**ORM**

**Arche**:

**Entity**:

An entity models a database table. The identifier uniquely identifies each row in that table. By default, the name of the table is assumed to be the same as the name of the entity. Follow JavaBean definition (no-argument constructor, getter and setter) and have identifier.

Mapping entity: @Entity annotation.

@Entity

@Table( catalog = "CRM", schema = "purchasing", name = "t\_simple" )

public class Simple {

@Id

@GeneratedValue

public Integer id;

}

* Hibernate guarantees equivalence of persistent identity (database row) and Java identity inside a particular session scope. So if we ask a Hibernate Session to load that specific Person multiple times we will actually get back the same instance.

Session session=...;

Person p1 = session.get( Person.class,1 );

Person p2 = session.get( Person.class,1 );

// this evaluates to true

assert p1==p2;

* Defining equals and hashCode: do not only use Id field if Id is auto-generated, because before flushing Id is not set yet. Use combination of Id and unique field (natural Id).
* Access strategies: the placement of the @Id annotation gives the default access strategy. When placed on a field, Hibernate will assume field-based access. Place on the identifier getter, Hibernate will use property-based access.
* Identifiers: use Primary Key generally. There are type restrictions as identifiers. Can be either assigned or generated. Can be single or composited.
  + Generated identifiers: only integer types will be supported. Hibernate will generate the value when the save/persist occurs. Use @GeneratedValue annotation.
  + Generator type: specify type in @GeneratedValue annotation, then specify generator and its configuration. Can be IDENTITY, SEQUENCE, TABLE. SEQUENCE grammar as follows:

@Id

@GeneratedValue( generation = SEQUENCE, name = "my\_sequence" )

@SequenceGenerator( name = "my\_sequence", schema = "globals", allocationSize = 30 )

public Integer id;

* @GenericGenerator can also be used that allows any generator implementation. See the following grammar.
* @Id
* @GeneratedValue(
* strategy = GenerationType.SEQUENCE,
* generator = "product\_generator"
* )
* @GenericGenerator(
* name = "product\_generator",
* strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
* parameters = {
* @Parameter(name = "sequence\_name", value = "product\_sequence"),
* @Parameter(name = "initial\_value", value = "1"),
* @Parameter(name = "increment\_size", value = "3"),
* @Parameter(name = "optimizer", value = "pooled-lo")
* }
* )
* private Long id;

**Association**

@ManyToOne: foreign key.

@ManyToOne

@JoinColumn(name = "person\_id",

foreignKey = @ForeignKey(name = "PERSON\_ID\_FK")

)

private Person person;

@OneToMany

Cascade: from parent (being referenced, the entity with OneToMany) to child (referencing parent, the entity with ManyToOne if bidirectional). orphanRemoval: determines whether standalone child without referencing any parent should be deleted.

* Unidirectional: use a link table (controlled by owning side) between two joining entities.
  + The unidirectional associations are not very efficient when it comes to removing child entities. In this particular example, upon flushing the persistence context, Hibernate deletes all database child entries of a given parent and reinserts the ones that are still found in the in-memory persistence context. (traverse link table to locate the relation rows (usually not needed because primary key either single or composite is automatically indexed) or use index (can query first index column on a composite query, as index uses lexicographical comparison) to locate relation rows, delete all rows with given parent, then insert all persistent children with given parent)
  + On the other hand, a bidirectional @OneToMany association is much more efficient because the child entity controls the association. (find child entity by indexed Id, then just nullify foreign key)
* Bidirectional: The bidirectional @OneToMany association also requires a @ManyToOne association on the child side. Behind the scenes, the relational database has only one foreign key for this relationship.
  + Every bidirectional association must have one owning side only, the other one being referred to as the inverse (or the mappedBy) side. Define adding/removing child methods in only one side (mappedBy side in Many to One relationship), taking care of each entity’s list adding/removing and foreign key setting.
    - Bidirectional Many to One relationship owning side: @ManyToOne, mappedBy side: @OneToMany
  + Unlike the unidirectional @OneToMany, the bidirectional association is much more efficient when managing the collection persistence state. Every element removal only requires a single update (in which the foreign key column is set to NULL), and, if the child entity lifecycle is bound to its owning parent so that the child cannot exist without its parent, then we can annotate the association with the orphan-removal attribute and disassociating the child will trigger a delete statement on the actual child table row as well.

@ OneToOne

* Unidirectional: foreign key with some unique constraints
* Bidirectional: it is like bidirectional @OneToMany. Still have the concept of child (owning side) and parent (mappedBy side) to determine which entity should contain foreign key and determine cascade direction. Hibernate enforces the unique constraint upon fetching the child-side.

@ ManyToMany

Requires a link table that joins two entities.

For @ManyToMany associations, the REMOVE entity state transition doesn’t make sense to be cascaded, because the children may reference other parents.

* Unidirectional: When an entity is removed from the @ManyToMany collection, Hibernate simply deletes the joining record in the link table. Unfortunately, like unidirectional OneToMany who also uses link tables, this operation requires removing all entries associated with a given parent and recreating the ones that are listed in the current running persistent context.
* Bidirectional
  + ManyToMany: when removing parent, still need to removing all children first.
  + Two bidirectional OneToMany: explicitly use a link entity to avoid removing all children, as we can manipulate with the link table and explicitly delete that row with given parent and child Id in the link table.

**Persist Entities**

entityManager.persist( person ); //persist entities

entityManager.remove( person ); //delete entities

book.setAuthor( entityManager.getReference( Person.class, personId ) ); //obtain entity reference without initializing

Person person = entityManager.find( Person.class, personId ); //obtain entity with data initialized; null is return if no matching

person.setName("John Doe");

entityManager.flush(); //modify persistent state; any changes will be automatically detected and persisted when the persistence context is flushed

entityManager.refresh( person ); //refresh/reload entity state; fetch data from database, so be careful when cascading transient entities

boolean contained = entityManager.contains( person ); //verify managed state

PersistenceUtil persistenceUnitUtil = Persistence.getPersistenceUtil();

boolean personInitialized = persistenceUnitUtil.isLoaded( person ); //verify laziness

entityManager.detach( person ); //detach/evict entity

entityManager.clear(); //clear persistence context

**Flushing**

Flushing is the process of synchronizing the state of the persistence context with the underlying database. The flush operation takes every entity state change and translates it to an INSERT, UPDATE or DELETE statement.

The flushing strategy is given by the flushMode.

* Auto (default): circumstances when flush is triggered:
  + prior to committing a Transaction
  + prior to executing a JPQL/HQL query that overlaps with the queued entity actions; if query doesn’t overlap entities, then no flush
  + before executing any native SQL query that has no registered synchronization

**HQL and JPQL**

JPQL is a subset of HQL.

* Obtaining Query reference: EntityManager#createQuery or #createNamedQuery methods.

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name"

);

TypedQuery<Person> typedQuery = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name", Person.class

);

@NamedQueries(

@NamedQuery(

name = "get\_person\_by\_name",

query = "select p from Person p where name = :name"

)

)

Query query = entityManager.createNamedQuery( "get\_person\_by\_name" );

TypedQuery<Person> typedQuery = entityManager.createNamedQuery(

"get\_person\_by\_name", Person.class

);

* Basic Query usage
  + Many of the settings controlling the execution of the query are defined as hints.
* Query query = entityManager.createQuery(
* "select p " +
* "from Person p " +
* "where p.name like :name" )
* // timeout - in milliseconds
* .setHint( "javax.persistence.query.timeout", 2000 )
* // flush only at commit time
* .setFlushMode( FlushModeType.COMMIT );
  + Parameter binding
    - Name binding

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" );

* + - Position binding

Query query = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like ?1" )

.setParameter( 1, "J%" );

* Get results: Query#getResultList() and Query#getSingleResult()

List<Person> persons = entityManager.createQuery(

"select p " +

"from Person p " +

"where p.name like :name" )

.setParameter( "name", "J%" )

.getResultList();

* Statements: SELECT, UPDATE and DELETE types.
  + SELECT statement

select\_statement :: =

[select\_clause]

from\_clause

[where\_clause]

[groupby\_clause]

[having\_clause]

[orderby\_clause]

* + UPDATE statement

int updatedEntities = entityManager.createQuery(

"update Person p " +

"set p.name = :newName " +

"where p.name = :oldName" )

.setParameter( "oldName", oldName )

.setParameter( "newName", newName )

.executeUpdate();

* + DELETE statement

delete\_statement ::=

delete\_clause [where\_clause]

delete\_clause ::=

DELETE FROM entity\_name [[AS] identification\_variable]

* Explicit joins: happen in FROM clause, can be either inner (default) or left outer.

List<Person> persons = entityManager.createQuery(

"select distinct pr " +

"from Person pr " +

"left join pr.phones ph " +

"where ph is null " +

" or ph.type = :phoneType", Person.class )

.setParameter( "phoneType", PhoneType.LAND\_LINE )

.getResultList();

List<Object[]> personsAndPhones = entityManager.createQuery(

"select pr.name, ph.number " +

"from Person pr " +

"left join pr.phones ph on ph.type = :phoneType " )

.setParameter( "phoneType", PhoneType.LAND\_LINE )

.getResultList();

* Fetch join: override the laziness of the joined association, use for performance tuning. See <https://stackoverflow.com/questions/17431312/difference-between-join-and-join-fetch-in-hibernate>
* Implicit join: WHERE clause, already treated as inner join.

List<Phone> phones = entityManager.createQuery(

"select ph " +

"from Phone ph " +

"where ph.person.address = :address ", Phone.class )

.setParameter( "address", address )

.getResultList();

// same as

List<Phone> phones = entityManager.createQuery(

"select ph " +

"from Phone ph " +

"join ph.person pr " +

"where pr.address = :address ", Phone.class )

.setParameter( "address", address)

.getResultList();

* DISTINCT: can be used on attributes or entities.
* Literals
  + String: single quote
  + Number: L denotes a long, D denotes a double, F denotes a float
* Aggregation: same as SQL.

List<Object[]> callCount = entityManager.createQuery(

"select p.number, count(c) " +

"from Call c " +

"join c.phone p " +

"group by p.number", Object[].class )

.getResultList();