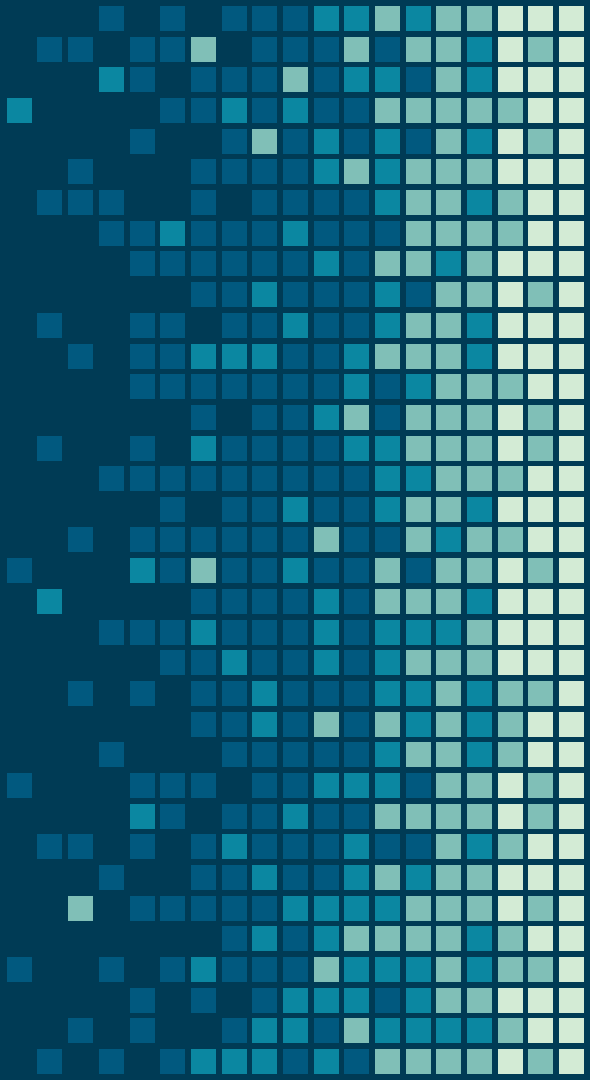
Java Persistence API – part 1

Entities, Relationships, Criteria API



Tamás Ambrus

Entities

ORM, POJO, Entity requirements

Revision: JDBC

```
try (Statement stmt = conn.createStatement()) {
    ResultSet rs = stmt.executeQuery("SELECT * FROM user");
    while (rs.next()) {
        int id = rs.getInt("Id");
        String username = rs.getString("Username");
        byte[] salt = rs.getBytes("Salt");
        System.out.println(id + " " + username + " " + new String(salt));
    }
}
```

- What's the problem with this approach?
  Easy typo, not type safe
- What's the problem with Strings at all?
  The possibility of typos,
  Doesn't get updated when changing something

- JDBC is a bridge between Java and SQL-based databases
- We worked with Strings, all queries used String parameters: the SQL commands.

We need some improvement!

More info at https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html

JPA (Java Persistence API)

„The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications." – Oracle documentation

ORM is a technique, maps the relational data and object to make the communication easier between DB and prog. language

More info at https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

JPA Entities

The power of JPA is that „entities" (that represent a table of the DB) are like regular Java objects.

Plain Old Java Object:

```
public class Employee {

    private int id;
    private String name;
    private long salary;

    public Employee() {
    }

    public Employee(int id) {
        this.id = id;
    }

    // getters and setters for all at
    public int getId() { return id; }
    public void setId(int id) { this.
    public String getName() { return
    public void setName(String name)
    public long getSalary() { return
    public void setSalary(long salary
}
```

```
@Entity
public class Employee {

    @Id
    private int id;
    private String name;
    private long salary;

    public Employee() {
    }

    public Employee(int id) {
        this.id = id;
    }

    // getters and setters for all attributes
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary(long salary) { this.salary = salary; }
}
```

How to be an Entity

- The class must be annotated with javax.persistence.Entity

  That's how JPA implementations find out that a type is a table.

- The class must have a public or protected, no-argument constructor

  Why? Because of reflection. JPA is based on reflection.

- Can't be final. Neither methods nor persistent instance variables

  For variables, reflection is the answer. And this is for methods.

- They may extend both entity and non-entity classes
  Non-entity classes may extend entity classes

More info at https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html

What types can Entity instance variables have?

- Primitive Java types: byte, short, int, long, float , double, boolean, char
- Wrappers of primitive Java types: Integer, Character, etc.
- Byte and Character array types: byte[], char[], Byte[], Character[]
- Large numeric types: java.math.BigInteger, java.math.BigDecimal
- Temporals: java.util.Date, java.sql.Date, Calendar, Time, TimeStamp
- String, enumerated types, serializable types, collection of all above

More info at https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html

Relationships

OneToOne, OneToMany, ManyToMany, ManyToOne

What relationships can occur in a DB?

- One-to-One: Each entity instance is related to a single one of another

Teacher:

ID	NAME	HOME_ID
3	Zsofi	2
4	Tamas	1
NULL	NULL	NULL

House:

ID	ADDRESS
1	Budapest. some street 36.
2	New York. famous street 1.
NULL	NULL

The owner type will have an extra column

```
@Entity
public class Teacher {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToOne
    private House home;
```

More info at https://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561

What relationships can occur in a DB?

- One-to-Many: Each entity instance is related to many of another

Teacher:

ID	NAME
1	Zsofi
2	Tamas
NULL	NULL

Student:

ID	AGE	NAME
3	21	Sophie
4	20	Peter
NULL	NULL	NULL

Teacher_Student:

Teacher_ID	students_ID
1	3
1	4
NULL	NULL

```
@Entity
public class Teacher {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToMany
    private Set<Student> students;
```

An extra table gets created to map primary keys

More info at https://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561

What relationships can occur in a DB?

- Many-to-One: It's the opposite of One-to-Many relationship

Teacher:

	ID	NAME
	1	Zsofi
	2	Tamas
	NULL	NULL

Student:

	ID	AGE	NAME	TEACHER_ID
	3	21	Sophie	2
	4	20	Peter	2
	NULL	NULL	NULL	NULL

```java
@Entity
public class Student {

    @Id
    @GeneratedValue
    private int id;

    private String name;
    private int age;

    @ManyToOne
    private Teacher teacher;
```

An extra column is enough to express many-to-one

More info at https://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561

What relationships can occur in a DB?

- Many-to-Many: Many entity instances are related to many of another

Teacher:

ID	NAME
1	Zsofi
2	Tamas
NULL	NULL

Student:

ID	AGE	NAME
3	21	Sophie
4	20	Peter
NULL	NULL	NULL

Teacher_Student:

teachers_ID	students_ID
1	3
1	4
NULL	NULL

```java
@Entity
public class Teacher {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @ManyToMany
    private List<Student> students;
```

```java
@Entity
public class Student {

    @Id
    @GeneratedValue
    private int id;

    private String name;
    private int age;

    @ManyToMany(mappedBy="students")
    private Set<Teacher> teachers;
```

More info at https://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561

What relationships can occur in JPA?

- unidirectional: the owning side knows about the inverse side entity, but not vice versa

- bidirectional: both owning and inverse sides know about the other

To make a relationship bidirectional, we use the mappedBy parameter.

Note that forgetting the mappedBy parameter would cause something different that you expected: it would result in 2 unidirectional relationships instead of 1 bidirectional.

More info at https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html

Bidirectional relationship example

Owning:

```
@Entity
public class Teacher {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToOne
    private House home;
```

Inverse:

```
@Entity
public class House {

    @Id
    @GeneratedValue
    private int id;

    private String address;

    @OneToOne(mappedBy = "home")
    private Teacher owner;
```

In this case we don't have to set the Teacher in the House object, **JPA will pair it automatically** to the owner.

EntityManager, Criteria API

Entity search, Criteria API, examples

EntityManager

The base of managing the entities is provided by the javax.persistence.EntityManager interface:

- it is able to create and remove persistent entity instances
- find entities by their primary key
- allow queries run on entities

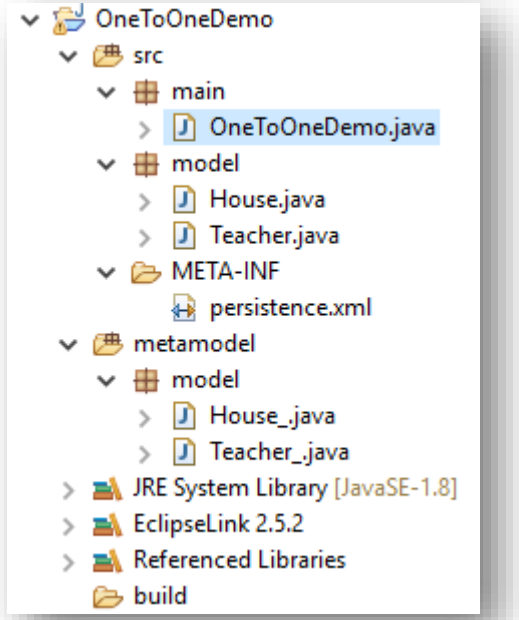
```
EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("PSE_persistence_unit");
EntityManager entityManager = emfactory.createEntityManager();

// you do your job here
/* e.g. */ House h = entityManager.find(House.class, 2); // house with ID=2
System.out.println(h.getAddress());

// always close both, they don't implement the "AutoCloseable" interface
entityManager.close();
emfactory.close();
```

You need to give the name of your persistence unit

More info at https://docs.oracle.com/javaee/6/tutorial/doc/bnbqw.html

Inserting data into DB example

```
public class OneToOneDemo {
    public static void main(String[] args) {

        EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("PSE_persistence_unit");
        EntityManager entityManager = emfactory.createEntityManager();
        entityManager.getTransaction().begin();

        House h1 = new House("Budapest, some street 36.");

        House h2 = new House("New York, famous street 1.");

        Teacher t1 = new Teacher("Zsofi", h2);

        Teacher t2 = new Teacher("Tamas", h1);

        entityManager.persist(h1);
        entityManager.persist(h2);
        entityManager.persist(t1);
        entityManager.persist(t2);

        entityManager.getTransaction().commit();

        entityManager.close();
        emfactory.close();

    }
}
```



Criteria API

The Criteria API is used to define queries on entities. These queries are **type-safe**.

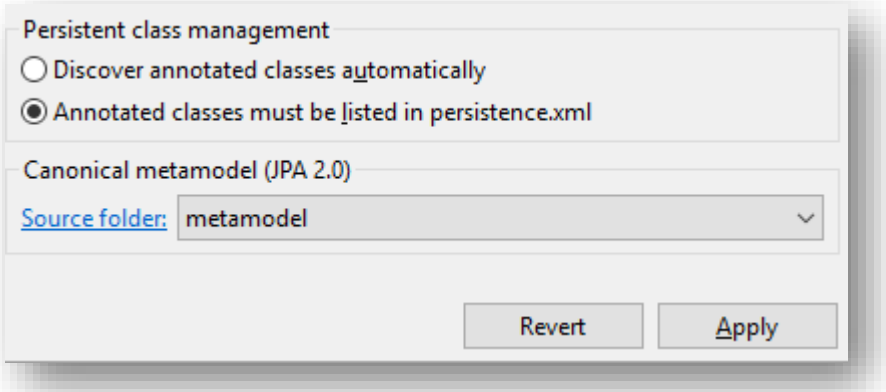
```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<House> house = cq.from(House.class);

cq.multiselect(house.get(House_.id), house.get(House_.teacher).get(Teacher_.name));

entityManager.createQuery(cq).getResultList().forEach(t -> {
    System.out.println("Id: " + t.get(0) + ", Owner's name: " + t.get(1));
});
```

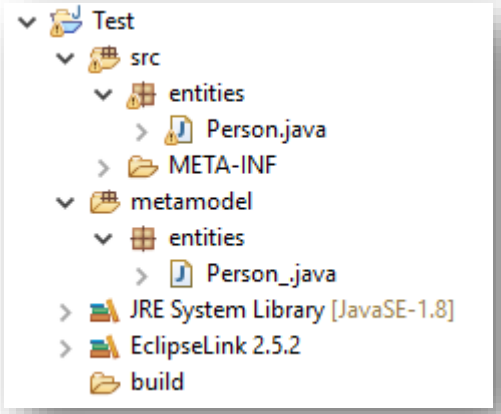
More info at https://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

Criteria API: Metamodel

To have all needed Java types and variables of entities, we need JPA to generate metamodels **automatically** for all entities:

- right click on your JPA project → Properties → JPA → Canonical metamodel
- select the „metamodel" source folder (create if doesn't exist)



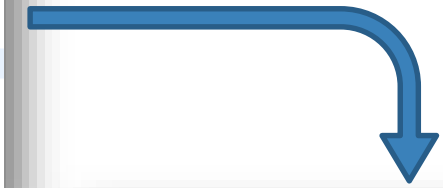


More info at https://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

Criteria API: Metamodel

During you edit your entities the metamodel changes appropriately.

```java
@Entity
public class Person {

    @Id
    private int id;

    private String nam;

    public Person() {}

}
```



```java
package entities;

import javax.annotation.Generated;

@Generated(value="Dali", date="2018-03-16T10:43:14.975+0100")
@StaticMetamodel(Person.class)
public class Person_ {
    public static volatile SingularAttribute<Person, Integer> id;
    public static volatile SingularAttribute<Person, String> nam;
}
```

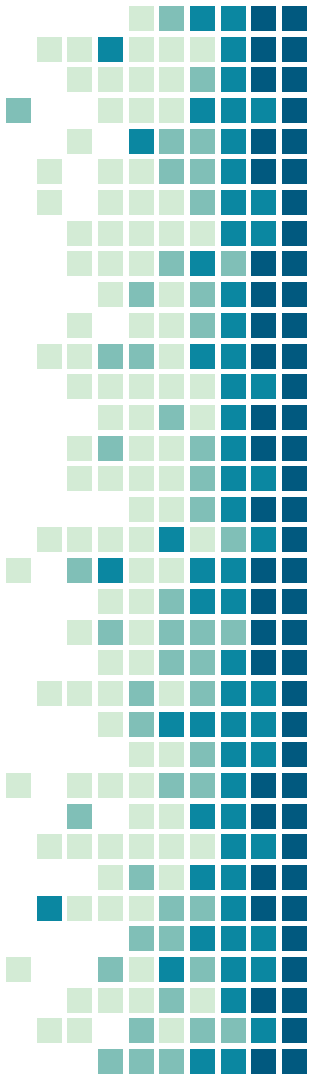
More info at https://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

Criteria API: Select clause

```
SELECT * FROM psetestdb.house;
```

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<House> cq = cb.createQuery(House.class);
Root<House> house = cq.from(House.class);

cq.select(house);

entityManager.createQuery(cq).getResultList().forEach(h -> {
    System.out.println("Id: " + h.getId() + ", Address: " + h.getAddress());
});
```

More info at https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html



Criteria API: Select clause

```
SELECT * FROM psetestdb.house;
```

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<House> cq = cb.createQuery(House.class);
Root<House> house = cq.from(House.class);

cq.select(house);

entityManager.createQuery(cq).getResultList().forEach(h -> {
    System.out.println("Id: " + h.getId() + ", Address: " + h.getAddress());
});
```

We need a single CriteriaBuilder per application

That's how we create an SQL query

We have House objects

We execute the query by calling this method

We make our query executable

More info at https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html

Criteria API: Select clause – some fields only

```
SELECT h.id, t.name FROM psetestdb.house h JOIN psetestdb.teacher t ON (t.home_id = h.id);
```

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<House> house = cq.from(House.class);

cq.multiselect(house.get(House_.id), house.get(House_.teacher).get(Teacher_.name));

entityManager.createQuery(cq).getResultList().forEach(t -> {
    System.out.println("Id: " + t.get(0) + ", Owner's name: " + t.get(1));
});
```

More info at https://stackoverflow.com/questions/12618489/jpa-criteria-api-select-only-specific-columns

Criteria API: Advanced SQL query

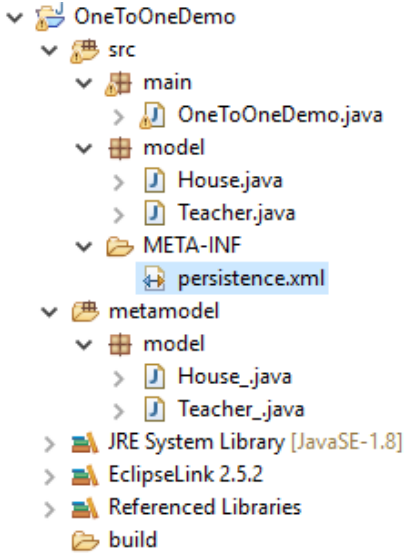
```sql
SELECT t.name, COUNT(*) FROM teacher t, house h
    WHERE t.home_id = h.id
    GROUP BY t.name
    HAVING COUNT(*) < 2
    ORDER BY t.name DESC;
```

```java
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<House> house = cq.from(House.class);

Path<String> teacherName = house.get(House_.teacher).get(Teacher_.name);
Expression<Long> homeCount = cb.count(house);

cq.multiselect(teacherName, homeCount);
cq.groupBy(teacherName);
cq.having(cb.lt(homeCount, 2));
cq.orderBy(cb.desc(teacherName));

entityManager.createQuery(cq).getResultList().forEach(t -> {
    System.out.println(t.get(0) + " has " + t.get(1) + " house(s).");
});
```

More info at https://docs.oracle.com/javaee/6/tutorial/doc/gjivm.html

Persistence.xml

In this xml file we set up **persistence units** that configure the connection.




```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="PSE_persistence_unit"
        transaction-type="RESOURCE_LOCAL">
        <class>model.Teacher</class>
        <class>model.House</class>

        <properties>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://localhost:3306/psetestdb" />
            <property name="javax.persistence.jdbc.user" value="tanulo" />
            <property name="javax.persistence.jdbc.password" value="asd123" />
            <property name="javax.persistence.jdbc.driver"
                value="com.mysql.jdbc.Driver" />
            <property name="eclipselink.logging.level" value="FINEST" />
            <property name="eclipselink.ddl-generation" value="create-tables" />
        </properties>

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

More info at https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/cfgdepds005.htm