**Hibernate with core java**

What is ORM?

ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting(mapping) data between object oriented programming languages such as Java and relational databases. **(Mapping java Objects to database, to store in database)**

# What is JPA

The Java Persistence API (JPA) **is a specification of Java. It is used to persist data** between Java object and relational database. 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. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.

## Hibernate Framework

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, **ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA** (Java Persistence API) for data persistence.

## Advantages of Hibernate Framework

### 1) Open Source and Lightweight

Hibernate framework is open source under the LGPL license and lightweight.

### 2) Fast Performance

The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.

### 3) Database Independent Query

HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

### 4) Automatic Table Creation

Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

### 5) Simplifies Complex Join

Fetching data from multiple tables is easy in hibernate framework.

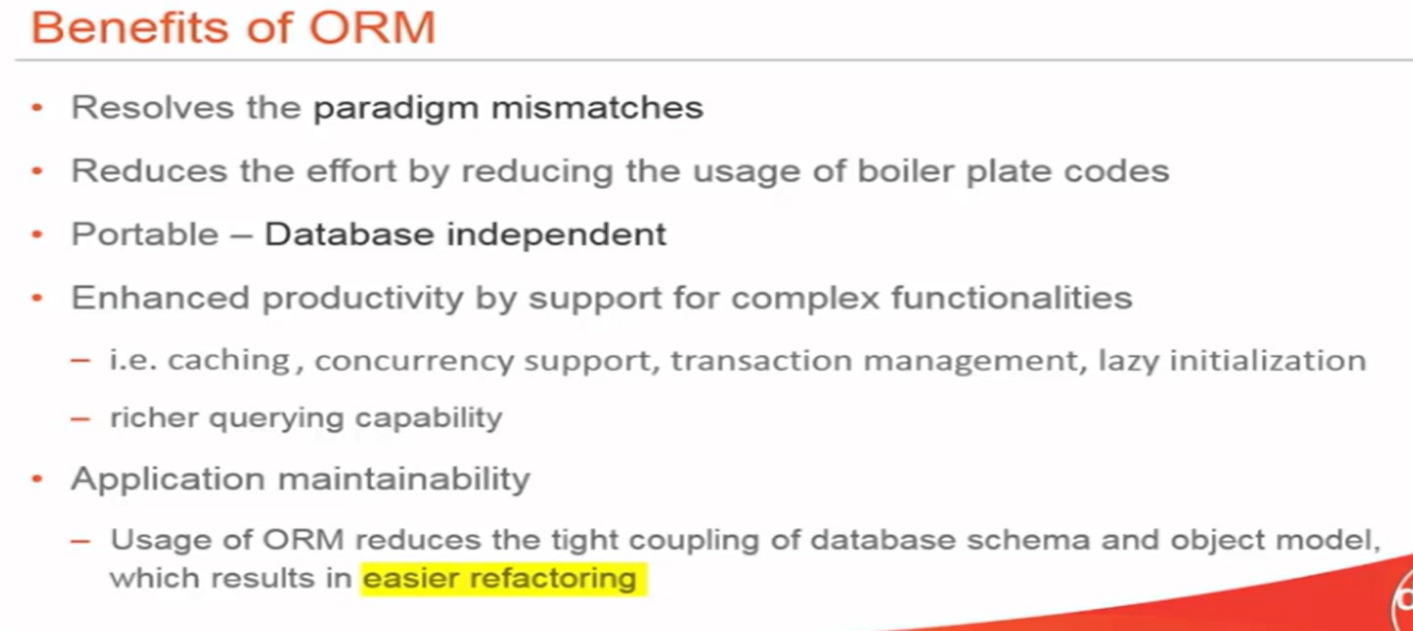
### 6) Provides Query Statistics and Database Status

Hibernate supports Query cache and provide statistics about query and database status.

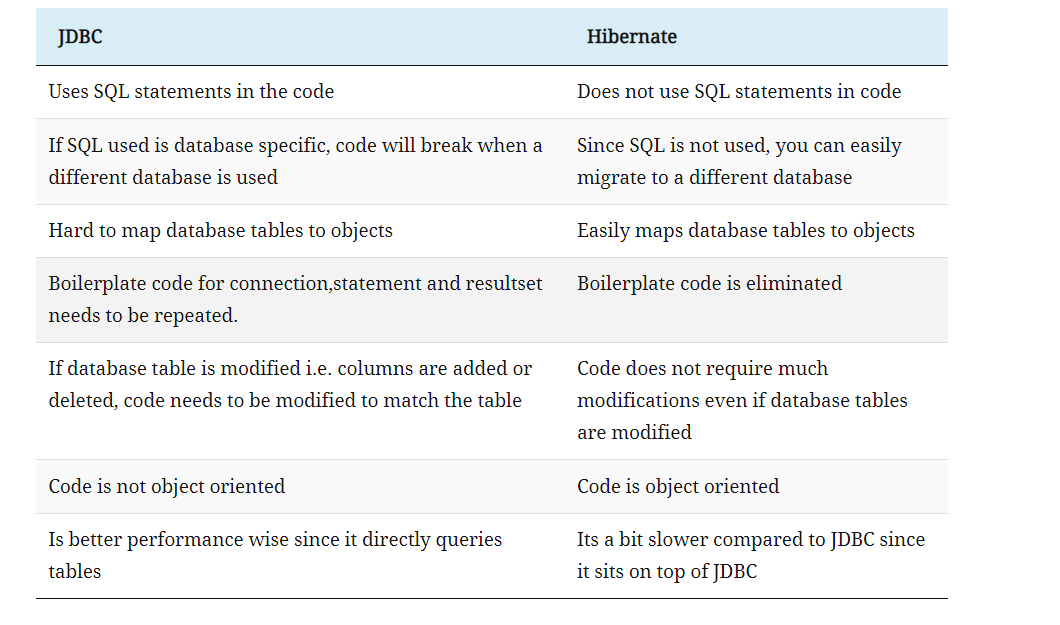
### What are the advantages of Hibernate over JDBC?

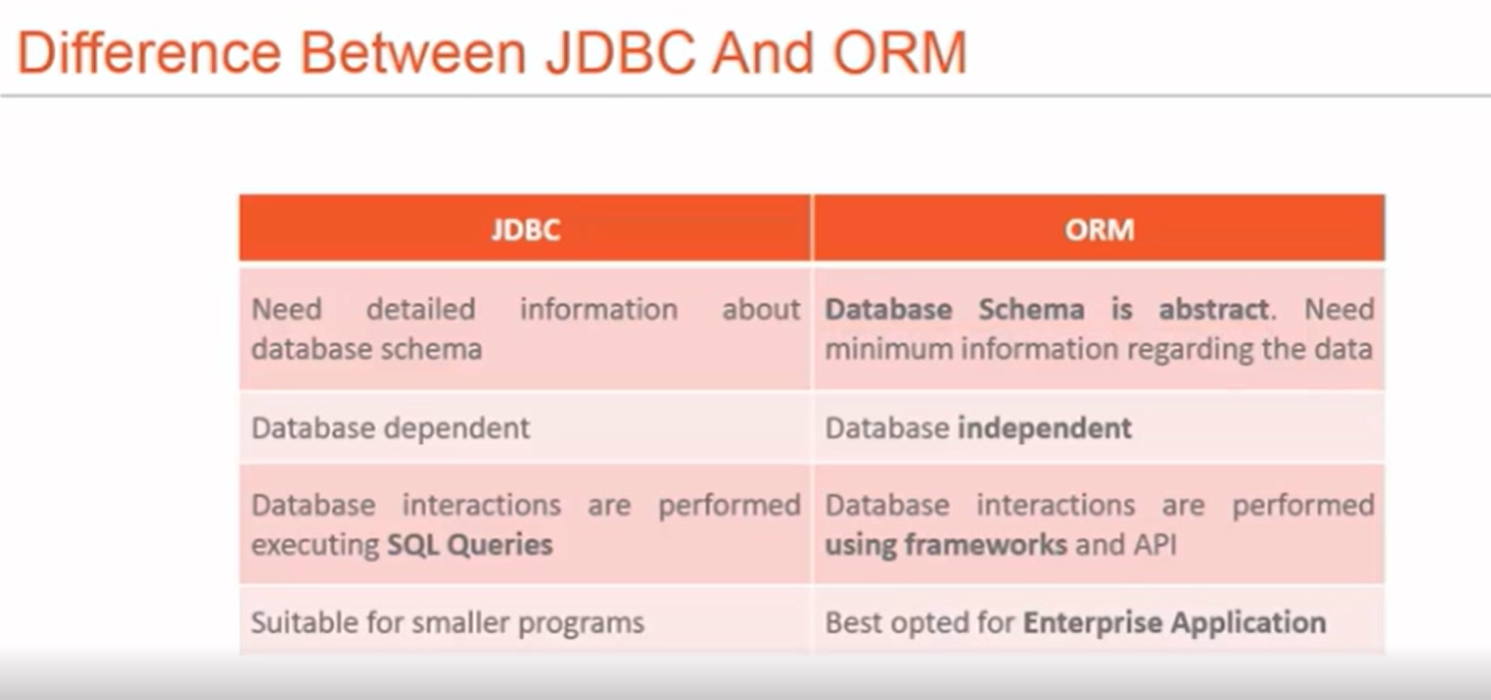
- The advantages of Hibernate over JDBC are listed below:

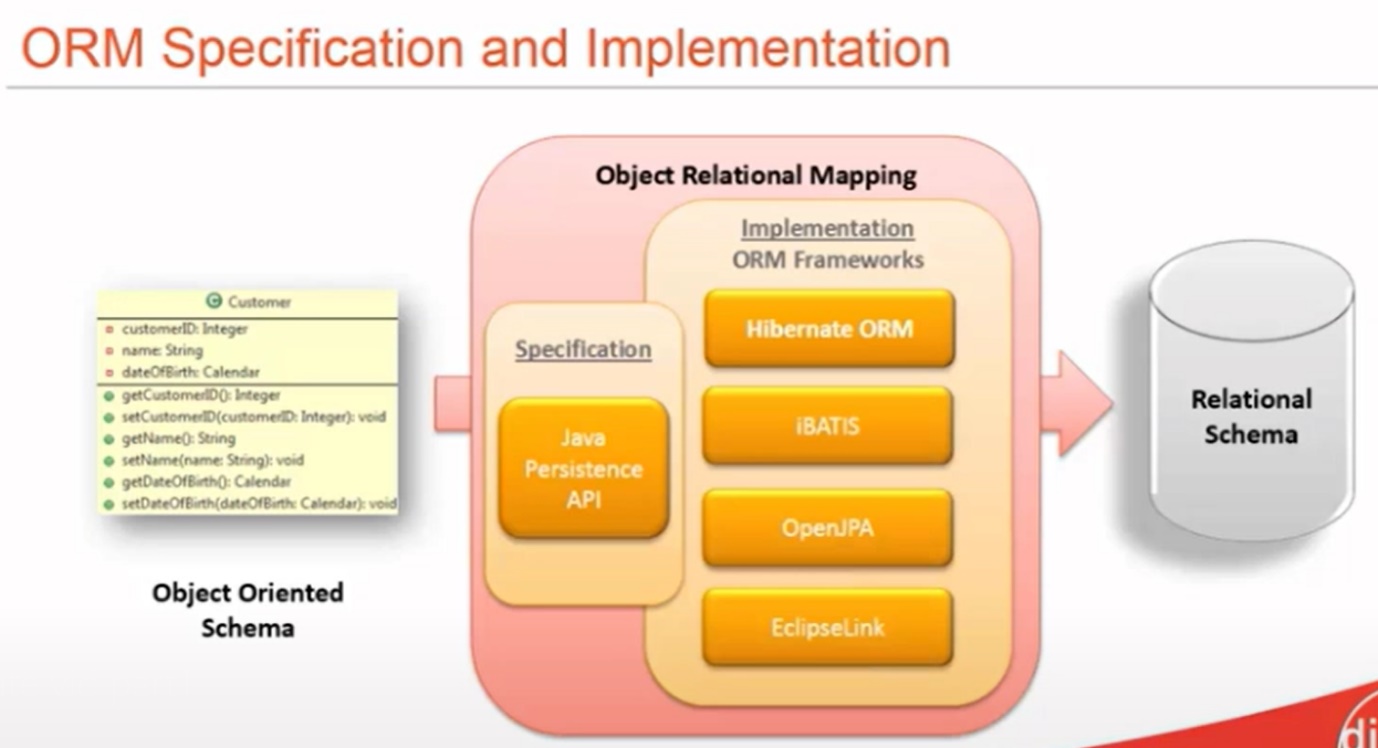
* **Clean Readable Code:** Using hibernate, helps in eliminating a lot of JDBC API-based boiler-plate codes, thereby making the code look cleaner and readable.
* **HQL (Hibernate Query Language):** Hibernate provides HQL which is closer to Java and is object-oriented in nature. This helps in reducing the burden on developers for writing database independent queries. In JDBC, this is not the case. A developer has to know the database-specific codes.
* **Transaction Management:** JDBC doesn't support implicit transaction management. It is upon the developer to write transaction management code using commit and rollback methods. Whereas, Hibernate implicity provides this feature.
* **Exception Handling:** Hibernate wraps the JDBC exceptions and throws unchecked exceptions like JDBCException or HibernateException. This along with the built-in transaction management system helps developers to avoid writing multiple try-catch blocks to handle exceptions. In the case of JDBC, it throws a checked exception called SQLException thereby mandating the developer to write try-catch blocks to handle this exception at compile time.
* **Special Features:** Hibernate supports OOPs features like inheritance, associations and also supports collections. These are not available in JDBC.



## JDBC vs Hibernate



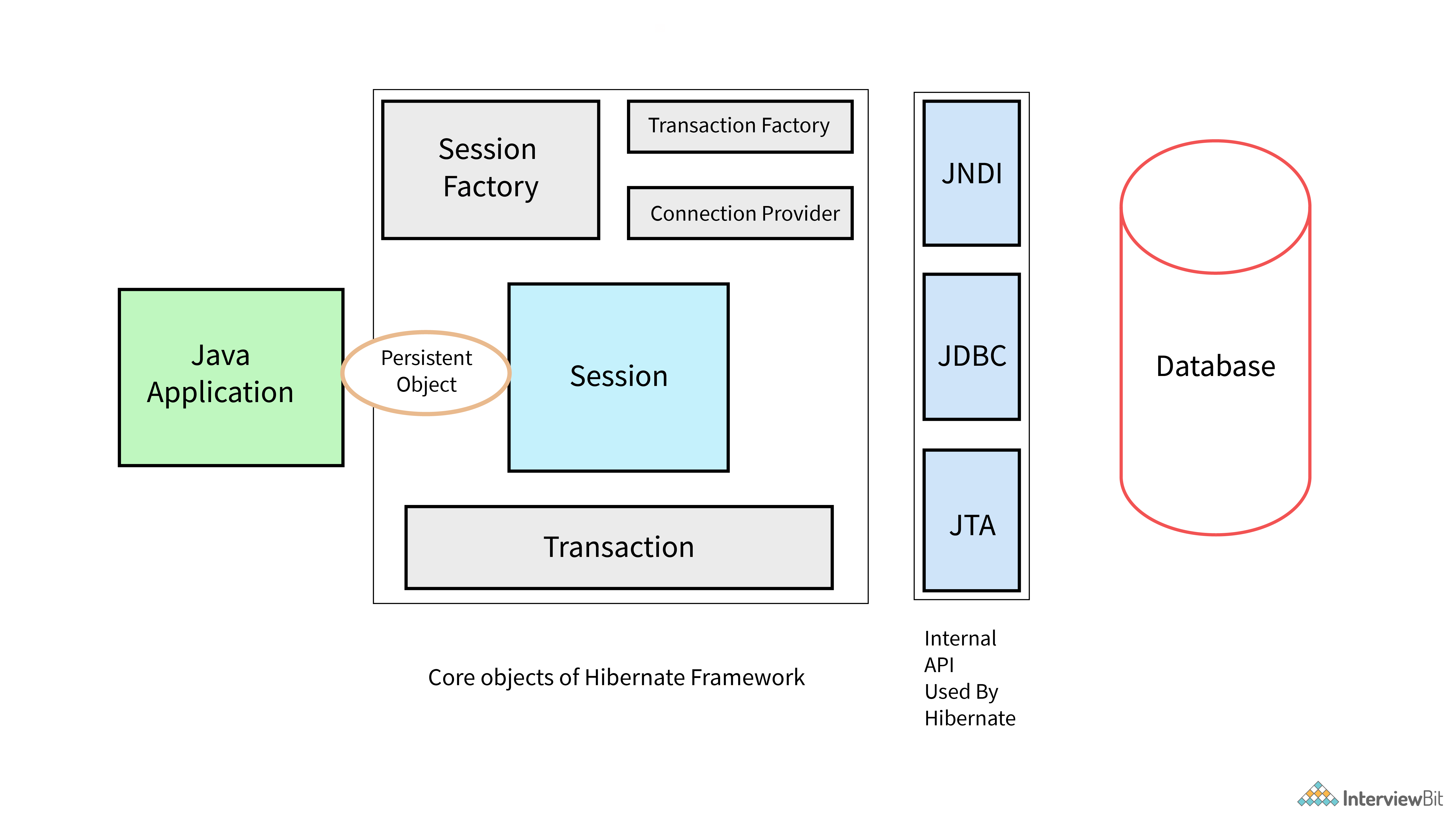




### Explain Hibernate architecture

The Hibernate architecture consists of many objects such as a persistent object, session factory, session, query, transaction, etc. Applications developed using Hibernate is mainly categorized into 4 parts:

* **Java Application**
* **Hibernate framework** - Configuration and Mapping Files
* **Internal API -**
  + JDBC (Java Database Connectivity)
  + JTA (Java Transaction API)
  + JNDI (Java Naming Directory Interface).
* **Database** - MySQL, PostGreSQL, Oracle, etc



#### **SessionFactory**

The SessionFactory is a factory of sessions. Session factory is created by the configuration object. which provides session based on the configuration properties to establish the relationship with database. **Second level cache** is available at the session factory if we enable it. we can have only one session factory per database. (If we have 2 or 3 databases, we can have 2 or 3 session factories) . **session factory is synchronized and** thread safe.The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

#### **Session**

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a **first-level cache** (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

#### **Transaction**

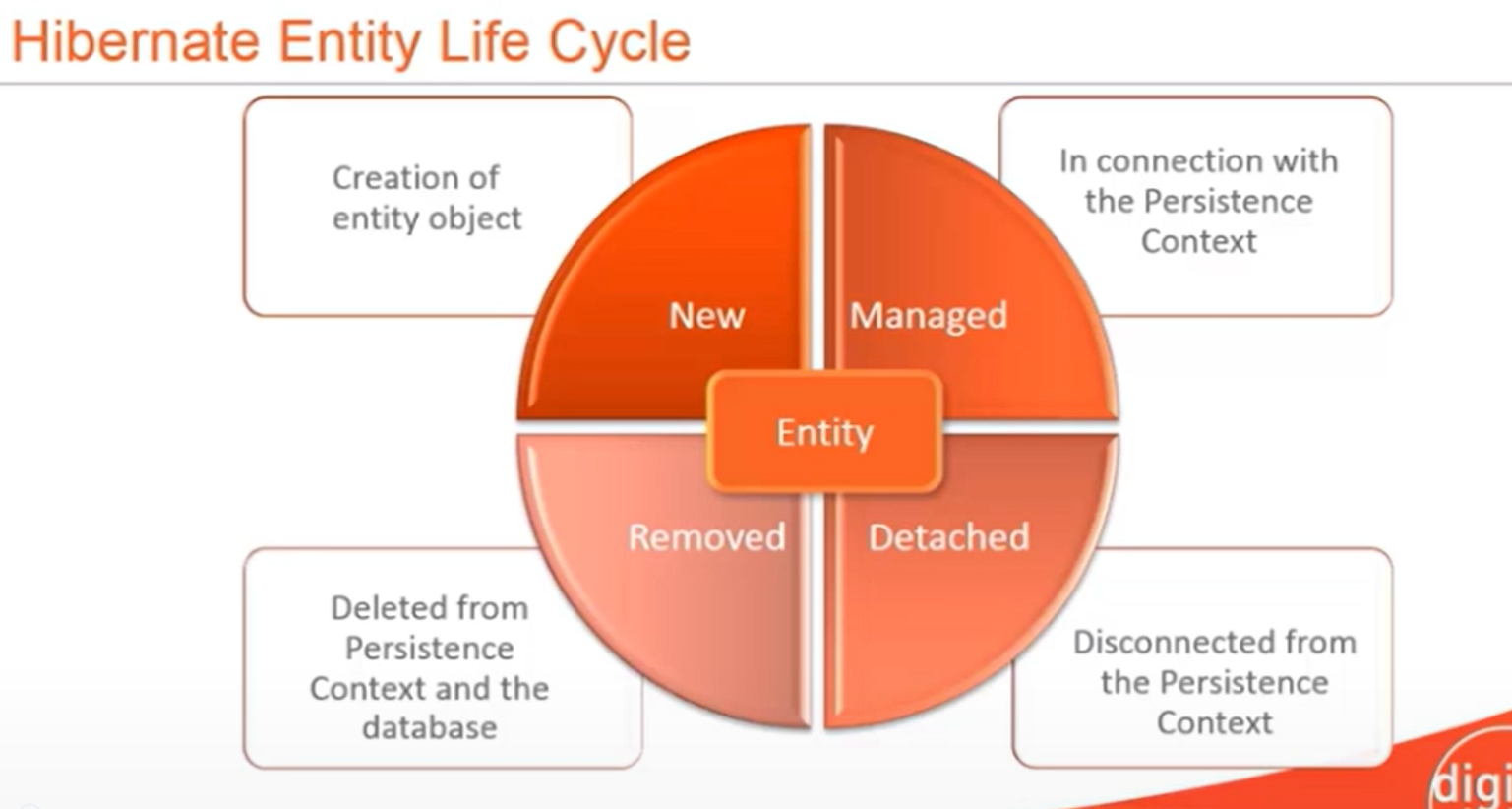
The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

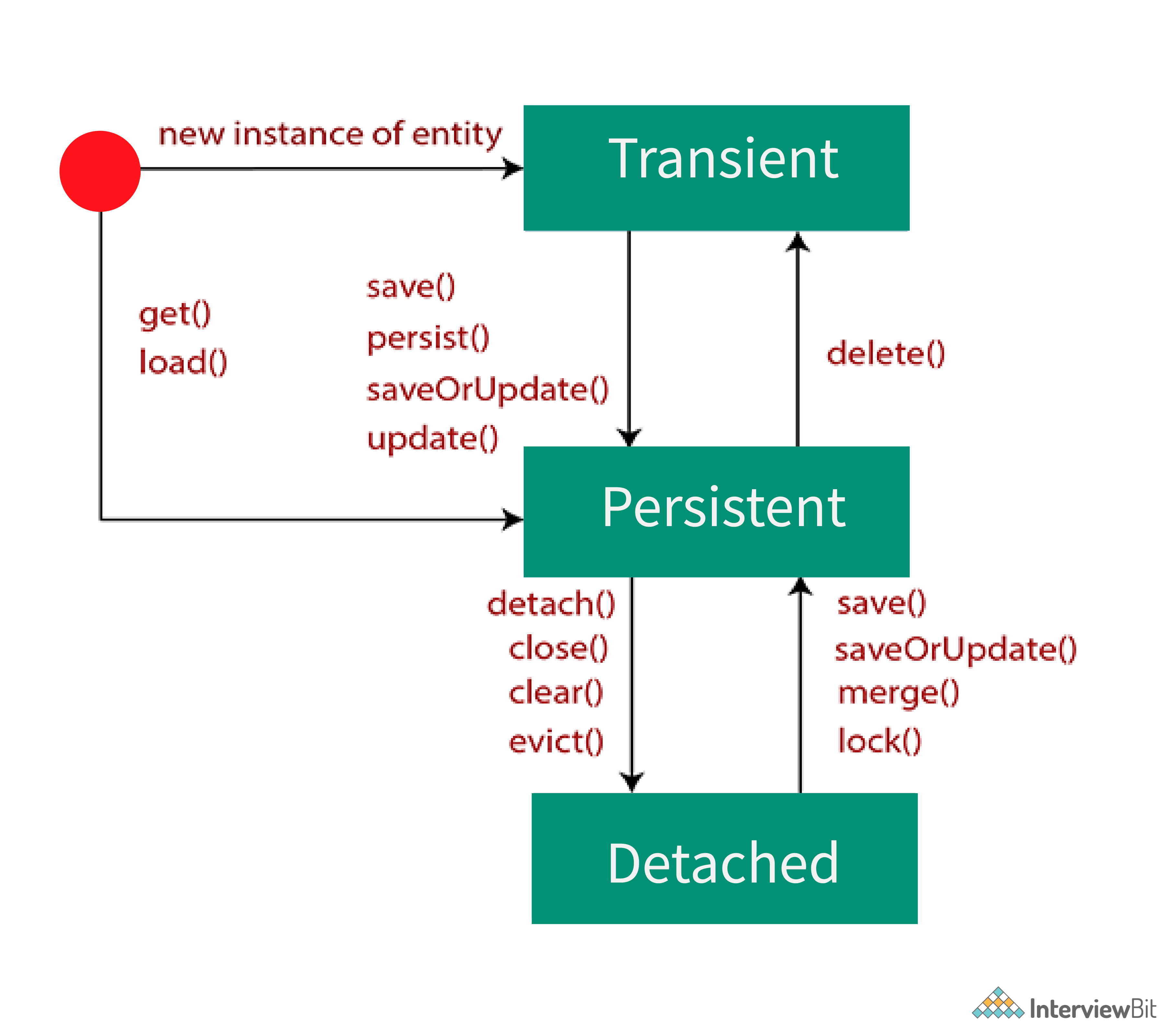
#### **ConnectionProvider**

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

#### **TransactionFactory**

It is a factory of Transaction. It is optional.





**Mainly there are Three States in Hibernate:**

**1.Transient state : (new state when a new object is created )**

**2.Persistent state (when an object is attached with session)**

**3 Detached state ( obj is present In data base but removed from session )**

**4.Removed state (obj deleted from data base and session ) we can call it as object gets detached and it will become new obj**

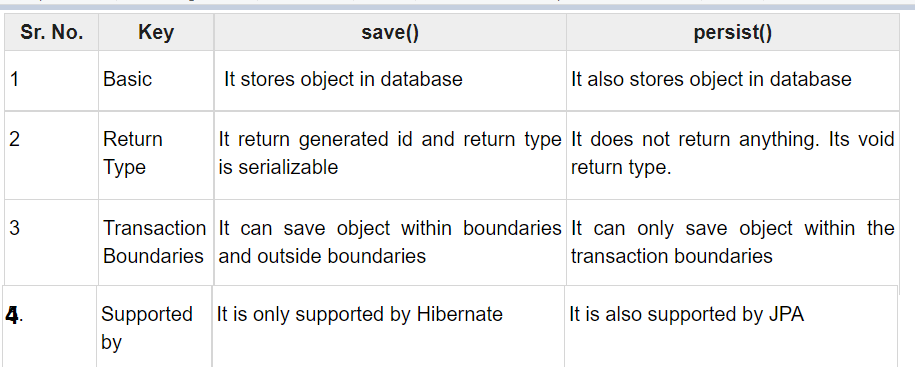
**Hibernate Methods:**

**Save:** Save method is used to stores an entity into the database. save() generates a new identifier and INSERT record into a database. If the identifier(primary key) is already exists in the table Then insertion fails (it will throw SQLIntegrityConstraintViolationException) . The return type for save method is Serializable Object( primary key id) which is the newly generated identifier id value.

If you want to make a transient entity into persistent we use save method. Save will always results in insert query. save method will work even outside of transaction boundaries (it means after transcation.commit also it works). Save method is related to hibernate but not JPA.

**Persist:** Persist is Similar to save method, persist is also used to stores an entity into the database persist also INSERT records into the database, but **return type of persist is void** .

persist will not work outside of transaction boundaries, if you try persist out of transcation boundaries you will get Exception as org.hibernate.PersistentObjectException: detached entity passed to persist: com.teja.hibcore.Doctor. Persist() method is JPA related. Hibernate provided implementation for that.



**Update:**  Update method in the hibernate is used for updating the object using identifier. If the identifier is missing or doesn’t exist, it will throw exception. ( Update is mainly used to make changes to detached entity). Update will result us update query.

update is used to update a detached entity's data to database by updating the entity in persistence context by using identifier

For update method refer below link <https://www.onlinetutorialspoint.com/hibernate/hibernate-update-query-example.html>

**SaveorUpdate:**

If you are not sure about the entity state in order to update a record first we need to check entity state then we need to invoke save or update methods according to the state if we use saveOrUpdate we don’t need to check the entity state explicity, hibernate will check the entity state if entity is in transient state it will invoke save if entity in in detached state then it will invoke update method.

SaveOrUpdate() calls either save() or update() on the basis of identifier exists or not. e.g if identifier does not exist, save() will be called or else update() will be called.

**Merge :**

merge is also used to transfer an object from detached stated to persistent state.

If we call merge() method, then it verifies whether the same object has existed in the session cache or not. If the object has lived in the session cache, then the current changes will be merged with session object otherwise, it will load the values to cache. Then it will update the data

**Update VS Merge:**

https://www.onlinetutorialspoint.com/hibernate/difference-between-update-vs-merge-example.html

Both update() and merge() methods are used to change the state of an object. That means we can call either update() or merge() to transfer an object from detached state to persistent state.

A detached state object can be made to persist by reattaching to a session. If the previous session has already been closed, it is also possible to create a new session and can attach to that session.

To reattach we can use update() or merge() methods. Both are doing the same functionality, but there are few differences internally.

When we call update() method on the session, if that session doesn’t contain the same object (provided in the update())  in the cache then update() method successfully executed and the object been converted detached state to persistent state.

When we call update() method on any object, it intern checks, if that object is already existed in session cache or not — if currently updating object is already there in session cache then it throws an exception called NonUniqueObjectException. Otherwise, it will update the object.

Like update() method merge is also used to transfer an object from detached stated to persistent state.

If we call merge() method, then it verifies whether the same object has existed in the session cache or not. If the object has lived in the cache, then the current changes are copied into the cache; otherwise, it will load the values to cache.

Extra link: <https://stackoverflow.com/questions/49604134/update-vs-merge-method-in-hibernate/49604264>

**Get:**

when we use get method to fetch data it will hit database and get data directly. when we use get method to fetch a data which is not present in database you will get null value in return, if you try to access any property with that null reference you get null pointer exception

get is eager loading, it means when we call get it will get all the asscoated child objects also even if we don’t require

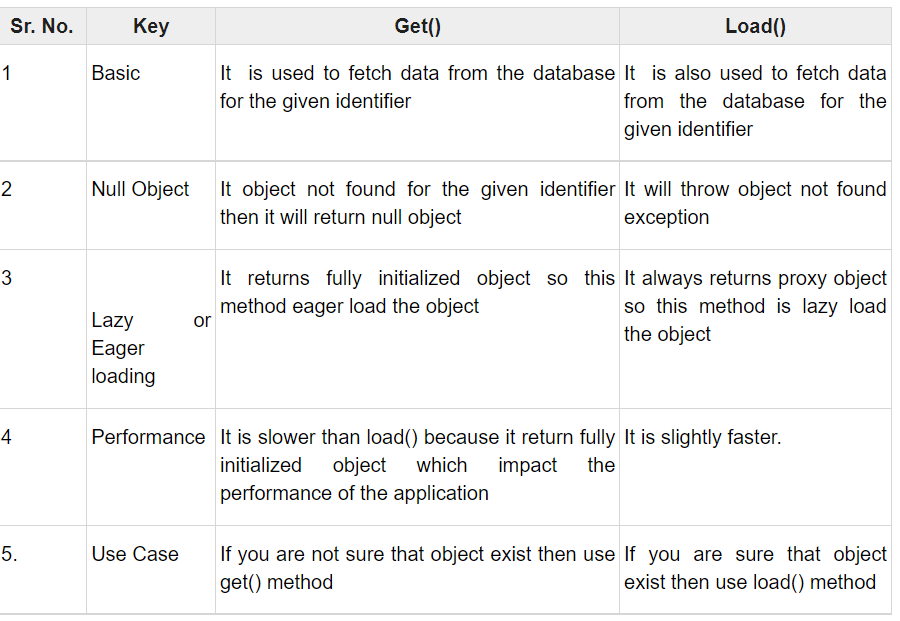
**Load**

when we use load it will check the data base if data is available then it will create a proxy object and it will return that object

load is lazy loading ,Proxy object will be created only with ID but without initializing other properties, which is lazily initialized, if you need if we call like d.getName() then only that properties get initialized

when we use load method to fetch a data which is not present in database you will get Exception in thread "main" org.hibernate.ObjectNotFoundException: No row with the given identifier exists: [com.teja.hibcore.Doctor#3]

**GET VS LOAD**



**Refresh**

* It is possible to re-load an object and all its collections at any time, using the refresh() method. This is useful when database triggers are used to initialize some of the properties of the object
* **session.refresh()  is used to refresh the values of an entity from database**
* **Flush :**
* Whenever you pass an object to save(), update() or saveOrUpdate(), and whenever you retrieve an object using load(), get(), list(), iterate() or scroll(), that object is added to the internal cache of the Session.
* When flush() is subsequently called, **the state of that object will be synchronized with the database.** If you do not want this synchronization to occur, or if you are processing a huge number of objects and need to manage memory efficiently, the evict() method can be used to remove the object and its collections from the first-level cache.
* **session.flush() Flushing the session forces Hibernate to synchronize the in-memory state of the Session with the database.**
* session.evict() Detach the object from session cache. After detaching the object from the session, any change to object will not be persisted.
* **Evict is used to remove a object from session (first level cache)**
* **Clear is used to remove all the objects from the session**

**Primary key id generation strategies**

**@Id** annotation **marks the property as unique identifier** for this entity.

Entity must have a primary key which uniquely identifies it. [The @Id annotation](https://www.baeldung.com/hibernate-identifiers) defines the primary key. We can generate the identifiers in different ways which are specified by the @GeneratedValue annotation.

We can choose from four id generation strategies with the strategy element. The value can be AUTO, TABLE, SEQUENCE, or IDENTITY.

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private String customerOrderId;

1. *AUTO*: Hibernate selects the generation strategy based on the used dialect,
2. *IDENTITY*: Hibernate relies on an auto-incremented database column to generate the primary key,
3. *SEQUENCE*: Hibernate requests the primary key value from a database sequence,
4. *TABLE*: Hibernate uses a database table to simulate a sequence.

**@Entity** annotation marks this class as an entity.

Entities are nothing but POJOs representing data that can be persisted to the database.Every instance of an entity represents a row in the table.

**@Table** annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name by default.

@Table(name="STUDENT")

**@Column** annotation specifies the details of the database column for this property or field (**mapping java property as database column**). If @Column annotation is not specified, property name will be used as the column name by default.

**@Temporal** is a JPA annotation and generally specified only for persistent fields or properties of type java.util.Date and java.util.Calendar. Generally you use @Temporal annotation on the date or calendar field in entity class when you want to map date or time type of property to database specific sql type of data & time. If we use Hibernate 5 above versions @Temporal annotation is not required.

**@TRANSIENT** ANNOTATION IN JPA OR HIBERNATE IS USED TO INDICATE THAT A FIELD IS NOT TO BE PERSISTED OR IGNORE FIELDS TO SAVE IN THE DATABASE.  @TRANSIENT EXIST IN JAVAX.PERSISTENCE PACKAGE

**Difference between session and session factory**

**https://www.netsurfingzone.com/sessionfactory-and-session-in-hibernate/**

**Session**:

1.Session is an object which is used to maintain a relationship or connection between the java application objects and the database.

2.First level cache is available at the session by default.

3.we can have any number of sessions per database (whenever we try to interact with database session gets created)

4.session is not synchronized. So not thread safe.

**Session factory**

1.Session factory is created by the configuration object. which provides session based on the configuration properties to establish the relationship with database.

2.Second level cache is available at the session factory if we enable it.

3.we can have only one session factory per database. (If we have 2 or 3 databases, we can have 2 or 3 session factories)

4.session factory is synchronized and thread safe.

**Difference between first level cache and second level cache?**

**First level cache:**

1.We can maintain first level cache at the session.

2.by default session contains first level cache, and we can't disable it.

3.if we destroy session then first level cache also gets destroyed.

**Second level cache:**

1.We can maintain second level cache at the session factory

2.by default, second level cache is disabled at the session factory, we can enable it by doing the configurations.

3.second leve

l cache is available throughout the application until we restart it.

**Difference between lazy loading and early loading?**

**Early loading:**

1.It is a process where one object or entity will load other relevant objects or entities associated with that data.

2. **ManyToOne and OneToOne associations uses Eager loading strategy by default.**

3. It can be enabled by using the annotation parameter :  
 fetch = FetchType.LAGY

**Lazy Loading:**

1.It is process where one object or entity will load other object or entity only when we specifically request for the data.

2. ManyToMany and OneToMany associations used lazy loading strategy by default.

3. It can be enabled by using the annotation parameter :  
 fetch = FetchType.EAGER

**Explain about hibernate Associations?**

hibernate Associations means Establishing relationship between the entities by using some of annotations like one-one, one-many, many-one, many-many.

For example like one vehicle will have only one unique VIN(vehicle identification number).

One-many: One vehicle will have many defects, many-many: many-cars can were repaired in many dealer service centres , many service centres repairs many cars.

Just check code once for mapping annotations in hibernate

**Extra-points :** You can map each of them as a unidirectional - or bidirectional association , in case of unidirectional we can access data only one way and also change’s done in one side will not impact other means if we delete one side it will not delete other side, but in bi-directional we can navigate in both ways and impact of changes will be on both sides . it defines in which direction you can use the relationship in your domain model

### What is HQL?

**Hibernate Query Language (HQL)** is used as an extension of **SQL**. It is very simple, efficient, and very flexible for performing complex operations on relational databases without writing complicated queries. HQL is the object-oriented representation of query language, i.e instead of using table name, we make use of the class name which makes this language independent of any database.

This makes use of the Query interface provided by Hibernate. The Query object is obtained by calling the createQuery() method of the hibernate Session interface.

Query query=session.createQuery("from Employee");

Query query=session.createQuery("from Employee e where e.id=3");

List<InterviewBitEmployee> list=query.list();

System.out.println(list.get(0));

<https://www.javatpoint.com/hql> refer for more info

<https://www.tutorialspoint.com/hibernate/hibernate_query_language.htm>

### Hibernate Native SQL Example:

### If we want to write a normal sql query instead of hibernate HQL query we can use SQLQuery object , with help of session.createSQLQuery method we can write our own sql querys

For Hibernate Native SQL Query, we use Session.createSQLQuery(String query) to

//SQLQuery query= session.createSQLQuery("select \* from doctor where id=3");

SQLQuery query= session.createSQLQuery("select \* from doctor where id=:d\_id");

query.setParameter("d\_id", 3);

query.addEntity(Doctor.**class**);

List<Doctor> l= query.list();

### NamedQuerys and NamedNativeQuerys

### If we are using a query repeatedly in our project, we can group such query in one place to reuse them when ever we required. So that we can make code readable and maintainable.

If there are a lot of queries, then they will cause a code mess because all the queries will be scattered throughout the project. That’s why Hibernate provides **Named Query** that we can define at a central location and use them anywhere in the code. We can created named queries for both HQL and Native SQL.

A named query is a statically defined query with a predefined unchangeable query string. They are validated when the session factory is created, thus making the application fail fast in case of an error.

Hibernate Named Query can be defined in Hibernate mapping files or through the use of JPA annotations @NamedQuery and @NamedNativeQuery. @Entity

@Table(name = "ADDRESS")

**@NamedQueries(**

**{ @NamedQuery(name = "HQL\_GET\_ALL\_ADDRESS",**

**query = "from Address")**

**}**

**)**

**@NamedNativeQueries(**

**{ @NamedNativeQuery(name = "SQL\_GET\_ALL\_ADDRESS",**

**query = "select emp\_id, address\_line1, city, zipcode from Address")**

**}**

**)**

public class Address {

TypedQuery query =session.getNamedQuery("**HQL\_GET\_ALL\_ADDRESS**");

query.setParameter("fullName","Hibernate");

List<InterviewBitEmployee> ibEmployees = query.getResultList();

## **Criteria**

If we want to perform any data filter operations based on some conditions, we can use criteria.

Criteria API, which allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

Criteria is used to fetch the records based on the specific criteria. The Criteria interface provides methods to apply criteria such as retrieving all the records of table whose salary is greater than 50000 etc.

The Hibernate **Session** interface provides **createCriteria()** method, which can be used to create a **Criteria** object that returns instances of the persistence object's class when your application executes a criteria query.

**Criteria cr = session.createCriteria(Employee.class);**

**cr.add(Restrictions.gt("salary", 2000));**

**List results = cr.list();**

We use Restrictions to filter data based on conidiations like greater than less than like equal ,like etc...

Eq- equal , gt-greater than , lt-less than ,like

**Pagination using hibernate :**

Criteria cr = session.createCriteria(Employee.class);

cr.setFirstResult(1);

cr.setMaxResults(10);

List results = cr.list();

## **Sorting the Results**

The Criteria API provides the **org.hibernate.criterion.Order** class to sort your result set in either ascending or descending order, according to one of your object's properties. This example demonstrates how you would use the Order class to sort the result set −

Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000

cr.add(Restrictions.gt("salary", 2000));

// To sort records in descening order

cr.addOrder(Order.desc("salary"));

// To sort records in ascending order

cr.addOrder(Order.asc("salary"));

List results = cr.list();

## **Projections & Aggregations**

The Criteria API provides the **org.hibernate.criterion.Projections** class, which can be used to aggregate data based on particular column, we can get average, maximum, or minimum of the property values. The Projections class is similar to the Restrictions class, in that it provides several static factory methods for obtaining **Projection** instances.

Following are the few examples covering different scenarios and can be used as per requirement −

Criteria cr = session.createCriteria(Employee.class);

// To get total row count.

cr.setProjection(Projections.rowCount());

// To get average of a property.

cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.

cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.

cr.setProjection(Projections.max("salary"));

// To get minimum of a property.

cr.setProjection(Projections.min("salary"));

// To get sum of a property.

cr.setProjection(Projections.sum("salary"));

**Explain about hibernate relationship annotations**

By using hibernate we can map our java classes to data base tables and

Establish relationship between two or more database tables by using annotations like @onetoone,@onetomany,@manytoone,@manytomany

Refer same entity code with mappings

can establish either unidirectional or bidirectional relation

The main difference is that bidirectional relationship provides navigational access in both directions,

so that you can access the other side without explicit queries. Also it allows you to apply cascading options to both directions it means if we delete or update one side other side also gets effected.

Defines the set of cascadable operations that are propagated to the associated entity. The value cascade=ALL is equivalent to cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}.

More information about cascading: https://www.baeldung.com/jpa-cascade-types

<https://www.baeldung.com/jpa-joincolumn-vs-mappedby>

Explain about hibernate Associations?

hibernate Associations means Establishing relationship between the entities by using some of annotations like one-one, one-many, many-one, many-many.

For example like one vehicle will have only one unique VIN(vehicle identification number).

One-many: One vehicle will have many defects, many-many: many-cars can were repaired in many dealer service centres , many service centres repairs many cars.

Just check code once for mapping annotations in hibernate

**Extra-points :** You can map each of them as a unidirectional - or bidirectional association , in case of unidirectional we can access data only one way and also change’s done in one side will not impact other means if we delete one side it will not delete other side, but in bi-directional we can navigate in both ways and impact of changes will be on both sides . it defines in which direction you can use the relationship in your domain model

@JoinColumn is used to create a foregin\_key column, Mostly we will be using this annotation in Many side in one to one we can use at any side. Many side will be taking ownership and other side in other entity class we use mapped by.

@OneToOne(cascade = CascadeType.***ALL***)

@JoinColumn(name="Fk\_passport\_ID")

**private** Passport passport;

The below will in Passport class here mappedBy is the property which is present in person class, so here person is mapped with passport

@OneToOne(mappedBy = "passport")

**private** Person person;

The [@JoinColumn](https://www.baeldung.com/jpa-join-column) annotation helps us specify the column we'll use for joining an entity association or element collection. On the other hand, the mappedBy attribute is used to define the referencing side (non-owning side) of the relationship.

**we need to place**[**the @JoinColumn annotation**](https://www.baeldung.com/jpa-join-column) to configure the name of the column in the users table that maps to the primary key in the address table.  **Simply put, whoever owns the foreign key column gets the @JoinColumn annotation.**

**Most of the time @joincolumn will be always many side who has foreign key**

The @JoinColumn annotation defines the actual physical mapping on the owning side. On the other hand, the referencing side is defined using the mappedBy attribute of the @OneToMany annotation.

=========one-many and many to one=================

@Entity

@Table

**public** **class** DummyCustomer {

@OneToMany(mappedBy ="dummyCustomer",cascade = CascadeType.***ALL***,fetch=FetchType.***EAGER***)

**private** List<PhoneNumbers> phonenumbers;

@Entity

@Table

**public** **class** PhoneNumbers {

@ManyToOne

@JoinColumn(name="fk\_cust\_id")

**private** DummyCustomer dummyCustomer;

========MANY-MANY====================================

@Entity

@Table(name = "MOVIE\_TABLE")

**public** **class** Movie {

@ManyToMany(cascade={CascadeType.***PERSIST***})

@JoinTable(

name="movie\_actor",

joinColumns={@JoinColumn(name="movie\_id")},

inverseJoinColumns={@JoinColumn(name="actor\_id")}

)

**private** Set<Actor> actors = **new** HashSet<Actor>();

@Entity

@Table(name = "ACTOR\_TABLE")

**public** **class** Actor {

@ManyToMany(mappedBy="actors")

**private** Set<Movie> movies = **new** HashSet<Movie>();

**How to Configure Hibernate configuration to connect from a core java app to data base?**

We can configure in two ways by using java configuration or by using xml (hibernate.cfg.xml). Best way is by using xml, because if we use java-based configuration in future if we change database, then we need to change code which is not a good practice.

* Below is sample hibernate.cfg.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

**<session-factory>**

<property name=*"hibernate.connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<property name=*"hibernate.connection.url"*>jdbc:mysql://localhost:3306/my\_java</property>

<property name=*"hibernate.connection.username"*>root</property>

<property name=*"hibernate.connection.password"*>root</property>

<property name=*"show\_sql"*>true</property>

<property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>

<property name=*"hibernate.hbm2ddl.auto"*>update</property>

<mapping class=*"hibpjo.person"*/>

<mapping class=*"hibpjo.passport"*/>

<mapping class=*"hibpjo.mobile"*/>

**</session-factory>**

* </hibernate-configuration>

**hibernate.dialect:**

<property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>

* This property makes Hibernate to generate the appropriate SQL Queries for the user specific database. Dialect value will be different for each database.

**hbm2ddl.auto:**

<property name=*"hibernate.hbm2ddl.auto"*>update</property>

hbm2ddl.auto–>create -Always create new schema

hbm2ddl.auto–>update -Update existing schema

**list of possible options are,**

* ***validate*:** validate the schema, makes no changes to the database.
* ***update***: update the schema.
* ***create*:** creates the schema, destroying previous data.
* ***create-drop*:** drop the schema when the SessionFactory is closed explicitly, typically when the application is stopped.
* ***none*:** does nothing with the schema, makes no changes to the database

**Show\_sql:**

<property name=*"show\_sql"*>true</property>

If we make show\_sql property as true it will show all the hibernate gernated queries in console or log file

Mapping Information:

<mapping class=*"hibpjo.person"*/>

We can do mapping in two ways

1. By using hbm.xml(hibernate mapping xml)

2.By Using annotations of javax.persistence package

**HibernateTemplate**

The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

**How will you integrate your spring application with data base in spring mvc application ?**

In Spring xml configuration file we will be creating BasicDataSource bean with the properties like database url, username, password, driver class name and then we will be injecting that bean into session factory bean and we will set some more properties to session factory bean like dialect ,show\_sql, hbm2ddl none , and entity class for mapping information. Then we will inject session factory bean with hibernate template bean , then in our dao class we will autowire hibernate template bean, with help of hibernate template methods we can perform database operations like save, update, delete and fetch data with find method form the database

<bean id=*"bds"* class=*"org.apache.commons.dbcp2.BasicDataSource"*>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/ushanew"*/>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

</bean>

<bean id=*"sf"* class=*"org.springframework.orm.hibernate4.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"bds"*/>

<property name=*"hibernateProperties"*>

<props>

<prop key=*"hibernate.show\_sql"*>true</prop>

<prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</prop>

<prop key=*"hibernate.hbm2ddl.auto"*>update</prop>

</props>

</property>

<property name=*"annotatedClasses"*>

<list>

<value>com.govtcollege.entity.GovtCollegeEntity</value>

<value>com.govtcollege.entity.UsersPojoEntity</value>

</list>

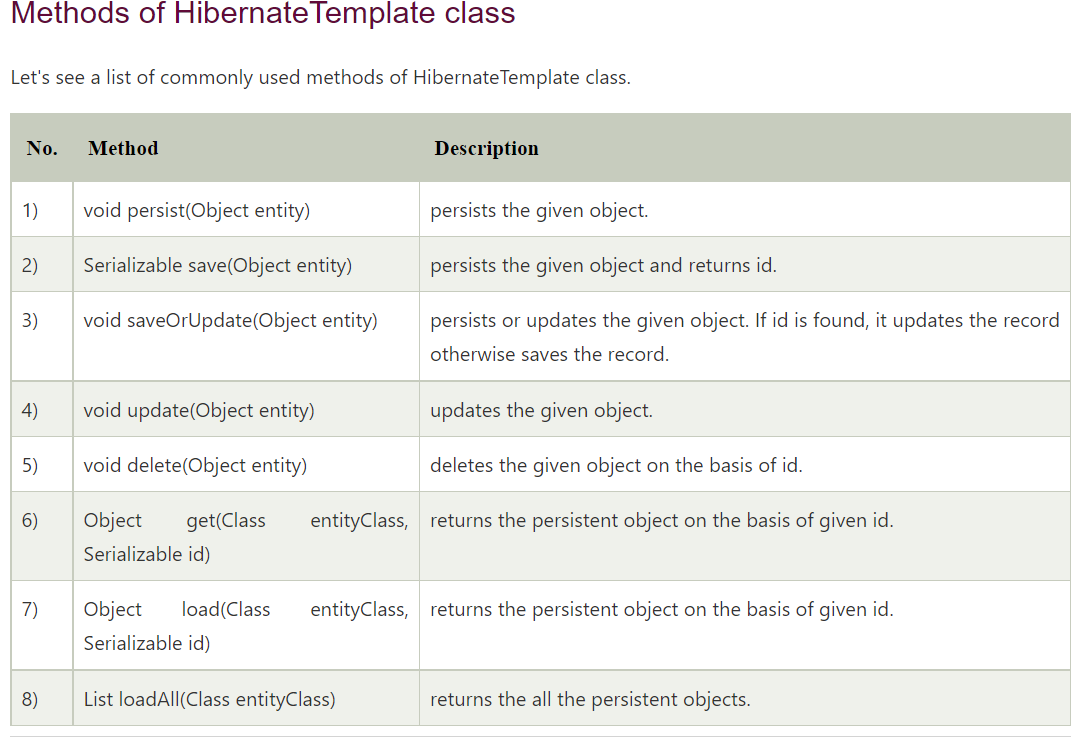
</property>

</bean>

<bean id=*"ht"* class=*"org.springframework.orm.hibernate4.HibernateTemplate"*>

<property name=*"sessionFactory"* ref=*"sf"*/>

</bean>



9 [find](https://docs.spring.io/spring-framework/docs/2.5.x/javadoc-api/org/springframework/orm/hibernate3/HibernateTemplate.html#find(java.lang.String))([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true) queryString) Executes an HQL query. Returns List

## **Intro to Spring Data JPA Is?**

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically

**Spring Data JPA is not a JPA provider**. It is a library / framework **that adds an extra layer of abstraction on the top of our JPA provider.** If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:

* [Spring Data JPA](http://projects.spring.io/spring-data-jpa/) provides support for creating JPA repositories by extending the Spring Data repository interfaces.
* [Spring Data Commons](https://github.com/spring-projects/spring-data-commons) provides the infrastructure that is shared by the datastore specific [Spring Data projects](http://projects.spring.io/spring-data/).
* The JPA Provider implements the Java Persistence API.

## **What is Spring Data JPA?**

Spring Data JPA is one of Spring Data module which provides predefined repository methods to perform CRUD operation. Using Spring Data JPA we define the repository interface and query methods(query creation from method names) to access the data from the database. It makes easier to build Spring applications that use data access technologies.

The OurRepository will be implemented from Spring Data JPA during runtime based on the SimpleJpaRepository (org.springframework.data.jpa.repository.support. SimpleJpaRepository)

<https://stackoverflow.com/questions/38509882/how-are-spring-data-repositories-actually-implemented>

[https://www.netsurfingzone.com/jpa/spring-data-jpa-interview-questions-and-answers/](https://www.netsurfingzone.com/jpa/spring-data-jpa-interview-questions-and-answers/%20)

## **What is Spring Data JPA?**

Spring Data JPA is one of Spring Data module which provides predefined repository methods to perform CRUD operation. Using Spring Data JPA we define the repository interface and query methods(query creation from method names) to access the data from the database. It makes easier to build Spring applications that use data access technologies.

## **How to create a custom repository in Spring Data JPA?**

We can create custom repository extending any of these interfaces according to need.

Repository  
CrudRepository  
PagingAndSortingRepository  
JpaRepository  
QueryByExampleExecutor

StudentRepository.java

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

public List<Student> findByNameAndRollNumber(String name, String rollNumber);

public List<Student> findByNameOrRollNumber(String name, String rollNumber);

public List<Student> findByNameAndRollNumberOrUniversity(String name, String rollNumber, String university);

}

See more details [here](https://netsurfingzone.com/jpa/how-to-create-a-custom-repository-in-spring-data-jpa/).

## **How you will write custom method in the repository in Spring Data JPA? What are rules to define Query methods(query creation from method names)?**

Consider we have an entity called Student.java and we have some records in the database as below.

Student.java

package com.netsurfingzone.entity;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

To write query methods first we need to define repository interface.

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

}

Rules to define Query methods.

Rule 1 – The name of the query method must start with findBy or getBy  or queryBy or countBy or readBy prefix. The findBy is mostly used by the developer.

For example findByName(String name), getByName(String name), queryByName(String name), countByName(String name), readByName(String name),

Note – All the above query methods will return all students whose name is “what ever name we provide as parameter”.

Rule 2 – The first character of field name should capital letter. Although if we write the first character of the field in small then it will work but we should use camelcase for the method name.

Both are valid query method defined below but we should follow the first way.

public List<Student> findByName(String name);

public List<Student> findByname(String name);

Rule 3 – While using findBy or getBy or queryBy or countBy or readBy the character B must be in capital letter, else we will get an exception while deployment.

Caused by: org.springframework.data.mapping.PropertyReferenceException: No property querybyName found for type Student!

Invalid query method.

public List<Student> findbyName(String name);

Rule 4 – We can write the query method using multiple fields using predefined keywords(eg. And, Or etc) but these keywords are case sensitive. We must use “And” instead of “and”.

**Write query method using @Query.**

Writing JPQL using Spring Data Jpa @Query.

@Query("select s from Student s where s.name = ?1")  
List<Student> getStudents(String name);

See a complete example of the JPQL using Spring Boot and Oracle [here](https://netsurfingzone.com/jpa/spring-data-jpa-jpql-and-native-query-example/).

**Writing the Named Parameter @Query.**

@Query("select s from Student s where s.name = :name")  
List<Student> findByName(@Param("name") String name);

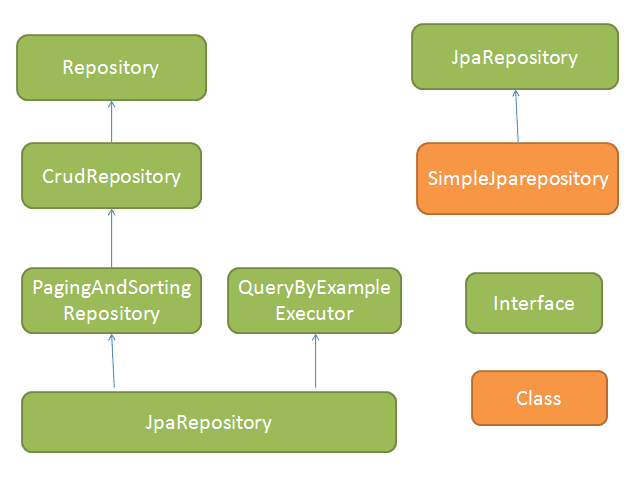
See a complete example of the Named Parameter using Spring Boot and Oracle [here](https://netsurfingzone.com/jpa/spring-data-jpa-named-parameters/).

## **What are the important predefined repository interfaces and classes in Spring Data JPA?**

* Repository – Top-level interface defined in Spring Data Hierarchy. This is a marker interface i.e doesn’t contain any method. See more details about the Repository interface [here](https://netsurfingzone.com/jpa/difference-between-repository-and-crudrepository/).
* CrudRepository – The CrudRepository interface extends Repository interface, provides methods to perform CRUD operation. See more details about the CrudRepository interface [here](https://netsurfingzone.com/jpa/crudrepository-methods-example/).
* PagingAndSortingRepository – The PagingAndSortingRepository interface extends CrudRepository interface and provides additional methods to retrieve entities using the pagination and sorting.
* QueryByExampleExecutor – The QueryByExampleExecutor interface used to execute Query by Example.
* JpaRepository – The JpaRepository interface extends PagingAndSortingRepository and QueryByExampleExecutor interface, provides some additional batch methods. See more details [here](https://netsurfingzone.com/jpa/difference-between-crudrepository-and-jparepository-in-spring-data-jpa/).
* SimpleJpaRepository – The SimpleJpaRepository is the implementation class of the CrudRepository interface.
* QueryDslJpaRepository – This is a class.

## **What is the hierarchy of repository interfaces/classes in Spring Data JPA?**

The Repository (marker interface) is the top-level interface defined in Spring Data Hierarchy.

[](https://netsurfingzone.com/jpa/spring-data-jpa-interview-questions-and-answers/attachment/jpaapi1/)

## **What are the important methods of Crudrepository to perform CRUD operations in Spring Data JPA?**

The Crudrepository contains below methods.

* save(S entity) – Used to save a single entity at a time. See an example of the save() method [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-save-method/).
* saveAll(Iterable<S> – we can save multiple entities at a time. See a complete example of the CrudRepository saveAll() method from scratch [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-saveall-and-findall/).
* findById(ID id) – Use to get entity basis of id. See an example of the findById() method [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-finbyid/).
* existsById(ID id) – Used to check whether an entity is already exited in DB for given Id. See an example of the existsById() method [here](https://netsurfingzone.com/jpa/spring-data-crudrepository-existsbyid/).
* findAll() – Find all entity of particular type. See an example of the findAll() [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-saveall-and-findall/).
* findAllById(Iterable<ID> ids)  – Return all entity of given ids. See an example [here](https://netsurfingzone.com/jpa/crudrepository-findallbyid-example-using-spring-boot/).
* count() – Returns the number of entities. See an example of the count() method [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-count-example/).
* deleteById(ID id) – Delete the entity on basis of id. See a complete example deleteById() method [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-deletebyid-example/).
* delete(T entity) – delete the entity which one we are passing. See a complete example [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-delete-and-deleteall/).
* delete(Iterable<? extends T> entities) – delete multiple entities which we are passing.  See a complete example [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-delete-and-deleteall/).
* deleteAll() – delete all entities. See a complete example [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-delete-and-deleteall/).

## **What are the features/benefits of Spring Data JPA?**

Spring Data JPA provides features to Query creation from method names. For example, consider we have a method defined in Studentrepository  public List<Student> findByName(String name);  and we have an entity Student.java as below.

StudentRepository.java

@Repository

public interface StudentRepository extends CrudRepository<Student, Serializable> {

public List<Student> findByName(String name);

}

Student.java

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

The following query will get created.

Hibernate:  
select  
student0\_.id as id1\_0\_,  
student0\_.name as name2\_0\_,  
student0\_.roll\_number as roll\_number3\_0\_,  
student0\_.university as university4\_0\_  
from  
student student0\_  
where  
student0\_.name=?

Easy to define query methods for data access. For example, public List<Student> findByName(String name); will return all students for the given name. Spring Data JPA provides predefined keywords using that we can write different query methods.

Let’s see some important keywords and corresponding Query Methods.

| **Keyword** | **Query Method** | **Generated Query Sample** |
| --- | --- | --- |
| And | List findByNameAndRollNumber(String name, String rollNumber) | here student0\_.name=? and student0\_.roll\_number=? |
| Or | List findByNameOrRollNumber(String name, String rollNumber) | where student0\_.name=? or student0\_.roll\_number=? |
| Beetween | List findByRollNumberBetween(String start, String end) | where student0\_.roll\_number between ? and ? |

See more example of each keyword separately [here](https://netsurfingzone.com/jpa/spring-data-jpa-query-methods/).

Since CrudRepository provides predefined methods and we can define query methods(for accessing data) which reduce code size and Using Spring Data Jpa we can remove a lot of code by creating an abstract repository class that provides CRUD operations for our entities.

We can write complex query using @Query annotation in Spring Data JPA. See an example [here](https://netsurfingzone.com/jpa/spring-data-jpa-query-annotation/).

We can also write JPQL and Native Query with entity using @NamedQuery and @NamedNativeQuery annotation. See an Example [here](https://netsurfingzone.com/jpa/spring-data-jpa-namedquery/).

Spring Data Jpa provides PagingAndSortingRepository interface for sorting and pagination support. See an Example [here](https://netsurfingzone.com/jpa/sorting-and-pagination-in-spring-data-jpa/).

Spring Data Jpa provides Named Parameters interface. See an Example [here](https://netsurfingzone.com/jpa/spring-data-jpa-named-parameters/).

We are looking at Spring Data JPA Interview Questions and Answers.

## **How to enable Spring Data JPA features.**

First, we need to define some configuration class let’s say JpaConfig.java and then we need to use @EnableJpaRepositories annotation with that class. Also, provide basePackages value where (we have our repositories). Let’s see below example.

package com.netsurfingzone.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@Configuration

@EnableJpaRepositories(basePackages = "com.netsurfingzone.repository")

public class JpaConfig {

}

we will see more Spring Data JPA Interview Questions and Answers in details.

## **Define a simple repository method which returns all records/entities for a given name?**

We can define a repository or query method as below. All three methods would do the same task.

import java.io.Serializable;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.netsurfingzone.entity.Student;

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

public List<Student> findByName(String name);

public List<Student> findByNameIs(String name);

public List<Student> findByNameEquals(String name);

}

Spring Data JPA Interview Questions and Answers

## **How to define Query Methods for the nested property.**

Consider we have two entities Student.java and Address.java. Student and Address entities are in one to one relationship and we want to fetch all students from the database who belongs to city pune.

Student.java

package com.netsurfingzone.entity;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "address\_id")

Address address;

}

Address.java

package com.netsurfingzone.entity;

@Entity

public class Address {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "house\_number")

private String houseNumber;

@Column(name = "city")

private String city;

}

Now we want all Students records which belong to city pune.

Query method to fetch all students who belong to city pune using Spring Data JPA.

List<Student> findByAddressCity(String city);

See an example [here](https://netsurfingzone.com/jpa/spring-data-jpa-nested-property-query-method/).

## **Write JPQL using @Query annotation in Spring Data JPA.**

Writing JPQL using Spring Data Jpa @Query.

@Query("select s from Student s where s.name = ?1")  
List<Student> getStudents(String name);

## **@NamedQuery vs @NamedNativeQuery in Spring Data JPA?**

@NamedQuery and @NamedNativeQuery annotations used with entity class.

@NamedQuery example.

package com.netsurfingzone.entity;

@Entity

@NamedQuery(name = "Student.findByName", query = "select s from Student s where s.name = ?1")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

@NamedNativeQuery example.

package com.netsurfingzone.entity;

@Entity

@NamedNativeQuery(name = "Student.findByName", query = "select \* from Student where name = ?1", resultClass = Student.class)

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

## **Difference between CrudRepository and JpaRepository in Spring Data JPA?**

| **CrudRepository** | **JpaRepository** |
| --- | --- |
| 1. CrudRepository extends Repository interface. | 1. JpaRepository extends PagingAndSortingRepository and QueryByExampleExecutor interface. |
| 2. CrudRepository provides methods to perform CRUD operations. | 2. JpaRepository provides additional methods like flush(), saveAndFlush(), deleteInBatch() etc. |
| 3. The saveAll(Iterable entities)  method of CrudRepository returns Iterable. | 3. The saveAll(Iterable entities)  method of JpaRepository returns List. |
| 4. If we have to perform mainly CRUD operation, define our repository using CrudRepository. | 4. If we have to perform CRUD as well as Batch operation define our repository extending JpaRepository. |

See more details [here](https://netsurfingzone.com/jpa/difference-between-crudrepository-and-jparepository-in-spring-data-jpa/).

## **Difference between Repository and CrudRepository in Spring Data JPA?**

Repository interface.

The Repository is a top-level interface in hierarchy.

The Repository is a marker interface. It doesn’t have any method.

Repository interface has been defined as below.

@Indexed

public interface Repository<T, ID> {

}

CrudRepository interface.

The CrudRepository extends Repository interface. It has below methods to perform CRUD operation.

save(S entity)  
saveAll(Iterable<S> entities)  
findById(ID id)  
existsById(ID id)  
findAll()  
findAllById(Iterable<ID> ids)  
count()  
deleteById(ID id)  
delete(T entity)  
delete(Iterable<? extends T> entities)  
deleteAll()

## **How CrudRepository save() methods internally works in Spring Data JPA?**

The CrudRepository’s save() method is used to perform save as well as update operation both. The implementation has been given in SimpleJpaRepository.java, where persist() and merge() is getting called. If we try to save entity first time then persist() method will get invoked and if we try to update the same entity merge() will get invoked.

public S save(S entity) {

if (entityInformation.isNew(entity)) {

em.persist(entity);

return entity;

} else {

return em.merge(entity);

}

}

See an example [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-save-method/).

## **Tell something about the CrudRepository saveAll() method.**

The CrudRepository saveAll() method used to save multiple entities and internally annotated with @Transactional annotation. It internally uses save() method only as below.

@Transactional

public <S extends T> List<S> saveAll(Iterable<S> entities) {

List<S> result = new ArrayList<S>();

for (S entity : entities) {

result.add(save(entity));

}

return result;

}

See an example [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-saveall-and-findall/).

## **How to write a query method for sorting using Spring Data JPA?**

Consider we have the following entity.

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

The first scenario **–** Retrieve the data on the basis of one field(i.e university) and sort on the basis of another field(name).

public List<Student> findByUniversity(String university) {  
List<Student> response = studentRepository.findByUniversityOrderByNameAsc(university);  
return response;  
}

The second scenario**–** Retrieve all record(rows)  and sort on the basis of some field(name).

public List<Student> findAll() {  
List<Student> response = (List<Student>) studentRepository.findAllByOrderByNameAsc();  
return response;  
}

## 

## **How to implement projection using Spring Data JPA?**

Consider we have an entity called Student.java as below.

package com.netsurfingzone.entity;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

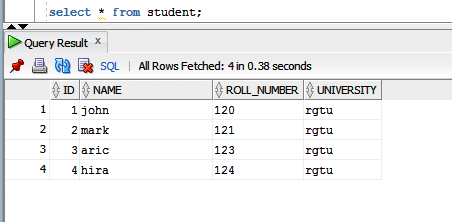
private String rollNumber;

@Column(name = "university")

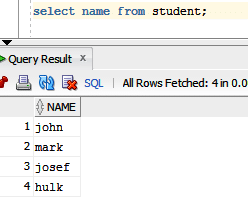
String university;

}

and we have some record in the database as below.

[](https://netsurfingzone.com/jpa/spring-data-jpa-and-or-example-using-spring-boot/attachment/nameandordb/)

We want to fetch any single column. For example, we want all student name as below.

[](https://netsurfingzone.com/jpa/spring-data-jpa-projection-example/attachment/projection/)

Let’s see how to define Projection for the above requirement.

Define Interface with some method.

package com.netsurfingzone.repository;

public interface StudentName {

String getName();

}

Define Repository as below.

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

List<StudentName> findAllBy();

}

Generated query.

Hibernate:  
select  
student0\_.name as col\_0\_0\_  
from  
student student0\_

Some more Spring Data JPA Interview Questions and Answers.

## **How to write query using @NamedQueries and @NamedNativeQueries in Spring Data JPA.**

Both @NamedQueries and @NamedNativeQueries annotations used with the entity.

Using @NamedQueries.

package com.netsurfingzone.entity;

@Entity

@NamedQueries({ @NamedQuery(name = "Student.findByName1", query = "select s from Student s where s.name = ?1"),

@NamedQuery(name = "Student.findByNameAndRollNumber", query = "select s from Student s where s.name = ?1 and s.rollNumber = ?2"),

@NamedQuery(name = "Student.findByNameOrRollNumber", query = "select s from Student s where s.name = ?1 or s.rollNumber = ?2") })

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

Using @NamedNativeQueries.

package com.netsurfingzone.entity;

@Entity

@NamedNativeQueries({

@NamedNativeQuery(name = "Student.findByName", query = "select \* from Student where name = ?1", resultClass = Student.class),

@NamedNativeQuery(name = "Student.findByNameAndRollNumber", query = "select \* from Student where name = ?1 and roll\_number = ?2", resultClass = Student.class),

@NamedNativeQuery(name = "Student.findByNameOrRollNumber", query = "select \* from Student where name = ?1 or roll\_number = ?2", resultClass = Student.class) })

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

Corresponding repository methods.

@Repository

public interface StudentRepository extends CrudRepository<Student, Serializable> {

List<Student> findByName(String name);

List<Student> findByNameAndRollNumber(String name, String rollNumber);

List<Student> findByNameOrRollNumber(String name, String rollNumber);

}

Spring Data JPA Interview Questions and Answers

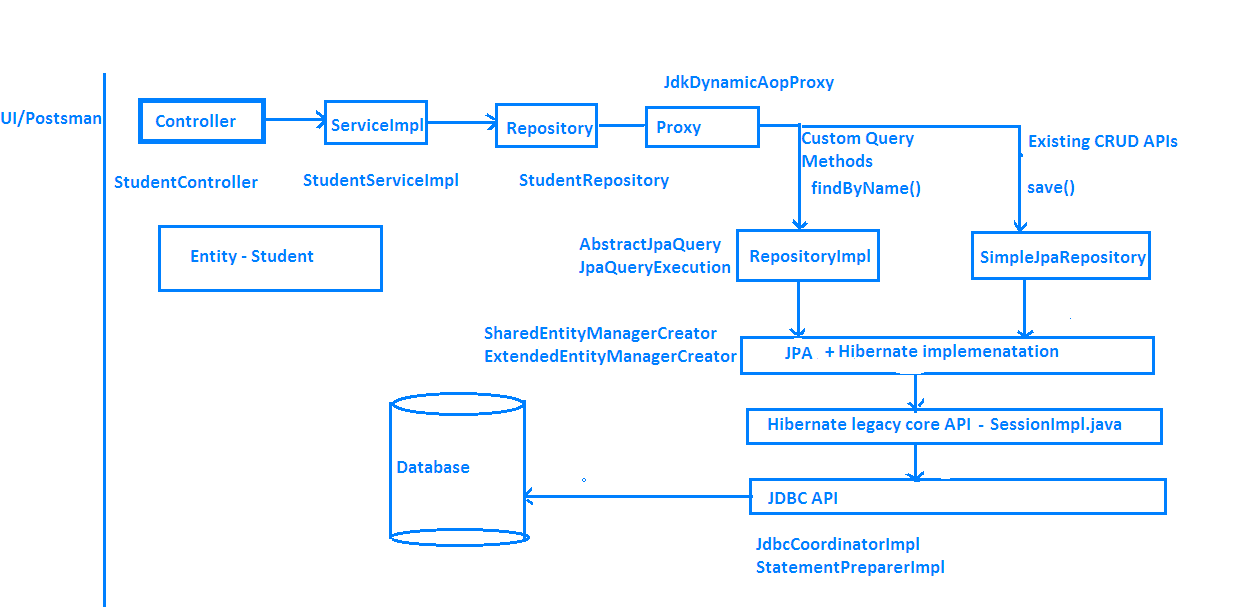
## **Difference between findById() and getOne() in Spring Data JPA?**

Let’s see some difference between findById() and getOne() methods.

|  |  |
| --- | --- |
| findById() | getOne() |
| 1. The findById() method is available in CrudRepository interface. | 1. The getOne() method is available in JpaRepositpry interface. |
| 2. The findById() method will return null if the record doesn’t exist in the database. | 2. The getOne() method throw EntityNotFoundException if the record doesn’t exist in the database. |
| 3. Internally findById() method use EntityManger find() method. | 3. Internally getOne() method use EntityManger getReference() method. |
| 4. Calling findById() returns a eager fetched entity. | 4. Calling getOne() returns a lazily fetched entity. |

See more details [here](https://netsurfingzone.com/jpa/spring-data-findbyid-vs-getone/).

## **Explain the basic flow of Basic Spring Data JPA Flow.**

[](https://netsurfingzone.com/jpa/spring-data-jpa-interview-questions-and-answers/attachment/springdataflow1-2/)

## **Difference between delete() vs deleteInBatch() Methods in Spring Data JPA.**

The delete() method has been defined in the CrudRepository interface with the following signature.

void delete(T entity);

The deleteInBatch() has been defined in the JpaRepository interface with the following signature.

void deleteInBatch(Iterable<T> entities);

The delete() method internally uses EntityManager’s remove() method as below.

@Transactional

public void delete(T entity) {

em.remove(em.contains(entity) ? entity : em.merge(entity));

}

The deleteInBatch() internally implemented as below.

public void deleteInBatch(Iterable<T> entities) {

applyAndBind(getQueryString(DELETE\_ALL\_QUERY\_STRING, entityInformation.getEntityName()), entities, em)

.executeUpdate();

}

Observe both methods implementation. The first one i.e delete() internally uses remove() method. Before call remove() method it calls contains()/merge() method. Then flow will go inside SessionImpl.java class(fireDelete() method) and a lot of stuff happens.

On the other hand, deleteInBatch() prepares the query and collect some other information and directly calls the executeUpdate() method.

With the help of delete() method, we can delete a single record at a time whereas using deleteInBatch() we can delete multiple records.

The delete() method is a little slower as compare deleteInBatch() as delete() does some extra stuff than deleteInBatch().

Almost done with Spring Data JPA Interview Questions and Answers.

## **Difference between deleteAll() Vs deleteAllInBatch() in Spring Data JPA.**

The deleteAll() method is defined in CrudRepository has two overloaded versions.

First one – void deleteAll(Iterable<? extends T> entities)

Using the above method we can delete all entities which we pass as request data. The deleteAll() internally use delete() method only.

public void deleteAll(Iterable<? extends T> entities) {

for (T entity : entities) {

delete(entity);

}

}

Spring Data JPA deleteAll() Vs deleteAllInBatch()

Second one – void deleteAll()

The above one will delete all records that belong to that repository. The deleteAll() internally uses findAll() and delete() method as below.

public void deleteAll() {

for (T element : findAll()) {

delete(element);

}

}

The query generated in case of deleteAll(). Suppose we are deleting twelve entities.

Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?  
Hibernate: delete from student where id=?

The deleteAllInBatch() method is defined in the JpaRepository interface as below.

void deleteAllInBatch()

Using deleteAllInBatch() method we can delete all entities from database. No need to pass entities as parameters.For example, let’s say we have an entity Student.java and also we have a corresponding repository class called StudentRepository.java something like below.

package com.netsurfingzone.repository;

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

}

If we want to delete all student records we need to do studentRepository.deleteAllInBatch().  See the below sample code.

@Transactional

public void deleteAllInBatch() {

studentRepository.deleteAllInBatch();

}

Note – Internally deleteAllInBatch()   uses createQuery() method as below.

public void deleteAllInBatch() {

em.createQuery(getDeleteAllQueryString()).executeUpdate();

}

In case of deleteAllInBatch() the query will generate as below.

Hibernate: delete from student

Both deleteAll() and deleteAllInBatch() use to delete all entities. The deleteAll() internally uses CrudRepository delete() method which further uses EntityManger remove() method which perform some other stuff. But in case of deleteAllInBatch(), EntityManger createQuery() method will preapare the query and it directly calls executeUpdate() method.

## **How to write named parameters in Spring Data JPA?**

@Param and @Query annotations used to define Named Parameters.

Consider we have entity called Student.java.

package com.netsurfingzone.entity;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

Query methods using named parameter in Spring Data JPA.

@Query("select s from Student s where s.name = :name")  
List<Student> findByName(@Param("name") String name);

@Query("select s from Student s where s.name = :name and s.rollNumber = :rollNumber")  
List<Student> findByNameAndRollNumber(@Param("name") String name, @Param("rollNumber") String rollNumber);

## **What will happen when we define wrong Query Methods in Spring Data JPA?**

Suppose we have an entity called Student.java and Student class has fields called id and name. We want all student records for the given name.

Our query method would be List<Student> findByName(String name);

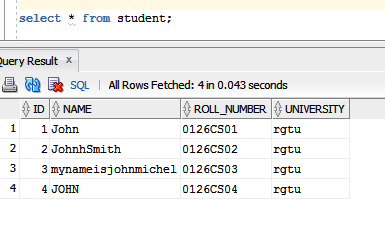
Just think by mistake developer writes findName(). instead of findByName(). In this case, while deployment we will get an error.

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name ‘studentController’: Unsatisfied dependency expressed through field ‘studentService’; nested exception is org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name ‘studentServiceImpl’: Unsatisfied dependency expressed through field ‘studentRepository’; nested exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with name ‘studentRepository’: Invocation of init method failed; nested exception is java.lang.IllegalArgumentException: Failed to create query for method public abstract java.util.List com.netsurfingzone.repository.StudentRepository.findName(java.lang.String)! No property findName found for type Student!

Spring scans all repository methods while deployment itself and checks query methods have been written properly or not.

## **How to define case insensitive search Query Methods in Spring Data JPA?**

Consider we have some records in the database as below.



We have to write a case insensitive search query that means for name “john”, our query method should return two records.

public List<Student> findByNameIgnoreCase(String name);

## **What will the output of the below query method? Will it work?**

@Query("select s from Student s where s.roll\_number = ?1")  
List<Student> findStudents(String rollNumber);

We will get an exception while deployment. Either we need to write JPQL or Native Query.

Writing a JPQL.  
@Query(“select s from Student s where s.rollNumber= ?1”)  
List<Student> findStudents(String rollNumber);

Writing a native query.  
@Query(value = “select \* from Student s where s.roll\_number = ?1”, nativeQuery = true)  
List<Student> findStudents(String rollNumber);

## **List of important keywords and corresponding Query Methods.**

| **Keyword** | **Query methods** | **JPQL** |
| --- | --- | --- |
| And | findByLastnameAndFirstname | ...where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | ...where x.lastname = ?1 or x.firstname = ?2 |
| Is, Equals | findByFirstnameEquals | ...where x.firstname = ?1 |
| Between | findByStartDateBetween | ...where x.startDate between ?1 and ? |
| LessThan | findByAgeLessThan | ...where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | ...where x.age <= ?1< td> |
| GreaterThan | findByAgeGreaterThan | ...where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | ...where x.age >= ?1 |
| After | findByStartDateAfter | ...where x.startDate > ?1 |
| Before | findByStartDateBefore | ...where x.startDate < ?1 |
| IsNull | findByAgeIsNull | ...where x.age is null |
| IsNotNull, NotNull | findByAge(Is)NotNull | ...where x.age not null |
| Like | findByFirstnameLike | ...where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | ...where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | ...where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | ...where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | ...where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | ...where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | ...where x.lastname <> ?1 |
| In | findByAgeIn(Collection ages) | ...where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection ages) | ...where x.age not in ?1 |
| True | findByActiveTrue() | ...where x.active = true |
| False | findByActiveFalse() | ...where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | ...where UPPER(x.firstame) = UPPER(?1) |

That’s all about Spring Data JPA Interview Questions and Answers.

You may like Other Spring Data JPA and Hibernate tutorials.

* [@Version Annotation Example In Hibernate](https://netsurfingzone.com/hibernate/version-annotation-example-in-hibernate/).
* [Hibernate Validator Constraints Example Using Spring Boot](https://netsurfingzone.com/hibernate/hibernate-validator-constraints-example-using-spring-boot/).
* [@Temporal Annotation Example In Hibernate/Jpa Using Spring Boot](https://netsurfingzone.com/hibernate/temporal-annotation-example-in-hibernate-jpa-using-spring-boot-and-oracle/).
* [Hibernate Table Per Concrete Class Spring Boot](https://netsurfingzone.com/hibernate/hibernate-table-per-concrete-class-spring-boot/).
* [Hibernate Table Per Subclass Inheritance Spring Boot.](https://netsurfingzone.com/hibernate/hibernate-table-per-subclass-inheritance-spring-boot/)
* [Hibernate Single Table Inheritance using Spring Boot.](https://netsurfingzone.com/hibernate/inheritance-mapping-single-table-example-in-hibernate-jpa-using-spring-boot/)
* [One To One Mapping Annotation Example in Hibernate/JPA using Spring Boot and Oracle.](https://netsurfingzone.com/hibernate/one-to-one-mapping-annotation-example-in-hibernate-jpa-using-spring-boot-and-oracle)
* [One To One Bidirectional Mapping Example In Hibernate/JPA Using Spring Boot and Oracle](https://netsurfingzone.com/hibernate/one-to-one-bidirectional-mapping-example-in-hibernate-jpa-using-spring-boot-and-oracle).
* [One To Many Mapping Annotation Example In Hibernate/JPA Using Spring Boot And Oracle](https://netsurfingzone.com/hibernate/one-to-many-mapping-annotation-example-in-hibernate-jpa-using-spring-boot-and-oracle).
* [Many To One Unidirectional Mapping In Hibernate/JPA Annotation Example Using Spring Boot and Oracle](https://netsurfingzone.com/hibernate/many-to-one-unidirectional-mapping-in-hibernate-jpa-annotation-example-using-spring-boot-and-oracle).
* [One To Many Bidirectional Mapping In Hibernate/JPA Annotation Example Using Spring Boot and Oracle](https://netsurfingzone.com/hibernate/one-to-many-bidirectional-mapping-in-hibernate-jpa-annotation-example-using-spring-boot-and-oracle).
* [Many To Many Mapping Annotation Example In Hibernate/JPA Using Spring Boot And](https://netsurfingzone.com/hibernate/many-to-many-mapping-annotation-example-in-hibernate-jpa-using-spring-boot-and-oracle)

## Pagination

Have a look at the  EmployeeRepository. The method accepts Pageable arguments. Pageable is an interface defined by Spring, which holds aPageRequest. Let’s see how to create a PageRequest.

1

Pageable pageable = PageRequest.of(0, 10);

2

Page<Employee> page = employeeRepository.findAll(pageable);

In the first line, we created a PageRequestof 10 employees and asked for the first (0) page. The page request was passed to findAll to get a page of Employees as a response.

If we want to access the next set of subsequent pages, we can increase the page number every time.

1

PageRequest.of(1, 10);

2

PageRequest.of(2, 10);

3

PageRequest.of(3, 10);

4

...

## Sorting

**Spring Data JPA** provides a Sort object in order to provide a sorting mechanism. Let's have a look at the ways of sorting.

1

employeeRepository.findAll(Sort.by("fistName"));

2

​

3

employeeRepository.findAll(Sort.by("fistName").ascending().and(Sort.by("lastName").descending());

Obviously, the first one simply sorts by ‘firstName’ and the other one sorts by ‘firstName’ ascending and ‘lastName’ descending.

### ****Pagination and Sort Together****

1

Pageable pageable = PageRequest.of(0, 20, Sort.by("firstName"));

2

​

3

​

4

Pageable pageable = PageRequest.of(0, 20, Sort.by("fistName").ascending().and(Sort.by("lastName").descending());

Certainly! Here are some SQL interview questions along with their answers:

**1. Find Employees with Salary Greater than $50,000**

**Question:** How would you write a query to find all employees with a salary greater than $50,000?

**Answer:**

sql

Copy code

SELECT \* FROM employees

WHERE salary > 50000;

**2. Retrieve Employee Names with Department Names**

**Question:** Write a query to retrieve the names of employees along with their department names, assuming you have employees and departments tables.

**Answer:**

sql

Copy code

SELECT e.name, d.name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id;

**3. Group Employees by Department**

**Question:** How would you write a query to find the number of employees in each department and only include departments with more than 10 employees?

**Answer:**

sql

Copy code

SELECT department\_id, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department\_id

HAVING COUNT(\*) > 10;

**4. Calculate Average Salary per Department**

**Question:** Write a query to calculate the average salary of employees in each department.

**Answer:**

sql

Copy code

SELECT department\_id, AVG(salary) AS average\_salary

FROM employees

GROUP BY department\_id;

**5. Find Employees Earning More than Average Salary**

**Question:** How would you write a query to find employees who earn more than the average salary?

**Answer:**

sql

Copy code

SELECT \* FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

**6. Employees Without a Department**

**Question:** Write a query to find employees who do not belong to any department.

**Answer:**

sql

Copy code

SELECT \* FROM employees

WHERE department\_id IS NULL;

**7. Find Employees Earning More than Their Manager**

**Question:** How would you find all employees who earn more than their manager?

**Answer:**

sql

Copy code

SELECT e.name AS employee\_name, e.salary AS employee\_salary, m.name AS manager\_name, m.salary AS manager\_salary

FROM employees e

JOIN employees m ON e.manager\_id = m.employee\_id

WHERE e.salary > m.salary;

**8. Ranking Employees by Salary within Departments**

**Question:** Write a query to retrieve the employee details along with their rank based on salary within each department.

**Answer:**

sql

Copy code

SELECT name, department\_id, salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS salary\_rank

FROM employees;

**9. Find All Employees Reporting to a Specific Manager**

**Question:** How would you write a recursive query to find all employees who report directly or indirectly to a specific manager?

**Answer:**

sql

Copy code

WITH RECURSIVE EmployeeHierarchy AS (

SELECT employee\_id, name, manager\_id

FROM employees

WHERE manager\_id = 101 -- Example manager\_id

UNION ALL

SELECT e.employee\_id, e.name, e.manager\_id

FROM employees e

INNER JOIN EmployeeHierarchy eh ON e.manager\_id = eh.employee\_id

)

SELECT \* FROM EmployeeHierarchy;

**10. Categorize Salaries Using CASE Statement**

**Question:** Write a query to categorize employees' salaries into 'High', 'Medium', and 'Low' based on predefined salary ranges.

**Answer:**

sql

Copy code

SELECT name, salary,

CASE

WHEN salary > 70000 THEN 'High'

WHEN salary BETWEEN 40000 AND 70000 THEN 'Medium'

ELSE 'Low'

END AS salary\_category

FROM employees;

**11. Find Second Highest Salary**

**Question:** How would you write a query to find the second highest salary in the company without using the LIMIT or TOP keyword?

**Answer:**

sql

Copy code

SELECT MAX(salary) AS second\_highest\_salary

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

**12. Pivot Sales Data by Quarter**

**Question:** Write a query to pivot sales data by quarter for each product.

**Answer:** (This example uses conditional aggregation.)

sql

Copy code

SELECT product\_id,

SUM(CASE WHEN QUARTER(sale\_date) = 1 THEN sale\_amount ELSE 0 END) AS Q1,

SUM(CASE WHEN QUARTER(sale\_date) = 2 THEN sale\_amount ELSE 0 END) AS Q2,

SUM(CASE WHEN QUARTER(sale\_date) = 3 THEN sale\_amount ELSE 0 END) AS Q3,

SUM(CASE WHEN QUARTER(sale\_date) = 4 THEN sale\_amount ELSE 0 END) AS Q4

FROM sales

GROUP BY product\_id;

**13. Correlated Subquery to Compare Salaries within Departments**

**Question:** Write a query to find employees whose salary is above the average salary of their respective department.

**Answer:**

sql

Copy code

SELECT e.name, e.salary, e.department\_id

FROM employees e

WHERE e.salary > (SELECT AVG(salary) FROM employees WHERE department\_id = e.department\_id);

**14. Optimize a Slow-Running Query**

**Question:** Explain how you would optimize a slow-running query.

**Answer:**

* **Check Indexes:** Ensure that indexes are being used on columns involved in JOINs, WHERE, and ORDER BY clauses.
* **Analyze Execution Plan:** Use EXPLAIN or EXPLAIN ANALYZE to understand the query's execution plan and identify bottlenecks.
* \**Avoid SELECT :* Only select the columns you need.
* **Limit Rows:** If possible, limit the number of rows returned using pagination or filtering.
* **Use Joins Wisely:** Replace subqueries with JOINs where appropriate and ensure you're using the most efficient type of JOIN.

**15. Count Rows with NULL Values**

**Question:** Write a query to return the number of employees who have NULL values in any of their columns.

**Answer:**

sql

Copy code

SELECT COUNT(\*) AS null\_count

FROM employees

WHERE name IS NULL OR salary IS NULL OR department\_id IS NULL; -- Add more columns if necessary

**16. Complex JOIN to Find Unpaid Orders**

**Question:** Write a query to get all customers who have placed orders but have not made any payments.

**Answer:**

sql

Copy code

SELECT c.customer\_id, c.name

FROM customers c

JOIN orders o ON c.customer\_id = o.customer\_id

LEFT JOIN payments p ON o.order\_id = p.order\_id

WHERE p.payment\_id IS NULL;

**17. Find Common Employees Between Two Departments**

**Question:** How would you write a query to find all employees who are in both sales and marketing departments?

**Answer:**

sql

Copy code

SELECT employee\_id, name

FROM employees

WHERE department\_id = 'sales'

INTERSECT

SELECT employee\_id, name

FROM employees

WHERE department\_id = 'marketing';

**18. Update Salary of Employees with More Than 5 Years of Experience**

**Question:** Write a query to update the salary of employees by 10% who have been with the company for more than 5 years.

**Answer:**

sql

Copy code

UPDATE employees

SET salary = salary \* 1.10

WHERE hire\_date <= DATE\_SUB(CURDATE(), INTERVAL 5 YEAR);

**19. Query to Utilize Index on a Specific Column**

**Question:** How would you write a query to ensure it makes use of an index on a specific column?

**Answer:**

* Make sure the column you're querying is indexed:

sql

Copy code

CREATE INDEX idx\_column\_name ON table\_name(column\_name);

* Then, write a query that directly uses that column:

sql

Copy code

SELECT \* FROM table\_name WHERE column\_name = 'value';

**20. Stored Procedure to Calculate Total Orders for a Customer**

**Question:** Write a stored procedure to calculate and return the total number of orders for a given customer ID.

**Answer:**

sql

Copy code

DELIMITER //

CREATE PROCEDURE GetTotalOrders(IN customerId INT, OUT totalOrders INT)

BEGIN

SELECT COUNT(\*) INTO totalOrders

FROM orders

WHERE customer\_id = customerId;

END //

DELIMITER ;

* **Usage:**

sql

Copy code

CALL GetTotalOrders(123, @totalOrders);

SELECT @totalOrders;

These answers cover a range of SQL scenarios that are common in technical interviews, particularly for experienced developers.

4o

Select \* from employee e where e.id=1

Select salary from employee where e.id=2

SELECT **DISTINCT** Country FROM Customers; // get unique country names only

SELECT \* FROM Products **ORDER BY** Price; // get price based sorting order

SELECT \* FROM Products **ORDER BY** Price DESC; // Highest to lowest

SELECT **COUNT**(ProductID) FROM Products WHERE Price > 20;

SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 1;

**What is an Index in a Database?**

An index in a database is a data structure **that improves the speed of data retrieval operations** on a table at the cost of additional storage space **and some performance overhead on data modification operations (such as INSERT, UPDATE, DELETE).** Indexes are used to quickly locate and access the data in a database without having to search every row in a table each time a database table is accessed.

**Why Do We Need Indexes?**

Indexes are essential for **optimizing query performance, especially for large datasets.** Without an index, a database must perform a full table scan to find the relevant data, which can be very slow for large tables. By creating an index, the database can quickly narrow down the search to a specific subset of rows.

**Advantages of Indexes**

1. **Faster Data Retrieval**: The primary advantage of using indexes is to speed up data retrieval operations. Queries that involve searching, sorting, or filtering data can benefit significantly from indexes.
2. **Efficient Sorting**: Indexes can be used to quickly retrieve sorted data without performing additional sorting operations.
3. **Enforcing Uniqueness**: Indexes, particularly unique indexes, can be used to enforce the uniqueness of values in a column (e.g., primary key constraints).
4. **Improved JOIN Operations**: Indexes on columns used in JOIN conditions can significantly improve the performance of join operations.

**Disadvantages of Indexes**

1. **Increased Storage Space**: Indexes consume additional disk space. The more indexes you create, the more storage is required.
2. **Slower Data Modifications**: While indexes speed up data retrieval, they can slow down data modification operations like INSERT, UPDATE, and DELETE. This is because the indexes must be updated whenever the data in the indexed columns changes.
3. **Maintenance Overhead**: Indexes require maintenance, especially in tables that undergo frequent data changes. This can lead to fragmentation, which can further degrade performance.
4. **Complexity**: Having too many indexes, or poorly designed indexes, can lead to suboptimal performance and complexity in managing the database schema.

**How to Create an Index**

Indexes can be created using SQL statements. The syntax to create an index varies slightly depending on the database system, but the basic structure is similar.

**Example in SQL:**

sql

Copy code

**CREATE INDEX index\_name**

**ON table\_name (column\_name);**

**Example of Creating an Index in MySQL:**

sql

Copy code

CREATE INDEX idx\_employee\_name

ON employees (last\_name);

**Types of Indexes**

1. **Single-Column Index**: Indexes a single column in a table.
2. **Composite Index**: Indexes multiple columns in a table.
3. **Unique Index**: Ensures that the indexed column(s) have unique values.
4. **Full-Text Index**: Used for full-text search capabilities (e.g., searching large text fields).
5. **Clustered Index**: Determines the physical order of data in a table (usually the primary key). A table can have only one clustered index.
6. **Non-Clustered Index**: Separate from the physical order of data, can have multiple non-clustered indexes per table.

**Conclusion**

Indexes are a powerful tool for optimizing database performance by speeding up data retrieval operations. However, they come with trade-offs, including increased storage requirements and potential slowdowns in data modification operations. Properly designing and maintaining indexes is crucial for achieving the best performance in a database.

**Use Case: E-commerce Application**

Imagine you're working on an e-commerce application with a database table named products that stores information about all the products available on the platform. The table has the following structure:

sql

Copy code

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(255),

category VARCHAR(255),

price DECIMAL(10, 2),

stock INT

);

This table contains millions of rows because your platform has a vast inventory.

**Scenario: Searching for Products by Category**

Users frequently search for products by category. For example, a user might want to see all the products in the "Electronics" category. The SQL query for this search would look like:

sql

Copy code

SELECT \* FROM products WHERE category = 'Electronics';

**The Problem Without an Index**

If there is **no index** on the category column, the database has to perform a **full table scan** to execute this query. A full table scan means that the database will check every row in the products table to find all rows where the category is "Electronics".

For a table with millions of rows, this operation can be **extremely slow** because the database engine must examine each row one by one.

**How an Index Solves the Problem**

To optimize this query, you can create an **index on the category column**. Here's how you would do that:

sql

Copy code

CREATE INDEX idx\_category

ON products (category);

**What Happens After Creating the Index?**

1. **Index Creation**: When you create an index on the category column, the database builds a separate data structure (often a B-tree or similar structure) that stores the category values along with pointers to the corresponding rows in the products table.
2. **Faster Searches**: Now, when the query SELECT \* FROM products WHERE category = 'Electronics'; is executed, the database engine can **quickly locate** the "Electronics" entries in the index, instead of scanning the entire table. It then uses the pointers to directly retrieve the relevant rows from the products table.
3. **Reduced I/O Operations**: The index allows the database to find the matching rows with far fewer I/O operations, significantly speeding up the query.

**Example: Query Performance Improvement**

* **Without Index**: Suppose the products table has 10 million rows. The query might take several seconds (or even minutes) to execute because the database is scanning all 10 million rows.
* **With Index**: After creating the index on category, the same query might take only a few milliseconds. The index allows the database to quickly narrow down the search to just the rows that match the "Electronics" category.

**Practical Considerations**

* **Index Maintenance**: When you INSERT, UPDATE, or DELETE rows in the products table, the category index must also be updated. This can introduce some overhead, but the trade-off is worth it for the performance gains in read-heavy applications like an e-commerce platform.
* **Storage**: The index itself requires additional storage space. However, the benefit of faster queries typically outweighs the cost of this additional storage.
* **Balancing Indexes**: While indexes can greatly improve read performance, having too many indexes can slow down write operations and increase storage requirements. It's important to strike a balance based on the specific access patterns of your application.

**Conclusion**

In this e-commerce use case, an index on the category column dramatically speeds up the search for products in a specific category. The index allows the database to quickly locate relevant rows without scanning the entire table, solving the problem of slow query performance in large datasets.

**Detailed Explanation of Index Creation and How It Works**

When you create an index on a column in a database, the database engine constructs a specialized data structure to optimize the search operations for that column. This data structure is commonly a **B-tree** or a variant of it, though other structures like **hash tables** or **bitmap indexes** might be used depending on the database system and the type of index.

**Step-by-Step Process of Index Creation**

**1. Building the Index (B-tree Structure)**

Let's say you have the following data in your products table:

| **product\_id** | **product\_name** | **category** | **price** | **stock** |
| --- | --- | --- | --- | --- |
| 1 | Laptop | Electronics | 1000 | 50 |
| 2 | Headphones | Electronics | 100 | 200 |
| 3 | T-Shirt | Clothing | 20 | 150 |
| 4 | Blender | Home & Kitchen | 50 | 30 |
| 5 | Smartphone | Electronics | 500 | 75 |

When you create an index on the category column:

sql

Copy code

CREATE INDEX idx\_category ON products (category);

The database engine will build a B-tree index based on the values in the category column.

**B-tree Basics**:

* A **B-tree** is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
* The B-tree stores the category values in a sorted order, and each category value is associated with a pointer (or reference) to the actual rows in the products table.

**Example B-tree for the category Index**:

css

Copy code

[Clothing]

/ \

[Electronics] [Home & Kitchen]

/ | |

[Laptop] [Headphones] [Blender]

|

[Smartphone]

In this simplified representation:

* The **root** of the tree contains pointers to the first values in each branch.
* Each branch (node) holds the sorted category values and pointers to the corresponding rows in the table.

**2. Storing Values and Pointers**

* **Leaf Nodes**: The lowest level of the B-tree (leaf nodes) contains the actual data entries. Each entry in a leaf node includes the category value and a pointer to the specific row(s) in the products table that contain that category.

For example:

* + The leaf node for Electronics might contain pointers to the rows with product\_id = 1, product\_id = 2, and product\_id = 5.
  + The leaf node for Clothing contains a pointer to the row with product\_id = 3.
* **Internal Nodes**: Higher levels of the B-tree (internal nodes) contain pointers to other nodes in the tree. These are used to navigate the tree efficiently. For instance, if you search for "Electronics", the B-tree allows the database to quickly skip other categories and jump directly to the "Electronics" node.

**3. Searching with the Index**

When you execute a query like:

sql

Copy code

SELECT \* FROM products WHERE category = 'Electronics';

The database engine performs the following steps:

* **Traverse the B-tree**: Starting from the root, the engine follows the pointers down the tree to quickly locate the "Electronics" node.
* **Retrieve Row Pointers**: Once the "Electronics" node is found, the engine retrieves the pointers to the rows in the products table that match the category "Electronics".
* **Fetch Data**: Finally, the engine uses these pointers to fetch the full rows from the table, returning the relevant data.

This process is much faster than scanning each row of the table because the B-tree allows the engine to bypass large portions of the data that don't match the search criteria.

**4. Handling Data Modifications**

When data in the products table is modified (e.g., a new product is inserted, or a product's category is updated), the B-tree index must also be updated to reflect these changes:

* **Insertion**: If a new row is inserted with a category that doesn't exist in the index, a new node might be created in the B-tree. If the category already exists, the new row pointer is added to the appropriate leaf node.
* **Update**: If an existing row's category value is updated, the database engine may need to remove the old pointer from one part of the B-tree and insert the new pointer in the appropriate place.
* **Deletion**: If a row is deleted, the corresponding pointer in the B-tree must be removed.

The B-tree's self-balancing nature ensures that the index remains efficient, even as data changes.

**Summary**

* **Indexes** like B-trees are data structures that allow fast searching by keeping data sorted and easily navigable.
* They store the indexed column values (e.g., category) in a structured way, with pointers to the actual rows in the table.
* **B-trees** ensure that the database can find specific data quickly, without needing to scan the entire table, making queries faster.
* However, maintaining these indexes comes with costs like additional storage and slower data modification operations, which are trade-offs for the improved read performance.
* **A stored procedure** An SQL **stored procedure** **is a group of pre-compiled SQL statements (prepared SQL code)** that can be reused by simply calling it whenever needed.
* It can be used to perform a wide range of database operations such as inserting, updating, or deleting data, generating reports, and performing complex calculations. Stored procedures are very useful because they allow you to **encapsulate (bundle) a set of SQL statements as a single unit** and execute them repeatedly with different parameters, making it easy to manage and reuse the code.
* *Procedures have similar structure as functions: they accept parameters and perform operations when we call them. But, the difference between them is that SQL stored procedures are simpler to write or create, whereas functions have a more rigid structure and support fewer clauses.*

DELIMITER $$

CREATE PROCEDURE GetDepartmentEmployees(IN dept\_name VARCHAR(100))

**BEGIN**

SELECT first\_name, last\_name, salary

FROM employees

WHERE department = dept\_name;

**END** $$

DELIMITER ;

How to call a stored procedure :

**CALL** GetDepartmentEmployees('Sales');

**Why Do We Need Stored Procedures?**

1. **Code Reusability**: Stored procedures allow you to Encapsulate complex queries or business logic, making them reusable across multiple applications or parts of an application.
2. **Performance**: Since stored procedures are precompiled and stored in the database, they can be executed faster than sending individual SQL statements from the application.
3. **Security**: Stored procedures can help enforce security by restricting direct access to underlying tables. Users can be granted permission to execute the procedure without granting access to the tables.
4. **Maintainability**: Centralizing complex SQL logic in stored procedures makes it easier to manage and update the code, reducing duplication and maintenance efforts.
5. **Transaction Management**: Stored procedures can manage transactions, ensuring that a series of operations either all succeed or all fail.

**Advantages of Stored Procedures**

1. **Improved Performance**: Precompiled and cached execution plans can reduce the overhead of parsing and optimizing SQL statements.
2. **Security**: Stored procedures provide a way to control data access and operations, reducing the risk of SQL injection attacks.
3. **Code Organization**: Encapsulating business logic in stored procedures keeps the application code cleaner and more organized.
4. **Easier Maintenance**: Changes to business logic can be made in one place (the stored procedure) rather than in multiple application codebases.
5. **Reduced Network Traffic**: Instead of sending multiple SQL statements from the application to the database, you can send a single call to a stored procedure.

**Disadvantages of Stored Procedures**

1. **Complexity**: Writing and maintaining stored procedures can become complex, especially for large applications with many procedures.
2. **Portability Issues**: Stored procedures are often written in a database-specific language (e.g., PL/SQL for Oracle, T-SQL for SQL Server), making it difficult to migrate to a different database system.
3. **Limited Debugging Tools**: Debugging stored procedures can be more challenging than debugging application code, as database-specific debugging tools are often less powerful.
4. **Version Control**: Managing stored procedures in version control systems can be more challenging compared to application code.

**Triggers**

A trigger is a stored procedure in a database that automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated. In simple words, a trigger is a collection of [SQL](https://www.geeksforgeeks.org/sql-tutorial)statements with particular names that are stored in system memory. It belongs to a specific class of stored procedures that are automatically invoked in response to database server events. Every trigger has a table attached to it.

Because a trigger cannot be called directly, unlike a stored procedure, it is referred to as a special procedure. A trigger is automatically called whenever a data modification event against a table takes place, which is the main distinction between a trigger and a procedure. On the other hand, a stored procedure must be called directly.

The following are the key differences between triggers and stored procedures:

1. Triggers cannot be manually invoked or executed.
2. There is no chance that triggers will receive parameters.
3. A transaction cannot be committed or rolled back inside a trigger.

**Syntax:**

*create trigger [trigger\_name]*

*[before | after]*

*{insert | update | delete}*

*on [table\_name]*

*[for each row]*

*[trigger\_body]*

***Why Do We Need Triggers?***

1. ***Automatic Enforcement of Business Rules****: Triggers ensure that complex business rules are consistently enforced without relying on application logic.*
2. ***Data Integrity****: Triggers can automatically enforce data integrity rules, such as cascading updates or deletes, ensuring that related data remains consistent.*
3. ***Auditing and Logging****: Triggers can be used to automatically log changes to sensitive data, providing an audit trail of who changed what and when.*
4. ***Automatic Data Validation****: Triggers can validate or modify data before it is inserted, updated, or deleted, ensuring that only valid data is stored.*

***Advantages of Triggers***

1. ***Automation****: Triggers automatically execute in response to specific events, eliminating the need for manual intervention.*
2. ***Consistent Enforcement****: Triggers ensure that business rules and data integrity constraints are consistently enforced at the database level.*
3. ***Centralized Logic****: Triggers allow you to centralize complex logic within the database, reducing the need for duplicate code in applications.*
4. ***Immediate Action****: Triggers respond immediately to data changes, ensuring that actions are taken as soon as the relevant event occurs.*

***Disadvantages of Triggers***

1. ***Complexity****: Triggers can add complexity to the database schema, making it harder to understand and maintain.*
2. ***Performance Overhead****: Triggers can introduce performance overhead, especially if they involve complex logic or operate on large datasets.*
3. ***Hidden Logic****: Since triggers are executed automatically, their effects can be less visible and harder to trace, leading to unexpected behavior.*
4. ***Debugging Challenges****: Debugging triggers can be difficult because they are not explicitly called in the application code, making it hard to determine what went wrong.*

**Stored Procedures:** "Stored procedures are precompiled SQL statements stored in the database that can be executed with a single call. They are used to encapsulate complex queries, improve performance through caching, and enhance security by controlling access to data. However, they can increase complexity and reduce portability across different database systems."

**Triggers:** "Triggers are automatic responses to specific events in the database, such as data changes. They are used to enforce business rules, maintain data integrity, and log changes. While they automate tasks and ensure consistent data management, they can introduce performance overhead and make debugging more complex."

***Normalization and Demoralization***

*R*esponse: **"Normalization** is the process of organizing a database to reduce redundancy and improve data integrity. It involves **dividing the database into multiple related tables,** which helps **avoid duplicate data** and anomalies when performing data operations like insert, update, or delete. For example, if customer information is stored in multiple places, updating a customer's address in a normalized database ensures that it is updated everywhere without causing inconsistencies.

However, in a highly normalized database, queries can become slow due to the need for multiple joins. This is where denormalization comes in.

**Denormalization** is the process of intentionally **adding redundancy back into the database to improve read performance.** For example, by combining customer and order tables into a single table, we can speed up queries that retrieve customer orders, even though this means storing the customer's information multiple times.

The key trade-off is between data integrity and query performance. **Normalization ensures consistency and reduces redundancy**, while **denormalization optimizes performance at the cost of increased redundancy** and potential data anomalies."

**SQL Injection:** "SQL Injection is a security vulnerability where an attacker manipulates SQL queries by injecting malicious input. This can lead to unauthorized access or damage to the database. For example, if user inputs are directly included in SQL queries without validation, an attacker might bypass authentication or alter data."

**Protection Measures:** "To protect against SQL injection, use prepared statements or parameterized queries to separate SQL code from data. Stored procedures can also help if used correctly. Additionally, validate and sanitize all user inputs, use ORM frameworks for data access, adhere to the principle of least privilege, perform regular security testing, and handle errors properly to avoid exposing sensitive information."

**Joins**

**Syntax: SELECT \* FROM TableA LEFT JOIN TableB ON TableA.id = TableB.id;**

 **INNER JOIN:** Use for matching records from both tables.

 **LEFT JOIN:** Use for all records from the left table and matching records from the right table.

 **RIGHT JOIN:** Use for all records from the right table and matching records from the left table.

 **SELF JOIN:** Self join allows you to join a table with itself to retrieve data based on a specific condition.

 **FULL JOIN:** Returns all records from both tables, and where there are no matches, it returns NULL for the missing side.

 **CROSS JOIN:** Use for Cartesian products, combining all rows from both tables.

**Important Database concepts to know by Java Developer**

As a Java developer, especially one working with enterprise applications, it's important to understand several key database concepts. These concepts are crucial for effectively designing, developing, and maintaining applications that interact with databases. Here are the most important database concepts you should know:

**1. SQL (Structured Query Language)**

* **Basics:** Understanding the fundamentals of SQL, including SELECT, INSERT, UPDATE, and DELETE statements.
* **Joins:** Inner joins, left joins, right joins, and full outer joins.
* **Subqueries:** Using nested queries for complex data retrieval.
* **Aggregations:** Functions like COUNT, SUM, AVG, MIN, and MAX.
* **Grouping:** Using GROUP BY and HAVING for data aggregation.
* **Sorting:** Using ORDER BY for sorting results.

**2. Database Design**

* **Normalization:** Understanding the process of organizing data to reduce redundancy (1NF, 2NF, 3NF, BCNF).
* **Denormalization:** Knowing when and why to denormalize for performance reasons.
* **Indexes:** Types of indexes (B-tree, bitmap), when to use them, and how they affect performance.
* **Primary and Foreign Keys:** The importance of these constraints in relational databases.
* **ER Diagrams:** Understanding entity-relationship diagrams for modeling data.

**3. Transactions and ACID Properties**

* **Transactions:** Grouping multiple SQL operations into a single unit of work.
* **ACID:** Understanding Atomicity, Consistency, Isolation, and Durability.
* **Isolation Levels:** Different isolation levels (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE) and their trade-offs.

**4. Connection Management**

* **JDBC (Java Database Connectivity):** Basics of connecting to a database using JDBC.
* **Connection Pooling:** Importance of connection pooling for performance, using tools like HikariCP, Apache DBCP.
* **ORM (Object-Relational Mapping):** Using tools like Hibernate or JPA to map Java objects to database tables.

**5. Data Integrity**

* **Constraints:** Primary key, foreign key, unique, not null, check constraints.
* **Referential Integrity:** Ensuring that relationships between tables remain consistent.
* **Triggers:** Database triggers to enforce business rules or data integrity.

**6. Performance Tuning**

* **Query Optimization:** Writing efficient queries and understanding how the database query optimizer works.
* **Index Tuning:** Understanding the impact of indexes on query performance.
* **Caching:** Using techniques like query caching or application-level caching (e.g., Ehcache, Redis) to improve performance.
* **Database Profiling:** Analyzing slow queries using tools like EXPLAIN in MySQL or EXPLAIN PLAN in Oracle.

**7. Data Modeling**

* **Relational vs. NoSQL Databases:** Knowing when to use relational databases (like MySQL, PostgreSQL) versus NoSQL databases (like MongoDB, Cassandra).
* **Schema Design:** Designing a schema that fits the application requirements.
* **Entity Relationships:** Understanding one-to-one, one-to-many, and many-to-many relationships.

**8. Database Security**

* **User Management:** Creating and managing database users and permissions.
* **Encryption:** Encrypting sensitive data at rest and in transit.
* **SQL Injection:** Understanding how to protect against SQL injection attacks.
* **Auditing:** Keeping track of changes and access to sensitive data.

**9. Backup and Recovery**

* **Backup Strategies:** Full, incremental, and differential backups.
* **Disaster Recovery:** Strategies for recovering from data loss or corruption.
* **Point-in-Time Recovery:** Recovering a database to a specific point in time.

**10. Working with Large Data Sets**

* **Partitioning:** Splitting large tables into smaller, more manageable pieces.
* **Sharding:** Distributing data across multiple servers.
* **ETL Processes:** Extract, Transform, Load processes for moving and transforming large amounts of data.

**11. Advanced SQL Concepts**

* **Stored Procedures and Functions:** Encapsulating logic in the database.
* **Views:** Creating virtual tables to simplify complex queries.
* **Window Functions:** Advanced SQL functions for analytical queries (e.g., ROW\_NUMBER, RANK, LEAD, LAG).

**12. ORM Frameworks**

* **JPA (Java Persistence API):** Understanding annotations, entity mappings, and the entity lifecycle.
* **Hibernate:** Using Hibernate as an implementation of JPA, including advanced features like caching, batch processing, and fetching strategies.

**13. Database Migration**

* **Tools:** Using tools like Liquibase or Flyway for versioning and applying database schema changes.
* **Migration Strategies:** Strategies for migrating data across different environments or database systems.

These concepts are essential for developing robust, efficient, and secure applications that interact with databases.

The “N+1 Select Problem” is a common performance issue that occurs in applications that interact with a database, particularly in object-relational mapping (ORM) frameworks like Hibernate, JPA, or Entity Framework. The problem arises when the application performs N+1 database queries instead of a single, optimized query.

**Explanation of the N+1 Select Problem**

* **Scenario:** Suppose you have a database with two tables: Parent and Child, where each parent has multiple children. You want to fetch all parents along with their children.
* **Problematic Query:**
  + **Step 1:** The application first queries the Parent table to fetch all parents. Let's say there are N parents in the database.
  + **Step 2:** For each of these N parents, the application then issues a separate query to fetch the associated children from the Child table. This results in N additional queries, one for each parent.

So, in total, the application issues 1 (initial query) + N (queries for each parent’s children) = N+1 queries.

**Why It's a Problem**

The N+1 Select Problem leads to performance issues because instead of retrieving all necessary data with a single optimized query, the application performs a large number of small queries. This can significantly increase the load on the database, especially as N grows larger, leading to:

* **Increased Latency:** The round-trip time between the application and the database increases with each query, resulting in slower response times.
* **Higher Load on the Database:** Multiple queries put additional stress on the database, potentially affecting other operations.
* **Inefficiency:** The same data might be fetched multiple times, leading to redundant work.

**Example**

Let’s say you have a Parent entity and a Child entity with a one-to-many relationship. Here’s how the N+1 problem might manifest:

**1. Initial Query to Fetch Parents**

sql

Copy code

SELECT \* FROM Parent;

Assume this query returns 10 parents.

**2. Additional Queries for Each Parent’s Children**

For each of the 10 parents, the application runs a query like this:

sql

Copy code

SELECT \* FROM Child WHERE parent\_id = 1;

SELECT \* FROM Child WHERE parent\_id = 2;

...

SELECT \* FROM Child WHERE parent\_id = 10;

This results in 10 additional queries, one for each parent.

**How to Avoid the N+1 Select Problem**

To avoid the N+1 Select Problem, you can use strategies like:

1. **Eager Loading:**
   * Fetch the parent entities along with their associated child entities in a single query using a JOIN.
   * Example in JPQL (Java Persistence Query Language):

java

Copy code

SELECT p FROM Parent p JOIN FETCH p.children;

1. **Batch Fetching:**
   * Fetch associated entities in batches rather than one at a time. Some ORM frameworks allow you to configure batch fetching.
2. **Optimized Queries:**
   * Write a custom query to fetch all the necessary data at once. For example, a single query using JOIN or a SELECT ... IN (...) approach.
   * Example in SQL:

sql

Copy code

SELECT p.\*, c.\* FROM Parent p

LEFT JOIN Child c ON p.id = c.parent\_id;

1. **Lazy Loading with Caution:**
   * While lazy loading can sometimes cause the N+1 problem, it’s useful in scenarios where associated data is rarely needed. Combine lazy loading with fetching strategies to avoid unnecessary queries.

By understanding and addressing the N+1 Select Problem, you can significantly improve the performance and scalability of your application.

4o

**You said:**

how does lazy loading and eager loading works what are advantages and disadvantages with this

**ChatGPT said:**

ChatGPT

**Lazy Loading** and **Eager Loading** are two strategies used in object-relational mapping (ORM) to manage how related data is fetched from a database. Each has its own advantages and disadvantages, and the choice between them depends on the specific use case.

**Lazy Loading**

**How It Works:**

* **Lazy Loading** delays the loading of related data until it is explicitly accessed.
* When you fetch an entity from the database, its related entities (e.g., child objects in a parent-child relationship) are not immediately loaded. Instead, a proxy or placeholder is returned.
* The related data is fetched only when it is accessed for the first time (i.e., when you try to use it).

**Example:**

java

Copy code

@Entity

public class Parent {

@OneToMany(fetch = FetchType.LAZY)

private List<Child> children;

}

// Accessing the children for the first time triggers a query to the database

List<Child> children = parent.getChildren();

**Advantages:**

1. **Performance Optimization**: Improves performance by loading only the data that is actually needed, reducing unnecessary database queries.
2. **Memory Efficiency**: Reduces memory usage by not loading large collections of data that might never be accessed.
3. **Scalability**: Works well in scenarios where not all related data is needed immediately or at all.

**Disadvantages:**

1. **N+1 Select Problem**: Can lead to the N+1 Select Problem, where multiple queries are executed to load related data incrementally, causing performance issues.
2. **Increased Complexity**: Managing lazy-loaded data can be complex, especially in larger applications or when dealing with detached entities.
3. **Potential for Runtime Errors**: If lazy-loaded data is accessed after the session/transaction has been closed, it can lead to runtime errors (e.g., LazyInitializationException in Hibernate).

**Eager Loading**

**How It Works:**

* **Eager Loading** loads related data immediately when the parent entity is fetched.
* When you retrieve an entity, its related entities are also fetched in the same database query, usually through a JOIN.

**Example:**

java

Copy code

@Entity

public class Parent {

@OneToMany(fetch = FetchType.EAGER)

private List<Child> children;

}

// Accessing the parent automatically loads the children

List<Child> children = parent.getChildren(); // Already loaded

**Advantages:**

1. **Simplified Code**: Simplifies code by ensuring all related data is available immediately, reducing the risk of LazyInitializationException.
2. **Avoids N+1 Problem**: Because related entities are loaded with a single query, it avoids the N+1 Select Problem.
3. **Consistent Data**: Guarantees that all necessary data is loaded upfront, which can be important in complex transactions.

**Disadvantages:**

1. **Performance Overhead**: Can lead to performance overhead by loading more data than is actually needed, especially with large data sets or complex relationships.
2. **Increased Memory Usage**: Consumes more memory by loading all related entities, which might be unnecessary.
3. **Reduced Flexibility**: Less flexible in scenarios where you only need some of the related data, as you have no control over what gets loaded.

**When to Use Each**

* **Lazy Loading**:
  + Use when related data is not always needed.
  + Ideal for scenarios where memory usage is a concern or where the related data is large or infrequently accessed.
  + Suitable for UI interactions where data is loaded on-demand, such as when expanding a list or navigating through pages.
* **Eager Loading**:
  + Use when you know you will always need the related data.
  + Ideal for batch processing or reports where all data is processed together.
  + Suitable for scenarios where you want to avoid the N+1 Select Problem or when data consistency across the entire object graph is critical.

**Summary**

* **Lazy Loading**:
  + **Pros:** Performance, memory efficiency, scalability.
  + **Cons:** N+1 Select Problem, complexity, potential runtime errors.
* **Eager Loading**:
  + **Pros:** Simplicity, avoids N+1 Select Problem, consistent data access.
  + **Cons:** Performance overhead, increased memory usage, reduced flexibility.

The choice between lazy and eager loading should be guided by the specific needs of your application, the nature of your data, and your performance considerations.

4o

**You said:**

explain internal working of lazy and eager with some use case example

**ChatGPT said:**

ChatGPT

**Internal Working of Lazy Loading and Eager Loading**

Both **Lazy Loading** and **Eager Loading** are techniques used in ORMs (Object-Relational Mappers) like Hibernate to manage how related entities are loaded from the database. Below, I’ll explain the internal workings of both strategies, along with examples and use cases.

**1. Lazy Loading**

**Internal Working:**

* **Proxy Creation:** When you configure a relationship to be lazily loaded (e.g., using @OneToMany(fetch = FetchType.LAZY)), the ORM does not immediately load the related entities from the database. Instead, it creates a proxy object, which is a placeholder or a lightweight wrapper around the related entity.
* **Proxy Interception:** The proxy object intercepts any method calls on the related entity. When you try to access a property or a method of the related entity, the proxy triggers a database query to load the actual data from the database.
* **Loading Data:** The ORM then fetches the required data from the database and populates the proxy with this data. The proxy object now behaves like the actual entity.

**Example Use Case:**

Imagine you have an application for managing schools, and you have the following entities:

* School: Represents a school.
* Student: Represents a student belonging to a school.

You want to load School entities frequently, but you only want to load the associated Student entities when necessary (e.g., when viewing details about a specific school).

java

Copy code

@Entity

public class School {

@Id

private Long id;

private String name;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "school")

private List<Student> students;

// getters and setters

}

@Entity

public class Student {

@Id

private Long id;

private String name;

@ManyToOne

private School school;

// getters and setters

}

java

Copy code

// Fetching a school without loading its students

School school = entityManager.find(School.class, 1L);

// At this point, students are not loaded yet

List<Student> students = school.getStudents(); // This triggers a SQL query

* **Use Case:**
  + **Scenario:** You load a list of schools on a UI without displaying the students initially. Students should only be loaded when the user explicitly requests to see them (e.g., by clicking on a school).
  + **Advantage:** This reduces the amount of data loaded initially, improving performance and memory usage, especially if the Student list is large.

**Internal SQL Operations:**

* **Initially:** Only a query to load the School entity is executed:

sql

Copy code

SELECT \* FROM School WHERE id = 1;

* **On Access:** When the students list is accessed, a query to load the students is triggered:

sql

Copy code

SELECT \* FROM Student WHERE school\_id = 1;

**2. Eager Loading**

**Internal Working:**

* **Join or Fetch:** When you configure a relationship to be eagerly loaded (e.g., using @OneToMany(fetch = FetchType.EAGER)), the ORM immediately loads the related entities along with the main entity using a single query. The ORM uses a JOIN or similar technique to fetch all related data in one go.
* **Direct Population:** There is no proxy creation. Instead, the ORM directly populates the related entities and includes them in the parent entity when it is first retrieved from the database.

**Example Use Case:**

Let’s extend the previous example to a scenario where you always need the list of students when you load a school.

java

Copy code

@Entity

public class School {

@Id

private Long id;

private String name;

@OneToMany(fetch = FetchType.EAGER, mappedBy = "school")

private List<Student> students;

// getters and setters

}

java

Copy code

// Fetching a school along with its students in one query

School school = entityManager.find(School.class, 1L);

// Students are already loaded at this point

List<Student> students = school.getStudents(); // No additional query is triggered

* **Use Case:**
  + **Scenario:** You are building a report or a view where you need to display schools along with all their students. Here, eager loading ensures that all necessary data is fetched in one go, avoiding the need for additional queries.
  + **Advantage:** This approach is efficient when you know that the related data will always be needed, avoiding the potential performance pitfalls of lazy loading.

**Internal SQL Operations:**

* **Single Query:** The ORM generates a single query using a JOIN to fetch both the School and its associated Student entities:

sql

Copy code

SELECT s.\*, st.\* FROM School s

LEFT JOIN Student st ON s.id = st.school\_id

WHERE s.id = 1;

**Summary of Advantages and Disadvantages:**

**Lazy Loading**

* **Advantages:**
  + Reduces initial load time and memory usage.
  + Loads only what is necessary, making it more efficient for large datasets.
* **Disadvantages:**
  + Can lead to the N+1 Select Problem, where multiple small queries degrade performance.
  + Risk of LazyInitializationException if the proxy is accessed outside of the session/transaction.
  + More complex to manage in certain situations.

**Eager Loading**

* **Advantages:**
  + Simple to use as all required data is available immediately.
  + Avoids N+1 Select Problem by using a single query.
  + Reduces the risk of runtime exceptions related to uninitialized proxies.
* **Disadvantages:**
  + Can load more data than necessary, leading to higher memory consumption and slower initial load times.
  + Less flexible if the related data is not always needed.

**Choosing Between Lazy and Eager Loading:**

* **Use Lazy Loading** when:
  + Related data is large and may not always be needed.
  + You want to optimize initial performance and memory usage.
* **Use Eager Loading** when:
  + Related data is small and is always needed along with the main entity.
  + You want to ensure all data is available immediately without additional queries.

Choosing between lazy and eager loading requires understanding the specific needs of your application and the trade-offs involved in each approach.

**How Eager and Lazy Queries are Generated**

In ORM frameworks, the way queries are generated for eager and lazy loading is fundamental to their performance characteristics. Here’s a detailed look at how each approach generates SQL queries:

**Eager Loading**

**How Eager Queries Are Generated:**

1. **Single Query with Joins:**
   * When an association is configured to use eager loading (e.g., fetch = FetchType.EAGER), the ORM generates a single SQL query that includes a JOIN clause to fetch both the parent and its related entities.
   * The idea is to retrieve all necessary data in one query, reducing the number of database round-trips.
2. **Query Structure:**
   * Typically, the ORM constructs a query using SQL JOIN operations. If there are multiple levels of associations, the query might include multiple JOIN clauses.
   * For example, if fetching a Parent with its Children, the ORM generates a query like:

sql

Copy code

SELECT p.\*, c.\*

FROM Parent p

LEFT JOIN Child c ON p.id = c.parent\_id

WHERE p.id = 1;

* + This query retrieves the Parent and associated Child records in a single operation.

**Example Use Case:**

Consider the following entities:

java

Copy code

@Entity

public class Parent {

@Id

private Long id;

private String name;

@OneToMany(fetch = FetchType.EAGER, mappedBy = "parent")

private List<Child> children;

// getters and setters

}

@Entity

public class Child {

@Id

private Long id;

private String name;

@ManyToOne

private Parent parent;

// getters and setters

}

When you query for a Parent, the generated SQL might be:

sql

Copy code

SELECT p.id, p.name, c.id, c.name

FROM Parent p

LEFT JOIN Child c ON p.id = c.parent\_id

WHERE p.id = 1;

**Lazy Loading**

**How Lazy Queries Are Generated:**

1. **Initial Query Only:**
   * When an association is configured to use lazy loading (e.g., fetch = FetchType.LAZY), the ORM generates an initial query to load the main entity (parent) only.
   * The related entities (children) are not fetched at this point; instead, a proxy or placeholder is returned.
2. **Subsequent Queries on Demand:**
   * When the application accesses the related entities (e.g., getChildren()), the ORM generates additional queries to fetch the related data.
   * For each access, the ORM executes a separate query to retrieve the related entities, which can lead to the N+1 Select Problem if not managed carefully.
3. **Query Structure:**
   * The initial query for the parent might look like:

sql

Copy code

SELECT \* FROM Parent WHERE id = 1;

* + When accessing the children, the ORM generates a subsequent query like:

sql

Copy code

SELECT \* FROM Child WHERE parent\_id = 1;

**Example Use Case:**

With the same entities as above:

java

Copy code

@Entity

public class Parent {

@Id

private Long id;

private String name;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "parent")

private List<Child> children;

// getters and setters

}

@Entity

public class Child {

@Id

private Long id;

private String name;

@ManyToOne

private Parent parent;

// getters and setters

}

When you query for a Parent, the SQL might be:

sql

Copy code

SELECT \* FROM Parent WHERE id = 1;

Later, when you access the children:

java

Copy code

List<Child> children = parent.getChildren(); // triggers another query

The ORM then generates:

sql

Copy code

SELECT \* FROM Child WHERE parent\_id = 1;

**Comparison of Eager and Lazy Query Generation:**

* **Eager Loading:**
  + **Single Query with Joins:** Efficient when all related data is needed upfront.
  + **Pros:** Reduces the number of queries and avoids the N+1 Select Problem.
  + **Cons:** Can lead to performance issues with large datasets and increased memory consumption due to retrieving all related data at once.
* **Lazy Loading:**
  + **Initial Query Only:** Starts with a basic query and fetches related data on demand.
  + **Pros:** Reduces initial load time and memory usage; only fetches data when required.
  + **Cons:** Can lead to the N+1 Select Problem and multiple database round-trips, which may degrade performance if not managed correctly.

**Summary**

* **Eager Loading** generates a single, complex query that retrieves both the main entity and its related entities in one go, using JOIN operations. This approach simplifies data access but can impact performance with large data sets.
* **Lazy Loading** generates separate queries for the main entity and each related entity as needed. It improves initial performance and memory usage but can lead to performance issues due to multiple queries (N+1 Select Problem) if not carefully managed.

4o mini

**CascadeType.ALL** to ensure that changes on one side of a relationship reflect on the other side, particularly in scenarios involving deletion,. However, the details and behavior depend on the type of relationship (e.g., one-to-many, many-to-many) and whether you need to handle orphan removal.

Here's a breakdown of how CascadeType.ALL and related configurations handle deletion and reflect changes in bidirectional relationships:

**CascadeType.ALL**

* **Definition:** CascadeType.ALL is a cascade type that includes all possible cascade operations: PERSIST, MERGE, REMOVE, REFRESH, and DETACH.
* **Usage:** When you use CascadeType.ALL, it means that all operations performed on the parent entity will be cascaded to its associated child entities. This includes automatic deletion of child entities when the parent entity is deleted.

**CascadeType.PERSIST**

* **Definition:** Propagates the persist operation from the parent entity to its child entities.
* **Usage:** When you persist a parent entity, all associated child entities are also persisted.

**CascadeType.MERGE**

* **Definition:** Propagates the merge operation from the parent entity to its child entities.
* **Usage:** When you merge a parent entity (update an existing entity), all associated child entities are also merged.

**CascadeType.REMOVE**

* **Definition:** Propagates the remove operation from the parent entity to its child entities.
* **Usage:** When you remove a parent entity, all associated child entities are also removed.

**ascadeType.REFRESH**

* **Definition:** Propagates the refresh operation from the parent entity to its child entities.
* **Usage:** When you refresh a parent entity (reload from the database), all associated child entities are also refreshed.

**CascadeType.DETACH**

* **Definition:** Propagates the detach operation from the parent entity to its child entities.
* **Usage:** When you detach a parent entity (remove it from the persistence context), all associated child entities are also detached.

"best" choice for primary key generation strategies, as the optimal strategy depends on various factors such as your specific use case, database system, performance requirements, and application architecture. Here’s a quick guide to help you choose the most appropriate strategy based on common scenarios:

**When to Use Each Strategy**

1. **GenerationType.IDENTITY**

 **How It Works:**

* Relies on **the database's auto-increment feature to generate unique primary key values.**
* The primary key is automatically generated by the database as new rows are inserted.
* The database manages the incrementing value internally.
  + **Best For:**
    - **Simple Applications:** When your application doesn't involve high concurrency or complex transactions.
    - **Auto-Increment Databases:** If you are using databases that support auto-increment columns well (e.g., MySQL).
    - **Low Overhead:** If you prefer a straightforward approach with minimal configuration.
  + **Advantages:**
    - Easy to set up.
    - No need for additional database objects like sequences.
  + **Disadvantages:**
    - May be less efficient in high-concurrency environments or when performing batch operations.
    - Less flexible for applications requiring database portability.

1. **GenerationType.SEQUENCE**

 **How It Works:**

* Uses a database sequence object to generate unique primary key values.
* A sequence is a database object specifically designed to generate unique numeric values in a sequential manner.
* The sequence can be shared among multiple tables or used by multiple entities.
  + **Best For:**
    - **High Concurrency:** When your application involves high concurrency and you need efficient key generation.
    - **Batch Operations:** If you perform batch inserts or updates.
    - **Databases with Sequence Support:** Ideal for databases like Oracle, PostgreSQL, or others that support sequences.
  + **Advantages:**
    - Efficient for high-concurrency environments.
    - Can pre-allocate a batch of IDs, reducing database round-trips.
    - Generally performs better with batch operations and large-scale data handling.
  + **Disadvantages:**
    - Requires the creation and management of a sequence object in the database.
    - Slightly more complex configuration compared to IDENTITY.

1. **GenerationType.TABLE**
   * **Best For:**
     + **Compatibility:** When using databases that don’t support sequences or auto-increment columns.
     + **Database Portability:** If you need a strategy that works across different database systems.
   * **Advantages:**
     + Can be used with any database, making it a good choice for database portability.
   * **Disadvantages:**
     + Generally less efficient than SEQUENCE and IDENTITY.
     + Can lead to performance issues due to frequent table updates and locking.
2. **GenerationType.AUTO**
   * **Best For:**
     + **Default Configuration:** When you want the JPA provider (like Hibernate) to automatically choose the best strategy based on the underlying database.
   * **Advantages:**
     + Simplifies configuration by letting the JPA provider handle the strategy selection.
   * **Disadvantages:**
     + Less control over the exact strategy used.
     + Behavior might vary depending on the JPA provider and database combination.

**Choosing the Best Strategy**

* **For High Performance and Scalability:**
  + **GenerationType.SEQUENCE** is often preferred due to its efficiency in handling high-concurrency and batch operations.
* **For Simplicity and Standard Use Cases:**
  + **GenerationType.IDENTITY** is suitable for applications where simplicity is more important than handling high-concurrency or batch operations.
* **For Database Portability:**
  + **GenerationType.TABLE** provides a portable solution but may come with performance trade-offs.
* **For Flexibility and Provider-Specific Solutions:**
  + **GenerationType.AUTO** allows the JPA provider to select the strategy, which can be a good default choice if you're unsure.

Ultimately, the "best" choice depends on your application’s needs and the specifics of your database and environment. Evaluating your performance requirements, concurrency levels, and database capabilities will help determine the most appropriate strategy.