What’s New and Exciting in JPA 2.0

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

• Enterprise Java architect at Oracle
• Almost 20 years experience in server-side and persistence implementations
• Active on multiple JCP expert groups, including JPA 2.0 (JSR-317) and Java EE 6 (JSR-316)
• JPA book: **Pro EJB 3: Java Persistence API** (Apress)
• Next edition: **Pro JPA 2: Mastering the Java Persistence API**
• Contributor to other specifications (e.g. OSGi, SCA, etc.)
• Presenter at numerous conferences and events
What 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, EclipseLink, etc.) without JPA?
- How many people that are not using JPA right now are planning to use it later?
- How many people think JPA is just for losers that can’t do SQL statements nested 5 levels deep?
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 JPQL query language
• Support Java API based query language
• Integrate with new validation standards
More Standardized Properties

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

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

...
Persistence Unit Properties

Should simply be:

```xml
</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 favourite properties and which ones do YOU think should be standardized?
More Mappings

- Can use Join Tables more:
  - Unidirectional/ bidirectional one-to-one

```java
@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

```java
@Entity
public class Vehicle {
    ...
    @OneToMany
    List<Part> parts;
    ...
}
```
Additional Collection Support

- Collections of basic objects or embeddables

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

    @ElementCollection @Temporal(DATE)
    @Column(name="SRVC_DATE")
    @OrderBy("DESC")
    List<Date> serviceDates;
    ...
}
```
Additional Collection Support

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

```java
@Entity
public class Vehicle {
    ...
    @ManyToOne
    @JoinColumn(name="VEH_DEALERS")
    @OrderColumn(name="SALES_RANK")
    List<Dealer> preferredDealers;
    ...
}
```
More “Map” Flexibility

- Map keys and values can be:
  - Basic objects, embeddables, entities

```java
@Entity
public class Vehicle {

  ...

  @OneToMany
  @JoinTable(name="PARTSUPP",
              joinColumns=@JoinColumn(name="VEH_ID"),
              inverseJoinColumns=@JoinColumn(name="SUPP_ID"))
  @MapKeyJoinColumn(name="PART_ID")
  Map<Part,Supplier> suppliers;

  ...
}
```
Enhanced Embedded Support

- Embeddables can be nested
- Embeddables can have relationships

```java
@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; // Stored in metric

    @Access(PROPERTY) @Column(name="FUEL_EFF")
    protected double getDbFuelEfficiency() {
        return convertToImperial(fuelEfficiency);
    }
    protected void setDbFuelEfficiency(double fuelEff) {
        fuelEfficiency = convertToMetric(fuelEff);
    }
    ...
}
Derived Identifiers (the JPA 1.0 way)

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

```java
@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

```java
@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

```java
@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

```java
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_READ
  > PESSIMISTIC_WRITE
  > PESSIMISTIC_FORCE_INCREMENT

- Optimistic locking still supported in pessimistic mode
- Multiple places to specify lock (depends upon need)
Advanced Locking

Read, then lock and refresh when needed:

```java
public void applyCharges() {

    Account acct = em.find(Account.class, acctId);

    // calculate charges, etc.
    int charge = ... ;

    if (charge > 0) {
        em.refresh(acct, PESSIMISTIC_WRITE);
        double balance = acct.getBalance();
        acct.setBalance(balance - charge);
    }
}
```
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 JPQL**

**Timestamp literals**

```sql
SELECT t from BankTransaction t
WHERE t.txTime > {ts '2008-06-01 10:00:01.0'}
```

**Non-polymorphic queries**

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

**IN expression may include collection parameter**

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

Ordered List indexing

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

CASE statement

```sql
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
```
Criteria API

- Have had many requests for an object-oriented query API
- Most products already have one
- Dynamic query creation without having to do string manipulation
- Additional level of compile-time checking
- Equivalent JPQL functionality, with vendor extensibility
- Objects represent JPQL concepts, and are used as building blocks to build the query definition
- Natural Java API allows constructing and storing intermediate objects
- Fits into existing Query execution model interface
- Option to use string-based or strongly-typed query approach
String-based Approach

Advantages:

• Simpler to create, easier to read
• Don’t need to use generated metamodel classes in queries
• Use raw types of criteria interfaces
• Akin to writing JPQL queries, but with a Java API
• Better support for dynamic query construction and result processing

Disadvantages:

• Does not offer the compile-time type-checking
• Easier to make attribute name typos (like JPQL)
Strongly Typed Approach

Advantages:

• Each node in the expression is strongly typed with generics
• Result type is also bound
• Compile-time safety of attributes, selections, results, etc.
• Code completion of attributes

Disadvantages:

• More technically difficult to create and to read
• Metamodel needs to be auto-generated or manually created
  • Account => Account_, "balance" property => Account_.balance
• Harder to create dynamic queries because of type lock-in
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 more dynamic querying?

Solution: Allow both types
Criteria API

QueryBuilder

> Factory for CriteriaQuery objects
> Defines many of the query utility methods for comparing, creating literals, collection operations, subqueries, boolean, string, numeric functions, etc.

CriteriaQuery

> Objectification of JPQL string
> Housed inside Query object -- leverages Query API
> Contains one or more “query roots” representing the domain type(s) being queried over
> Set selection objects, "where" criteria, ordering, etc.
JPQL:

```
SELECT a FROM Account a
```

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

CriteriaQuery:

```
QueryBuilder qb = em.getQueryBuilderFactory();
CriteriaQuery cq = qb.create();
Root account = cq.from(Account.class);
cq.select(account);
Query q = em.createQuery(cq);
```
Criteria API

**JP QL:**

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

**CriteriaQuery:**

```java
Root acct = cq.from(Account.class);
acct.select( acct.get("id") )
    .where( qb.gt( acct.get("balance"), 100 ) );
```
Criteria API

JPQL:

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

CriteriaQuery:

```java
Root emp = cq.from(Employee.class);
Root mgr = cq.from(Employee.class);
cq.select(emp)
    .where(qb.conjunction(
        qb.equal( emp.get("manager"), mgr ),
        qb.equal( mgr.get("level"), "C" )
    ));
```
Criteria API

JPQL:

SELECT c.name, a
FROM Account a JOIN a.customer c
WHERE c.city = :custCity

CriteriaQuery:

Root acct = cq.from(Account.class);
Join cust = acct.join("customer");
acct.select(cust.get("name"), acct)
   .where(qb.equal(cust.get("city"),
         qb.parameter(String.class, "custCity")));

Strongly Typed API

**JP QL:**

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

**CriteriaQuery:**

```java
CriteriaQuery<Account> cq = qb.createQuery(Account.class);
Root<Account> acct = cq.from(Account.class);
acct.select( acct )
    .where( qb.gt( acct.get(Account_.balance), 100 ) );
```
Strongly Typed API

**JP QL:**

```sql
SELECT c.name, a
FROM Account a JOIN a.customer c
WHERE c.city = :city
```

**CriteriaQuery:**

```java
CriteriaQuery<Tuple> cq = qb.createTupleQuery(Tuple.class);
Root<Account> acct = cq.from(Account.class);
Join<Account,Customer> cust = acct.join(Account_.customer);
acct.select( qb.multiSelect( cust.get(Customer_.name),
                         acct ) )
    .where( qb.equal( cust.get(Customer_.city),
                     qb.parameter(String.class,"city") ) );
```
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!
To Find out More...

✓ JPA 2.0 being shipped as part of the Java EE 6 release (Sept 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](http://www.eclipse.org/eclipselink)

✓ Download JPA 2.0 Proposed Final Draft and have a look

✓ If you have any suggestions talk to an expert group member or send email to feedback alias:
  ✓ jsr-317-edr-feedback@sun.com
...or read the Book!