ORM / JPA / JPQL

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

Updating EclipseLink

Use single eclipselink dependency in pom.xml

```
<dependency>
		<groupId>org.eclipse.persistence</groupId>
		<artifactId>eclipselink</artifactId>
		<version>2.6.4</version>
</dependency>
```

JAVA PERSISTENCE API

PersistenceUnit

Registers the database and specifies entity classes

Design / Source

XML file

- Other sources…
  - src/main/resources/META-INF/persistence.xml

persistence.xml

- create / drop-and-create / none
  - drop-and-create does not remove old or changed tables just empties

Add / Remove entity classes to persist

Logging SQL and parameters

```
<property name="eclipselink.logging.level.sql" value="fine" />
<property name="eclipselink.logging.parameters" value="true" />
```

SQL scripts

- 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

import javax.persistence.Persistence

Persistence.generateSchema(PUName, PUProperties)	To generate schema
Persistence.createEntityManagerFactory(PUName)	To get entity manager factory

HashMap of persistence unit properties can be altered to change persistence unit properties, such as password, sql script or table generation strategies

Entity classes

- Entity package
- Empty constructor
- Setters / Getters
- Add to arraylists
- 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
Do not persist field in database

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
@TableGenerator(name="TG", allocationSize=10, initialValue=22)
@GeneratedValue(strategy=GenerationType.TABLE, generator="TG")

Relationships
Enums
@Enumerated(EnumType.STRING)
Use enum strings
ElementCollection
@ElementCollection()
OneToMany (NonEntities)
Cardinality
One / Many
@OneToOne / @OneToMany / @ManyToMany
Direction
Unidirectional / Bidirectional

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
One-to-many / Many-to-one	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

Alt + Enter -> Choose relationship

MappedBy

(mappedBy = "address")
Owning side / Field

@JoinColumn

Owned side / Field

Create “Add to arraylist method” for arraylists, besides set arraylist and get arraylist methods

Fetch

Eager:	Fetch fields immediately for entities	(fetch = FetchType.EAGER)
Lazy:	Fetch fields when used for entities	(fetch = FetchType.LAZY)

Cascading

Parent / Child associations
Trigger what happens to children when parent changes
Cascade

@ManyToMany(cascade = CascadeType.ALL)
Cascade operations to referenced fields

OrphanRemoval

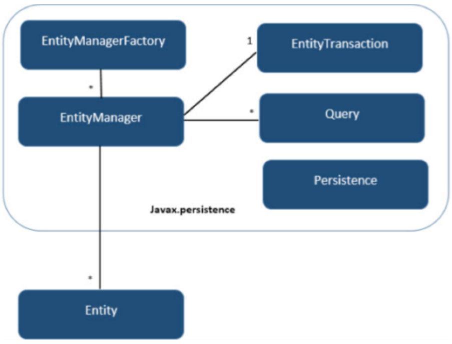
@OneToMany(orphanRemoval = true)
Removes orphans more strictly on OneToOne and OneToMany relations

Inheritance

3 strategies for mapping inheritance in database tables
SingleTable, Joined, TablePerClass

- SingleTable: A single table is used to store all of the instances of the entire inheritance hierarchy
  + Single table inheritance is the simplest, and default, and often the best performing solution.
  + Works well when the hierarchy is relatively simple and stable
  - Problematic when adding attributes
- Joined: A table is defined for each class in the inheritance hierarchy to store only the local attributes of that class
  + Joined inheritance is the inheritance strategy that most closely mirrors the object model into the data model
  - Querying can have performance impact
- TablePerClass : A table per concrete entity class where all attributes of the root entity will also be mapped to columns of the child entity table
  + Performs well when querying instances of one entity
  - Denormalizes the model

ENTITYMANAGER / ENTITYMANAGERFACTORY



Entities are managed by entity managers which are javax.persistence.EntityManager instances
Each EntityManager instance is associated with a persistence context / a set of managed entity instances

Central piece in JPA that manages the state and life cycle of entities and is able to handle JPQL queries

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "NamePU" );
EntityManager em = emf.createEntityManager();
```

There are two main states for entities: managed / detached

Query the managed objects using the Entity Manager
Query objects and fields instead of tables and columns

Can create, retrieve, update or delete objects from a database

persist, merge, find, remove

Persist:

Use for new entities
Entity becomes managed / Changes afterwards are saved

```
em.persist(new Employee(“John”, “Richards”));
```

Merge:

Use for existing changed entities
Entity does not become managed / Changes afterwards are not saved

```
em.merge(new Employee(“John”, “Richards”));
```

Find:

Entity becomes managed / Changes afterwards are saved
Combination of class and primary key

```
em.find(Employee.class, 1);
```

Remove:

Remove a retrieved object

```
em.remove(employee);
```

FACADESETUP

FaçadeInterface

```
public User getUser(Long id);
public User addUser(User u);
public User deleteUser(Long id);
public User editUser(User u);
public List<User> getUsers();
```

Façade

```
private EntityManagerFactory emf;

public FacadeUser(EntityManagerFactory emf)
{
   this.emf = emf;
}

@Override
public User getUser(Long id)
{
   EntityManager em = emf.createEntityManager();
   try
   {
      em.getTransaction().begin();
      em.getTransaction().commit();
```

```
    }
    finally
    {
      em.close();
    }
}
```

FacadeTest

```
FacadePerson fp = new FacadePerson( Persistence.createEntityManagerFactory( "jpaPU" ) );
```

JPQL

Similar to SQL / Not SQL

Two types of queries

- Named queries
  - Used for static queries that will be used many times in the application
  - Can be defined once in entity and then reused
- Dynamic queries
  - Normally used when the query depends on the context
  - Needs to be translated when invoked

Queries

Named

```
@NamedQueries(
{
@NamedQuery(name = "Person.findAll", query = "SELECT p FROM Person p")
, @NamedQuery(name = "Person.findByMail", query = "SELECT p FROM Person p WHERE p.mail = :mail")
})

Query Nquery = em.createNamedQuery("findAllPersonsWithFirstName");
```

Dynamic

```
Query Dquery1 = em.createQuery("SELECT p FROM Person p");
Query Dquery2 = em.createQuery("SELECT UPPER(p.firstName) from Person p");
Query Dquery3 = em.createQuery("SELECT MAX(p.phoneNumber) from Person p");
Query Dquery4 = em.createQuery("SELECT p FROM Person p WHERE p.member = :tinyint");
```

Parameters

```
query.setParameter("firstName", "Dan");
```

ResultList

```
List<Person> persons = query.getResultList();
for(Person p:persons)
{
  System.out.println("Person: " + p);
}
```

JPA CRITERIA API

Used to define more advanced, dynamic queries for entities

javax.persistence.criteria