EJB 3.0: The Next Generation

Mike Keith
Oracle Corp.
michael.keith@oracle.com
About Me

- Co-spec Lead of EJB 3.0 (JSR 220)
- Java EE 5 (JSR 244) expert group member
- Architect for OracleAS TopLink and OracleAS EJB Container in OracleAS OC4J
- 15+ years experience in OO persistence and numerous persistence implementations
- Presenter at JavaOne, JavaPolis, TSS Java Symposium, JAOO, CSS, JavaPro Live, etc.
Audience Poll

- How many people are using EJB in current application development?
- How many have used them since EJB 1.1?
- How many people are using any one of:
  - Oracle TopLink
  - Spring or Hibernate
  - SolarMetric Kodo
Goal

Learn more about where EJB 3.0 came from, what it has to offer and why it is important to you
Agenda

1. Introduction - The Early Years
2. EJB 3.0 Goals
3. A Simplified Component Model
4. Java Persistence API
5. Summary
Agenda

1. Introduction - The Early Years
2. EJB 3.0 Goals
3. A Simplified Component Model
4. Java Persistence API
5. Summary
Life Before EJB

Disorganization!

- Emergence of middle-tier Java architectures
- Lack of standard mechanisms for encapsulating and specifying business logic
- No organized cohesive technology base for enterprise Java (pre-J2EE)
- Developers had to keep re-inventing the server-side wheel of transactions, concurrency, and security
- Community acquired the vision of pluggable server-side Java components
Goals of EJB

The initial goals of EJB were:

- Allow components developed separately to be deployed together and interoperate in the server
- Define development and deployment contracts so that the development tools can produce interoperable components
- Lessen the knowledge required to develop components
- Provide access to low-level APIs for advanced developers
- "Write once, run on any EJB Container!"
- Interoperability with non-Java applications; compatibility with CORBA
EJB Timeline

- EJB 1.0 — JDK 1.1
  - Feb 1997
  - Apr 1997
  - Mar 1998
  - Dec 1998
  - Jun 1999
  - Dec 1999
  - May 2000
  - Sep 2001
  - Feb 2002
  - Apr 2004
  - Sep 2004

- EJB 2.0 (J2EE 1.3)
  - Sep 2004

- EJB 2.1 (J2EE 1.4)
  - J2SE 5.0

- EJB 1.1 (J2EE 1.2)
  - J2SE, J2EE announced
  - Jun 1999

- Java 2 (JCP started)
  - Mar 1998
Agenda

1. Introduction - The Early Years
2. EJB 3.0 Goals
3. A Simplified Component Model
4. Java Persistence API
5. Having it All
6. Summary
EJB 3.0 Goals

- Simplify developer programming model
  - EJB development should just be Java programming

- Simplify the client programming model
  - Reduce complexity of obtaining and using EJB’s

- Reduce the learning curve for new developers
  - Not just for “experts”

- Reduce the number of artifacts
  - Make unnecessary artifacts optional
  - Metadata should be simple or non-existent

- Configuration by exception
EJB 3.0 Goals

- Add support for popular features and practices
  - Separation of concerns, interceptors, injection
- Standardize persistence API for Java platform
  - Based on success of leading ORM solutions
- Enable persistence in both Java EE and Java SE environments
- Allow pluggability of Persistence Providers into Java EE Containers
- Guarantee backward compatibility
Agenda

1. Introduction - The Early Years
2. EJB 3.0 Goals
3. A Simplified Component Model
4. Java Persistence API
5. Summary
Session Bean / MDB

- POJO and POJI
  - EJB class is POJO
  - Business interface does not have to extend EJBOBJECT
  - Home interfaces not needed
  - Annotations
    - EJB Type
    - Local / Remote Business Interfaces
    - Transaction Attributes
    - Callbacks and Interceptors
Session Bean / MDB

- Dependency Injection
  - Support for both field and property injection
  - Declarative through either annotations or XML
  - Most Java EE resource types:
    - ejb-ref, ejb-local-ref
    - resource-ref, resource-env-ref
    - environment-entry
    - EntityManagers and factories
Session Bean / MDB

- Enhanced lifecycle methods
  - Custom methods on bean class for standard lifecycle events
  - Interceptor classes may be used to delegate lifecycle management

- Interceptors
  - Interceptor classes may be registered to intercept business methods
  - Provides equivalent of AOP around advice
EJB 2.1 Session Bean Interfaces

```java
public interface Cart extends EJBObject {
    public void add(String item) throws RemoteException;
    public Collection getItems() throws RemoteException;
    public void completeOrder() throws RemoteException;
}

public interface CartHome extends EJBHome {
    public Cart create()
        throws CreateException, RemoteException;
}
```
EJB 2.1 Session Bean Class

```java
public class CartEJB implements SessionBean {
    protected Collection items = new ArrayList();

    public void add(String item) { items.add(item); }
    public Collection getItems() { return items; }
    public void completeOrder(){ .. }

    public void ejbCreate(){}
    public void ejbRemove(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void setSessionContext(                   SessionContext context){}
}
```
EJB 2.1 Deployment Descriptor

```xml
<session>
  <display-name>Shopping Cart</display-name>
  <ejb-name>MyCart</ejb-name>
  <home>CartHome</home>
  <remote>Cart</remote>
  <ejb-class>CartEJB</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
</session>

... 

<assembly-descriptor>
  <container-transaction>
    <method> ... 
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
```
Problems

- Heavy-weight programming view
  - Too many interfaces for a simple bean
- Implementation constraints
  - Must implement javax.ejb.SessionBean
  - Can’t implement actual business interface
  - Life cycle code clutter
- Deployment descriptor
  - Verbose and hard to read
  - Contains redundant contextual information
  - Coupled with classes, but spatially dislocated
EJB 3.0 Session Bean Interface

```java
@Remote
public interface Cart {
    public void add(String item);
    public Collection getItems();
    public void completeOrder();
}
```
EJB 3.0 Session Bean Class

```java
@Stateful
public class CartBean implements Cart {
    private ArrayList items = new ArrayList();

    public void add(String item) {
        items.add(item);
    }

    public Collection getItems() {
        return items;
    }

    @Remove
    public void void completeOrder() {...}

```
Deployment Descriptor

```xml
<session>
  <display-name>Shopping Cart</display-name>
  <ejb-name>MyCart</ejb-name>
  <home>com.acme.CartHome</home>
  <remote>com.acme.Cart</remote>
  <ejb-class>com.acme.CartEJB</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
</session>

<assembly-descriptor>
  ...
  <container-transaction>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```
Some Simplifications

- Eliminate requirement for Home Interface
  - Not needed for session beans
- Business interface is a POJI
  - Bean implements it
  - Bean can have multiple business interfaces
  - EJBOBJECT removed from client view
  - RemoteExceptions removed from programmer & client view
- Eliminated requirement for callback methods
  - Don’t need to implement javax.ejb.SessionBean
Simplification Through Defaults

- Minimize use of metadata
  - Defaulting of interface generation
  - Defaulting of names
  - Defaulting use of transaction management types
  - Defaulting of transaction attributes
  - Default use of unchecked methods
  - Default local/remoteness
  - Default use of caller identity
EJB 2.1 Versus 3.0 - Example

- **Lines of Code**
  - EJB 2.1: 670
  - EJB 3.0: 651

- **Lines of XML**
  - EJB 2.1: 326
  - EJB 3.0: 44

- **Classes**
  - EJB 2.1: 16
  - EJB 3.0: 6

- **Descriptors**
  - EJB 2.1: 5
  - EJB 3.0: 2
Simplified Access to Environment

- Get JNDI APIs out of developer's view
  - Not a good “Hello World” experience
- Techniques / mechanisms
  - Declarative expression of dependencies in metadata
  - Container injection of resource entries, *etc.*
  - Simple programmatic lookup mechanisms
- Different usages, both have their place
  - Injection: Less code, facilitates testability
  - Lookup: More flexible, dynamic, Java clients
Injection

- Field injection
  - Happens at “setEJBContext time”
  - Container figures out what to inject

- Setter injection
  - Also happens at “setEJBContext time”
  - Injection can be done manually outside the container
  - Offers better testability

- Can inject resources, EJBContext, EntityManager, session bean references

- J2EE scoped, not just EJB
Examples

```java
@EJB AdminService bean;
public void privilegedTask() {
    bean.adminTask();
}
```

```java
@Resource SessionContext sc;
...
    TimerService ts = sc.getTimerService();
```

```java
@Resource(name="myDB")
public void setDataSource(DataSource ds) {
    dataSource = ds;
}
```
Callbacks and Interceptors

- Callbacks occur at a given life cycle state or event
- Can only get notified for events that apply to the component type
  - MDB – PostConstruct, PreDestroy
  - SLSB – PostConstruct, PreDestroy
  - SFSB – PostConstruct, PreDestroy, PrePassivate, PostActivate
- Interceptors provide AOP-like advice on business methods
- Callbacks or interceptors may be configured using annotations or XML
Callback Example

```java
@Stateful
public class AccountSession implements Account {
    AuditSink audit;

    @PostConstruct @PostActivate
    private void initAuditSink() {
        auditSink = AuditSink.obtainAuditSink();
    }

    @PreDestroy @PrePassivate
    public void cleanUpAuditSink() {
        AuditSink.returnAuditSink(auditSink);
    }

    public void doStuff() {
        ...
    }

    @Remove
    public void logOff() {
        ...
    }
}
```
Interceptor Example

```java
@Stateless
@Interceptors(com.acme.CustomSecurity.class)
public class AccountManagementBean
    implements AccountManagement {

    public void createAccount(int acctNum,
                   AccountDetails details) { ... }
    public void deleteAccount(int acctNum) { ... }
    public void activateAccount(int acctNum) { ... }
    public void deactivateAccount(int acctNum) { ... }
    ...
}
```
Interceptor Example

```java
public class CustomSecurity {

    @AroundInvoke
    public Object customSecurity(InvocationContext inv) throws Exception {
        Principal user = inv.getEJBContext().getCallerPrincipal();
        if (user.getName() != "Mike")
            throw new SecurityException("Not Mike!");
        return inv.proceed();
    }
}
```
Using XML

- Instead of annotating the bean class with @Interceptors, we could add a snippet of XML to the session bean as follows:

```xml
<interceptor-binding>
    <ejb-name>AccountManagementBean</ejb-name>
    <interceptor-class>
        com.acme.CustomSecurity
    </interceptor-class>
</interceptor-binding>
```
Agenda

1. Introduction - The Early Years
2. EJB 3.0 Goals
3. A Simplified Component Model
4. Java Persistence API
5. Summary
EJB Persistence API Goals

- Simplify programming model
  - Use POJO’s
  - Remove unnecessary artifacts
- Improve modelling capabilities
  - Inheritance and polymorphism
  - O/R mapping
- Extend query capabilities and query language
- Make instances usable outside the container
  - Facilitate testability
  - Remove need for anti-patterns
Along The Way

- Persistence API expanded
  - Evolved into “common” Java persistence API that can be used both inside and outside Java EE Containers
  - Merger of expertise from Hibernate, TopLink, JDO, EJB vendors and individuals
  - API draws from all of these sources

- Support for pluggability
  - Can mix-and-match EJB Containers and third-party persistence providers
  - Can switch Container or Persistence Provider in isolation from the other
POJO Entities

- Concrete classes (no longer abstract)
- No required interfaces
  - No required business interfaces
  - No required callback interfaces
- Support new() for instance creation
- Direct access or getter/setter methods
  - Can contain logic (e.g. for validation, etc.)
- “Managed” by an EntityManager
- Can leave the Container (become “detached”)

POJO Entities

- Concrete classes (no longer abstract)
- No required interfaces
  - No required business interfaces
  - No required callback interfaces
- Support new() for instance creation
- Direct access or getter/setter methods
  - Can contain logic (e.g. for validation, etc.)
- “Managed” by an EntityManager
- Can leave the Container (become “detached”)

POJO Entities

- Concrete classes (no longer abstract)
- No required interfaces
  - No required business interfaces
  - No required callback interfaces
- Support new() for instance creation
- Direct access or getter/setter methods
  - Can contain logic (e.g. for validation, etc.)
- “Managed” by an EntityManager
- Can leave the Container (become “detached”)

POJO Entities

- Concrete classes (no longer abstract)
- No required interfaces
  - No required business interfaces
  - No required callback interfaces
- Support new() for instance creation
- Direct access or getter/setter methods
  - Can contain logic (e.g. for validation, etc.)
- “Managed” by an EntityManager
- Can leave the Container (become “detached”)

POJO Entities

- Concrete classes (no longer abstract)
- No required interfaces
  - No required business interfaces
  - No required callback interfaces
- Support new() for instance creation
- Direct access or getter/setter methods
  - Can contain logic (e.g. for validation, etc.)
- “Managed” by an EntityManager
- Can leave the Container (become “detached”)

POJO Entities

- Concrete classes (no longer abstract)
- No required interfaces
  - No required business interfaces
  - No required callback interfaces
- Support new() for instance creation
- Direct access or getter/setter methods
  - Can contain logic (e.g. for validation, etc.)
- “Managed” by an EntityManager
- Can leave the Container (become “detached”)
Persistence Contexts

- Persistence context
  - Set of managed entity instances
  - Persistent identity equivalent to Java object identity
  - Analogous to “transaction context” in JTA

- Scope of a persistence context
  - Normally scoped to a transaction
  - “Extended” persistence context may span multiple sequential transactions
Operations on Entities

- **EntityManager API**
  - `persist()` - Insert the identity of an entity into the db
  - `remove()` - Delete the persistent identity of the entity from the db
  - `refresh()` - Reload the entity state from the db
  - `merge()` - Synchronize the state of detached entity with the pc
  - `find()` - Execute a simple PK query
  - `createQuery()` - Create query instance using dynamic EJB QL
  - `createNamedQuery()` - Create instance for a predefined query
  - `createNativeQuery()` - Create instance for an SQL query
  - `contains()` - Determine if entity is managed by pc
  - `flush()` - Force synchronization of pc to database
Persist Operation

```java
public Order createNewOrder(Customer customer) {
    Order order = new Order(customer);
    entityManager.persist(order);
    return order;
}
```

- Can only pass new or managed instances to `persist()`
- Exception thrown if object was detached
- Exception may be thrown immediately or at commit time
Find and Remove Operation

```java
public void removeOrder(Long orderId) {
    Order order =
        entityManager.find(Order.class, orderId);
    entityManager.remove(order);
}
```

- Can only pass managed instances to `remove()`
- Exception thrown if object was detached
- Detached instances must first be merged, or managed instances with same persistence identity must be obtained
Merge Operation

```java
public OrderLine updateOrderLine(OrderLine orderLine) {
    return entityManager.merge(orderLine);
}
```

- Detached instances become managed
- Detached state merged into the persistence context
- Merge returns managed instance with the same persistent identity but with different Java object identity
- Managed objects ignored
Extended Persistence Contexts

- Acts as an entity cache of managed instances when clients access the same component across multiple requests
- Spans multiple sequential database/JTA transactions
- Applies to:
  - Stateful session beans
  - HTTP session
- Optimistic locking semantics – may need to retry
Without XPC

```java
@Stateless
public ShoppingCartBean implements ShoppingCart {
    @PersistenceContext
    private EntityManager entityManager;

    public OrderLine createOrderLine(
            Product product, Order order) {
        OrderLine orderLine = new OrderLine(order, product);
        entityManager.persist(orderLine);
        return orderLine;
    }

    public OrderLine updateOrderLine(
            OrderLine orderLine) {
        return entityManager.merge(orderLine);
    }
}
```
With XPC

```java
@Stateful
public ShoppingCartBean implements ShoppingCart {
  @PersistenceContext(type=EXTENDED)
  private EntityManager entityManager;
  private OrderLine orderLine;

  public OrderLine getOrderLine() { return orderLine; }

  public OrderLine createOrderLine(…) {
    ...
  }

  public OrderLine updateOrderLine(int quantity) {
    return getOrderLine().setQuantity(quantity);
  }

  @Remove public void finish() {
  }
}
```
Entity Callbacks

- An EntityListener may be attached to certain defined entity lifecycle events
  - PrePersist—when the application calls persist()
  - PostPersist—after the SQL INSERT
  - PreRemove—when the application calls remove()
  - PostRemove—after the SQL DELETE
  - PreUpdate—when the container detects that an instance is dirty
  - PostUpdate—after the SQL UPDATE
  - PostLoad—after an instance was loaded

- Attached to an entity class by specifying an @EntityListener annotation
Entity Callbacks

@Entity
@EntityListener(AuditCallbackLister.class)
public class Order extends Auditable {
    ...
}

Entity Callbacks

```java
@MappedSuperclass
public class Auditable {
    private Date createTime;
    private User createdBy;

    public Date getCreateTime() { return createTime; }
    public void setCreateTime(Date dt) {
        createTime = dt;
    }
    @ManyToMany
    public User getCreatedBy() { return createdBy; }
    public void setCreatedBy(User user) {
        createdBy = user;
    }
    ...
}
```
public class AuditCallbackListener {
    @PrePersist
    public setCreateInfo(Auditable auditable) {
        auditable.setCreateTime(new Date());
        auditable.setCreatedBy(User.getCurrent());
    }
}
Object/Relational Mapping

- Specified as annotations or XML
- Logical and physical mapping views
  - Logical—object model (e.g. `@OneToMany`)
  - Physical—DB tables and columns (e.g. `@Table`)
- Support for basic, serialized objects and LOBs
- Unary, n-ary relationship mappings
- Rules for defaulting of DB table and column names
- Access to object state using fields or properties
  - Multiple tables, composite relationship keys
Primary Keys

- Id field required in the domain entity
  1. Can be a simple field using @Id
     ```java
     @Id int custId;
     ```
  2. Use @EmbeddedId to indicate a single id field to store an instance of a composite PK class
     ```java
     @EmbeddedId CustPK id;
     ```
  3. Compatibility with EJB 2.x style of composite PK class
     ```java
     @IdClass(CustPK.class)
     ```
Fetch Mode

- Hint to the Container to defer loading specific fields or relationships of the object until they are accessed
- Specified as metadata on the mappings
- Defaults applied by the Container
  - Simple and single-valued relationships — EAGER
  - Multi-valued relationships — LAZY
- Entity should not access its data directly from fields
Cascade Mode

- Can cause specific life cycle operations to cascade across relationships
- Specified as metadata on the mappings
- Can cascade combinations of
  - PERSIST, MERGE, REMOVE, REFRESH, ALL
- Default is for no cascading to occur
- Used when EntityManager operations are invoked
- Deployment configuration level will allow setting all entities in a given configuration
Simple Mappings

- Direct mappings of fields to columns
  - @Basic - field type maps to standard column type
  - @Lob - field maps to BLOB or CLOB column type
- Used in conjunction with @Column (physical mapping annotation)
- Defaults to the type deemed most appropriate if no mapping annotation is present
- Can override any of the defaults
Simple Mappings

```java
@Entity(access=FIELD)
public class Customer {
    @Id
    int id;

    String name;

    int c_rating;

    @Lob
    Image photo;
}
```
Simple Mappings

```java
@Entity(access=FIELD)
public class Customer {
    @Id
    int id;

    String name;
    @Column(name="CREDIT")
    int c_rating;

    @Lob
    Image photo;
}
```
Relationship Mappings

- Common relationship mappings supported
  - @ManyToMany, @OneToMany - collection of entities
  - @ManyToOne, @OneToOne - single entity
- Unidirectional or bidirectional
- Owning and inverse sides, owning side specifies the physical mapping
  - @JoinColumn to specify foreign key column
  - @JoinTable for decoupling relationship from source entity (e.g. ManyToMany)
ManyToOne Mapping

```java
@Entity(access=FIELD)
public class Customer {
    @Id
    int id;
    @ManyToOne
    Address addr;
}
```

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ADDR_ID</td>
</tr>
<tr>
<td></td>
<td>ID</td>
</tr>
</tbody>
</table>
OneToMany Mapping

@ManyToOne
@OneToMany(mappedBy="cust")

CUSTOMER
ID . . .

ORDER
ID | CUST_ID | . . .

```java
class Customer {
    @Id
    int id;
    ...
    @OneToMany(mappedBy="cust")
    Set<Order> orders;
}
```

```java
class Order {
    @Id
    int id;
    ...
    @ManyToOne
    Customer cust;
}
```
@Entity(access=FIELD)
public class Customer {
    @Id
    int id;
    ...  
    @ManyToMany
    Collection<Phone> phones;
}

@Entity(access=FIELD)
public class Phone {
    @Id
    int id;
    ...  
    @ManyToMany(mappedBy="phones")
    Collection<Customer> custs;
}
ManyToMany Mapping

```java
@Entity(access=FIELD)
public class Customer {
    ...
    @ManyToMany
    @JoinTable(table=@Table(name="CUST_PHONE"),
               joinColumns=@JoinColumn(name="CUST_ID"),
               inverseJoinColumns=@JoinColumn(name="PHON_ID"))
    Collection<Phone> phones;
}
```

**CUSTOMER**

| ID | ... |

**PHONE**

| ID | ... |

**CUST_PHONE**

| CUST_ID | PHON_ID |
Mapping of Embedded Objects

```java
@Entity(access=FIELD)
public class Customer {
    @Id
    int id;
    @Embedded
    CustomerInfo info;
}

@Embeddable(access=FIELD)
public class CustomerInfo {
    String name;
    int credit;
    Image photo;
}
```

<table>
<thead>
<tr>
<th>CUSTOMER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Inheritance

- Entities can extend
  - Other entities — concrete or abstract
  - Non-entity classes — concrete or abstract

- Map inheritance hierarchies in three ways
  1. Single table — all classes stored in the same table
  2. Joined — Each class (concrete or abstract) stored in a separate table
  3. Table per concrete class — Each concrete class stored in separate table (optional)
Object Model

```java
public abstract class Animal {
    int id;
    String name;
}

public class LandAnimal extends Animal {
    int legCount;
}

public class AirAnimal extends Animal {
    short wingSpan;
}
```
**Data Models**

**Single table:**

<table>
<thead>
<tr>
<th>ANIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

**Joined:**

- **LAND_ANML**
  | ID | LEG_COUNT |

- **AIR_ANML**
  | ID | WING_SPAN |

**Table per Class:**

- **LAND_ANML**
  | ID | NAME | LEG_COUNT |

- **AIR_ANML**
  | ID | NAME | WING_SPAN |
Queries

- Dynamic or statically defined and named
- Criteria using EJB QL
- Native SQL support (when required)
- Named parameters bound at execution time
- Pagination and ability to restrict size of result
- Single / multiple-entity results, data projections
- Bulk update and delete operation on an entity
- Standard hooks for vendor-specific hints
Dynamic Queries

```java
public class CustomerQueries {

    EntityManager em = getEntityManager();

    public List findCustByName (String name) {
        return em.createQuery (        "SELECT c FROM Customer c " +        "WHERE c.name LIKE :custName")        .setParameter("custName", name)        .setMaxResults(10)        .getResultList();
    }
}
```
### Named Queries

```java
@NamedQuery(name="findCustomersByNamedQuery", 
    query="SELECT c FROM Customer c 
         WHERE c.name LIKE :custName"
)
@Entity public class Customer { … }

public List findCustByName (String name) {
    return 
        em.createNamedQuery("findCustomersByNamedQuery")
            .setParameter("custName", name)
            .setMaxResults(10)
            .getResultList();
}
```
EJB QL Enhancements

- Support for joins in the from clause
  - `select o from Order o left join o.lineItems li where li.amount > 100`

- Support for subselects
  - `select o from Order o where exists(select li from o.lineItems li where li.amount > 100)`

- Support for aggregation
  - `select o.id, sum(li.amount) from Order o join o.lineItems li group by o.id`

- Additional EJB QL functions
  - `trim()`, `locate()`, `concat()`, `substring()`, `lower()`, `upper()`, `length()`, `abs()`, `sqrt()`, `mod()`, `size()`

- Update and delete operations
  - `delete from Customer cust where cust.id = 12345`
  - `update OrderLine ol set ol.fulfilled = 'Y' where ol.order.id = 9876543`
Agenda

1. Introduction - The Early Years
2. EJB 3.0 Goals
3. A Simplified Component Model
4. Java Persistence API
5. Summary
EJB 3.0 Summary

✔ POJO-centric view of all enterprise beans
✔ POJI’s for session beans
✔ Simplification of environment access
✔ Simplification of entity beans
  ✔ Clear O/R mapping orientation
  ✔ Improvement of query language capabilities
✔ Metadata is major enabling technology
✔ Specification Public Draft is available
EJB 3.0 Components Summary

✓ EJB components are now easier to use but also more powerful than at any time in the history of Java EE
  ✓ Fewer artifacts
  ✓ Fewer programming constraints
  ✓ Easier access to resources through dependency injection
  ✓ Choice of using annotations or XML for configuration
  ✓ Flexible callback and interceptor mechanisms
EJB 3.0 Persistence Summary

- The Java Persistence API has been modeled after the most popular commercial and open source products
  - POJO entities that are usable in other application tiers
  - Dynamic queries and flexible query options
  - EntityManager API for managing persistent entities
  - Inheritance, polymorphism, O/R mapping, etc.
Summary

✓ EJB 3.0 has evolved from being an ivory tower spec to a standardization of best practices

✓ Enthusiasm from developers and vendors is staggering

✓ Reference Implementation will put EJB 3.0 in the hands of everybody that wants to use it
Summary

Standardization based upon existing and successful products and practices is establishing EJB components and the EJB Persistence API as the enterprise standard for the foreseeable future.
Links

- Reference Implementation of EJB 3.0 (joint project between Sun and Oracle using Oracle TopLink):
  
  http://glassfish.dev.java.net/

- Eclipse project in WTP will provide an open source environment for EJB 3.0 development

Links

- EJB 3.0 Specification Public Draft


- OracleAS EJB 3.0 Preview release (Free!)
  Download from:

  http://otn.oracle.com/ejb3