Java Persistence API

A Short Course
(Part I)

Jeferson L. R. Souza (jefecomp)

PhD candidate / IT Consultant

jefersonsouza@boldint.com

June 15, 2015

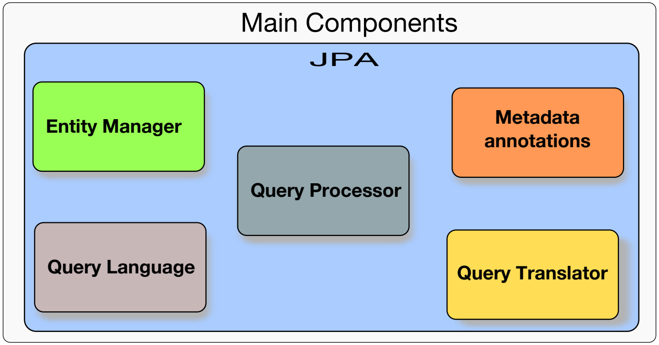
What is the Java Persistence API?

- The Java Persistence API (JPA) is a specification defining a standard way to persist objects on storage systems;
- The JPA itself is part of the Enterprise Java Beans (EJBs) 3.0 specification, which is defined in the JSR 220 (`https://jcp.org/en/jsr/detail?id=220`);
- In theory, the storage system utilised to persist objects of Java applications can be anything. However, JPA has been initially designed to allow saving and querying Plain Old Java Objects (POJOs) on relational databases such as MySQL, PostgreSQL, Oracle, etc;

JPA FEATURES

- Object-oriented data and persistence model;
- Rich Metadata annotations for definition of persistence entities;
- Object and query-level caching support;
- Transactional object model support;
- Enable on both JavaEE and JavaSE applications.

Main Components



The main abstract components of the Java Persistence API.

Entity Manager

- Responsible to manager the life cycle of persistence entities within a domain dubbed Persistence Context;
- Exposed interface with operations for save, delete, update, and query of entities on a storage system;
- Internal characteristics and behaviour depends on JPA implementations;

Entity Manager (1)

```
public interface EntityManager {
public void persist(Object entity);
public <T> merge (T entity);
public void remove (Object entity);
public <T> find (Class<T> entityClass, Object primaryKey);
public <T> getReference (Class<T> entityClass, Object primaryKey);
public void flush();
                          .
                          .
                          .
}
```

Metadata Annotations

- The way a POJO is identified as a persistence entity by JPA;
- Types of annotations:
  - Entity Definition and Inheritance (*@Entity, @MappedSuper-Class, . . .*);
  - Entity Relations (*@OneToOne, @OneToMany, @ManyToOne, @MaToMany*);
  - Object Relational Mapping (*@Id, @Table, @Column, @JoinCol-umn, . . .*).

Metadata Annotations (1)

Entity Example:

```
@Entity
public class Person implements Serializable {

                    .
                    .
                    .

}
```

Metadata Annotations (2)

- Types of annotations:
  - Query definition(*@NamedQuery, @NamedNativeQuery, ...*);
  - Management and context definitions (*@PersistenceUnit, @PersistenceContext, ...*);
  - Callback methods(*@PrePersist, @PostUpdate, @PreRemove, ...*).
- Internal characteristics and behaviour of a Entity Manager depends directly on JPA implementations.

Metadata Annotations (3)

Example of NamedQuery:

```
@NamedQuery(name="findAllPersons", query="SELECT p FROM Person
p")
public class Person {

...
}
```

Usage:

```
private EntityManager em;

...

TypedQuery<Person> query = em.createNamedQuery("findAllPersons",
Person.class);

List<Person> results = query.getResultList();
```

QUERY LANGUAGE

Example:

```
Select p FROM Person p
```

- An specific query language to load objects from a given storage systems;
- The specified query language is dubbed Java Persistence Query Language (JPQL);
- A lot of similarities with Structured Query Language (SQL), but object-oriented;

Query Language(1)

Three Common (and important!) clauses:

- SELECT: SELECT address FROM Address address;
- UPDATE: UPDATE Person person SET person.surname = 'Souza' WHERE person.name = 'Jeferson';
- DELETE: DELETE FROM Person person WHERE person.name = 'Jeferson'.

The JPQL has much more clauses we can use to build object-oriented queries. Keeping its use as minimum as possible reduces the complexity of your Data Access Object (DAO) layer.

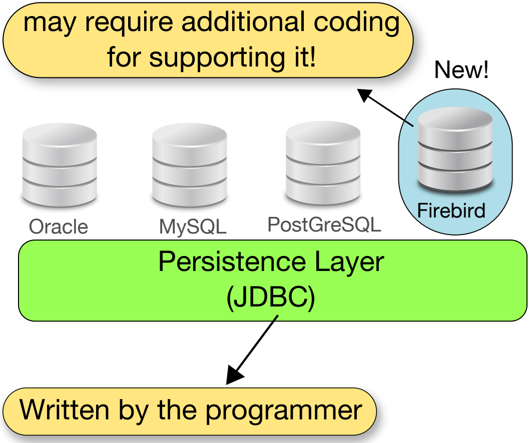
Query Processor

- Processing the received query to identify the elements involved; separating JPQL commands, entities, and dynamic parameters.
- Using the Query translator to actually transform a JPQL query into, e.g., a specific SQL query for the target database system.

QUERY TRANSLATOR

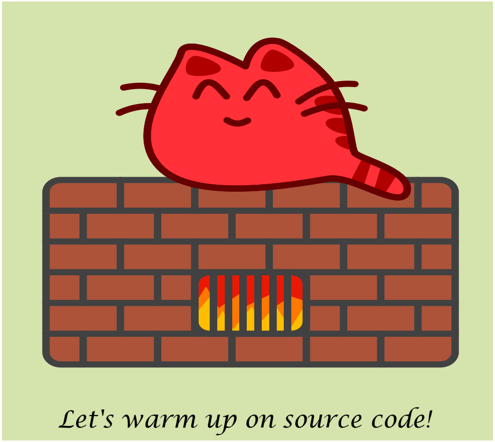
- Translates the pre-processed JPQL query to, e.g., the specific SQL dialect of the target database system;
- Although SQL is a standard language, almost all database systems have their own specify subset of non-standard clauses which boost performance on query processing. JPA implementations then take advantage of Dialect classes to perform a more efficient $JPQL \iff SQL$ translation.

Pure JDBC Model

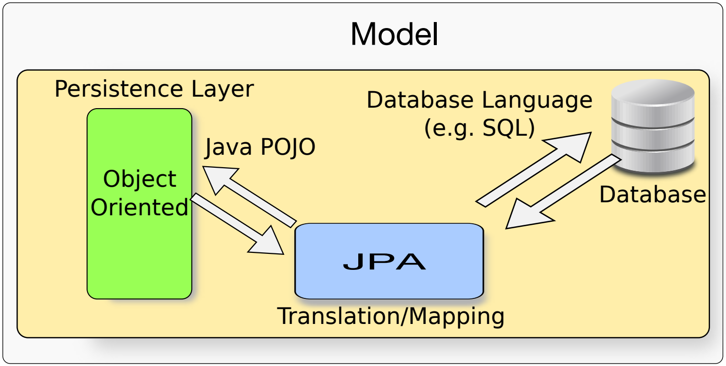

may require additional coding for supporting it!
New!
Oracle
MySQL
PostGreSQL
Firebird
Persistence Layer (JDBC)
Written by the programmer

Sometimes (i.e., OFTEN!) a modular and stable Pure JDBC model is hard to achieve.

Pure JDBC Model (1)



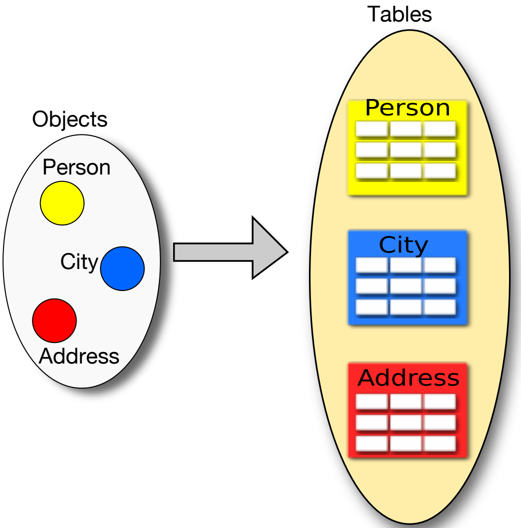
JPA Model



Transparent translation and mapping of a object oriented model to different kind of storage systems.

JPA Model (1)

Object-Relational Mapping (ORM)



Mapping Strategies

- A Table Per Concrete Class;
- A Single Table Per Class Hierarchy;
- A Joined Subclass Strategy.

A Table per concrete class

Advantages:

- Direct mapping between classes defined on the Object Oriented model and Tables on a given database;
- All properties (included those inherited from a superclass) are included as table columns.

Drawbacks:

- Poor query performance when the Object Oriented model has many relations between entities;
- A considerable number of SQL JOINS must be need to retrieve a given set of objects from database.

A Table per class hierarchy

Advantages:

- Classes sharing common properties are grouped within a single table;
- Fast queries when the purpose is to find entities which are part of the same class hierarchy.

Drawbacks:

- As all subclasses are grouped on the same table, all specific fields of each subclass should be nullable;
- All basic validations required to guarantee model consistency on a given subclass when inserting and/or updating data must be done by the programmer.

A Joined Subclass Strategy

Advantages:

- Superclass and Subclass fields are represented as separated tables.
- Provides support for polymorphic relationships.

Drawbacks:

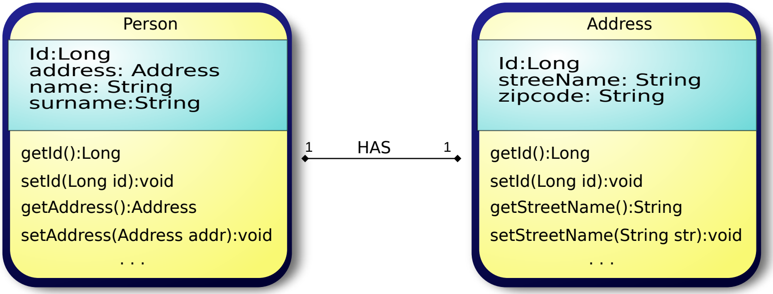
- May require or or more SQL JOIN operations to instantiate objects from specific subclasses;
- The performance of database access is dependent on the complexity of class hierarchy.

RELATIONSHIP BETWEEN ENTITIES

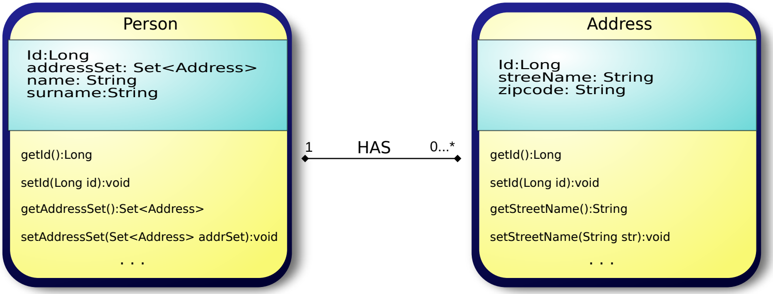
The relationship between persistent entities can be of four different types (excluding inheritance):

- OneToOne;
- OneToMany;
- ManyToOne;
- ManyToMany.

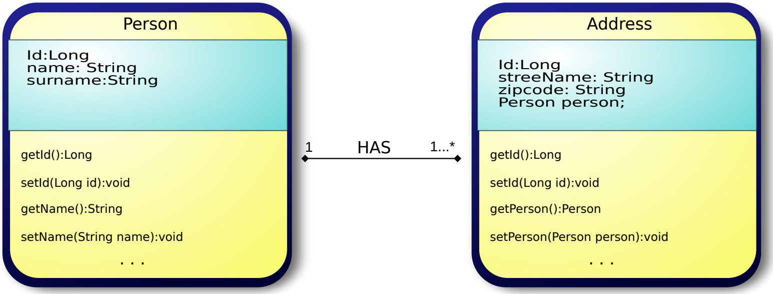
OneToOne


Person
Id:Long
address: Address
name: String
surname:String
getId():Long
setId(Long id):void
getAddress():Address
setAddress(Address addr):void
. . .
1
HAS
1
Address
Id:Long
streeName: String
zipcode: String
getId():Long
setId(Long id):void
getStreetName():String
setStreetName(String str):void
. . .

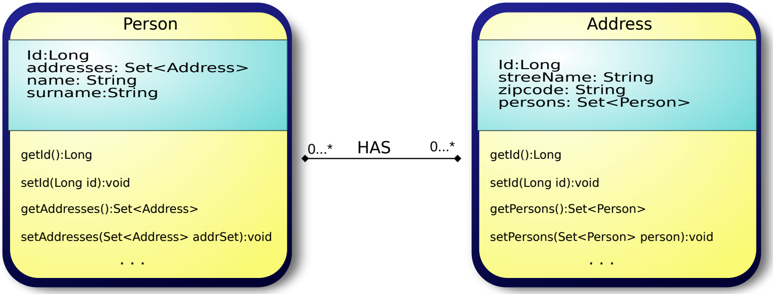
OneToMany


Person
Id:Long
addressSet: Set<Address>
name: String
surname:String
getId():Long
setId(Long id):void
getAddressSet():Set<Address>
setAddressSet(Set<Address> addrSet):void
. . .
1
HAS
0...*
Address
Id:Long
streeName: String
zipcode: String
getId():Long
setId(Long id):void
getStreetName():String
setStreetName(String str):void
. . .

ManyToOne


Person
Id:Long
name: String
surname:String
getId():Long
setId(Long id):void
getName():String
setName(String name):void
. . .
1
HAS
1...*
Address
Id:Long
streeName: String
zipcode: String
Person person;
getId():Long
setId(Long id):void
getPerson():Person
setPerson(Person person):void
. . .

ManyToMany


Person
Id:Long
addresses: Set<Address>
name: String
surname:String
getId():Long
setId(Long id):void
getAddresses():Set<Address>
setAddresses(Set<Address> addrSet):void
. . .
0...*
HAS
0...*
Address
Id:Long
streeName: String
zipcode: String
persons: Set<Person>
getId():Long
setId(Long id):void
getPersons():Set<Person>
setPersons(Set<Person> person):void
. . .

Persistence Provider

- It is the concrete implementation of the JPA exposed interfaces;
- As the behaviour of implementations are not defined within the Standard, different Persistence Providers may behave differently on the same set of inputs;
- The two most utilised persistence providers are the *Hibernate* and the *EclipseLink*.

Hibernate: The Entrepreneur

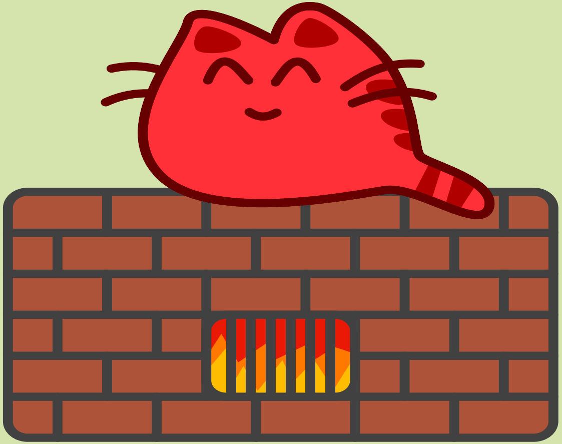
EclipseLink: The reference implementation

The complete Hello World JPA example

EXTERNAL SOURCES

- JPA Specification: JSR 220 (`https://www.jcp.org/en/jsr/detail?id=220`);
- OpenClipart (`https://openclipart.org/`);
- Inkscape (`https://inkscape.org/en/`);
- LaTeX Project (`http://www.latex-project.org/`);
- LaTeX Beamer (`https://bitbucket.org/rivanvx/beamer/wiki/Home`).

That's it folks!



Thank you for your attention!