University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

# EAF Zusammenfassung

22. November 2013

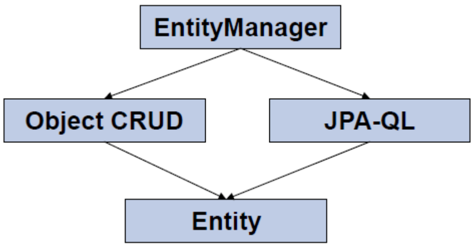
## Inhaltsverzeichnis

Institute of computer science,
www.fhnw.ch

# 1 Spring Configuration

# 2 Java Persistence API (JPA)

## 2.1 General Info



**JPA Components**

- EntityManager provides access to the objects (similar to a DAO)
  - find / persist / update / remove
  - Query API and JPA-QL
- Controlled Lifecycle

**Entity Metadata**

- Form:
  - Annotations
  - XML Files
- Configuration by Exception

## 2.2 Entity Annotations

Listing 1: Entity

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable {
        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        private int id;
        private String firstName;
        @Column(name="NAME")
        private String lastName;
        protected Customer(){}
        public Customer(String firstName, String lastName){
                this.firstName = firstName;
                this.lastName = lastName;
                // id is not set!
        }
        public int getId() { return this.id; } // read only
        public String getFirstName() { return this.firstName; }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getLastName() { return this.lastName; }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
```

}

Folie 10 - 19, Foliensatz JPA1.pdf sind Spezifikationen und Anforderungen an Entity Klasse

#### 2.2.1 Primary Keys: Generation

- Assigned
  - Primary keys may be assigned by application, i.e. no key generation
    - E.g. language table: Primary Key is the ISO country code
- Identity
  - Auto increment supported by some DBs
- Sequence
  - Some DBs support sequences which generate unique values (e.g. Oracle, PostgreSql)
- Table
  - Primary keys are stored in a separate PK table

**Performance Comparison**

- 10'000 insert statements
- AUTO (Identity) 7534 msec
- TABLE (allocationSize = 32768) 2244 msec
- TABLE (allocationSize = 1) 9612 msec
- TABLE (allocationSize = 2) 7429 msec
- TABLE (allocationSize = 4) 5856 msec
- ASSIGNED (user defined) 1959 msec

#### 2.2.2 Associations

- OneToOne, owning side contains the foreign key
- OneToMany
- ManyToOne
- ManyToMany, either side may be the owning side
- Owning side determines the updates to the relationships in the database

**Relationships can be:**

- Unidirectional
  - Has an owning side
- Bidirectional
  - Has an owning side
  - Has an inverse side

**ManyToOne: User - Rental: bidirectional**
Bei bidirektionalen Beziehungen sind die Many-Side die owning Side.
Example:

Listing 2: Example

```
@Entity
public class Rental implements Serializable { @Id
        private int id;
        @ManyToOne // Rental is the owner of the relationship
        @JoinColumn(name="USER_FK") // optional
        // JPA macht bei OneToMany und ManyToOne eine Zwischentable.
        // Mit Keyword JoinColumn wird dies unterbunden. => Foreign Key = Owning side
        private User user;
        public Rental(){}
        public user getUser() { return user; }
        public void setUser (Customer user) {
                this.user = user;
        }
        public int getId() { return id; }
        public void setId(int id) { this.id = id; }
}
```

Inverse Side Example:

Listing 3: InverseSideExample

```
@Entity
public class User implements Serializable {
        ...
        @OneToMany(mappedBy="user") private Collection<Rental> rentals;
        // this is the inverse side of the relationship
        public Collection<Rental> getRentals() {
                return rentals;
        }
        public void setRentals(Collection<Rental> rentals) {
                this.rentals = rentals;
        }
}
```

**Only references from $n$ to $1$ are persisted!**

Listing 4: OneToMany bidirectional

```
em.getTransaction().begin();
Customer c = new Customer();
Order o1 = new Order();
Order o2 = new Order();
List<Order> orders = new LinkedList<Order>();
orders.add(o1); orders.add(o2);
c.setOrders(orders);
em.persist(c);
em.getTransaction().commit();
```

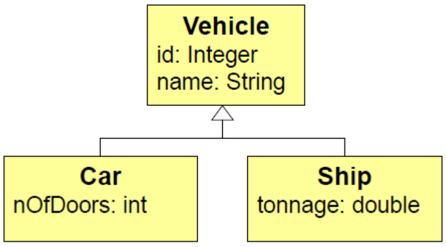
- the two orders are stored in the DB (due to the cascade=PERSIST)
- the associations are NOT persisted!!!

**OneToOne / OneToMany / ManyToOne / ManyToMany - Attributes**

- fetch EAGER / LAZY
  - determines fetch type
- cascade MERGE / PERSIST / REFRESH / DETACH REMOVE / ALL
  - determines cascade operation
- mappedBy String, not for ManyToOne
  - used for bidirectional associations (on the inverse side)

- optional boolean, only for OneToOne/ManyToOne
  - determines, whether null is possible (0..1)
- orphanRemoval boolean, only for OneToOne/OneToMany

**Example:**
@OneToOne(cascade=CascadeType.PERSIST, CascadeType.REMOVE)
Slides 20 & 21, Foliensatz JPA2_ Slides.pdf Cascading und Fetch Types werden erklärt
Im Zusammenhang mit Fetch Types ist Lazy Loading Problem erklärt im Foliensatz 00_ JPA2_ Arbeitsblatt_ Besprechung.pdf **Inheritance**



### 2.2.3 Examples

Listing 5: Example

```
@Entity @Table(name=”EMP”)
public class Employee {
        public enum Type {FULL, PART_TIME};
        protected Employee(){}
        public Employee(String name, Type type){
                this.name = name; this.type = type;
        }
        @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
        long id;
        @Enumerated(EnumType.STRING)
        @Column(name=”EMP_TYPE”, nullable=false)
        Type type;
        @Lob byte[] picture;
        String name;
}

create table EMP (
        id bigint generated by default as identity (start with 1),
        picture longvarbinary,
        EMP_TYPE varchar(255) not null,
        primary key (id)
)
```

- Representation
  - SINGLE TABLE (default)
  - TABLE_ PER_ CLASS (per concrete class a table is defined)
  - JOINED (one table per class)
- Specification
  - Inheritance type can be specified on root entity using @Inheritance annotation

**Single Table Example (@Inheritance(strategy=InheritanceType.SINGLE_TABLE))**

| DTYPE | ID | NAME | NOFDOORS | TONNAGE |
|---|---|---|---|---|
| Car | 1 | VW Sharan | 5 | (null) |
| Car | 2 | Smart | 2 | (null) |
| Ship | 3 | Queen Mary | (null) | 76000 |

**Disadvantages**

- All fields added in subclasses must be nullable
- Foreign keys can only refer to the base class

**Joined Table Example (@Inheritance(strategy=InheritanceType.JOINED))**

| ID | NAME |
|---|---|
| 1 | VW Sharan |
| 2 | Smart |
| 3 | Queen Mary |

| ID | NOFDOORS |
|---|---|
| 1 | 5 |
| 2 | 2 |

| ID | TONNAGE |
|---|---|
| 3 | 76000 |

- Advantages:
  - normalized schema, a database table for each class
  - All fields can be defined with not null conditions
  - Foreign-key references to concrete subclasses are possible
- Disadvantages:
  - Each entity access has to go over several tables

**TABLE_ PER_ CLASS Table Example (@Inheritance(strategy=InheritanceType.TABLE_ PER_ CLASS))**

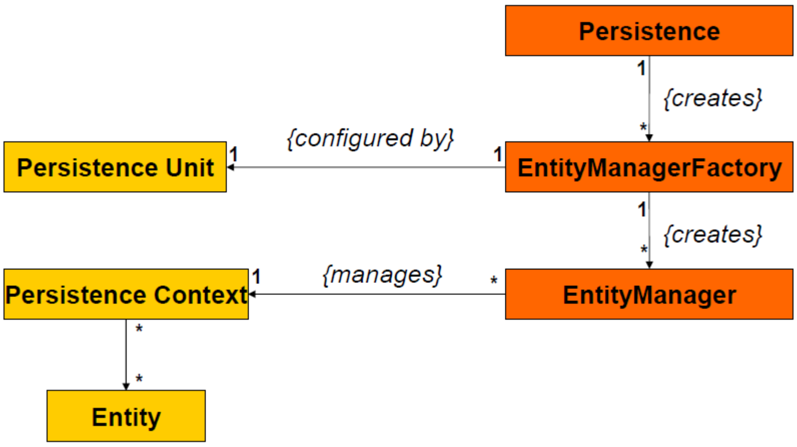
| ID | NAME | NOFDOORS |
|---|---|---|
| 1 | VW Sharan | 5 |
| 2 | Smart | 2 |

| ID | NAME | TONNAGE |
|---|---|---|
| 3 | Queen Mary | 76000 |

- Advantages:
  - Non-null constraints can be defined
  - Foreign-key references to concrete subclasses are possible (but not to abstract base classes)
- Disadvantages:
  - Polymorphic queries need to access several tables
  - Identity generator cannot be used
  - Not required by JPA 2.0-Spec (but provided by Hibernate)

## 2.3 Entity Manager

Entity Manager API auf Seite 8-11, Foliensatz JPA2_ Slides.pdf



**Only one Java instance with the same persistent identity may exist in a Persistence Context**

Listing 6: Access to Entity Manager

```
// J2SE: using factory
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("movierental");
```

```
EntityManager em = emf.createEntityManager();

// J2EE: injected by container
@PersistenceUnit(name="movierental")
EntityManagerFactory emf = null;

@PersistenceContext(unitName="movierental")
private EntityManager em;
// => container managed entity manager
```

### 2.3.1 Queries

- JPQL
  - Used to query/manipulate database
  - Inspired by SQL, but it operates directly on the entities and its fields
- Statements
  - select_ statement ::= select_ clause from_ clause [where_ clause] [groupby_ clause] [having_ clause] [orderby_ clause]
  - update_ statement ::= update_ clause [where_ clause]
  - delete_ statement ::= delete_ clause [where_ clause]

## 2.4 Transactions

**Access to the EntityManager must run within a transaction**

## 2.5 Data Transfer Object

- Detached Entity objects as DTOs
  - Hibernate developers say that you can use hibernate entity or domain objects as result types in service methods
  - Problems:
    - Lazy load exceptions are thrown if "not-loaded"fields are accessed
    - Having an accessor which does throw an exception is contract violating
    - Accessing the type of a result using reflection returns a proxy type (which is not serializable)
- Data Transfer Objects
  - Are used to transfer data across layers of your application
  - Only the data needed by the requesting layer are passed, i.e. not all properties need to be
  - No Lazy Loading Exception surprises
  - Clients are independent of ORM technology used

### 2.5.1 Service Method (Dozer)

- Dozer is a Java Bean to Java Bean mapper that recursively copies data from one object to another ⇒ can be used to copy DTO
- Dozer supports simple property mapping, complex type mapping, bi-directional mapping, implicit-explicit mapping, as well as recursive mapping. This includes mapping collection attributes that also need mapping at the element level

### 2.5.2 Con

- Code Duplication
  - In particular when DTOs have the same fields as domain objects
- Code to copy attributes back and forth
  - Dozer / Spring BeanUtils / JPA

### 2.5.3 Pro

- Lazy Loading Problem
  - You are not catched by a Lazy Loading Exception
    - neither on client side
    - nor upon serialization
- Triggers Design
  - Forces you to think about the interface of the remote service façades
  - Information from multiple domain objects can be combined in one DTO

# 3 Spring Remoting

## 3.1 Prüfung

**Facade:** Zuständig für abstraktion von Service Layer und benutzt am besten DTO’s

**Service Layer:** Service Schnittstellen stellt Business Cases dar

**Domain Model:** Entities

**Database Access:** Wird von JPA Implementation übernommen

# 4 Anhang

Listing 7: Persistence Unit

```
<persistence>
        <persistence−unit name=”movierental”
                        transaction−type=”RESOURCE_LOCAL”>
                <class>ch.fhnw.edu.rental.model.Movie</class>
                ...
                <properties>
                <property name=”hibernate.connection.driver_class”
                        value=”org.hsqldb.jdbcDriver” />
                <property name=”hibernate.connection.url”
                        value=”jdbc:hsqldb:hsql://localhost/lab−db” />
                <property name=”hibernate.connection.username”
                        value=”sa” />
                <property name=”hibernate.connection.password”
                        value=”” />
                </properties>
        </persistence−unit>
</persistence>
```

Listing 8: Query Examples

```
TypedQuery<Movie> q = em.createQuery(
        ”select m from Movie m where m.title = :title”,
Movie.class);
q.setParameter(”title”, title);
List<Movie> movies = q.getResultList();

@NamedQueries({
        @NamedQuery(name=”movie.all”, query=”from Movie”),
        @NamedQuery(name=”movie.byTitle”,
                query=”select m from Movie m where m.title = :title”)
})
class Movie {...}

TypedQuery<Movie> q = em.createNamedQuery(
        ”movie.byTitle”, Movie.class);
q.setParameter(”title”, title);
List<Movie> movies = q.getResultList();

SELECT c FROM Customer c WHERE c.address.city = ’Basel’
SELECT c.name, c.prename FROM Customer c
SELECT DISTINCT c.address.city FROM Customer c
SELECT NEW ch.fhnw.edu.Person(c.name,c.prename) FROM Customer c
SELECT pk FROM PriceCategory pk

TypedQuery<Movie> q = em.createQuery(
        ”select m from Movie m order by m.name”, Movie.class);
q.setFirstResult(20);
q.setMaxResults(10);
List<Movie> movies = q.getResultList();

Query q = em.createQuery(
        ”delete from Movie m where m.id > 1000”);
int result = q.executeUpdate();
```