Conception of Information Systems
Lecture 8: J2EE and EJBs

3 May 2005
http://lsirwww.epfl.ch/courses/cis/2005ss/

Outline

- Components
- J2EE and Enterprise Java Beans
  - EJB Architecture
  - Entity Bean vs. Session Bean
  - Finite State Machines
  - Transactions with EJBs

Components

- Components
  - self-contained, reusable, portable, configurable SW pieces
  - equipped to live within a specific environment = *container*
  - standardized interface to the container
  - introspection, configurability, packaging
- Component model: Contract between components and container
  - a set of interfaces and classes that a component uses and supports
  - Difficulty: combining ease of use with flexibility
- Distinctions to be made from the software engineering viewpoint
  - Classes and Objects
  - Design Patterns
  - Frameworks
  - Components

From a software engineering perspective the server-side objects that are developed for an OTM and that are taking advantage of the environment of the OTM are what is called *components*.

Components are characterized by the fact that they are self-contained pieces of software that can be deployed (without further implementation effort) in a specific environment, which is the container. The idea is that such an approach will allow to assemble complex software constructs from smaller pieces - the components -, which have proven functionality, can be reused in many different application contexts, and therefore also can be easily adapted to different application environments. This is also a reason why the deployment descriptor plays such a central role. The components interact with the container through a standardized interface. They support introspection, i.e. they allow tools to discover their interfaces, and configurability, for customizing their properties for the applications. A packaging mechanism is provided, that allows to bind together all the necessary parts of a component into one file, that can then be *deployed* on a container.

A *component model* describes the interface between components and the container. We may consider it as the language in which contracts between components and containers are established. The difficulty is to find for the component model a tradeoff between being sufficiently flexible and easy of use. For example, the generic CORBA model was arbitrarily flexible but very difficult to use.

Components are different from other paradigms for increasing reusability from software engineering.

Objects are encapsulated pieces of software that make up part of components, but are not self-contained. *Patterns* are proven solution methods that are described *textually* according to a specific method. *Frameworks* are partially implemented solutions of which the code must be completed.

Server-Side Components

- Client-side (e.g. Java beans) and server-side components
- Server-Side Components
  - serving multiple clients on a middle-tier server
  - transactional, persistent, secure, high-performing
  - Components once developed can be deployed on every application server (portability)

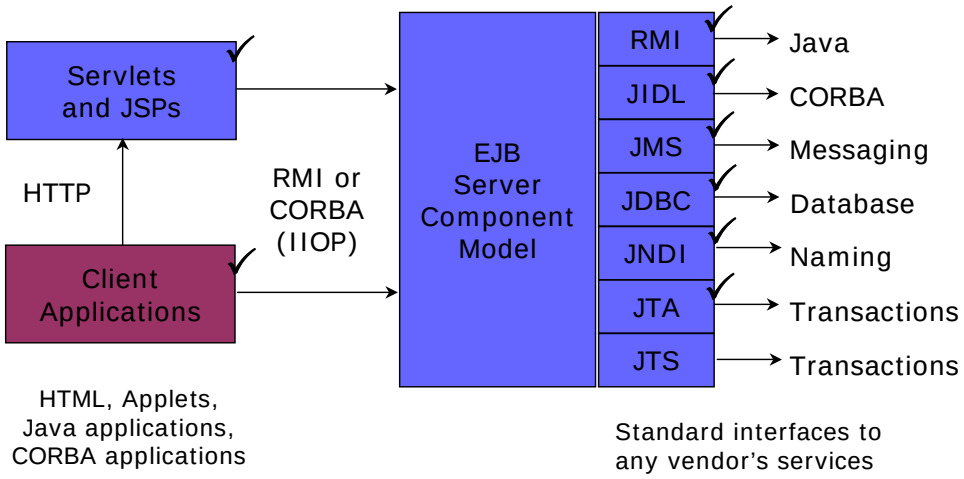
On the client side the component concept has been very successful for user interface development e.g. with the introduction of JavaBeans. The server-side objects running on OTMs are, following the characterization given for components before, clearly also components. Thus we will in the following no longer call them server-side objects, but server-side components. Their main properties are summarized on this slide.

J2EE and Enterprise Java Beans (EJBs)

- Motivation
  - From ad-hoc and vendor specific component models (like WebLogic Enterprise, which is CORBA based) to standard and portable components
  - Integration of Web technologies with Object Transaction Monitor technology to provide scalable Web servers and services
- Java component models
  - Enterprise Java Beans
    - server-side component models: interprocess components
  - Java Beans
    - client-side component models: intraprocess components
  - The two models share nothing but the programming language and name !

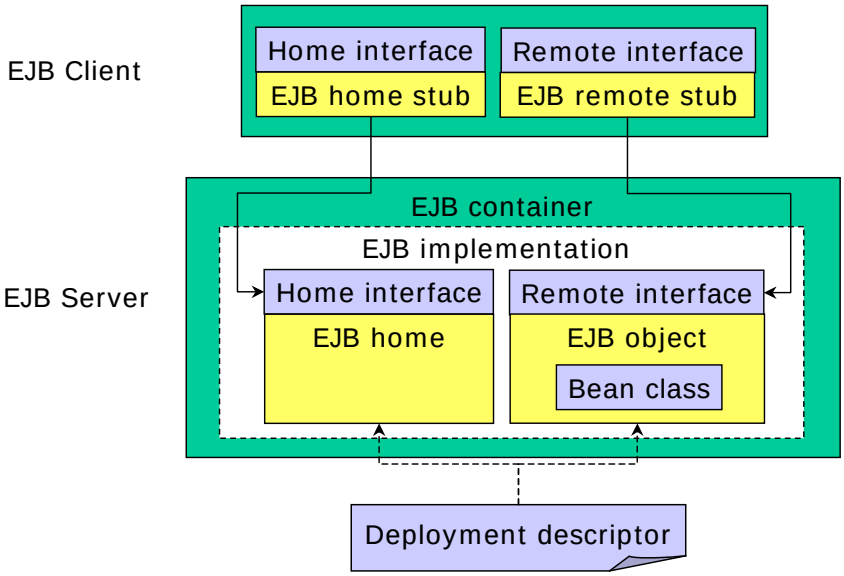
Server-side components appeared first in the context of CORBA-based OTMs. The difficulty is that the component models used by different vendors were not standardized, and thus components would not be reusable across platforms. The standardization of a CORBA based component model, as attempted in CORBA 3.0, lagged behind development. In parallel, Sun defined a Java-based component model, that was heavily borrowing from the CORBA model. This component model, which is called Enterprise Java Beans (EJBs), and is in the meanwhile the de-facto standard. Basing the component model on the Java programming language had the additional advantage that a strong integration with other Web-technologies was made possible.

Enterprise Java Platform (J2EE)



The EJB model is the core of the J2EE platform, the Java 2 Enterprise Edition platform of Sun. This figure shows the different parts of J2EE. As we can see we are already familiar with most of its parts. So in the following we want to complete the picture by introducing the EJB model in detail.

EJB Architecture



The core of the EJB architecture is the component model, which is illustrated here. A component always distinguishes between an EJB home (object) and an EJB (remote) object. The home interface consists of methods, that are used to handle the bean's life-cycle, whereas the remote interface are the "business" methods, that a client can invoke on the EJB object (often called the "bean"). Other equivalent terms for "home" would be "factory" or "class object". The home is implemented by the container, whereas the implementations of the "business" methods the EJB object, are supplied by the developer through a so-called *bean class*. The home object together with the EJB object make up the implementation of the EJB that is running in the EJB container. The properties of the bean can be controlled by a deployment descriptor. The objects (home and EJB objects) are accessed from the client through stubs (exactly as in CORBA). The stubs are generated by the platform. In the figure those parts that are supplied by the developer are darker, whereas the system-generated parts of the EJB are kept lighter.

The Deployment Descriptors contain the specifications for runtime management of beans (transactions etc.) and are packaged as XML file.

In CORBA the EJB object would correspond to the CORBA object and the bean class would correspond to the servant.

Steps in Developing an EJB

- Write a home interface
  - extends `javax.ejb.EJBHome`
  - lists one or more create methods that can be used to create an instance of this enterprise bean
  - will be implemented by the container
- Write a remote interface
  - extends `javax.ejb.EJBObject`
  - describes the business methods of the bean
- Write an enterprise bean class
  - implements `javax.ejb.SessionBean` or `javax.ejb.EntityBean`
  - the method names and signatures must exactly match the method names and signatures of the remote interface
- Write a deployment descriptor
  - specifies functional properties of the bean

For developing a bean the "dark grey" boxes on the previous slide need to be supplied: This includes the interfaces, namely the home interface derived from EJBHome, declaring the methods to be used for creating beans and the remote interfaces derived from EJBObject for declaring the business methods that are supported by the beans.

The implementations for the remote interface are then supplied as implementations of classes SessionBean or EntityBean (the difference will be explained later). For the home interface methods no implementations need to be supplied, but for persistent beans (entity beans) a primary key class needs to be implemented to support the container in managing the objects. Finally the deployment descriptor is supplied.

EJB Example: Home and Remote Interfaces

- Home interface of the bean

```
import javax.ejb.*;
import java.rmi.RemoteException;

public interface authorHome extends EJBHome {
  public author create(authorId aId, String nam)
      throws CreateException, RemoteException;
  public author findByPrimaryKey(authorId aId)
      throws FinderException, RemoteException;
}
```

- Remote interface of a bean

```
import java.rmi.RemoteException;
import javax.ejb.*;

public interface author extends EJBObject {
  public String getName() throws RemoteException;
  public void setName(String nam) throws RemoteException;
}
```


We illustrate the various steps by means of an example for implementing an "author" bean:

First the home interface is defined. One can see that it contains two methods: a create method, which returns a bean object (The fact that the method returns an object of an application specific type is the reason why the developer has to declare, though not to implement, this method), and the findByPrimaryKey method, which allows to find an object based on its persistent identifier. The type of the identifier is authorId, which is the primary key class. It we will described on the next slide and is also supplied by the developer. A method for destroying the object is inherited from the EJBHome interface. It needs not to be declared since it contains no application-specific types.

The remote interface declares the application specific methods of the bean.

EJB Example: Primary Key Class

- The primary key class

```
public class authorId implements java.io.Serializable {
     public int id;

     public int hashCode() {
            return id; }
     public boolean equals(Object obj) {
            if(obj instanceof authorId) {
                if (((authorId)obj).id==id)
                    return true;
            }
            return false;
     }
 }
```


The primary key class implements the functions that are needed in order to support the container in identifying objects that are stored in stable storage. The methods for hashing an identifier and for testing equality of identifiers are required by the container in order to implement methods making use of the primary key, such as findByPrimaryKey.

EJB Example: Bean Class

- Bean class

```
import javax.ejb.*;

public abstract class authorBean implements EntityBean {
   private EntityContext ctx;

   public abstract String getAuthId();
   public abstract void setAuthId(authorId aId);
   public abstract String getName();
   public abstract void setName(String nam);

   public void setEntityContext(EntityContext ctx) {
     this.ctx = ctx;
   }
   public void unsetEntityContext() {
     this.ctx = null;
   }

…
```


The bean class itself implements the application-specific methods as well as the life-cycle management methods, that are required by the container. Even if the life-cycle methods are not performing any actions, they need to be provided with empty implementations. The only life cycle method that performs an activity for the author bean is ejbCreate. It is used to set the identifier of the object to the ID value supplied by the container.

An entity bean with container-managed persistence has persistent and relationship fields. These fields are virtual, they do not need to be coded in the class as instance variables. Instead, they are specified in the bean's deployment descriptor. To permit access to the fields, abstract get and set methods must be defined in the entity bean class.

As can be seen from the remote interface, the only business methods implemented in the example one get and one set .

Bean Class - Continuation

…

```
    public void ejbActivate() {  }
    public void ejbPassivate() {  }
    public void ejbLoad() {  }
    public void ejbStore() {  }
    public void ejbRemove() throws RemoveException  {  }
    public String ejbCreate(authorId aId, String nam)
      throws CreateException
    {
      setAuthId(aId);
      setName(nam);
      return null;
    }

    public void ejbPostCreate(authorId aId, String nam)  {  }

}
```

EJB Example: Deployment Descriptor

```
?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
 Enterprise JavaBeans 2.0//EN'  'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
   <display-name>Author Bean</display-name>
   <enterprise-beans>
     <entity>
       <display-name>authorBean</display-name>
       <ejb-name>authorBean</ejb-name>
       <home>authorHome</home>
       <remote>author</remote>
       <ejb-class>authorBean</ejb-class>
       <persistence-type>Container</persistence-type>
       <prim-key-class>authorId</prim-key-class>
       <reentrant>False</reentrant>
       <cmp-version>2.x</cmp-version>
       <abstract-schema-name>authors</abstract-schema-name>
       <cmp-field>
         <description>no description</description>
         <field-name>authId</field-name>
       </cmp-field>
       <cmp-field>
         <description>no description</description>
         <field-name>name</field-name>
       </cmp-field>
       <primkey-field>authId</primkey-field>
     </entity>
   </enterprise-beans>
...
</ejb-jar>
```


Finally a deployment descriptor is supplied. It provides the necessary information in order to locate all parts of the implementation and declares some functional properties of the bean. In this example it is declared, that the persistent state of the bean should be managed by the container and the bean is non-reentrant, which essentially forbids cyclic invocations of the bean by other beans. Once all the parts of the bean implementation are available, they are packaged, and can be deployed on a container.

Session and Entity Beans

- Entity beans: persistent, used to model data
  - Container-managed (implicit management)
  - Bean-managed (explicit management)
- Session beans: non-persistent, used to model processes
  - Stateful vs. stateless
  - Can implement the interaction among entity beans
- Stateless session beans
  - No state is maintained in between calls
- Stateful session beans
  - Sessions: maintain a state from one invocation to the next for a particular client (state not shared and not persistent)
  - Allows to implement conversations with the client
  - Are dedicated to a single client and are not pooled
  - Have a preset timeout period

We have already mentioned the fact that there exist different kinds of beans. The differences are related to whether beans have a persistent state and whether they support sessions with clients.

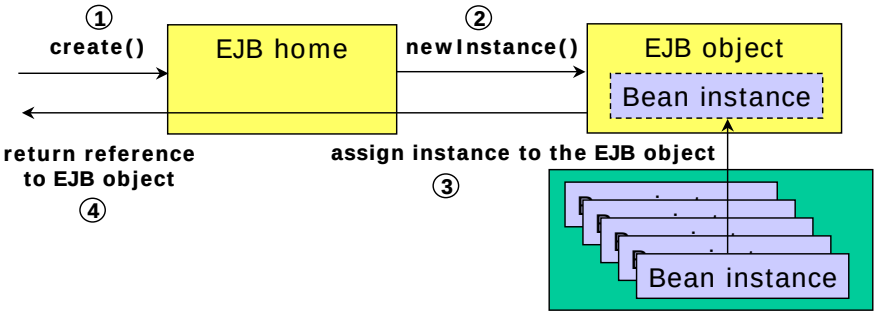
Entity beans are the beans that have a persistent state, that are thus used to model data and that survive the lifetime of any process using the bean. For entity beans one has two options of implementation: explicit or implicit. With implicit management of persistent state, the container is taking over the task to make the bean's state persistent. That means it implements a mapping to the storage system. With bean-managed persistence the developer is responsible to manage the persistent state. This is the approach that typically would be taken if the persistent state of the beans is stored in existing databases, that would for example be accessed via JDBC. The developer has in that case to supply implementations of methods, that are used by the container to read and write object state to and from the persistent storage.

Session beans maintain no persistent state. That means, they exist only throughout the lifetime of a server process. However a distinction is made between beans that maintain a session with a client or not. Stateful session beans allow multiple invocations on the same bean for a particular client (similarly as we have already seen earlier for servlets). Note that, maintaining the state for a session is not related to the problem of having a persistent state, the state is only kept within the transient memory of the server process. As for servlets, the advantage is that complex conversations with clients can be supported, as they are for example required in ecommerce interactions. The sessions are always exclusive to one client are terminated after a timeout is exceeded.

The distinction between entity and session beans is comparable to the distinction between transactional and recoverable objects in the CORBA transaction service model.

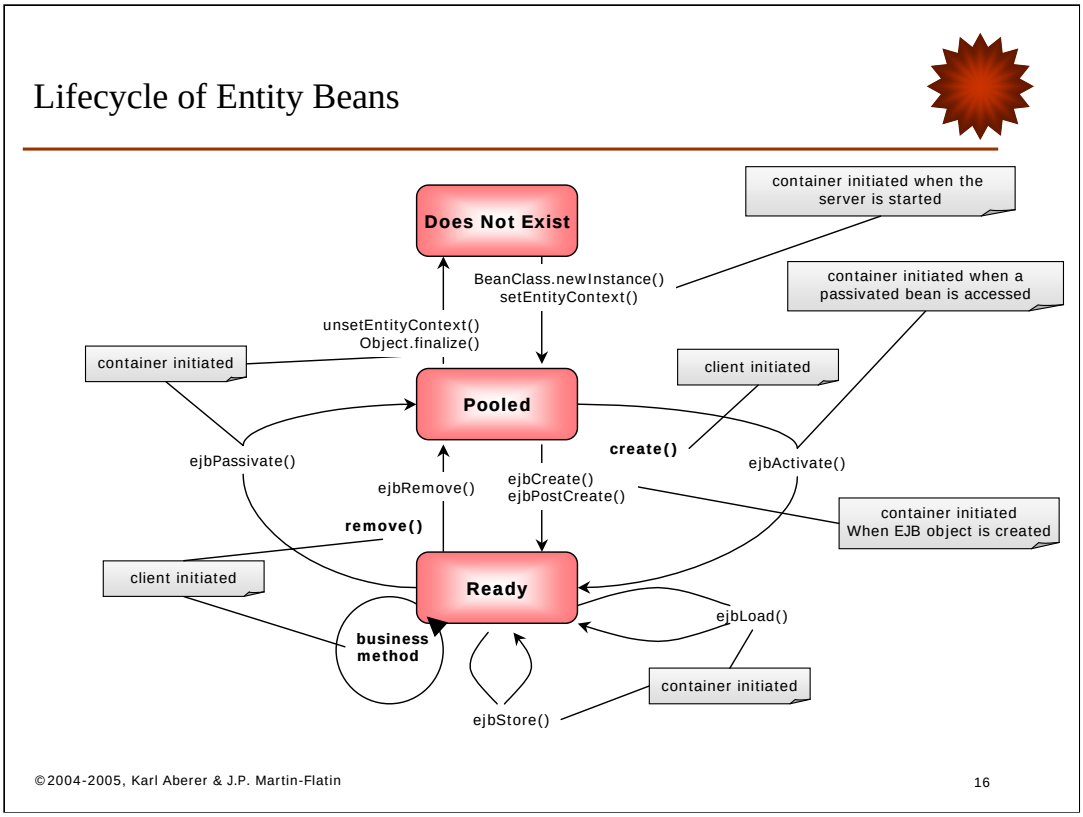
Resource Management

- EJB objects can be dynamically associated with Bean Instances
  - Several clients access the same bean instance through different EJB objects
  - Like CORBA objects with servants
- Advantages
  - Fewer resources needed (memory)
  - Reusing instances more efficiently (fewer object creations and deletions)



Similarly as with a POA also the association of EJB objects (the equivalent of a CORBA object) and Bean Instances (the equivalent of a servant) can be managed in a flexible manner, such that different EJB objects can share different bean instances from a pool of bean instances. In fact, it is the normal modus operandi of an EJB container. This reduces the amount of resources consumed (memory) and improves performance as fewer objects need to be created and deleted. The basic life cycle of a bean, as perceived from the client side, consists of invoking a create() method on the EJB home which creates a new instance of the EJB object class, which is then dynamically associated with a bean instance from a bean instance pool. Then the reference to the EJB object is returned to the application.

Depending on the bean type (entity, stateless session, stateful session) the life cycles of the beans as seen from a system perspective (or better: the container perspective) differ substantially.

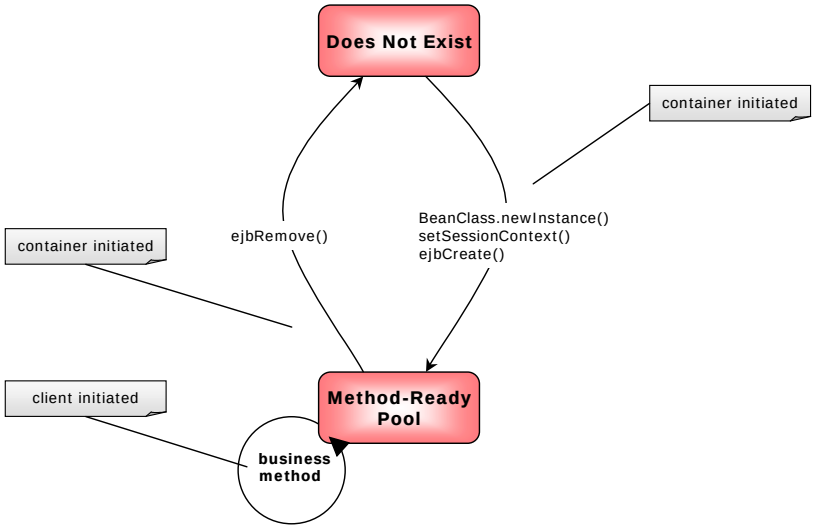


Here we see the life cycle of an entity bean. We discuss it step by step passing through the different phases of the bean life:

Initially the bean does not exist. When the server is started and the container is initiated, the first thing it does, is to instantiate bean instances (objects of the bean implementation class, e.g. authorBean) and put them into the pool. How many of them are initiated is system-specific. Before the bean object is put into the pool (and thus into the pooled state) the setEntityContext() method is called. It gives the bean instance a handle on the EntityContext object, which in turn gives the bean instance the possibility to access throughout its lifetime relevant information from the container on the EJBObjects it is associated with (and that are changing).

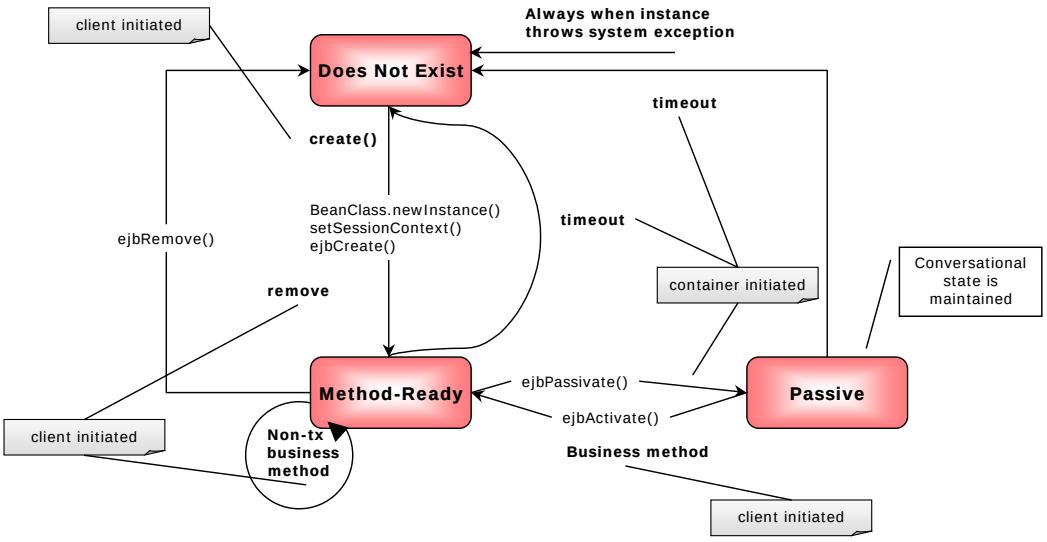
The next step is the creation of the EJBobject which is initiated by the user. As part of the creation the EJB object the container invokes the ejbCreate() and ejbPost Create() methods from the bean interface. ejbCreate() creates typically a primary key and ejbPostCreate() can already make use of this primary key. Before methods can be invoked on beans with bean-managed persistence, the ejbLoad() method is invoked in order to retrieve the bean's persistent state from the stable storage. Then the bean is in the ready state and the business methods are called by the client. There exist now two ways of how the bean can return from the ready state back to the pooled state, i.e. of how the bean instance can be dissociated from the EJBobject. Either initiated by the container after a timeout (that is system-specific), i.e. when the bean has not received methods for a while, or by the user when he removes the bean with remove(), which implies that actually the object is deleted and also removed from the presistent storage. With container-initiated passivation, the ejbStore() method is used to save the beans state (if the state is bean managed) and then the ejbPassivate method is invoked. Passivated beans still are connected to the clients by means of the EJBobject, and as soon as methods are invoked they are again activated, which implies that the container invokes the ejbActivate() and ejbLoad() methods. When the bean is destructed the ejbRemove() method is invoked. The life cycle of the bean instance ends when the container decides to do so, either because it wants to free the resources for other purposes or the server is shut down.

Lifecycle of Stateless Session Beans



The life cycle of stateless session beans is substantially simpler than the one for entity beans. All the method invocations from the side of the container are self-explaining. The beans in the method-ready pool answer incoming request and then immediately return into the pool. The container controls the number of bean instances that are kept in the pool, depending on the request load. Note that since the EJBobjects are not persistent, no identifiers need to be supplied by the application, and the ejbCreate() method is executed when the bean instance is created, not when the client invokes the create() method for the EJBObject from the home interface, when it wants to acces the bean.

Lifecycle of Stateful Session Beans



The life cycle of a stateful session bean is again quite different from the two previous ones. The difference lies in the fact that the stateful session beans are not using session pooling. In order to save resources the only thing that can occur, is that the bean is passivated. This implies that the bean instance is removed from the memory and the conversational state of the sesssion is preserved.

Therefore the life cycle of the bean starts differently: upon the invocation of the create() method on the home interface by the client, a EJB object is created AND a bean instance is put in the method-ready pool and associated with the EJBObject after executing the setSessionContext() and ejbCreate() methods. The assignment of the SessionContext reference allows the bean instance to access information on the session of the EJBObject, to which it is associated with.

Then the business methods are executed. After a timeout the bean is passivated by the container and its conversational state is conserved. Upon the invocation of other business methods the bean is again activated. If it is not accessed a too long period the bean is removed (this also happens when the bean instance throws an exception). It can also be the case that beans are directly removed from the method-ready state. Also the client can remove the bean by calling the remove() method on the EJBobject.

Transactions

- Managed explicitly or implicitly
  - Explicitly: Java transaction API (JTA)
  - Implicitly (normal): EJB transaction attributes
    - Set for the whole bean or particular methods
    - Specified in the deployment descriptor
- EJB transaction attributes for methods
  - `NotSupported`: the method does not take part in the transaction of the caller
  - `Supports`: the method will be executed within the callers transaction if it exists
  - `Required`: the method must be executed within a transaction, either it is called as part of a transaction and takes part in that transaction or it initiates a new transaction
  - `RequiresNew`: always a new transaction is started when the method is called
  - `Mandatory`: the caller must call the method from within a transaction
  - `Never`: the method never must be called from within a transaction

Transactions for bean methods can be handled in two ways: explicitly and implicitly. Explicit transaction management is what we would call in the CORBA transaction service indirect management with implicit propagation, and corresponds to the procedural style of transaction invocation as known already from X/Open, and as provided in the Java world by means of JTA. Implicit management of transactions is however the standard way of how beans should be used: it takes fully advantage of the possibility of declaring transactional properties of methods in the deployment descriptors. The transactional properties can there be declared either for the whole bean or for specific methods of the bean. The different possibilities of how methods participate (or not participate) in transactions are listed.

Example Specification

```
<ejb-jar><enterprise-beans>
<entity>
     <ejb-name>authorEJB</ejb-name>...
</entity>
<assembly-descriptor>
     <container-transaction>
             <method>
                     <ejb-name>authorBean</ejb-name>
                     <method-name>setName</method-name>

             </method>
             <trans-attribute>Required</trans-attribute>
     </container-transaction>
</assembly-descriptor>
...
</enterprise-beans></ejb-jar>
```


This example shows of how in an EJB deployment descriptor the transactional properties are set for our example bean authorBean. For the method setName of the bean authorBean (which is an entity bean), the transaction attribute is set to "required". This means that either it must be invoked within a transaction or it starts itself a new transaction. This makes of course sense for a method that updates the state of the entity bean.

Persistence

- Two possibilities
  - Container-managed persistence
  - Bean-managed persistence
- Container-managed persistence
  - Depends on EJB platform
  - Depends on underlying storage systems
  - Usually object-to-relational persistence: mapping to relational DB
  - Based on the mapping definition provided by the developer (e.g., in a deployment descriptor), the persistence is automatically managed by the container
  - Mapping can be complex and costly
- Bean-managed persistence
  - Typically required when persistent data is stored in legacy database
  - Application developer has to provide the code to implement persistent storage of bean objects (e.g. using JDBC)

We have already mentioned the two possibilities of managing the persistent state of entity beans: either by the container or by the bean (resp. the application).

The implementation of container-managed persistence depends clearly on the EJB platform and the available storage systems. The standard solution is that the container uses an object-to-relation mapping and stores the beans state in a relational database. The mapping itself must be provided by the developer, for example, in a form of an extended deployment descriptor. As we also know from the problem of mapping XML to a relational representation, this can be a non-trivial task. Care has to be taken that the mapping does not introduces substantial inefficiencies.

The second alternative is to store the beans state in a database and access the database through a programming language API, in the Java world naturally through JDBC. Then the application developer has to provide the methods in order to store and retrieve the state from the database. These methods are made available to the container through the ejbLoad() and ejbStore() methods.

Example: Container-Managed Persistence

- Definition of a relational table
```
CREATE TABLE AUTHOR(ID INT PRIMARY KEY, NAME CHAR(3))
```
- Primary key class property must be public
```
public class authorId implements java.io.Serializable {
     public int id;
...
```
- Deployment descriptor lists the (potentially) persistent fields
```
<ejb-jar><enterprise-beans><entity>
     ...
     <prim-key-class>epfl.lib.author.authorId</prim-key-class>
     <persistence-type>Container</persistence-type>
     ...
     <cmp-field><field-name>authId</field-name><cmp-field>
     <cmp-field><field-name>name</field-name><cmp-field>
</entity></enterprise-beans></ejb-jar>
```
- The home interface supports the `findByPrimaryKey()` method
  - implemented by the container
  - Other "finder" methods can be provided

In the following we give a more detailed description of how a mechanism for container-managed persistence works. Let us assume that we have defined a table author that should be used in order to store the state of the author bean, that we have used in the earlier examples. One has to take care, that the primary key class is declared public (which we did already, wisely enough). The deployment descriptor now contains information on

-the primary key of the entity bean (tag <prim-key-class>)

-The type of persistence management, namely container-managed persistence (tag <persistence-type>)

-And a list of the fields that should be made persistent (tag <cmp-field>, cmp=container-managed-persistent)

This part of the descriptor is according to the EJB standard.

Given this information the container will, for example, by default provide an implementation for the findByPrimaryKey() method, in order to locate and access entity beans from the stable storage. In order to be able to do this the container needs however further information on the mapping to the database. The specification of this mapping is vendor-specific. We look at it next.

Mapping EJB Object State to Relational Databases

- Vendor-specific solutions for container-managed persistence
  - Deployment descriptors
  - Graphical interfaces
  - Command line interfaces
- Example: RDBMS persistence using WebLogic
  - Deployment descriptor defines mappings from database fields to object attributes
  - Includes specification of methods to find objects in the database
  - Available as ejbHome finder methods
  - These are declared in a specific deployment descriptor

```
public Enumeration findAuthor(string name)
        throws FinderException, RemoteException;
```


There exist different solutions for specifying the mapping, including graphical interfaces or command line interface. We look at the solution that is taken by the Weblogic Server which is particularly elegant, since it builds on the already existing concept of deployment descriptors.

We assume, that in addition to the relational tables, also application specific methods for locating objects have been included in the home interface as extensions of the standard ejbHome finder methods, such as a method findAuthor, that locates an author based on his/her name and the ID. The implementation of such a method will be given in the mapping specification.

Example: WebLogic RDBMS Deployment Descriptor

```
<!DOCTYPE weblogic-rdbms-bean $
PUBLIC  '-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB RDBMS
Persistence//EN' 'http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-
persistence.dtd'>
<weblogic-rdbms-bean>
  <pool-name>demoPool</pool-name>
  <table-name>authors</table-name>
  <attribute-map>
    <object-link>
      <bean-field>authId</bean-field>
      <dbms-column>id</dbms-column>
    </object-link>
    <object-link>
      <bean-field>name</bean-field>
      <dbms-column>name</dbms-column>
    </object-link>
 </attribute-map>
  <finder-list>
     <finder>
      <method-name>findAuthor</method-name>
      <method-params>
        <method-param>string</method-param>
      </method-params>
      <finder-query><![CDATA[(= name $0)]]></finder-query>
    </finder></finder-list>
</weblogic-rdbms-bean>
```


Here we see now of how the mapping from the relational table to the EJBobject is given.

-First the table name is given (tag <table-name>)

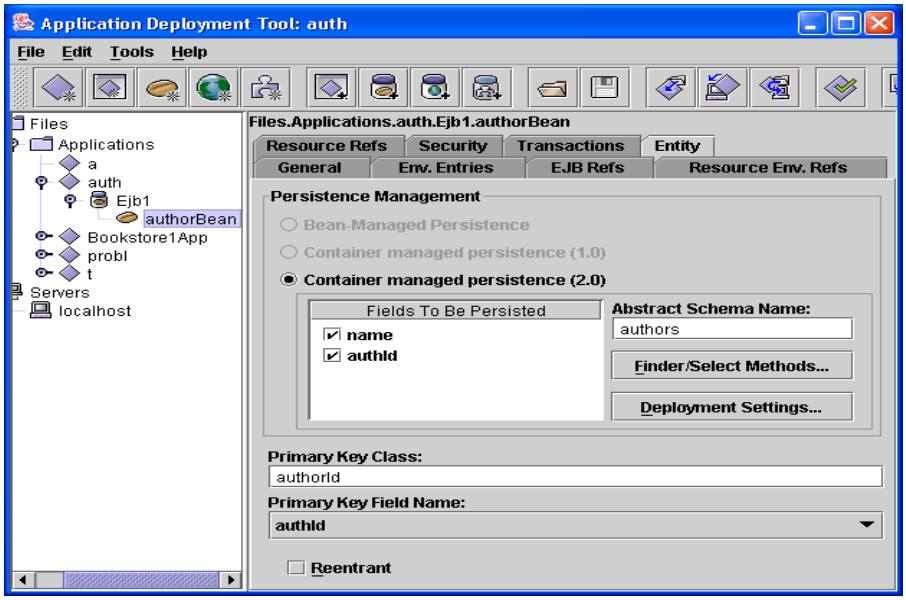
-Then an attribute map, consisting of the mappings of different attributes, is given (tag <attribute-map>)

-Within the attribute map a mapping between object attributes and table attributes is established (tags <object-link>). For example, the bean attribute authId is mapped to the table attribute id.

-Then a finder list is given, i.e. methods used to locate objects in the database (tag <finder-list>). For each finder method the method name, the method parameters and a query (actually an awkwardly encoded relational query) implementing it are given

One can in particular notice that the expressiveness of the mapping is rather limited. It allows only to map single attributes to single columns. Thus, this approach can only be used in order to map to databases that have been specifically designed for storing the bean objects' state. More recent versions of Weblogic provide for that purpose tools to generated the corresponding tables automatically.

Example: Storage Mapping Through GUI



Summary

- Components
  - Ready-to-run software packages that support an object-oriented client interface and support a component interface (contract) with the container
  - EJBs are the Java-based component model
- Web Application Servers
  - Are forming the backbone of the Internet computing infrastructure

The presence of declarative container services does not absolve the bean developer (and deployer) from understanding how transactions, communications, and exception handling work in the context of EJB technology.
(Professional Java Server Programming, J2EE 1.3 Edition, WROX Press, 2001)

References

- Books
  - R. Monson-Haefel, *Enterprise Java Beans*, 2nd Edition, O'Reilly, 2000.
  - P. Gomez and P. Zadroznyy, *Java 2 Enterprise Edition with BEA WebLogic Server*, Wrox Press, 2000.
- Websites
  - J2EE Tutorial: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/
  - WebLogic EJB OTM: http://www.weblogic.com/docs51/resources.html

Integration of Distributed Objects Checklist

- Task: integrate distributed, heterogeneous (and probably autonomous) transactional resources (like databases)

- Abstract Model ✓
  - Object model, IDL, EJB
- Embedding ✓
  - Object adaptors, Containers
- Architectures and Systems ✓
  - Object Management Architecture (OMA)
  - J2EE
- Methodology