ORACLE

What's New and Exciting in JPA 2.0

Mike Keith
michael.keith@oracle.com

Something About Me

- Java and persistence architect at Oracle
- Almost 20 years experience in server-side and persistence implementations
- Member of JCP expert groups, including JPA 2.0 (JSR-317), Java EE 6 (JSR-316)
- Book: Pro EJB 3: Java Persistence API (Apress)
- Contributer to other specifications (e.g. SCA, SDO, OSGi, etc.)
- Presenter at numerous conferences and events

It's All About You

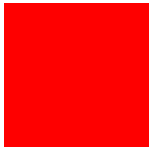
- How many people don't know very much about JPA yet?
- How many people are using JPA?
- How many people are still using proprietary persistence APIs (Hibernate, TopLink, etc.) without JPA?
- How many people that are not using JPA right now are planning to use it later?
- How many people think that they "don't need no stinking JPA" and that they will never use it?

Where We Are

- Vendors have implemented it
- All the cool people are doing it
- The newbies are asking questions about it
- Architects are talking about it
- Consultants are charging for it

Meanwhile…

- We are trying to finish the 2.0 version of the specification in time for the Java EE 6 release



Main Focus

- Standardize useful properties
- Fill in ORM mapping gaps
- Make object modeling more flexible
- Offer simple cache control abstraction
- Allow advanced locking settings
- Provide more hooks for vendor properties
- Add API for better tooling support
- Enhance JP QL query language
- Support Java API based query language
- Integrate with established validation standards

More Standardized Properties

- Some properties are used by every provider
- Need to duplicate JDBC properties in persistence.xml for each provider

```
<properties>
  ...
  <!-- TopLink -->
  <property name="toplink.jdbc.driver"
     value="oracle.jdbc.OracleDriver"/>
  <property name="toplink.jdbc.url"
     value="jdbc:oracle:thin:@localhost:1521:XE"/>
  <property name="toplink.jdbc.user
      value="scott"/>
  <property name="toplink.jdbc.password"
     value="tiger"/>
```

Persistence Unit Properties

```
  ...
  <!-- Hibernate -->
  <property name="hibernate.connection.driver_class"
      value="oracle.jdbc.OracleDriver"/>
  <property name="hibernate.connection.url"
      value="jdbc:oracle:thin:@localhost:1521:XE"/>
  <property name="hibernate.connection.username"
      value="scott"/>
  <property name="hibernate.connection.password"
      value="tiger"/>
  ...
</properties>
```

Persistence Unit Properties

- Should simply be:

```
</properties>
  <property name="javax.persistence.jdbc.driver"
      value="oracle.jdbc.OracleDriver"/>
  <property name="javax.persistence.jdbc.url"
      value="jdbc:oracle:thin:@localhost:1521:XE"/>
  <property name="javax.persistence.jdbc.user"
      value="scott"/>
  <property name="javax.persistence.jdbc.password"
      value="tiger"/>
  ...
</properties>
```

Persistence Unit Properties

Question:

What are **YOUR** favorite properties and which properties do **YOU** think should be standardized?

More Mappings

Can use Join Tables more:

- Unidirectional/bidirectional one-to-one

```
@Entity public class Vehicle {
    …
    @OneToOne @JoinTable(name="VEHIC_REC")
    VehicleRecord record;
    …
}
```

- Bidirectional many-to-one/one-to-many

More Mappings

Can use Join Tables less:

- Unidirectional one-to-many with target foreign key

```
@Entity
public class Vehicle {
        …
    @OneToMany @JoinColumn(name="VEHIC")
    List<Part> parts;
      …
}
```

Additional Collection Support

- Collections of basic objects or embeddables

```
@Entity
public class Vehicle {
    …
    @ElementCollection(targetClass=Assembly.class)
    @CollectionTable(name="ASSEMBLY")
    Collection assemblies;

    @ElementCollection @Temporal(DATE)
    @Column(name="SRVC_DATE")
    @OrderBy("value")
    List<Date> serviceDates;
   …
}
```

Is there a better name?

Should we have separate annotations for each of these mappings?

for basic objects?

Additional Collection Support

- List order can be persisted without being mapped as part of the entity

```
@Entity
public class Vehicle {
    …
    @ManyToMany
    @JoinTable(name="VEH_DEALERS")
    @OrderColumn(name="SALES_RANK")
    List<Dealer> preferredDealers;
   …
}
```

More “Map” Flexibility

Map keys and values can be:

- Basic objects, embeddables, entities

```
@Entity
public class Vehicle {
  …
  @OneToMany
  @CollectionTable(name="PART_SUPP",
     joinColumns=@JoinColumn(name="VEH_ID"))
  @MapKeyJoinColumn(name="PART_ID")
  @JoinColumn(name="SUPP_ID")
  Map<Part,Supplier> suppliers;
    …
}
```

Enhanced Embedded Support

- Embeddables can be nested
- Embeddables can have relationships

```
@Embeddable
public class Assembly {
    …
    @Embedded
    ShippingDetail shipDetails;

    @ManyToOne
    Supplier supplier;
   …
}
```

Access Type Options

- Mix access modes in a hierarchy
- Combine access modes in a single class

```
@Entity @Access(FIELD)
public class Vehicle {
    @Id int id;
    @Transient double fuelEfficiency;

    @Access(PROPERTY) @Column(name="FUEL_EFF")
    public double getDbFuelEfficiency() {
      return convertToMetric(fuelEfficiency);
    }
    public void setDbFuelEfficiency(double fuelEff) {
      fuelEfficiency = convertToImperial(fuelEff);
    }
  …
}
```

Derived Identifiers

- Identifier that includes a relationship
  - Require a additional foreign key field
  - Indicate one of the mappings as read-only
  - Duplicate mapping info

```
@Entity
public class Part {

    @Id int partNo;
    @Column(name="SUPP_ID")
    @Id int suppId;

    @ManyToOne
    @JoinColumn(name="SUPP_ID",
                insertable=false, updatable=false);
    Supplier supplier;
  …
}
```

Derived Identifiers

- Identifiers can be derived from relationships

```
@Entity @IdClass(PartPK.class)
public class Part {

    @Id int partNo;
    @Id @ManyToOne
    Supplier supplier;
    …
}

public class PartPK {
    int partNo;
    int supplier;
    …
}
```

Derived Identifiers

- Can use different identifier types

```
@Entity
public class Part {
    @EmbeddedId PartPK partPk;
    @ManyToOne @MappedById
    Supplier supplier;
    …
}

@Embeddable
public class PartPK {
    int partNo;
    int supplier;
    …
}
```

Shared Cache API

- API for operating on entity cache shared across all EntityManagers within a given persistence unit
  - Accessible from EntityManagerFactory
- Supports only very basic cache operations
  - Can be extended by vendors

```
public class Cache {
   public boolean contains(Class cls, Object pk);

   public void evict(Class cls, Object pk);

   public void evict(Class cls);

   public void evictAll();
}
```

Advanced Locking

- Previously only supported optimistic locking, will now be able to acquire pessimistic locks
- New LockMode values introduced:
  - OPTIMISTIC (= READ)
  - OPTIMISTIC_FORCE_INCREMENT (= WRITE)
  - PESSIMISTIC
  - PESSIMISTIC_FORCE_INCREMENT
- Optimistic locking still supported in pessimistic mode
- Multiple places to specify lock (depends upon need)

Scenario

```
public void applyCharges() {
    em.getTransaction().begin();
    Account acct = em.find(Account.class, acctId);
    // … Validate acct status, calculate charges, etc …
    double balance = acct.getBalance();
    if (charge > 0) {
       acct.setBalance(balance - charge);
    }
    em.getTransaction().commit();
}
```

Option 1

- Read then lock:

```
public void applyCharges() {
    em.getTransaction().begin();
    Account acct = em.find(Account.class, acctId);
    // … Validate acct status, calculate charges, etc …
    double balance = acct.getBalance();
    if (charge > 0) {
        em.lock(acct, PESSIMISTIC);
        acct.setBalance(balance - charge);
    }
    em.getTransaction().commit();
}
```

Option 2

- Read and lock:

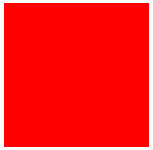
```
public void applyCharges() {
    em.getTransaction().begin();
    Account acct = em.find(Account.class, acctId,
                                          PESSIMISTIC);
    // … Validate acct status, calculate charges, etc …
    double balance = acct.getBalance();
    if (charge > 0) {
       acct.setBalance(balance - charge);
    }
    em.getTransaction().commit();
}
```

Advanced Locking

- Trade-offs of getting lock too soon, or using stale data for the update and getting opt lock exception
- In this case: read then lock and refresh

```
Account acct = em.find(Account.class, acctId);
// lock and refresh
em.refresh(acct, PESSIMISTIC);
double balance = acct.getBalance();
acct.setBalance(balance - charge);
```

- The "right" approach will depend on the requirements and expectations of the application



API Additions

- Additional API provides more options for vendor support and more flexibility for the user
- EntityManager:
  - LockMode parameter added to find, refresh
  - Properties parameter added to find, refresh, lock
  - Other useful additions
    - `void detach(Object entity)`
    - `<T> T unwrap(Class<T> cls)`
    - `getEntityManagerFactory()`

API Additions

- Tools need the ability to do introspection
- Additional APIs on EntityManager:
  - `Set<String> getSupportedProperties()`
  - `Map getProperties()`
  - `LockModeType getLockMode(Object entity)`
- Additional APIs on Query:
  - `int getFirstResult(), int getMaxResults`
  - `Map getHints()`
  - `Set<String> getSupportedHints()`
  - `FlushModeType getFlushMode()`
  - `Map getNamedParameters()`

Enhanced JP QL

- Timestamp literals

```
SELECT t from BankTransaction t
  WHERE t.txTime > '2008-06-01 10:00:01.0-09:00'
```

- Non-polymorphic queries

```
SELECT e FROM Employee e
  WHERE CLASS(e) = FullTimeEmployee
     OR e.wage = "SALARY"
```

- IN expression may include collection parameter

```
SELECT emp FROM Employee emp
  WHERE emp.project.id IN [:projectIds]
```

Enhanced JP QL

- Ordered List indexing

```
SELECT t FROM CreditCard c
  JOIN c.transactionHistory t
  WHERE INDEX(t) BETWEEN 0 AND 9
```

- CASE statement

```
UPDATE Employee e SET e.salary =
  CASE e.rating WHEN 1 THEN e.salary * 1.1
                WHEN 2 THEN e.salary * 1.05
  ELSE e.salary * 1.01
  END
```

Expressions and Criteria API

- Have had many requests for an object-oriented query API
- Most products already have them
- Dynamic query creation without having to do string manipulation
- Additional level of compile-time checking
- Equivalent JP QL functionality, with vendor extensibility
- Objects represent JP QL concepts, and are used as building blocks to build the query definition
- Natural Java API allows constructing and storing intermediate objects

Expressions and Criteria API

- **`QueryDefinition`**
  - Objectification of JP QL string
  - Constructed from a QueryBuilder factory
  - Housed inside Query object -- leverages Query API
  - Contains one or more "query roots" representing the domain type(s) being queried over
- **`DomainObject`**
  - Equivalent to an identification variable
  - Represents single instance of entity/embeddable type
  - Extends QueryDefinition for increased ease of use

Expressions and Criteria API

JP QL: `SELECT a FROM Account a`

```
Query q = em.createQuery(
                "SELECT a FROM Account a");
```

QueryDefinition:

```
QueryDefinition qdef = em.getQueryBuilder().
        createQueryDefinition(Account.class);
Query q = em.createQuery(qdef);
```

Expressions and Criteria API

JP QL:

```
SELECT a.id FROM Account a
            WHERE a.balance > 100
```

Expression:

```
queryBuilder qb = em.getQueryBuilder();
DomainObject acct = qb.
        createQueryDefinition(Account.class);
acct.select(acct.get("id"))
    .where(acct.get("balance")
                .greaterThan(100));
```

Expressions and Criteria API

Question:

Do you think the ease-of-use/complexity trade-off is worth it?

Alternative:

```
queryBuilder qb = em.getQueryBuilder();
QueryDefinition qdef =
                    qb.createQueryDefinition();
DomainObject acct = qdef.addRoot(Account.class);
qdef.select(acct.get("id"))
    .where(acct.get("balance")
                    .greaterThan(100));
```

Expressions and Criteria API

JP QL:

```
SELECT e
  FROM Employee e, Employee mgr
  WHERE e.manager = mgr AND mgr.level = "C"
```

Expression:

```
DomainObject emp = qb.
        createQueryDefinition(Employee.class);
DomainObject mgr = emp.addRoot(Employee.class);
emp.select(emp)
    .where(emp.get("manager").equal(mgr)
       .and(mgr.get("level").equal("C")));
```

Expressions and Criteria API

JP QL:

```
SELECT c.id, a
  FROM Account a JOIN a.customer c
  WHERE c.name = :custName
```

Expression:

```
DomainObject a = qb.
        createQueryDefinition(Account.class);
DomainObject c = a.join("customer");
a.select(c.get("id"), a)
  .where(c.get("name")
            .equals(a.param("custName")));
```

Strongly Typed API Instead?

- Each node in the expression is strongly typed with generics
- Compile-time safety of properties
- Provider generates a typed metamodel layer either at compile-time or statically
  - Account => Account_
  - Account "balance" property => Account_.balance
- Developer passes the metamodel types in as parameters to the expression API
- The rest is fairly similar

Strongly Typed API

JP QL:

```
SELECT a.id FROM Account a
            WHERE a.balance > 100
```

Expression:

```
QueryDefinition q = qb.createQueryDefinition();
Root<Account> a = q.addRoot(Account.class);
a.select(a.get(Account_.id))
 .where(qb.greaterThan(
     a.get(Account_.balance), 100));
```

Strongly Typed API

JP QL:

```
SELECT c.id, a
  FROM Account a JOIN a.customer c
  WHERE c.name = :custName
```

Expression:

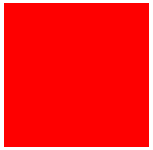
```
QueryDefinition q = qb.createQueryDefinition();
Root<Account> a = q.addRoot(Account.class);
Join<Account,Customer> c = acct.join(Account_.customer);
q.select(c.get(Customer_.id), a)
 .where(qb.equals(c.get(Customer_.name),
                  q.param("custName"))));
```

Questions

- Does the metamodel add too much confusion to the API?
- Is strong typing worth the cost of the extra metamodel generation and client usage?
- Is the metamodel generation going to be problematic?
  - What about when inside an IDE?
  - What about when metadata is in XML form?
- Will the typed API be able to support 3rd party tool layers and frameworks that do a great deal of dynamic querying?
- Should the typed API be a layer *beside*, or even *on top of* the untyped String property API?

Summary

- JPA 2.0 shipped as part of the Java EE 6 release (J1 09)
- JPA 2.0 Reference Implementation will be EclipseLink project (open source TopLink)
  - Shipped with WLS, Glassfish, Spring, or standalone
  - **http:/www.eclipse.org/eclipselink**
- Download JPA 2.0 Public draft and have a look
  - **http://www.jcp.org/en/jsr/detail?id=317**
- If you have any suggestions talk to an expert group member or send email to feedback alias:
  - **jsr-317-edr-feedback@sun.com**



Summary

- JPA 2.0 is introducing many of the things that were missing and that people asked for
- Have reached the 90-95% level
- JPA will never include *everything* that *everybody* wants
- There are now even fewer reasons to use a proprietary persistence API without JPA
- Just because a feature is there doesn't mean you have to use it!

Early Access

- You can access some of the new features as they are available and offer your input:

http://www.eclipse.org/eclipselink

- If you like the challenge of developing the internals you can also get involved in the project!