1. What is ORM?

The term `Object/Relational Mapping` refers to the technique of mapping data from an object model representation to a relational data model representation (and vice versa).

For example, the below diagram shows an `Object Relational Mapping` between **Student** Java class and **student** table in the database.

2. What is Hibernate Framework?

Hibernate is an `Object/Relational Mapping` solution for Java environments. Object-relational mapping or ORM is the programming technique to map application domain model objects to the relational database tables. Hibernate is a java based ORM tool that provides a framework for mapping application domain objects to the relational database tables and vice versa.

For example, the below diagram shows a Hibernate `Object Relational Mapping` between **Student** Java class and **student** table in the database.

Hibernate provides a reference implementation of `Java Persistence API`, which makes it a great choice as an ORM tool with the benefits of loose coupling.

3. What is Java Persistence API (JPA)?

The Java Persistence API (JPA) is the specification of Java that is used to persist data between Java object and the relational database table. JPA acts as a bridge between object-oriented domain models and relational database systems.

As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. Therefore, ORM tools like **Hibernate**, TopLink, and iBatis implement JPA specifications for data persistence.

JPA specifications are defined with annotations in a `javax.persistence` package. Using JPA annotation helps us in writing implementation-independent code.

4. How does Hibernate relate to JDBC?

Hibernate uses **JDBC** for all database communications. Hibernate internally uses **JDBC** to interact with the database.

5. Explain Hibernate Architecture?

Hibernate, as an ORM solution, effectively "sits between" the Java application data access layer and the Relational Database, as can be seen in the diagram above.

The Java application makes use of the Hibernate APIs to load, store, query, etc its domain data.

The below diagram shows important interfaces and classes of Hibernate framework:

SessionFactory (org.hibernate.SessionFactory)

SessionFactory is a thread-safe (and immutable) representation of the mapping of the application domain model to a database.

SessionFactory acts as a factory for `org.hibernate.Session` instances. So the `EntityManagerFactory` is the JPA equivalent of a `SessionFactory` and basically, those two converge into the same `SessionFactory` implementation.

A `SessionFactory` is very expensive to create, so, for any given database, the application should have only one associated SessionFactory.

The `SessionFactory` maintains services that Hibernate uses across all `Session` (s) such as second-level caches, connection pools, transaction system integrations, etc.

Session (org.hibernate.Session)

The Session is a single-threaded, short-lived object conceptually modeling a "Unit of Work". In JPA nomenclature, the `Session` is represented by an `EntityManager` .

Behind the scenes, the Hibernate Session wraps a JDBC `java.sql.Connection` and acts as a factory for `org.hibernate.Transaction` instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

Transaction (org.hibernate.Transaction)

The Transaction is a single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries. The `EntityTransaction` is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).

6. What are the important benefits of using Hibernate Framework?

Some of the important benefits of using **hibernate framework** are:

- Hibernate is awesome as it eliminates all the boiler-plate code that comes with **JDBC** and takes care of managing resources, so we can focus on business logic.
- **Hibernate framework** provides support for XML as well as JPA annotations, which makes our code implementation independent.
- **Hibernate framework** provides a powerful query language that is **Hibernate Query Language(HQL)** which is similar to SQL. However, HQL is fully object-oriented and understands concepts like inheritance, polymorphism, and association, etc.
- Hibernate is easy to integrate with other Java EE frameworks, it's so popular that Spring Framework provides built-in support for integrating Hibernate with Spring applications.
- Hibernate supports lazy initialization using proxy objects and performs actual database queries only when it's required.
- Hibernate framework provides a cache (first level and second level) that helps us in getting the better performance of the Java applications.
- Hibernate framework supports native SQL queries so Hibernate is suitable for a database vendor-specific feature.
- Overall Hibernate is the best choice in the current market for the ORM tool, it contains all the features that you will ever need in an ORM tool.

7. What are the advantages of Hibernate over JDBC?

Some of the important advantages of Hibernate framework over **JDBC** are:

- Hibernate removes a lot of boiler-plate code that comes with **JDBC** API, the code looks cleaner and readable.
- Hibernate supports inheritance, associations, and collections. These features are not present with **JDBC** API.
- Hibernate implicitly provides transaction management, in fact, most of the queries can't be executed outside a transaction. In JDBC API, we need to write code for transaction management using commit and rollback.
- **JDBC** API throws `SQLException` which is a checked exception, so we need to write a lot of try-catch block code. Most of the time it's redundant in every JDBC call and used for transaction management. Hibernate wraps JDBC exceptions and throws JDBCException or HibernateException un-checked exception, so we don't need to write code to handle it. Hibernate built-in transaction management removes the usage of try-catch blocks.
- **Hibernate Query Language (HQL)** is more object-oriented and close to a **Java programming language**. For **JDBC**, we need to write native SQL queries.
- Hibernate supports caching that is better for performance, **JDBC** queries are not cached hence performance is low.
- Hibernate provides an option through which we can create database tables too, for JDBC tables must exist in the database.
- Hibernate configuration helps us in using JDBC-like connection as well as JNDI `DataSource` for a connection pool. This is a very important feature in enterprise applications and completely missing in **JDBC** API.
- Hibernate supports JPA annotations, so the code is independent of the implementation and easily replaceable with other ORM tools. **JDBC** code is very tightly coupled with the application.

8. What is Hibernate Configuration File?

Hibernate configuration file includes the database-specific configurations that are required to begin database connection using hibernate framework.

The configuration file is usually an XML document with the name `hibernate.cfg.xml`. We provide database credentials or JNDI resource information in the hibernate configuration XML file. Some other important parts of hibernate configuration file are Dialect information, so that hibernate knows the database type and mapping file or class details.

Here is a sample hibernate.cfg.xml file:

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- JDBC Database connection settings -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/hibernate_db?useSSL=false</property>
        <property name="connection.username">root</property>
        <property name="connection.password">root</property>
        <!-- JDBC connection pool settings ... using built-in test pool -->
        <property name="connection.pool_size">1</property>
        <!-- Select our SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>
        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create-drop</property>
        <!-- dbcp connection pool configuration -->
        <property name="hibernate.dbcp.initialSize">5</property>
        <property name="hibernate.dbcp.maxTotal">20</property>
        <property name="hibernate.dbcp.maxIdle">10</property>
        <property name="hibernate.dbcp.minIdle">5</property>
        <property name="hibernate.dbcp.maxWaitMillis">-1</property>
        <mapping class="net.javaguides.hibernate.entity.Student" />
    </session-factory>
</hibernate-configuration>
```

Read more about Hibernate XML configuration file at **http://www.javaguides.net/2018/11/hibernate-5-xml-configuration-example.html**

In case you prefer to use Java-based configuration, check out more about Hibernate Java-based configuration at **http://www.javaguides.net/2018/11/hibernate-5-java-configuration-example.html**

9. What are possible ways to configure object-table mapping?

There are two ways to map Java objects to a relational table:

1. Using XML based hibernate mapping
2. Annotation-based hibernate mapping

10. What is Hibernate SessionFactory and How to Configure it?

`SessionFactory` is the factory class used to get the `Session` objects. `SessionFactory` is responsible to read the hibernate configuration parameters and connect to the database and provide `Session` objects. Usually, an application has a single `SessionFactory` instance, and threads servicing client requests obtain Session instances from this factory.

The `SessionFactory` maintains services that Hibernate uses across all Session(s) such as second-level caches, connection pools, transaction system integrations, etc.

The internal state of SessionFactory is immutable, so it's **thread-safe**. Multiple threads can access it simultaneously to get Session instances.

11. Hibernate SessionFactory is a Thread-Safe?

The internal state of `SessionFactory` is immutable, so it's thread-safe. Multiple threads can access it simultaneously to get Session instances.

12. What is Hibernate Session and How to Get it?

A `Session` is used to get a physical connection with a database. The `Session` object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

Behind the scenes, the Hibernate Session wraps a JDBC `java.sql.Connectio` n and acts as a factory for `org.hibernate.Transaction` instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

13. Hibernate Session is a Thread-Safe?

Hibernate Session object is **not thread-safe**, every thread should get its own session instance and close it after its work is finished.

14. What is the difference Between openSession and getCurrentSession?

Hibernate `SessionFactory getCurrentSession()` method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file.

Since this session object belongs to the hibernate context, we don't need to close it. Once the session factory is closed, this session object gets closed.

```
<property name="hibernate.current_session_context_class">thread</property>
```

Hibernate `SessionFactory openSession()` method always opens a new session. We should close this session object once we are done with all the database operations. We should open a new session for each request in a multi-threaded environment.

There is another method `openStatelessSession()` that returns a stateless session, for more details with examples please read Hibernate `openSession vs getCurrentSession` .

15. Explain Hibernate Transaction Interface

In the Hibernate framework, we have a `Transaction` interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA, JDBC).

A transaction is associated with a `Session` and instantiated by calling session. `beginTransaction()` .

The methods of Transaction interface are as follows:

- `void begin()` - starts a new transaction.

- `void commit()` - ends the unit of work unless we are in FlushMode.NEVER.
- `void rollback()` - forces this transaction to rollback.
- `void setTimeout(int seconds)` - it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
- `boolean isAlive()` - checks if the transaction is still alive.
- `void registerSynchronization(Synchronization s)` - registers a user synchronization callback for this transaction.
- `boolean wasCommited()` - checks if the transaction is committed successfully.
- `boolean wasRolledBack()` - checks if the transaction is rolled back successfully.

Read more about the Transaction interface at **Hibernate API Java doc**.

16. What is the Requirement for a Java Object to Become a Hibernate Entity Object?

Let's discuss what are the rules or requirements to create a JPA entity class. An entity class must follow these requirements.

- The class must be annotated with the `javax.persistence.Entity` annotation.
- The class must have a `public` or `protected`, `no-argument` constructor. The class may have other constructors.
- The 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, such as through a session bean's remote business interface, 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. Clients must access the entity's state through accessor or business methods.

Read more about Hibernate/JPA entity at **http://www.javaguides.net/2018/12/jpa-entity-class-basics.html**

17. What is the Difference Between Hibernate Session get() and load() method?

1. The key difference between the **get() method** and **load() method** is that **load() method** will throw an exception if an object with `id` passed to them is not found, but **get() method** will return null.

2. The second important difference is that load() method can return proxy without hitting the database unless required (when you access any attribute other than id) but the get() method always go to the database, so sometimes using the load() method can be faster than the get() method.

Note: Use the load() method, if you know the object exists, and get() method if you are not sure about the object's existence.

Read more about the get() and load() methods at **Hibernate 5 - get(), load() and byId() Method Examples**

18. What is difference between Hibernate save(), saveOrUpdate() and persist() methods?

1. Hibernate **save() method** can be used to save entities to the database. The problem with the **save() method** is that it can be invoked without a transaction and if we have mapping entities, then only the primary object gets saved causing data inconsistencies. Also, save returns the generated id immediately.

2. Hibernate **persist() method** is similar to the **save() method** with a transaction. It's better to use **persist() method** than the **save() method** because we can't use it outside the boundary of a transaction, so all the object mappings are preserved. Also, `persist()` method doesn't return the generated id immediately, so data persistence happens when needed.

3. Hibernate **saveOrUpdate() method** results into insert or update queries based on the provided data. If the data is present in the database, the update query is executed. We can use **saveOrUpdate()** without transaction also, but again you will face the issues with mapped objects not getting saved if a session is not flushed.

19. Why we should not Make an Entity Class final?

Hibernate uses proxy classes for lazy loading of data, only when it's needed. This is done by extending the entity bean, if the entity bean will be final then lazy loading will not be possible, hence low performance.

20. What is HQL and What are its Benefits?

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.

HQL queries are translated by Hibernate into conventional SQL queries, which in turn perform an action on a database. Although you can use SQL statements directly with Hibernate using Native SQL, I would recommend using HQL whenever possible to avoid database portability hassles and to take advantage of Hibernate's SQL generation and caching strategies.

Hibernate query language is case-insensitive except for java class and variable names.

The HQL queries are cached but we should avoid it as much as possible, otherwise, we will have to take care of associations. However, it's a better choice than a native SQL query because of the Object-Oriented approach.

Read more about HQL at **http://www.javaguides.net/2018/11/hibernate-query-language-basics.html**

21. What is Named Query in Hibernate?

In Hibernate, a named query is a `JPQL` or `SQL` expression with a predefined unchangeable query string. You can define a named query either in hibernate mapping file or in an entity class.

Basically, named queries in hibernate is a technique to group the HQL statements in a single location, and lately refer them by some name whenever the need to use them. It helps largely in code cleanup because these HQL statements are no longer scattered in the whole code.

Apart from the above, below are some minor advantages of named queries:

1. **Fail fast:** Their syntax is checked when the session factory is created, making the application fail fast in case of an error.
2. **Reusable:** They can be accessed and used from several places which increases re-usability.

Hibernate Named Queries can be defined in Hibernate mapping files or through the use of JPA annotations `@NamedQuery` .

The following is an example of the definition of a named query in the JPQL/HQL:

```
@NamedQuery(
        name="findAllCustomersWithName",
        query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

Hibernate provides support to create named queries using Native SQL. Here are `@NamedNativeQuery` and `@NamedNativeQueries` annotations to create a named SQL queries.

Read more about Named Query at **http://www.javaguides.net/2018/11/hibernate-5-named-query-tutorial-with-examples.html**.

22. What are the benefits of Named SQL Query?

Hibernate Named Query helps us in grouping queries at a central location rather than letting them scattered all over the code.

Hibernate Named Query syntax is checked when the hibernate session factory is created, thus making the application fail fast in case of any error in the named queries. Hibernate Named Query is global, which means once defined it can be used throughout the application.

However one of the major disadvantages of a Named query is that it's hard to debug because we need to find out the location where it's defined.

23. What is Cascading and What Are Different Types of Cascading?

Cascading is about persistence actions involving one object propagating to other objects via an association. Cascading can apply to a variety of Hibernate actions, and it is typically transitive.

The "cascade=..." attribute of the annotation that defines the association says what actions should cascade for that association.

JPA allows you to propagate the state transition from a parent entity to a child. For this purpose, the JPA `javax.persistence.CascadeType` defines various cascade types:

- `ALL` - cascades all entity state transitions

- `PERSIST` - cascades the entity persist operation.
- `MERGE` - cascades the entity merge operation.
- `REMOVE` - cascades the entity remove operation.
- `REFRESH` - cascades the entity refresh operation.
- `DETACH` - cascades the entity detach operation.

Additionally, the `CascadeType.ALL` will propagate any Hibernate-specific operation, which is defined by the `org.hibernate.annotations.CascadeType` enum:

- `SAVE_UPDATE` - cascades the entity saveOrUpdate operation.
- `REPLICATE` - cascades the entity replicate operation.
- `LOCK` - cascades the entity lock operation.

Cascading only makes sense only for **Parent-Child** associations (the Parent entity state transition being cascaded to its Child entities). Cascading from Child to Parent is not very useful and usually, it's a mapping code smell.

The following examples will explain some of the aforementioned cascade operations using the following entities:

```
@Entity
public class Person {

    @Id
    private Long id;

    private String name;

    @OneToMany(mappedBy = "owner", cascade = CascadeType.ALL)
    private List < Phone > phones = new ArrayList < > ();

    //Getters and setters are omitted for brevity

    public void addPhone(Phone phone) {
        this.phones.add(phone);
        phone.setOwner(this);
    }
}


@Entity
public class Phone {

    @Id
    private Long id;

    @Column(name = "`number`")
    private String number;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person owner;

    //Getters and setters are omitted for brevity
}
```

Read more about cascading at **http://www.javaguides.net/2018/11/guide-to-jpa-and-hibernate-cascade-types.html**.

24. What are Design Patterns used in Hibernate framework?

Some of the design patterns used in Hibernate Framework are:

Read more in detail at **https://ramesh-java-design-patterns.blogspot.com/2018/04/design-patterns-used-in-hibernate.html**

25. What are the different types of caches available in Hibernate?

Caching is a mechanism to enhance the performance of a system. It is a buffer memory that lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

Hibernate provides 3 types of caching.

1. **Session Cache** - The session cache caches objects within the current session. It is enabled by default in Hibernate. Objects in the session cache reside in the same memory location.
2. **Second Level Cache** - The second level cache is responsible for caching objects across sessions. When this is turned on, objects will first be searched in the cache and if they are not found, a database query will be fired. The second-level cache will be used when the objects are loaded using their primary key. This includes fetching of associations. Second-level cache objects are constructed and reside in different memory locations. The second-level cache is required to be configured with external cache provider like EhCache.
3. **Query Cache** - Query Cache is used to cache the results of a query. When the query cache is turned on, the results of the query are stored against the combination query and parameters. Every time the query is fired the cache manager checks for the combination of parameters and query. If the results are found in the cache, they are returned, otherwise, a database transaction is initiated.

26. What are the Different States in Hibernate?

1. **transient** - New objects created in the Java program but not associated with any hibernate Session are said to be in the transient state.
2. **persistent** - An object which is associated with a Hibernate session is called a Persistent object. While an object which was earlier associated with Hibernate session but currently it's not associated is known as a detached object. You can call the `save()` or `persist()` method to store those objects into the database and bring them into the Persistent state.
3. **detached** - You can re-attach a detached object to hibernate sessions by calling either `update()` or `saveOrUpdate()` method.

27. Name JPA Annotations?

Answer: **All JPA Annotations: Mapping Annotations** - A quick reference to all JPA mapping annotations.

28. Name Hibernate Annotations?

Answer: **All Hibernate Annotations: Mapping Annotations** - A quick reference to all Hibernate mapping annotations.

29. Explain Hibernate/JPA Primary key Generation Strategies?

The JPA specification supports 4 different primary key generation strategies which generate the primary key values programmatically or use database features, like auto-incremented columns or sequences.

1. `GenerationType.AUTO` - The **GenerationType.AUTO** is the default generation type and lets the persistence provider choose the generation strategy.
2. `GenerationType.IDENTITY` - The **GenerationType.IDENTITY** is the easiest to use but not the best one from a performance point of view.
3. `GenerationType.SEQUENCE` - The **GenerationType.SEQUENCE** is to generates primary key values and uses a database sequence to generate unique values.
4. `GenerationType.TABLE` - The **GenerationType.TABLE** gets only rarely used nowadays. It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into sequential order.

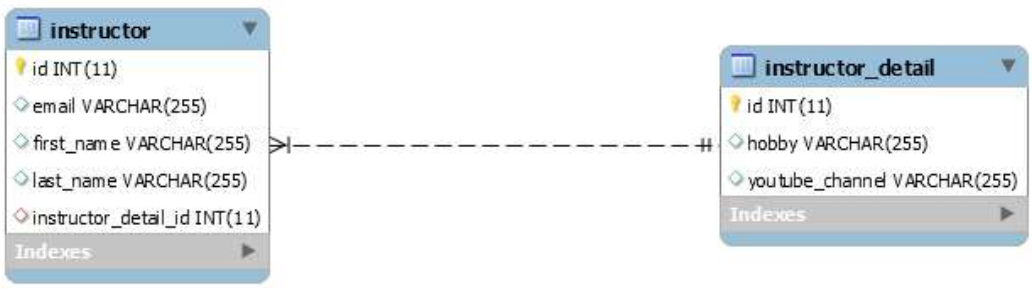
30. Which are the Different Types of Relationships Available in Hibernate Mapping?

There are three different types of relationships that can be implemented in hibernate. They are:

One to One Mapping

It represents the ONE-TO-ONE relationship between two tables.

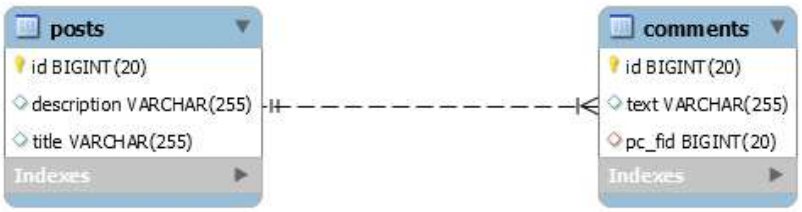
For example, we create `Instructor` and `InstructorDetail` entities and we make a one-to-one mapping between them.



One to Many Mapping

It represents the ONE-TO-MANY relationships between two tables. The one-to-many mapping means that one row in a table is mapped to multiple rows in another table.

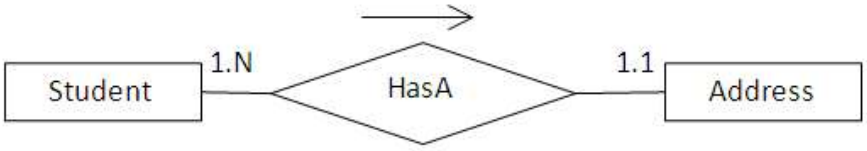
For example, consider the following two tables - **posts** and **comments** of a Blog database schema where the posts table has a one-to-many relationship with the comments table:



Many to One Mapping

It represents the one-to-many or many-to-one relationships between two tables.

For example, consider the following relationship between the `Student` and `Address` entity.

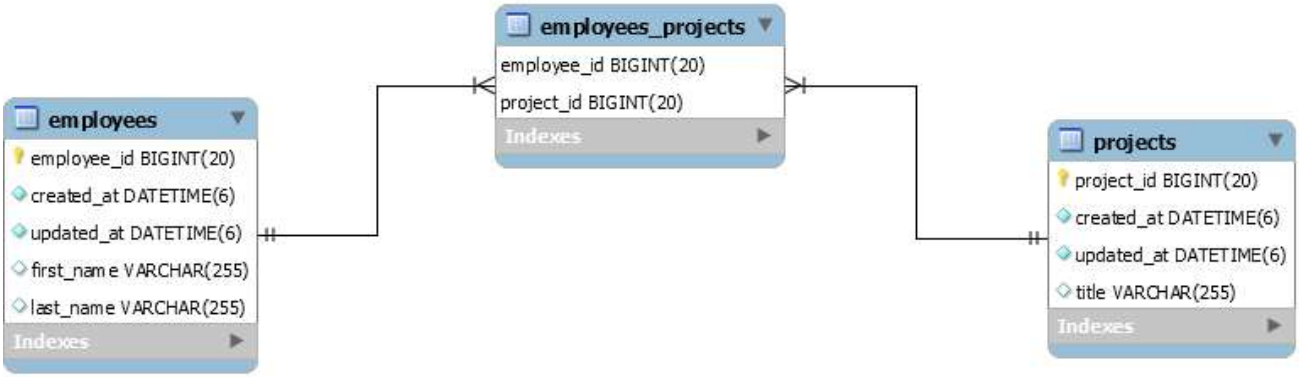


According to the relationship, many students can have the same address.

Many to Many Mapping

It represents the many to many relationships between two tables.

For example, consider the following tables where `employees` and `projects` exhibit a many-to-many relationship with each other -



The many-to-many relationship is implemented using a third table called `employees_projects` which contains the details of the `employees` and their associated `projects`. Note that here Employee is a primary entity.

31. What is the Difference Between JPA and Hibernate?

Here is an answer at **What is the Difference Between JPA and Hibernate?**

> You can learn Hibernate framework from end-to-end at **Hibernate ORM 5 Tutorials**

> You can learn Java Persistence API at **Java Persistence API**