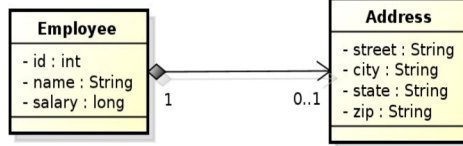


EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	STATE
	ZIP_CODE



@Embeddable

```

@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
  
```

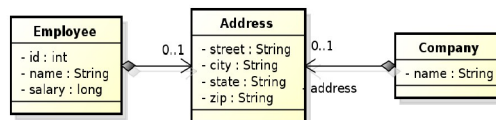
@Entity

```

public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    private Address address;
}
  
```

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	PROVINCE
	POSTAL_CODE

COMPANY	
PK	NAME
	STREET
	CITY
	STATE
	ZIP_CODE



@Entity

```

public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="state", column=@Column(name="PROVINCE")),
        @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
    })
    private Address address;
}
  
```

1 Advanced JPA

1.1 Embedded Objects

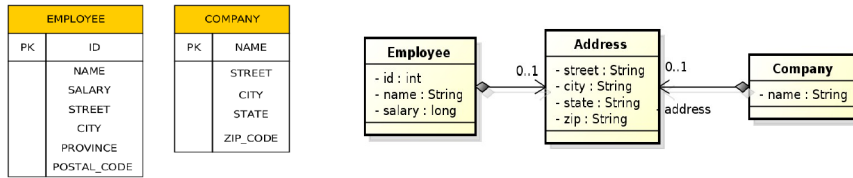
Embedded Objects

Embedded Objects with Attribute Overriding

Embedded Objects with Attribute Overriding II

1.2 Mapping Legacy Databases

Mapping Legacy Databases



```

@Entity
public class Company {
    @Id private String name;
    @Embedded
    private Address address;
}

```

1. One entity to many tables: `@SecondaryTable`, `@Column(table=...)`

```

@SecondaryTables({
    @SecondaryTable(name="ADDRESS")
})
public class Person {

    @Id
    private Long id;

    @Column(table="ADDRESS")
    private String city;

    // getters + setters
}

```

```

PERSON
=====
ID bigint PRIMARY KEY NOT NULL
HASNAME varchar(255)

```

```

ADDRESS
=====
ID bigint
    PRIMARY KEY NOT NULL
CITY varchar(255)
FOREIGN KEY (id)
    REFERENCES person (id)

```

2. Multiple entities to one table: *@Embedded*, *@EmbeddedId*, *@Embeddable*

```
@Entity
public class Person {
    @Id
    private Long id;
    private String hasName;

    @Embedded
    private Birth birth;
    // getters + setters
}
```

```
@Embeddable
public class Birth {
    private String hasPlace;

    @Temporal(value=TemporalType.DATE)
    private Date hasDateOfBirth;
    // getters + setters
}
```

```
PERSON
=====
ID bigint PRIMARY KEY NOT NULL
HASNAME varchar(255)
HASDATEOFBIRTH date
HASPLACE varchar(255)
```

Mapping Legacy Databases

1.3 Cascades

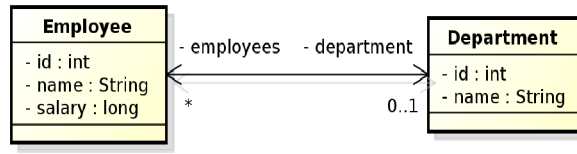
Cascading Operations

- Cascading allows to apply selected entity manager operations transitively to referenced entities
- Cascading types:
 - CascadeType.ALL
 - CascadeType.PERSIST
 - CascadeType.MERGE
 - CascadeType.REMOVE
 - CascadeType.REFRESH
 - CascadeType.DETACH

Cascade Persist

```
@Entity
public class Employee {
    // ...
    @ManyToOne(cascade=CascadeType.PERSIST)
    private Address address;
    // ...
}
```

```
Employee emp = new Employee();
emp.setName("Sherlock");
emp.setSalary(25000);
Address addr = new Address();
```



```

addr.setStreet("Baker Street");
addr.setCity("London");
addr.setCountry("British Empire");
emp.setAddress(addr);

// em.persist(addr); -- Not needed anymore
em.persist(emp);

```

Cascading and Bidirectional Relationships

```

final Department dept = new Department();
dept.setName("Accounting");
final Employee emp = new Employee();
emp.setName("Rob");
emp.setSalary(25000);
// @OneToMany(mappedBy="department", cascade = CascadeType.PERSIST)
dept.getEmployees().add(emp);

em.persist(dept);

```

```
emp.getDepartment() == null
```

2 Collection Mapping

Collection Mapping

- Collection-valued relationships
 - @OneToMany
 - @ManyToMany
- Element collections – since JPA 2.0
 - @ElementCollection
 - Collections of embeddables
 - Collections of basic types
 - Lists, Maps

Collection Mapping

Collection Mapping

```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // ...
    @ElementCollection(targetClass=VacationEntry.class);
    private Collection vacationBookings;

    @ElementCollection
    private Set<String> nickName;
    // ...
}

@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;

    @Column(name="DAYS")
    private int daysTaken;
    // ...
}

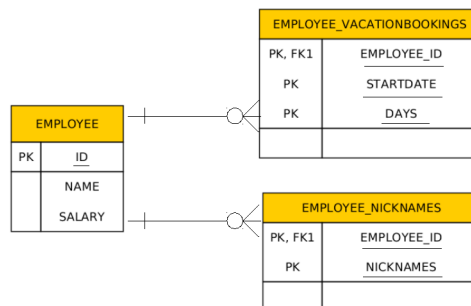
```

```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // ...
    @ElementCollection(targetClass=VacationEntry.class);
    private Collection vacationBookings;

    @ElementCollection
    private Set<String> nickName;
    // ...
}

```



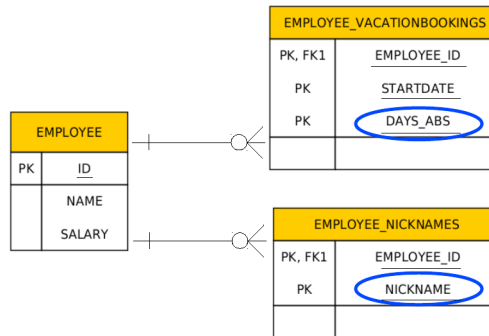
```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // ...
    @ElementCollection(targetClass = VacationEntry.class);
    @CollectionTable(
        name="VACATION",
        joinColumn=@JoinColumn(name="EMP_ID", column="EMPLOYEE_ID")
    @AttributeOverride(name="daysTaken", column="DAYS_ABS")
    private Collection vacationBookings;

    @ElementCollection
    @Column(name="NICKNAME")
    private Set<String> nickName;
    // ...
}

@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;
    // ...
    @Column(name="DAYS")
    private int daysTaken;
    // ...
}

```



Collection Mapping

2.1 Ordering

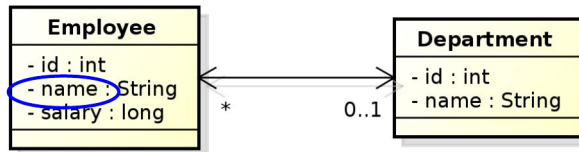
Collection Mapping – List Ordering

- Ordering by entity or element attribute
 - Ordering according to the state that exists in each entity or element in the list
 - @OrderBy
 - Multiple attributes can be specified
- Persistently ordered lists
 - Ordering by means of database column(s)
 - Typically, instances are in the order in which they were inserted into the table

Collection Mapping – List Ordering

Collection Mapping – List Ordering

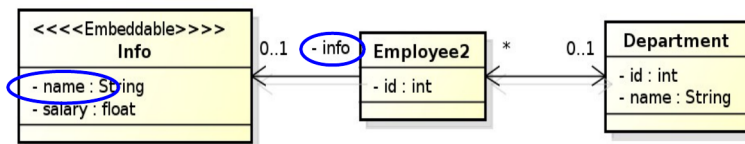
Collection Mapping – List Ordering



```

@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @OrderBy("name ASC")
    private List<Employee> employees;
    // ...
}
  
```

Figure 1: Ordering by Entity or Element Attribute



```

@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @OrderBy("info.name ASC")
    private List<Employee2> employees;
    // ...
}
  
```

Figure 2: Ordering by Entity or Element Attribute

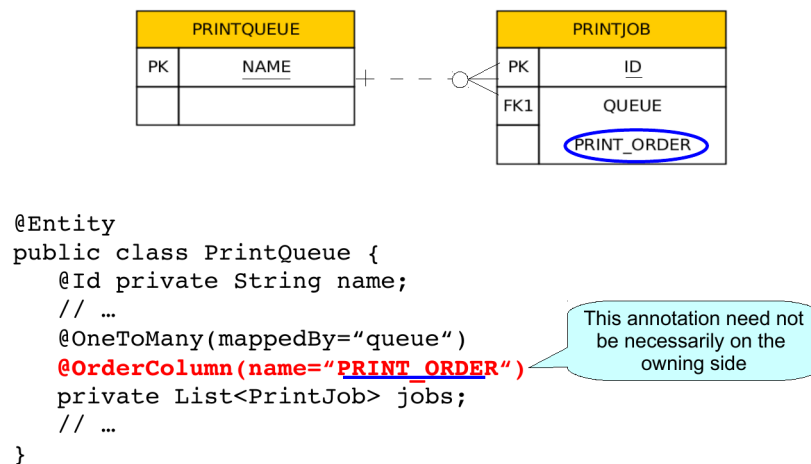


Figure 3: Persistently ordered lists

2.2 Maps

Collection Mapping – Maps

- Map is a collection that maps keys to values
- It cannot contain duplicate keys, each key can map to at most one value
- **Keys**
 - Basic types (incl. enums) – stored directly in the table being referred to
 - * Join table
 - * Collection table
 - * Target entity table
 - Embeddable types
 - * DTO
 - Entities – only foreign key is stored in the table
- **Values**
 - Entities – @OneToMany or @ManyToMany
 - Basic or embeddable types – mapped to an element collection

Collection Mapping

Collection Mapping


```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    @ElementCollection
    @CollectionTable(name="EMP_PHONE")
    @MapKeyColumn(name="PHONE_TYPE")
    @Column(name="PHONE_NUM")
    private Map<String, String> phoneNumbers;
    // ...
}

```

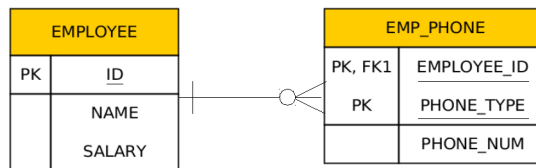


Figure 4: Basic type key – String

```

@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    @ElementCollection
    @CollectionTable(name="EMP_PHONE")
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name="PHONE_TYPE")
    @Column(name="PHONE_NUM")
    private Map<PhoneType, String> phoneNumbers;
    // ...
}

```

```

Public enum PhoneType {
    Home,
    Mobile,
    Work
}

```

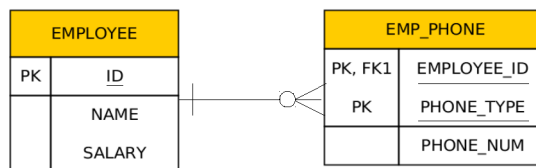


Figure 5: Basic type key – enumeration

```

@Entity
public class Department {
    @Id private int id;
    private String name;

    @OneToMany(mappedBy="department")
    @MapKeyColumn(name="CUB_ID")
    private Map<String, Employee> employeesByCubicle;
    // ...
}

```

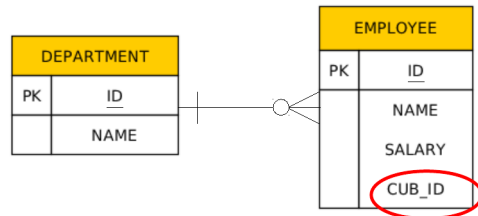


Figure 6: Basic type key – 1:N relationship using a Map with a String key

Collection Mapping

Collection Mapping

Collection Mapping

Collection Mapping

3 Compound and Shared Keys

3.1 Compound Primary Keys

Compound Primary Keys – @IdClass

```

public class EmployeeId implements Serializable {
    private String country;
    private Integer id;
}

```

```

@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id
    private String country;
    @Id
    @Column(name="EMP_ID")
    private Integer id;

    private String name;
    private Long salary;
    // ...
}

```

```

@Entity
public class Department {
    @Id private int id;
    private String name;

    @ManyToMany
    @JoinTable(name="DEPT_EMP",
        joinColumns=@JoinColumn(name="DEPT_ID"),
        inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    @MapKeyColumn(name="CUB_ID")
    private Map<String, Employee> employeesByCubicle;
    // ...
}

```

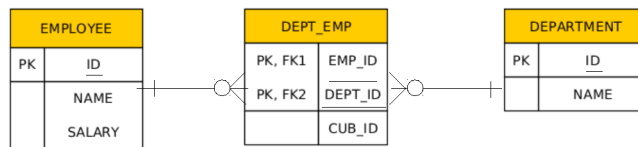


Figure 7: Basic type key – N:M relationship using a Map with a String key

```

@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @MapKey(name="id")
    private Map<Integer, Employee> employees;
    // ...
}

```

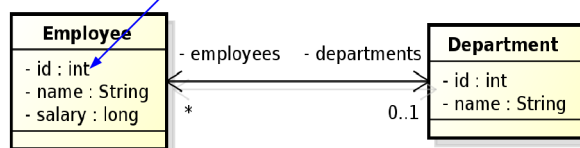


Figure 8: Entity attribute key

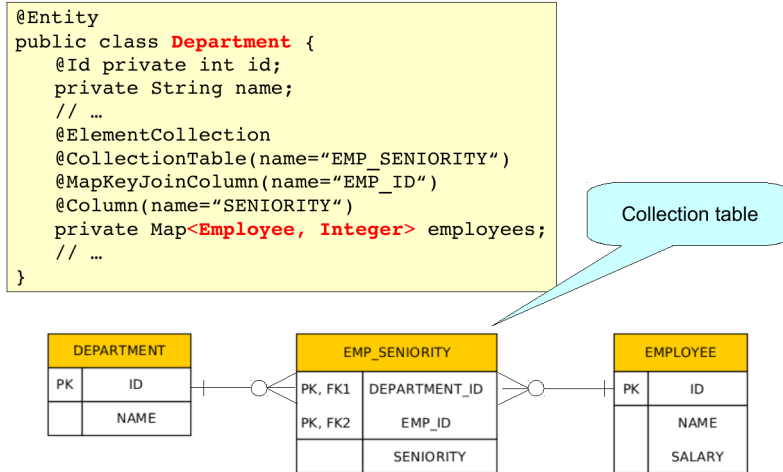


Figure 9: Entity as a key

EMPLOYEE	
PK	<u>COUNTRY</u>
PK	<u>EMP_ID</u>
	NAME
	SALARY

Usage:

```

em.createQuery("SELECT e FROM Employee e WHERE e.country = :country AND e.id = :id",
    Employee.class).getSingleResult();
// OR
EmployeeId id = new EmployeeId(country, id);
em.find(Employee.class, id);

```

Compound Primary Keys – @EmbeddedId

```

@Embeddable
public class EmployeeId implements Serializable {
    private String country;
    @Column(name="EMP_ID")
    private Integer id;
}

```

```

@Entity
public class Employee {
    @EmbeddedId
    private EmployeeId id;
}

```

EMPLOYEE	
PK	<u>COUNTRY</u>
PK	<u>EMP_ID</u>
	NAME
	SALARY

```

private String name;
private Long salary;
// ...
}

```

Usage:

```

em.createQuery("SELECT e FROM Employee e WHERE e.id.country = :country AND e.id.id = :id", Employee.class).getSingleResult();
// OR
EmployeeId id = new EmployeeId(country, id);
em.find(Employee.class, id);

```

3.2 Shared Primary Keys

Shared Primary Key

```

@Entity
public class Employee {
    @Id
    private Integer id;
    private String name;

    @OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)
    private EmployeeDetail detail;
    // ...
}

```

```

@Entity
public class EmployeeDetail {

    @Id
    private Integer id;

    @MapsId
    @OneToOne
    private Employee employee;
    // ...
}

```

- The primary key of EmployeeDetail is of the same type as Employee
 - Can be compound as well
- Relationship owner must be the entity with dependent id
- The relationship need not be bidirectional

Shared Primary Key

```

@Entity
public class Employee {
    @Id
    private Integer id;
    private String name;

    @OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)

```

EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY
FK1	MGR_COUNTRY
FK1	MGR_ID

```

@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id
    @Column name="EMP_ID")
    private int id;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="MGR_COUNTRY",
            referencedColumnName="COUNTRY"),
        @JoinColumn(name="MGR_ID",
            referencedColumnName="EMP_ID")
    })
    private Employee manager;

    @OneToMany(mappedBy="manager")
    private Collection<Employee> directs;
    // ...
}

```

```

private EmployeeDetail detail;
// ...
}

```

```

@Entity
public class EmployeeDetail {

    @Id
    @OneToOne
    private Employee employee;
    // ...
}

```

- The primary key in the dependent entity need not be explicitly mapped to an attribute

3.3 Compound Join Columns

Compound Join Columns

Compound Join Columns

4 Various Attributes and Access Types

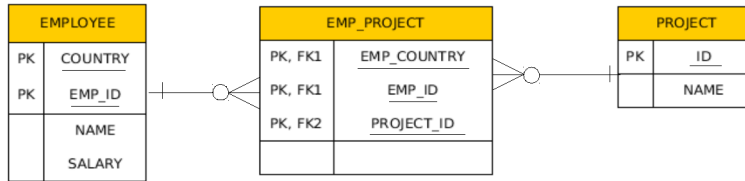
Read-only Mapping and Optionality

```

@Entity
public class EmployeeView {
    @Id
    private Integer id

    @Column(insertable=false, updatable=false)

```



```

@Entity
@IdClass(EmployeeId.class)
public class Employee
{
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;
    @ManyToOne
    @JoinTable(
        name="EMP_PROJECT",
        joinColumns={
            @JoinColumn(name="EMP_COUNTRY", referencedColumnName="COUNTRY"),
            @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")},
        inverseJoinColumns=@JoinColumn(name="PROJECT_ID"))
    private Collection<Project> projects;
}
    
```

```

private String name;

@ManyToOne(optional = false)
@JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
private Department department;
// ...
}
    
```

- @Column attributes are written also into DDL
- Checked on commit

Overriding Mapping

- @AttributeOverride allows to override mapping of basic and ID attributes
- May be applied also to element collections
- Allows to override mapping inherited from a mapped superclass or from an embeddable class

Overriding Mapping – Example

```

@Embeddable
public class Address {

    protected String street;
    protected String city;
    protected String state;

    @Embedded
    protected Zipcode zipcode;
}
    
```

```
@Embeddable
public class Zipcode {

    protected String zip;
    protected String plusFour;
}
```

```
@Entity
public class Customer {

    @Id
    protected Integer id;
    protected String name;

    @AttributeOverrides({
        @AttributeOverride(name="state",
            column=@Column(name="ADDR_STATE")),
        @AttributeOverride(name="zipcode.zip",
            column=@Column(name="ADDR_ZIP"))
    })
    @Embedded
    protected Address address;
    // ...
}
```

Access types – Field access

```
@Entity
public class EmployeeView {
    @Id
    private Integer id;
    private String name;
    // ...
    public Integer getId() {return id;}
    public void setId(Integer id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
}
```

- Provider will get and set attribute values directly via fields
- *Getters/setters will not be invoked*

Access types – Property access

```
@Entity
public class EmployeeView {
    private Integer id;
    private String name;
    // ...
    @Id
    public Integer getId() {return id;}
    public void setId(Integer id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
}
```


JPQL very similar to SQL (especially in JPA 2.0)

```
SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND p.type = 'CELL'
```

Conditions do not stick on values of database columns, but on entities and their properties.

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
```

- Annotate getter or setter
- Provider will get and set attribute values by invoking getters and setters

Access types – Mixed access

- Field access with property access can be combined within the same entity or within the same entity hierarchy
- `@Access` defines default access mode for entity
 - May be overridden for subclasses

5 Queries

Queries

- **JPQL** – Java Persistence Query Language
- **SQL** – native queries
- **Criteria API** – programmatic query API

5.1 JPQL

JPQL

JPQL – Query Parameters

JPQL – Using Parameters

JPQL – Named Queries

- positional

```
SELECT e
FROM Employee e
WHERE e.department = ?1 AND e.salary > ?2
```

- named

```
SELECT e
FROM Employee e
WHERE e.department = :dept AND salary > :base
```

```
String QUERY = "SELECT e.salary FROM Employee e " +
               "WHERE e.department.name = :deptName " +
               "AND e.name = :empName";

public long queryEmpSalary(String deptName, String empName) {
    return em.createQuery(QUERY, Long.class)
        .setParameter("deptName", deptName)
        .setParameter("empName", empName)
        .getSingleResult();
}
```

```
@NamedQuery(name="Employee.findByName",
             query="SELECT e FROM Employee e " +
                 "WHERE e.name = :name")
```

```
public Employee findEmployeeByName(String name) {
    return em.createNamedQuery("Employee.findByName",
                               Employee.class)
        .setParameter("name", name)
        .getSingleResult();
}
```

Modifications of entities not only by `em.persist()` or `em.remove()`;

```
em.createQuery("UPDATE Employee e SET e.manager = ?1 " +  
              "WHERE e.department = ?2")  
    .setParameter(1, manager)  
    .setParameter(2, dept)  
    .executeUpdate();
```

```
em.createQuery("DELETE FROM Project p " +  
              "WHERE p.employees IS EMPTY")  
    .executeUpdate();
```

If REMOVE cascade option is set for a relationship, cascading remove occurs.

Native SQL update and delete operations should not be applied to tables mapped by an entity (transaction, cascading).

```
@NamedNativeQuery(  
    name="getStructureReportingTo",  
    query = "SELECT emp_id, name, salary, manager_id," +  
            "dept_id, address_id " +  
            "FROM emp ",  
    resultClass = Employee.class  
)
```

Mapping is straightforward

JPQL – Bulk Updates

5.2 Native Queries

Native (SQL) Queries

Native (SQL) Queries

Native (SQL) Queries

5.3 Criteria API

Criteria API

```
SELECT p FROM Product p WHERE p.name LIKE '%p%'
```

Static Metamodel

```

@NamedNativeQuery(
    name="getEmployeeAddress",
    query = "SELECT emp_id, name, salary, manager_id," +
            "dept_id, address_id, id, street, city, " +
            "state, zip " +
            "FROM emp JOIN address "
            "ON emp.address_id = address.id)"
)

```

Mapping less straightforward

```

@SqlResultSetMapping(
    name="EmployeeWithAddress",
    entities={@EntityResult(entityClass=Employee.class),
              @EntityResult(entityClass=Address.class)}
)

```

```

Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
    "o.quantity AS order_quantity, " +
    "o.item AS order_item, " +
    "i.name AS item_name, " +
    "FROM Order o, Item i " +
    "WHERE (order_quantity > 25) AND (order_item = i.id)",
    "OrderResults");

@SqlResultSetMapping(name="OrderResults",
    entities={
        @EntityResult(entityClass=com.acme.Order.class,
            fields={
                @FieldResult(name="id", column="order_id"),
                @FieldResult(name="quantity",
                    column="order_quantity"),
                @FieldResult(name="item",
                    column="order_item")
            }
        ),
        @EntityResult(entityClass=com.acme.Item.class,
            fields={
                @FieldResult(name="name", column="item_name")
            }
        )
    },
    columns={
        @ColumnResult(name="item_name")
    }
)

```

```

private long pageSize    = 800;
private long currentPage = 0;

public List getCurrentResults() {
    return em.createNamedQuery("Employee.findByDept",
                               Employee.class)
           .setFirstResult(currentPage * pageSize)
           .setMaxResults(pageSize)
           .getResultList();
}

public void next() {
    currentPage++;
}

```

```

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Product> r = cq.from(Product.class);
cq.where(
    cb.like(
        r.get(Product_.name)
        , "%p%"
    )
);
return em.createQuery(cq).getResultList();

```

Metamodel

```

Metamodel m = em.getMetamodel();
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
Root<Product> r = cq.from(Product.class);
cq.where(
    cb.like(
        r.get(
            m.entity(Product.class)
            .getSingularAttribute("name",
                                   String.class)
        , "%p%"
    )
);
return em.createQuery(cq).getResultList();

```

Query API – Pagination

6 Beyond JPA

Graph Databases

- Comparing to RDBMS – relation types are **first-class citizens**
- Suitable for relaxed data schemas
- Suitable for analytics using graph algorithms

- E.g. Neo4j

Spring Data Neo4j

Model:

```
import org.neo4j.ogm.annotation.*;
@NodeEntity
public class Person {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @Relationship(type = "ACTED_IN")
    private List<Movie> movies = new ArrayList<>();
}
```

Repository:

```
@RepositoryRestResource(collectionResourceRel = "movies", path = "movies")
public interface MovieRepository extends Neo4jRepository<Movie, Long> {
    @Query("MATCH (m:Movie) <-[r:ACTED_IN]-(a:Person) RETURN m,r,a")
    List<Person> getActors();
}
```

RDF Triple Stores

- Comparing to RDBMS – relation types are **first-class citizens**
- Suitable for relaxed data schemas
- Suitable for Linked Data, Semantic Web, ontologies
- E.g. RDF4J, Virtuoso, Fuseki, Blazegraph, GraphDB, ...

JOPA is a library for accesing triples using Java objects.

JOPA

Model:

```
import cz.cvut.kbss.jopa.model.annotations.*;

@OWLClass(iri = "http://example.org/ontology/student")
public class Student implements Serializable {

    @Id
    private URI uri;

    @OWLDataProperty(iri = "http://example.org/ontology/email")
    private String email;
}
```

DAO similar to JPA EntityManager. See <https://github.com/kbss-cvut/jopa>

Resources

- JSR 338 Java Persistence 2.2 Final Release <https://jcp.org/en/jsr/detail?id=338>
- WikiBooks https://en.wikibooks.org/wiki/Java_Persistence

The End

Thank You