**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.

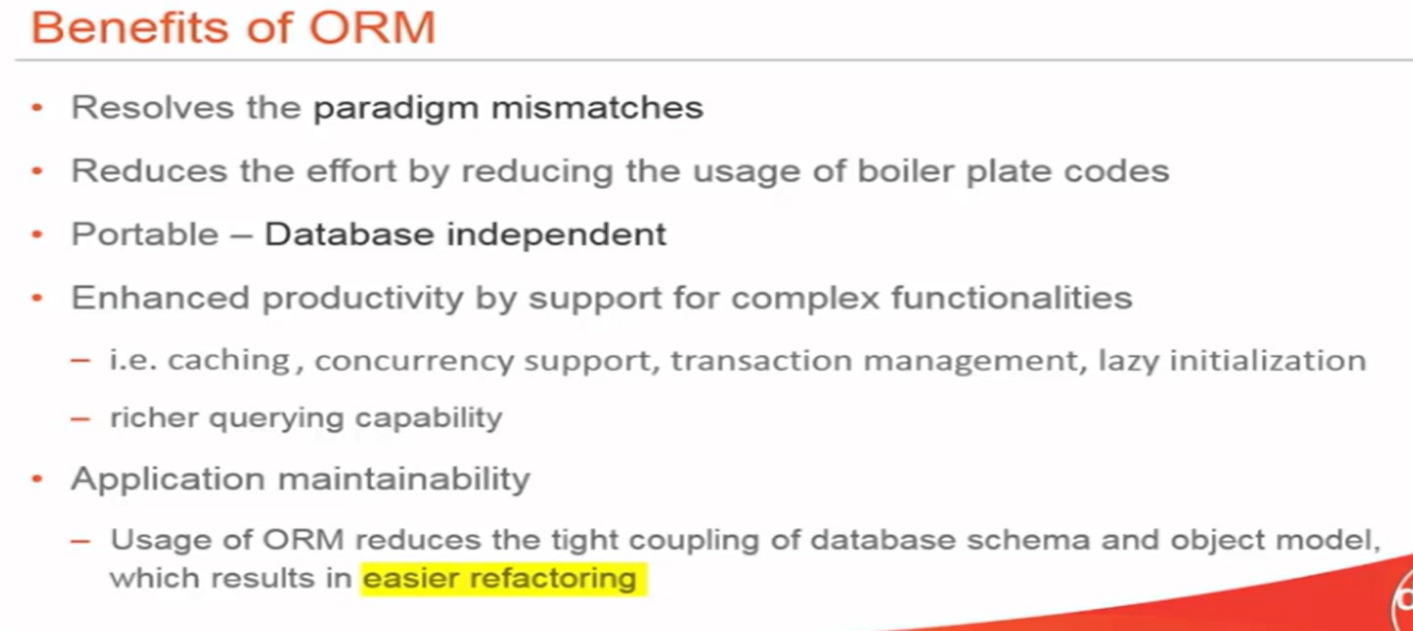
### 6) Provides Query Statistics and Database Status

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

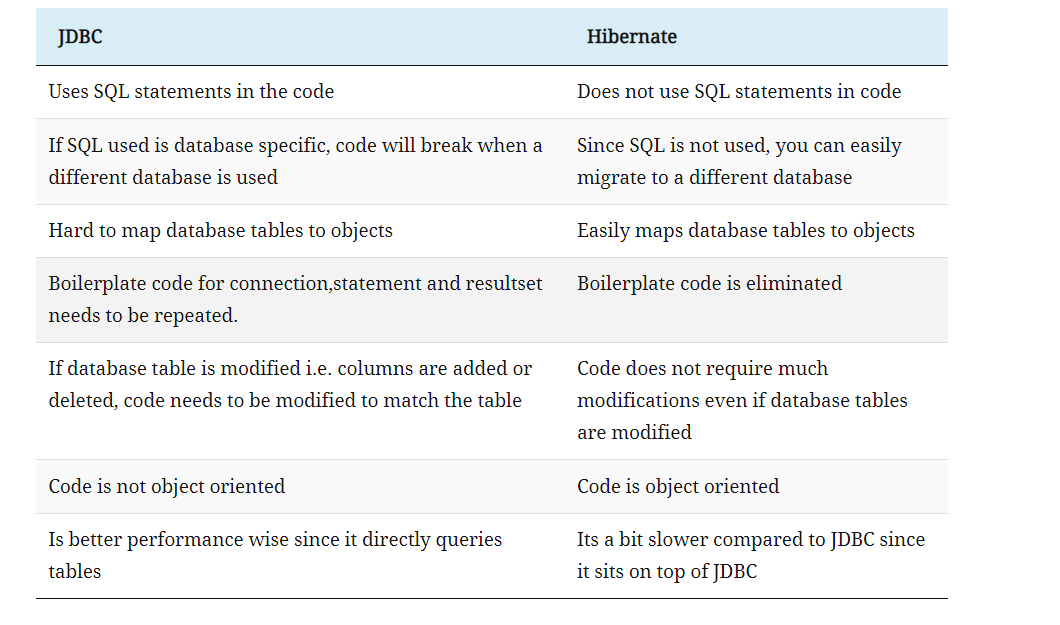
### What are the advantages of Hibernate over JDBC?

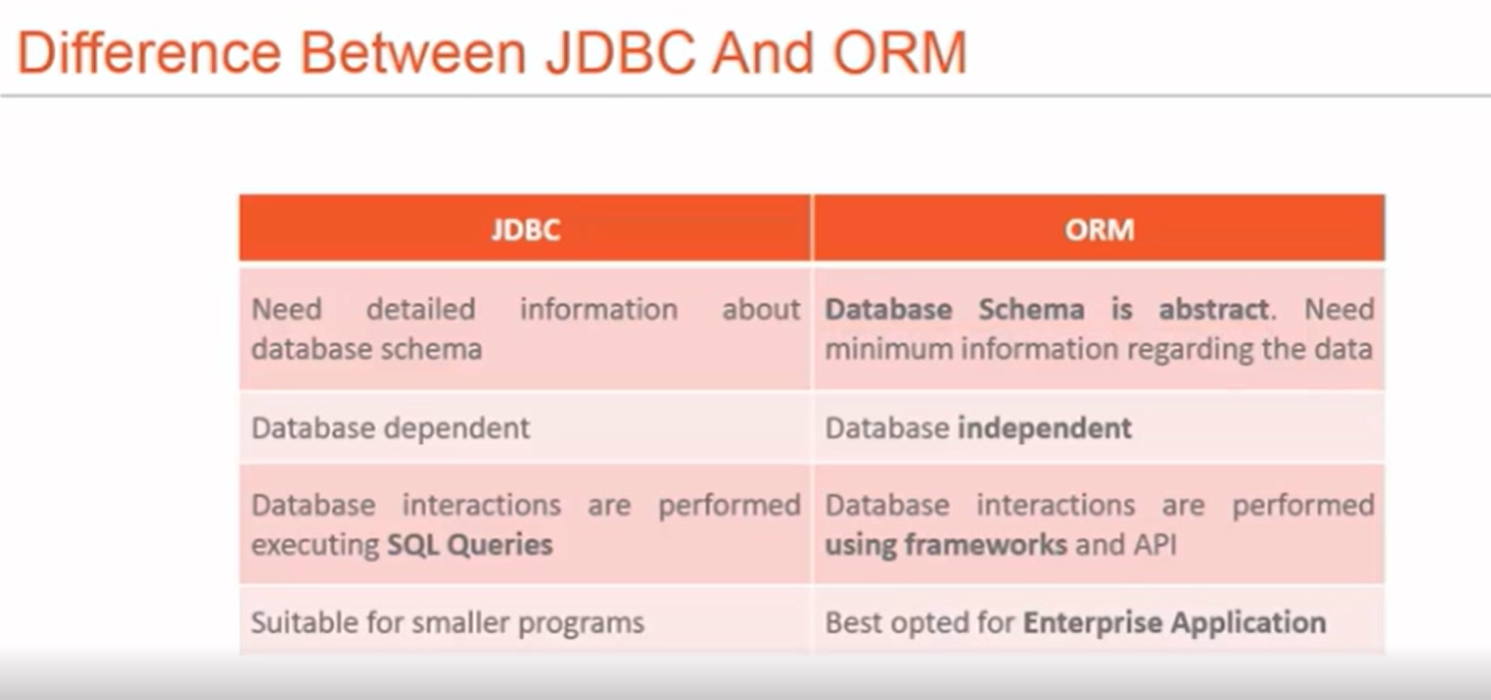
- The advantages of Hibernate over JDBC are listed below:

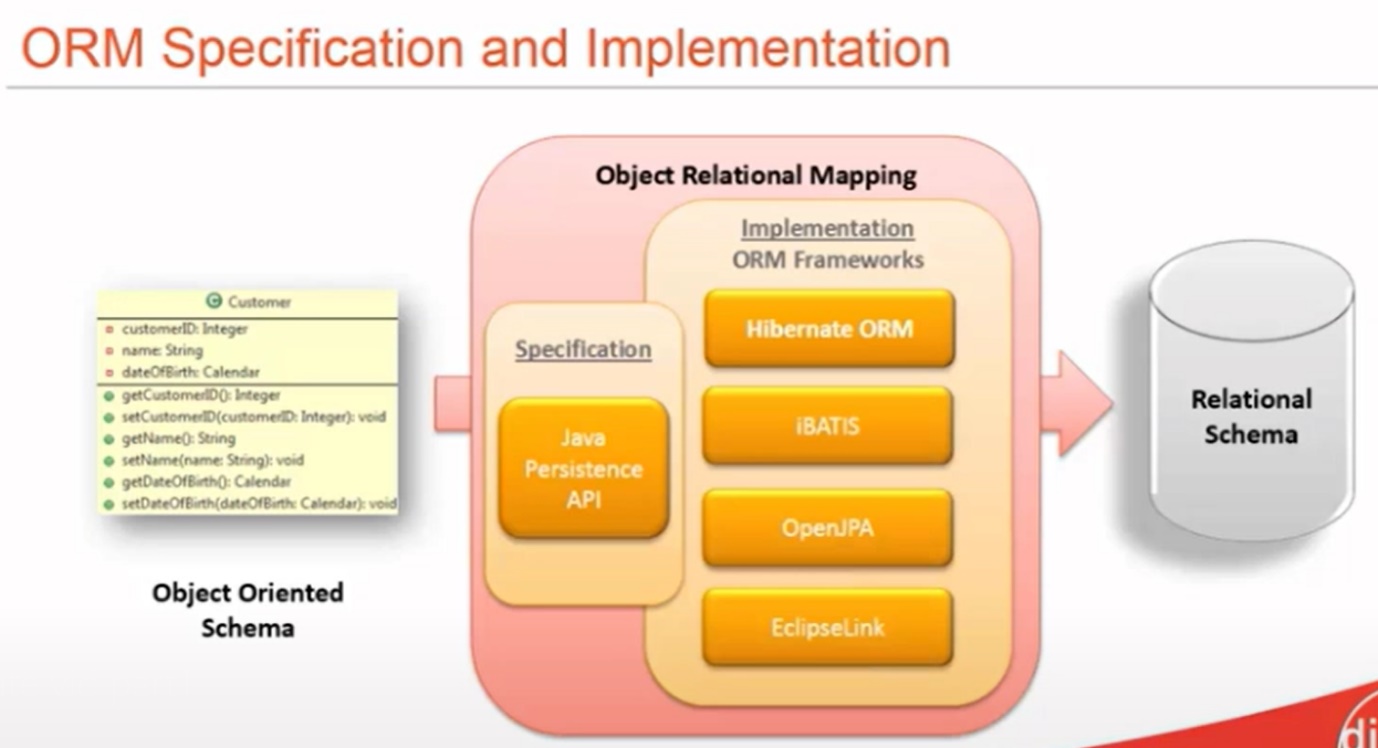
* **Clean Readable Code:** Using hibernate, helps in eliminating a lot of JDBC API-based boiler-plate codes, thereby making the code look cleaner and readable.
* **HQL (Hibernate Query Language):** Hibernate provides HQL which is closer to Java and is object-oriented in nature. This helps in reducing 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



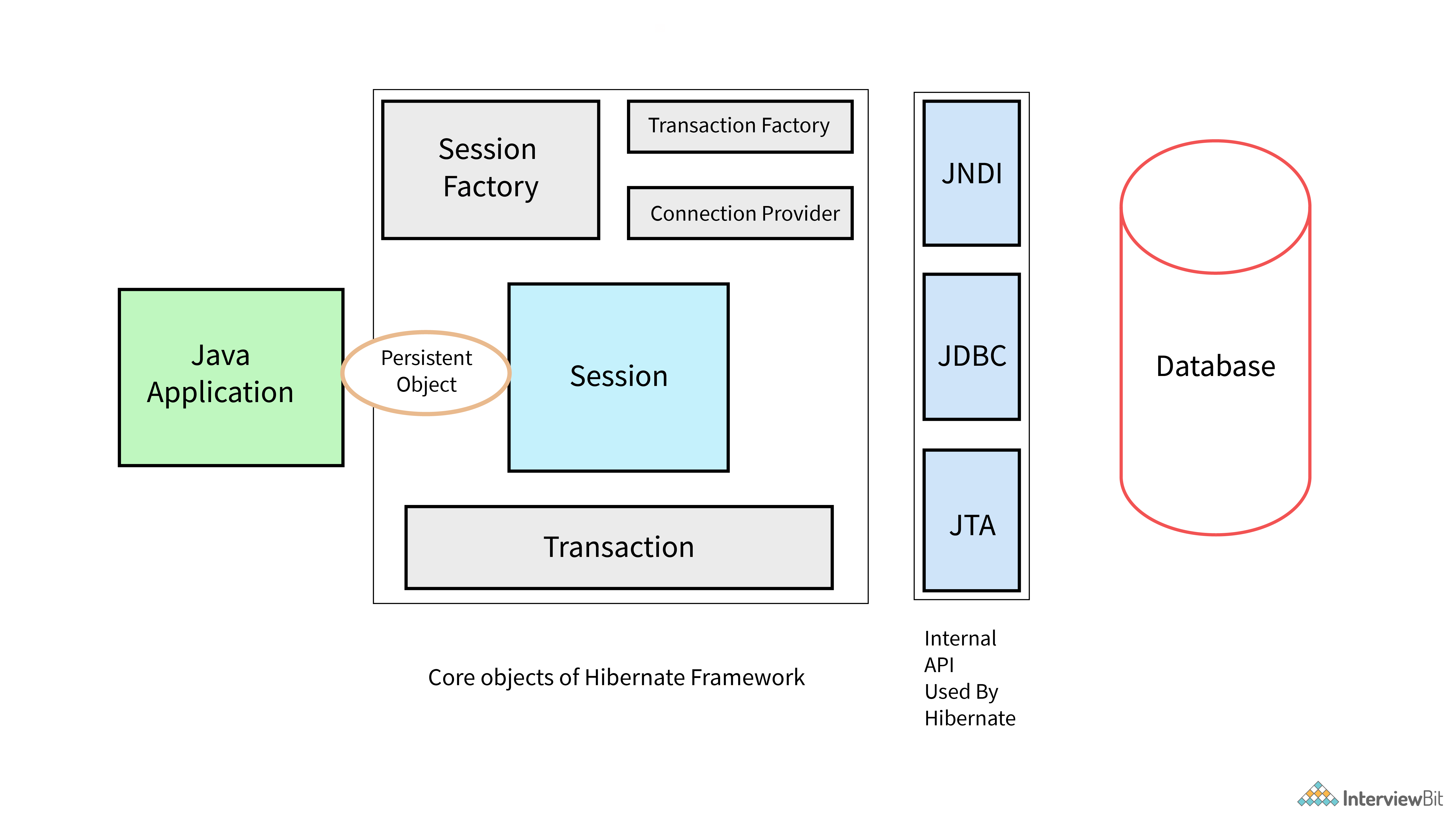




### Explain Hibernate architecture

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**

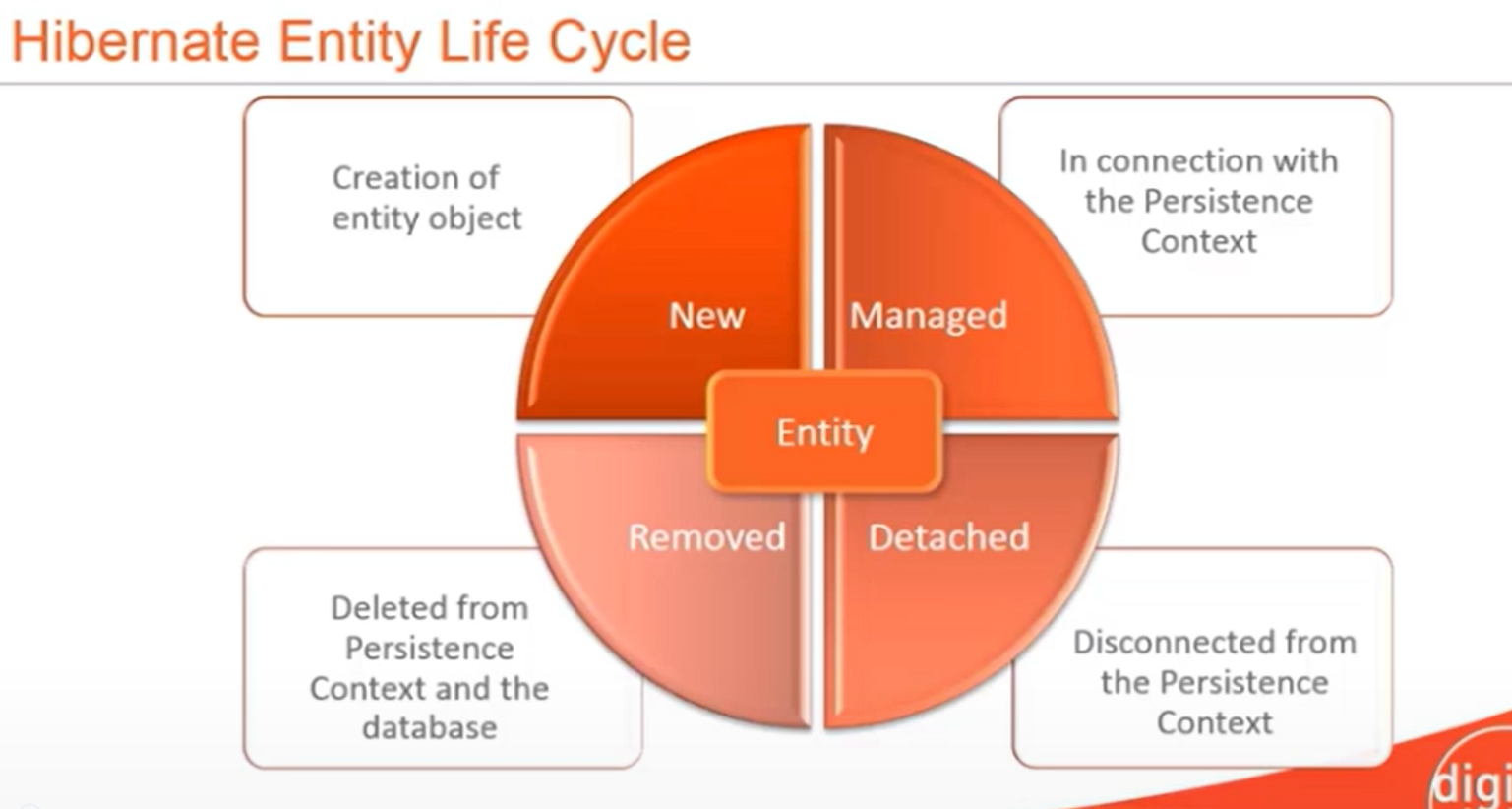
The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

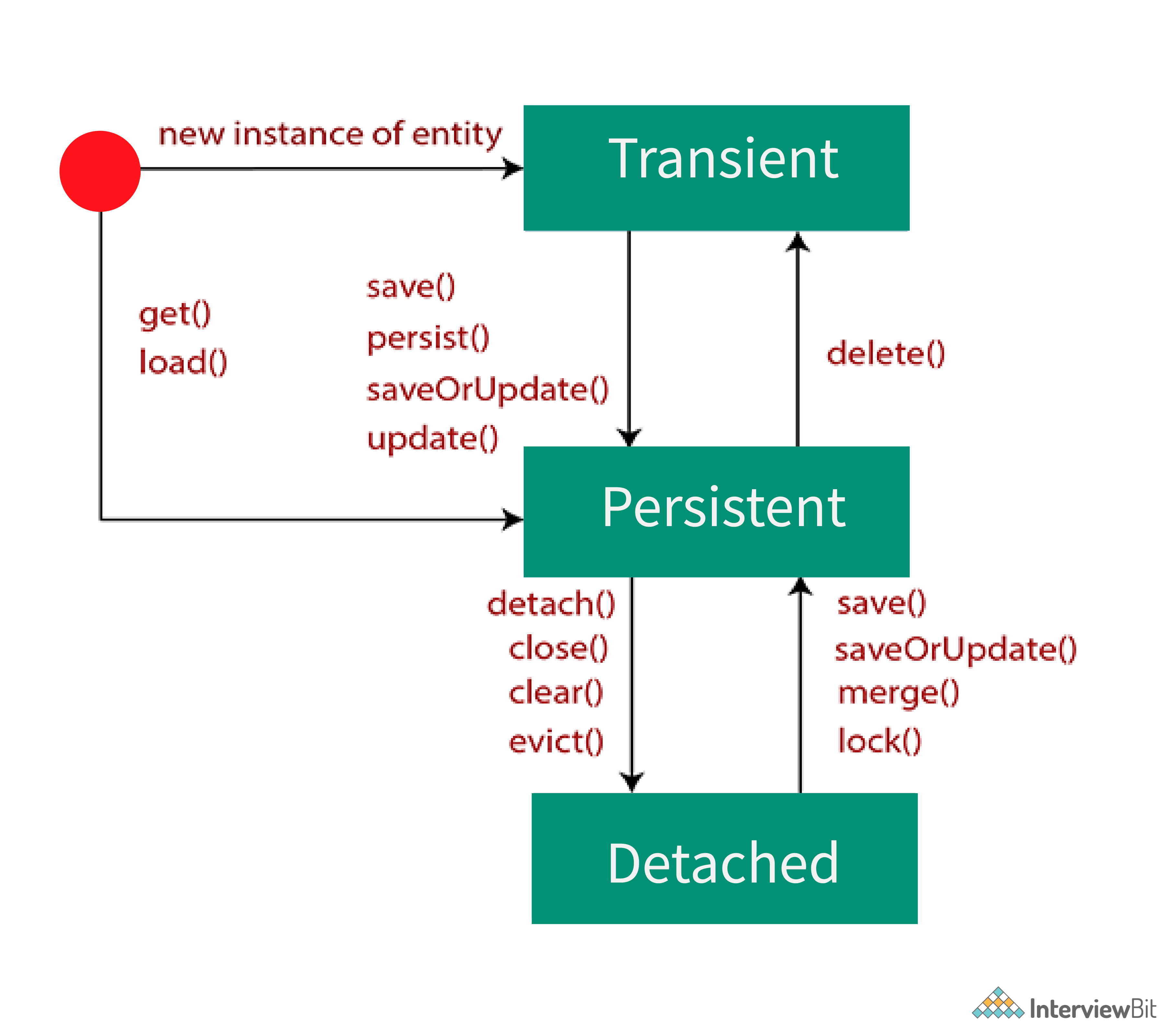
#### **ConnectionProvider**

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**

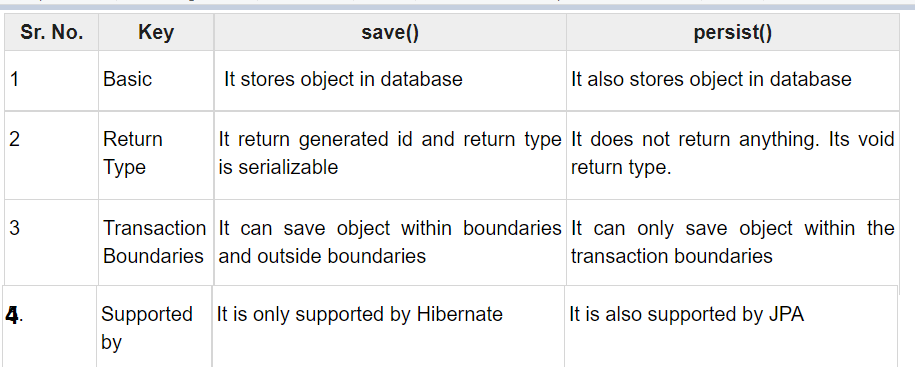
**Hibernate Methods:**

**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

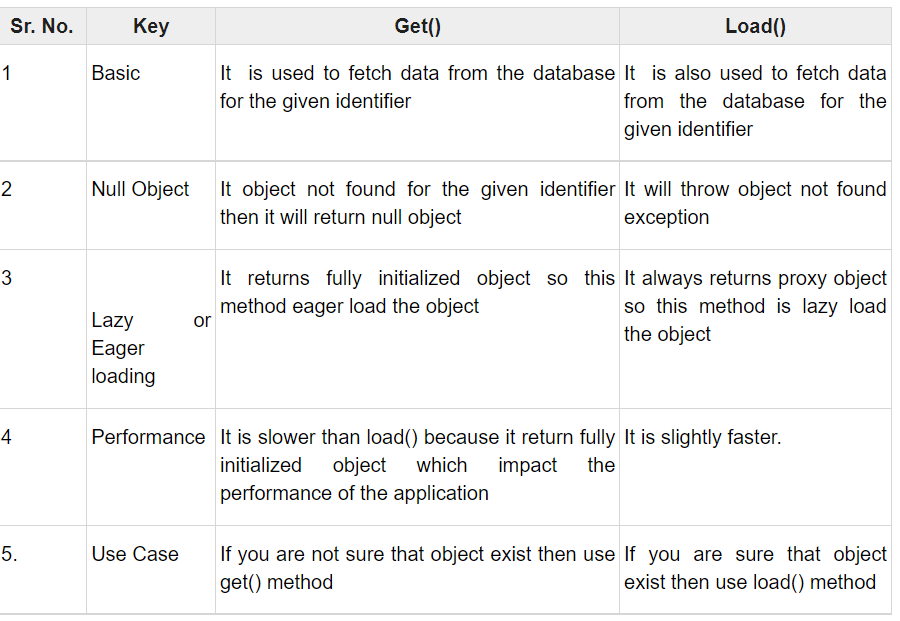
**Load**

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 level 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 lazy loading strategy by default.**

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

**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.LAZY

**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>

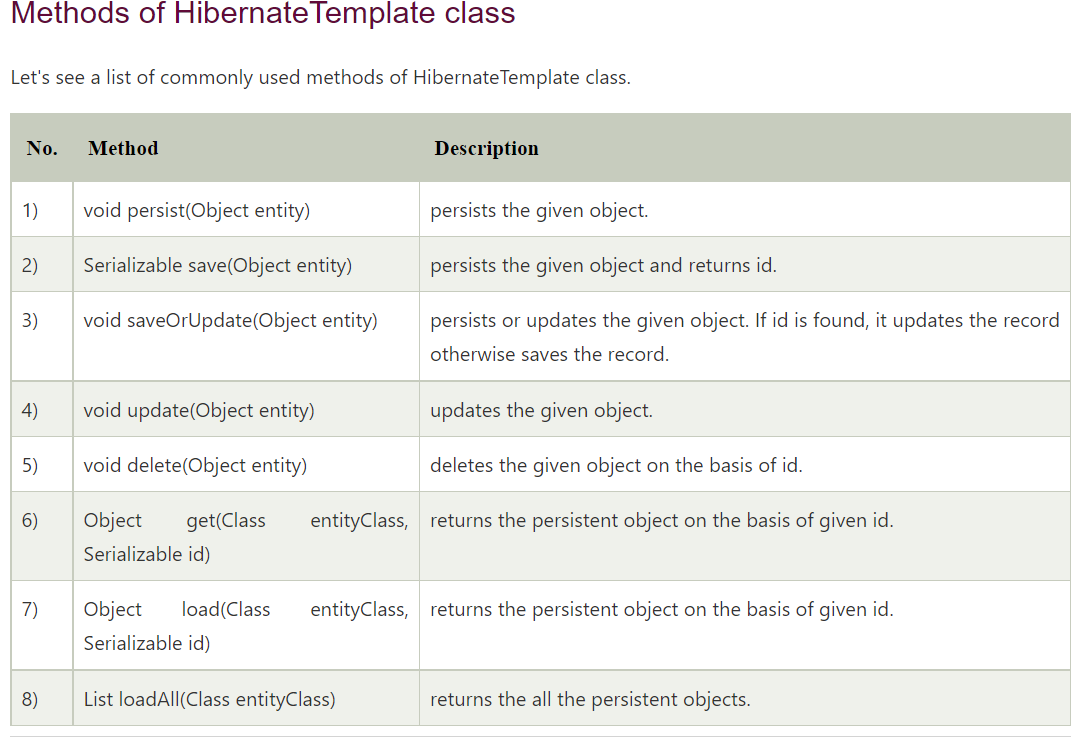
</property>

</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);

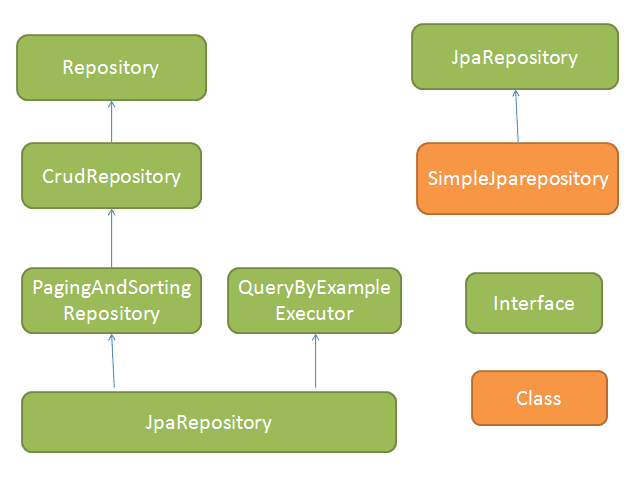
See a complete example of the Named Parameter using Spring Boot and Oracle [here](https://netsurfingzone.com/jpa/spring-data-jpa-named-parameters/).

## **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")

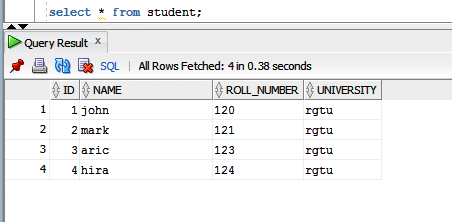
private String rollNumber;

@Column(name = "university")

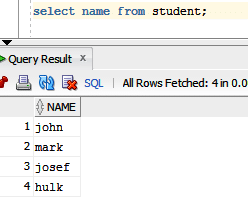
String university;

}

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

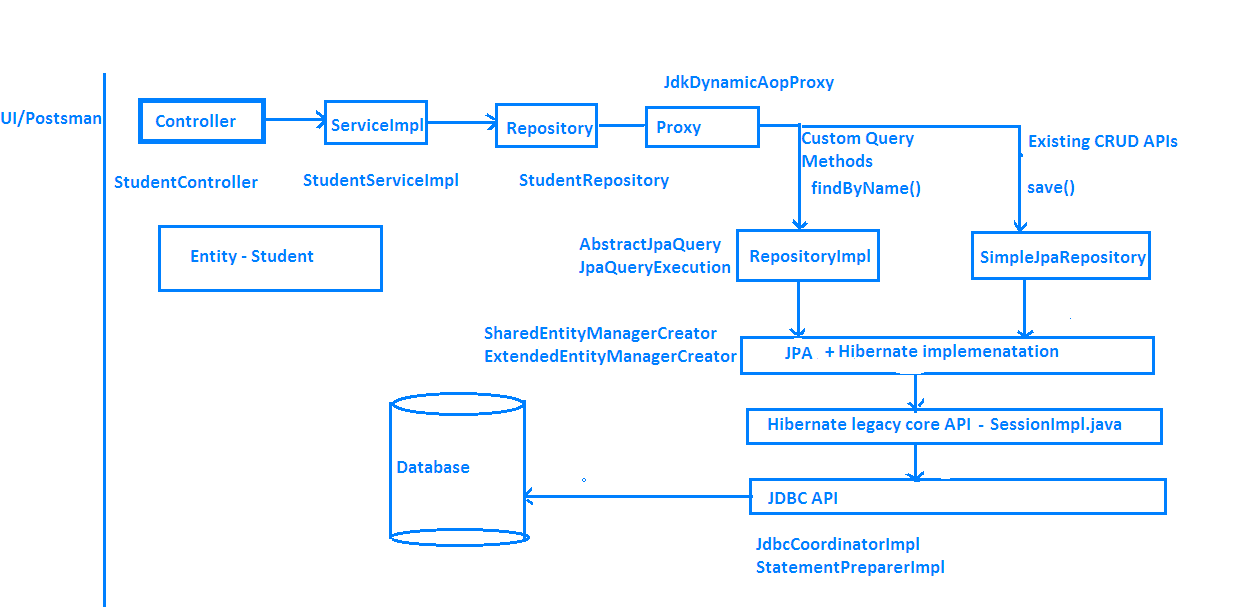
## **Difference between findById() and getOne() in Spring Data JPA?**

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);

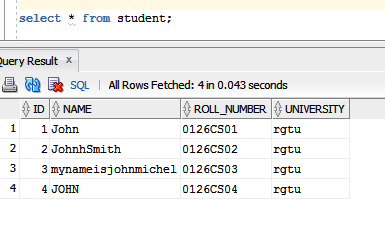
Just think by mistake developer writes findName(). instead of findByName(). In this case, while deployment we will get an error.

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());