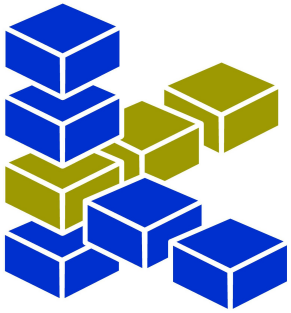


Kodo™

Introduction to the EJB 3 Java Persistence API

Patrick Linskey

plinskey@bea.com

bea
Think liquid.™

Introductions

Patrick Linskey

- EJB Team Lead at BEA
- Frequent presenter at JUGs, conferences (JavaOne, JavaPolis, JAOO, etc.)
- Active member of EJB and JDO expert groups
- Co-author of *Bitter EJB*

EJB 3.0: A Two-Part Specification

- EJB 3 specification consists of two documents:
  - Session and Message-Driven Beans
  - EJB 3 Java Persistence API
- Future versions of the Persistence portion of the EJB 3.0 specification will be spun off into a separate specification
- This talk covers the Java Persistence API (JPA)

Where We Started: Ease-of-Use

- Support simple Java classes
- Improve support for domain modelling
- Expand query capabilities
- Standardize ORM metadata
- Fix Data Transfer Object (DTO) anti-pattern
- Get rid of remote access to entities
- Improve testability

Where We Are

- Java Persistence API expanded to include use outside of Java EE containers
- Evolved into “common” Java persistence API
  - Merger of expertise the entire Java ORM space
- Support for pluggable, third-party persistence providers inside an EJB3 container

EJB3 Java Persistence API

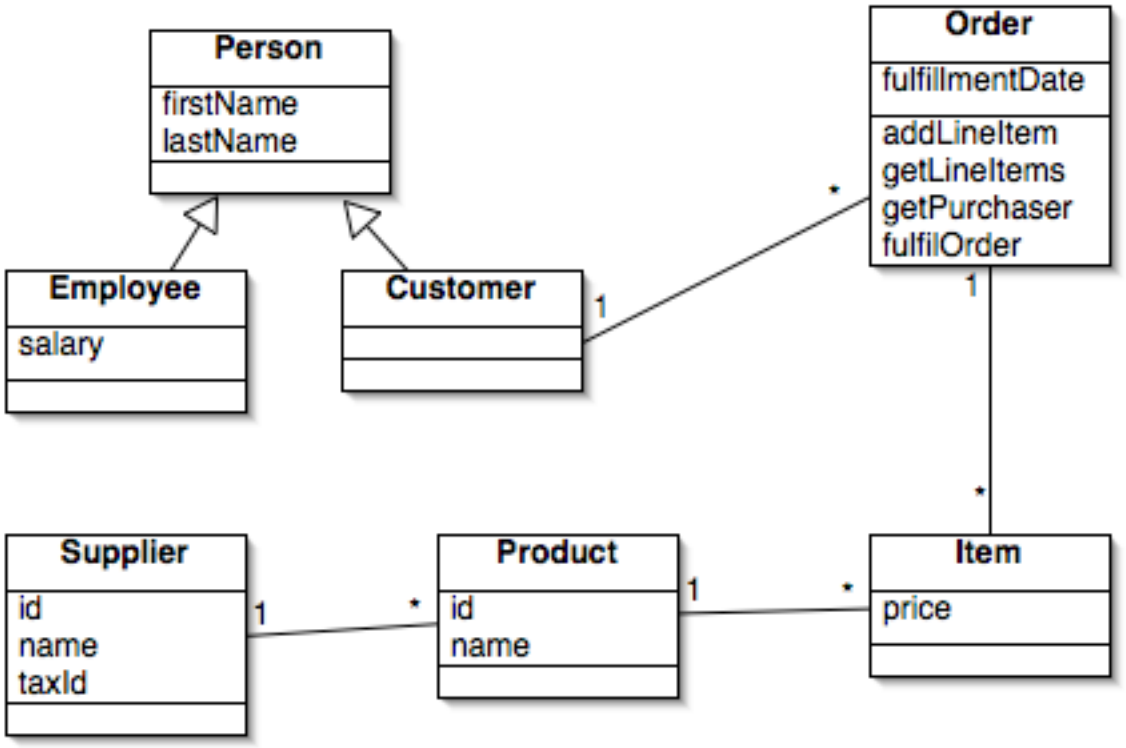
The EJB3 JPA is made of six main parts:

- Domain Object Model Requirements
- Metadata
- Application Programming Interface
- Queries
- Lifecycle Model, including detachment
- Callbacks

Domain Model: Entities

- Entities support standard OO domain modeling techniques
  - Inheritance
  - Encapsulation
  - Polymorphic Relationships
- Can be created with the `new` operator
- There is no required superclass / interface in the EJB3 Persistence API

Example Object Model



Entity Restrictions

- Must have one or more fields mapped to the database primary key
- Must not be `final`
- For portable optimistic locking support, must have a single version field
- Persistent attributes must be represented via member fields (`AccessType.FIELD`) or methods (`AccessType.PROPERTY`)

Persistent Fields in Entities

- Fields marked `transient` are not persisted
- Primitives, primitive wrappers, and Strings automatically become columns in the database
- Relations to other persistent types must be explicitly marked up, and map to foreign key columns / join tables as appropriate

JPA Metadata

- Two types of metadata:
  - **logical** metadata describing the domain object model
  - **mapping** metadata describing how the object model maps to the database schema
- Java 5 annotations and / or XML
- Most metadata settings have intelligent defaults

Logical Metadata

```
@Entity
public class Order {
    @Id @GeneratedValue long pk;
    @Version int oplock;
    @ManyToOne Customer purchaser;
    @OneToMany(cascade=CascadeType.PERSIST) Set<Item> items =
        new HashSet<Item>();
    // no annotation needed for simple types
    Date fulfillmentDate;

    protected Order() { } // for loading from db
    public Order(Customer cust) { purchaser = cust; }

    public void addLineItem(Item item) { items.add(item); }
    public Set<Item> getLineItems() { return items; }
    public Customer getPurchaser() { return purchaser; }
    public void fulfilOrder() { fulfillmentDate = new Date(); }
}
```

Logical Metadata

```
@Entity
public class Customer extends Person {

    @OneToMany(mappedBy="purchaser") Set<Order> orders =
        new HashSet<Order>();

    protected Customer() { } // for loading from db
    public Customer(String fname, String lname) {
        super(fname, lname);
    }

    public void addOrder(Order o) { orders.add(o); }
    public Set<Order> getOrders() { return orders; }
}
```

ORM Metadata

- Can be specified in annotations
  - Useful for mock-ups, proofs-of-concept
- For real-world projects, XML is often a better choice for ORM metadata
  - Schema lifecycle differs from object model lifecycle
  - Schema information, database type often differs in different deployment contexts (production, QA, development, etc.)

ORM Metadata

```
@Entity
@Table(name="PRODUCTS")
public class Product {
    @Id @GeneratedValue
    @Column(name="PRODUCT_PK")
    long id;

    @Version int oplock; // column defaults to "OPLOCK"
    String name; // column defaults to "NAME"

    @ManyToOne
    @JoinColumn(name="SUPP_FK", referencedColumnName="SUPP_PK")
    Supplier supplier;

    ...
}
```

Java Persistence APIs

- Interfaces defined in the `javax.persistence` package
- Key interfaces:
  - `EntityManager`
  - `Query`
- In EJB context, the `EntityManager` is:
  - injected into enterprise beans
  - integrated with the active transaction context

JPA Usage Example

```
abstract class AbstractOrderManager {

    protected Order newOrderForProduct(EntityManager em,
        Customer cust, long productId) {

        Product p = em.find(Product.class, productId);

        Order o = new Order(customer);
        em.persist(o);

        // The new item will be automatically persisted because
        // of the CascadeType.PERSIST directive in Order.java
        o.addLineItem(new Item(p));

        return o;
    }
}
```

Session Bean Code

```
@Stateless
public OrderManagerImpl
    extends AbstractOrderManager
    implements OrderManager {

    @PersistenceContext private EntityManager em;
    @Resource private SessionContext ctx;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Order newOrderForProduct(long productId) {

        Customer cust = (Customer) ctx.lookup("cust");
        return newOrderForProduct(em, cust, productId);
    }
}
```

APIs for J2SE usage

- When using an `EntityManager` without an EJB container, transactions and bootstrapping must be handled by the application code
- Key interfaces and classes:
  - `Persistence`
  - `EntityManagerFactory`
  - `EntityTransaction`

J2SE Example

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("orders");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

try {
    Collection<Customer> customers = loadCustomersFromFile
        (new File("nightly-upload.csv"));
    for (Customer customer : customers)
        em.persist(customer);
    em.getTransaction().commit();
} finally {
    if (em.getTransaction().isActive())
        em.getTransaction().rollback();
}
em.close();
emf.close();
```

Queries: EJBQL

- Fully backward-compatible with EJBQL 2
- Declarative syntax for entity-level queries
- Dynamic or static (named) query support
- SQL-like syntax
- Supports most ORM / OLTP query needs
- Can use raw SQL if needed

EJBQL Syntax Examples

```
// customers 20-30 named 'Joe', ordered by last name
Query q = em.createQuery("select c from Customer c where
    c.firstName = :fname order by c.lastName");
q.setParameter("fname", "Joe");
q.setFirstResult(20);
q.setMaxResults(10);
List<Customer> customers = (List<Customer>) q.getResultList();


// all orders, as a named query
@Entity
@NamedQuery(name="Order:findAllOrders", query="select o from Order o");
public class Order { ... }

Query q = em.createNamedQuery("Order:findAllOrders");
```

EJBQL Syntax Examples

```
// all people, via a custom SQL statement
Query q = em.createNativeQuery("SELECT ID, VERSION, SUBCLASS,
    FIRSTNAME, LASTNAME FROM PERSON", Person.class);
List<Person> people = (List<Person>) q.getResultList();


// single-result aggregate: average order total price
Query q = em.createQuery("select avg(i.price) from Item i");
Number avgPrice = (Number) q.getSingleResult();


// traverse to-many relations
Query q = em.createQuery("select o from Order o
    left join o.items li where li.price > :price");
q.setParameter("price", 1000);
List<Order> orders = (List<Order>) q.getResultList();
```

EJBQL Syntax Examples

```
// bulk update: give everyone a 10% raise
Query q = em.createQuery("update Employee emp
    set emp.salary = emp.salary * 1.10");
int updateCount = q.executeUpdate();


// bulk delete: get rid of fulfilled orders
Query q = em.createQuery("delete from Order o
    where o.fulfilledDate is not null");
int deleteCount = q.executeUpdate();


// subselects: all orders with an expensive line item
Query q = em.createQuery("select o from Order o where exists
    (select li from o.items li where li.price > 10000)");
```

Lifecycle Model

- New instances to be inserted can be registered in-place

```
Person p = new Person("Barney", "Rubble");

em.persist(p);
```

- Records can be registered for deletion:

```
em.remove(p.getAddress());
```

- Instances become detached when the persistence context ends, or upon serialization
- Changes to detached instances can be merged into the current transaction:

```
void updatePerson(Person personDTO) {
    Person p = em.merge(personDTO);
    p.setLastUpdated(new Date());
}
```

Detachment

- Detached instances are useful for transfer to a different physical tier
- Represent a conversion from the persistent domain to the data transfer domain
- Best used with a well-defined service tier
- Should not be used just to communicate between business logic and presentation logic, if collocated on same physical tier

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(OrderValidator.class)
public class Order { ... }

public class OrderValidator {
    @PrePersist
    public void checkItemAmounts(Order o) {
        for (Item item : o.getLineItems())
            if (item.getPrice() < 0)
                throw new IllegalStateException(o);
    }

    @PreRemove
    public void assertNotFulfilled(Order o) {
        if (o.getFulfilledDate() != null)
            throw new IllegalStateException
                ("fulfilled orders cannot be deleted");
    }
}
```

Resources

- EJB3 Preview:

  http://solarmetric.com/kodo/beta/4.0.0EA

- EJB3 Quick Reference:

  http://solarmetric.com/resources/ejb-api-quickref.pdf

- EJBQL Quick Reference:

  http://solarmetric.com/resources/ejbql-quickref.pdf

- More Info:

  info@solarmetric.com