**What is JPA**

**Table of Content**

* **What is JPA?**
* **What is ORM?**
* **Why use JPA?**
* **JPA Relationship Types**

**What is JPA?**

**The Java Persistence API (JPA) is the Java standard for mapping Java objects to a relational database.**

**JPA is one possible approach to ORM. Via JPA, the developer can map, store, update, and retrieve data from relational databases to Java objects and vice versa.**

**JPA can be used in Java-EE and Java-SE applications. JPA is a specification and several implementations are available.**

**Objet Relational Mapping (ORM)**

**The Object Relational Mapping is the base of JPA, which is all about representing and accessing data in the form of plain Java Objects – called Entities.**

**Hibernate, EclipseLink, and Apache OpenJPA are some of the JPA implementations, out of which, Hibernate is more popular and widely used.**

**Why use JPA?**

**To reduce the burden of writing codes for relational object management of an enterprise application, a programmer follows the JPA Provider framework (an implementation of JPA), which allows easy interaction with the database instance.**

**JPA Relationship Types**

**In a relational database, the relationships between the two tables are defined by foreign keys. Typically, one table has a column that contains the primary key of another table’s row.**

**In JPA, we deal with entity objects that are Java representations of database tables. So we need a different way of establishing a relationship between two entities.**

**JPA entity relationships define how these entities refer to each other.**

**One-to-one relationships**

**The @OneToOne annotation is used to define a one-to-one relationship between two entities.**

**For example, you may have a User entity that contains a user's name, email, and password, but you may want to maintain additional information about a user (such as age, gender, and favorite color) in a separate UserProfile entity.**

**The @OneToOne annotation facilitates breaking down your data and entities this way.**

**The User class below has a single UserProfile instance. The UserProfile maps to a single User instance.**

**@Entity**

**public class User {**

**@Id**

**private Integer id;**

**private String email;**

**private String name;**

**private String password;**

**@OneToOne(mappedBy = "user")**

**private UserProfile profile;**

**...**

**}**

**@Entity**

**public class UserProfile {**

**@Id**

**private Integer id;**

**private int age;**

**private String gender;**

**private String favoriteColor;**

**@OneToOne**

**private User user;**

**...**

**}**

**The JPA provider uses UserProfile's user field to map UserProfile to User.  
The mapping is specified in the mappedBy attribute in the @OneToOne annotation.**

**One-to-many and many-to-one relationships**

**The @OneToMany and @ManyToOne annotations facilitate both sides of the same relationship.**

**Consider an example where a Book can have only one Author, but an Author may have many books.**

**The Book entity would define a @ManyToOne relationship with Author and the Author entity would define a @OneToMany  
relationship with Book.**

**@Entity**

**public class Book {**

**@Id**

**private Integer id;**

**private String name;**

**@ManyToOne**

**@JoinColumn(...)**

**...**

**}**

**@Entity**

**public class Author {**

**@Id**

**@GeneratedValue**

**private Integer id;**

**private String name;**

**@OneToMany(mappedBy = "author")**

**private List<Book> books = new ArrayList<>();**

**...**

**}**

**In this case, the Author class maintains a list of all of the books written by that author, and the Book class maintains a reference to its single author.**

**Additionally, the @JoinColumn specifies the name of the column in the Book table to store the ID of the Author.**

**Many-to-many relationships**

**Finally, the @ManyToMany annotation facilitates a many-to-many relationship between entities. Here's a case where a Book entity has multiple Authors:**

**@Entity**

**public class Book {**

**@Id**

**private Integer id;**

**private String name;**

**@ManyToMany**

**@JoinTable(name = "BOOK\_AUTHORS",**

**joinColumns = @JoinColumn(name = "BOOK\_ID"),**

**inverseJoinColumns = @JoinColumn(name = "AUTHOR\_ID"))**

**private List<Author> authors = new ArrayList<>();**

**...**

**}**

**@Entity**

**public class Author {**

**@Id**

**@GeneratedValue**

**private Integer id;**

**private String name;**

**@ManyToMany(mappedBy = "author")**

**private List<Book> books = new ArrayList<>();**

**...**

**}**

**In this example, we create a new table, BOOK\_AUTHORS, with two columns: BOOK\_ID and AUTHOR\_ID.**

**Using the joinColumns and inverseJoinColumns attributes tell your JPA framework how to map these classes in a many-to-many relationship.**

**The @ManyToMany annotation in the Author class references the field in the Book class that manages the relationship; namely the Author's property.**

**EclipseLink and Hibernate**

**As I said before, you need a JPA provider, if you want to use the JPA specification in your project. It implements the interfaces as defined by the specification. The most popular ones are EclipseLink and Hibernate.**

**One advantage of the standardized API provided by JPA is that you just need to add its implementation at runtime and that you can replace it with a different one without changing any code. The standardized API makes EclipseLink and Hibernate interchangeable.**

**So, why do you need different implementations?**

**The JPA implementations are managed by independent teams, and you can choose the one that provides the best performance or support for your application and technology stack. They also differentiate themselves by providing additional, non-standard functionalities. This is often used to drive innovation. Today’s popular, proprietary feature might be the first step to the next addition to the JPA standard. Using any of these proprietary features, obviously, makes it a lot harder to replace a specific JPA implementation.**

**EclipseLink**

**EclipseLink is JPA’s reference implementation and implements JPA version 2.2. It was one of the first projects that became part of EE4J.**

**The easiest way to add EclipseLink to your project is to use the following Maven coordinates.**

**<dependency>**

**<groupId>org.eclipse.persistence</groupId>**

**<artifactId>eclipselink</artifactId>**

**<version>2.7.1</version>**

**</dependency>**

**Interesting proprietary features**

**In addition to the features defined by the JPA standard, EclipseLink also offers several interesting, proprietary features, like:**

**Handling of database change events**

**Composite persistence units to map entities to tables in multiple databases**

**Support for multi-tenancy**

**Hibernate**

**Hibernate is Red Hat’s very popular implementation of the JPA specification. It implements almost all features defined by JPA 2.2 and will release a fully compliant version soon.**

**The following Maven dependency adds Hibernate to your project.**

**<dependency>**

**<groupId>org.hibernate</groupId>**

**<artifactId>hibernate-core</artifactId>**

**<version>5.1.11</version>**

**</dependency>**

**Interesting proprietary features**

**Similar to EclipseLink, Hibernate provides a bunch of interesting, proprietary features, like:**