Enterprise Connection mit JNDI

von Muhammet Öztürk, Ralf Bratenberg

Service ist alles!

Anwendungen werden immer häufiger über verschiedene Systeme verteilt, wodurch sich der Zugriff auf zentrale Informationen und Ressourcen problematisch und komplex gestaltet. Das Java Naming and Directory Interface (JNDI), welches eine generische Architektur besitzt, ermöglicht es, solche Probleme über eine standardisierte Schnittstelle einfach zu lösen. Für Java-Anwendungen bietet JNDI eine generische Schnittstelle zu Directory und Naming Services, wie zum Beispiel LDAP.

Durch die Einführung des Lightweight Directory Access Protocols (LDAP) in Java Enterprise-Applikationen nutzt man die Vorteile dieses bewährten Systems. Directory Server stellen den Enterprise-Anwendungen über JNDI eine standardisierte Schnittstelle zur Verfügung, sodass sie auf zentrale Enterprise-Informationen einheitlich mittels dieser Schnittstelle zugreifen können. Java Enterprise-Anwendungen sind dadurch in der Lage Objekte zu speichern, zu suchen und zu lesen. Dieses Konzept hat sich auch durch den Einsatz in EJB-Containern bewährt. Wir werden in diesem Artikel einen Überblick über Name und Directory Services und die JNDI-Architektur geben. Anschließend werden wir mit einem Beispiel demonstrieren, wie man auf die Informationen in einem LDAP-Server zugreift und in einer Swing-Anwendung anzeigt. Das Beispiel zeigt auch die Basisfunktionalitäten eines LDAP-Servers sowie das Erzeugen, Löschen und Lesen eines Eintrags.

Name Services

Name Services sind fundamentale Dienste in allen Computersystemen. Sie sind in der Lage, System-Ressourcen mit für Menschen verständlichen Namen zu verbinden und bieten die Möglichkeit über diese Namen auf die jeweiligen Ressourcen zuzugreifen. Es gibt verschiedene Naming Services. Ein Beispiel dafür ist das Datei-System eines Betriebssystems, in dem Dateien mit Dateinamen verbunden sind. Ein anderes Beispiel ist ein Domain Name Service (DNS), welcher die IP-Adresse mit dem Host-Name verbindet. Viele Naming Services speichern die Objektreferenz anstelle des Objekts. Die Objektreferenz beinhaltet die wichtigsten Informationen über das Objekt, während das Objekt den Objektzustand selbst darstellt.

Alle Objekte in einem Naming Service unterliegen der gleichen Namenskonvention. In Abbildung 1 werden die Konventionen von DNS und NFS (Network File System) dargestellt. Im D-Name Service wird der Name von rechts nach links ausgewertet und durch einen Punkt (.) getrennt. Im UNIX Datei-System wird dagegen ein Name durch Slash (/) getrennt und von links nach rechts ausgewertet.

Ein Naming Service verwaltet eine Reihe von „Bindings“. Unter einem Binding

versteht man das Verbinden eines Namens mit einem Objekt. Der Client benutzt den Naming Service, um auf die Objekte über deren Name zuzugreifen (Abb. 2).

Directory Services

Directory Services sind die Erweiterung von Naming Services, welche nicht nur den Namen, sondern zusätzlich auch Attribute mit einem Objekt verbinden. Über Attribute kann zusätzlich zu einem Naming Service auf bestimmte Objekte zugegriffen bzw. danach gesucht werden (Abb. 3).

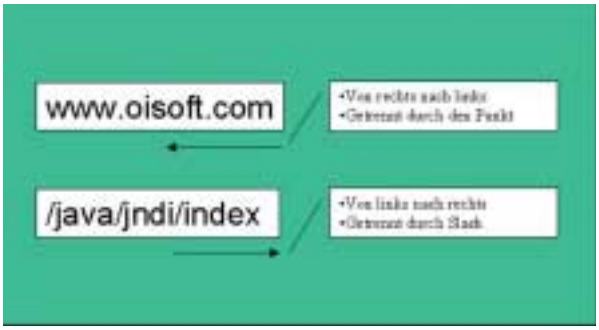
Directory Services bieten die Möglichkeit, verteilte Informationen zu verwalten. Solche Informationen können von der Email-Adresse der Mitarbeiter bis zu IP-Adressen oder Netzwerkdrucker reichen. Im folgenden Beispiel wird ein Eintrag dargestellt, welcher mehrere Attribute besitzt. Zu beachten ist, dass das Attribut *Email* mehrfach vorkommen kann.

Name: Ralf Bratenberg
Addresse:Hauptstr.12, Schwabenland
Email: RBG@oisoft.com
Email: Ralf.Bratenberg@oisoft.com

Man kann sich einen Directory Service als eine Art Datenbank vorstellen. So wie relationale Datenbanken bietet der Directory Service search filter-Funktionalität, sodass man Attribute des zu suchenden Objekts spezifizieren kann.

Naming und Directory Services sind meistens logische Partner. In der Tat bieten alle Produkte beide Funktionalitäten an. Das Lightweight Directory Access Protocol (LDAP) bietet zum Beispiel Naming- sowie Directory Service-Funktionalität.

Abb. 1: Zwei Beispiele verschiedener Naming-Konventions (DNS und NFS)



Was ist JNDI?

Das Java Naming and Directory Interface (JNDI) ist ein API, welches von Sun den Entwicklern zur Verfügung gestellt wird und die Möglichkeit bietet, Naming- und Directory Service-Funktionalitäten in Java-Anwendungen zu nutzen. Damit definiert Sun eine abstrakte Schnittstelle zum Zugriff auf verschiedene Naming und Directory Services. Eine Java-Anwendung kann durch diese abstrakte Schnittstelle einheitlich auf verschiedene Naming und Directory Services zugreifen. Dadurch können diese ausgetauscht werden, ohne die Java-Anwendung neu zu kompilieren (nur wenn die Namen im neuen Service gleich bleiben).

In Abbildung 4 ist die Architektur von JNDI dargestellt. Diese besteht aus dem JNDI-API und dem Service Provider Interface (SPI). Die Architektur basiert somit auf dem gleichen Konzept wie JDBC und stellt eine abstrakte Schicht zwischen Java-Anwendungen und Java-Services dar.

Java-Applikationen benutzen das JNDI-API zum Zugriff auf verschiedene Naming und Directory Services, welche als Plug-In „generisch" ins JNDI-Framework angehängt werden. Der Naming Manager verwaltet die verschiedenen Anforderungen der Java-Applikationen und liefert die Ergebnisse dieser Anforderungen an die jeweilige Anwendung zurück.

Das Service Provider Interface (SPI) bietet die Möglichkeit, Produkte verschiedener Hersteller als Plug-In in JNDI anzubinden, sodass Java-Anwendungen auf diese Dienste zugreifen können. Das Interface wird von verschiedenen Services implementiert.

Beispiel

Als Beispiel, um die Funktionalität des Frameworks darzustellen, wurde ein einfaches Adressbuch als Java Swing-Applikation realisiert. Das GUI der Swing-Applikation besteht aus drei Bereichen, die jeweils eine bestimmte Zugriffsart demonstrieren. Jeder Bereich ist als eigenständiges Adressbuch anzusehen. Auf der linken Seite des GUI wird die hierarchische Struktur der jeweiligen Daten als Baum angezeigt. Die rechte Seite zeigt die Detailansicht eines Eintrags (Abb. 5).

Die Anwendung zeigt die wichtigsten Funktionalitäten (Anlegen, Löschen und Ändern eines Eintrags), indem auf einen LDAP-Server zugegriffen wird. JNDI

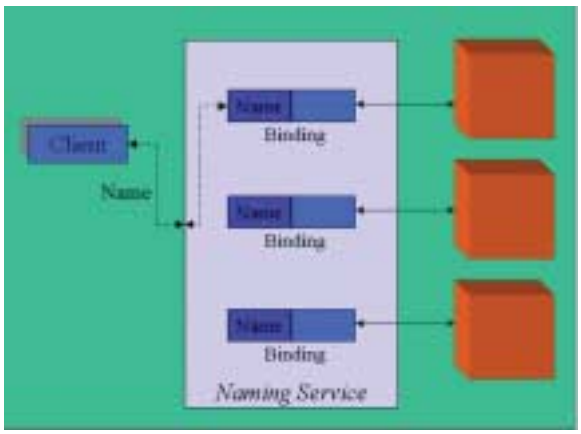
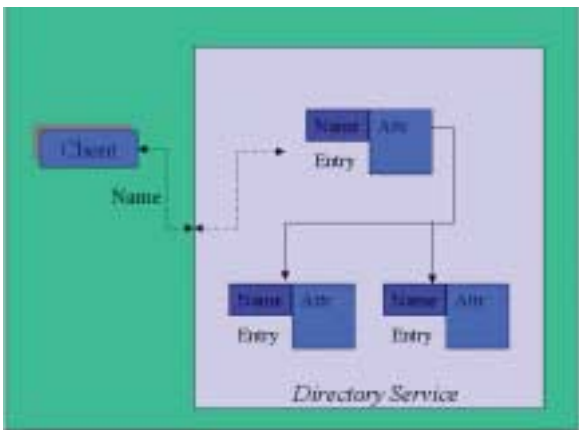


Abb. 2: Naming Service

Abb. 3: Directory Service



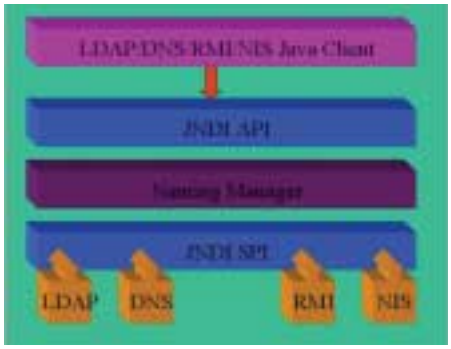


Abb. 4: JNDI-Architektur

unterstützt verschiedene Möglichkeiten, um auf die Informationen im Server zuzugreifen. Wir haben in diesem Beispiel drei Arten des Zugriffs implementiert. Als Server kam openLDAP zum Einsatz, der im Internet sowohl für Windows als auch für Linux frei erhältlich ist [3]. Wir haben die Applikation unter JDK 1.3 mit openLDAP 2.0 getestet. Das JNDI-API ist ab Version 1.3 im JDK enthalten. Für ältere Versionen muss das JNDI-API zusätzlich installiert werden. Die Installationsanleitung von openLDAP ist im Internet zu finden [3]. Damit das Beispiel ohne Änderungen lauffähig ist, sollten in der Server-Konfigurationsdatei *slap.conf* folgende Einträge vorgenommen werden:

```
include     C:/oisoft/ openldap-2_0_19_debug/schema/
                                              core.schema
include     C:/oisoft/ openldap-2_0_19_debug/schema/
                                              cosine.schema
include     C:/oisoft /openldap-2_0_19_debug/schema/
                                              inetorgperson.schema
include     C:/oisoft /openldap-2_0_19_debug/schema/
                                              java.schema
include     C:/oisoft /JNDI/openldap-2_0_19_debug/
                                              schema/openldap.schema
...
suffix      "dc=oisoft,dc=com"
rootdn      "cn=Manager,dc=oisoft,dc=com"
```

Abb. 5: Das GUI der Swing-Applikation

Wir liefern mit dem Beispiel zusätzlich zwei *ldif*-Dateien (*init.ldif* und *example.ldif*), um die Directory-Struktur des Servers anzulegen. Dazu müssen Sie die folgenden Befehle nach dem Start des Servers ausführen:

Für Linux/Unix:

```
ldapadd -x -D "cn=Manager,dc=oisoft, dc=com" -W -f init.ldif
ldapadd -x -D "cn=Manager,dc=oisoft, dc=com" -W -f exam-
ple.ldif
```

Für Windows:

```
ldapmodify –a -x -D "cn=Manager,dc=oisoft,
                                    dc=com" -W -f init.ldif
ldapmodify –a -x -D "cn=Manager,dc=oisoft,
                                    dc=com" -W -f example.ldif
```

Nach der erfolgreichen Ausführung dieser Befehle steht der openLDAP-Server für die Applikation zur Verfügung. Die Applikation wird mit dem Befehl *java Examlpe* gestartet. Um die verschiedenen Zugriffsarten aufzuzeigen, haben wir eine Klasse *HandlerFactory* nach dem Factory Pattern implementiert [4]. Diese liefert dem GUI, je nach ausgewähltem Bereich, ein entsprechendes Objekt zurück, über das auf den Server zugegriffen wird. Im Folgenden werden wir die wichtigsten Programmteile erläutern. Für ein umfassendes Verständnis der Applikation empfehlen wir, einen näheren Blick auf den gesamten Quellcode (auf der beiliegenden CD oder unter [5]) zu werfen.

Alle Server-Operationen erfolgen relativ zu einem Context-Objekt (Abb. 3). Ein Context-Objekt repräsentiert eine Menge von Zuordnungen zwischen Namen und Objekten. Zum Beispiel bilden die Dateien eines Verzeichnisses in einem Dateisystem einen Context. Ein Name in diesem Context ist einer Datei zugeordnet. Ein Context-Objekt bietet verschiedene Operationen zum Anlegen, Suchen, Löschen und Modifizieren von Einträgen. Ein Context-Objekt kann wiederum Context-Objekte beinhalten sowie ein Verzeichnis eines Dateisystems, das wiederum weitere Verzeichnisse enthalten kann. Daraus resultiert eine hierarchische Struktur. DirContext ist abgeleitet von Context und besitzt zusätzlich Attribute und Methoden, um auf diese zuzugreifen.

Eine Applikation kann ein Context-Objekt nicht selbst erzeugen. Da es immer nur von einem anderen Context geliefert wird, muss ein Einstiegspunkt erzeugt werden. Dazu wird ein InitialContext erzeugt. Alle weiteren Zugriffe geschehen relativ zu diesem initialen Context. In Listing 1 wird ein InitialDirContext erzeugt, wodurch die initiale Verbindung zum LDAP-Server aufgebaut wird. Es wird spezifiziert, mit welchem Service-Provider die Verbindung hergestellt werden soll. In unserem Falle wird ein LDAP Service Provider benutzt. Mit dem Aufruf:

Listing 1: Example-Konstruktor

```
public Example() {
  ...
     Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,"
                        com.sun.jndi.ldap.LdapCtxFactory");
     env.put(Context.PROVIDER_URL, "ldap://localhost:
                        389/o=Example,dc=oisoft,dc=com");
     env.put(Context.SECURITY_PRINCIPAL, "cn=Manager,
                                        dc=oisoft,dc=com");
     env.put(Context.SECURITY_CREDENTIALS, "secret");
  ...
  try {
     rootContext   = new InitialDirContext(env);
  }
  ...
}
```

Listing 2: Inhalt eines Context auflisten – Klasse *ObjectHandler*

```
Object object = rootContext.lookup("cn=Entry Name");

protected void getTree (DirContext rootContext,
 DefaultMutableTreeNode node) throws NamingException {
     ...
  NamingEnumeration list = rootContext.listBindings("");
     while (list.hasMore()) {
    ...
        Binding binding = (Binding)list.next();
        String name   = binding.getName();
    ...
    Object object  = binding.getObject();
    ...
    if ( object instanceof DirContext ) {
           ...
           DirContext subContext  = (DirContext)object;
           ...

     } else {
    Person person = (Person)object;
    ...
  }
  ...
}
```

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
```

könnte zum Beispiel auf das Dateisystem zugegriffen werden.

Die Parameter werden über eine Hashtable dem Konstruktor der Klasse *InitialDirContext* übergeben. Es besteht gleichzeitig die Möglichkeit, diese Parameter in eine Property-Datei auszulagern, wodurch die Parameter ohne eine Neukompilation des Codes geändert werden können (Listing 1).

Listing 2 zeigt wie der Inhalt eines Context ausgelesen wird. Da in unserem Adressbuch Untergruppen gebildet werden können, muss das zurückgegebene Objekt darauf geprüft werden, ob dieses eine Untergruppe repräsentiert. Ist der Name eines Eintrags bekannt, so kann auch über die Methode *lookup* direkt auf diesen zugegriffen werden.

Wie bereits erwähnt wurde, erlaubt ein DirContext neben der Zuordnung von Namen zu Objekten auch die Zuordnung von Attributen. Daher bietet DirContext Methoden, um auf diese Attribute zuzugreifen. Objekte können so wie bei Naming Services über ihren Namen und zusätzlich über ihre Attribute gesucht werden. Die Attribute eines Objekts können manipuliert werden. In Listing 3 werden gezielt bestimmte Attribute eines Objekts gelesen bzw. modifiziert.

Die Methode *createPerson* in Listing 4 zeigt wie ein neues Objekt unter einem bestimmten Namen und mit bestimmten Attributen angelegt wird. Über die Methoden *rebind* und *unbind* kann das betreffende Objekt gelöscht bzw. ausgetauscht werden.

Bis jetzt wurden die grundsätzlichen Funktionalitäten von JNDI aufgezeigt. Wie wir am Anfang dieses Artikels erwähnt haben, wurde der Zugriff auf die Einträge des Adressbuchs auf verschiedene Arten realisiert. Ein LDAP-Server kann im Grunde genommen keine Java-Objekte verwalten. Diese müssen in eine Form umgewandelt werden, die vom Server verstanden wird. Die Umwandlung von Objekten zu einer für den LDAP-Server verständlichen Form und zurück wird normalerweise vom Service Provider übernommen. In der ersten Ansicht unseres Beispiels haben wir diese Umwandlung in der Applikation selbst vorgenommen. Ein *Person*- Objekt wird in Attribute zerlegt und diese werden in einem DirContext gespeichert. Beim Auslesen wird aus diesen Attributen wieder ein Objekt erzeugt (Listing 3).

Um diese Verantwortung aus der Applikation zu nehmen, bietet das JNDI-

Listing 3: Attribute auslesen und modifizieren – Klasse *AttributeHandler*

```
public Person getPerson( DirContext rootContext, String personPath ) throws NamingException {

    ...
    String[] attributeNames = { SN, GIVENNAME, MAIL, TELEPHONENUMBER };
    Attributes attributes  = rootContext.getAttributes(relativePath, attributeNames);

    String lastName     = (String)attributes.get(SN).get();
    String firstName    = (String)attributes.get(GIVENNAME).get();
    String mail         = (String)attributes.get(MAIL).get();
    String telephoneNumber = (String)attributes.get(TELEPHONENUMBER).get();
    ...
  }
...
public void modifyPerson( DirContext rootContext, String entryPath, Person person ) throws NamingException {
    Name relativePath = getRelativePath( rootContext,entryPath );
    ModificationItem[] mods = new ModificationItem[2];
    mods[0] = new ModificationItem( DirContext.REPLACE_ATTRIBUTE, new BasicAttribute( MAIL, person.getMail() ));
    mods[1] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE, new BasicAttribute
                                                    ( TELEPHONENUMBER, person.getTelephoneNumber() ));

    rootContext.modifyAttributes(relativePath, mods );
  }
```

Listing 4: Objekt anlegen, modifizieren und löschen – Klasse *SerializationHandler*

```
public void createPerson( DirContext rootContext, String groupPath, Person person ) throws NamingException {
  ...
    subContext.bind( entryName, person,new Basic-Attributes(CN, person.getFirstName() + " " + person.getLastName()) );
 }
...
public void modifyPerson( DirContext rootContext, String entryPath, Person person ) throws NamingException {
  ...
    rootContext.rebind(entryPath, person );
 }
public void deletePerson( DirContext rootContext, String entryPath) throws NamingException {
  ...
  rootContext.unbind(entryPath);
 }
```

Framework die Verwendung von Factories. Factories übernehmen die nötigen Umwandlungen und werden vom Framework aufgerufen. Es gibt zwei Arten von Factories. StateFactories sind für die Umwandlung von Objekten zu Attributen und ObjectFactories für die Umwandlung von Attributen zu Objekten verantwortlich. Mit den folgenden Parametern beim Erzeugen des InitialDirContext-Objekts wird dem JNDI-Framework mitgeteilt, welche Factories zur Verfügung stehen:

```
env.put("java.naming.factory.object",
                                "PersonObjectFactory");
env.put("java.naming.factory.state",
                                "PersonStateFactory");
```

Soll ein Objekt umgewandelt werden, so werden automatisch alle verfügbaren Factories aufgerufen, bis eine zuständige Factory gefunden wird. Jede Factory entscheidet selbst, ob sie für ein bestimmtes Objekt zuständig ist (Listing 5). Wird keine passende Factory gefunden, versucht das JNDI-Framework das Objekt in serialisierter Form zu speichern. Wird das Interface *java.oi.Serializable* vom zu speichernden Objekt nicht implementiert, so kann dieses nicht serialisiert werden und es wird eine Exception geworfen.

Wir haben in unserem Beispiel sowohl den Zugriff über serialisierte Objekte wie auch über Factories realisiert. Dazu besitzt die *Person*-Klasse ein Flag, welches je nach gewünschtem Zugriff gesetzt wird. Die State Factory, welche bei jeder Speicherung aufgerufen wird, überprüft dieses Flag. Ist es nicht gesetzt, so wird das Objekt von der Factory nicht verarbeitet, wodurch es automatisch durch das Framework serialisiert wird. Bei der Speicherung wird zusätzlich ein *Dummy*-Attribut gesetzt, um der zugehörigen Object Factory zu signalisieren, dass sie für die Wiedererstellung des Objekts zuständig ist. Dieses Vorgehen ist nur notwendig, da wir in einer Applikation mehrere Zugriffsarten demonstrieren wollten. Im realen Einsatz würde man einen bestimmten Objekttyp stets auf die gleiche Art behandeln. Es ist noch zu bemerken, dass von Seiten der Applikation der Zugriff in beiden Fällen genau der gleiche ist.

Listing 5: Factories

```
// PersonStateFactory
  public DirStateFactory.Result getStateToBind(Object obj,
                  Name name, Context ctx, Hashtable env,
                                  Attributes inAttrs)
  throws NamingException {

    // Only interested in Person objects
    if (obj instanceof Person && ((Person)obj).
                          getAStoreItThroughFactory ()) {
    ...
    Person per = (Person)obj;
    if (per.getLastName() != null) {
        outAttrs.put("sn", per.getLastName());
    }else{...}
    if (per.getFirstName() != null) {
       outAttrs.put("givenName", per.getFirstName());
    } else{...}
  ...
    return new DirStateFactory.Result(null, outAttrs);
  }
  return null;
  }

// PersonObjectFactory
  public Object getObjectInstance(Object obj, Name name,
                  Context ctx, Hashtable env, Attributes attrs)
  throws Exception {
    Attribute departmentNumber = (attrs != null ? attrs.get
                                  ("departmentNumber") : null);
    if (departmentNumber != null ) {
       Person per = new
Person((String)attrs.get("sn").get(), (String)attrs.get
                  ("givenName").get(),(String)attrs.get
          ("telephoneNumber").get(),(String)attrs.get
                                  ("mail").get());
    return per;
  }
  return null;
  }
```

Fazit

Zentrale Naming- und Directory-Dienste werden mehr und mehr zu einem integralen Bestandteil moderner IT-Infrastrukturen. JNDI ist ein einfaches, leicht zu erlernendes API, mit dem Applikationen effizient und einheitlich auf verschiedenste Dienste zugreifen können. Seine Architektur ermöglicht es, eigene Service Provider zu implementieren, wodurch auch zukünftige Dienste eingebunden werden können. Durch den einheitlichen Zugriff und seine Erweiterbarkeit erweisen sich JNDI-basierende Architekturen als flexibel und zukunftssicher. Nicht umsonst ist JNDI ein grundlegender Bestandteil der EJB-Spezifikation. ■

Dipl. Ing (M.E) Muhammet Öztürk arbeitet als Projektmanager bei der Object International Software GmbH. Er ist Sun zertifizierter Programmer und Developer. mailto:oeztuerk@oisoft.com. Dipl. Inf. Ralf Bratenberg ist ebenfalls Sun Certified Programmer und arbeitet als Berater bei der Object International Software GmbH. mailto:bratenberg@oisoft.com

Links & Literatur

[1] www.devx.com/upload/free/features/javapro/1999/12dec99/sg1299/sg1299.asp

[2] www.javaworld.com/javaworld/jw-01-2000/jw-01-howto.html

[3] www.openldap.org/

[4] Erich Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison-Wesley, 1997

[5] Sourcecode zum Artikel: www.object-international.com/news&events/fachartikel