

# EJB 3.0 Persistence

Sang Shin  
Java Technology Architect  
[sang.shin@sun.com](mailto:sang.shin@sun.com)  
[javapassion.com](http://javapassion.com)

# Agenda

- EJB 3.0 Persistence Requirements
- EJB 3.0 Programming Model
- Entity Life-cycle & Entity Manager
- Detached Entities
- Entity Relationships
- Demo: EJB 3.0 Persistence using NetBeans Enterprise Pack
- O/R Mapping
- Entity Listeners
- Query
- GlassFish

# EJB 3.0 Persistence Requirements

# EJB 3.0 Persistence Requirements

- Simplification of the persistence model
- Light-weight persistence model
  - > In terms of programming and deployment model as well as runtime performance
- Testability outside of the containers
  - > Create test clients that would use entities in a non-managed environment
- Domain modelling through inheritance and polymorphism
- Object/Relational (O/R) mapping
- Extensive querying capabilities

# Common Java Persistence Between J2SE and J2EE Environments

- Persistence API expanded to include use **outside** of EJB container
- Evolved into “common” Java persistence API
  - > You can use new Java persistence API in Java SE, Web, and EJB applications
- Support for pluggable, third-party persistence providers

# EJB 3.0 Programming Model

# EJB 3.0 Persistence Programming Model

- Entity is a POJO (Plain Old Java Object)
- Use of Annotation to denote a POJO as an entity

**// @Entity is an annotation**

**// It annotates Employee POJO class to be Entity**

**@Entity**

```
public class Employee {  
    // Persistent/transient fields  
    // Property accessor methods  
    // Persistence logic methods  
}
```

# Persistence Entity Example

```
@Entity public class Customer {
```

Annotated as "Entity"

```
    private Long id;
    private String name;
    private Address address;
    private Collection<Order> orders = new HashSet();
```

```
public Customer() {}
```

```
@Id public Long getID() {
    return id;
}
```

```
protected void setID (Long id) {
    this.id = id;
}
```

@Id denotes primary key



Getters/setters to access state



...

# Persistence Entity Example (Contd.)

...

**// Relationship between Customer and Orders**

**@OneToMany**

```
public Collection<Order> getOrders() {  
    return orders;  
}
```

```
public void setOrders(Collection<Order> orders) {  
    this.orders = orders;  
}
```

**// Other business methods**

...

```
}
```

# Client View: From Stateless Session Bean

```
@Stateless public class OrderEntry {  
  
    // Dependency injection of Entity Manager for  
    // the given persistence unit  
    @PersistenceContext  
    EntityManager em;  
  
    public void enterOrder(int custID, Order newOrder){  
  
        // Use find method to locate customer entity  
        Customer c = em.find(Customer.class, custID);  
        // Add a new order to the Orders  
        c.getOrders().add(newOrder);  
        newOrder.setCustomer(c);  
    }  
  
    // other business methods  
}
```

# Animal Entity Class

**@Entity** ← Annotated as “Entity”  
 public class Animal implements Serializable {

**@Column(name="animalName")**  
 String name;

String kind;  
 String weight;

**@ManyToOne**  
 Pavilion pavilion;

**@Id** ← @Id denotes primary key  
**@GeneratedValue(strategy = GenerationType.AUTO)**  
 private Long id;

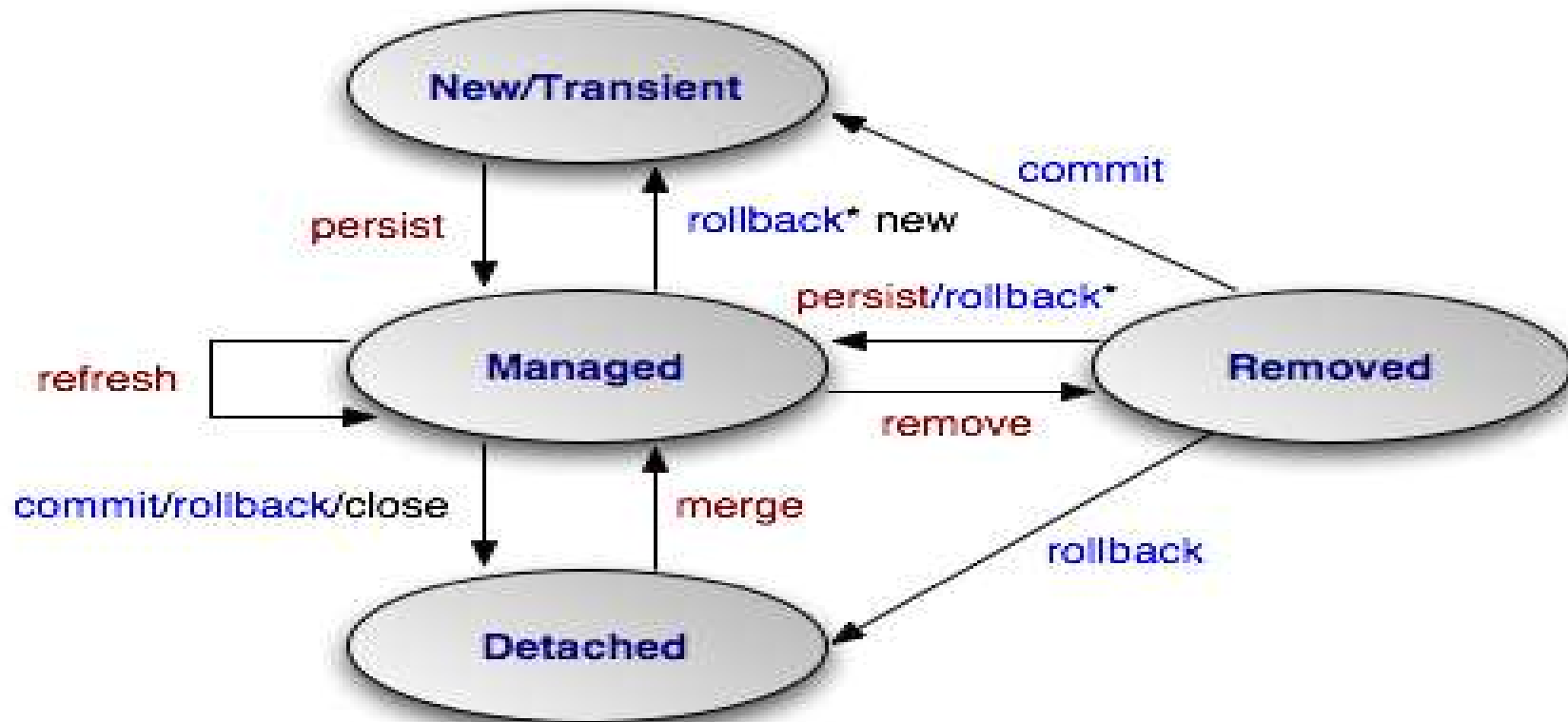
...

# Client Code: From Java SE Client

```
public class AnimalController {  
    ...  
    public String create() {  
        EntityManager em = getEntityManager();  
        try {  
            utx.begin();  
            em.persist(animal);  
            utx.commit();  
            addSuccessMessage("Animal was successfully created.");  
        } catch (Exception ex) {  
            ...  
        }  
        em.close();  
        return "animal_list";  
    }  
}
```

# Entity Life-cycle

# Entity Lifecycle



\* = Extended persistence context

# Entity Manager Controls Life-cycle of Entities

# EntityManager

- Similar in functionality to Hibernate Session, JDO PersistenceManager, etc.
- Controls lifecycle of entities
  - > `persist()` - insert an entity into the DB
  - > `remove()` - remove an entity from the DB
  - > `merge()` - synchronize the state of detached entities
  - > `refresh()` - make sure the persistent state of an instance is synchronized with the values in the datastore

# Persist Operation

```
public Order createNewOrder(Customer customer) {  
    Order order = new Order(customer);  
  
    // Transitions new instances to managed. On the  
    // next flush or commit, the newly persisted  
    // instances will be inserted into the datastore.  
    entityManager.persist(order);  
  
    return order;  
}
```

# Find and Remove Operations

```
public void removeOrder(Long orderId) {  
    Order order =  
        entityManager.find(Order.class, orderId);  
  
    // The instances will be deleted from the datastore  
    // on the next flush or commit. Accessing a  
    // removed entity has undefined results.  
    entityManager.remove(order);  
}
```

# Merge Operation

```
public OrderLine updateOrderLine(OrderLine
    orderLine) {

    // The merge method returns a managed copy of
    // the given detached entity. Changes made to the
    // persistent state of the detached entity are
    // applied to this managed instance.
    return entityManager.merge(orderLine);
}
```

# Detached Entities

# Detached Entities

- Must implement **Serializable** interface if detached object has to be sent across the wire
- No need for **DTO (Data Transfer Object)** anti-design pattern
- Merge of detached objects can be cascaded

# O/R Mapping

# O/R Mapping

- Comprehensive set of annotations defined for mapping
  - > Relationships
  - > Joins
  - > Database tables and columns
  - > Database sequence generators
  - > Much more
- Specified using standard description elements in a separate mapping file or within the code as annotations

# Simple Mappings

```

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

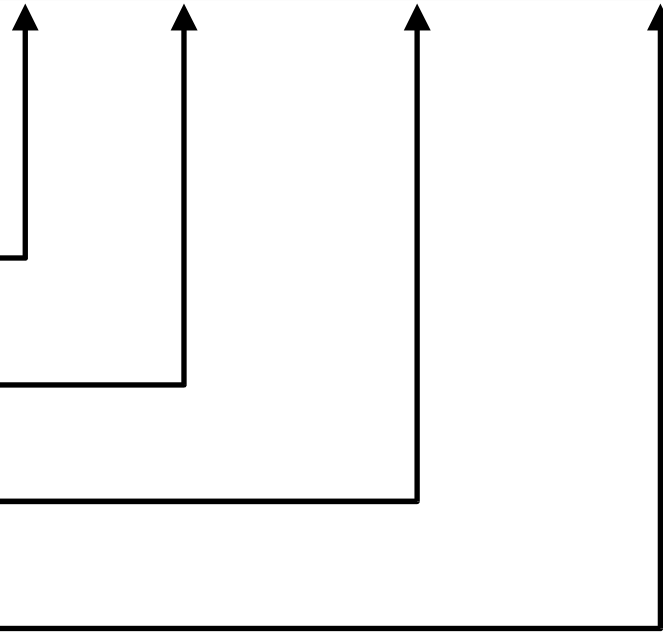
    String name;

    int c_rating;

    @Lob
    Image photo;
}

```

CUSTOMER			
ID	NAME	C_RATING	PHOTO



# Simple Mappings

```

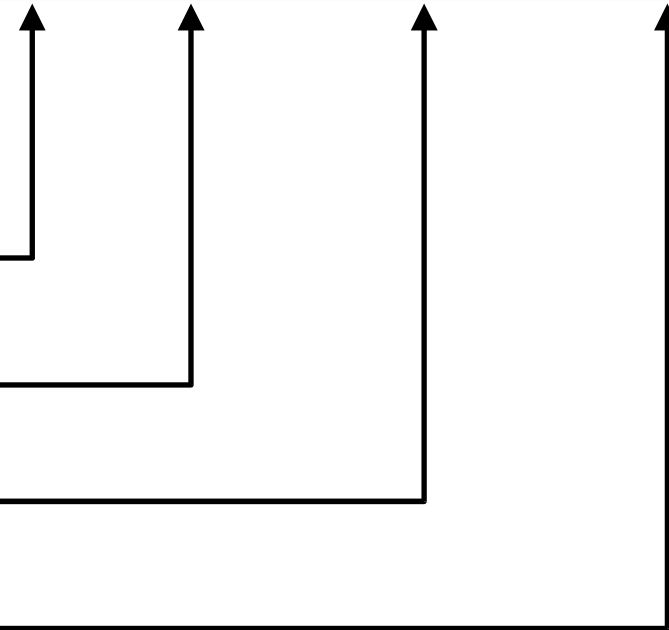
@Entity(access=FIELD)
public class Customer {

    @Id
    int id;

    String name;

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

    @Lob
    Image photo;
}
    
```



# O/R Mapping Examples

## @Entity

```
@Table(name="EMPLOYEE", schema="EMPLOYEE_SCHEMA")
uniqueConstraints=
{@UniqueConstraint(columnNames={"EMP_ID", "EMP_NAME"})}
public class EMPLOYEE {
    ...
    @Column(name="NAME", nullable=false, length=30)
    public String getName() { return name; }
}
```

---

## @Version

```
@Column("OPTLOCK")
protected int getVersionNum() { return versionNum; }
```

---

## @ManyToOne

```
@JoinColumn(name="ADDR_ID")
public Address getAddress() { return address; }
```

# Entity Listeners

# Entity Listeners

- Listeners or callback methods are designated to receive invocations from persistence provider at various stages of entity lifecycle
- Callback methods
  - > Annotate callback handling methods right in the entity class or put them in a separate listener class
  - > Annotations
    - > PrePersist / PostPersist
    - > PreRemove/ PostRemove
    - > PreUpdate / PostUpdate
    - > PostLoad

# Entity Listeners: Example – 1

**@Entity**

**@EntityListener(com.acme.AlertMonitor.class)**

**public class AccountBean implements Account {**

**Long accountId;**

**Integer balance;**

**boolean preferred;**

**public Long getAccountId() { ... }**

**public Integer getBalance() { ... }**

**@Transient context**

**public boolean isPreferred() { ... }**

**public void deposit(Integer amount) { ... }**

**public Integer withdraw(Integer amount) throws NSFException {... }**

# Entity Listeners: Example – 2

## **@PrePersist**

```
public void validateCreate() {  
    if (getBalance() < MIN_REQUIRED_BALANCE)  
        throw new AccountException("Insufficient balance to  
                                    open an account");  
}
```

## **@PostLoad**

```
public void adjustPreferredStatus() {  
    preferred =(getBalance() >=  
                AccountManager.getPreferredStatusLevel());  
}  
}
```

# Entity Listeners: Example – 3

```
public class AlertMonitor {
```

```
    @PostPersist
```

```
    public void newAccountAlert(Account acct) {  
        Alerts.sendMarketingInfo(acct.getAccountId(),  
                                acct.getBalance());  
    }
```

```
}
```

# Entity Relationships

# Entity Relationships

- Models association between entities
- Supports unidirectional as well as bidirectional relationships
  - > Unidirectional relationship: Entity A references B, but B doesn't reference A
- Cardinalities
  - > One to one
  - > One to many
  - > Many to one
  - > Many to many

# Entity Relationships: Example

## Many to Many

**@Entity**

```
public class Project {
```

```
    private Collection<Employee> employees;
```

**@ManyToMany**

```
public Collection<Employee> getEmployees() {  
    return employees;  
}
```

```
public void setEmployees(Collection<Employee> employees) {  
    this.employees = employees;  
}
```

```
...  
}
```

# Cascading Behavior

- Cascading is used to propagate the effect of an operation to associated entities
- Cascading operations will work only when entities are associated to the persistence context
  - > If a cascaded operation takes place on detached entity, `IllegalArgumentException` is thrown
- Cascade=PERSIST
- Cascade=REMOVE
- Cascade=MERGE
- Cascade=REFRESH
- Cascade=ALL

# Entities Inheritance

# Entity Inheritance

## Mapping Classes to Tables

- Use Java™ application metadata to specify mapping
- Support for various inheritance mapping strategies
  - > **Single table**
    - > All the classes in a hierarchy are mapped to a single table
    - > Root table has a discriminator column whose value identifies the specific subclass to which the instance represented by row belongs
  - > **Table per class**
    - > Each class in a hierarchy mapped to a separate table and hence, all properties of the class (incl. inherited properties) are mapped to columns of this table
  - > **Joined subclass**
    - > The root of the hierarchy is represented by a single table
    - > Each subclass is represented by a separate table that contains fields specific to the subclass as well as the columns that represent its primary key(s)

# Inheritance Mapping Example

**@Entity**

**@Table(name="CUST")**

**@Inheritance(strategy=SINGLE\_TABLE,  
discriminatorType=STRING,  
discriminatorValue="CUST")**

**public class Customer {...}**

**@Entity**

**@Inheritance(discriminatorValue="VCUST")**

**public class ValuedCustomer extends Customer{...}**

# Query

# EJB-QL Enhancements

- Bulk update and delete operations
- Group By / Having
- Subqueries
- Additional SQL functions
  - > UPPER, LOWER, TRIM, CURRENT\_DATE, ...
- Polymorphic queries
- Support for dynamic queries in addition to named queries or static queries

# Polymorphic Queries

- All Queries are polymorphic by default
  - > That is to say that the FROM clause of a query designates not only instances of the specific entity class to which it explicitly refers but of subclasses as well

```
select avg(e.salary) from Employee e where e.salary > 80000
```

This example returns average salaries of all employees, including subtypes of Employee, such as Manager.

# Joins

- Adds keyword JOIN in EJB-QL
- Supports
  - > Inner Joins
  - > Left Joins/Left outer joins
  - > Fetch join
    - > Enables pre-fetching of association data as a side-effect of the query

```
SELECT DISTINCT c FROM Customer c LEFT JOIN FETCH c.orders  
WHERE c.address.state = 'MA'
```

# Dynamic Queries

**// Build and execute queries dynamically at runtime.**

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

# Named Queries

// Named queries are a useful way to create reusable queries

```
@NamedQuery(  
    name="findCustomersByName",  
    queryString="SELECT c FROM Customer c" +  
                "WHERE c.name LIKE :custName"  
)
```

```
@PersistenceContext public EntityManager em;  
List customers = em.createNamedQuery("findCustomersByName").  
setParameter("custName", "smith").getResultList();
```

# Transactions

# Transactions

- Local as well as XA transactions supported
- Entity managers define support for either JTA or local transactions at the time of entity manager factory creation
- JTA entity managers
  - > Used in Java EE i.e. Managed environments
  - > Container-managed entity managers must support JTA
- Local entity managers
  - > [EntityTransaction](#) API can be used for application-controlled local transactions

# Transactions – Example

**@Stateless**

**@Transaction(REQUIRED)**

**public class ShoppingCartImpl implements ShoppingCart {**

**@PersistenceContext EntityManager em;**

```
public Order getOrder(Long id) {  
    return em.find(Order.class, id);  
}
```

```
public Product getProduct(String name) {  
    return (Product) em.createQuery  
    ("select p from Product p where p.name = :name")  
    .setParameter("name", name).getSingleResult();  
}
```

# Transactions – Example

```
public LineItem createLineItem(Order order,  
                                Product product, int quantity) {  
    LineItem li = new LineItem(order, product, quantity);  
    order.getLineItems().add(li);  
    em.persist(li);  
    return li;  
}
```

A good example, where we could have used Cascade=PERSIST, REMOVE to avoid making explicit call to em.persist(li)

# Embedded Objects

- `@Embeddable` used to mark an embeddable object
- Embeddable object is stored as intrinsic part of an owning entity
  - > Shares identity of that identity; doesn't have its own identity
- Each persistent field/property of embeddable object is mapped to a database table

# Embedded Objects – Example

**@Embeddable(access=FIELD)**

```
public class EmploymentPeriod {  
    java.util.Date startDate;  
    java.util.Date endDate;  
}
```

**@Embedded**

```
public EmploymentPeriod getEmploymentPeriod() { ... }
```

# Primary Keys and Entity Identity

- Simple or composite primary key must be serializable
- Composite key support
  - > Primary key class must be defined to represent composite key
  - > Can use embedded class as a primary key class
  - > Composite key class is then mapped to a field/property or fields/properties of the entity class

# **Demo: EJB 3.0 Persistence Programming using GlassFish & NetBeans Enterprise Pack**

# Demo Scenario

- <http://www.netbeans.org/kb/55/persistence.html>
- Building Web application which uses Java Persistence API
- Create **Animal** and **Pavilion** Entity classes that reflect ANIMAL and PAVILION database tables
- Create JSF page that access the database tables through Animal and Pavilion Entity classes

# Demo Scenario

- <http://www.netbeans.org/kb/55/ejb3-preview.html>
- Build OrderSystem application
  - > EJB module
  - > Web module
- EJB module
  - > Item entity class
  - > Order entity class
  - > ProcessOrder session class
  - > Order and Items have One to Many relationship
- Web module
  - > JSF application invokes ProcessOrder session bean

**GlassFish:  
The Next Generation  
Sun Java System App Server  
with Java EE 5 (EJB 3.0  
Persistence) Support**

# GlassFish Project

- Supports Java EE 5
  - > EJB 3.0 persistence
  - > Oracle donated TopLink code
- Being developed as open source project

# Project GlassFish – How Does It Work?

- Community contributions encouraged!
  - > Commit privileges to qualified developers
- Download with CVS
- Configure and build with Maven
- Use NetBeans (optional)
  - > To create Java EE applications
  - > To build GlassFish



# Summary, Resources, Sun Developer Network

# EJB 3.0 Persistence Summary

- Simplifies persistence model
- Supports Light-weight persistence model
- Support both J2SE and J2EE environments
- Extensive querying capabilities

# Resources

- Glassfish persistence homepage
  - > <https://glassfish.dev.java.net/javaee5/persistence>
- Persistence support page
  - > <https://glassfish.dev.java.net/javaee5/persistence/entity-persistence-support.html>
- Blog on using persistence in Web applications
  - > [http://weblogs.java.net/blog/ss141213/archive/2005/12/using\\_java\\_pers.html](http://weblogs.java.net/blog/ss141213/archive/2005/12/using_java_pers.html)
- Blog on schema generation
  - > [http://blogs.sun.com/roller/page/java2dbInGlassFish#automatic\\_table\\_generation\\_feature\\_in](http://blogs.sun.com/roller/page/java2dbInGlassFish#automatic_table_generation_feature_in)

# Sun Developer Network

Empowering the Developer

Increasing developer productivity with:

- W Technical articles
- W Tutorials and sample codes
- W Monitored forum support
- W Community involvement through user groups, events, and conferences
- W And more...



<http://developer.sun.com>

# SDN Expert Developer Help

- Project Lifeguard
  - > Sun Developer Network **Beta** Program providing expert developer help
  - > Pilot limited in scope starting November 15, Beta program can end without notice
  - > Products supported during beta:
    - > Java Studio Creator
    - > Java Studio Enterprise (December 2005)
    - > J2SE 1.4/2.5 Core
    - > Sun Studio 10 and 11 for Solaris (December 2005)
  - > Per incident, online help
  - > 24 hour acknowledgment, M-F, 9-5 Bangalore TMZ
  - > Provides help with programming “how to”
- [http://developer.sun.com/developer\\_help](http://developer.sun.com/developer_help)

# Questions?

# EJB 3.0 Persistence Programming Model

- Entity is a POJO (Plain Old Java Object)
- Use of Annotation to denote a POJO as an entity

**// @Entity is an annotation**

**// It annotates Employee POJO class to be Entity**

**@Entity**

```
public class Employee {  
    // Persistent/transient fields  
    // Property accessor methods  
    // Persistence logic methods  
}
```