Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Information Technology

Persistence with JPA
Module: APPE

Research and Development
CC Distributed Secure Software Systems
Roland Christen
Research Associate

T direct +41 41 349 33 84
roland.christen@hslu.ch

FH Zentralschweiz

EFQM
Recognised for excellence
4 star

Agenda

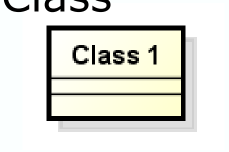
1. Data Modeling with UML
2. Implementation as Relational Schema
3. Refresher: JPA / ORM Basics
4. Creating the Persistence Layer
5. CRUD Operations
6. JPQL
7. Pitfalls and Best Practices

Data Modeling with UML

The goal of data modeling is to define the data requirements needed for the business cases in the system. We use an UML class diagram to model the data structure of the system.

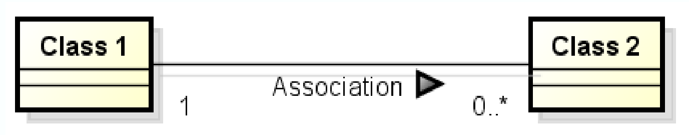
Elements:

Class



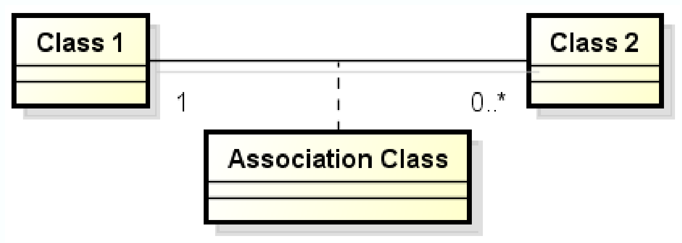
defines the structural attributes

Association



defines the relationship between classes

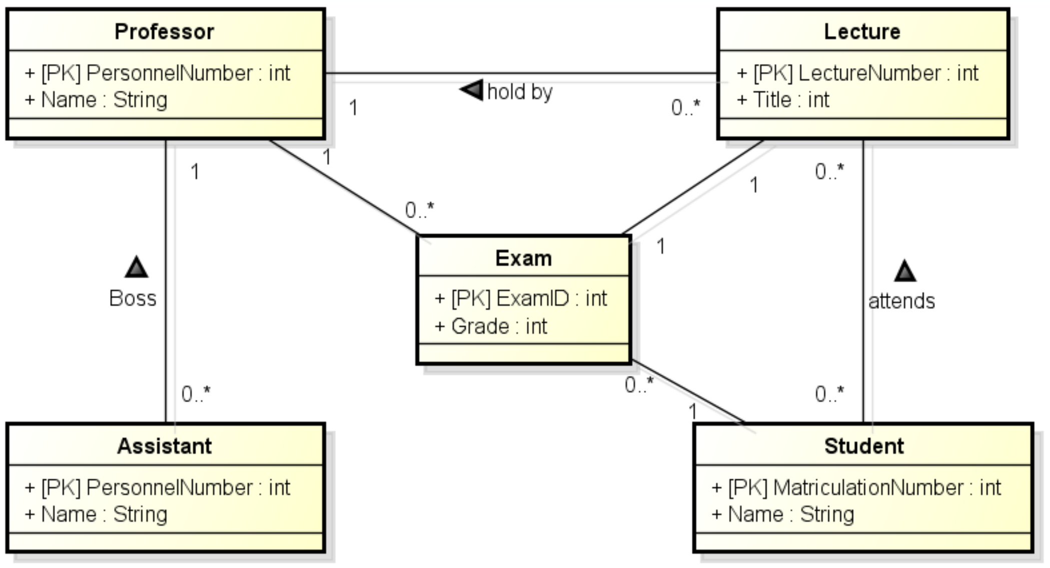
Association Class



defines association with attributes

Data Modeling with UML

Logical Data Model



Editor:
http://astah.net/editions/community

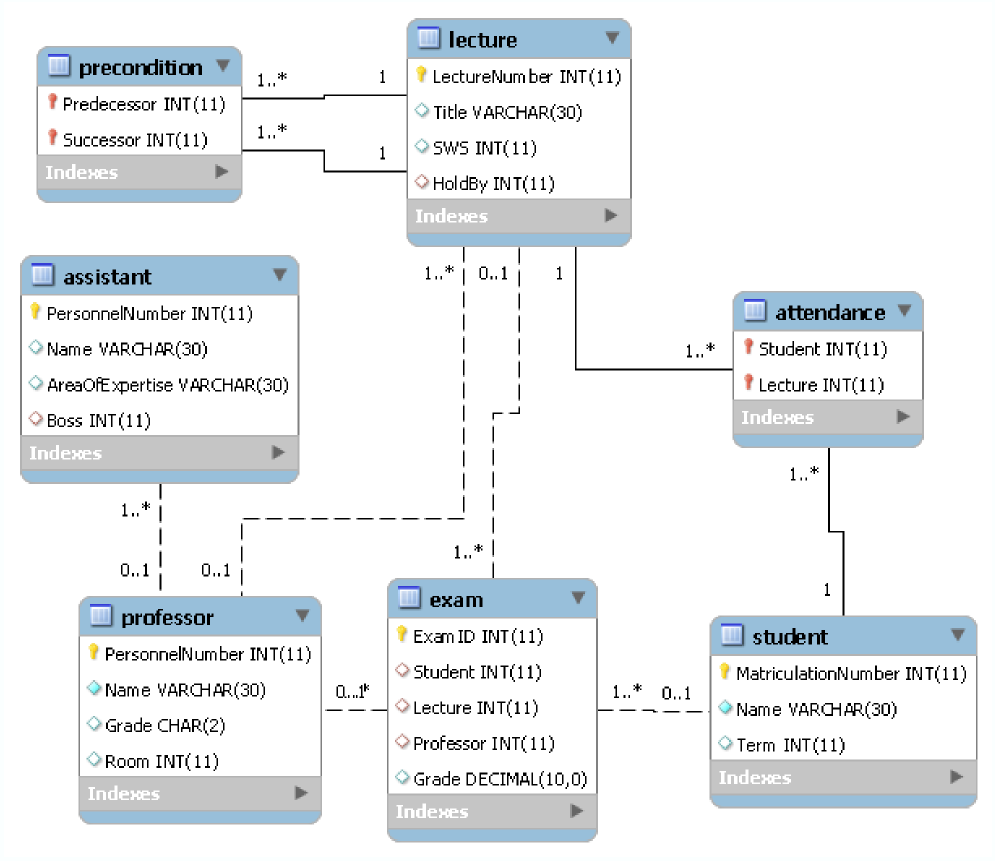
List of Unified Modeling Language tools:
http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

Implementation as Relational Schema

The relational Schema is a common way to implement a logical data model. We can use SQL DDL or the user interface of a DBMS to create the schema.

Creating a physical data model for a database:

- define all details to produce database
- map m:n relationships to associative tables
- define PK, FK and indexes
- ....

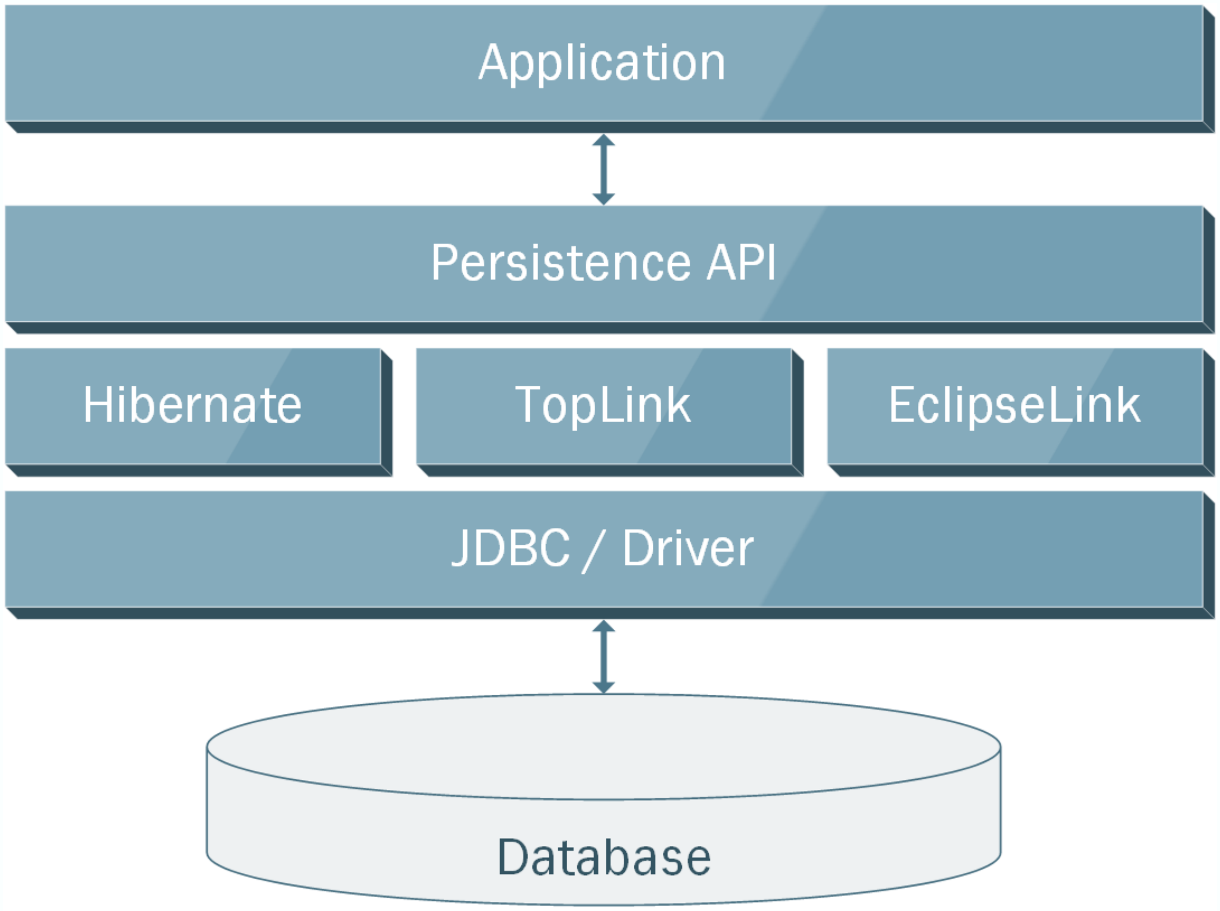


Implementation as Relational Schema

Tips:

- Associative tables with attributes: add autonumber field and create artificial primary key (Surrogate Key)
- Pure associative tables without any attributes: use FKs as primary key (Compound Key) -> no extra relationship class needed

Refresher: JPA / ORM Basics



Refresher: JPA / ORM Basics

JPA Elements:

- Entity: Java class, mapped to a table in the database
- Annotation: Definition of the OR mapping (Java class to database table)
- EntityManager: Class to access entity objects and perform persistence operations
- Persistence Unit: Configuration data for connectivity, DB driver, …
- JPQL: Object-oriented query language, platform-independent

Refresher: JPA / ORM Basics

JPA Annotation:

- @Entity: Designate a java class as entity
- @Table: Specify the table associated with the Entity (optional)
- @Id: Designate a field as primary key
- @Column: Associate a field with a column (optional)
- @OneToMany: Define association with one-to-many multiplicity
- @ManyToMany: Define association with many-to-many multiplicity
- …

Creating the Persistence Layer

There are two approaches to create the persistence artifacts:

- Code First:
  Create all entity classes in Java, add the annotations to map the classes to tables and let the ORM create the database.

- Database First: Create all tables in the database and use an IDE to generate all entity classes and the required annotations.

My Approach:

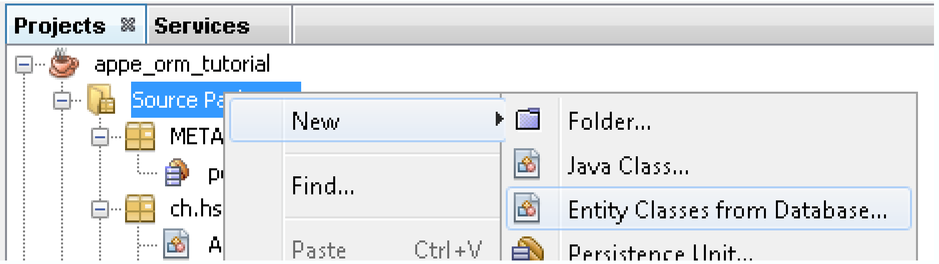
- Initial Project Setup: Database First. Less to write + SQL easier to verify than JPA

- Development with local Database: Code First with strategy *Drop and Create*. Schema is implicitly under version control via entity classes.

- Development / Integration with shared or productive Database: no auto generation, in order to detect compatibility issues.

Creating the Persistence Layer
with Netbeans

1. Create the database schema (with SQL or DBMS Editor) use Netbeans / Services or an external Tool such as MySql Workbench.
2. In Netbeans add the library **EclipseLink** (or the ORM of your choice) to the project as well as the **MySQL JDBC Driver**.
3. In Netbeans choose **new** and **Entity Classes from Database** and follow the steps in the wizard.



CRUD Operations
with referenced Entities

Create:

```
// prepare
EntityManagerFactory emf =
Persistence.createEntityManagerFactory(Config.DB_Link);
EntityManager em = emf.createEntityManager();

// Retrieve required Data
Professor p = em.find(Professor.class, NamedRecords.ProfSokrates);

// create Record
Assistant me = new Assistant();
me.setName("Roli Christen");
me.setAreaOfExpertise("SW Development");
me.setBoss(p);

// insert
em.getTransaction().begin();
em.persist(me);
em.getTransaction().commit();
```

CRUD Operations
with referenced Entities

Read:

```
// retrieve Entity and related Entities
Professor p = em.find(Professor.class, NamedRecords.ProfSokrates);
for (Lecture l : p.getLectureCollection()) {
        System.out.println(
                String.format("Professor %s holds lecture %s",
                p.getName(), l.getTitle())
        );
}
```

- Referenced Entities are automatically loaded (fetching strategies: EAGER or LAZY)

CRUD Operations
with referenced Entities

Update:

```
// update multiple Entities
Professor p = em.find(Professor.class, NamedRecords.ProfSokrates);
em.getTransaction().begin();
p.setName(p.getName() + " 2.0");
for (Assistant a : p.getAssistantCollection()) {
        a.setName(a.getName() + " 2.0");
}
em.getTransaction().commit();
```

- Modifications to entities are stored in the database when the transaction is committed or the method *flush* is invoked.

- Modifications to entities are only stored in the database when the entities are managed (managed and detached entities, see **Pitfalls and Best Practices**)

CRUD Operations
with referenced Entities

Delete:

```
// remove Entity
Lecture l = em.find(Lecture.class, LectureMaeeutik);
em.getTransaction().begin();
em.remove(l);
em.getTransaction().commit();
```

- Caution: already loaded collections, containing the now deleted entity, are not automatically updated (2nd Level Cache Problem).
  See **Pitfalls and Best Practices**

JPQL
Object-Oriented Queries

Select (typed):

```
// query all Professors with Grade C4
TypedQuery<Professor> query = em.createQuery("SELECT p FROM Professor p
WHERE p.grade = 'C4'", Professor.class);
List<Professor> results = query.getResultList();
```

Select (mixed)

```
// query all Professors with Assistants
Query query = em.createQuery("SELECT a.boss, COUNT(a) AS C FROM Assistant a
GROUP BY a.boss HAVING COUNT(a) > 0");
List<Object[]> results = query.getResultList();

// show Results
for (Object[] record : results) {
        Professor p = (Professor) record[0];
        long employees = (long) record[1];
        …
}
```

JPQL
Object-Oriented Queries

Select (with JOIN):

```
// query all Professors with Assistants
TypedQuery<Student> query = em.createQuery("SELECT DISTINCT s FROM Student
s JOIN s.lectureCollection l WHERE l.holdBy.grade = :grade",
Student.class);
query.setParameter("grade", "C4"); List<Student> results =
query.getResultList();
```

Update

```
// set Grades 2 to Grades 1
em.getTransaction().begin();
Query query = em.createQuery("UPDATE Exam e SET e.grade = 1 WHERE e.grade =
2");
int rowCount = query.executeUpdate();
em.getTransaction().commit();
```

JPQL
Object-Oriented Queries

Delete

```
// remove all lectures from Prof. Popper
em.getTransaction().begin();
Query query = em.createQuery("DELETE FROM Lecture l WHERE
l.holdBy.personnelNumber = :prof");
query.setParameter("prof", NamedRecords.ProfPopper);
int rowCount = query.executeUpdate();
em.getTransaction().commit();
```

Pitfalls and Best Practices
Managed and detached Entities

Persist vs Merge

- Persist: takes an entity instance, adds it to the context and makes that instance managed (future updates to the entity will be tracked).
- Merge: creates a new instance of an entity, copies the state from the supplied entity, and makes the new copy managed. The instance passed in will not be managed (any changes made to the entity will not be part of the transaction - unless *merge* is invoked again).

Pitfalls and Best Practices
Managed and detached Entities

Scenario A:

```
t.begin();
p = new Professor();
p.setName("Professor A1");
em.persist(p);
p.setName("Professor A2");
t.commit();
```

- ➢ *Professor is stored in the database with name "**Professor A2**"*

Scenario B:

```
t.begin();
p = new Professor();
p.setName("Professor B1");
em.merge(p);
p.setName("Professor B2");
t.commit();
```

- ➢ *Professor is stored in the database with name "**Professor B1**"*
- ➢ *Changes made after merging are not synchronized*
- ➢ *Passed entity **p** is not managed*

Pitfalls and Best Practices
Managed and detached Entities

Scenario C:

```
t.begin();
p1 = new Professor();
p1.setName("Professor C1");
Professor p2 = em.merge(p1);
p1.setName("Professor C2");
p2.setName("Professor C3");
t.commit();
```

- ➢ *Professor is stored in the database with name "**Professor C3**".*
- ➢ *Entity **p1** is still named "**Professor C2**" (detached)*
- ➢ *Entity **p2** is named "**Professor C3**" (managed)*

Pitfalls and Best Practices
Cached Entity Collections

The following snipped shows the problem of cached entity collections. First and second output both are **3**.

```
// load Professor Sokrates and count Lectures
Professor p = em.find(Professor.class, NamedRecords.ProfSokrates);
System.out.println(p.getLectureCollection().size());

// remove first Lecture
em.getTransaction().begin();
Lecture firstLecture = Stream.getByIndex(p.getLectureCollection(), 0);
em.remove(firstLecture);
em.getTransaction().commit();

// load Professor again and count Lectures
p = em.find(Professor.class, NamedRecords.ProfSokrates);
System.out.println(p.getLectureCollection().size());
```

Pitfalls and Best Practices
Cached Entity Collections

When entities are removed, it must be ensured that containing collections stay synchronized.

Solutions:

- Refresh the entity that holds the containing collection:

```
em.refresh(firstLecture.getHoldBy());
```

- Clear the cache (bad performance)

```
em.getEntityManagerFactory().getCache().evictAll();
```

- Manually remove the entity from all containing collections

```
firstLecture.getHoldBy().getLectureCollection().remove(firstLecture);
em.remove(firstLecture);
```

- Use framework features to remove orphaned entities (ORM specific).

```
@OneToMany(mappedBy = "holdBy", orphanRemoval = true)
private Collection<Lecture> lectureCollection;
```

Pitfalls and Best Practices
Cached Entity Collections

Automatization with Entity Lifecycle Hook

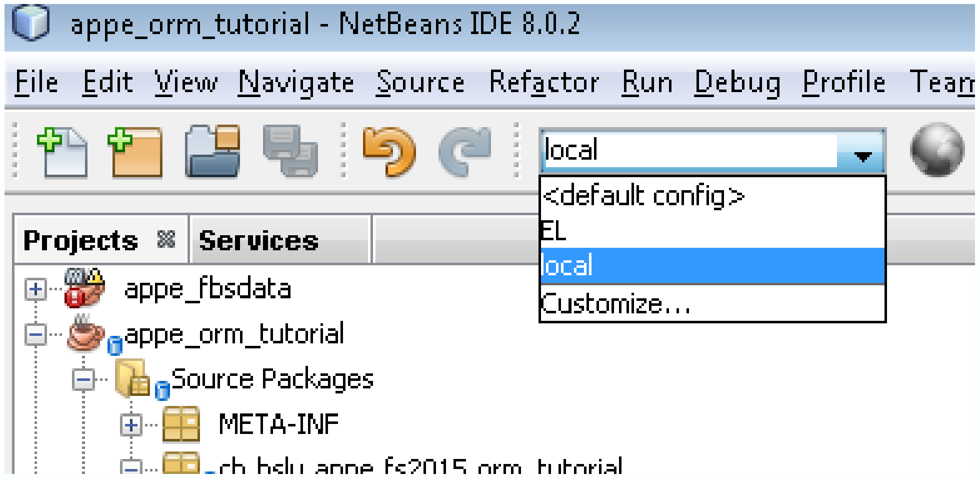
The solutions on the previous slide depend on the user to invoke the right methods when using the entity objects. A better approach is to automate the desired behaviour.

We can use a hook to invoke a method every time an entity is removed:

```
@PreRemove public void preRemove() {
        holdBy.getLectureCollection().remove(this);
}
```

Pitfalls and Best Practices
Multiple Persistence Units

- Multiple project configurations with different persistence units
- Easily switch between local and remote databases
- Easily switch between different database generation strategies


appe_orm_tutorial - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team
local
<default config>
EL
local
Customize...
Projects
Services
appe_fbsdata
appe_orm_tutorial
Source Packages
META-INF
ch.hslu.appe.fs2015.orm.tutorial

Pitfalls and Best Practices
Multithread Environment

Possible reasons for multiple threads:

- Fat clients often use multi-threading, to separate service calls and rendering to avoid freezing windows.
- Service / Application-Frameworks may use multi-threading to increase performance, improve response time or enable load balancing
- …

Note:

- Entity Manager are not thread-safe
- Do not share entity manager and other JPA objects among multiple threads
- Use transactions to isolate threads operations
- Choose adequate transaction isolation setting
- Transaction isolation settings are always a trade-off between accuracy and speed.

Further Reading:
http://en.wikipedia.org/wiki/Isolation_%28database_systems%29

Pitfalls and Best Practices
Tips

- Using multiple entity manager is possible, but more difficult to handle (every entity manager has its own context)
- Entity Manager are not thread-safe: do not share an entity manager among multiple threads
- Always perform modification to entities within transactions
- Note that modifications are not stored to the database before the end of the transaction (commit or flush required)
- Keep in mind that already loaded collections are not automatically updated after an entity removal
- Use JPA transaction features in the logic layer.
- Use only JPQL / JPA in Logic Layer, no SQL
- No not use entities in presentation layer

Questions
Answers