Business-Driven Software-Engineering (5. Vorlesung)
EJB: Session Beans and EJB 3.0 Entities
Thomas Gschwind <thg at zurich.ibm.com>
Agenda

- EJB3.0 Cookie Session Bean
  - Server
  - Client
  - Stateless vs. Stateful Session Bean

- EJB3.0 Entities
  - Overview
  - OR Mapping
  - Relationship Management

- Entity Bean Demo
Agenda

- EJB3.0 Cookie Session Bean
  - Server
  - Client
  - Stateless vs. Stateful Session Bean

- EJB3.0 Entities
  - Overview
  - OR Mapping
  - Relationship Management

- Entity Bean Demo
A Simple Quote Service

- Our first example returned quotes to its clients
  - Probably the most complicated and inefficient way to write a quote service
  - BUT, it nicely shows how to write a simple bean

- Which kind of bean?
  - A session bean!
The Implementation class

```java
@Stateless(name="QuoteService")
@Remote(CookieServiceRemote.class)
public class QuoteService implements QuoteServiceRemote {
    @Resource
    SessionContext context;

    @PrePassivate
    public void ejbPassivate() { ... }

    @PostActivate
    public void ejbActivate() { ... }

    public String getQuote() {
        ... return quote;
    }
}
```
Session Bean Annotations

- **@Stateless** or **@Stateful** indicate that this is a stateless or stateful session bean
  - The name attribute may specify the JNDI name

- **@Remote** specifies the remote interface as parameter
  - Not needed if the remote interface is implemented by this class and already annotated

- **@Resource** to initialize the SessionContext attribute
  - Useful to retrieve references to other beans and to interact with the transaction and security API
The SessionContext

```java
public interface javax.ejb.SessionContext extends javax.ejb.EJBContext {
    public javax.ejb.EJBLocalObject getEJBLocalObject();
    public javax.ejb.EJBObject getEJBObject();
}
```

- **EJBContext provides methods to obtain**
  - The home reference
  - Transaction attributes
  - Security information
Session Bean Lifecycle Annotations

- **Creation methods**
  - `@Init` for EJB2.1 backwards compatibility (avoid)
  - `@PostConstruct` is invoked after the bean has been initialized

- **Removal methods**
  - `@PreDestroy` is invoked before the bean is destroyed
  - `@Remove` indicates the end of a session (stateful session beans)

- **Passivation of a session (stateful session beans)**
  - `@PrePassivate`
    - Called before bean is passivated (swapped out)
    - Fields serialized using Java serialization
    - Release resources that cannot be handled by Java serialization
  - `@PostActivate`
    - Called after bean activation
    - Fields deserialized using Java serialization
    - Acquire resources needed (those released in `@PrePassivate`)
Session Bean Lifecycle Annotations (cont’d)

- Container responsible for creation & destruction of beans
  - Container maintains the pool of bean instances
  - The invocation of the lifecycle methods does not necessarily correspond to their time of creation or removal
  - Do not rely on the removal methods to be invoked

- Alternatively, to providing the lifecycle methods as part of the bean implementation, you may use a lifecycle listener
  - Specified with the @Interceptors(Listener.class) annotation
  - Listener uses the same annotations as the bean class
  - Listener could be reused for multiple different session beans
Deployment Descriptor

- Provides information about the bean
- May be used alternatively to or in addition to annotations
- If both is provided, deployment descriptor overrides annotations in the code

- In Eclipse, a deployment descriptor stub can be generated
  - Right clicking the deployment descriptor and select “Generate Deployment Descriptor Stub”
  - Deployment Descriptor can be found in ejbModule/META-INF/ejb-jar.xml
Deployment Descriptor (cont’d)

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

   <display-name>ch.uzh.ejb.ex01</display-name>
   <enterprise-beans>
   <session>
      <ejb-name>QuoteService</ejb-name>
      <ejb-class>ch.uzh.ejb.ex01.QuoteService</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
   </session>
   </enterprise-beans>
</ejb-jar>
```
Packaging The Beans

- Just put everything into a .jar file

```
jar cf bean.jar ...
```

Enterprise Bean

Vendor-specific files

Deployment Descriptor

Helper Classes

Local Remote Interfaces

Local Home Interfaces

Remote Interfaces

Home Interfaces

Bean Classes
Enterprise JavaBeans Client

- Get a reference to the JNDI interface
  - Possibly specify system properties where to locate the repository
- Look up the bean
- Invoke the bean’s business methods
Enterprise JavaBeans Client (cont’d)

```java
public class QuoteClient {
    public static void main(String[] args) throws Exception {
        Properties props=new Properties();
        props.put("java.naming.factory.initial",
                "org.jnp.interfaces.NamingContextFactory");
        props.put("java.naming.provider.url",
                "localhost:1099");
        Context ctx=new InitialContext(props);
        QuoteServiceRemote quoteService = (QuoteServiceRemote)
                ctx.lookup("QuoteService/remote");
        System.out.println(quoteService.getQuote());
        System.out.println(quoteService.getQuote());
        quoteService.ejbRemove();
    }
}
```
EJB 3.0 Bean Interaction

1. Retrieve EJB reference (optional)
2. Invoke business method
3. Invoke container
4. Invoke container
5. Delegate request to bean
6. Invoke container
7. Return result
A Stateful Quote Service

- We want to convert the Quote Service into a stateful service
- What needs to be changed on the server side?
- How can we test our service on the client side?
Agenda

- EJB3.0 Cookie Session Bean
  - Server
  - Client
  - Stateless vs. Stateful Session Bean

- EJB3.0 Entities
  - Overview
  - OR Mapping
  - Relationship Management

- Entity Bean Demo
Persistence Approaches

- **Java Object Serialization**
  - Allows to convert an object into a byte-stream
  - Usable for only small number of objects

- **Object Databases**
  - Holds entire objects
  - Few vendors, not yet as sophisticated
Java Persistence

- Defines Entities
- Not tied to Java EE, can be tested in JSE
- Defines a service provider interface
  - Supports different persistent providers without affecting entity code
  - Persistence provider can use any kind of underlying storage mechanism, provided they implement the service provider interface
- Provides a standard OR mapping
  - Integrates many concepts of Hibernate or JDO
Object Relational Mapping

- More sophisticated than serialization
- Object-Relational Mapping
  - Store objects in a database
  - Retrieve data from db when object is created
Entities

- Model an application’s permanent data
- Entities represent things (nouns)
- Provide a view into a database (beans know how to store themself)
  - Provide a primary key to identify data
  - Potentially long lifetime
- Bean update ⇔ database update
- Can be shared across clients
- Coarse grained data
- Examples: account or portfolio item
Entities vs. JDBC

- Why not use JDBC directly?
- Because
  - Entity beans know how to store themselves
  - Access to data via a Java interface
  - Automatically updates value of underlying data
  - Survive critical failures
  - Maintainability
JPA Entities

```java
@Entity
@Table(name="QUOTES")
public class Quote {
    private long quoteId;
    private String quote;
    private long counter;

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
        generator="QUOTES_SEQ")
    @SequenceGenerator(name="QUOTES_SEQ", sequenceName="QUOTES_SEQ")
    public long getQuoteId() { return quoteId; }
    public void setQuoteId(long id) { this.quoteId=id; }

    // ...

    // other business methods
    public void incCounter() { this.counter++;
    // ...
}
```
JPA Entities (cont’d)

- Implemented as “Plain Old Java Object”
  - No need to implement EntityBean interface, not a bean
  - No need to implement Serializable (necessary for pre-3.0 entity beans or if entity to be sent via remote interface)

- Need to provide some annotations for the persistence API to generate the OR mapping
  - Annotate with @Entity (may specify the name of the database table, i.e., QUOTES)
  - Indicate primary key with @Id annotation (i.e., quoteld)
    - May use sequences to automatically assign a key
      - @GeneratedValue specifies the sequence
    - @SequenceGenerator specifies how to map the sequence to the database
  - Columns for every field/getter/setter
  - Can expose additional business methods
Using Entities

@Stateless
public class QuoteService
implements QuoteServiceRemote, QuoteServiceLocal {
    @PersistenceContext(unitName="QuoteService")
    private EntityManager em;

    public void addQuote(String text) {
        Quote q = new Quote();
        // q.setQuoteId(System.currentTimeMillis());
        q.setText(text);
        em.persist(q);
    }

    public String getQuote() {
        Query q=em.createQuery("SELECT q FROM Quote as q");
        List<Quote> quotes = q.getResultList();
        String quote = quotes.get(i++%quotes.size()).getText();
        return quote;
    }
}
By default, the entity manager takes care of entities until the transaction aborts/commits
Using Entities

```java
@Stateless
public class QuoteService implements QuoteServiceRemote, QuoteServiceLocal {
    @PersistenceContext(unitName="QuoteService")
    private EntityManager em;

    public void addQuote(String text) {
        Quote q = new Quote();
        // q.setQuoteId(System.currentTimeMillis());
        q.setText(text);
        em.persist(q);
    }

    public String getQuote() {
        Query q=em.createQuery("SELECT q FROM Quote as q");
        List<Quote> quotes = q.getResultList();
        String quote = quotes.get(i++%quotes.size()).getText();
        return quote;
    }
}
```
A persistence unit is a logical grouping of entity data, mapping metadata, and database-related configuration data.

- We must provide at least one persistence unit.
- If only one is provided, by default all entities are part of this unit.
Persistence Unit Properties

- hibernate.dialect specifies the SQL dialect to use
- hibernate.hbm2ddl.auto specifies how to map entities to database tables
  - validate: validate the schema, make no changes to the database
  - update: update the schema
  - create: create the schema, destroying old data
  - create-drop: create the schema and drop at the end of the session
- Be careful with hibernate.hbm2ddl.auto in production environments
  - create and create-drop can destroy data
  - update does not work in all situations
  - These options are useful, however, on development machines
Programming Style

- Problems in the API as we have used it
  - SQL statements are contained as text in the program
  - SQL statements cannot be checked
  - SQL statements cannot be precompiled
    (important if they are complex and are used frequently)
  - More importantly, SQL statements are spread all over the session beans making use of entities

- Solution: Named Queries
  - Checked and compiled during deployment time
Named Queries

```java
@Entity
@Table(name="QUOTES")
@NamedQuery(name="findByText",
            query="SELECT DISTINCT OBJECT(c)
                   FROM Quote q WHERE q.text like :text")

public class Quote {
    // ...
}

@Stateless
public class QuoteService implements QuoteServiceRemote, ...
{
    @PersistenceContext(unitName="QuoteService")
    private EntityManager em;

    public List<Quote> findQuote(String text) {
        Query q=em.createNamedQuery("findByText");
        q.setParameter("text",text);
        return q.getResultList();
    }
}
```
Relations

- Types of relations handled
  - 1:n / n:1
  - 1:1
  - n:m

- Directionality
  - Specify the relationship on each side where you want to traverse it

- Aggregation vs. Composition
CM 1:n Relationship

<table>
<thead>
<tr>
<th>OrderId</th>
<th>OrderName</th>
<th>ShipmentId</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>JBoss</td>
<td>517</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LineItemId</th>
<th>Description</th>
<th>OrderId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1213</td>
<td>JBoss Server</td>
<td>600</td>
</tr>
<tr>
<td>1214</td>
<td>JBoss Documentation</td>
<td>600</td>
</tr>
</tbody>
</table>

- Provide foreign key in :n table
- 1:n side: return Collection/Set of objects
- n:1 side: return related object
LineItems to Order Relationship

```
@Entity
public class LineItem {
    private long lineItemId;
    private String name;
    private Order order;

    @Id
    public long getLineItemId() { ... }
    public void setLineItemId(long id) { ... }
    // ...

    @ManyToMany
    @JoinColumn(name="orderId", nullable=false)
    public Order getOrder() {
        return order;
    }
}
```
Order to LineItems Relationship

@Entity
public class Order {
    private long orderId;
    private String orderName;
    private Shipment shipment;
    private Set<LineItem> lineItems;

    @Id
    public long getOrderId() { return orderId; }
    public void setOrderId(long id) { this.orderId=id; }
    // ...

    @OneToMany(cascade=ALL, mappedBy="orderId")
    public Set<LineItem> getLineItems() {
        return lineItems;
    }
}

Relationship Attributes

- **@ManyToMany**
  - cascade: operations that should be cascaded over this relationship
  - fetch: lazy or eager (active) loading of relationship
  - optional: whether the relationship is optional

- **@OneToMany**
  - cascade: operations that should be cascaded over this relationship
  - fetch: lazy or eager (active) loading of relationship
  - mappedBy: owning field for the non-owning relationship side

- **@JoinColumn**
  - name: name of the foreign key column
  - insertable, nullable, unique: database constraints for the join column
Lazy vs. Eager Retrieval

- **Eager**
  - Useful if the other side of the relationship is likely to be retrieved
  - Useful if small amount of data is being retrieved

- **Lazy**
  - Necessary, if otherwise a large set of columns is being retrieved

- Eager retrieval poses the danger of loading the entire database into memory
- Lazy retrieval the problem of being slow
- If in doubt use lazy retrieval of data
CM 1:1 Relationship

<table>
<thead>
<tr>
<th>OrderPK</th>
<th>OrderName</th>
<th>ShipmentFKPK</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>JBoss</td>
<td>517</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ShipmentPK</th>
<th>City</th>
<th>ZIPCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>517</td>
<td>Vienna</td>
<td>1040</td>
</tr>
</tbody>
</table>

- Provide foreign key in source table
Order to Shipment Relationship

```java
@Entity
public class Order {
    private long orderId;
    private String orderName;
    private Shipment shipment;

    @Id
    public long getOrderId() { return orderId; }
    public void setOrderId(long id) { this.orderId=id; }
    // ...

    @OneToOne
    @JoinColumn(name="shipmentId", unique=true)
    public User getShipment() { return shipment; }
    public void setShipment(Shipment s) { this.shipment=s; }
}
```
Relationship Attributes

- **@OneToOne**
  - cascade: operations that should be cascaded over this relationship
  - fetch: lazy or eager (active) loading of relationship
  - mappedBy: owning field for the non-owning relationship side (i.e., for the Shipment class side)
  - optional: whether the relationship is optional

- **@JoinColumn**
  - name: name of the foreign key column
  - insertable, nullable, unique: database constraints for the join column
m:n Relations

- May be modeled indirectly using m:1 and 1:n relations
  - May store additional information

- May be modeled directly with JPA
  - Association table managed by JPA
  - Similar to 1:n relation => see book
Advanced Topics

- **Directionality**
  - Just get rid of the getter/setter pair
  - BM => only implement one half

- **EJB-QL**
  - Traverse relationships
  - E.g.,
    ```sql
    SELECT o.customer FROM order o
    ```
Agenda

- EJB3.0 Cookie Session Bean
  - Server
  - Client
  - Stateless vs. Stateful Session Bean

- EJB3.0 Entities
  - Overview
  - OR Mapping
  - Relationship Management

- Entity Bean Demo
The JBoss Server

- **Directories**
  - jboss/bin … executables
  - jboss/client … client jars
  - jboss/server/default … default server files
    - conf, deploy, lib, log

- **Management interface**
  - http://localhost:8080/jmx-console/

- **Add JBoss to your Eclipse environment**
  - In Eclipse: New -> Other -> Server -> Server
  - Select JBoss 5.0
  - Specify the path where you unpacked JBoss to
JBoss

- Eclipse cannot create a JPA Entity within an EJB project
  - Create the class manually
- Got “Persistence provider caller does not implement the EJB3 spec correctly.”
  - Ignore this error
- Need to create or look into database tables
  - Launch the hsql frontend (also useful for debugging purposes)
  - Go to http://localhost:8080/jmx-console/
  - In the jboss section, go to database=localDB,service=Hypersonic
  - Invoke startDatabaseManager
Summary

- EJB3.0 Cookie Session Bean
  - Server
  - Client
  - Stateless vs. Stateful Session Bean

- EJB3.0 Entities
  - Overview
  - OR Mapping
  - Relationship Management

- Entity Bean Demo
Review Questions

- What is the purpose of the session context?
- What are the lifecycle methods of a session bean and what is their purpose?
- What is a JPA entity?
- What is the lifecycle of a JPA entity?
- Why do JPA Entities provide named queries?
- What is the difference between lazy and eager loading/retrieval?
Tasks

1. Download the latest example files and make it run on your machine

2. Play with the different types of session bean and update the example such that it shows the differences
Outlook

- Transaction Management
- Security Management