



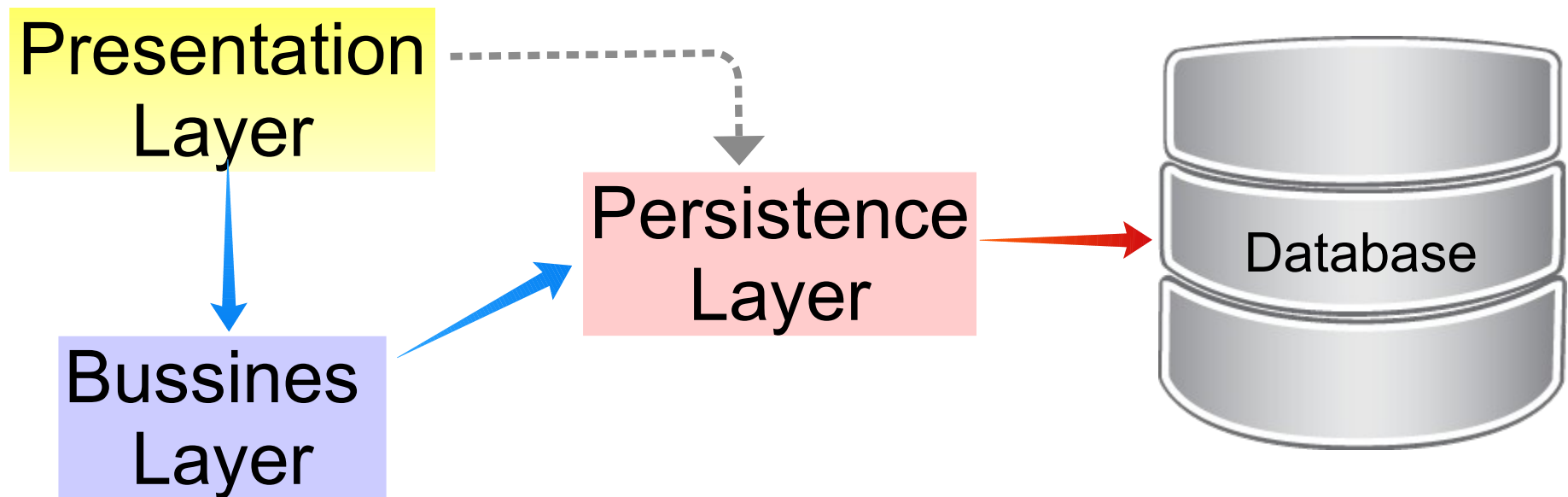
Advanced Programming

Java Persistence API

(JPA) - *Introduction*

Persistence Layer

The Persistence Layer is used by an application in order to persist its state, that is to store and retrieve information using some sort of database management system.



Different Perspectives About Data

- Relational Level

```
CREATE TABLE persons (  
  id integer NOT NULL,  
  name varchar(50) NOT NULL,  
  salary float,  
  PRIMARY KEY(id));
```

```
INSERT INTO persons (id, name) VALUES (1, 'John Doe');
```

```
UPDATE persons SET salary=2000 WHERE id=1;
```

SQL Guy



- Object Oriented Level

```
public class Person {  
  public String name;  
  public float salary;  
  public Person(String name) { ... }  
}  
  
Person p = new Person("John Doe");  
PersistenceLayer.save(p);  
p.setSalary(2000);  
PersistenceLayer.update(p);
```

Programmer



JDBC

an “SQL” API for Programmers

// Specify the driver

```
Class.forName("org.postgresql.Driver");
```

// Create a connection to the database

```
Connection con = DriverManager.getConnection(
    "jdbc:postgresql://localhost/demo", "dba", "sql");
```

// Create an SQL statement

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select id, name from persons");
```

// Iterate through the ResultSet (SQL Cursor)

```
while (rs.next()) {
    int id = rs.getInt("id");
    String nume = rs.getString("name");
    System.out.println(id + ". " + name);
}
rs.close();           // Don't forget to close the ResultSet!
stmt.close();         // Don't forget to close the Statement!
con.close();          // Don't forget to close the Connection!!!
```

Object-Relational Mapping (ORM)

- **Accessing relational data using OO paradigm**
- **Objects ↔ Mapping Layer ↔ Relations**
- **Advantages:**
 - Simplified development using automated conversions between objects and tables. No more SQL in the Java code.
 - Less code compared to embedded SQL and stored procedures
 - Superior performance if object caching is used properly
 - Applications are easier to maintain
- **Disadvantages:**
 - the additional layer may slow execution *sometimes*
 - defining the mapping may be difficult *sometimes*

“Impedance Mismatch”

Graph of objects vs Relations (sets of tuples)

❏ Granularity

→ How many classes vs *How many tables*

❏ Subtypes

→ Inheritance vs *None*

❏ Identity

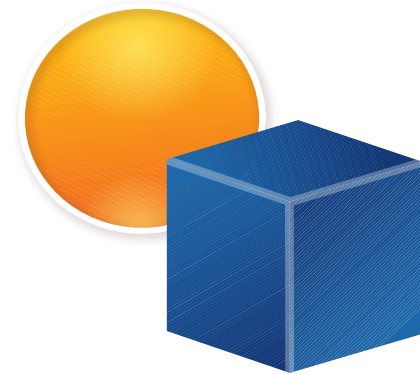
→ `==` or equals vs *Primary Keys*

❏ Associations

→ Unidirectional references vs *ForeignKeys*

❏ Data Navigation

→ One object to another vs *Queries*



The Mapping Layer

- *What database, what SQL dialect, etc?*
 - **Initialization parameters**
- *How to define the mapping?*
 - **Mapping metadata: XML files, annotations**
 - Class 'Person' ↔ table 'persons', ...
- *How to persist an object?*
 - **Standard programming interface (API)**
- *How to find an object?*
 - **Object oriented query language**

Mapping: Example 1

class ↔ table, property ↔ column, easy...

```
// The class
```

```
class Person {  
    int id;  
    String name;  
    Date birthDate;  
    double salary;  
}
```

```
--The table
```

```
persons (  
    id int NOT NULL,  
    name varchar(50),  
    date_of_birth date,  
    income numeric(10,2),  
)
```

```
// The programmer writes:
```

```
Person p = new Person();  
p.setId(1);  
p.setName("John Doe");  
p.setBirthDate(  
    new SimpleDateFormat("dd.MM.yyyy")  
        .parse("01.01.1991");  
p.setSalary(2000);
```

```
// Wishful thinking:
```

```
mappingLayer.persist(p);
```

```
//The SQL code is generated:
```

```
INSERT INTO persons  
(id, name, date_of_birth, income)  
VALUES  
(1, 'John Doe', Date('1991/01/01'), 2000);
```


Mapping: Example 2

class ↔ table, property ↔ ..., *easy?*

```
class Order {  
    int id;  
    Date date;  
    Set<Item> items;  
}  
  
class Item {  
    private int id;  
    String product;  
    double quantity, price;  
    Order order;  
}
```

```
orders(  
    id int,  
    date timestamp  
)  
  
items(  
    id int,  
    order_id int  
        references orders  
        on delete cascade,  
    product varchar(50),  
    quantity double,  
    price double  
)
```

Mapping: Example 3

2 classes \leftrightarrow 3 tables, *not so easy...*

```
class Product {  
    int id;  
    String name;  
    double price;  
    Set<Category> categories;  
}
```

```
class Category {  
    private int id;  
    String name;  
}
```

```
products(  
    id int,  
    name varchar(50),  
    price double  
)  
categories(  
    id int,  
    name varchar(50)  
)  
products_categories(  
    product_id int  
        references products  
        on delete cascade,  
    category_id int  
        references categories  
        on delete restrict  
)
```

Mapping Relations Problem

- Types of Relations (Multiplicity / Direction)

- One-to-One

- Employee 0..1 -holds- 1 Position (uni)

- Employee 1 -owns- 0..1 Computer (bi)

- One-to-Many

- Employee 1..* -assigned- 1 Duty (uni)

- Employee 0..* -works- 1 Division (bi)

- Many-to-Many

- Employee 0..* -assigned- 0..* Printer (uni)

- Employee 0..* -assigned- 0..* Project (bi)

- The “Cascade” Issue

```
Order order = new Order();
```

```
Item item = new Item(...);
```

```
order.addItem(item); ...
```

```
mappingLayer.save(order);
```

```
→ item.setOrderId(order.getId());
```

```
→ mappingLayer.persist(item);
```

The Inheritance Problem

There is no natural, efficient way to represent an inheritance relationship in a relational database.

Person

-> Customer

-> Employee

-> Executive

```
abstract class Person {  
    String name;  
}  
class Customer extends Person {  
    String preferences;  
}  
class Employee extends Person {  
    float salary;  
}  
class Executive extends Employee {  
    float bonus;  
}
```

• Single Table

persons(id, type, name, salary, bonus)

persons(id, isCust, isEmpl, isExec, name, salary, bonus)

• Table Per Concrete Class

customers (id, name, preferences)

employees (id, name, salary)

executives (id, name, salary, bonus)

• Joined

persons (id, name)

customers (personId, preferences)

employees (personId, salary)

executives (employeeId, bonus)

The Primary Key Problem

Every entity object in the database is uniquely identified (and can be retrieved from the database) by the combination of its type and its primary key.

- ❏ **Semantic (Simple or Composite)**

- Application Set Primary Key

- ❏ **Surrogate** (not derived from application data)

- Automatic/Generated Primary Key

- Autoincrement
 - Sequences
 - Universally Unique Identifiers (UUID)
 - HIGH-LOW

The SQL Dialect Problem

The dialect of the database defines the specific features of the SQL language that are available when accessing a database.

- DDL

--Sybase

```
CREATE TABLE persons (id integer NOT NULL DEFAULT autoincrement,  
    name varchar(50) NOT NULL, "key" integer, PRIMARY KEY(id));
```

--MySQL

```
CREATE TABLE persons (id integer NOT NULL auto_increment,  
    name varchar(50) NOT NULL, 'key' integer, PRIMARY KEY (id)  
) ENGINE = InnoDB ;
```

- DML

--Sybase

```
SELECT FIRST id FROM clients WHERE name = 'Popescu' ORDER BY id;
```

```
UPDATE clients, cities
```

```
    SET clients.cityName = cities.name
```

```
    WHERE cities.id = clients.cityId;
```

--PostgreSQL

```
SELECT id FROM clients WHERE name = 'Popescu' ORDER BY id LIMIT 1;
```

```
UPDATE clients SET cityName = city.name
```

```
    FROM (select id, name from cities ) as city
```

```
    WHERE cityId = city.id;
```

ORM Implementations

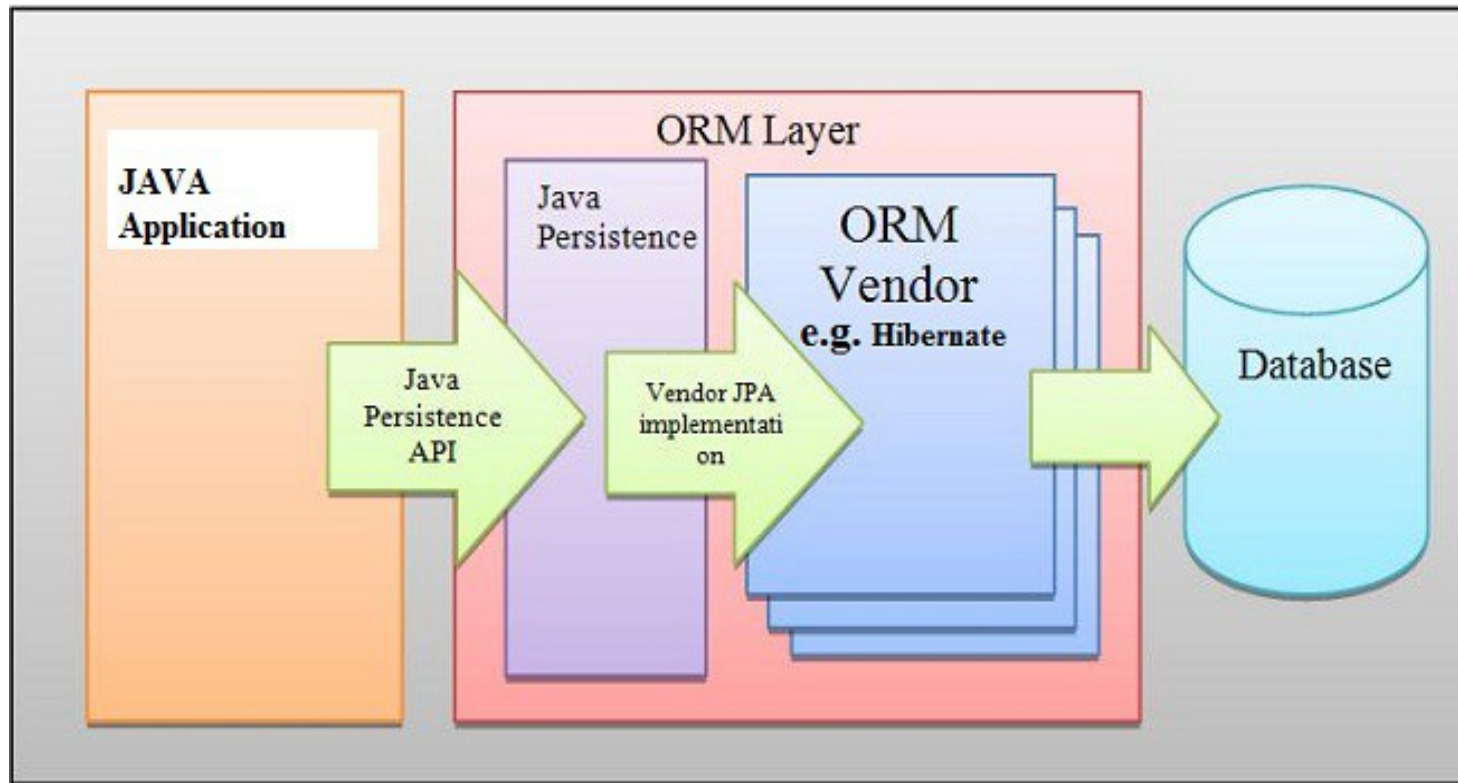
- Java
 - Hibernate (JBoss)
 - TopLink (Oracle) → EclipseLink
 - OpenJPA (Apache),
- .NET
 - ADO.NET Entity Framework
 - NHibernate ,...
- PHP
 - Doctrine, Redbean, ...
- ... *Every respectable programming platform has it*

Java Persistence API

- Object/relational mapping **specifications** for managing relational data in Java applications
- Consists of:
 - The Java Persistence **API**
 - Java Persistence Query Language (**JPQL**)
 - The Java Persistence **Criteria API**
 - O/R mapping metadata (**Persistence Annotations**)
- Implemented by:
 - most of the Java ORM producers

What JPA-ORM Should I Use?

It doesn't (shouldn't) matter from the programmer perspective...



You don't like Hibernate runtime performance anymore?
Simply replace its libraries with another implementation,
like EclipseLink or OpenJPA for instance.

Entities

- **Entity** = lightweight persistence domain object:
 - an entity class represents a table and
 - an entity instance corresponds to a row in that table.
- **Persistence annotations** are used to map the entities to the relational data.
- **Convention over configuration**

```
@Entity
@Table(name = "persons") //only configure the exceptions
public class Person implements Serializable {
    @Id
    private Integer id;

    private String name;
}
```

Persistence Annotations

```
@Entity
@Table(name = "PERSONS")
public class Person implements Serializable {
    @Id
    @SequenceGenerator(name = "sequence",
                       sequenceName = "persons_id_seq")
    @GeneratedValue(generator = "sequence")
    @Column(name = "PERSON_ID")
    private Integer id;

    @Column(name = "NAME")
    private String name;

    @JoinColumn(name = "DEPT_ID")
    @ManyToOne
    private Departament departament;
}
```

Persistence Units

- A **persistence unit** defines the set of all entity classes that are managed by an application.
 - Defined at **design time** in **persistence.xml**
 - *javax.persistence.PersistenceUnit*
- This set of entity classes represents the data contained within a single data store.
 - An application may use multiple persistence units
- A **persistence context** defines a set of entity instances managed at **runtime**.
 - *javax.persistence.PersistenceContext*

Example: *persistence.xml*

```
<persistence>
  <persistence-unit name="MyApplicationPU"
    transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>myapp.entity.Person</class>
    <class>myapp.entity.Departament</class>

    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.connection.driver_class"
        value="org.postgresql.Driver"/>
      <property name="hibernate.connection.url"
        value="jdbc:postgresql://localhost/timetable"/>
    </properties>

  </persistence-unit>
</persistence>
```

Managing Entities

- Entities are managed by ... the **EntityManager**.
- Each EntityManager instance is associated with a **persistence context**: *a set of managed entity instances that exist in a particular data store.*
- The EntityManager defines the methods used to interact with the persistence context:
 - **`persist, remove, refresh, find, ...`**
- An EntityManager
is created by ...
an **EntityManagerFactory**
 - ← not expensive
not thread safe
 - ← expensive
thread safe

Creating an *EntityManager*

- **Application-Managed Entity Managers**

```
EntityManagerFactory factory =  
    Persistence.createEntityManagerFactory(  
        "MyApplicationPU", properties);  
EntityManager em = factory.createEntityManager();  
...  
em.close();  
...  
factory.close();
```

```
<persistence-unit  
    name="MyApplicationPU"  
    transaction-type="RESOURCE_LOCAL">
```

Example

The relational model

```
create table products(  
    id integer not null  
        generated always as identity (start with 1, increment by 1),  
    name varchar(100) not null,  
    primary key (id)  
);
```

```
create table orders(  
    id integer not null  
        generated always as identity (start with 1, increment by 1),  
    order_date date not null,  
    primary key (id)  
);
```

```
create table order_items (  
    id integer not null  
        generated always as identity (start with 1, increment by 1),  
    order_id integer not null  
        references orders on delete cascade,  
    product_id integer not null  
        references products on delete restrict,  
    quantity double not null,  
    price double not null,  
    primary key (id)  
);
```


The *Product* class

```
@Entity
@Table(name = "PRODUCTS")
@NamedQueries({
    @NamedQuery(name = "Product.findByName",
        query = "SELECT p FROM Product p WHERE p.name=:name") })
public class Product implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "ID")
    private Integer id;

    @Basic(optional = false)
    @Column(name = "NAME")
    private String name;
    ...
}
```

The *Order* class

```
@Entity
@Table(name = "ORDERS")
public class Order implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "ID")
    private Integer id;

    @Column(name = "ORDER_DATE")
    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "order")
    private List<OrderItem> items = new ArrayList<>();

    @OneToOne(mappedBy = "order");
    private Invoice invoice;

    ...
}
```

The *OrderItem* class

```
@Entity
@Table(name = "ORDER_ITEMS")
public class OrderItem implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "ID")
    private Integer id;

    @Column(name = "QUANTITY")
    private double quantity;

    @Column(name = "PRICE")
    private double price;

    @JoinColumn(name = "PRODUCT_ID", referencedColumnName = "ID")
    @ManyToOne
    private Product product;

    @JoinColumn(name = "ORDER_ID", referencedColumnName = "ID")
    @ManyToOne
    private Order order;
    ...
}
```

Persist

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("DemoPU");  
EntityManager em = emf.createEntityManager();
```

```
em.getTransaction().begin();  
Product water = new Product("Still water");  
Product rice = new Product("Rice");  
Product fish = new Product("Salmon");
```

```
Order order = new Order();  
Date today = new Date();  
order.setOrderDate(today);
```

```
order.addItem(new OrderItem(rice, 1, 10));  
order.addItem(new OrderItem(fish, 1, 30));  
order.addItem(new OrderItem(water, 2, 5));
```

```
em.persist(order);  
em.getTransaction().commit();
```

```
em.close();  
emf.close();
```

Find, *Update* and Delete

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("DemoPU");
EntityManager em = emf.createEntityManager();

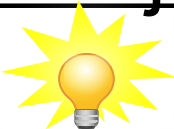
em.getTransaction().begin();
int waterId = 1;
Product water = em.find(Product.class, waterId);
water.setName("Sparkling water");

Product rice =
    (Product) em.createNamedQuery("Product.findByName")
        .setParameter("name", "Rice")
        .getSingleResult();
rice.setName("White rice");

int orderId = 25;
Order order = em.find(Order.class, orderId);
em.remove(order);
em.getTransaction().commit();

em.close();
emf.close();
```

Automatic Dirty Checking

- An ORM doesn't update the database row of every single persistent object in memory at the end of the unit of work.
- An ORM must have a strategy for detecting which persistent objects have been modified by the application. 
- An ORM should be able to detect exactly which properties have been modified so that it's possible to include only the columns that need updating in the SQL UPDATE statement.

Lifecycle Callbacks

- *@PostLoad*
- *@PrePersist*, *@PostPersist*,
- *@PreUpdate*, *@PostUpdate*
- *@PreRemove*, *@PostRemove*

```
public class OrderLogger {  
    @PostPersist  
    public void logAddition(Object order) {  
        System.out.println("Added:" + order);  
    }  
    @PreRemove  
    public void logDeletion(Object order) {  
        System.out.println("Deleted:" + order);  
    }  
}
```

```
@Entity  
@EntityListeners({ OrderLogger.class })  
public class Order {  
    ...  
}
```

JP Query Language (JPQL)

- **Queries for entities** and their persistent state.
- **Portable queries** that work regardless of the underlying data store.
- Uses an **SQL-like syntax**

```
QL_statement ::= select_clause from_clause  
               [where_clause]  
               [groupby_clause]  
               [having_clause]  
               [orderby_clause]
```

```
SELECT p FROM Person p WHERE p.name LIKE "%Duke%"
```


Creating Queries

- Dynamic Queries

```
public List<Person> findByName(String name) {  
    Query query = entityManager.createQuery(  
        "SELECT p FROM Person p WHERE p.name LIKE :personName")  
        .setParameter("personName", name)  
        .setMaxResults(10);  
    return query.getResultList();  
}
```

- Static Queries

```
@NamedQuery(name="findById",  
    query="SELECT p FROM Person p WHERE p.id = :personId")  
)  
@Entity  
@Table(name="persons")  
class Person { ... }
```

Using a static query:

```
Person p = em.createNamedQuery("findById")  
    .setParameter("personId", 1)  
    .getSingleResult();
```

Queries That Navigate to Related Entities

- An expression can traverse, or navigate, to related entities. JP-QL navigates to related entities, whereas SQL joins tables.

- Using JOIN

```
- List<Person> persons = entityManager().createQuery(  
    "select p from Person p join p.departaments as d "  
    + "where d.name = :dept "  
    + "order by p.name").  
    setParameter("dept", someDeptName).  
    getResultList();
```

- Using IN

```
- select distinct p from Person, in (p.departaments) d
```

*The **p variable** represents the Person entity, and the **d variable** represents the related Department entity. The declaration for d references the previously declared p variable. The **p.departaments** expression navigates from a Person to its related Departament.*

Native Queries

Going back to good old SQL ...

- Simple SQL queries

```
BigInteger count = (BigInteger) entityManager.createNativeQuery(  
    "select count(*) from persons where date_of_birth=:date").  
    setParameter("date", birthDate).  
    getSingleResult();
```

- SQL queries mapped to entites

```
String sqlQuery = "select * from persons where date_of_birth = ?";  
Query q = entityManager.createNativeQuery(sqlQuery, Person.class);  
q.setParameter( 1, birthDate);  
List<Person> persList = q.getResultList();
```

- To be continued ...