

ORM

OBJECT RELATIONAL MAPPING

Store and manipulate objects in database

Store business entities (java objects) as relational entities (database tables)

Bridge between object models (Java program) and relational models (database program)

Table structure in database versus graph structure in objects (references containing references)

OO -> Non scalar values -> Several values -> Object / Array

Relational database -> Scalar values -> Only one value -> Int / String

Java: Collections are used to manage lists of objects

Databases: Tables are used to manage lists of entities

Conversion of data:

Conversion between object values into groups of simple values

Tables in relational database <-> Classes in object oriented language

Rows in relational database <-> Objects in object oriented language

Virtual object database

Advantages

Reduces amount of written code

No low level JDBC / SQL code

Independence of and separation from database / schema

Only OO paradigm

Disadvantages

Low level is not clear

Not optimally designed databases

Difficulties with mismatching object system to relational database

High level of abstraction / Low level JDBC and SQL code

The main goal is to do less work when working with persistence of objects

Mismatch issues

- How are tables, columns and rows mapped to objects?
- How are relationships handled?
- How is inheritance handled?
- How is composition and aggregation handled?
- How are conflicting database type systems handled?

JPA

Specification / Set of interfaces / Not a product

Classes and interfaces are used for storing entities into a database as a record

4 areas

Java persistence involves 4 areas:

- Java Persistence API
Implementation specification
- Object relational mapping metadata
Metadata / annotations used for mapping
- JPQL – Java Persistence Query language
Query entities and relationships
- Java Persistence Criteria API
Alternative way of defining a JPQL query

ORM / Hibernate & EclipseLink / JPA

ORM

Approach of taking object oriented data and mapping to a relational database

Concept / process of converting the data from object oriented language to relational DB and vice versa

JPA

High level API and specification that different ORM tools can implement, so that it provides the flexibility for developers to change the implementation from one ORM to another

Provider

Implementation of an ORM framework

Persistence library

EclipseLink 2.6.4

Hibernate 5.2.10

JDBC

Database driver

Handles conversion between databases automatically

JPA in Netbeans - Generate tables from classes

1. Create maven project
2. Create new empty database / schema
3. Create database connection (MySQL Connector/J driver)
4. Create persistence unit (EclipseLink/Hibernate)
5. Add dependency mysql-connector-java 5.1.43
6. Create packages entity and jpacontrol
7. Create entity.java class in entity package and add it to persistence unit
8. Create structure.java class in jpacontrol package

JPA in Netbeans - Generate classes from tables

1. Create maven project
2. Use existing database / schema
3. Create database connection (MySQL Connector/J driver)
4. Create persistence unit (EclipseLink/Hibernate)
5. Add dependency mysql-connector-java 5.1.43
6. Create packages entity and jpacontrol
7. Create new entity classes from database
8. Add tables, deselect JAXB annotations and use columns names in relationships
9. Create structure.java class in jpacontrol package

JAVA PERSISTENCE API

PersistenceUnit

Registers the database and specifies entity classes

persistence.xml

create / drop-and-create / none

drop-and-create does not remove old tables just overwrites

```
import javax.persistence.Persistence
Persistence.generateSchema(PUName, PUProperties)
```

SQL

Can be executed via persistence unit
Create new folder scripts in src/main/resources
Create new SQL file in scripts folder
Add SQL file to persistence unit

Entity classes

Entity package
Empty constructor
Setters / Getters
ToString

OBJECT RELATIONAL MAPPING METADATA

3 purposes

- Identify persistent classes
- Override default JPA behavior
- Provide JPA implementation with information that it cannot get by just reflecting on the persistent class

Entities / Rules

- Lightweight persistence domain object
- Typically represents a table in a relational database and each entity instance corresponds to a row in that table.
- Entities <-> Tables / Instance <-> Row
- Configuration by exception / Only apply when changes are required
- Class must be annotated with the **@Entity** annotation
- Class must at least have a public or protected, no-argument constructor
- Class must not be declared final
- No methods or persistent instance variables must be declared final
- If an entity instance is passed by value as a detached object, the class must implement the Serializable interface
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes
- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods

ANNOTATIONS

Annotations make it possible to configure entities
Annotations are used for classes, properties, and methods
Annotations starts with '@' symbol

@Entity

@Table(name="tbl_person")

Rename table

@Column(name="PERSON_ID")

Rename column

Date time

@Temporal(TemporalType.DATE)

private Date creationDate;

@Temporal(TemporalType.TIMESTAMP)

private Date expirationDate;

Transient

Primary keys - AutoGenerated values

@Id

@GeneratedValue

(strategy = GenerationType.AUTO) -> Persistence provider chooses strategy

(strategy = GenerationType.IDENTITY) -> Auto increment values

(strategy = GenerationType.SEQUENCE) -> Use a sequence for values

(strategy = GenerationType.TABLE) -> Use a table for values

Composite

Enums