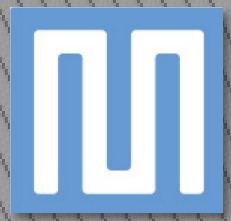
JPA

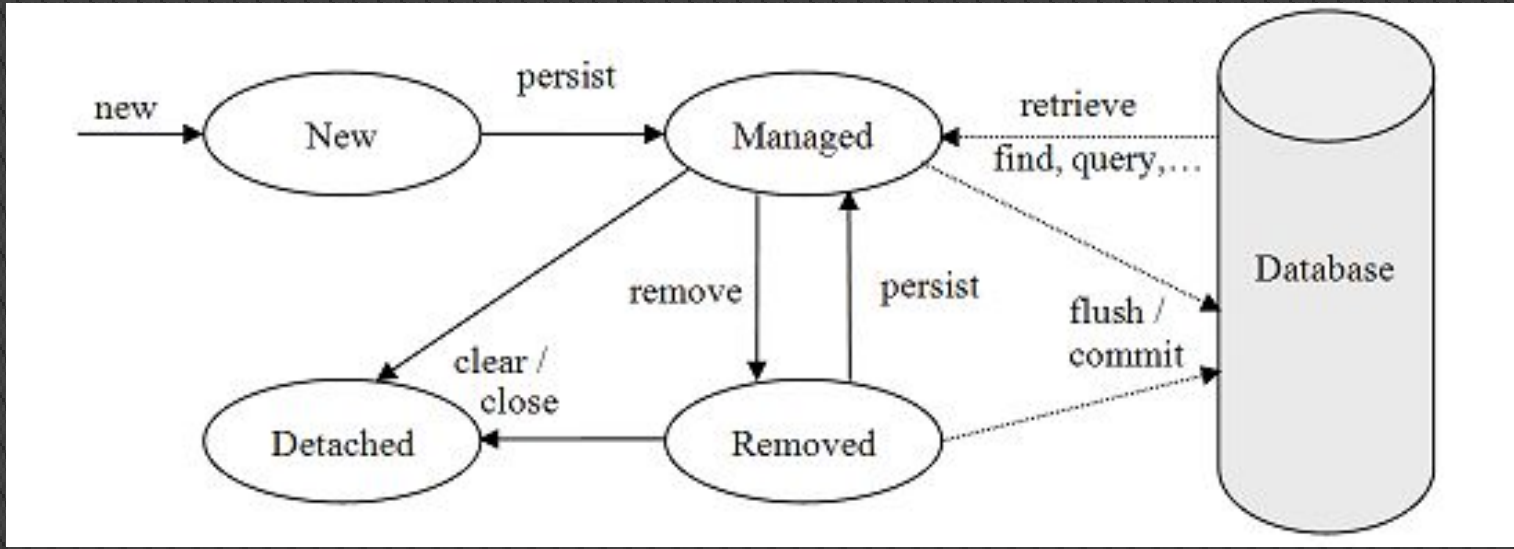
Queries



Tartalom

- Queries
- JPQL
- Criteria
- Transaction management

Entity lifecycle



EntityManager methods

- **<T> T find**(java.lang.Class<T> entityClass, java.lang.Object primaryKey) – *„SELECT * FROM customer where id=?”*
- **<T> T getReference**(java.lang.Class<T> entityClass, java.lang.Object primaryKey) – like find but returns a proxy object–> lazy initialization
- **void persist**(java.lang.Object entity) – *„insert into …”* – exception when primary key already exists
- **<T> T merge**(T entity) – *update vagy insert* – if primary key already exists, SQL UPDATE, if not, INSERT
- **remove**(java.lang.Object entity) – DELETE

EntityManager methods

- **flush**() – inserts into DB everything from Persistence Context
- **detach**(java.lang.Object entity) – removes entity from persistence context

Join Fetch

Defines that after loading the entity also the associated relations will be loaded

- **LAZY vs EAGER fetch**
- **SELECT c form Customer c LEFT JOIN FETCH c.address a where a.street like ‚Madison Avenue';**

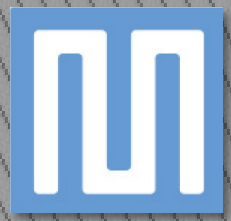
Queries

- All the queries can be called through the entity manager:
- Find by primary key:

<T> T find(Class<T> entityClass,Object primaryKey)

- Lekérdezés teljesen **dinamikusan**:
  - JPQL (EJB-QL) : public Query createQuery(String ejbqlString)
    - SQL-like query but returns with entities
    - Pl. **SELECT c from Customer c WHERE c.username=?1**
  - native SQL: public Query createNativeQuery(String sqlString)
- **Statikusan** definiált, névvel azonosítható lekérdezés:
  public Query createNamedQuery(String nameOfQuery)
  a lekérdezés @NamedQueries-ben van definiálva az entitás osztálynál

JPQL



The Java Persistence Query Language

- sql queries which are independent from the database (abstraction)
- portable
- sql like syntax but returns with entities

JPQL queries (dynamic)

- SELECT:

```
TypedQuery<Customer> query= entityManager.createQuery( "SELECT c FROM Customer c WHERE c.name LIKE :custName”, Customer.class ) ;
query.setParameter("custName", name) .setMaxResults(10)
 .getResultList();     –> Customer listát ad vissza
.getSingleResult(); –> egy példányt ad vissza, ha több találat, akkor kivétel
```

- UPDATE:

```
entityManager.createQuery(”UPDATE CUSTOMER SET NAME =‚Joe’ where id=:custId)”
.setParameter("custName", 3)
.executeQuery();
```

JPQL queries

Query methods:

- setParameter: name or index based (:parameter vagy ?1)
- getSingleResult() – if there are more hits, exception
- getResultList() – more results
- executeUpdate() – called after update/delete

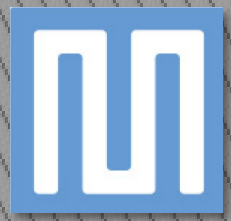
Named Queries (static)

- Defined with an annotation on entity class:

```
@Entity
@Table(name=customer”
@NamedQuery( name="findAllCustomersWithName", query="SELECT c FROM
Customer c WHERE c.name LIKE :custName" )
public class Customer{
}
```

```
TypedQuery<Customer> query = entityManager.
createNamedQuery("findAllCustomersWithName”,
Customer.class) ;
qurey.setParameter("custName", "Smith")
.getResultList();
```

Criteria API



Criteria API

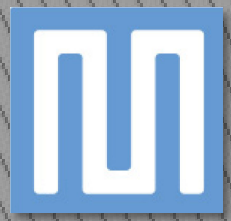
- Framework – queries can be defined in an object–oriented way
- object oriented, typesafe alternative to JPQL :

*= >SELECT * FROM customer where id=3*

Criteria API példa

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Customer> cq =
        cb.createQuery(Customer.class);
        Root<Customer> customer= cq.from(Customer.class);
        cq.select(customer);
        cq.where(cb.equal(customer.get(„lastname"),
        ”User1”);
        TypedQuery<Customer> query =
                em.createQuery(cq);
List <Customer> rows = query.getResultList();
```

Transaction management



Transaction management principles (ACID)

- **Atomicity**: if one part of the transaction fails then the entire transaction fails
- Consistency: any transaction will bring the DB from a valid state to another valid state
- **Izoláció**: concurrent clients will not see each others unfinished changes
- **Duribality**: no changes will be lost due to hardware or network errors

Transaction types

- **Bean Managed**
- Implementing class is responsible for starting, closing and rolling back the transactions
- Transaction management is responsibility of programmer
- **Container Managed**
- Wrapper generated by container is responsible for opening/closing the transactions
- Rollback can be also done by programmer (container automatically does rollback after RuntimeException or @ApplicationException(rollback=true) esetén)

Transaction attributes

- **@TransactionManagement** annotation (Container or bean)

```
@TransactionMnagement(TransactionManagementType.CONTAINER)
```

- **@TransactionAttribute**: (@Required default)

```
@TransactionAttribute(value = TransactionAttributeType.REQUIRES_NEW)
```

Transaction attributes

- If EJB calls T2 transaction, T1 calling transaction will be stopped until T2 finishes
- result of T2 does not influence T1

Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	TransactionRequiredException
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	RemoteException

Java Transaction API (JTA)

- Transaction management through this API
- javax.transaction.UserTransaction interface
- Important methods:
  - begin(),
  - commit(),
  - rollback(),
  - setRollbackOnly()

Rollback

Container automatically roles back transaction:

- Runtime exception
- Checked exceptions annotated with **@ApplicationException(rollback=true)**

Manual rollback (after catching an error):

- ejbContext.setRollbackOnly() called

A UserTransaction interface

```
    @Resource
    EJBContext ejbContext;


    private void method(){
//bean által kezdeményezett tranzakció indítása
        ejbContext.getUserTransaction().begin();

//bean által kezdeményezett tranzakció lezárása/visszagörgetése
        ejbContext.getUserTransaction().commit()/rollback();

//Konténer által kezdeményezett tranzakció visszagörgetésének
kikényszerítése:
     ejbContext.setRollbackOnly();


}
```