

EJB 3.0 and JPA

The old model is rather complex and it is difficult to implement and to deploy EJB's. Therefore a new specification has been published. This is called EJB 3.0.

The old EJB did have a lot of contractual requirements such as Home interface, EJB object interface, bean class etc. The object interface is indirectly implemented through mappings in the container.

In the new EJB you write a business interface that specifies the methods you need and then implements this interface as an ordinary Java class, known as a POJO (Plain Old Java Object).

The container still needs to be involved in the bean management. To achieve this, basically two techniques are used

- Annotations
- Injections

Annotations is a standard Java technique to associate features with objects and other java components. That is, you add information about your code that can be used by different tools that uses the code.

Might look like this

```
@CodeCategory(visibility = Visibility.PUBLIC,  
              isUnitTested = true,  
              isReviewed = true)  
public class ClassTwo {  
    ...  
}
```

This add some bits of information to the class.

The annotation used has to be declared:

```
package examples.annotations;  
  
import java.lang.annotation;  
  
enum Visibility {  
    INTERNAL, PUBLIC  
}  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface CodeCategory(  
    Visibility visibility() default Visibility.INTERNAL;  
    boolean isUnitTested() default false;  
    boolean isReviewed() default false;  
)
```

To access annotations at runtime you use the Java reflection API (*getAnnotation* method).

The EJB 3 specification implies

- No home or object interface
- No component interface is required
- Use of Java metadata annotations
- Simplifications of the APIs for accessing bean's environment
- Injection is used by the container to insert data in my code
- Entity beans are not part of EJB 3, they are still using EJB 2.1. However a new API, JPA (Java Persistence API) is introduced.

Instead of a home interface and an object interface, a business interface is used. This is either remote, local or both. It defines signatures for our business methods that my EJB should implement. All complexity with Remote Exceptions is encapsulated within the container.

The component interface lists different methods that the container can use to notify the bean about different events like EJB creation and destruction. These are now introduced as callback methods in a separate class or directly implemented in my EJB. In both cases annotation is used.

Annotations are used like

```
@Stateful
public class exampleBean implements
    BeanBusinessInterface
{
    @Remove
    public void removeBean()
    {
        // close any resources
    }
}
```

Most of the information given as Annotation can also be specified in the deployment descriptor.

One reason for this is that annotations are compiled into the code and access to the source is required to change them. The deployment descriptor can be used to override them.

The old EJB specification relies on JNDI to get hold of environmental entries like home interfaces.

In EJB 3.0, dependency injections and a simple EJB-Context object is used to lookup resources.

Dependency injections is a mechanism followed by the container to inject the requested environmental entry and make it available to the bean instance, before any business are invoked on that particular instance.

The bean provider has to inform the container about what to inject using annotations or deployment descriptors. The container uses the Java Bean naming conventions to actually do the injection.

To implement an EJB 3.0 bean you do

- Write the java code for the business interface and the bean class
- Compile the Java sources in step 1
- Provide a deployment descriptor
- Create the EJB-jar file containing classes generated above
- Deploy the EJB in a your container
- Write a client to test the bean

The business interface:

```
package examples.session.stateless;
```

```
/**
```

```
 * This is the Hello business interface
```

```
 */
```

```
public interface Hello {  
    public String hello();  
}
```

The bean class

```
package examples.session.stateless;

import javax.ejb.Remote;
import javax.ejb.Stateless;

/**
 * stateless session bean
 */

@Stateless
@Remote(Hello.class)
public class HelloBean implements Hello {
    public String hello() {
        System.out.println("Hello()");
        return "Hello, World!";
    }
}
```

We don't really need a deployment descriptor for this but there is no resources used here.

A skeleton could be

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns=
    "http://java.sun.com/xml/ns/javaee"
    xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
    full="false" version="3.0"
    xsi:schemaLocation=
    "http://java.sun.com/xml/ns/javaee http://
    java.sun.com/xml/ns/javaee/.ejb-jar_3_0.xsd">

    <enterprise-beans>
    </enterprise-beans>
</ejb-jar>
```

An other advantage with EJB 3.0 is that a client can be any Java program, it doesn't have to be in a container. A client example is:

```
package examples.session.stateless;

import javax.naming.Context;
import javax.naming.InitialContext;

/**
 * This class is an example of client code which
 * invokes methods on a simple, remote stateless
 * session bean.
 */

public class HelloClient {

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

    /**
     * Obtain the JNDI initial context.
     *
     * The initial context is a starting point for
     * connecting to a JNDI tree. We
     * choose our JNDI driver, the network
     * location of the server, etc
     * by passing in the environment
     * properties.
     */

        System.out.println(
```

```

        "about to create initialcontext");

        Context ctx = new InitialContext();

        System.out.println(
            "Got initial context ... yeah ");

        /*
         * Get a reference to a bean instance,
         * looked up by class name
         */

        Hello hello
            = (Hello) ctx.lookup("HelloBean");

        /*
         * Call the hello() method on the bean.
         * We then print the result to the screen.
         */

        System.out.println(hello.hello());
    }
}

```

The example from last time

```
/*  
 * Copyright 2007 Sun Microsystems, Inc.  
 * All rights reserved. You may not modify, use,  
 * reproduce, or distribute this software except in  
 * compliance with the terms of the License at:  
 * http://developer.sun.com/berkeley\_license.html  
 */
```

```
package converter.ejb;
```

```
import java.math.BigDecimal;
```

```
import javax.ejb.Remote;
```

```
@Remote
```

```
public interface Converter {
```

```
    public BigDecimal dollarToYen(  
                                     BigDecimal dollars);
```

```
    public BigDecimal yenToEuro(  
                                   BigDecimal yen);
```

```
}
```

```

/*
 * Copyright 2007 Sun Microsystems, Inc.
 * All rights reserved. You may not modify, use,
 * reproduce, or distribute this software except in
 * compliance with the terms of the License at:
 * http://developer.sun.com/berkeley\_license.html
 */

package converter.ejb;

import java.math.BigDecimal;
import javax.ejb.Stateless;

/**
 * This is the bean class for the
 * ConverterBean enterprise bean.
 * Created Jan 20, 2006 1:14:27 PM
 * @author ian
 */

@Stateless
public class ConverterBean
    implements converter.ejb.Converter {
    private BigDecimal euroRate =
        new BigDecimal("0.0070");
    private BigDecimal yenRate =
        new BigDecimal("112.58");

    public BigDecimal dollarToYen(
        BigDecimal dollars) {
        BigDecimal result =
            dollars.multiply(yenRate);
    }
}

```

```
        return result.setScale(2,
                               BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(
                               BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2,
                               BigDecimal.ROUND_UP);
    }
}
```

```
/*  
 * Copyright 2007 Sun Microsystems, Inc.  
 * All rights reserved. You may not modify, use,  
 * reproduce, or distribute this software except in  
 * compliance with the terms of the License at:  
 * http://developer.sun.com/berkeley\_license.html  
 */
```

```
package converter.client;
```

```
import converter.ejb.Converter;  
import java.math.BigDecimal;  
import javax.ejb.EJB;
```

```
/**  
 *  
 * @author ian  
 */
```

```
public class ConverterClient {  
    @EJB  
    private static Converter converter;  
  
    /** Creates a new instance of Client */  
    public ConverterClient(String[] args) {  
    }  
}
```

```
/**  
 * @param args the command line arguments  
 */
```

```
public static void main(String[] args) {  
    ConverterClient client =
```

```

        new ConverterClient(args);
    client.doConversion();
}

public void doConversion() {
    try {
        BigDecimal param =
            new BigDecimal("100.00");
        BigDecimal yenAmount =
            converter.dollarToYen(param);
        System.out.println("$" + param + " is "
            + yenAmount + " Yen.");

        BigDecimal euroAmount =
            converter.yenToEuro(yenAmount);
        System.out.println(yenAmount + " Yen is "
            + euroAmount + " Euro.");

        System.exit(0);
    } catch (Exception ex) {
        System.err.println(
            "Caught an unexpected exception!");
        ex.printStackTrace();
    }
}
}
}

```

```

<%@ page import="converter.ejb.Converter,
                java.math.*, javax.naming.*"%>

<%!

    private Converter converter = null;

    public void jspInit() {

    try {
        InitialContext ic = new InitialContext();
        converter = (Converter) ic.lookup(
            Converter.class.getName());
    } catch (Exception ex) {
        System.out.println(
            "Couldn't create converter bean."
            + ex.getMessage());
    }
    }

    public void jspDestroy() {
        converter = null;
    }
%>

<html>
    <head>
        <title>Converter</title>
    </head>

    <body bgcolor="white">
        <h1>Converter</h1>

```

```

<hr>
<p>Enter an amount to convert:</p>
<form method="get">
  <input type="text" name="amount"
        size="25">
  <br>
  <p>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
  </p>
</form>

```

```

<%
  String amount =
    request.getParameter("amount");
  if ( amount != null && amount.length() > 0 ) {
    BigDecimal d = new BigDecimal(amount);

    BigDecimal yenAmount =
      converter.dollarToYen(d);
  %>
  <p>
    <%= amount %> dollars are
      <%= yenAmount %> Yen.
  </p>
  <%
    BigDecimal euroAmount =
      converter.yenToEuro(yenAmount);
  %>
  <%= yenAmount %> Yen are
    <%= euroAmount %> Euro.
  <%

```

```
}  
%>
```

```
</body>  
</html>
```

Another example can be

```
package com.sun.firstcup.ejb;
```

```
import java.util.Date;  
import javax.ejb.Remote;
```

```
/**  
*  
* @author olle  
*/
```

```
@Remote  
public interface DukesBirthdayBeanRemote {  
    int getAgeDifference(Date date);  
}
```

```

package com.sun.firstcup.ejb;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.logging.Logger;
import javax.ejb.Stateless;

/**
 *
 * @author olle
 */

@Stateless
public class DukesBirthdayBeanBean implements
    DukesBirthdayBeanRemote {

    private static Logger logger =Logger.getLogger
        ("com.sun.firstcup.ejb.DukesBirthdayBean");

    public int getAgeDifference(Date date) {
        int ageDifference;

        Calendar theirBirthday = new GregorianCalendar();
        Calendar dukesBirthday =
            new GregorianCalendar(1995,
                Calendar.MAY, 23);
        // Set the Calendar object to the passed in Date

        theirBirthday.setTime(date);
    }
}

```

```

        // Subtract the user's age from Duke's age

ageDifference =
    dukesBirthday.get(Calendar.YEAR) -
    theirBirthday.get(Calendar.YEAR);
logger.info("Raw ageDifference is: " + ageDifference);

// Check to see if Duke's birthday occurs before the
// user's. If so subtract one from the age difference

if(dukesBirthday.before(theirBirthday) &&
    (ageDifference > 0)){
    ageDifference--;
}
logger.info(
    "Final ageDifference is: " + ageDifference);
return ageDifference;
}
}

```

The client is a bean that runs as part of a standard web application inside a container.

```
package com.sun.firstcup.web;

import com.sun.firstcup.ejb.DukesBirthdayBeanRemote;
import java.util.Date;
import javax.ejb.EJB;

/**
 * @author olle
 */

public class DukesBDay {

    @EJB
    private DukesBirthdayBeanRemote dukesBirthday;
    private int age;
    private Date yourBD;
    private int ageDiff;
    private int absAgeDiff;

    public DukesBDay(){
        age=-1;
        yourBD = null;
        ageDiff = -1;
        absAgeDiff = -1;
    }

    public int getAge() {
        try { // Call Web Service Operation
            com.sun.firstcup.webservice.DukesAgeService service =
                new com.sun.firstcup.webservice.DukesAgeService();
            com.sun.firstcup.webservice.DukesAge port =
                service.getDukesAgePort(
);
            // TODO process result here

            age = port.getDukesAge();
        } catch (Exception ex) {

            // TODO handle custom exceptions here

```

```

    }

    return age;
}

public void setAge(int age) {
    this.age = age;
}

public Date getYourBD() {
    return yourBD;
}

public void setYourBD(Date yourBD) {
    this.yourBD = yourBD;
}

public int getAgeDiff() {
    ageDiff = dukesBirthday.getAgeDifference(yourBD);
    if(ageDiff < 0) {
        setAbsAgeDiff(Math.abs(ageDiff));
    }
    else {
        setAbsAgeDiff(ageDiff);
    }
    return ageDiff;
}

public void setAgeDiff(int ageDiff) {
    this.ageDiff = ageDiff;
}

public int getAbsAgeDiff() {
    return absAgeDiff;
}

public void setAbsAgeDiff(int absAgeDiff) {
    this.absAgeDiff = absAgeDiff;
}
}

```

JPA, the persistence API

JPA is an API to provide persistent storage for your object. This means that the state of the objects will be stored in a permanent storage.

JPA consists of

- The API
- The Query Language, EJB-QL
- Object/relational mapping metadata

This is not a new version of entity beans but something completely different

- Provides a standard for object-relational mapping (ORM).
- It is not tied to the Java EE container and can be tested and used with standard Java
- Defines a service provider interface so that the code can be developed independent of the database and other resources used.

ORM, the object relational mapping is a description of how my object structure should be mapped into a relational model.

An example is

```
public class BankAccount {  
    private String accountID;  
    private String ownerName;  
    private double balance;  
    ...  
}
```

that would map into a table with three columns using the accountID as the primary key.

Entities are persistent data objects within an application. They are simple or complex data that you want to save.

- They have a client visible persistent identity (the primary key).
- Entities have persistent, client visible state
- They are not remotely accessible
- Their lifetime is independent of the applications lifetime

A simple example is

```
package examples.entity.intro;
```

```
import java.io.Serializable;  
import javax.persistence.Entity;  
import javax.persistence.Id;
```

```
/**
```

```
 * This demo entity represents a Bank Account.
```

```
 * <p>
```

```
 * The entity is not a remote object and can only be  
 * accessed locally by
```

```
 * clients. However, it is made serializable so that
```

```
 * instances can be passed by
```

```
 * value to remote clients for local inspection.
```

```
 * <p>
```

```
 * Access to persistent state is by direct field access.
```

```
 */
```

```
@Entity
```

```
public class Account implements Serializable {
```

```
    /** The account number is the primary key for the  
        persistent object */
```

```
    @Id
```

```
    public int accountNumber;
```

```
    public String ownerName;
```

```
    public int balance;
```

```

/**
 * Entity beans must have a
 *   public no-arg constructor
 */

public Account() {

    // auto-generation

    accountNumber = (int) System.nanoTime();
}

public void deposit(int amount) {
    balance += amount,
}

public int withdraw(int amount) {
    if (amount > balance) {
        return 0;
    }
    else {
        balance -= amount;
        return amount;
    }
}
}

```

To use this in a stateless EJB you can do

```
package examples.entity.intro;

import java.util.List;
import javax.ejb.Stateless;
import javax.ejb.Remote;
import javax.persistence.*;

/**
 * Stateless session bean facade for account entities,
 * remotely accessible
 */

@Stateless
@Remote(Bank.class)

public class BankBean implements Bank {

/**
 * The entity manager object, injected by
 * the container
 */

    @PersistenceContext
    private EntityManager manager;
```

```
public List<Account> listAccounts() {  
    Query query = manager.createQuery(  
        "SELECT a FROM Account a");  
    return query.getResultList();  
}
```

```
public Account openAccount(String ownerName) {  
  
    Account account = new Account();  
    account.ownerName = ownerName;  
    manager.persist(account);  
    return account;  
}
```

```
public int getBalance(int accountNumber) {  
    Account account = manager.find(  
        Account.class, accountNumber);  
    return account.balance;  
}
```

```
public void deposit(int accountNumber,  
                    int amount) {  
    Account account = manager.find(  
        Account.class, accountNumber);  
    account.deposit(amount);  
}
```

```
public int withdraw(int accountNumber,  
                    int amount) {  
    Account account = manager.find(  
        Account.class, accountNumber);  
    return account.withdraw(amount);  
}  
  
public void close(int accountNumber) {  
    Account account = manager.find(  
        Account.class, accountNumber);  
    manager.remove(account);  
}  
}
```

To deploy this you need a descriptor file that set up the environment such as data source and other mapping. The name of this file is persistence.xml.

ORM states a number of problems that we haven't discusses here. These includes:

- How to map class hierarchies
- How to map relations of different kinds

JPA is implemented by a number of products. One of them is Hibernate.