JPA

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database.

@**Entity** - A class which should be persisted in a database. JPA uses a database table for every entity. Persisted instances of the class will be represented as one row in the table.

@**Cacheable** – ALL, NONE, ENABLE\_SELECTIVE, DISABLE\_SELECTIVE  
 cache.contains(Animal.class, idAnimal)  
<http://www.thejavageek.com/2014/09/25/jpa-caching-example/>

@**MappedSuperclass** – Designates a class whose mapping information is applied to the entities that inherit from it. A mapped superclass has no separate table defined for it.

no table exists for the mapped superclass itself, mapping apply to subclasses. Mapping can be overridden with ***AttributeOverride and AssociationOverride***

@**ID** - Specifies the primary key of an entity.

@**Version** – version of an Entity class, ensures integrity on merging  
only on 1 property/field of a class, must be mapped to corresponding table.

@**Transient** – field will not be saved in the DB, it is not persistent. Used for mapping a entity class, mapped superclass, or embeddable class.

@**EnumType** – Defines mapping for enumerated types.

@**Temporal –** for persistent fields: Date & Calendar

@**OrderBy** – when the association or collection is retrieved.  
 orderby\_list::= orderby\_item [,orderby\_item]\*  
 orderby\_item::= [property\_or\_field\_name] [ASC | DESC]  
if not specified = by the primary key

@**MapKey** – for java.utils.Map when map key is itself the PK or a persistent field or property of the entity that is the value of the map.  
The *MapKeyClass* annotation is not used when MapKey is specified and vice versa.

**CascadeType** – only for parent-child associations = Parent entity state transition being cascaded to its Child entities.  
All – Toate restu.  
 @OneToOne(mappedBy = "post",  
        cascade = CascadeType.ALL, orphanRemoval = true)  
 private PostDetails details;  
si pe @OneToMany !

1. **CascadeType.PERSIST** : cascade type presist means that save() or persist() operations cascade to related entities.
2. **CascadeType.MERGE** : cascade type merge means that related entities are merged when the owning entity is merged. (Update)
3. **CascadeType.REFRESH** : cascade type refresh does the same thing for the refresh() operation. (PULL)
4. **CascadeType.REMOVE** : cascade type remove removes all related entities association with this setting when the owning entity is deleted.
5. **CascadeType.DETACH** : cascade type detach detaches all related entities if a “manual detach” occurs.
6. **CascadeType.ALL** : cascade type all is shorthand for all of the above cascade operations.

**FetchType - EAGER / LAZY**

Cu LAZY – se face load doar la datele din entitaea parinte.  
Cu EAGER se face load si la entitatile fiu. Ex. Lista studenti -> universitate.

LAZY = fetch when needed  
EAGER = fetch immediately@Relatie(fetch = FetchType.LAZY)

FetchType valueOf(String name)  
Returns the enum constant of this type with the specified name.

FetchType[] values()  
Returns an array containing the constants of this enum type.

**@GeneratedValue**(strategy = ***GenerationType***.IDENTITY)

*GenerationType*: Defines the types of primary key generation strategies.  
*GenerationType* **AUTO**:   
Indicates that the persistence provider should pick an appropriate strategy for the particular database. If table has defined any default value or it has defined any auto increment in table.  
*GenerationType* **IDENTITY**Primary keys for the entity will be chosen using DB identity column. allows an integer/bigint column to be auto-incremented on demand.  
*GenerationType* **SEQUENCE**  
Indicates that the persistence provider must assign primary keys for the entity using a database sequence. We need to create a sequence generator in database and refer that name in the code.

@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="course\_seq")

@SequenceGenerator(

name="course\_seq",

sequenceName="course\_sequence",

allocationSize=20

)

*GenerationType* **TABLE**  
Indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.  
=SEQUENCE pe baza unui table.

***DDL –* Data Definition Language**

to specify how EclipseLink generates DDL (Data Definition Language) for the database schema (tables and constraints) on deployment  
In persistence.xml.

<property name="eclipselink.ddl-generation" value="create-tables"/>

Values: create-tables  
 create-or-extend-tables  
 drop-and-create-tables  
 none

You can use create-or-extend-tables only when eclipselink.ddl-generation.output-mode = database.:

<property name="eclipselink.ddl-generation.output-mode" value="database"/>

**Persistence-Unit.**

<persistence-unit name="todos" transaction-type="RESOURCE\_LOCAL">

JPA implementations have the choice of managing transactions themselves (RESOURCE\_LOCAL), or having them managed by the application server's JTA implementation.

Pt JTA -> EntitiManagerFactory- stabilit de el. -> Folosim @PersistenceContext.

Local -> noi stabilim EMF -> Folosim @PersistenceUnit.

The set of entities that can be managed by a given EntityManager instance is defined by a persistence unit. A persistence unit defines the set of all classes that are related or grouped by the application, and which must be colocated in their mapping to a single database.

**EntityManager** – (interface for interaction with persistence context)

Manages the **persistence context**, which is a set of entity instances == For each persistent entity identity = 1 unique entity instance. Persistence context = manages entities.  
*EntityManager* manages the entity instances and their lifecycle.  
*EntityManager* *API* – used to create and remove persistent entity instances, find entities by key, query over entities.  
when *persistence context* ends, entities -> detached = no longer under control of the EM

**EntityManagerFactory** (interface used to interact with the EMF for the persistence unit)

when application finished / shutdown -> app closes EMF -> all entitiy managers = closed state.

**EntitiManager methods:**

[**createQuery**](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#createQuery-javax.persistence.criteria.CriteriaDelete-)() creates an instance of Query : tipuri: Query, TypedQuery

[**flush**](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#flush--)() Synchronize the persistence context to the underlying database.  
= queries for inserting/updating/deleting associated entities are executed in the database.

**persist()** The persist operation must be used only for new entities. Entity new = never been associated with a database row, meaning that there is no table record in the database to match the entity in question.  
Persist takes an entity instance, adds it to the context and makes that instance managed (ie future updates to the entity will be tracked).

**merge()** Merging is required only for detached entities.   
Merge creates a new instance of your entity, copies the state from the supplied entity, and makes the new copy managed. The instance you pass in will not be managed (any changes you make will not be part of the transaction - unless you call merge again).

[**remove**](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#remove-java.lang.Object-)([**Object**](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html?is-external=true) entity) - Remove the entity instance.

[**getTransaction**](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#getTransaction--)()- Return the resource-level EntityTransaction object.

find() - pt a returna un obiect dupa id.

Person person7 = em.find(Person.class, 7); -> Autoboxing; (Integer) 7- boxing. Invers = Unboxing

Em = EntityManager

EntityManager entityManager = entityManagerFactory.createEntityManager();

entityManager.getTransaction().begin();

entityManager.persist( new Event( "Our very first event!", new Date() ) );

entityManager.persist( new Event( "A follow up event", new Date() ) );

entityManager.getTransaction().commit();

entityManager.close();

**Session**

construct used to mediate connections with the database.   
Dialog with relational DB.  
Instance of Database Usage  
Session opens 1 DB connection when is created and holds onto it until session is closed.  
A transaction is an independent service used within the session.  
Implementation of UnitOfWork pattern. UOW – keeps track of everything you do during a business transaction that can affect the DB.

Session facilitates the possibility of transactions?

**Transaction**

set of separate actions; Group of tasks.  
Logical unit (UOW) executed for data (retrieval) or updates -> represents a changes state.  
TRANSACTION must be ACID = Atomic, Consistent, Isolated, Durable

**JPA Query**

**JPQL –** uses the entity object model instead ot the DB tables to define a query.  
wont affect the DB directly  
JPQL works directly with Java classes and instances.  
The [EntityManager.createQuery](http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html" \t "_top) method creates a Query instance from a given JPQL string.  
Invoking [Query.getResultList](http://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html" \l "getResultList()" \t "_top) executes the query and returns a List containing the matching objects.  
TypeQuery interface extends Query Interface.

[**executeUpdate**](https://docs.oracle.com/javaee/7/api/javax/persistence/Query.html#executeUpdate--)(); [**getResultList**](https://docs.oracle.com/javaee/7/api/javax/persistence/Query.html#getResultList--)(); [**getSingleResult**](https://docs.oracle.com/javaee/7/api/javax/persistence/Query.html#getSingleResult--)(); [**setParameter**](https://docs.oracle.com/javaee/7/api/javax/persistence/Query.html#setParameter-int-java.util.Calendar-javax.persistence.TemporalType-)() ;

public class Query extends Object   
Constructs query object constraints.

**Mockito Captor**

Class ArgumentCaptor<T>

Use it to capture argument values for further assertions.

EJB – Enterprise Java Beans

**server-side component that encapsulates the business logic of an application.**

The business logic is the code that fulfills the purpose of the application.

**Benefits**

1. EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services, such as transaction management and security authorization.
2. the beans rather than the clients contain the application’s business logic
3. enterprise beans are portable components, the application assembler can build new applications from existing beans.\

**Session Beans – NOT persistent (its data)**

A session bean encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean’s methods. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

Types: stateful, stateless, and singleton.

*A stateless session bean can implement a web service, but a stateful session bean cannot.*

**Stateful Session Beans**

The instance variables represent the state of a unique client/bean session. = conversational State.  
In a stateful session bean, the instance variables represent the state of a unique client/bean session  
A session bean is not shared; it can have only one client  
The state is retained for the duration of the client/bean session.

**Stateless Session Beans (better for apps with large nr of clients)**

A stateless session bean does not maintain a conversational state with the client.  
When the method is finished, the client-specific state should not be retained.  
Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

**Singleton Session Beans**

A singleton session bean is instantiated once per application and exists for the lifecycle of the application.

**Message-Driven Bean (**resembles a stateless session bean.**)**

A message-driven bean is an enterprise bean that allows Java EE applications to process messages asynchronously. This type of bean normally acts as a JMS message listener, which is similar to an event listener but receives JMS messages instead of events.

**WEB.XML**

**ASTA TREBE daca facem @Inject, cu @EJB nu cred ca mai trebe, se uita singur in proiect dupa EJB-uri.**

deployment descriptor file to determine how URLs map to servlets

A web application's deployment descriptor describes the classes, resources and configuration of the application and how the web server uses them to serve web requests

web.xml defines mappings between URL paths and the servlets that handle requests with those paths

**SERVLET**

A Java servlet is a Java software component that extends the capabilities of a server. Although servlets can respond to any types of requests, they most commonly implement web containers for hosting web applications on web servers

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

**NAMED QUERIES**

Statically defined with predefined unchangeable quert string.

Uses parameters.

[@Entity](https://www.objectdb.com/api/java/jpa/Entity)

[@NamedQuery](https://www.objectdb.com/api/java/jpa/NamedQuery)([name](https://www.objectdb.com/api/java/jpa/NamedQuery/name)="Country.findAll", [query](https://www.objectdb.com/api/java/jpa/NamedQuery/query)="SELECT c FROM Country c")

public class Country {

+ 2 optional elements, lockMode and hint.

[@Entity](https://www.objectdb.com/api/java/jpa/Entity)

@NamedQueries({

[@NamedQuery](https://www.objectdb.com/api/java/jpa/NamedQuery)([name](https://www.objectdb.com/api/java/jpa/NamedQuery/name)="Country.findAll",

[query](https://www.objectdb.com/api/java/jpa/NamedQuery/query)="SELECT c FROM Country c",

hints={@QueryHint(name= QueryHints.REFRESH, value= HintValues.TRUE)}),

[@NamedQuery](https://www.objectdb.com/api/java/jpa/NamedQuery)([name](https://www.objectdb.com/api/java/jpa/NamedQuery/name)="Country.findByName",

[query](https://www.objectdb.com/api/java/jpa/NamedQuery/query)="SELECT c FROM Country c WHERE c.name = :name"),

})

public class Country {

[TypedQuery](https://www.objectdb.com/api/java/jpa/TypedQuery)<Country> query =

em.[createNamedQuery](https://www.objectdb.com/api/java/jpa/EntityManager/createNamedQuery_String_Class_)("Country.findAll", Country.class);

List<Country> results = query.[getResultList](https://www.objectdb.com/api/java/jpa/TypedQuery/getResultList)();

Can be also defined in JPA XML instead of @NamedQuery

Managed classes must be listed in a persistence unit (persistence.xml)

em.[getMetamodel](https://www.objectdb.com/api/java/jpa/EntityManager/getMetamodel)().[managedType](https://www.objectdb.com/api/java/jpa/metamodel/Metamodel/managedType_Class_)(MyEntity.class);

Following the above code ObjectDB will include MyEntity in searching named queries. (in case the entity is unknown).

Quey **HINTS:**

You can specify EclipseLink query hints (JPA query extensions) by:

* Using the @QueryHint annotation
* Including the hints in the orm.xml or eclipselink-orm.xml file
* Using the setHint() method when executing a named or dynamic query (JPQL or Criteria)

List of hints: <http://www.eclipse.org/eclipselink/documentation/2.4/jpa/extensions/queryhints.htm>

**CACHING**

[**https://dzone.com/articles/jpa-caching**](https://dzone.com/articles/jpa-caching)

JPA has 2 caching levels: **Persistence Context & JPA second level (L2) caching**

**Persistence context:**

The JPA Entity Manager maintains a set of Managed Entities in the Persistence Context.

The Entity Manager guarantees that within a single Persistence Context, for any particular database row, there will be only one object instance. However the same entity could be managed in another User's transaction, so you should use either optimistic or pessimistic locking

**JPA second level (L2) caching**

JPA second level (L2) caching shares entity state across various persistence contexts.

*If L2 caching is enabled, entities not found in persistence context, will be loaded from L2 cache, if found.*

The advantages of L2 caching are:  
 avoids database access for already loaded entities  
 faster for reading frequently accessed unmodified entities

More on link.

**MOCK VS SPY**

When using mock objects, the default behavior of the method when not stub is do nothing. Simple means, if its a void method, then it will do nothing when you call the method or if its a method with a return then it may return null, empty or the default value.

While in spy objects, of course, since it is a real method, when you are not stubbing the method, then it will call the real method behavior. If you want to change and mock the method, then you need to stub it.

  @Test

    public void testMockWithStub() {

        //try stubbing a method

        String expected = "Mock 100";

        when(mockList.get(100)).thenReturn(expected);

        assertEquals(expected, mockList.get(100));

    }

    @Test

    public void testSpyWithStub() {

        //stubbing a spy method will result the same as the mock object

        String expected = "Spy 100";

        //take note of using doReturn instead of when

        doReturn(expected).when(spyList).get(100);

        assertEquals(expected, spyList.get(100));

    }

DOCKER & CONTAINERS

= application build and deployment tool.

package an application(code) with all of its dependencies into a deployable standardized unit called **container**. => Container = standardized unit.

application works seamlessly in any environment.

To isolate and organize apps in own “virtual machines”. -> multi apps but no conflicts between.

small, lightweight execution environments that share the operating system kernel.

For combining software components

process isolation

portability

containers provide most of the isolation of virtual machines at a fraction of the computing power.

Namespaces deal with resource isolation for a single process, while cgroups manage resources for a group of processes. Together, cgroups and namespaces were used to create a container technology called, appropriately enough, Linux Containers, or LXC.

A Docker Image is nothing but a blueprint to deploy multiple containers of the same configurations

**PODS**

= UNIT OF DEPLOYMENT

Encapsulates one or more application containers(that share resources = work together = cohesive unit of service.), with shared storage, network and specification for how to run the containers.

1 POD = 1 Instance of a given APP.

In PODS: The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated.

The shared context of a pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container.