Enterprise Java Beans

Overview

Notes from:

"Advanced Java 2 Platform How to Program" Deitel Deitel Santry

"Thinking in Enterprise Java" Bruce Eckel et. Al.

Sun downloadable documentation
EJB 2.1 API
http://java.sun.com/webapps/download/Display

Enterprise Java Beans

- Server-Side
  EJB objects may offer a remote view (via a remote procedure call protocol) a local view (direct procedure call) or both
- Managed
  EJB container services are more involved than the plain old JVM
- Components
  distributed in binary format and are configurable

Server-Side Implications

- In order to pull off the RPC trick we need:

- A naming service
  -- e.g. RMI clients make requests on the rmiregistry
- RPC proxies
  -- communications code along with the appropriate interface

Managed: EJB Container Services

- Object Persistence and Synchronization
- Declarative Security Control
- Declarative Transaction Control
- Concurrency Management
- Scalability Management

EJB Types

- Entity Beans
- Session Beans
- Message-Driven Beans

EJB Types

- Entity Beans
- Session Beans

} RMI-based server side components Accessed using distributed object Protocols (RMI IIOP)

- Message-Driven Beans

} New in EJB 2.0 Asynchronous server side component that responds to JMS asynchronous messages (Think provider like JAXM)

Entity Beans

- Represent real world entities (customers, orders, etc.)
- Persistent Objects typically stored in a relational database using CMP (Container Managed Persistence) or BMP (Bean Managed Persistence)
- The client sees no difference between CMP and BMP beans
- CMP promotes component portability (less reliant on the container)

Session Beans

- Are an extension of the client application
- Manage processes or tasks
- Are not persistent
- Often employ several different kinds of entity beans
- Implement business logic
- Come in two types (which can be more easily shared?)
  - Stateless session beans (no memory between calls)
    purchase(severalItems,creditCardNo);

  - Stateful session beans (remember earlier calls)
    addToCart(item);
    purchase(creditCardNo);

Message-Driven Beans

- Work in cooperation with Java Messaging System (JMS)
- JMS is an abstraction API on top of Message-Oriented Middleware (MOM) – like JDBC is an abstraction API on top of SQL databases
- Each MOM vendor implements things differently
- MDB's allow the developer to program using the publish-subscribe messaging model based on asynchronous, distributed message queues
- The MOM vendor need only provide a service provider for JMS (IBM's MQSeries or Progress' SonicMQ)

Message-Driven Beans

- Are like session beans
- Have no persistent state
- Coordinate tasks involving other session beans or entity beans
- Listen for asynchronous messages
- Unlike Session beans, provide no remote interface describing the methods that can be called

Message-Driven Beans

- Are receivers of MOM messages coming through the JMS API.
- Usually take action when a message is received.
- Unlike session and entity beans, Message-Driven Beans expose no remote or local view. They are not called directly called by a client.

Before working with EJB's

- Understand the role of JNDI (Java Naming and Directory Interface)

Naming

- Concepts from the JNDI Tutorial
- Java Naming and Directory Interface

Naming Concepts

- We need to map people friendly names to objects
- Examples

  mm6@andrew.cmu.edu -> Mike's mailbox

  www.cnn.com -> cnn's web server

  c:\somedir\f1.dat -> a file on a C drive

Naming Conventions

- Different naming systems use different conventions (or syntax) for names

- Examples:
  DOS uses slashes and colons and periods c:\some\f.dat

  Unix uses slashes /usr/local/filename

  DNS uses dots www.cnn.com

  LDAP (The lightweight directory access protocol) uses name, value pairs cn=Rosanna Lee, o=Sun, c=US

Bindings

- A *binding* is the association of a name with an object or an object reference.

  Examples:

  www.cnn.com is bound to an IP address

  c:\exam1.doc is bound to a file

  a phone number is bound to a phone

Bindings

- A *binding* is the association of a name with an object or object reference.

  The phone book maps names to numbers which act as references to objects. The number is a communication endpoint.

- A *communication endpoint* is specific information on how to access an object

Context

A *context* is a set of name-to-object bindings.

Every context has an associated naming convention.

A context may allow operations such as bind, unbind, lookup.

A context may associate a name with another context (subcontext, or subdirectory)

Naming System

- A *naming system* is a connected set of contexts of the same type (they have the same naming convention) and provides a common set of operations. DNS and LDAP, for example, are two naming system.
- A naming system provides a *naming service* to its customers for performing naming-related operations. For example, associate a new name with an IP address.
- A *namespace* is the set of names in a naming system.

Directory Service

- A *Directory Service* is an extension of a naming service that allows one to lookup objects based on names or based on attributes.
- Attributes have *attribute identifiers* and a set of *attribute values*.

For example, UDDI - Universal Description, Discovery and Integration is a Directory Service.

Reverse lookup or *content-based searching*

- Example queries to directory services:

  Find all machines whose IP address begins with 192.115.50.

  Find all companies that provide hardware support services.

Directory Enabled Applications

A *directory-enabled* application is an application that uses a naming or directory service.

Applications can share the common infrastructure provided by the directory.

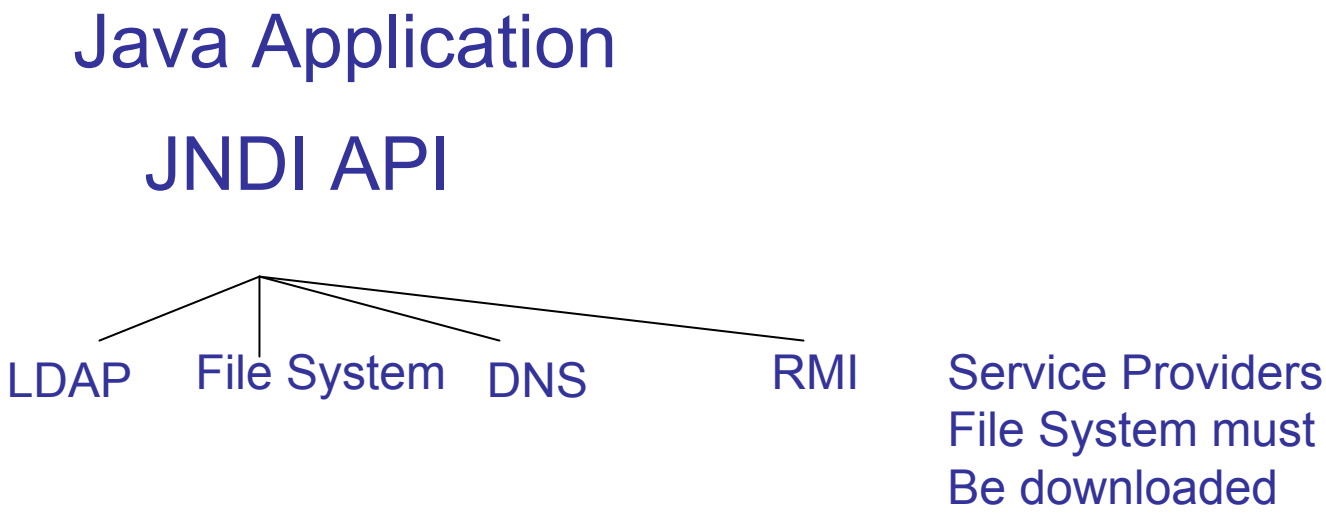
Example: A mail client, scheduling systems and mail forwarding program might all use the same address book stored in a common directory.

The directory may also be used as an Object store for programs needing the same object.

Java Naming and Directory Interface JNDI

Java Naming and Directory Interface (JNDI)

- An abstraction API (like JDBC handles different RDBMS databases)
- The JNDI API handles or sits on top of different naming services.



JNDI

- The javax.naming packages contains mostly Java interfaces.
- Some vendor implements the interface to provide JNDI support for their service.
- To use the service, you need a JNDI Service Provider that implements the interface
- JDK1.4 comes with RMI, DNS, COS, and LDAP Service providers.
- Sun's web site has an additional JNDI Service Provider that works with the local file system

JNDI

- A namespace is a logical space in which names can be defined.
- The same names in different namespaces cause no collisions.
- Namespaces can be nested:
  - file system directories are nested
  - the Internet DNS domains and sub-domains are nested

Namespaces are represented by the Context Interface

- Different classes implement this interface differently depending on which naming service they are accessing.
- Has methods to
  - bind and unbind objects to names
  - create and delete sub-contexts
  - lookup and list names
- Since a Context is a Java object it can be registered in another Context with its own name.

The Context Interface

- Start from some “root” context.
- Get the “root” from the InitialContext class
- Examples

  LookUp.java

  ListCurrentDirectory.java

LookUp.java

```
// before running download JNDI provider from Sun
// add .jar files to classpath

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;
import java.io.File;
```

```
public class LookUp {

  public static void main(String args[]) throws NamingException {

   try {

     System.out.println("Using a file system (FS) provider");

     // initialize the context with properties for provider
     // and current directory
     Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.fscontext.RefFSContextFactory");
     env.put(Context.PROVIDER_URL,
                 "file:D:\\McCarthy\\www\\95-702\\examples\\JNDI");

     Context ctx = new InitialContext(env);

     Object obj = ctx.lookup(args[0]);
```

```
    if(obj instanceof File) {

        System.out.println("Found a file object");

        System.out.println(args[0] + " is bound to: " + obj);

        File f = (File) obj;

        System.out.println(f + " is " + f.length() + " bytes long");
     }
     // Close the context when we're done
     ctx.close();
    }
    catch(NamingException e) {
        System.out.println("Naming exception caught" + e);
    }
  }
}
```

```
D:\McCarthy\www\95-702\examples\JNDI>java LookUp LookUp.java
Using a file system (FS) provider
Found a file object
LookUp.java is bound to: D:\McCarthy\www\95-702\examples\JNDI\LookUp.java
D:\McCarthy\www\95-702\examples\JNDI\LookUp.java is 1255 bytes long
```

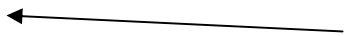
ListCurrentDirectory.java

```
// Use JNDI to list the contents of the current
// directory

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.naming.NamingEnumeration;
import javax.naming.NameClassPair;
import java.util.Hashtable;
import java.io.File;
```

```
public class ListCurrentDirectory  {

  public static void main(String args[]) throws NamingException {

   try {

     Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.fscontext.RefFSContextFactory");
     env.put(Context.PROVIDER_URL,
                 "file:D:\\McCarthy\\www\\95-702\\examples\\JNDI");
```

```
    Context ctx = new InitialContext(env);

    NamingEnumeration list = ctx.list(".");

    while (list.hasMore()) {
        NameClassPair nc = (NameClassPair)list.next();
        System.out.println(nc);
     }
    ctx.close();
   }
   catch(NamingException e) {
       System.out.println("Naming exception caught" + e);
   }
  }
}
```

```
D:\McCarthy\www\95-702\examples\JNDI>java ListCurrentDirectory
ListCurrentDirectory.class: java.io.File
ListCurrentDirectory.java: java.io.File
LookUp.java: java.io.File
SimpleJNDI.java: java.io.File
x: javax.naming.Context
```



x is a DOS directory

```
// Use JNDI to change to a sub directory and list contents

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.naming.NamingEnumeration;
import javax.naming.NameClassPair;
import java.util.Hashtable;
import java.io.File;

public class ChangeContext  {

  public static void main(String args[]) throws NamingException {

   try {
     Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.fscontext.RefFSContextFactory");
     env.put(Context.PROVIDER_URL,
                "file:D:\\McCarthy\\www\\95-702\\examples\\JNDI");
```

```
    Context ctx = new InitialContext(env);

    // a subdirectory called x contains a file f.txt and a subdirectory t
    Context sub = (Context)ctx.lookup("x");

    NamingEnumeration list = sub.list(".");

    while (list.hasMore()) {
         NameClassPair nc = (NameClassPair)list.next();
         System.out.println(nc);
     }
    ctx.close();
    sub.close();

   }
   catch(NamingException e) {
       System.out.println("Naming exception caught" + e);
   }
  }
}
```

```
D:\McCarthy\www\95-702\examples\JNDI>java ChangeContext

f.txt: java.io.File
t: javax.naming.Context
```

Back to EJB

- Implementing session and entity beans
- Implementing message-driven beans

Implementing Entity and Session Beans

- Define the component interfaces
  - You may choose to define all or only some of these depending on how you want your bean used
  - local interfaces do not require RMI overhead
- Define a bean class
- For entity beans define a primary key

Implementing Entity and Session Beans

- Define the component interfaces
  - The remote interface specifies how the outside world can access the bean's business methods
  - The remote home interface specifies how the outside world can access the bean's life-cycle methods (for creating, removing and finding)
  - The local interface specifies how the inside world (same EJB container) can access the bean's business methods
  - The local home interface specifies how the inside world can access the bean's life-cycle methods

Implementing Entity and Session Beans

- Implement the bean
  - Fill in the code for the business and life-cycle methods
  - It's not normal to directly implement the interfaces as we do in standard Java (though you must provide many of the methods). The calls to methods are not normal Java calls. They first go through the container.
  - Session beans implement javax.ejb.SessionBean
  - Entity beans implement javax.ejb.EntityBean
  - Both beans extend javax.ejb.EnterpriseBean

For Entity Beans

- Define a primary key class
  - Required for entity beans
  - Provides a pointer into the database
  - Must implement Java.io.Serializable
- The EJB instance represents a particular row in the corresponding database table
- The home interface for the entity EJB represents the table as a whole (has finder methods.)

Implementing a Message-Driven Bean

- Has no local, local home, remote, or remote home interfaces to define.
- The container will call the onMessage() method when an asynchronous message arrives. (Like JAXM message provider.)
- Extends the EnterpriseBean class and implements the javax.ejb.MessageDrivenBean and javax.jms.MessageListener interfaces

Implementing a Message-Driven Bean

- Two basic messaging-system models

  (1) point-to-point model allows messages to be sent to a message queue to be read by exactly one message consumer

  (2) publish/subscribe model allows components to publish messages on a topic to a server to be read by zero or more subscribers

In both models

- The messages hold

  -- a header containing the destination and the sending time.

  -- message properties to allow the receiver to select which messages they would like to receive. These may be set by the sender.

  -- the message body itself

Point-to-point On the client side

```
import javax.jms.*;

QueueConnection qCon;
QueueSession qSes;
QueueSender qSen;
```

Through JNDI get access to a QueueSender.

Build messages and send.

Point-To-Point On the server side

import javax.jms.*;

QueueConnection qCon;
QueueSession qSes;
QueueReceiver qRec;

Through JNDI get access to a QueueSender.
Build a MessageListener with an onMessage method.