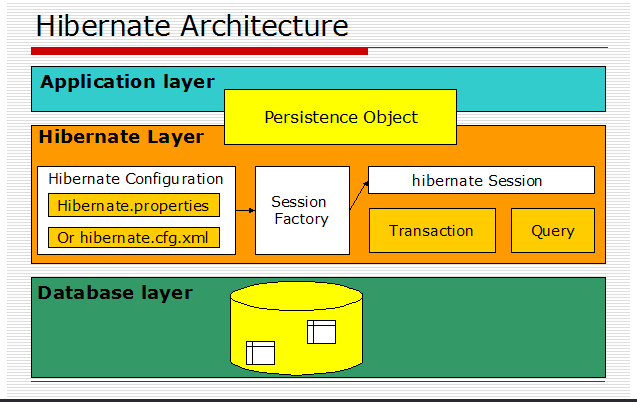
**Why is Hibernate better than JDBC**

Hey Java Developer,

1. Have you ever felt irritated by repeating the same lines of code over and over again in your application for fetching data from a database?
2. Are you struggling to map objects to your database tables?
3. Is it difficult for you to implement oops with your JDBC code?
4. Does it takes too much rework while migrating from one database to another?
5. Have you ever found it difficult to create associations between tables using JDBC?
6. **Relational Persistence**: for JAVA Working with both Object-Oriented software and Relational Database is complicated task with JDBC because there is mismatch between how data is represented in objects versus relational database. So with JDBC, developer has to write code to map an object model's data representation to a relational data model and its corresponding database schema.
7. **HQL** are DB independent.
8. **Performance** because of cache.
9. **Maintenance** is easy no need to modify query’s.
10. **Database Independent** can change data base at any time.
11. **Hibernate** supports inheritance, Collections and relationships.
12. In jdbc all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have Un-checked exceptions, so no need to write try, catch

**Disadvantages of Hibernate:**

1. But there is one major disadvantage, which was boilerplate code issue, but spring eliminated this.
2. Little slower than JDBC.
3. **Poor performance in Batch processing:** It advisable to use pure JDBC for batch processing as Hibernate performance is not good in Batch processing.
4. **Not good for small project:** Small project will have less number of tables, introducing entire Hibernate framework will be **overhead**than useful.
5. **Generates complex quires with many joins :** For complex data, mapping from Object-to-tables and vice versa reduces performance and increases time of conversion.

****

**SessionFactory Object**

Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a **thread safe** object and used by all the threads of an application.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. You would need one SessionFactory object **per database** using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

**Session Object**

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

The session objects should not be kept open for a long time because they are **not thread safe** and they should be created and destroyed them as needed.

Session factory objects are to be implemented using the singleton design pattern. Instances of SessionFactory are thread-safe and typically shared throughout an application. As these objects are heavy weight because they contains the connection information, hibernate configuration information and mapping files,location path. So creating number of instances will make our application heavy weight. But the session objects are not thread safe. So in short it is - SessionFactory objects are one per application and Session objects are one per client.

**session.load()**

1. It will always return a “**proxy**” (Hibernate term) without hitting the database. In Hibernate, proxy is an object with the given identifier value, its properties are not initialized yet, it just look like a temporary fake object.
2. If no row found , it will throws an **ObjectNotFoundException**.

**session.get()**

1. It always hit the database and return the real object, an object that represent the database row, not proxy.
2. If no row found , it return null.

**GenerationType.AUTO**

The *GenerationType.AUTO* is the default generation type and lets the persistence provider choose the generation strategy.

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

**GenerationType.IDENTITY**

The *GenerationType.IDENTITY* is the easiest to use but not the best one from a performance point of view.

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

**GenerationType.SEQUENCE**

The *GenerationType.SEQUENCE* is my preferred way to generate primary key values and uses a database sequence to generate unique values.

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book\_generator")

@SequenceGenerator(name="book\_generator", sequenceName = "book\_seq", allocationSize=50)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

**HQL**

Query query = session.createQuery("from Stock where stockCode = ?");

query.setParameter(1, "7277");

List list = query.list();

Query query=session.createQuery("update Employee set age=:age where name=:name");

query.setInteger("age", 32);

query.setString("name", "Lokesh Gupta");

int modifications=query.executeUpdate();

more <https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/>

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

**Criteria:**

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

List results = cr.list();

**Restrictions with Criteria**

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

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

List results = cr.list();

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

**NamedQuerie:**

@NamedQueries({

@NamedQuery(

name = "findStockByStockCode",

query = "from Stock s where s.stockCode = :stockCode"

)

})

@Entity

@Table(name = "stock", catalog = "mkyong")

public class Stock implements java.io.Serializable {

**NamedNativeQuerie:**

@NamedNativeQueries({

@NamedNativeQuery(

name = "findStockByStockCodeNativeSQL",

query = "select \* from stock s where s.stock\_code = :stockCode",

resultClass = Stock.class

)

})

**Calling:**

Query query = session.getNamedQuery("findStockByStockCode")

.setString("stockCode", "7277");

**Procedure:**

1. **Native SQL**

Query query = session.createSQLQuery(

"CALL GetStocks(:stockCode)")

.addEntity(Stock.class)

.setParameter("stockCode", "7277");

1. **NamedNativeQuery**

@NamedNativeQueries({

@NamedNativeQuery(

name = "callStockStoreProcedure",

query = "CALL GetStocks(:stockCode)",

resultClass = Stock.class

)

})

@Entity

@Table(name = "stock")

public class Stock implements java.io.Serializable {

}

1. **@NamedStoredProcedureQuery annotation**

@NamedStoredProcedureQuery(

  name="GetFoosByName",

  procedureName="GetFoosByName",

  resultClasses = { Foo.class },

  parameters={

    @StoredProcedureParameter(name="fooName", type=String.class, mode=ParameterMode.IN)

  }

)

**One-To-Many Unidirectional:**

class Post {

@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)

@JoinColumn(name = "post\_id")

private List<PostComment> comments = new ArrayList<>();

}

class PostComment {

//not required.

}

**One-To-Many Bidirectional:**

class Post {

@OneToMany(

mappedBy = "post",

cascade = CascadeType.ALL,

orphanRemoval = true )

private List<PostComment> comments = new ArrayList<>();

}

class PostComment {

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "post\_id")

private Post post;

}

**Many-To-One Unidirectional:**

class Post {

//not required

}

// you can use Student and University also.

class PostComment {

@ManyToOne(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)

@JoinColumn(name = "post\_id")

private Post post;

}

**Many-To-One Bidirectional:**

class Post {

@OneToMany(

mappedBy = "post",

cascade = CascadeType.ALL,

orphanRemoval = true)

private List<PostComment> comments = new ArrayList<>();

}

class PostComment {

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "post\_id")

private Post post;

}

**Many-To-Many Unidirectional:**

class Post {

@ManyToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)

@JoinTable(

name = "post\_tag",

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

inverseJoinColumns = { @JoinColumn(name = "tag\_id") } private List<PostComment> comments = new ArrayList<>();

}

class Tag {

//not required.

}

**Many-To-Many Bidirectional:**

class Post {

@ManyToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)

@JoinTable(

name = "post\_tag",

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

inverseJoinColumns = { @JoinColumn(name = "tag\_id") } private List<Tag> tags = new ArrayList<Tag>();

}

class Tag {

@ManyToMany(mappedBy="tags")

private List<Post> post = new ArrayList<Post>();

}

**Why we should not make Entity Class final?**

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

**Which design patterns are used in Hibernate framework?**

Some of the design patterns used in Hibernate Framework are:

* Domain Model Pattern – An object model of the domain that incorporates both behaviour and data.
* Data Mapper – A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.
* [Proxy Pattern](http://www.journaldev.com/1572/proxy-design-pattern-in-java-example-tutorial) for lazy loading
* [Factory pattern](http://www.journaldev.com/1392/factory-design-pattern-in-java) in SessionFactory
* Singleton

**What is difference between sorted and ordered collection in hibernate?**

This is one of the easy Hibernate interview question you ever face. A sorted collection is sorted in memory by using [Java Comparator](http://java67.blogspot.com/2012/10/how-to-sort-object-in-java-comparator-comparable-example.html), while a ordered collection uses database's order by clause for ordering. For large data set it's better to use ordered collection to avoid any [OutOfMemoryError in Java](http://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html), by trying to sort them in memory.

**What is query cache in Hibernate?**

This question, Some times asked as a follow-up of last Hibernate Interview question, QueryCache actually stores result of sql query for future calls. Query cache can be used along with second level cache for improved performance. Hibernate support various open source caching solution to implement Query cache e.g. EhCache.

**Can we make an Hibernate Entity Class final?**

Yes, you can make an Hibernate Entity class final, but that's not a good practice. Since Hibernate uses proxy pattern for performance improvement in case of lazy association, by making an entity final, Hibernate will **no longer be able to use proxy**, because [Java doesn't allow extension of final class](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html), thus limiting your performance improvement options. Though, you can avoid this penalty, if your persistent class is an implementation of interface, which declares all public methods defined in Entity class.

**When to use HQL or Criteria Queries**

There are lots of different reasons and a ton of opinions on which is better HQL or Criteria Queries. Assuming that you are using **Named Queries** for your HQL performance is similar in both. Here is a list of why you may prefer one over the other.

* Criteria queries are ideal for dynamic queries. It is very simple to add restrictions and ordering as well as pagination.
* HQL is ideal for static queries especially if you are using named queries as they underlying SQL is generated when your session factory starts.
* JPA does not support Criteria queries, so if you need portability use HQL queries.
* There is a difference in terms of performance between HQL and criteriaQuery, every time you fire a query using a criteria query, it creates a new alias for the table name which does not reflect in the last queried cache for any DB. This leads to an overhead of compiling the generated SQL, taking more time to execute, but not as much as you may think.
* **HQL** can perform both select and non-select operations. **Criteria** can only select data, you can’t perform non-select operations using criteria queries.
* **HQL** does not support pagination, but pagination can be achieved with **Criteria.**
* **Criteria** is safe from SQL injection. HQL isvulnerableto SQL injection as your queries are either fixed or parametrized.

#### **Relative Paths**

* index.html
* /graphics/image.png
* /help/articles/how-do-i-set-up-a-webpage.html

#### **Absolute Paths**

* http://www.mysite.com
* http://www.mysite.com/graphics/image.png
* <http://www.mysite.com/help/articles/how-do-i-set-up-a-webpage.html>

**What are the various transaction isolation issues in Hibernate?**

The ANSI SQL standard defines the standard transaction isolation levels in terms of which of these phenomenon are permissible:

\* Last update

\* Dirty read

\* Unrepeatable read

\* Second lost updates problem

\* Phantom read

**Explain the Phantom read?**

A transaction executes a query twice, and the second result set includes rows that weren't visible in the first result set. This situation is caused by another transaction inserting new rows between the execution of the two queries.

**What is dirty read in transaction isolation issues?**

When one transaction reads changes made by another transaction that hasn't yet been committed is called dirty read. This is very dangerous, because those changes might later be rolled back.

**What is Hibernate Templa te?**

Ans: The spring framework provides HibernateTemplate (org.springframework.orm.hibernate.HibernateTemplate) which is kind of helper class and provides following benefits.

* Hibernate Template class simplifies interaction with Hibernate session.
* Common functions are simplified to single method calls.
* Sessions are automatically closed.
* Exception are automatically caught and converted to runtime exceptions.

**What does Session lock() method do in Hibernate?**

This one is one of the tricky Hibernate Interview question, because Session's lock() method reattach object without synchronizing or updating with database. So you need to be very careful while using lock() method. By the way you can always use Session's update() method to sync with database during reattachment. Some time this Hibernate question is also asked as *what is difference between Session's lock() and update() method*. You can use this key point to answer that question as well.

**Define cascade and inverse option in one-many mapping?**

cascade - enable operations to cascade to child entities.

cascade="all|none|save-update|delete|all-delete-orphan"  
inverse - mark this collection as the "inverse" end of a bidirectional association.  
inverse="true|false"   
Essentially "inverse" indicates which end of a relationship should be ignored, so when persisting a parent who has a collection of children, should you ask the parent for its list of children, or ask the children who the parents are?

**How can a whole class be mapped as immutable?**

Mark the class as mutable="false" (Default is true),. This specifies that instances of the class are (not) mutable. Immutable classes, may not be updated or deleted by the application.