



Something About Me

- Enterprise Java architect at Oracle
- Over 20 years experience in server-side, distributed and persistence implementations
- Active member of many JCP expert groups
- Author, **Pro EJB 3: Java Persistence API** (Apress)
Pro JPA 2: Mastering the Java Persistence API
- Active member of OSGi Alliance Core Platform Expert Group (CPEG) and Enterprise Expert Group (EEG)
- Project Lead of **Eclipse Gemini** (Enterprise Modules)

ORACLE

Main Focus

- Standardize useful properties
- Add “specialized” ORM mappings
- Offer simple cache control abstraction
- Allow advanced locking settings
- Provide more hooks for vendor properties
- Add API for better tooling support
- Enhance JP QL
- Support Java-based “criteria” query language

ORACLE

Persistence Unit Properties

- Some properties are used by every provider

```
</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>
```

ORACLE

More Mappings

- Can use Join Tables more:

- Uni/bidirectional one-to-one

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



ORACLE

More Mappings

- Can also use a join table in a bidirectional one-to-many/many-to-one

Pop Quiz:

Does it matter which side you put @JoinTable on?

- In a many-to-many relationship it may be on either side
- In a bidirectional one-to-one relationship (see last slide) it can be on either side

ORACLE

More Mappings

Try putting it on the one-to-many side:

```
@Entity
public class Vehicle {
    ...
    @OneToMany
    List<Part> parts;
    ...
}
@Entity @JoinTable( ... )
public class Part {
    ...
    @ManyToOne
    Vehicle vehicle;
    ...
}
```

VEHIC_PARTS	
V_ID	P_ID

VEHICLE	
ID	...

PART	
ID	...

ORACLE

More Mappings

- Can use Join Tables less:
 - Unidirectional one-to-many with target foreign key

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

VEHICLE	
ID	...

PART	
PART_ID	V_ID

ORACLE

Additional Collection Types

- 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("DESC")
    List<Date> serviceDates;
    ...
}
```

VEHICLE	
ID	...

ASSEMBLY	
VEHICLE_ID	...

VEHICLE_SERVICEDATES	
VEHICLE_ID	SRVC_DATE

ORACLE

Ordered Lists (the 1.0 way)

- List order determined by an attribute in the entity
- Order is computed by the provider when objects are inserted into the list

```
@Entity
public class Vehicle {
    ...
    @ManyToMany
    @JoinTable(name="VEH_DEALERS")
    @OrderBy("sales")
    List<Dealer> dealersBySales;
    ...
}
```

VEH_DEALERS	
VEHICLE_ID	DEALER_ID

VEHICLE	
ID	...

DEALER		
ID	SALES	...

ORACLE

Lists with Persistent Ordering

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

```
@Entity
public class Vehicle {
    ...
    @ManyToOne
    @JoinTable (name="VEH_DEALERS")
    @OrderColumn (name="RANKING")
    List<Dealer> preferredDealers;
    ...
}
```

VEH_DEALERS		
VEHICLE_ID	DEALER_ID	RANKING

VEHICLE	
ID	...

DEALER	
ID	...

ORACLE

More "Map" Flexibility

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

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

PART_SUPP		
VEH_ID	SUPP_ID	PART_ID

VEHICLE	
ID	...

PART	
ID	...

SUPPLIER	
ID	...

ORACLE

Enhanced Embedded Support

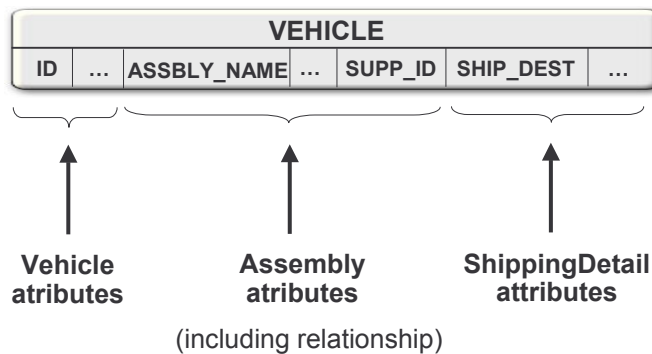
- Embeddables can be nested, and can have relationships

```
@Entity public class Vehicle {
    @Id int id;
    @Embedded Assembly assembly;
    ...
}

@Embeddable public class Assembly {
    @Column(name="ASSBLY_NAME")
    String name;
    @Embedded ShippingDetail shipDetails;
    @ManyToOne @JoinColumn(name="SUPP_ID")
    Supplier supplier;
    ...
}
```

ORACLE

Enhanced Embedded Support



ORACLE

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; // in metric

    @Access(PROPERTY) @Column(name="FUEL EFF")
    protected double getDbFuelEfficiency() {
        return convertToImperial(fuelEfficiency);
    }
    protected void setDbFuelEfficiency(double fuelEff) {
        fuelEfficiency = convertToMetric(fuelEff);
    }
    ...
}
```

ORACLE

Derived Identifiers (the 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
- Need an id class
 - Attributes must be of the same name and type as the attributes in the entity

ORACLE

Derived Identifiers (the 1.0 way)

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

    @Id int partNo;
    @Column(name="SUPP_ID")
    @Id int suppId;
    @ManyToOne
    @JoinColumn(name="SUPP_ID",
                insertable=false, updatable=false);
    Supplier supplier;
    ...
}
public class PartPK {
    int partNo;
    int suppId;
    ...
}
```

ORACLE

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;
    ...
}
```

ORACLE

Derived Identifiers

- Can use different identifier types

```
@Entity
public class Part {
    @EmbeddedId PartPK partPk;
    @ManyToOne @MapsId
    Supplier supplier;
    ...
}
@Embeddable
public class PartPK {
    int partNo;
    int supplier;
    ...
}
```

ORACLE

Shared Cache API

- API for operating on entity cache shared across all EntityManagers within a given persistence unit
 - > Accessible from EntityManagerFactory
- Supports only 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();
}
```

ORACLE

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)

ORACLE

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
 - Additional accessor methods, e.g. `getProperties()`
 - Other useful additions
 - `void detach(Object entity)`
 - `<T> T unwrap(Class<T> cls)`
 - `getEntityManagerFactory()`

ORACLE

Enhanced JP QL

- Timestamp literals

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

- Non-polymorphic queries

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

- IN expression may include collection parameter

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

ORACLE

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
```

ORACLE

Typed Query

- Can create a TypedQuery from a JP QL query string
- Subclass of Query, so all of the API still works
- Result type is bound to query type (no casting)

```
String jpql = "SELECT a FROM Account a " +  
             "WHERE a.balance > 1000";  
  
TypedQuery<Account> q =  
    em.createQuery(jpql, Account.class);  
  
List<Account> result = q.getResultList();
```

ORACLE

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 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
- Fits into existing Query execution model interface
- Option to use string-based or strongly-typed query approach

ORACLE

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
 - Similar to writing JP QL queries, but with a Java API
 - Better support for dynamic query construction and result processing
- Disadvantages:
 - Doesn't provide as much compile-time type-checking
 - Easier to make attribute name typos (like JP QL)

ORACLE

Strongly Typed Approach

- Advantages:
 - Each node in the expression is strongly typed with generics
 - Easier to bind result type
 - 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
 - Harder to create dynamic queries because of type lock-in

ORACLE

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

ORACLE

Criteria API

- **CriteriaBuilder**
 - 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 JP QL string
 - Housed inside a 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.

ORACLE

Criteria API

- JP QL:

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

- CriteriaQuery:

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery c = cb.createQuery();  
Root account = c.from(Account.class);  
c.select(account);  
Query q = em.createQuery(c);
```

ORACLE

Criteria API

- JP QL:

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

- CriteriaQuery:

```
CriteriaQuery<String> c =  
    cb.createQuery(String.class);  
Root<Account> acct = c.from(Account.class);  
c.select(acct.<String>get("id"))  
    .where(cb.gt(  
        acct.<Double>get("balance"), 100));
```

ORACLE

Criteria API

- JP QL:

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

- CriteriaQuery:

```
Root<Employee> emp = c.from(Employee.class);
Root<Employee> mgr = c.from(Employee.class);
c.select(emp)
  .where(cb.and(
    cb.equal(emp.get("manager"), mgr),
    cb.equal(mgr.get("level"), "C"))));
```

ORACLE

Criteria API

- JP QL:

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

- CriteriaQuery:

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

ORACLE

Strongly Typed API

- JP QL:

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

- CriteriaQuery:

```
CriteriaQuery<Account> c =
    cb.createQuery(Account.class);
Root<Account> acct = c.from(Account.class);
c.select(acct)
  .where(cb.gt(
    acct.get(Account_.balance), 100));
```

ORACLE

Strongly Typed API

- JP QL:

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

- CriteriaQuery:

```
CriteriaQuery<Tuple> c = cb.createTupleQuery();
Root<Account> acct = c.from(Account.class);
Join<Account, Customer> cust =
    acct.join(Account_.customer);
c.select(cb.tuple(cust.get(Customer_.name), acct))
  .where(cb.equal(
    cust.get(Customer_.city),
    cb.parameter(String.class, "city"))));
```

ORACLE

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!

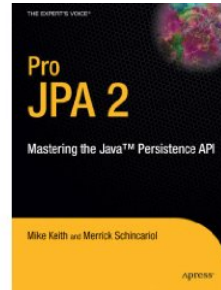
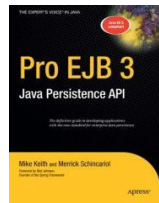
ORACLE

To Find out More...

- ✓ JPA 2.0 was shipped as part of Java EE 6 (Dec 2009)
- ✓ JPA 2.0 Reference Implementation - EclipseLink
 - ✓ Works with WLS, Glassfish, Spring, ..., or standalone
 - ✓ <http://www.eclipse.org/eclipselink>
- ✓ Download JPA 2.0 and have a look
 - ✓ <http://www.jcp.org/en/jsr/detail?id=317>
- ✓ Feedback alias for next release:
 - ✓ jsr-317-feedback@sun.com

ORACLE

...or read the Book!



ORACLE