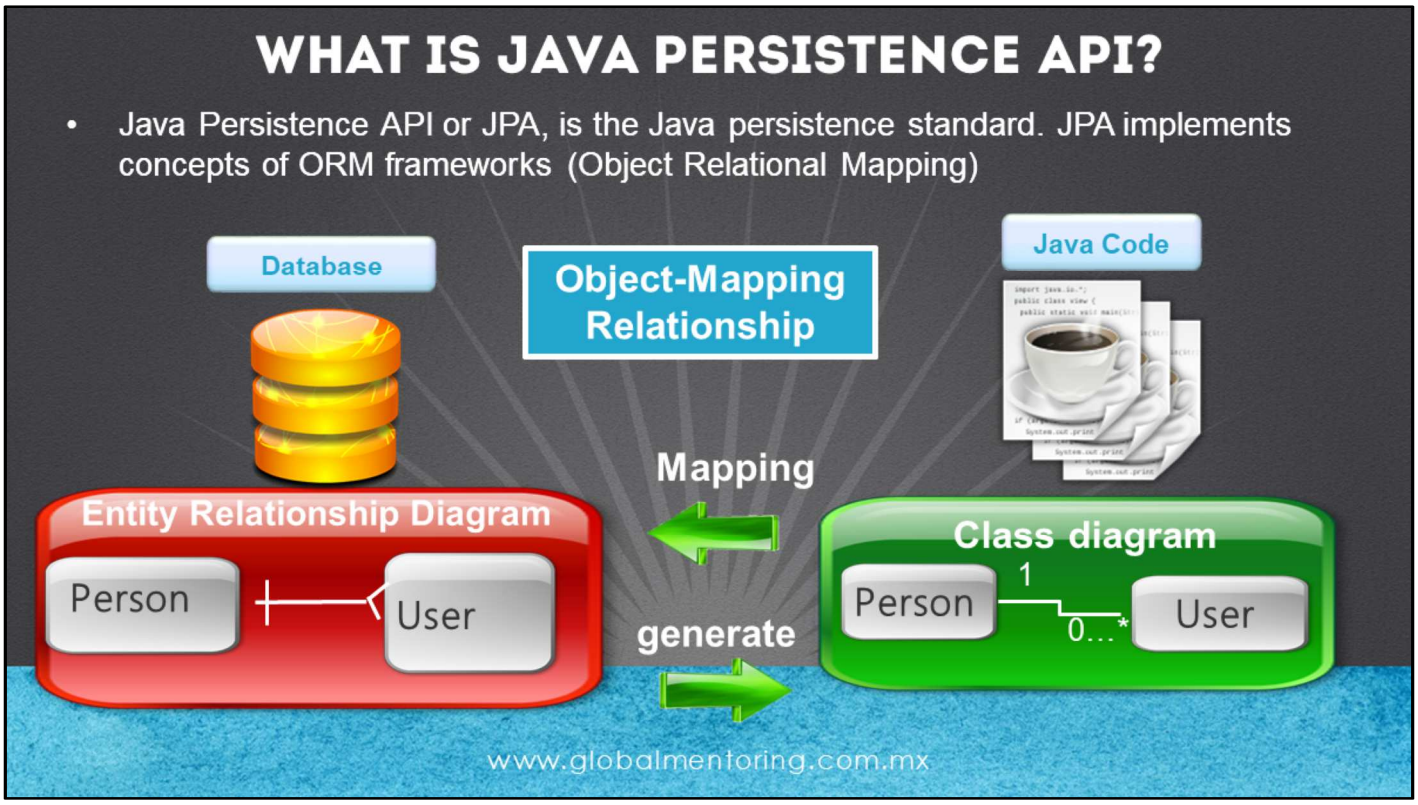
Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson.

We are going to review how to integrate the Java Persistence API (JPA) with Java EE.

Are you ready? Let's go!



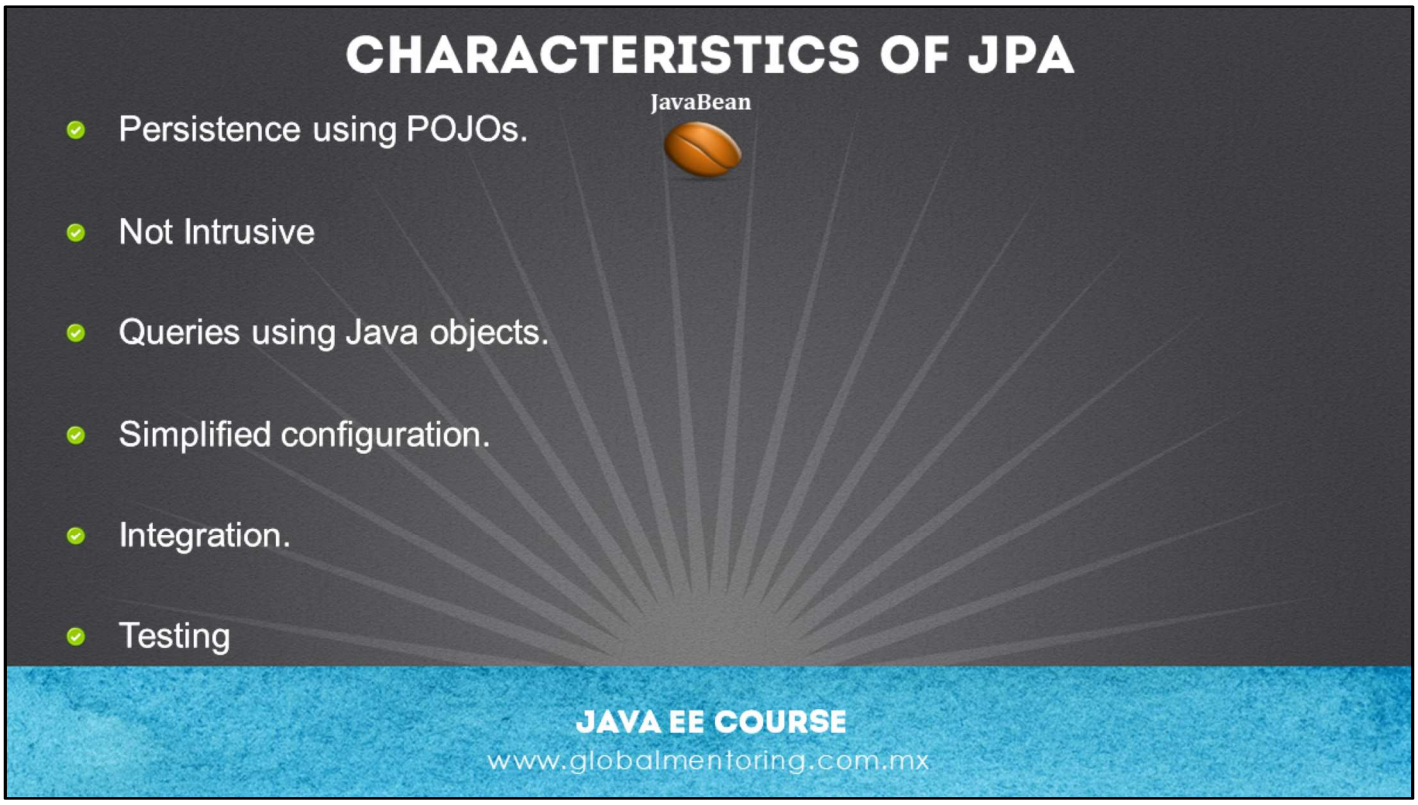
Most of the information of the business applications is stored in relational databases. The persistence of data in Java, and in general in information systems, has been one of the great issues to solve in the world of programming.

When using only JDBC we have the problem of creating too much code to be able to execute a simple query. Therefore, to simplify the process of interaction with a database (select, insert, update, delete), the concept of ORM (Object Relational Mapping) frameworks, such as Hibernate, has been used for several years now.

As we can see in the figure, an ORM framework allows us to "map" a Java class with a Database table. For example, the Person class, when creating an object in memory, we can store it directly in the Person table, simply by executing a line of code: em.persist (person). This has greatly simplified the amount of code to write to the data layer of a business application.

In this lesson we will study Java Persistence API technology, better known as JPA. This Java technology is the standard of persistence in Java, and the good news is that it has several implementations, such as Hibernate, EclipseLink, OpenJPA, among others. So if you already know Hibernate, or some Java persistence framework, many of the concepts we will study in this lesson will be very familiar to you. If you still do not have knowledge of this technology and want to go deeper, we invite you to study the Hibernate and JPA course that we have available in Global Mentoring.

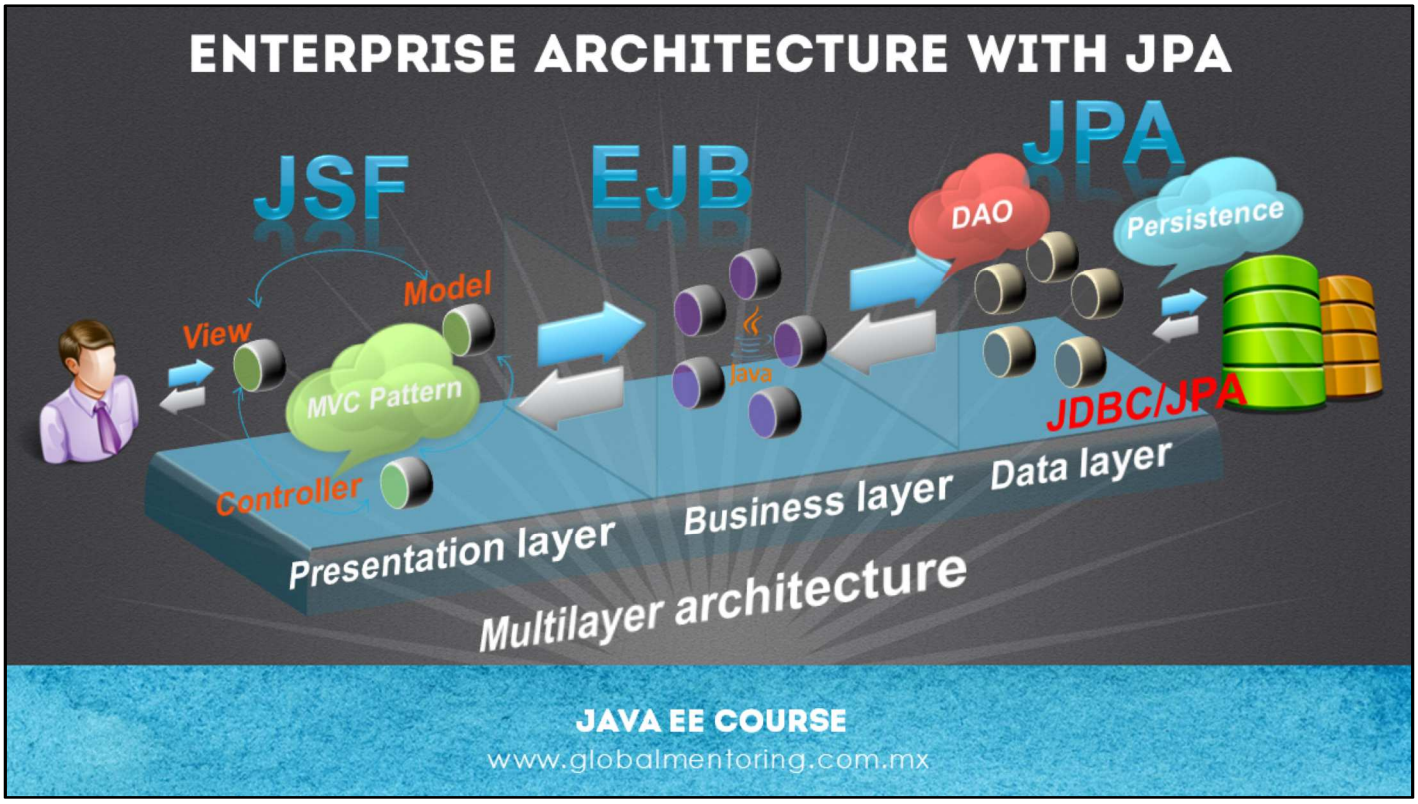
We will learn topics such as: An overview of JPA, Transaction Management using EJB's and JPA, Integrating EJB with JPA, among other topics. This topic is crucial to generate a robust, flexible data layer that can communicate with any relational database.



The idea of the JPA API is to work with Java objects and not with SQL code, in such a way that we can focus on Java code. JPA allows to abstract the communication with the databases and creates a standard to execute queries and manipulate the information of a database.

Features of Java Persistence API:

- **Persistence using POJOs**: This is possibly the most important aspect of JPA, because any Java class can be converted into an entity class, simply adding annotations and / or adding an xml mapping file.
- **Non-Intrusive**: JPA is a separate layer of the objects to persist. Therefore, the Java classes of Entity do not require any particular functionality or knowledge of the existence of JPA, so it is non-intrusive.
- **Queries using Java Objects**: JPA allows you to execute expressed queries in terms of Java objects and their relationships, without the need to use the SQL language. The queries are translated by the JPA API into the equivalent SQL code.
- **Simple configuration**: Many of the JPA options are configured with default options, however if we want to customize them, it is very simple, either with annotations or through configuration xml files.
- **Integration**: Because Java enterprise architectures are multilayer by nature, transparent integration is very valuable for Java and JPA programmers allows integration with other layers in a very simple way.
- **Testing**: With JPA it is now possible to perform unit tests, or use any class with a main method outside the server, simply using the Java Standard Version. This allows to reduce the development times of business applications considerably.



In the figure we can see the role of JPA in a Java Enterprise architecture. This technology applies directly to the data layer, which handles tasks such as:

- Information retrieval through queries (select
- Handling information of Java objects in the respective database tables (insert, update, delete)
- Management of a persistence unit for the creation and destruction of connections to the database
- Transaction management, respecting the propagation scheme defined in the business layer in the Session EJBs.
- Portability to other databases with less impact, as well as low coupling with the other business layers.
- Among several other tasks.

In addition, to perform the tasks of persistence, we can use design patterns such as:

- DAO (Data Access Object): This design pattern usually defines an interface and an implementation of said interface, to perform the most common operations with the respective Entity. For example, for the Person entity, we will generate the DaoPerson interface, and add the methods addPerson, modifyPerson, deletePerson, findAllPeople, etc.
- DTO (Data Transfer Object): This design pattern allows you to define a class, which is sometimes very similar to the feature class, since it contains the same attributes, but with the aim of transmitting it to the following layers, even the Web layer. Therefore they are known as valuables or transfer objects.

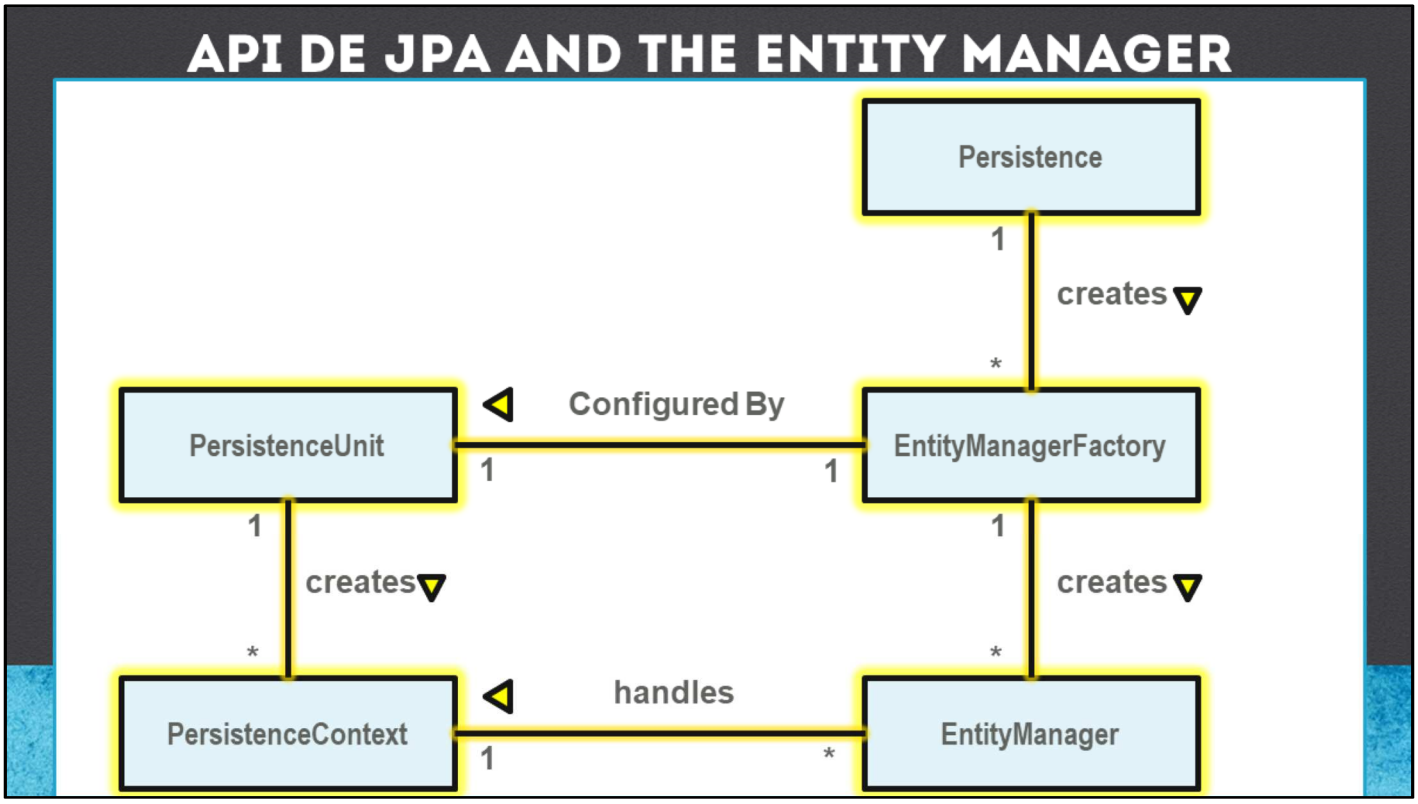
Currently, and with the simplification of the layers of a business architecture, it is optional to use these and other business design patterns, however, many Java applications have been built with these patterns, so it is worth understanding what they are used for. and how to apply them.

In the early versions of J2EE, there was the concept of Entity Beans for persistence management. However, this technology was complicated for real world systems, resulting in low performance. Likewise, querying using Entity objects was very complicated, a Java application server was needed to execute an Entity type EJB, there were no unit tests, and therefore any change in our code involved a new deploy, consuming a lot of time just to test our persistence code, among several more details.

However, the concept of Entity Beans was very good from the point of view of object-oriented programming. Hibernate was one of the first frameworks to incorporate many of the features that we can use today with JPA. Hibernate simplified by far the technology of the Entity EJBs, and although it is not a standard, in many Java applications, choosing Hibernate is the guarantee of an excellent persistence framework.

As of today, a class known as Entity is simply a POJO, and in combination with the use of annotations, it is sufficient to convert it into an Entity class, which represents a record of a database table. This type of concepts, inherited from frameworks such as Hibernate, TopLink, JDO, among others, contributed to what we know today as the Java persistence standard known as JPA.

The JPA API can be used in a standard Java application or in a Java Enterprise or Web server. Now it is already possible to perform unit tests on our Entity classes and Queries on Entity objects, dramatically decreasing the development time of our entity classes, queries, and in general in the creation of the data layer of a business application.



For a feature class to be persistent, a call to the JPA API must be made. In fact, many of the operations are carried out through this API, which is separated from our Entity classes.

In the figure we can see the JPA API, which has as its main element the EntityManager object, this being an interface. An implementation of this interface is the one that really works the persistence work, the synchronization with the database, the transactionality, the validation of mapping, the conversion of Java code to SQL, among many other tasks.

Therefore, a class of Entity, from the point of view of the previous view, is just a normal Java class, which is when linked with an EntityManager, it is persisted in the database.

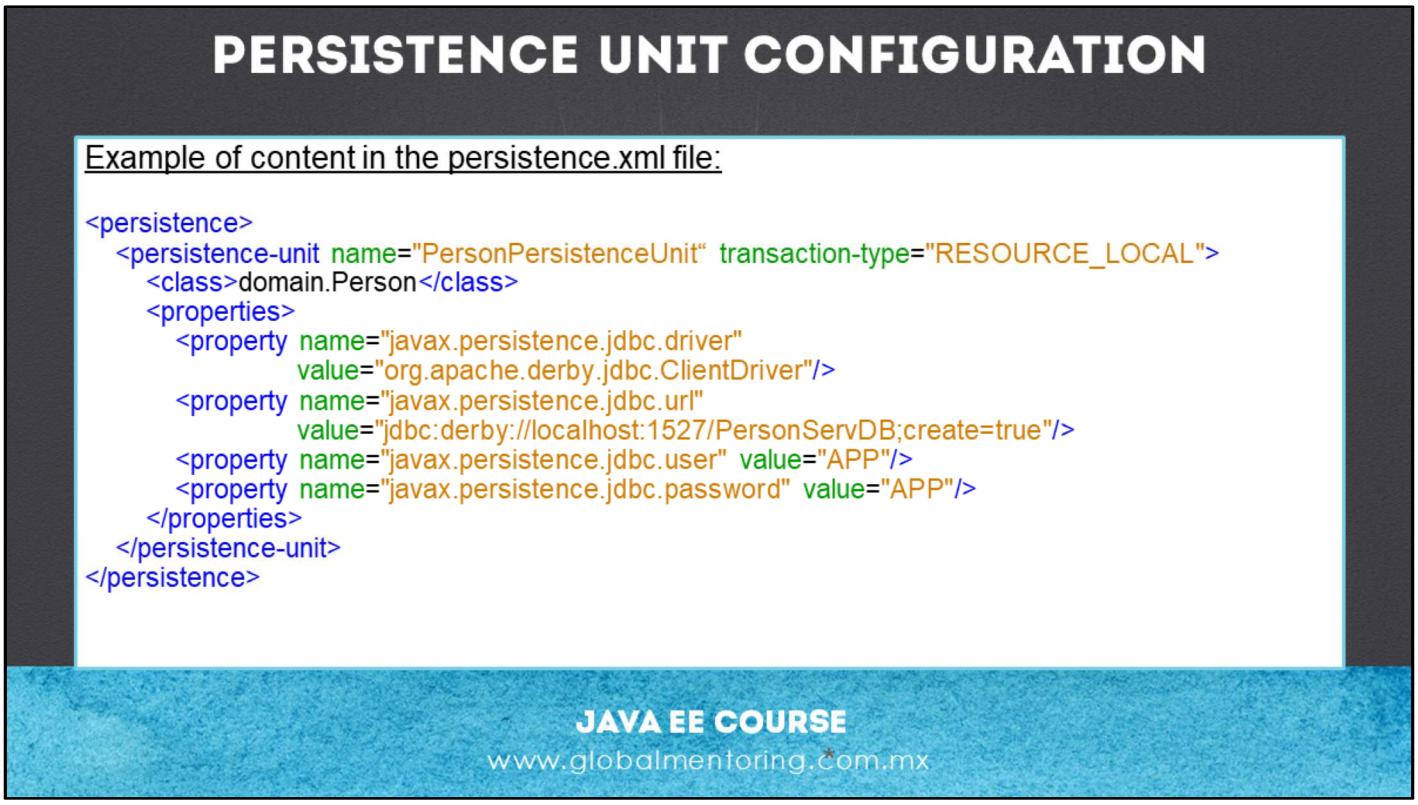
The EntityManager object obtained from an object factory is known as the EntityManagerFactory, and this object is associated with a JPA provider, you can choose between several providers according to the chosen JPA implementation (Hibernate, EclipseLink, OpenJPA, etc.).

The object Persistence unit, is responsible for making the configuration of the provider The medium of an xml file, in addition to defining other elements such as the form of contacts with the Database, the Entity class in the application, if it goes to use JTA for transactional management, among several other characteristics.

In turn, the set of Entity objects managed by JPA in a specific time of the application is known as PersistenceContext, thus JPA ensures that there are no duplicate Entity objects in memory, among other tasks.

An example of how to use the JPA API for an Entity object is shown below:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory (“PersonPersistenceUnit");
EntityManager em = emf.createEntityManager ();
Person person = new Person(15) ;
em.persist (person);
```



To make the configuration of the Persistence Unit, an xml file named persitence.xml must be used.

The name of the persistence-unit element indicates the name of the persistence unit, and it is very important to remember it, since it is the name we will use in our Java code when using the EntityManagerFactory object.

The transaction-type attribute specifies the type of transactionality that will be used, and JTA can be selected as the provider.

You can also specify the classes of Entity of the system, being able to define several classes. If the application is deployed on a Java server, it is not necessary to declare these classes, however, for Java SE applications (Standard Edition) it is necessary to specify the classes of System Entity.

The properties section specifies characteristics of the provider to be used, as well as the data of connection to the database.

When packaging a Java application, the persistence.xml file must be located in the META-INF/persistence.xml path of the .jar file.

USING THE PERSISTENCE UNIT

Example of using the persistence unit and the Entity Manager:

```
@Stateless
public class PersonServiceBean implements PersonService {

    @PersistenceContext(unitName="PersonPersistenceUnit")
    EntityManager em;

    public void addPerson(Persona person) {
        em.persist(person);
    }

    public Person findPerson(int idPerson) {
        return em.find(Person.class, idPerson);
    }

    public Person modifyPersonName(int idPerson, String newName) {
        Person person = em.find(Person.class, idPerson);
        if (person != null) {
            person.setName(newName);
        }
        return person;
    }

    public void deletePerson(int idPerson) {
        Person person = em.find(Person.class, idPerson);
        em.remove(person);
    }
}
```

In the figure we can see an example of a code where we use the persistence unit to obtain the EntityManager object. Once we have obtained a reference to the EntityManager object, we can begin to perform the operations with the Entity objects.

For example, we can perform tasks such as:

- **Insertion**: To persist an entity the persist method of the EntityManager is used. With this method we can generate a record in the database. This record will not be saved until the transaction is complete (commit). Recall that the methods of a Session Bean are transactional by default, this means that when you finish executing the method addPerson and when executing in a Java business container, the commit will be performed automatically. In later lessons we will review the topic of transactions inside and outside an application server.

- **Find**: Once we have a persisted object, we can retrieve the information from the database record using the find method of the EntityManager object, and it is enough to specify the type (class) and the id (primary key) that we are looking for.

- **Modification**: The modification changes a bit, because JPA needs to know first what entity is being worked on, so we need to recover the entity object. Once recovered, we make the necessary modifications, and if the object is in an active transaction, JPA will review automatically if it is necessary to update the registry. The interesting thing is that it is not necessary to call the persist method again, this call is optional.

- **Removal**: Similar to the modification, first the entity with the find method must be recovered, and once in memory, we call the remove method.

These are the basic operations using the EntityManager object over our entity objects. Next we will review some examples to deepen more in this subject.