Java Persistence 2.0

Linda DeMichiel
Java Persistence 2.0 Specification Lead

Java Persistence API: Brief History

Java Persistence 1.0

- Part of the EJB™ 3.0 simplification effort
- Standardized object/relational mapping
- Covered most of the essential features – a great start
- Available as part of the Java EE 5 Platform or standalone

Java Persistence 2.0

- More and better
- Greater portability for application developers
- Available as part of the Java EE 6 Platform or standalone
- Reference Implementation is EclipseLink; available as part of GlassFish v3

Java Persistence 2.0 New Features

Richer modeling capabilities

Expanded O/R mapping functionality

Additions to Java Persistence query language

Metamodel API

Criteria API

Pessimistic locking

Support for validation

Standardization of many configuration options

Expanded modeling and mapping

- Collections of basic types
- Improved support for embeddable classes
  - Collections of embeddables
  - Multiple levels of embedding
  - Embeddables with relationships
- Real (generalized) map support
  - Keys, values can be entities, embeddables, or basic types
  - Including support for ternary relationships
- Persistently ordered lists

Expanded modeling and mapping

- Orphan deletion
- Derived identities
  - Overlapping primary/foreign keys
- Combinations of access types
- Additional relationship mapping options
  - Unidirectional one-many foreign key mappings
  - One-one, many-one/one-many join table mappings

Collections of Basic Types

```
@Entity
public class Person {
  @Id protected String ssn;
  protected String name;
  protected Date birthDate;
  ...
  @ElementCollection
  protected Set<String> nickNames;
  ...
}
```

Collections of Basic Types

```
@Entity
public class Person {
  @Id protected String ssn;
  protected String name;
  protected Date birthDate;
  ...
  @ElementCollection
  @CollectionTable(name="ALIAS")
  protected Set<String> nickNames;
  ...
}
```

Collections of Embeddable Types

```
@Embeddable public class Address {
  String street;
  String city;
  String state;
  ...
}

@Entity public class RichGuy extends Person {
  ...
  @ElementCollection
  protected Set<Address> vacationHomes;
  ...
}
```

Multiple Levels of Embedding

```
@Embeddable public class ContactInfo {
  @Embedded Address address;
  ...
}

@Entity public class Employee {
  @Id int empId;
  String name;
  ContactInfo contactInfo;
  ...
}
```

Embeddables with Relationships

```
@Embeddable public class ContactInfo {
  @Embedded Address address;
  @OneToMany Set<Phone> phones;
  ...
}

@Entity public class Employee {
  @Id int empId;
  String name;
  ContactInfo contactInfo;
  ...
}
```

Ordered Lists

```
@Entity public class CreditCard {
   @Id long cardNumber;
   @OneToOne Person cardHolder;
   ...
   @OneToMany
   @OrderColumn
   List<CardTransaction> transactions;
   ...
}
```

Maps

```
@Entity public class VideoStore {
  @Id Integer storeId;
  Address location;
  ...
  @ElementCollection
  Map<Movie, Integer> inventory;
  ...
}

@Entity public class Movie {
  @Id String title;
  String director;
  ...
}
```

Automatic Orphan Deletion

For entities logically “owned” by “parent”

```
@Entity public class Order {
    @Id int orderId;
    ...
    @OneToMany(cascade=PERSIST, orphanRemoval=true )
    Set<Item> lineItems;
    ...
}
```

Java Persistence Query Language

Support for all new modeling and mapping features

Operators and functions in select list

Case, coalesce, nullif expressions

Restricted polymorphism

Collection-valued parameters for IN-expressions

JPQL New Operators

INDEX

- For ordered Lists

KEY, VALUE, ENTRY

- For maps

CASE, COALESCE, NULLIF

- For case expressions, etc.

TYPE

- For restricted polymorphism

Ordered Lists

```
SELECT t
FROM CreditCard c JOIN c.transactions t
WHERE c.cardHolder.name = 'John Doe'
  AND INDEX(t) < 10
```

Maps

```
// Inventory is Map<Movie,Integer>

SELECT v.location.street, KEY(i).title, VALUE(i)
FROM VideoStore v JOIN v.inventory i
WHERE KEY(i).director LIKE '%Hitchcock%'
  AND VALUE(i) > 0
```

Case Expressions

```
UPDATE Employee e
SET e.salary =
  CASE e.rating
     WHEN 1 THEN e.salary * 1.05
     WHEN 2 THEN e.salary * 1.02
     ELSE e.salary * .95
  END
```

Restricted Polymorphism

```
SELECT e
FROM Employee e
WHERE TYPE(e) IN :empTypes
```

Criteria API

- Object-based API for building queries
- Designed to mirror JPQL semantics
- Strongly typed
  - Based on type-safe metamodel of persistence unit
  - Heavy use of Java generics
- Supports object-based or string-based navigation

Criteria API: Core Interfaces

CriteriaBuilder

- Used to construct criteria queries, selections, predicates, orderings

CriteriaQuery

- Used to add / replace / browse query elements
- from, select, where, orderBy, groupBy, having, ... methods

Root

- Query roots

Join, ListJoin, MapJoin, ...

- Joins from a root or existing join

Path

- Navigation from a root, join, or path

Subquery

How to Build a Criteria Query

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<ResultType> cquery =
   cb.createQuery(ResultType.class);

Root<MyEntity> e = cquery.from(MyEntity.class);
Join<MyEntity, RelatedEntity> j = e.join(...);
...
cquery.select(...)
      .where(...)
      .orderBy(...)
      .groupBy(...);
TypedQuery<ResultType> tq = em.createQuery(cquery);
List<ResultType> result = tq.getResultList();
...
```

Criteria Query

```
SELECT DISTINCT c
FROM Customer c JOIN c.orders o
WHERE o.product.productType = 'printer'

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cquery =
    cb.createQuery(Customer.class);
Root<Customer> c = cquery.from(Customer.class);
Join<Customer, Order> o = c.join("orders");
cquery.where(
          cb.equal(o.get("product")
                    .get("productType"),
          "printer")
          )
      .select(c).distinct(true);
```

Metamodel

- Abstract, “schema-level” view of managed classes
  - Entities, mapped superclasses, embeddables
- Accessed dynamically
  - EntityManagerFactory.getMetamodel()
  - EntityManager.getMetamodel()
- And/or materialized as static metamodel clases
  - Used to create strongly-typed criteria queries
  - Spec defines canonical format

Entity Class

```
package com.example;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Embedded;
import javax.persistence.OneToMany;
import java.util.Set;

@Entity
public class Customer {
  @Id int custId;
  @Embedded Address address;
  @OneToMany Set<Order> orders;
  ...
}
```

Metamodel Class

```
package com.example;

import javax.annotation.Generated;
import javax.persistence.metamodel.SetAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@Generated("EclipseLink JPA 2.0 Canonical Model Generation")
@StaticMetamodel(Customer.class)
public class Customer_ {
  public static volatile SingularAttribute<Customer,Integer> custId;
  public static volatile SingularAttribute<Customer,Address> address;
  public static volatile SetAttribute<Customer,Order> orders;
  ...
}
```

Criteria Query

```
SELECT DISTINCT c
FROM Customer c JOIN c.orders o
WHERE o.product.productType = 'printer'

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cquery =
    cb.createQuery(Customer.class);
Root<Customer> c = cquery.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
cquery.where(
          cb.equal(o.get(Order_.product)
                   .get(Product_.productType),
          "printer")
          )
      .select(c).distinct(true);
```

Pessimistic Locking

- Grab database locks upfront
- Lock Modes
  - PESSIMISTIC_READ - grab shared lock
  - PESSIMISTIC_WRITE - grab exclusive lock
  - PESSIMISTIC_FORCE_INCREMENT – update version
- Normal (default) pessimistic locking
  - Persistent state of entity (except element collections)
  - Relationships where entity holds foreign key
- Extended pessimistic locking
  - Element collections and relationships in join talbes

Locking APIs

EntityManager methods: lock, find, refresh

Query and TypedQuery methods: setLockMode, setHint

NamedQuery annotation: lockMode element

javax.persistence.lock.scope property

javax.persistence.lock.timeout hint

PessimisticLockException (if transaction rolls back)

LockTimeoutException (if only statement rolls back)

Validation

Leverages work of Bean Validation (JSR 303)

Automatic validation upon lifecycle events

- PrePersist
- PreUpdate
- PreRemove

validation-mode element in persistence.xml

- AUTO
- CALLBACK
- NONE

Validation

```
@Entity public class Employee {
  @Id Integer empId;
  String name;
  @Max(15) Integer vacationDays;
  @Valid Address worksite;
  ...
}

@Embeddable public class Address {
  @Size(max=30) String street;
  @Size(max=20) String city;
  @Zipcode String zipcode;
  ...
}
```

Summary of New Features

More flexible modeling capabilities

Expanded O/R mapping functionality

Additions to Java Persistence query language

Metamodel API

Criteria API

Pessimistic locking

Support for validation

Standardization of many configuration options

Resources

Java EE 6 and GlassFish v3

Java EE 6 Home

http://java.sun.com/javaee

Java EE 6 Downloads

http://java.sun.com/javaee/downloads/index.jsp

Upcoming Training

http://java.sun.com/javaee/support/training/

Sun GlassFish Enterprise Server v3 Home

http://www.sun.com/glassfishv3

Community Page

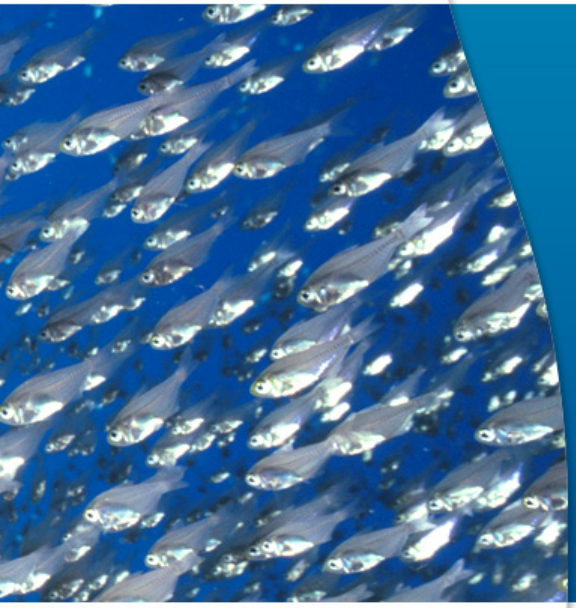
glassfish.org

White Papers/Webinars

http://www.sun.com/glassfish/resources

Java EE 6

GlassFish



Thank You