Getting to Know JPA
The New Enterprise Persistence Standard

Mike Keith
michael.keith@oracle.com

http://www.oracle.com/technology/ejb3
About Me

- Co-spec Lead of EJB 3.0 (JSR 220)
- Java EE 5 (JSR 244) expert group member
- Co-author “Pro EJB 3: Java Persistence API”
- Persistence/EJB/Container Architect for Oracle
- 15+ years experience in distributed, server-side and persistence implementations
- Presenter at numerous conferences and events
About You

- How many people have already used EJB 3.0 Java Persistence API (JPA)?
- How many people are using proprietary persistence APIs?
- How many people are happy with a proprietary persistence API?
Learn the basic features of the EJB 3.0 Java Persistence API and how it can be used to attain portable persistence within enterprise applications.
About JPA

- Persistence API for operating on POJO entities
- Merger of expertise from TopLink, Hibernate, JDO, EJB vendors and individuals
- Created as part of EJB 3.0 within JSR 220
- Released May 2006 as part of Java EE 5
- Integration with Java EE web and EJB containers provides enterprise “ease of use” features
- “Bootstrap API” can also be used in Java SE
- Pluggable Container-Provider SPI
Reference Implementation

- Part of “Glassfish” project on java.net
  - RI for entire Java EE platform
- Sun and Oracle partnership
  - Sun Application Server + Oracle persistence
- JPA part called “TopLink Essentials”
  - Derived from and donated by Oracle TopLink
- All Open Source (under CDDL license)
  - Anyone can download/use source code or binary code in development or production
Anatomy of an Entity

- Abstract or concrete top level Java class
  - Non-final fields/properties, no-arg constructor
- No required interfaces
  - No required business or callback interfaces (but you may use them if you want to)
- Direct field or property-based access
  - Getter/setter can contain logic (e.g. for validation)
- May be Serializable, but not required
  - Only needed if passed by value (in a remote call)
The Minimal Entity

- Must be indicated as an Entity
  1. @Entity annotation on the class

```java
@Entity
public class Employee { … }
```

  2. Entity entry in XML mapping file

```
<entity class="com.acme.Employee"/>
```
The Minimal Entity

- Must have a persistent identifier (primary key)

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

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
}
```
Persistent Identity

- Identifier (id) in entity, primary key in database
- Uniquely identifies entity in memory and in db

1. Simple id – single field/property
   ```
   @Id int id;
   ```

2. Compound id – multiple fields/properties
   ```
   @Id int id;
   @Id String name;
   ```

3. Embedded id – single field of PK class type
   ```
   @EmbeddedId EmployeePK id;
   ```
Persistence Context

- Abstraction representing a set of “managed” entity instances
  - Entities keyed by their persistent identity
  - Java object identity also important
  - All entities in the PC are managed

- Controlled and managed by EntityManager
  - Contents of PC change as a result of operations on EntityManager API
  - PC may be managed by multiple EntityManagers (but not concurrently) due to PC propagation
Persistence Context

Application

EntityManager

MyEntity a
MyEntity b

Persistence Context

MyEntity A
MyEntity B
MyEntity C

Entities

Entity state
Persistence Context Scope

- **Transaction scope**
  - **Transaction-scoped persistence context** is the usual (default) case
  - Only available within the managed environment
  - All entities read in/operated on during the course of a JTA transaction become managed (part of the PC)
  - When transaction completes the entities are cut loose or “detached”
  - PC is automatically propagated to other components participating in the same JTA transaction
Persistence Context Scope

- Extended Scope
  - An extended persistence context keeps entities managed across multiple (sequential) transactions
  - Explicit support for longer-running sequences of operations when client is co-located on the server
  - Is like a cache, except that additionally the entity references remain valid across transactions
  - Will grow indiscriminately until the persistence context is closed, or it can be explicitly cleared
  - Can be inherited by child extended persistence contexts
EntityManager

- Client-visible artifact for operating on entities
  - API for all the basic persistence operations
- Can think of it as a proxy to a persistence context
  - May access multiple different persistence contexts throughout its lifetime
- Multi-dimensionality leads to different aspects of EntityManager (and persistence context) naming
  - Transaction type
  - Life cycle
EntityManager and Transactions

- Transaction type for an EntityManager is defined statically and may be one of two types
  - **JTA EntityManager**
    - Associated with current thread-bound JTA transaction
  - **Resource-local EntityManager**
    - Only associated with a local JDBC transaction
    - Independent from any JTA transaction
- Only JTA EntityManagers may be of the transaction variety
EntityManager Life Cycle

- Life cycle involves obtaining and disposing of PC
- Container-managed EntityManager
  - Container obtains and closes persistence context at the appropriate time
  - Must be a JTA EntityManager to support propagation
  - Transaction-scoped persistence context propagated across multiple transaction-sharing EJB components
  - Extended persistence context propagated from SFSB to SLSB and locally created SFSB
  - Client obtains it through injection or JNDI lookup
Container-managed EM

- Using injection:

  ```java
  @PersistenceContext
  EntityManager myEM;
  ```

- Using JNDI lookup:

  ```java
  @Stateless
  @PersistenceContext(name="jndiEM")
  public class MyBean implements MyInterface {
    @Resource SessionContext ctx;
    public void doNothing() {
      EntityManager em = (EntityManager) ctx.lookup("jndiEM");
  ```
Application-managed EM

- Application manually obtains and closes a persistence context whenever it wants.
- Application uses EntityManagerFactory call to obtain an EntityManager with a new “stand-alone” persistence context.
- Persistence context is never propagated.
- Application-managed API can be used both in Java EE as well as in Java SE (the only difference is how the EMF is obtained).
Getting an EntityManagerFactory

- Using injection:

  ```java
  @PersistenceUnit
  EntityManagerFactory myEMF;
  ```

- Using JNDI lookup:

  ```java
  @Stateless
  @PersistenceUnit(name="jndiEMF")
  public class MyBean implements MyInterface {
      @Resource SessionContext ctx;
      public void doNothing() {
          EntityManagerFactory emf =
              (EntityManagerFactory) ctx.lookup("jndiEMF");
      }
  }
  ```
Application-Managed EM

@Stateless
class MyBean implements MyInterface {
    @PersistenceUnit
    EntityManagerFactory myEMF;

    public void doNothing() {
        EntityManager em = emf.createEntityManager();
        ...
        em.close();
    }
}
Entity Lifecycle

- **New**
  - Entity has been created in memory *(e.g. using a Java constructor)* but has not yet been persisted
  - Is not in the persistence context

- **Managed**
  - Entity has been persisted during this transaction or at some point in the past
  - Is in the persistence context
Entity Lifecycle

- Removed
  - Entity has been removed in the current transaction
  - Is in the persistence context

- Detached
  - Entity was persisted at some point in the past and exists in the database
  - Is not in the persistence context
Operations on Entities

- **EntityManager API**
  - `persist()` - Insert the identity of an entity into the db
  - `remove()` - Delete the persistent identity of the entity from the db
  - `refresh()` - Reload the entity state from the db
  - `merge()` - Synchronize the state of detached entity with the pc
  - `find()` - Execute a simple PK query
  - `createQuery()` - Create query instance using dynamic EJB QL
  - `createNamedQuery()` - Create instance for a predefined query
  - `createNativeQuery()` - Create instance for an SQL query
  - `contains()` - Determine if entity is managed by pc
  - `flush()` - Force synchronization of pc to database
**persist()**

- Insert a new entity instance into the database
- Save the persistent state of the entity and any owned relationship references
- Entity instance becomes managed
- Persist optionally cascades to related entities

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

- **find()**
  - Obtain a managed entity instance with a given persistent identity – return null if not found

- **remove()**
  - Delete a managed 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()

- 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);
}
```
flush()

- Flush all changes to entities in the persistence context to the database
- Inserts and deletes also occur

```java
public Customer updateCustomerName(
    int id, String name){
    Customer cust =
        entityManager.find(Customer.class, custId);
    cust.setname(name);
    entityManager.flush();
    return cust;
}
```
refresh()

- Refresh a managed entity in the persistence context with the contents of the database
- Any dirty changes to the entity will be lost

```java
public Customer refreshCustomer(int id){
    // find managed customer in pc
    Customer cust =
        entityManager.find(Customer.class, custId);
    entityManager.refresh(cust);
    return cust;
}
```
Entity Lifecycle Callbacks

- Life cycle events trigger callbacks that may be defined on the entity or on an entity listener class.
- Callback methods are indicated using annotations or XML elements in mapping descriptor.
- May have any name, but must have a specified signature.
- On entity class: `void <method>()`
- On listener class: `void <method>(Object)`
  - Can list a specific entity param type instead of Object.
Entity Lifecycle Callbacks

- Lifecycle events include:
  - `@PrePersist` — when the application calls `persist()`
  - `@PostPersist` — after the SQL `INSERT`
  - `@PreRemove` — when the application calls `remove()`
  - `@PostRemove` — after the SQL `DELETE`
  - `@PreUpdate` — when container detects instance is dirty
  - `@PostUpdate` — after the SQL `UPDATE`
  - `@PostLoad` — after an instance was loaded

- “Post” and PreUpdate callbacks may occur as part of the operation or at transaction commit time
Entity Callbacks

```java
@Entity
@EntityListener(Validator.class)
public class Account {
    @Id Long accountId;
    Integer balance;

    public Integer getAccountId() { ... }
    public Integer getBalance() { ... }
    public void deposit(Integer amount) { ... }
    public Integer withdraw(Integer amount) { ... }
}
```
public class Validator {
    @PrePersist
    protected void validateCreate(Account acct) {
        if (acct.getBalance() < MINIMUM_BALANCE)
            throw new AccountException("Insufficient balance to open an account");
    }

    @PreUpdate
    public void validateBalance(Account acct) {
        if (acct.getBalance() < 0)
            throw new NSFException("Insufficient funds");
    }
}
Culture Break – Maple Trees

- Colour of the leaves are partly due to genetic disposition of the tree.

- Over 200 different maple varieties

- Maple syrup made from “sugar maple”
Queries

- Dynamic or statically defined (named queries)
- Criteria using JP QL (extension of EJB QL)
- Native SQL support (when required)
- Named parameters bound at execution time
- Pagination and ability to restrict size of result
- Single/multiple-entity results, data projections
- Bulk update and delete operation on an entity
- Standard hooks for vendor-specific hints
 Queries

- Query instances obtained from factory methods on EntityManager
- Query API:
  - `getResultList()` – execute query returning multiple results
  - `getSingleResult()` – execute query returning single result
  - `executeUpdate()` – execute bulk update or delete
  - `setFirstResult()` – set the first result to retrieve
  - `setMaxResults()` – set the maximum number of results to retrieve
  - `setParameter()` – bind a value to a named or positional parameter
  - `setHint()` – apply a vendor-specific hint to the query
  - `setFlushMode()` – apply a flush mode to the query when it gets run
Dynamic Queries

- Use `createQuery()` factory method at runtime and pass in the JP QL query string
- Use correct execution method
  - `getResultList()`, `getSingleResult()`, `executeUpdate()`
- Query may be compiled/checked at creation time or when executed
- Maximal flexibility for query definition and execution
### Dynamic Queries

```java
public List findAll(String entityName) {
    return entityManager.createQuery(
        "select e from " + entityName + " e"
    ).setMaxResults(100)
    .getResultList();
}
```

- Return all instances of the given entity type
- JP QL string composed from entity type. For example, if “Account” was passed in then JP QL string would be: “select e from Account e”
Named Queries

- Use `createNamedQuery()` factory method at runtime and pass in the query name
- Query must have already been statically defined either in an annotation or XML
- Query names are “globally” scoped (for some definition of global that we will get to later)
- Provider has opportunity to precompile the queries and return errors at deployment time
- Can include parameters and hints in static query definition
Named Queries

```java
@NamedQuery(name="Sale.findByCustId",
    query="select s from Sale s
            where s.customer.id = :custId
            order by s.salesDate")

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

- Return all sales for a given customer
- Use a named parameter to specify customer id
Native Queries

- Can use SQL in either dynamic or named queries
  - Provider will map the result set to entities
- At runtime use createNativeQuery() and createNamedQuery() for dynamic/static creation
- Common case for query to map to a single class
  - createNativeQuery(String sql, Class cls)
- For update/delete statements use simpler form
  - createNativeQuery(String sql)
- For other cases use result set mappings
  - createNativeQuery(String sql, SqlResultSetMapping m)
Native Queries

```java
public List getItemsOfBigSales() {
    Query q = em.createNativeQuery("SELECT "i.id, i.name, i.description " + "FROM Sale s, Item i " + "WHERE (s.quantity > 25) " + "AND (s.item = i.id)", Item.class)
    return q.getResultList();
}
```

- Return all items that some customer bought more than 25 of
- Result set maps simply to the Item entity
Object/Relational Mapping

- Map persistent object state to relational database
- Map relationships to other entities
- Metadata may be annotations or XML (or both)
- Annotations
  - Logical—object model (e.g. @OneToMany)
  - Physical—DB tables and columns (e.g. @Table)
- XML
  - Can additionally specify scoped settings or defaults
- Standard rules for default db table/column names
Object/Relational Mapping

- State or relationships may be loaded or “fetched” as EAGER or LAZY
  - LAZY is a hint to the Container to defer loading until the field or property is accessed
  - EAGER requires that the field or relationship be loaded when the referencing entity is loaded

- Cascading of entity operations to related entities
  - Setting may be defined per relationship
  - Configurable globally in mapping file 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, wrappers, enumerated, serializable, etc.
- 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
@javax.persistence.Entity
public class Customer {
    @javax.persistence.Id
    int id;
    String name;
    @javax.persistence.Column(name="CREDIT")
    int c_rating;
    @javax.persistence.Lob
    Image photo;
}
```
Simple Mappings

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

- Common relationship mappings supported
  - `@ManyToMany`, `@OneToMany`—collection of entities
  - `@ManyToOne`, `@OneToOne`—single entity
- 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
ManyToMany Mapping

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

    ...}

@ManyToOne
Customer cust;
```
ManyToMany Mapping

```xml
<entity class="com.acme.Sale">
  <attributes>
    <id name="id"/>
    ...
    <many-to-one name="cust"/>
  </attributes>
</entity>
```
@Entity
class Customer {
    @Id
    int id;
    ...
    @ManyToOne
    @OneToMany(mappedBy="cust")
    Set<Sale> sales;
    }

@Entity
class Sale {
    @Id
    int id;
    ...
    @ManyToOne
    Customer cust;
    }
OneToMany Mapping

```xml
<entity class="com.acme.Customer">
  <attributes>
    <id name="id"/>
    ...
    <one-to-many name="sales" mapped-by="cust"/>
  </attributes>
</entity>
```
@Entity
public class Customer {
    @Id
    int id;
    ...
    @ManyToMany
    Collection<Phone> phones;
}

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

CUSTOMER

<table>
<thead>
<tr>
<th>ID</th>
<th>...</th>
</tr>
</thead>
</table>

PHONE

| ID | ... |

CUSTOMER_PHONE

| CUSTS_ID | PHONES_ID |

PHONES_ID
ManyToMany Mapping

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

```xml
<entity class="com.acme.Customer">
  <attributes>
    ...
    <many-to-many name="phones">
      <join-table name="CUST_PHONE">
        <join-column name="CUST_ID"/>
        <inverse-join-column name="PH_ID"/>
      </join-table>
    </many-to-many>
  </attributes>
</entity>
```
@Entity
public class Customer
{
    @Id
    int id;
    
    @Embedded
    CustomerInfo info;
}

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

```xml
<entity class="com.acme.Customer">
  <attributes>
    ...
    <embedded name="info"/>
  </attributes>
</entity>

<embeddable class="com.acme.CustomerInfo">
  <attributes>
    <basic name="photo"><lob/></basic>
  </attributes>
</embeddable>
```
Identifier Generation

- Identifiers can be generated in the database by specifying `@GeneratedValue` on the identifier.
- Generators may pre-exist or be generated using `@SequenceGenerator` or `@TableGenerator`.
- 3 pre-defined generation strategies:
  - `IDENTITY`, `SEQUENCE`, `TABLE`
- Specifying strategy of `AUTO` indicates that the provider will choose a strategy.

```java
@Id @GeneratedValue
int id;
```
Identifier Generation

```java
@SequenceGenerator(name="CUST_GEN", 
sequenceName="MY_SEQ", 
allocationSize=100)

@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, 
generator="CUST_GEN")
    int id;
    ...  
}
```
Persistence Unit

- Set of entities and related classes that share the same configuration
- “Global” scoping boundary for queries
- Convenient packaging and deployment unit
- Associated with exactly one EntityManagerFactory, plus all EntityManagers created using it
- Runtime configuration defined in persistence.xml
- May include some metadata by containment or refer to other metadata by reference
Persistence in Java SE

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

- Resource-local EntityManagers
  - Application uses a local EntityTransaction obtained from the EntityManager

- New application-managed persistence context for each and every EntityManager
  - No propagation of persistence contexts
Entity Transactions

- Only used by Resource-local EntityManagers
- Isolated from transactions in other EntityManagers
- Transaction demarcation under explicit application control using EntityTransaction API
  - `begin()`, `commit()`, `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 PersistenceProgram {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("SomePUUnit");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        // Perform finds, execute queries, 
        // update entities, etc.
        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```
IDE Support

- Eclipse “Dali” project (http://www.eclipse.org/dali)
  - JPA support
  - Oracle (project lead), BEA, JBoss, Versant

- NetBeans (http://community.java.net/netbeans)
  - EJB 3.0 support including JPA (Beta 2)
  - Sun

- JDeveloper (http://www.oracle.com/technology/jdev)
  - EJB 3.0 support including JPA (10.1.3.1)
  - Oracle

- All 3 IDEs were developed against the JPA RI (TopLink Essentials) as the target platform
Summary

✓ JPA emerged from best practices of existing best of breed ORM products
✓ Lightweight persistent POJOs, no extra baggage
✓ Simple, compact and powerful API
✓ Standardized object-relational mapping metadata specified using annotations or XML
✓ Feature-rich query language
✓ Java EE integration, additional API for Java SE
✓ “Industrial strength” Reference Implementation
Summary

Broad persistence standardization, mass vendor adoption and sweeping community acceptance show that we finally have an enterprise persistence standard in the Java Persistence API.
Links and Resources

- JPA RI (TopLink Essentials) on Glassfish
  http://glassfish.dev.java.net/javaee5/persistence

- JPA white papers, tutorials and resources
  http://otn.oracle.com/jpa

- Pro EJB 3: Java Persistence API
  Mike Keith & Merrick Schincariol
  (Apress)
If you remember just one thing...

JPA

API

One Ring to Rule Them All