Client/Server-Programmierung II

EJB / JSP: Schritt-für-Schritt Anleitung

Version 1.3, 22.12.2016

Eingesetzte Software:

- Apache Tomcat 6.0.18 (http://archive.apache.org/dist/tomcat/tomcat-6/v6.0.18/bin/apache-tomcat-6.0.18.zip)
- OpenEJB 7.0.2 (http://tomee.apache.org/downloads.html; 'OpenEJB Standalone')

Hinweis:

Bevor mit dem EJB-Container OpenEJB im Labor H-A 4111 das erste Mal gearbeitet werden kann, ist das Installationsskript `/opt/dist/tools/openejb_install.sh` auszuführen. Sie sollten in Ihre `$HOME/.profile`-Datei dann die folgenden Anweisungen einfügen:

```
export OPENEJB_HOME=$HOME/Soft/openejb-7.0.2
export PATH=$OPENEJB_HOME/bin:$PATH
```

Siehe dazu auch Abschnitt 5.4.5 des Vorlesungsskripts.

EJB Session Beans

Im folgenden Beispiel wird von der Implementierung eines einfachen Hello-World-Programms ausgegangen und der Vorgang bis zum Deployment als *stateless Session-Bean* beschrieben.

Zunächst sind zwei Verzeichnisse zu erstellen. Das eine heißt `META-INF` und beinhaltet den XML-Deployment-Deskriptor `ejb-jar.xml`, der im einfachsten Fall folgendermaßen aussieht:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar/>
```

Das zweite Verzeichnis beinhaltet das Paket mit den Java-Dateien der Bean, in unserem Fall haben wir ein Verzeichnis `org` mit Unterverzeichnis `Hello` (also `org/Hello`). Dort werden zwei Dateien erstellt. Die Datei `HelloRemote.java` beinhaltet das *Remote Interface*:

```
package org.Hello;
import javax.ejb.Remote;

@Remote
public interface HelloRemote
{
  public String sayHello();
}
```

Die Datei `HelloImpl.java` beinhaltet die Implementierung der eigentlichen EJB, welche auf dem Server im EJB-Container OpenEJB ausgeführt werden soll:

```
package org.Hello;
import javax.ejb.Stateless;

@Stateless
public class HelloImpl implements HelloRemote
{
  public String sayHello()
  {
    return "Hallo? Jemand da?";
  }
}
```

Nun erfolgt eine Kompilierung der Java-Dateien mittels

```
javac -cp $OPENEJB_HOME/lib/javaee-api-7.0-1.jar:. org/Hello/*.java
```

Im nächsten Schritt erfolgt die Paketierung der Bean. Dies geschieht mit

```
jar cvf myHelloEjb.jar org/Hello/*.class META-INF
```

Das entstandene Paket mit dem Namen `myHelloEjb.jar` müssen Sie nun dem OpenEJB-Container bekannt machen. Dazu starten Sie zunächst den Container (sofern er noch nicht aktiv ist), indem Sie

```
openejb start
```

ausführen. Das Deployment erfolgt dann mit

```
openejb deploy –s ejbd://localhost:<port> myHelloEjb.jar
```

Dabei ist *<port>* durch den Ihnen zugewiesenen Port des EJB-Containers zu ersetzen. Diesen gibt der Container beim Start aus (in der Zeile, die mit '`ejbd`' beginnt). Falls bereits eine EJB desselben Namens *deployt* ist, geben Sie zusätzlich die Option –u an. Sie legt fest, dass zuerst ein *Undeployment* der EJB gemacht werden soll. Ohne diese Option, erhalten Sie in diesem Fall eine Fehlermeldung.

Nun steht der Dienst der EJB zur Verfügung. Falls Sie im *Deployment*-Deskriptor den Zugriff auf einige (oder alle) Dienste beschränkt haben (siehe Vorlesung Kap. 5.4.7), müssen Sie nun ggf. noch die beiden folgenden Dateien editieren:

- `$OPENEJB_HOME/conf/users.properties`
  Diese Datei enthält die erlaubten Benutzernamen und die zugehörigen Passworte in der Syntax:
  benutzername=passwort
- $`OPENEJB_HOME`/conf/groups.properties
  In dieser Datei werden die vorhandenen Gruppen (Rollen) mit ihren Mitgliedern definiert. Die Syntax eines Eintrags ist:
  gruppenname=benutzername{**,***benutzername*}

Um den Dienst nutzen zu können, benötigen Sie nun noch eine Java-Applikation (im Beispiel `HelloClient.java`) als Client. Es wird einfach über JNDI eine Referenz auf das EJB-Objekt geholt, auf das dann genauso zugegriffen werden kann, wie auf ein lokales Objekt:

```
import org.Hello.*;
import javax.naming.InitialContext;
import java.util.Properties;

public class HelloClient
{
```

```
   public static void main(String args[])
   {
      try
      {
         Properties p = new Properties();
         p.put("java.naming.factory.initial",
               "org.apache.openejb.client.RemoteInitialContextFactory");
         p.put("java.naming.provider.url", "127.0.0.1:4201");
         p.put("java.naming.security.principal", "Benutzername");
         p.put("java.naming.security.credentials", "Passwort");
         InitialContext ctx = new InitialContext(p);

         Object obj = ctx.lookup("HelloImplRemote");
         HelloRemote hello = (HelloRemote)obj;
         System.out.println(hello.sayHello());
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

Beachten Sie, dass Sie beim Setzen der Ressource `java.naming.provider.url` den Port und ggf. auch die IP-Adresse bzw. den Rechnernamen des OpenEJB-Containers anpassen müssen! Ebenso müssen Sie Benutzernamen und Passwort anpassen.

Um den Client zu kompilieren, benötigt er neben dem Archiv `javaee-api-7.0-1.jar` die Interface-Datei der EJB (`HelloRemote.class`) in seinem *Classpath*. Kopieren Sie diese Datei (oder besser die zugehörige Java-Datei) in ein relatives Unterverzeichnis `org/Hello` zu der Client-Applikation `HelloClient.java`. Die Kompilierung wird dann folgendermaßen durchgeführt:

```
javac -cp $OPENEJB_HOME/lib/javaee-api-7.0-1.jar:. HelloClient.java
```

Damit JNDI die `RemoteInitialContextFactory` von OpenEJB finden kann, müssen Sie beim Start des Clients auch noch mindestens das Archiv `openejb-client-7.0.2.jar` im *Classpath* führen. Der Client wird daher folgendermaßen ausgeführt (die Zeilenumbrüche sind nur des Drucks wegen vorhanden und dürfen nicht eingegeben werden!):

```
java  -cp $OPENEJB_HOME/lib/javaee-api-7.0-1.jar:
          $OPENEJB_HOME/lib/openejb-client-7.0.2.jar:.
      HelloClient
```

EJB Entities

Wenn Sie in Ihrer EJB-Anwendung *Entities* verwenden wollen, müssen Sie zunächst eine Klasse für die *Entity* erstellen, z.B.:

```
package org.Hello;
import javax.persistence.*;

@Entity
public class Account implements java.io.Serializable {
  @Id
```

```
  private int accountNo;
  private String name;

  public int getAccountNo() { return accountNo; }
  public String getName() { return name; }
  public void setName(String nm) { name = nm; }

  public Account(int no, String nm) {
    accountNo = no;
    name = nm;
  }
}
```

In der Session-Bean benötigen Sie dann eine Referenz auf einen entsprechenden *Entity Manager*, die Sie durch *Dependency Injection* wie folgt erhalten können:

```
package org.Hello;
import javax.ejb.*;
import javax.persistence.*;

@Stateless
public class BankImpl implements BankRemote
{
  @PersistenceContext(unitName="intro")
  private EntityManager manager;
  ...
```

Der Parameter `unitName` verweist dabei auf die zu verwendende *Persistence Unit*, die im *Deployment*-Deskriptor `META_INF/persistence.xml` der *Entity* näher beschrieben sein muss:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
version="1.0">
  <persistence-unit name="intro">
    <jta-data-source>My DataSource</jta-data-source>
    <non-jta-data-source>My Unmanaged DataSource</non-jta-data-source>
    <class>org.Hello.Account</class>

    <properties>
      <property name="openjpa.jdbc.SynchronizeMappings"
                value="buildSchema(ForeignKeys=true)"/>
    </properties>
  </persistence-unit>
</persistence>
```

Dieser *Deployment*-Deskriptor nimmt Bezug auf eine Datenquelle, die im Beispiel '`My DataSource`' bzw. '`My Unmanaged DataSource`' heißt, je nachdem ob JTA-Unterstützung gefordert ist oder nicht. Diese Datenquellen wiederum werden bei OpenEJB in der Konfigurationsdatei `$OPENEJB_HOME/conf/openejb.xml` näher spezifiziert:

```
<Resource id="My DataSource" type="DataSource">
  JdbcDriver org.hsqldb.jdbcDriver
  JdbcUrl jdbc:hsqldb:file:data/hsqldb/hsqldb
  UserName sa
  Password
  JtaManaged true
</Resource>
<Resource id="My Unmanaged DataSource" type="DataSource">
  JdbcDriver org.hsqldb.jdbcDriver
```

```
  JdbcUrl jdbc:hsqldb:file:data/hsqldb/hsqldb
  UserName sa
  Password
  JtaManaged false
</Resource>
```

Angegeben sind hier der zu verwendende JDBC-Treiber, die URL der Datenbank, sowie der Benutzername und das Passwort für den Datenbank-Zugriff. Die oben gezeigten Einträge sind standardmäßig in OpenEJB konfiguriert und verwenden das im Container eingebaute einfache Datenbanksystem HSQLDB.

Wenn Sie im obigen Beispiel nun in der *Session-Bean* einen neuen Datenbank-Eintrag erzeugen wollen, erzeugen Sie zunächst ein neues `Account`-Objekt und machen es dann über den *Entity-Manager* persistent:

```
Account acc = new Account(n, name);
manager.persist(acc);
```

Wegen des Eintrags

```
<property name="openjpa.jdbc.SynchronizeMappings"
          value="buildSchema(ForeignKeys=true)"/>
```

in `META_INF/persistence.xml` erzeugt OpenJPA dabei ggf. auch das Datenbank-Schema automatisch mit bzw. aktualisiert es (wenn Sie der *Entity* z.B. ein Attribut hinzufügen). Über den Methodenaufruf

```
Account acc = manager.find(Account.class, n);
```

erhalten Sie für einen gegebenen Primärschlüssel (hier '`n`') die zugehörige *Entity* (sofern für diesen Schlüssel ein Datenbank-Eintrag vorhanden ist). Mit dem Methodenaufruf

```
manager.remove(acc);
```

können Sie den zu einer *Entity* gehörigen Datenbank-Eintrag wieder löschen.

Manchmal kann es notwendig werden, die HSQL-Datenbank komplett zu löschen. Gehen Sie dazu wie folgt vor:

- Beenden Sie, falls nötig, den OpenEJB Container.
- Löschen Sie das Verzeichnis `$HOME/Soft/openejb-7.0.2/data/hsqldb`.

Wenn Sie zusätzlich zur HSQL-Datenbank auch die im Labor eingerichtete MySQL-Datenbank verwenden wollen, müssen Sie in der Konfigurationsdatei `$OPENEJB_HOME/conf/openejb.xml` folgende Einträge ergänzen:

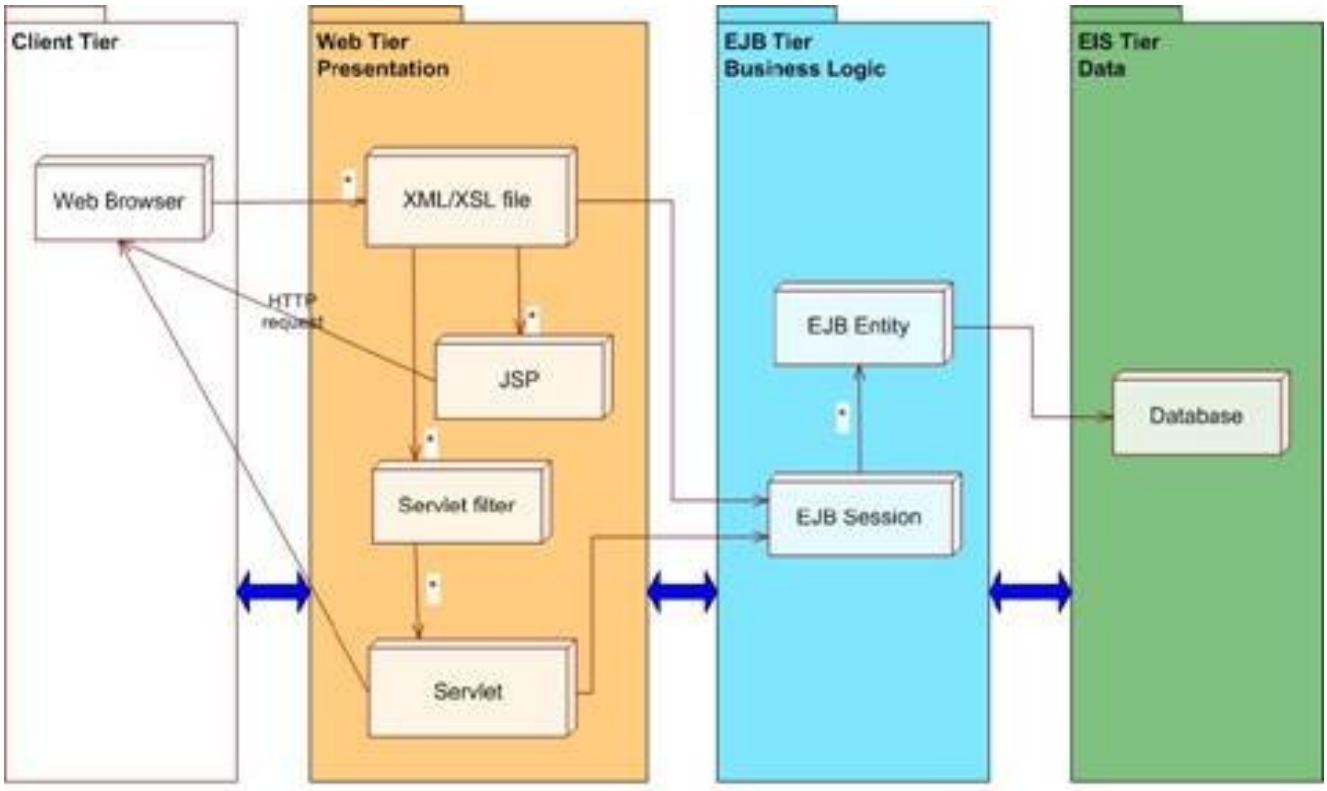
```
<Resource id="My SQL DB" type="DataSource">
  JdbcDriver com.mysql.jdbc.Driver
  JdbcUrl jdbc:mysql://bslabserv01.lab.bvs/cspdb
  UserName
  Password
  JtaManaged true
</Resource>
<Resource id="My Unmanaged SQL DB" type="DataSource">
  JdbcDriver com.mysql.jdbc.Driver
  JdbcUrl jdbc:mysql://bslabserv01.lab.bvs/cspdb
  UserName
```

```
    Password
    JtaManaged false
  </Resource>
```

Natürlich müssen Sie dann im *Deployment*-Deskriptor der entsprechenden *Entities* auch auf diese Datenquelle Bezug nehmen. Damit der OpenEJB-Container den MySQL-Treiber findet, müssen Sie die entsprechende JAR-Datei in das Verzeichnis `$OPENEJB_HOME/lib` kopieren. Beachten Sie, dass die MySQL-Datenbank im Labor nur lesenden Zugriff erlaubt.

JSP & EJB

Wie eine *Bean* mit Tomcat als *Bean-Container* angesprochen wird, ist bereits auf dem Übungsblatt erläutert. Im Rahmen des Tutorials wird nun dargelegt, wie man mittels eines client-seitigen Internet-Browsers über JSP auf einem Web-Server auf eine EJB zugreift. Die client-seitige Java-Applikation wird dadurch ersetzt. Die JSP-Seiten bauen eine Verbindung zu dem EJB-Container OpenEJB als Applikationsserver auf, der wiederum mit einer Datenbank, z.B. MySQL, kommunizieren kann. Auf diese Weise ergibt sich eine klassische 4-Tier-Architektur:



Als erste Voraussetzung sollten Sie prüfen, ob Ihre Tomcat-Konfigurationsdatei `$CATALINA_BASE/conf/catalina.properties` die folgende Definition enthält und diese ggf. ergänzen:

```
shared.loader=${catalina.base}/shared/lib/*.jar
```

Dieser Eintrag weist Tomcat an, in dem angegebenen Verzeichnis nach gemeinsamen JAR-Archiven für alle JSPs zu suchen. Als nächstes müssen Sie die JAR-Dateien

- `openejb-client-7.0.2.jar`
- `javaee-api-7.0-1.jar`

aus dem Verzeichnis `$OPENEJB_HOME/lib` des OpenEJB-Containers in das Verzeichnis `$CATALINA_BASE/shared/lib` des Tomcat kopieren. Dieses Verzeichnis ist ggf. vorher zu erstellen. Nun ist Tomcat in der Lage, mit dem EJB-Container zusammenzuarbeiten.

Nach dem Deployment der HelloWorld-EJB und dem Start des OpenEJB-Containers kann nun die HelloWorld-EJB über die folgende JSP-Datei angesprochen werden. Diese sollte im Verzeichnis `$CATALINA_BASE/webapps/ROOT` von Tomcat angelegt werden und die Endung `.jsp` besitzen (wieder müssen ggf. Host und Port der `java.naming.provider.url` sowie Benutzername und Passwort angepasst werden):

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en"
  "http://www.w3.org/TR/REC-html40/strict.dtd">
<%@ page import="org.Hello.*,
                 javax.naming.InitialContext,
                 java.util.Properties" %>
<html>
  <body>
    <%
        Properties p = new Properties();
        p.put("java.naming.factory.initial",
              "org.openejb.client.RemoteInitialContextFactory");
        p.put("java.naming.provider.url", "127.0.0.1:4201");
        p.put("java.naming.security.principal", "Benutzername");
        p.put("java.naming.security.credentials", "Passwort");
        InitialContext ctx = new InitialContext(p);
        Object obj = ctx.lookup("HelloImplRemote");
        HelloRemote hello = (HelloRemote)obj;
        out.println(hello.sayHello());
    %>
  </body>
</html>
```

Damit die JSP-Seite das Interface `HelloRemote` laden kann, muss die Datei `HelloRemote.class` in das Verzeichnis `$CATALINA_BASE/webapps/ROOT/WEB-INF/classes/org/Hello` kopiert werden (Verzeichnis ggf. erstellen; im Allgemeinen muss es ein Verzeichnis `WEB-INF/classes` relativ zur Position der JSP-Datei sein). Beachten Sie, dass bei jeder Änderung von `class`- oder `jar`-Dateien der Tomcat-Server neu gestartet werden sollte!

Der Start der Anwendung erfolgt nun über den Internet-Browser. Da die Referenz `hello` nur für eine JSP-Datei gilt, muss diese Referenz bei Einsatz einer *stateful Session-Bean*, wie es bei der Börsenanwendung notwendig ist, als *Session*-Attribut in Tomcat gehalten werden. Dies geschieht über das Anlegen des Attributs, sofern es in der Tomcat-*Session* noch nicht vorhanden ist. Ist es bereits vorhanden, so wird die dort gespeicherte Referenz weiter verwendet:

```
Depot depot;
// Versuch, Session-Objekt holen
depot = (Depot)session.getAttribute("DepotSession");
if (depot == null) {
   // wenn Session-Objekt nicht existiert -> anlegen!
   Properties p = new Properties();
   p.put("java.naming.factory.initial",
         "org.openejb.client.RemoteInitialContextFactory");
   p.put("java.naming.provider.url", "127.0.0.1:4201");
   p.put("java.naming.security.principal", "Benutzername");
   p.put("java.naming.security.credentials", "Passwort");
   InitialContext ctx = new InitialContext(p);
   depot = (Depot)ctx.lookup("DepotImplRemote");
```

```
   session.setAttribute("DepotSession", depot);
}
// Aufrufen und Auswerten der Methode aus der EJB
... = depot.buy(...)
```

Ein Hinweis zur Sicherheit von JSP unter Tomcat:

Wenn der Java-Code in der JSP-Seite nicht korrekt übersetzt werden kann oder eine Exception wirft, sendet Tomcat standardmäßig eine HTML-Seite an den Browser, die zum einen die Exception-Meldung enthält, zum anderen aber auch einen Ausschnitt des JSP-Codes. Das bedeutet, dass in den obigen Beispielen u.U. das OpenEJB Passwort nach außen gegeben werden kann! Zur Abhilfe editieren Sie die Datei `$CATALINA_BASE/conf/web.xml`, suchen dort den Eintrag

```
<servlet>
   <servlet-name>jsp</servlet-name>
   <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
```

und fügen die folgende Definition ein:

```
      <init-param>
         <param-name>displaySourceFragment</param-name>
         <param-value>false</param-value>
      </init-param>
```