Java Persistence API (JPA)

Petr Křemen

petr.kremen@fel.cvut.cz

Winter Term 2016
Contents

1. Data Persistence
2. From JDBC to JPA
3. JPA Basics
4. Object-Relational Mapping (ORM) Basics
5. ORM Relationship Mapping
6. ORM Inheritance Mapping
Data Persistence
What “data persistence” means?

We manipulate data (represented as object state) that need to be stored persistently to survive a single run of the application, queriably to be able to retrieve/access them, scalably to be able to handle large data volumes, transactionally to ensure their consistency.
How to achieve persistence I

Serialization
- simple, yet hardly queriable, not transactional, ...
- stream persisting an instance of class C is deprecated once definition of C is modified (e.g. field added/removed).

Relational Databases (MySQL, PostgreSQL, Oracle, ...)
- efficient storage for data with rigid schema
- well-established and most popular technology
- efficient search using SQL standard
- secure and Transactional (ACID)
How to achieve persistence II

NoSQL Databases
- Key-value storages (MongoDB, Hadoop, ...)
  - suitable for data without rigid schema
- Object Databases
  - designed in 90’s to capture complexity of object models (e.g. inheritance)
  - Issues: scalability, standardized queries

RDF Triple Stores (SDB, TDB, Sesame, Virtuoso, ...)
- graph stores for distributed semantic web data – RDF(S), OWL
Programmatic Access to Relational Databases (RDBMS)

- **JDBC (JSR 221)**
  - Java standard to ensure independence on the particular RDBMS (at least theoretically)

- **EJB 2.1 (JSR 153)**
  - Provides Object Relational Mapping (ORM), but complicated
    
    \[\text{single entity} = \text{several Java files + XMLs}\]

  - distributed transactions, load balancing

- **iBatis, Hibernate – ORM driving forces for JPA 2**

- **JPA 2 (JSR 317)**
  - Standardized ORM solution for both standalone and Java EE applications
From JDBC to JPA
Java standard to ensure independence on the particular RDBMS (at least theoretically)

```java
Connection connection = null;
PreparedStatement statement = null;
try {
    Class.forName("org.postgresql.Driver");
    connection = DriverManager.getConnection(jdbcURL, dbUser, dbPassword);
    statement = connection.prepareStatement("SELECT * FROM PERSON WHERE HASNAME LIKE ?");
    statement.setString(1, "%Pepa%" );
    ResultSet rs = statement.executeQuery();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
    if ( statement != null ) {
        try {
            statement.close();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    }
    if ( connection != null ) {
        try {
            connection.close();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    }
}
```
### JDBC – entities CRUD

**Create**

```java
PreparedStatement statement = connection.prepareStatement("INSERT INTO PERSON (id, hasname) VALUES (?,?)");
statement.setLong(1, 10);
statement.setString(2, "Honza");
statement.executeUpdate();
```

**Retrieve**

```java
PreparedStatement statement = connection.prepareStatement("SELECT * FROM PERSON WHERE ID=?");
statement.setLong(1, 2);
ResultSet rs = statement.executeQuery();
```

**Update**

```java
PreparedStatement statement = connection.prepareStatement("UPDATE PERSON SET HASNAME='Jirka' WHERE ID=?");
statement.setLong(1, 2);
statement.executeUpdate();
```

**Delete**

```java
PreparedStatement statement = connection.prepareStatement("DELETE FROM PERSON WHERE ID=?");
statement.setLong(1, 1);
statement.executeUpdate();
```
Question 1: Why prepared statements?

```java
PreparedStatement statement = connection.prepareStatement("INSERT INTO PERSON (id,hasname) VALUES (?,?)");
statement.setLong(1,10);
statement.setString(2,"Honza");
statement.executeUpdate();
```
How to avoid boilerplate code?

- Boilerplate code
  - Obtaining (pooled) connection
  - SQLException handling
  - creating Java objects out of the query results:

```java
ResultSet rs = ...
while(rs.next()) {
    Person p = new Person();
    p.setId(rs.getLong("ID"));
    p.setHasName(rs.getString("HASNAME"));
}
```

- Although SQL is a standard – there are still differences in implementations (MySQL autoincrement, PostgreSQL serial ...)

  solution = Object Relational Mapping (ORM)
ORM Architecture

- idea: “map whole Java classes to database records”
- a typical system architecture with ORM:

```java
@Entity
public Person {
    @Id
    private Long id;
    private String hasName;
    // setters+getters
}
```

```sql
CREATE TABLE PERSON (  
    ID bigint PRIMARY KEY NOT NULL,  
    HASNAME varchar(255)  
);
```
CRUD using JPA 2.0

Initialization

```java
EntityManagerFactory f = Persistence.createEntityManagerFactory("pu");
EntityManager em = f.createEntityManager();
EntityTransaction t = em.getTransaction();
t.begin();
```

Create

```java
Person person = new Person();
person.setId(10);
Person.setHasName("Honza");
em.persist(person);
```

Retrieve

```java
Person person = em.find(Person.class, 2);
```

Update

```java
Person person = em.find(Person.class, 2);
person.setHasName("Jirka");
```

Delete

```java
Person person = em.find(Person.class, 1);
em.remove(person);
```

Finalization

```java
t.commit();
```
JPA 2.1

- Java Persistence API 2.1 (JSR-338)
- Although part of Java EE 7 specifications, JPA 2.1 can be used both in EE and SE applications.
- Main topics covered:
  - Basic scenarios
  - Controller logic – EntityManager interface
  - ORM strategies
  - JPQL + Criteria API
JPA 2.1 – Entity Example

- Minimal example (configuration by exception):

```java
@Entity
public class Person {

@Id
@GeneratedValue
private Integer id;

private String name;

// setters + getters
}
```
Let’s have a set of „suitably annotated“ POJOs, called entities, describing your domain model.

A set of entities is logically grouped into a persistence unit.

JPA providers:
- generate persistence unit from existing database,
- generate database schema from existing persistence unit.

**Question:** What is the benefit of keeping Your domain model in the persistence unit entities (OO) instead of the database schema (SQL)?
JPA – Model
JPA 2.0 – Persistence Context

- In runtime, the application accesses the object counterpart (represented by entity instances) of the database data. These (managed) entities comprise a **persistence context (PC)**.
  - PC is synchronized with the database on demand (refresh, flush) or at transaction commit.
  - PC is accessed by an EntityManager instance and can be shared by several EntityManager instances.
JPA – Operations

This is true for "transaction scoped" persistence context. In case of "extended scope", persistence context is not destroyed on commit.

Relational Database

Transaction.begin() 
create

em.refresh()
em.find(…)
query

Persistent Context

em.merge(…)
em.persist()
remove(…)

destroy

This is true for "transaction scoped" persistence context. In case of "extended scope", persistence context is not destroyed on commit.

Transaction.commit()
Transaction.rollback()
JPA – Entity States

JPA – Operation Details

**persist** stores a new entity into persistence context (PC). The PC must not contain an entity with the same id,

**merge** merges a detached entity with its managed version (inside PC),

**find** finds an entity in the DB and fetches it into PC,

**refresh** “reverts” a managed entity state from DB,

**remove** deletes a managed entity from PC.
**EntityManager (EM)** instance is in fact a generic DAO, while entities can be understand as DPO (managed) or DTO (detached).

Selected operations on EM (CRUD):

- **Create** : `em.persist(Object o)`
- **Read** : `em.find(Object id), em.refresh(Object o)`
- **Update** : `em.merge(Object o)`
- **Delete** : `em.remove(Object o)`

Native/JPQL queries : `em.createNativeQuery, em.createQuery, etc.`

Resource-local transactions :

```
em.getTransaction.[begin(),commit(),rollback()]
```
Object-Relational Mapping (ORM) Basics
Which one is correct?
Database model

Which one is correct?
Object-Relational Mapping (ORM) Basics

ORM Basics

Simple View
Java Classes = Entities = SQL tables
Java Fields/accessors = Entity properties = SQL columns

- The ORM is realized by means of Java annotations/XML.
- Physical Schema annotations
  - @Table, @Column, @JoinColumn, @JoinTable, etc.
- Logical Schema annotations
  - @Entity, @OneToOne, @OneToMany, @ManyToMany, etc.
- Each property can be fetched lazily/eagerly.
Mapping basic types

Primitive Java types:
- `String` → varchar/text,
- `Integer` → int,
- `byte[]` → blob,
- etc.

- `@Column` – physical schema properties of the particular column (insertable, updatable, precise data type, defaults, etc.)
- `@Lob` – large objects
- Default EAGER fetching (except `@Lobs`)

```java
@Column(name="id")
private String getName();
```
Mapping enums/temporals

Enums

```java
@Enumerated(value=EnumType.String)
private EnumPersonType type;
```

Stored either in a text column, or in an int column

Temporals

```java
@Temporal(TemporalType.Date)
private java.util.Date datum;
```

Stored in respective column type according to the TemporalType.
ORM – Identifiers

- Single-attribute: `@Id`
- Multiple-attribute – an identifier class must exist
  - Id. class: `@IdClass`, entity ids: `@Id`
  - Id. class: `@Embeddable`, entity id: `@EmbeddedId`

```java
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE)
private int id;
```

**Question:** How to write `hashCode`, `equals` for entities?
ORM – Generating Identifiers

Strategies

**AUTO** – the provider picks its own strategy

**TABLE** – special table keeps the last generated values

**SEQUENCE** – using the database native SEQUENCE functionality (PostgreSQL)

**IDENTITY** – some DBMSs implement autonumber column

For database-related strategies, the value of id is set only on

- `Transaction.commit()`
- `em.flush()`
- `em.refresh()`
ORM – Generated Identifiers TABLE strategy

```java
@TableGenerator(
    name="Address_Gen",
    table="ID_GEN",
    pkColumnName="GEN_NAME",
    valueColumnName="GEN_VAL",
    initialValue=10000,
    allocationSize=100)
@Id
@GeneratedValue(generator=AddressGen)
private int id;
```
ORM Relationship Mapping
ORM – Relationships

**Unidirectional**
- accessed from **one side** only
  - `emp.getProjects()`
  - `prj.getEmployees()`

**Bidirectional**
- accessed from **both sides**
  - `empl.getProjects()`
  - `prj.getEmployees()`
- **owning side** = side used for changing the relationship
- **inverse side** = read-only side
Unidirectional many-to-one relationship

owning side = Employee

In DB, the N:1 relationship is implemented using a foreign key inside the Employee table. In this case, the foreign key has a default name.
Unidirectional many-to-one relationship II

```java
@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;
    @ManyToOne
    @JoinColumn(name=DEPT_ID)
    private Department department;
}
```

owning side = Employee.
Here, the foreign key is defined using the @JoinColumn annotation.
Bidirectional many-to-one relationship

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;
    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department department;
}
```

```
@Entity
public class Department {
    @Id
    private int id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
}
```

owning side = Employee
inverse side = Department

Here, the foreign key is defined using the @JoinColumn annotation.
Unidirectional one-to-one relationship

@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
}

owning side = Employee.
Bidirectional one-to-one relationship

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
}
```

```
@Entity
public class ParkingSpace {
    @Id
    private int id;
    private int lot;
    private String location;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
}
```

owning side = Employee

inverse side = ParkingSpace
Bidirectional many-to-many relationship

```
@Generated(projection="org.hibernate.annotations.JpaGeneratedIdentity")

@Entity
public class Employee {
    @Entity
    public class Project {

        @Id
        private int id;
        private String name;

        @ManyToMany(mappedBy="projects")
        private Collection<Employee> employees;
    }

    @Id
    private int id;
    private String name;
    private long salary;

    @ManyToMany
    private Collection<Project> project;
}
```

Owning side = Employee

Inverse side = ParkingSpace
Conceptual Modeling Intermezzo

- M:N relationship is a **conceptual modeling** primitive

![Diagram showing a M:N relationship between Patient and Hospital]

- Does it mean that
  - A patient has **one** treatment that is handled in **more** hospitals?
  - A patient has **more** treatments, each handled in a **single** hospital?
  - A patient has **more** treatments, each handled in **more** hospitals?

- Partialities and cardinalities are too weak in this case.

Careful modeling often leads to decomposing M:N relationships on the **conceptual level** (not on the logical level, like JPA).
Bidirectional many-to-many relationship

```
@Entity
public class Employee {
    @Id private int id;
    private String Name;
    private long salary;
    @ManyToMany
    @JoinTable(name="EMP_PROJ",
                joinColumns=@JoinColumn(name="EMP_ID"),
                inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
    private Collection<Project> projects;
}

@Entity
public class Project {
    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
}
```

owning side = Employee

inverse side = ParkingSpace
Unidirectional many-to-many relationship

@Entity
public class Employee {
    @Id private int id;
    private String Name;
    private long salary;
    @ManyToMany
    @JoinTable(name="EMP_PROJ",
                joinColumns=
                @JoinColumn(name="EMP_ID"),
                inverseJoinColumns=
                @JoinColumn(name="PROJ_ID"))
    private Collection<Project> projects;
}

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

owning side = Employee
Unidirectional one-to-many relationship

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToMany
    @JoinTable(name=EMP_PHONE,
               joinColumns=
               @JoinColumn(name=EMP_ID),
               inverseJoinColumns=
               @JoinColumn(name=PHONE_ID))
    private Collection<Phone> phones;
}

@Entity
public class Phone {
    @Id private int id;
    private String type;
    private String num;
}
```

owning side = Employee
Unidirectional one-to-many relationship

```java
@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    @OneToMany
    @JoinTable(name="EMP_PHONE",
               joinColumns=@JoinColumn(name="EMP_ID"),
               inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
    private Collection<Phone> phones;
}

@Entity
public class Phone {
    @Id
    private int id;
    private String type;
    private String num;
}
```
Lazy Loading

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;

    private ParkingSpace parkingSpace;
}
```

```java
@Entity
public class Employee {
    @Id private int id;
    private String name;

    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
}
```

`parkingSpace` instance fetched from the DB at the time of reading the `parkingSpace` field.
ORM Inheritance Mapping
Inheritance

How to map inheritance into DB?
Strategies for inheritance mapping

**single table**

**joined**

**single table**
Inheritance mapping (single-table)

```java
@Entity
@Table(name="DB_PERSON_C")
@Inheritance /* same as
    @Inheritance(strategy=InheritanceType.SINGLE_TABLE)*/
@DiscriminationColumn(name="EMP_TYPE")
public abstract class Person {...}

@Entity
@DiscriminatorValue("Emp")
Public class Employee extends Person {...}

@Entity
@DiscriminatorValue("Stud")
Public class Student extends Person {...}
```
Inheritance mapping (joined)

```java
@Entity
@Table(name="DB_PERSON_C")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminationColumn(name="EMP_TYPE",
                discriminatorType=discriminatorType.INTEGER)
public abstract class Person {...}

@Entity
@Table(name="DB_EMPLOYEE_C")
@DiscriminatorValue("1")
public class Employee extends Person {...}

@Entity
@Table(name="DB_STUDENT_C")
@DiscriminatorValue("2")
public class Student extends Person {...}
```
Inheritance mapping (table-per-class)

@javax.persistence.Entity
@javax.persistence.Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Person {
}

@javax.persistence.Entity
@javax.persistence.Table(name=DB_EMPLOYEE_C)
@javax.persistence.AttributeOverride(name=name, column=@javax.persistence.Column(name=FULLNAME))
public class Employee extends Person {
}

@javax.persistence.Entity
@javax.persistence.Table(name=DB_STUDENT_C)
public class Student extends Person {...}