Problems with Using JDBC

Using JDBC means you can execute any kind of SQL statements. For a simple task, you have to code many **SELECT**, **INSERT**, **UPDATE**, and **DELETE** statements repeatedly. This results in the following issues:

• *Too much copy code:* When you perform object retrieval, you need to copy the fields in a **ResultSet** to the properties of an object. When you perform object persistence, you need to copy the properties of an object to the parameters in **PreparedStatement**.

• *Manually handled associations:* When you perform object retrieval, you have to perform a table join or read from several tables to get an object graph. When you perform object persistence, you must update several tables accordingly.

• *Database dependent:* The SQL statements you wrote for one database may not work with another brand of database. Although the chance is very small, you may have to migrate to another database.

Hibernate usually uses javassist

```
<!-- https://mvnrepository.com/artifact/org.javassist/javassist -->
<dependency>
   <groupId>org.javassist</groupId>
   <artifactId>javassist</artifactId>
   <version>3.22.0-CR1</version>
</dependency>
```

Javassist makes Java bytecode manipulation simple. It is a class library for editing bytecodes in Java.

A common application of javassist is to generate proxy classes at runtime.

Lazy loading is loading objects from the database upon first access. Hibernate uses proxies to intercept method invocation on entities to implement lazy loading.

Spring framework uses proxies to implement its support for AOP, which among other things powers its support for declarative transactions.

EJB uses proxies to implement container managed transactions, authorization checking and to apply user defined interceptors.

Configuration configuration = new Configuration().configure();

Configure() method loads the hibernate.cfg.xml from the root classpath. The new Configuration() loads the hibernate.properties file, and the configure() method loads the hibernate.cfg.xml if hibernate.properties is not found. If you need to load another configuration file located elsewhere use the code

Configuration configuration = new Configuration().configure("/config/recipes.cfg.xml");

This code looks for recipes.cfg.xml in the config directory of your classpath.

Hibernate session objects represents a unit of work and is bound to the current thread. It represents a transaction in the database. A session begins when getCurrentSession() is first called on the current thread object. The session is then bound to the current thread. When the transaction ends with a commit or a rollback, hibernate unbounds the session from the current thread and closes it.

Book book = (Book) session.load(Book.class, isbn);
and
Book book = (Book) session.get(Book.class, isbn);
What's the difference between load() and get()?
First, when the given ID can't be found, load() throws an exception org.hibernate.ObjectNotFoundException, whereas get() returns a null object. Second, load() just returns a proxy by default; the database isn't hit until the proxy is first invoked.
get() hits the database immediately.
The load method is useful when you only need a proxy and don't need to make a database call. You just need a proxy, when in a given session you need to associate an entity before persisting.

@Entity is defined in EJB 3.0 specification to annotate an entity bean. An entity represents a lightweight persistent domain object.
An entity class must have a public or protected no-arg constructor. It may have other constructors as well.
It should be a top level class.
It must not be final.
If the entity is to passed by value (that is through a remote interface) it must implement Serializable.
The state of the entity is represented by entity's instance variables. They must be accessed only from within the class. The client of the entity should not be able to access the state of the entity directly. The instance variables must have private, protected or package visibility.
Every entity must have a primary key, which must be declared only once in the entity hierarchy.

Static properties are not persisted by the EntityManager when you map persistent properties.

```
<hibernate-mapping
         schema="schemaName"                                    1
         catalog="catalogName"                                  2
         default-cascade="cascade_style"                        3
         default-access="field|property|ClassName"              4
         default-lazy="true|false"                              5
         auto-import="true|false"                               6
         package="package.name"                                 7
 />
```

1. `schema` (optional): the name of a database schema.
2. `catalog` (optional): the name of a database catalog.
3. `default-cascade` (optional - defaults to `none`): a default cascade style.
4. `default-access` (optional - defaults to `property`): the strategy Hibernate should use for accessing all properties. It can be a custom implementation of `PropertyAccessor`.
5. `default-lazy` (optional - defaults to `true`): the default value for unspecified `lazy` attributes of class and collection mappings.
6. `auto-import` (optional - defaults to `true`): specifies whether we can use unqualified class names of classes in this mapping in the query language.
7. `package` (optional): specifies a package prefix to use for unqualified class names in the

mapping document.

```
<class
        name="ClassName"                                    (1)
        table="tableName"                                   (2)
        discriminator-value="discriminator_value"           (3)
        mutable="true|false"                                (4)
        schema="owner"                                      (5)
        catalog="catalog"                                   (6)
        proxy="ProxyInterface"                              (7)
        dynamic-update="true|false"                         (8)
        dynamic-insert="true|false"                         (9)
        select-before-update="true|false"                   (10)
        polymorphism="implicit|explicit"                    (11)
        where="arbitrary sql where condition"               (12)
        persister="PersisterClass"                          (13)
        batch-size="N"                                      (14)
        optimistic-lock="none|version|dirty|all"            (15)
        lazy="true|false"                                   (16)
        entity-name="EntityName"                            (17)
        check="arbitrary sql check condition"               (18)
        rowid="rowid"                                       (19)
        subselect="SQL expression"                          (20)
        abstract="true|false"                               (21)
        node="element-name"
/>
```

(1) `name` (optional): the fully qualified Java class name of the persistent class or interface. If this attribute is missing, it is assumed that the mapping is for a non-POJO entity.

(2) `table` (optional - defaults to the unqualified class name): the name of its database table.

(3) `discriminator-value` (optional - defaults to the class name): a value that distinguishes individual subclasses that is used for polymorphic behavior. Acceptable values include `null` and `not null`.

(4) `mutable` (optional - defaults to `true`): specifies that instances of the class are (not) mutable.

(5) `schema` (optional): overrides the schema name specified by the root `<hibernate-mapping>` element.

(6) `catalog` (optional): overrides the catalog name specified by the root `<hibernate-mapping>` element.

(7) `proxy` (optional): specifies an interface to use for lazy initializing proxies. You can specify the name of the class itself.

(8) `dynamic-update` (optional - defaults to `false`): specifies that `UPDATE` SQL should be generated at runtime and can contain only those columns whose values have changed.

(9) `dynamic-insert` (optional - defaults to `false`): specifies that `INSERT` SQL should be generated at runtime and contain only the columns whose values are not null.

(10) `select-before-update` (optional - defaults to `false`): specifies that Hibernate should

never perform an SQL `UPDATE` unless it is certain that an object is actually modified. Only when a transient object has been associated with a new session using `update()`, will Hibernate perform an extra SQL `SELECT` to determine if an `UPDATE` is actually required.

11. `polymorphism` (optional - defaults to `implicit`): determines whether implicit or explicit query polymorphism is used.
12. `where` (optional): specifies an arbitrary SQL `WHERE` condition to be used when retrieving objects of this class.
13. `persister` (optional): specifies a custom `ClassPersister`.
14. `batch-size` (optional - defaults to `1`): specifies a "batch size" for fetching instances of this class by identifier.
15. `optimistic-lock` (optional - defaults to `version`): determines the optimistic locking strategy.

(16) `lazy` (optional): lazy fetching can be disabled by setting `lazy="false"`.

(17) `entity-name` (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See Section 4.4, "Dynamic models" and Chapter 18, *XML Mapping* for more information.

(18) `check` (optional): an SQL expression used to generate a multi-row *check* constraint for automatic schema generation.

(19) `rowid` (optional): Hibernate can use ROWIDs on databases. On Oracle, for example, Hibernate can use the `rowid` extra column for fast updates once this option has been set to `rowid`. A ROWID is an implementation detail and represents the physical location of a stored tuple.

(20) `subselect` (optional): maps an immutable and read-only entity to a database subselect. This is useful if you want to have a view instead of a base table. See below for more information.

(21) `abstract` (optional): is used to mark abstract superclasses in `<union-subclass>` hierarchies.

https://stackoverflow.com/questions/5640778/hibernate-sessionfactory-vs-entitymanagerfactory

Generators in hibernate

1. Assigned – default, programmers responsibility to generate the primary key, supported by all databases
2. Increment – supported in all databases and database independent. Formula used is maximum id in the primary column + 1. If there is no record initially in the database then, first value is 1.
3. Sequence – database dependent. Does not work with MySQL. Next value is assigned from the sequence. Sequence must be created in the database in advanced. Sequence name must be passed inside generator tag. If sequence does not exist then hibernate will create its own

sequence called hibernate_sequence. But if you want hibernate to generate its own sequence them hbm2ddl.auto must be enabled.

4. Identity – database dependent. Does not work with oracle.

- *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*: there is also an undocumented value none to disable it entirely.