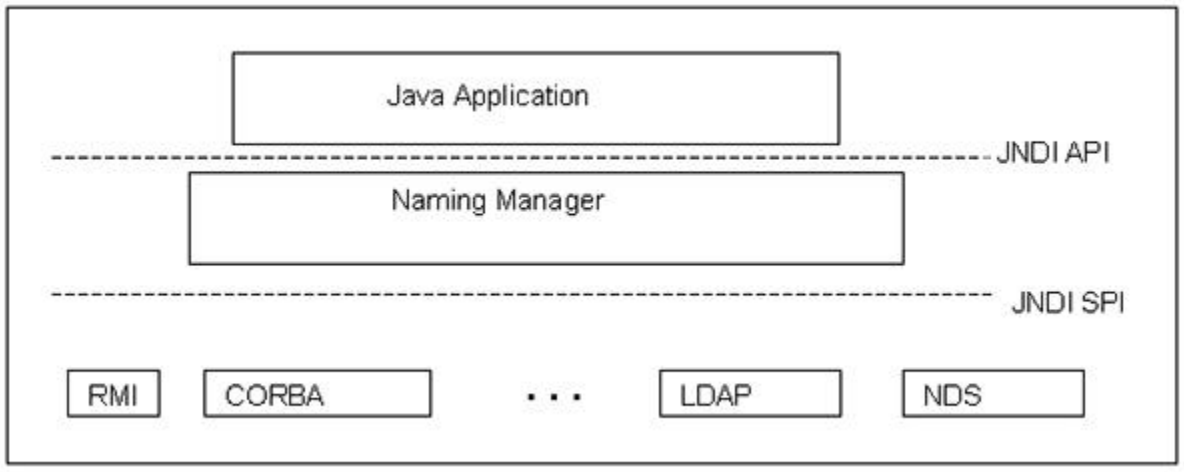
Unix	/myDir/mySubDir/myFile
DOS	c:\myDir\mySubDir\myFile
LDAP	cn=Leonid,o=jhu,c=USA

DNS	apl.jhu.edu

What is JNDI?

JNDI is an API specified in the Java programming language that provides directory and naming functionality to Java applications. It is defined to be independent of any specific directory service implementation.

- JNDI is a standard Java interface for accessing distributed services and data by name. It provides a portable, unified interface for services that need naming and directory services. The JNDI specification was defined independently of any specific naming or directory service implementation, but a developer can use it to access different, multiple naming and directory services such as LDAP, NDS, and NIS (YP). The JNDI specification also includes a specification for a service provider interface (SPI). The SPI provides a way for naming and directory service providers to make their services available through the JNDI interface.
- JNDI distinguishes between naming services and directory services:
  - A **naming service** provides a mechanism to name objects (bind) and retrieve objects by name (lookup).
  - A **directory service** is a naming service that also allows for attributes to be associated with each object and provides a way to retrieve an object based on its attributes rather than its name (search).
- JNDI includes two parts: the client API and the Service Provider Interface (SPI).



The JNDI API allows Java applications to access a variety of naming and directory services. The JNDI SPI is designed to be used by arbitrary service providers including directory providers. This enables a variety of directory and naming services to be plugged in transparently to the Java application that is using the JNDI API.

Important Concepts of JNDI

Naming Services

- An **atomic name** is an indivisible component of a name.

- A **compound name** represents a sequence of zero or more atomic names composed according to the naming convention.
- A **binding** is the association of an atomic name with an object.
- A **context** is an object whose state is a set of bindings with distinct atomic names. Every context provides a lookup operation that returns the object and may provide operations such as for binding names, unbinding names, listing bound names.
- An atomic name in one context object can be bound to another context object of the same type called a **subcontext**.
- A **naming system** is a connected set of contexts of the same type providing the same set of operations with identical semantics.
- A **namespace** is the set of all names in a naming system.
- A **composite name** is a name that spans multiple naming systems.
- Client can obtain an **initial context** object that provides a starting point for resolution of names.

Directory Services

- **Directory** is a particular type of object that is used to represent the variety of information in a computing environment.
- **Directory service** provides structure to a set of directory objects that is usually hierarchical in nature.
- A directory object can have an **attribute** associated with it that has an identifier and a set of values.
- Since the objects that can be stored in a directory service can be of nearly any type or structure, we can serialize Java objects and store them in the directory service.

JNDI Naming Services

How to use JNDI Naming Services to Locate and Access Remote Objects

The **java.naming** package defines the **Context** interface. A context represents a starting point for a naming or directory services. Once created, an object implementing the Context interface represents the root for all naming operations performed.

Here are the steps necessary to code a JNDI client:

1. Create an instance of **java.util.Hashtable** or **java.util.Properties**.
2. Add JNDI-specific variables to **Hashtable** or **Properties** object:
    a. The JNDI class name provided by naming service (**weblogic.jndi.WLInitialContextFactory** class for WebLogic Server).
    b. A URL - the naming server location.
    c. Security credential.
3. Create an InitialContext using **Hashtable** or **Properties** object or via a jndi.properties file. For a complete listing of available properties see the Java APIs documentation java.naming.Context and weblogic.jndi.WLInitialContextFactory (for WebLogic-specific properties).

```
java.utils.Properties p = new java.utils.Properties();
. . .
p.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFa
ctory");
. . .
p.put(Context.PROVIDER_URL, url);
. . .
p.put(Context.SECURITY_PRINCIPAL, user);
```

```
. . .
p.put(Context.SECURITY_CREDENTIALS,password);
InitialContext ctx = new InitialContext(p);
```

4. With use of a WebLogic Environment object **weblogic.jndi.Environment**[1] this process can be accomplished even more simply:

```
Environment env = new Environment();
Context ctx = env.getInitialContext();
```

(See WebLogic Server documentation for details.)

5. After creation of an InitialContext object we perform different operations:
   . To bind an object in a context:

```
bind(String, Object)
bind(Name, Object)
```

The name must not be bound to another object. All intermediate contexts must already exist.

   a. To lookup an object in a context:

```
lookup(String)
lookup(Name)
```

   b. To remove an object from a context:

```
unbind(String)
unbind(Name)
```

   c. To list all objects from a context:

```
list(String)
list(Name)
listBindings(Name)
listBindings(String)
```

   d. To create a subcontext:

```
createSubcontext(Name)
createSubcontext(String)
```

   e. To delete a subcontext:

```
destroySubcontext(Name)
destroySubcontext(String)
```

6. **Note:** [1] - Please see WebLogic Server API Reference

(http://docs.oracle.com/cd/E24329_01/web.1211/e24489/toc.htm#CHDGBCAI)

as a reference to weblogic.jndi package as well as all WebLogic Server proprietary classes and interfaces.

JNDI Directory Services

How to use JNDI Directory Services to Locate and Access Remote Objects

- A directory service manages a directory of **entries**. An entry can refer to any type of object or abstract concept.
- An entry has **attributes** associated with it. An attribute is a name/value pair. It might have one or more values.
- A search function on a directory is much like search on a database. It might involve filtering.
- Directory service providers allow searches not only by name, but also based on a set of search criteria.
- The directory package java.naming.directory is an extension to the java.naming package. All of the searching methods in naming services are also applicable to directory services.

If you want to learn more about directory services and API classes see Sun's JNDI API reference and other sources listed on the Links and References section.

Use Case Scenarios

So, what are the typical scenarios of applying JNDI capabilities in applications?

- User authentication information (like passwords) can be stored as attributes associated with each user in the directory. Then it becomes just a matter of looking up the attribute ("password") of the user and verifying the authenticity.
- An Email or mailing address associated with a particular user can be stored in the directory. Again, facilitating a simple lookup to obtain the necessary information.
- Database applications can use the directory to locate database servers and other information on how to access a particular database (that's how you will use JNDI in upcoming homework assignment).
- The information about shared network devices (like shared printers or file servers) can be stored in the directory.
- Remote objects (like Enterprise JavaBeans) can bind themselves into naming or directory facilities. The client then has to look up given object information.

DataSource Objects

Using a DataSource Object to get a JDBC Connection

- As an alternative way for connecting to a data repository, a DataSource object from the JDBC optional package allows JDBC clients to obtain DBMS connection. It makes code more portable and easier to maintain.
- A DataSource object represents a real world data source. Depending on how it is implemented, the data source can be anything from a relational database to a spreadsheet or a file in tabular format.
- When a DataSource object has been registered with a JNDI naming service an application can retrieve it from the naming service and use it to make a connection to the data source it represents.
- To create a DataSource object with WebLogic Server we need to define it as an entry in the config.xml file. In order to do that we typically use the Administration Console GUI tool.

- DataSource objects can be defined with or without Java Transaction Services (JTS) enabled. Creating a DataSource object with JTS enabled causes the connection to behave like a JTS connection with support for transactions.

Client Connection

Obtaining a Client Connection using DataSource

Do you remember the chain of actions to gain database connection through the DriverManager? If not you might want to review the "Programming with JDBC" section of this session.

With DataSource - database connection becomes much simpler. All you need to do to obtain a database connection from a JDBC client is to use a Java Naming and Directory Interface (JDNI) look up to locate the DataSource object.

```
Context initcntx = null;
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory
");
ht.put(Context.PROVIDER_URL, "t3://localhost:7001");
try
{
initcntx = new InitialContext(ht);
javax.sql.DataSource ds = (javax.sql.DataSource)
initcntx.lookup("myDataSource");
java.sql.Connection conn = ds.getConnection();
Statement stmt = conn.createStatement();
stmt.execute("select * from myTable");
ResultSet rs = stmt.getResultSet();
. . .
stmt.close();
conn.close();
}
catch (NamingException e) {
// an action to process a failure
}
finally {
try {initcntx.close();}
catch (Exception e) {
// a failure occurred
}
}
```