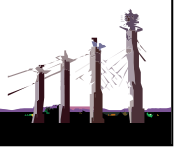

Java Database
Programming with JPA
Sérgio Alexandre Ferreira
Moredata, Lda
B15
Day, April 30, 2008 • 01:00 p.m. – 02:00 p.m.
2008 IIUG Informix Conference
www.iiug.org
2008 IIUG

Introduction

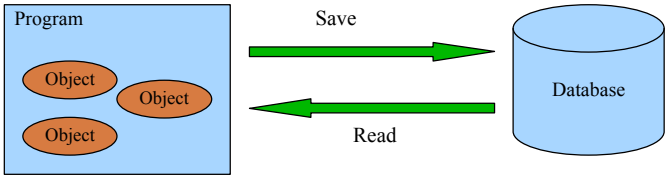
- The Java Persistence API provides an object/relational mapping facility for managing relational data in Java applications
- Created as part of EJB 3.0 within JSR 220
- Merger of expertise from TopLink, Hibernate, JDO, EJB vendors and individuals
- Released May 2006 as part of Java EE 5
- Integration with Java EE web and EJB containers provides enterprise “ease of use” features
- Can also be used in Java SE



Why persistence

- In programs data resides in memory
- In OO programs data is very well organized and structured
- Relational databases however are a very good way to store and access to data
- RDB are the most used mechanism to store large amounts of data.
- OO memory data is very different from relational organization of information
- Because we need to store data in databases there should exist an easy way to do it

How persistence works



What is JPA

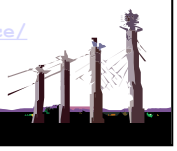
- Stands for Java Persistence Architecture
- It's a Java standard – JSR
- Evolved from the merge of several working projects on the industry (EJB, Hibernate, etc).
- Its an API and a set of tools including a query language

JSR = formal documents that describe proposed specifications and technologies to be added to the Java platform

Java Persistence

- Java Persistence consists of three areas:
  - The Java Persistence API
  - The query language
  - Object/relational mapping metadata
- JPA reference implementation
  - TopLink Essentials by GlassFish project
    - javax.persistence package
    - open source (under CDDL license)

https://glassfish.dev.java.net/javaee5/persistence/



What can we do with JPA

- Map classes and fields to tables and columns in a declarative way
  - Write data in objects to database just with a method call
  - Read data from the database tables into objects
- Generate the table schema to store data in objects
- Query the data from the database directly from objects

How we do it with JPA

- Annotate classes and fields to inform the tables where the class is persisted
- Execute methods to read / write /synchronize data from objects with the database
- Make queries in JPQL to access to the information
- The annotated classes are called Entities :-)

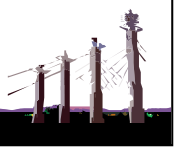
Entities

- An **entity** is a lightweight persistence domain object
- Java class that typically represents a table in a relational database, instances correspond to rows
- Requirements:
  - annotated with the `javax.persistence.Entity` annotation
  - `public` or `protected`, no-argument constructor
  - the class must not be declared `final`
  - no methods or persistent instance variables can be declared `final`

Requirements for Entities (cont.)

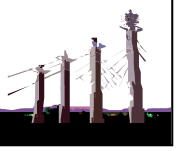
- May be **`Serializable`**, but not required
  - Only needed if passed by value (in a remote call)
- Entities may extend both entity and non-entity classes
- Non-entity classes may extend entity classes
- Persistent instance variables must be declared private, protected, or package-private
- No required business/callback interfaces
- Example:

```
@Entity
class Employee{
    . . .
}
```



Persistent Fields and Properties

- The persistent state of an entity can be accessed:
  - through the entity's instance variables
  - through JavaBeans-style properties
- Supported types:
  - primitive types, String, other serializable types, enumerated types
  - other entities and/or collections of entities
  - embeddable classes
- All fields not annotated with `@Transient` or not marked as Java transient will be persisted to the data store!



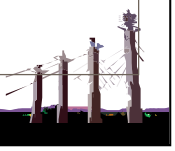
Primary Keys in Entities

- Each entity must have a unique object identifier (persistent identifier)

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private Date age;

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

primary key



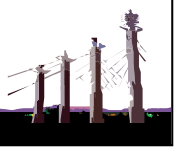
Persistent Identity

- Identifier (id) in entity = primary key in database
- Uniquely identifies entity in memory and in DB
- Persistent identity types:
  - Simple id – single field/property
    ```
    @Id int id;
    ```
  - Compound id – multiple fields/properties
    ```
    @Id int id;
    @Id String name;
    ```
  - Embedded id – single field of PK class type
    ```
    @EmbeddedId EmployeePK id;
    ```

Identifier Generation

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

```
@Id @GeneratedValue
private int id;
```



Customizing the Entity Object

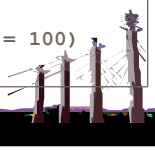
- In most of the cases, the defaults are sufficient
- By default the table name corresponds to the unqualified name of the class
  - Customization:

```
@Entity(name = "FULLTIME_EMPLOYEE")
public class Employee{ …… }
```

- The defaults of columns can be customized using the `@Column` annotation

```
@Id @Column(name = "EMPLOYEE_ID", nullable = false)
private String id;

@Column(name = "FULL_NAME" nullable = true, length = 100)
private String name;
```



Entity Relationships

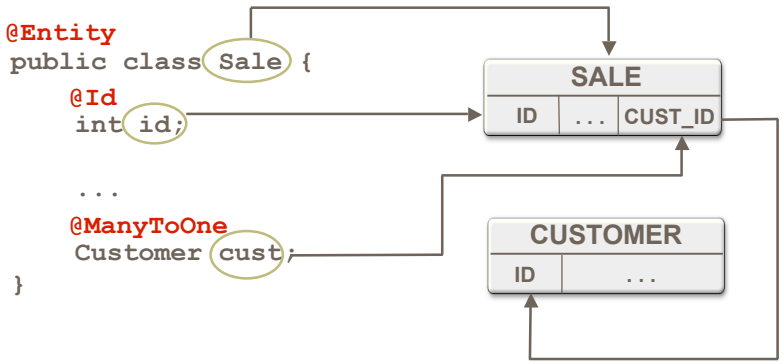
- There are four types of relationship multiplicities:
  - `@OneToOne`
  - `@OneToMany`
  - `@ManyToOne`
  - `@ManyToMany`
- The direction of a relationship can be:
  - **bidirectional** – owning side and inverse side
  - **unidirectional** – owning side only
- Owning side specifies the physical mapping

Relation Attributes

- CascadeType
  - ALL, PERSIST, MERGE, REMOVE, REFRESH
- FetchType
  - LAZY, EAGER

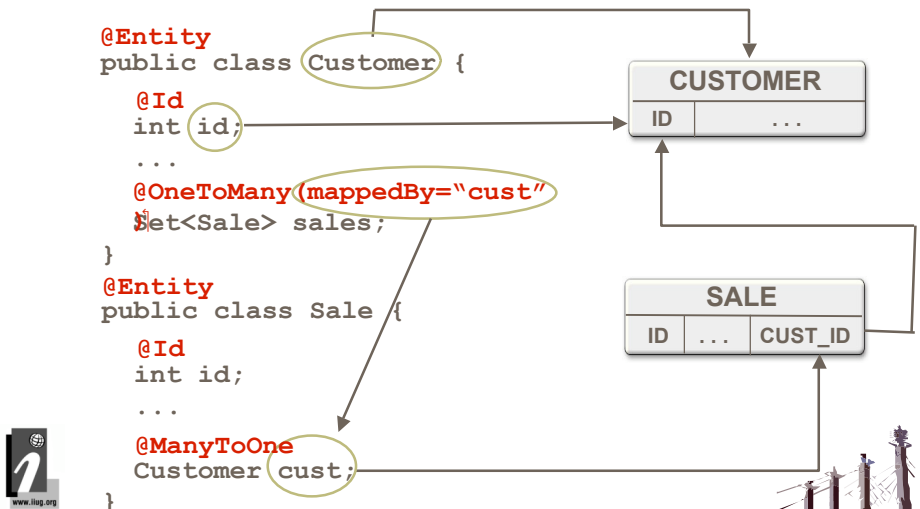
```
@ManyToMany(
    cascade = {CascadeType.PERSIST, CascadeType.MERGE},
    fetch = FetchType.EAGER)
```

ManyToOne Mapping



OneToMany Mapping

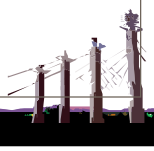
```
@Entity
public class Customer {
   @Id
   int id;
   ...
   @OneToMany(mappedBy="cust"
   Set<Sale> sales;
}
@Entity
public class Sale {
   @Id
   int id;
   ...
   @ManyToOne
   Customer cust;
}
```



ManyToMany Mapping

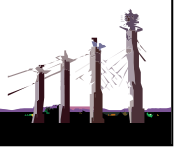
```
@Entity
public class Customer {
  ...
  @JoinTable(
        name="CUSTOMER_SALE",
    joinColumns=@JoinColumn(
        name="CUSTOMER_ID",referencedColumnName="customer_id"),
    inverseJoinColumns=@JoinColumn(
        name="SALE_ID", referencesColumnName="sale_id")
  Collection<Sale> sales;
}
```

```
@Entity
public class Sale {
   ...
  @ManyToMany(mappedBy="sales")
  Collection<Customer> customers;
}
```



Entity Inheritance

- An important capability of the JPA is its support for inheritance and polymorphism
- Entities can inherit from other entities and from non-entities
- The `@Inheritance` annotation identifies a mapping strategy:
  - SINGLE_TABLE
  - JOINED
  - TABLE_PER_CLASS

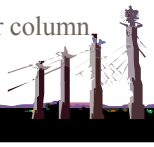


Inheritance Example

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=STRING)
@DiscriminatorValue(name="CUSTOMER")
public class Customer { . . . }

@Entity
@DiscriminatorValue(name="VCUSTOMER")
public class ValuedCustomer extends Customer { . . . }
```

- SINGLE_TABLE strategy - all classes in the hierarchy are mapped to a single table in the database
- Discriminator column - contains a value that identifies the subclass
- Discriminator type - {STRING, CHAR, INTEGER}
- Discriminator value - value entered into the discriminator column for each entity in a class hierarchy

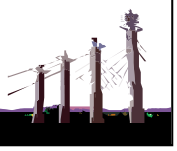


Managing Entities

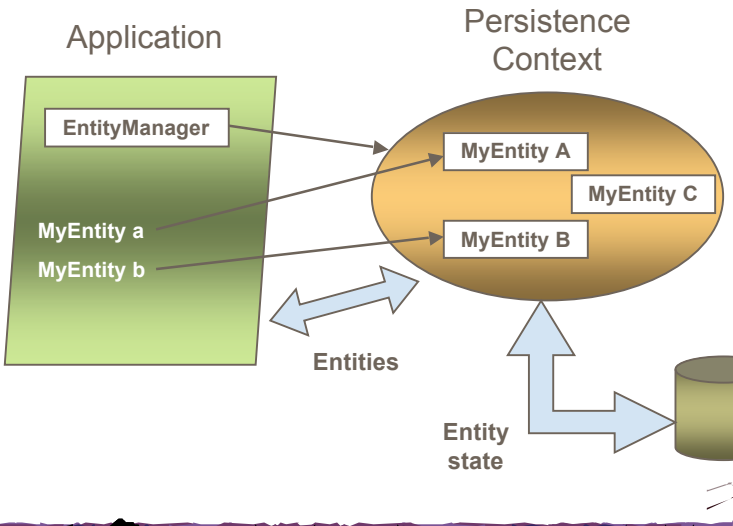
- Entities are managed by the **entity manager**
- The entity manager is represented by `javax.persistence.EntityManager` instances
- Each EntityManager instance is associated with a **persistence context**
- A persistence context defines the scope under which particular entity instances are created, persisted, and removed

Persistence Context

- A persistence context is a set of managed entity instances that exist in a particular data store
  - Entities keyed by their persistent identity
  - Only one entity with a given persistent identity may exist in the persistence context
  - Entities are added to the persistence context, but are not individually removable ("detached")
- Controlled and managed by EntityManager
  - Contents of persistence context change as a result of operations on EntityManager API

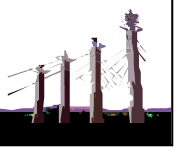


Persistence Context



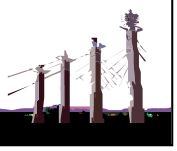
Entity Manager

- An EntityManager instance is used to manage the state and life cycle of entities within a persistence context
- The EntityManager API:
  - creates and removes persistent entity instances
  - finds entities by the entity's primary key
  - allows queries to be run on entities
- There are two types of EntityManagers:
  - Application-Managed EntityManagers
  - Container-Managed EntityManagers



Application-Managed EntityManager

- Applications create EntityManager instances by using directly **`Persistence`** and **`EntityManagerFactory:`**
  - **`javax.persistence.Persistence`**
    - Root class for obtaining an EntityManager
    - Locates 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



Application-Managed EntityManager

```
public class PersistenceProgram {
  public static void main(String[] args)
  {
    EntityManagerFactory emf =
   Persistence.createEntityManagerFactory("SomePUnit");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    // Perform finds, execute queries,
    // update entities, etc.
    em.getTransaction().commit();
    em.close();
    emf.close();
  }
}
```

Container-Managed EntityManagers

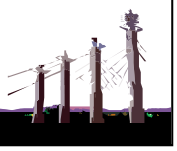
- An EntityManager with a transactional persistence context can be injected by using the `@PersistenceContext` annotation

```
public class BookmarkSeviceImpl implements BookmarkService {
    @PersistenceContext
    private EntityManager em;

    public void save(Bookmark bookmark) {
        if (bookmark.getId() == null) {
            em.persist(bookmark);
        } else {
            em.merge(bookmark);
        }
}
```

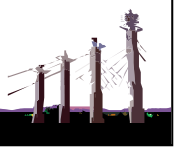
Transactions

- JPA transactions can be managed by:
  - the users application
  - a framework (such as Spring)
  - a J2EE container
- Transactions can be controller in two ways:
  - Java Transaction API (JTA)
    - container-managed entity manager
  - EntityTransaction API (`tx.begin(), tx.commit(), etc`)
    - application-managed entity manager



Operations on Entity Objects

- EntityManager API operations:
  - `persist()` - Insert the state of an entity into the db
  - `remove()` - Delete the entity state 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 JP 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

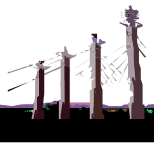


Entity Instance's Life Cycle

- Instances are in one of four states:
  - New
  - Managed
  - Detached
  - Removed
- The state of persistent entities is synchronized to the database when the transaction commits
- To force synchronization of the managed entity to the data store, invoke the `flush()` method

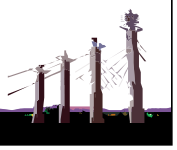
Persistence Units

- A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application
- For example, some set of entities can share one common provider (Toplink), whereas other set of entities can depend on a different provider (Hibernate)
- Persistence units are defined by the persistence.xml configuration file



And JPQL

- It is a query language
- Operates against entity objects
- Serves as a proxy to access to the database
- Used to maintain independence from the database

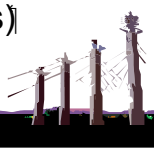


How do I start

- Download one of the implementations:
  - Toplink
  - Hibernate
  - Apache OpenJPA
- Read the documents
- Start a project on eclipse or netbeans

Conclusion

- JPA its a standard to maintain objects synchronized with databases
- JPA is easy to use
- JPA don't need an application server
- JPA can be used in J2EE or J2SE
- There are several JPA implementations
- Code made with JPA it's portable between implementations and databases
- With JPA software is a lot more independent of vendors (Application Servers and Databases)



Session ####

Java Database Programming with JPA

Sérgio Ferreira

Moredata

Sergio.Ferreira@moredata.pt