Java Persistence API

Mike Keith, Oracle Corporation
Linda DeMichiel, Sun Microsystems
Goal of This Talk

Learn more about the Java Persistence API
Agenda

Entities
Persistence Units
Persistence Contexts
Entity Manager API
Queries
Object/Relational Mapping
Persistence in Java SE
Primary Features

- Simple to use and intuitive to learn
- POJO development model
- Object-oriented - inheritance, polymorphism, etc.
- Standardized metadata for object-relational mapping expressed in annotations, XML or combination of the two
- Entity detachment to other tiers and JVMs
- Java Persistence Query Language uses entity schema for querying across entities in the database
- Java SE persistence model for testing and client apps
Entities

- Persistent objects
  - Entities, not entity beans
  - Java objects, not “components”
  - Serializable, detachable and mergeable
  - Indicated by @Entity

- Persistent state
  - Defined by persistent fields or properties
  - Must minimally include persistent identity
  - Entities may also have non-persistent state
Entities

- Persistent identity
  - Uniquely identifies entity in database (primary key)

1. Simple field/property

   ```
   @Id int custId;
   ```

2. Single field/property to store an instance of a composite PK class

   ```
   @EmbeddedId CustPK id;
   ```

3. Multiple fields/properties that are all present in a composite PK class (compatible with EJB 2.x)

   ```
   @IdClass(CustPK.class)
   ```
Example

@Entity
public class Customer {
    @Id private Long id;
    private String name;

    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
Persistence Unit

- Set of entities and related classes that share the same configuration
- Convenient packaging and deployment unit
- Runtime configuration defined in persistence.xml
- Can reference additional classes on classpath or additional jar
- One or more O/R mapping files
- Scoping boundary for queries and id generators
Example

```xml
<persistence>
  <persistence-unit name="OrderMgmt">
    <provider>com.acme.PersistenceProvider</provider>
    <jta-data-source>jdbc/MyOrderDB</jta-data-source>
    <mapping-file>order-mappings.xml</mapping-file>
    <jar-file>myparts.jar</jar-file>
    <properties>
      <property
        name="com.acme.persistence.logSQL"
        value="ALL"/>
    </properties>
  </persistence-unit>
</persistence>
```
Persistence Context

- Set of “managed” entity instances at runtime
  - Unique entity instance for any given persistent identity
- Maintained and operated on via EntityManager API
- Normally scoped to a container transaction
- May also be extended to keep entities managed across multiple sequential transactions
  - Stateful session beans
  - Application-managed EntityManagers
- Propagated across EJB components within JTA transactions
Persistence Context Propagation

```java
@Stateless public class Bean1Class implements Bean1 {
    @PersistenceContext EntityManager em1;
    @EJB Bean2 bean2;
    public void bean1Method() {
        em1.persist(new Customer(100));
        bean2.bean2Method();
    }
}

@Stateless public class Bean2Class implements Bean2 {
    @PersistenceContext EntityManager em2;
    public void bean2Method() {
        Customer cust = em2.find(Customer.class, 100);
        cust.setName("");
    }
}
```
Entity Manager API

- Used by application to operate on entities
  - `persist()` - Insert the entity into the database
  - `remove()` - Delete the entity from the database
  - `refresh()` - Reload the entity state from the database
  - `merge()` - Synchronize state of detached entity with the pc
  - `find()` - Execute a simple PK query
  - `createQuery()` - Create dynamic query using EJB QL
  - `createNamedQuery()` - Create a predefined query
  - `createNativeQuery()` - Create SQL query
  - `contains()` - Determine if entity is managed by pc
  - `flush()` - Explicitly synchronize pc to database
Persist Operation

- Insert a new instance of the entity into the database
- Save the persistent state of the entity and any owned relationship references
- The entity instance becomes “managed” in the pc
- Persist operation optionally cascades to related objects

```java
public Customer createCustomer(int id, String name) {
    Customer cust = new Customer(id, name);
    entityManager.persist(cust);
    return cust;
}
```
Find and Remove Operations

- **Find**
  - Obtain a managed entity instance with a given persistent identity
  - Return null if not found

- **Remove**
  - Delete entity with the given persistent identity from the database
  - Optionally cascades to related objects

```java
public void removeCustomer(Long custId) {
    Customer cust =
    entityManager.find(Customer.class, custId);
    entityManager.remove(cust);
}
```
Merge Operation

- State of detached entity gets merged into a managed copy of the detached entity
- Managed entity that is returned has a different Java identity from the detached entity
  - May be an entity instance that was previously in the pc
- Merge operation optionally cascades to related objects

```java
public Customer storeUpdatedCustomer(Customer cust) {
    return entityManager.merge(cust);
}
```
Entity Lifecycle Callbacks

- Entity Listeners may be defined to receive notification of 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
Queries

- EntityManager acts as a factory for Query objects
- Uses Java Persistence Query Language
- Native queries allow native SQL customization
- Can use positional or named parameters
- Can use static queries or dynamic queries
  - Dynamic query string specified at query creation time
  - Static queries are defined at development time as annotations or in XML
- Control over query execution, parameter binding and pagination
- Queries can return entities, non-entities, or projections of entity data
Dynamic Query

```java
public List findAll(String entityName){
    return entityManager.createQuery(
        "select e from " + entityName + " e"
    ).getResultList();
}
```

- Return all instances of the given entity type
- Query string composed from entity type
  For example, if “Customer” was passed in then query string would be: “select e from Customer e”
Static Query

@NamedQuery(name="findByCustId", 
    query="select o from Order o 
    where o.customer.id = :custId 
    order by o.createdDate")

public List findOrdersByCustomer(Customer cust) {
    return entityManager.createNamedQuery("findByCustId")
        .setParameter("custId", cust.getId())
        .getResultList();
}

• Return orders for a given customer
• Use a named parameter to pass in customer id
Static Query

```xml
<entity-mappings>

... 

<entity class="com.acme.Order">

  <named-query name="findByCustId">

    <query>select o from Order o

    where o.customer.id = :custId

    order by o.createdDate

    </query>

  </named-query>

</entity>

</entity-mappings>
```
Query Language Enhancements

- Support for joins
  - select o from Order o
    left join o.lineItems li
    where li.amount > 100

- Support for subqueries
  - select distinct o from Order o
    where exists
      (select li from o.lineItems li
       where li.amount > 100)

- Support for aggregation
  - select o.id, sum(li.amount) from Order o
    join o.lineItems li group by o.id
Query Language Enhancements

- **Data projections**
  - `select o.item.name, o.quantity from Order o
   where o.quantity > 100`

- **Additional functions**
  - `trim()`, `locate()`, `concat()`, `substring()`, `lower()`,
    `upper()`, `length()`, `abs()`, `sqrt()`, `mod()`, `size()`

- **Update and delete operations**
  - `update OrderLine ol set ol.fulfilled = 'Y'
    where ol.order.id = 9876543`
  - `delete from Customer cust where cust.id = 12345`
Object/Relational Mapping

- Logical and physical mapping views
  - Logical—object model (e.g. @OneToMany, @Id, @Transient)
  - Physical—DB tables and columns (e.g. @Table, @Column)
- Support for basic, serialized objects, LOBs, enums, etc.
- Access to object state using fields or properties
- Single-valued and collection-valued relationship mappings
- Multiple tables, composite relationship keys
- Rules for defaulting of database table and column names
- Specified as annotations or XML
Object/Relational Mapping

- Can specify EAGER or LAZY loading of fields or relationships
  - Fetch mode LAZY is a hint to the Container to defer loading until the field or property is accessed
  - Fetch mode EAGER requires that the field or relationship be loaded eagerly

- Cascading of entity operations to related entities
  - Can cascade PERSIST, REMOVE, MERGE, REFRESH or ALL
  - Setting may be defined per relationship
  - Configurable globally in orm.xml for persistence-by-reachability
Simple Mappings

- Direct mappings of fields/properties to columns
  - @Basic - optional annotation to indicate simple mapped attribute

- Maps any of the common simple Java types
  - Primitives, wrapper types, Date, Serializable, byte[], ...

- Used in conjunction with @Column

- Defaults to the type deemed most appropriate if no mapping annotation is present

- Can override any of the defaults
Simple Mappings

```java
@Entity
public class Customer {
    @Id
    int id;

    String name;

    int c_rating;

    @Lob
    Image photo;
}
```
Simple Mappings

```java
@Entity
public class Customer {
    @Id
    int id;
    @Column(length=50)
    String name;
    @Column(name="CREDIT")
    int c_rating;
    @Lob
    Image photo;
}
```
Simple Mappings

```xml
<entity class="com.acme.Customer">
  <attributes>
    <id name="id"/>
    <basic name="name">
      <column length="50"/>
    </basic>
    <basic name="c_rating">
      <column name="CREDIT"/>
    </basic>
    <basic name="photo">
      <lob/>
    </basic>
  </attributes>
</entity>
```
Relationship Mappings

- Common relationship mappings supported
  - `@ManyToOne, @OneToOne` - single entity
  - `@OneToMany, @ManyToMany` - collection of entities
- Unidirectional or bidirectional
- Owning and inverse sides of every bidirectional relationship
- Owning side specifies the physical mapping
  - `@JoinColumn` to specify foreign key column
  - `@JoinTable` decouples physical relationship mappings from entity tables
ManyToOne Mapping

```java
@Entity
public class Customer {
    @Id
    int id;
    @ManyToMany
    Address addr;
}
```
OneToMany Mapping

```java
@Entity
public class Customer {
    @Id
    int id;
    ...
    @OneToMany(mappedBy="cust")
    Set<Order> orders;
}
```

```java
@Entity
public class Order {
    @Id
    int id;
    ...
    @ManyToMany
    Customer cust;
}
```
ManyToMany Mapping

```java
@Entity
public class Customer {
    @Id
    int id;
    ...
    @ManyToMany
    Collection<Phone> phones;
}
```

```java
@Entity
public class Phone {
    @Id
    int id;
    ...
    @ManyToMany(mappedBy="phones")
    Collection<Customer> custs;
}
```
ManyToMany Mapping

@Entity
public class Customer {
    ...
    @ManyToMany
    @JoinTable(table="CUST_PHONE"),
    joinColumns=@JoinColumn(name="CUST_ID"),
    inverseJoinColumns=@JoinColumn(name="PHON_ID")
    Collection<Phone> phones;
}
Mapping of Embedded Objects

@Embeddable
public class CustomerInfo {
  String name;
  int credit;
  Image photo;
}

@Embeddable
public class CustomerInfo {
  String name;
  int credit;
  Image photo;
}
Inheritance

- Entities can extend
  - Other entities — concrete or abstract
  - Non-entity classes — concrete or abstract

- Map inheritance hierarchies in three ways
  1. SINGLE_TABLE — all classes stored in the same table
  2. JOINED — Each class (concrete or abstract) stored in a separate table
  3. TABLE_PER_CLASS — Each concrete class stored in separate table (optional)
Object Model

@Entity public abstract class Animal {
    @Id int id;
    String name;
}

@Entity public class LandAnimal extends Animal {
    int legCount;
}

@Entity public class AirAnimal extends Animal {
    short wingSpan;
}
Data Models

Single table:

<table>
<thead>
<tr>
<th>ANIMAL</th>
<th>ID</th>
<th>DISC</th>
<th>NAME</th>
<th>LEG_COUNT</th>
<th>WING_SPAN</th>
</tr>
</thead>
</table>

Joined:

<table>
<thead>
<tr>
<th>ANIMAL</th>
<th>ID</th>
<th>NAME</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>LAND_ANIMAL</th>
<th>ID</th>
<th>LEG_COUNT</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>AIR_ANIMAL</th>
<th>ID</th>
<th>WING_SPAN</th>
</tr>
</thead>
</table>

Table per Class:

<table>
<thead>
<tr>
<th>LAND_ANIMAL</th>
<th>ID</th>
<th>NAME</th>
<th>LEG_COUNT</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>AIR_ANIMAL</th>
<th>ID</th>
<th>NAME</th>
<th>WING_SPAN</th>
</tr>
</thead>
</table>
Persistence in Java SE

- No deployment phase
  - Application must use a “Bootstrap API” to obtain an EntityManagerFactory

- Typically use resource-local EntityManagers
  - Application uses a local EntityTransaction obtained from the EntityManager

- New persistence context for each and every EntityManager that is created
  - No propagation of persistence contexts
Entity Transactions

- Resource-level transaction akin to a JDBC transaction
  - Isolated from transactions in other EntityManagers
- Transaction demarcation under explicit application control using EntityTransaction API
  - `begin()`, `commit()`, `setRollbackOnly()`, `rollback()`, `isActive()`
- Underlying (JDBC) resources allocated by EntityManager as required
Bootstrap Classes

`javax.persistence.Persistence`
- Root class for bootstrapping an EntityManager
- Locates a provider service for a named persistence unit
- Invokes on the provider to obtain an EntityManagerFactory

`javax.persistence.EntityManagerFactory`
- Creates EntityManagers for a named persistence unit or configuration
Example

```java
public class SalaryChanger {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("HRSystem");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Employee emp = em.find(
            Employee.class, new Integer(args[0]));
        emp.setSalary(new Integer(args[1]));
        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```
Summary

- Entities are simple Java classes
  - Easy to develop and intuitive to use
  - Can be moved to other server and client tiers
- EntityManager
  - Simple and elegant API for operating on entities
  - Supports use inside and outside Java EE containers
- Standardization
  - O/R mapping using annotations or XML
  - Named and dynamic query definition using Java Persistence QL
  - SPI for pluggable Persistence Providers
Links

- Java Persistence API Reference Implementation (RI)
  - Oracle TopLink Essentials
  - Part of Sun Glassfish open source project

  [http://glassfish.dev.java.net/](http://glassfish.dev.java.net/)

- EJB 3.0 Proposed Final Draft


Final draft to be released with Java EE 5
Questions