**Hibernate**

Java has a standard tool set for accessing databases. It is called the Java Database Connectivity (JDBC) application programming interface (API). The API was very well used in Java applications until recently. While the API is well suited for small projects, it becomes quite cumbersome (and sometimes out of hand) as the domain model increases in complexity. The API also includes a lot of repetitious boilerplate code, requiring the developer to do a lot of manual coding. Furthermore, handling of the object-relational model mapping is heavy-handed too!

The standard steps to follow in creating a Hibernate application are:

1. Configure the database connection.

2. Create mapping definitions.

3. Persist the classes.

There are two pieces of configuration required in any Hibernate application: one creates the database connections, and the other creates the object-to-table mapping.

**Session Factory:-**

The SessionFactory, represented by the org.hibernate.SessionFactory class, is a factory class for churning out our Session instances. It is a thread-safe object and hence can be shared across various classes without our having to worry about the data being corrupted.

The SessionFactory also maintains a second level of cache, which is available globally across all other components in the application. Global cache is used if multiple applications require the same data that’s been loaded from the database. This will speed up the application’s request times.

**Session:-**

Session is a single-threaded object and therefore should not be shared across various components. It represents a single unit of work.

We use*factory.getOpenSession()*to fetch a session from the factory.

Note that the first level of cache is maintained in the session; (i.e., all the objects that were fetched or accessed will be held in the session until the session is closed).

We should not use the session per individual operations. Ideally, all the related database operations should be grouped under one transaction.

**Transaction:-**

In simple terms, transactions keep our work segregated from others, and synchronize with the durable storage to avoid incorrect data being read or written.

There are four fundamental properties that database transactions revolve around: atomicity, consistency, isolation, and durability, which are collectively known as ACID properties.

Typically, there are two modes of fetching transactions: a container can create and manage your work in its transaction, or we can create our own transaction.

session.beginTrasaction()

We initiate a transaction by invoking the *session.beginTrasaction()* method, which creates a new Transaction object and returns the reference to us. It gets associated with the session and is open until that transaction is committed or rolled back.

session.getTransaction().commit();

session.getTransaction().rollback();

We perform the required work in a transaction, and then issue a commit on this transaction. At this stage, the entities are persisted to the database. While persisting, if for whatever reason there are any errors, the Hibernate runtime will catch and throw a HibernateException (which is an unchecked RuntimeException). We then have to catch the exception and roll back the transaction.

**Annotations: @**

* Annotations can be kept above variable declaration or getter method or constructor.

**@Entity & @Table – javax.persistence.Entity / javax.persistence.Table**

To make this class persistent using annotations, we first define it as an Entity. We do so by annotating the class with the @Entity annotation.

If the name of the table is the same as that of the class then declaring only @Entity is sufficient.

However, if table name is different; then we need to let Hibernate know that by adding a**@Table** annotation too.

@Entity

@Table(name = "TBL\_EMPLOYEE")

public class Employee {

...

}

Note- @Entity also has name attribute and does the same as @Table’s name attribute but that name will affect while using HQL query. So caution while using Entity’s name attribute

**@Id –javax.persistence.Id**

Now that we have a persistent entity, our next step is to define an identifier. Remember, all persistent entities must have their identifiers defined; otherwise, Hibernate will moan and groan. So, we use the @Id annotation to indicate the object’s identifier to the framework.

One more thing to note: most databases will not allow us to create a field called ID, as it is a reserved keyword. If the object field and the column name do not match, we need to set the column name explicitly using an additional annotation, @Column.

@Entity

public class Employee {

@Id

@Column(name="EMPLOYEE\_ID")

privateint id =0;

...

**@Column-javax.persistence.Column**

If your variable doesn’t match the column name, you must specify the column name using the @Column annotation.

If the column accepts non-null data, we fulfill this option by setting nullable=false. Or we specify unique=true if the column should be generated with a unique constraint.

@Entity

@Table(name = "TBL\_EMPLOYEE")

public class Employee {

@Id

@Column(name="EMPLOYEE\_ID", nullable = false, unique = true)

privateintempoyeeId = 0;

...

**@Basic - javax.persistence.Basic**

By default the fields will be basic but if you need to configure few attributes of the @Basic annotation (couple of Basic attributes). Otherwise, no need to use this annotation.

**@Transient – javax.persistence.Transient**

The field would not be persisted or it will be ignored by Hibernate. Another way is to mark the field as static field.

**@Temporal – javax.persistence.Temporal**

This annotation can only be used for date type field. If you need to save only date or only time or both, this property can be used. There are few javax.persistence.TemporalType that could be saved.

[@Temporal(TemporalType.Date)](mailto:p@Temporal(TemporalType.Date))

privateDatetoday;

**@Lob – javax.persistence.Lob**

@Lob can be used for large fields which above 256 characters. If @Lob is above a String field then a CLOB type would be created in database and if the @Lob is above byte array then in database BLOB type would be created.

**@Embeddable or @Embedded – javax.persistence.Embedded or Embedable**

@Embeddable annotation should be used on dependent class file, while @Embedded should be used above variable declaration. If dependent class is annotated with @Embeddable then @Embedded is not required.

**@AttributeOverrides and @AttributeOverride – javax.persistence.AttributeOverrides**

AttributeOverride annotation can be used to rename the column name of embedded class.

Eg.

@Embedded

@AttributeOverrides({

@AttributeOverride(name”street”, column=@Column(name=”HOME\_STREET”)),

@AttributeOverride(name”city”, column=@Column(name=”HOME\_CITY”)),

@AttributeOverride(name”state”, column=@Column(name=”HOME\_STATE”))})

Private Address address;

**@EmbeddedId – javax.persistence.EmbeddedId**

When the primary key is another dependent object with @Embeddable annotation, @Id is not allowed on complex type so have to use @EmbeddedId.

@EmbeddedId

Private LoginName user;

**@ElementCollection – javax.persistence.ElementCollection**

@ElementCollection can be used to save collections of object when associations are not referenced. It creates a separate table to save the collections. The dependent class in the collection should be defined as @Embeddable.

@ElementCollection

Private set<Address> addresses = new HashSet<Address>();

**@JoinTable – javax.persistence.JoinTable**

@JoinTable annotation can be used to create a separate table of the dependent object. It has an attribute, *name* which sets the name of the table. Similarly, it has attribute joinColumns which helps to set the joined column name which is the primary key of the class.

@JoinTable also has inverseJoinColumns not need here but will be used in association, to set name of the column of primary key of dependent object.

Eg.

@ElementCollection

@JoinTable(name=”USER\_ADDRESS”,

joinColumns=@JoinColumn(name=”USER\_ID”)}

private Collection<Address>listofAddresses = new ArrayList();

**ID Generation Strategies –**

Hibernate provides various identifier generation strategies using annotations. If the ID generation strategy is not set, then by default it is an AUTO strategy. When the strategy is AUTO, Hibernate relies on our database to generate primary keys.

We can set up different strategies depending on our requirements. All we need to do is add @GeneratedValue annotation to the id variable. This annotation accepts two attributes: strategy and generator. The strategy attribute indicates the type of identifier generation that we would like to use, while generator defines the methods to generate the identifiers.

@Entity(name = "TBL\_EMPLOYEE")

public class Employee {

@Id

@Column(name="ID")

@GeneratedValue(strategy = GenerationType.IDENTITY)

privateintemployeeId =0;

...

}

The strategy should be the GeneratorType value, as described in the following:

GeneratorType.AUTO

This is the default strategy and is portable across different databases. Hibernate chooses the appropriate ID based on the database.

GeneratorType.IDENTITY

This setting is based on the identity provided by some databases; it is the responsibility of the database to provide a unique identifier. The identity function is supported by a few databases, including MySQL, Sybase, and DB2.

GeneratorType.SEQUENCE

Some databases provide a mechanism of sequenced numbers, so this setting will let Hibernate use the sequence number.

GeneratorType.TABLE

Sometimes the primary keys have been created from a unique column in another table. In this case, use the TABLE generator.

For employing a sequence strategy, you must define both the strategy and the sequence generator:

public class Employee {

@Id

@Column(name="EMPLOYEE\_ID")

@GeneratedValue (strategy= GenerationType.SEQUENCE, generator="empSeqGen")

@SequenceGenerator(name = "empSeqGen", sequenceName = "EMP\_SEQ\_GEN")

privateintemployeeId =0;

...

}

The strategy is defined as a SEQUENCE, and accordingly the generator is given a reference to a sequence generator, empSeqGen, which refers to a sequence object in the database. Using the @SequenceGenerator, we reference EMP\_SEQ\_GEN, which is a sequence object created in the database.

As with sequences, we can use an @TableGenerator strategy if we have a database table that provides our primary keys. The following snippet demonstrates this:

public class Employee {

@Id

@Column(name="ID")

@GeneratedValue (strategy= GenerationType.TABLE, generator="empTableGen")

@TableGenerator(name = "empTableGen", table = "EMP\_ID\_TABLE")

privateintempoyeeId =0;

...

}

**Composite Identifiers –**

We don’t always have a single column (surrogate key) as our primary key identifying the unique row. Sometimes we have a combination of columns providing a business key, commonly called a composite or compound key. In this case, we need to use a different mechanism for setting the appropriate object identifier.

There are three ways of setting the composite-id identifiers –

1. **Using Primary Key Class and @Id**

We create a separate class representing the business key. We annotate this class with @Embeddable, making it a composite-id class.

In the following example, the CoursePK consists of two variables, tutor and title. The combination of these two attributes makes a composite key for Course class.

@Embeddable

public class CoursePK implements Serializable{

private String tutor = null;

private String title = null;

// Default constructor

publicCoursePK() {

}

...

}

Notice that the class has been decorated with the @Embeddable annotation.

*Important* - Make sure the class implements the java.io.Serializable interface with a default constructor. Also, it must have the hashCode and equals methods implemented; they will help Hibernate distinguish uniqueness.

The next step is to embed this class on our persistent class’s id variable. We do this by using our simple @Id annotation:

@Entity

@Table(name="COURSE\_ANNOTATION")

public class Course {

@Id

privateCoursePK id = null;

privateinttotalStudents = 0;

privateintregisteredStudents = 0;

publicCoursePKgetId() {

return id;

}

public void setId(CoursePK id) {

this.id = id;

}

...

}

The id is a CoursePK type—a composite class with two variables, tutor and title—making up the primary class for our Course object.

private void persist() {

...

Course course = new Course();

CoursePKcoursePk = new CoursePK();

coursePk.setTitle("Computer Science");

coursePk.setTutor("Prof. Harry Barry");

course.setId(coursePk);

course.setTotalStudents(20);

course.setRegisteredStudents(12);

session.save(course);

...

}

1. **Using Primary Key Class and @EmbeddedId**

We annotate the identifier of the Course object with @EmbeddedId (instead of annotating with @Id as we did in our earlier case).

@Entity

@Table(name = "COURSE\_ANNOTATION\_V2")

public class Course2 {

@EmbeddedId

private CoursePK2 id = null;

privateinttotalStudents = 0;

privateintregisteredStudents = 0;

public Course2(String title, String tutor) {

id = new CoursePK2();

id.setTitle(title);

id.setTutor(tutor);

}

...

}

The id field is annotated with @EmbeddedId in the preceding class. Notice that the constructor is doing the job of creating and populating the composite primary key. You can do this outside of the constructor too, if you wish.

However, we do not have to annotate @Embeddable on the primary key class, as we did in the first method.

public class CoursePK2 implements Serializable {

private String tutor = null;

private String title = null;

// Default constructor

public CoursePK2() { }

// Implement hashCode and equals methods

@Override

publicinthashCode() { ... }

@Override

publicboolean equals(Object obj) { ... }

Test runner –

private void persist() {

Course2 course = new Course2("Financial Risk Management", "Harry Barry");

course.setTotalStudents(20);

course.setRegisteredStudents(12);

session.save(course);

...

}

We instantiate the Course2 object with title and tutor, which internally creates the composite key by setting these values on the CoursePK2 object.

1. **Using @IdClass (**not a standard practice)

In this method, we create a composite class (primary key) with all the required primary key attributes. However, we do not annotate this class, so it remains a plain Java class.

public class CoursePK3 implements Serializable {

private String tutor = null;

private String title = null;

public CoursePK3() {}

...

}

Now we need to declare the main entity with the additional class-level annotation @IdClass. This refers to our composite primary key class.

When it comes to the main entity class, there is a disadvantage of following this path. We need to duplicate the composite key identifiers (tutor and title) on our main class too in this method. They must be decorated with @Id.

See the class definition of our main entity class here:

@IdClass(value = CoursePK3.class)

@Entity

@Table(name = "COURSE\_ANNOTATION\_V3")

public class Course3 {

// We must duplicate the identifiers

// defined in our primary class here too

@Id

private String title = null;

@Id

private String tutor = null;

...

}

Using the @IdClass method for defining the composite key is not a standard practice and is best avoided. I suggest you pick either of the first two methods instead.

**Retrieveing Objects:**

To retrieve object from session use following command - session.get(Class\_type, primary\_key).

Eg.

session.get(UserDetails.class, 1);

**Persisting Collections**

Persisting simple values from Hibernate’s view is different from persisting collection elements such as java.util.List or java.util.Map structures.The familiar Java collections—such as List, Set, Array, and Map—are all supported by Hibernate. In fact, it goes one step further and creates a couple of other collections, such as *bag* and *idbag*.

**Important Note -**Always use interfaces when you are defining your collection variables. Hibernate does not like it when we use concrete classes as the variable types:

ArrayList<String> actors = new ArrayList<String>();

Instead, we should define the types using interfaces, like so:

List<String> actors = new ArrayList<String>();

The reason is that behind the scenes, Hibernate uses its own implementation of List!

Persisting Lists

Persisting Sets

Persisting Maps

Persisting Bags andIdBags

**Associations**

**One-to-One Associations:**

There are two ways of establishing a one-to-one association: using a primary key and using a foreign key.

1. **Using a Primary Key**

The basic idea is that both tables exhibit the one-to-one relationship by sharing the same primary key. The tables are designed to share the primary key.

@Entity

@Table(name="CAR\_ONE2ONE\_ANN")

public class Car {

@Id

@Column(name="CAR\_ID")

@GeneratedValue(strategy= GenerationType.AUTO)

private int id;

private String name = null;

@OneToOne (cascade= CascadeType.ALL)

@JoinColumn(name="ENGINE\_ID")

private Engine engine = null;

...

We declare the one-to-one mapping using the @OneToOne annotation. As we have to join the car to the ENGINE table, the @JoinColumn is used on the ENGINE\_ID column.

The Engine class is much simpler. Apart from the usual class-level declarations, such as the @Entity and @Table annotations, the relevant annotation is on the car field (@OneToOne):

@Entity

@Table(name="ENGINE\_ONE2ONE\_ANN")

public class Engine {

@Id

@Column(name="ENGINE\_ID")

@GeneratedValue(strategy= GenerationType.AUTO)

private int id = 0;

@OneToOne(mappedBy="car")

private Car car = null;

...

Test the code -

/\* Test for one-to-one mapping using shared primary key \*/

public class OneToOneTest {

...

private void persist() {

...

// First create an instance of Car, set id and other properties

Car car = new Car();

// Remember, we are using application generator for ids

car.setId(1);

car.setName("Cadillac ATS Sedan");

car.setColor("White");

// Next, create an instance of engine and set values.

// Note: you are not setting the id!

Engine engine = new Engine();

engine.setMake("V8 Series");

engine.setModel("DTS");

engine.setSize("1.6 V8 GAS");

// Now we associate them together using the setter on the car

car.setEngine(engine);

engine.setCar(car);

// Lastly, we are persisting them

session.save(car);

session.save(engine);

...

}

}

Table created as below -

*// CAR table*

CREATE TABLE CAR (

CAR\_ID int(10) NOT NULL,

NAME varchar(20) DEFAULT NULL,

COLOR varchar(20) DEFAULT NULL,

PRIMARY KEY (CAR\_ID))

*// ENGINE table*

CREATE TABLE ENGINE (

CAR\_ID int(10) NOT NULL,

SIZE varchar(20) DEFAULT NULL,

MAKE varchar(20) DEFAULT NULL,

MODEL varchar(20) DEFAULT NULL,

PRIMARY KEY (CAR\_ID),

FOREIGN KEY (CAR\_ID) REFERENCES car (CAR\_ID))

We can see that the primary key CAR\_ID is shared across both tables.

1. **Using a Foreign key**

To use the foreign key strategy, we have to alter the table and mapping definitions.

CREATE TABLE CAR\_V2 (

CAR\_ID int(10) NOT NULL,

ENGINE\_ID int(10) NOT NULL,

COLOR varchar(20) DEFAULT NULL,

NAME varchar(20) DEFAULT NULL,

PRIMARY KEY (CAR\_ID),

CONSTRAINT FK\_ENG\_ID FOREIGN KEY (engine\_id) REFERENCES ENGINE\_v2 (ENGINE\_ID)

)

CREATE TABLE ENGINE\_V2 (

ENGINE\_ID int(10) NOT NULL,

MAKE varchar(20) DEFAULT NULL,

MODEL varchar(20) DEFAULT NULL,

SIZE varchar(20) DEFAULT NULL,

PRIMARY KEY (ENGINE\_ID)

)

We declare the one-to-one mapping using the @OneToOne annotation. As we have to join the car to the ENGINE table, the @JoinColumn is used on the ENGINE\_ID column.

@Entity

@Table(name="CAR\_ONE2ONE\_ANN")

**public class Car** {

@Id

@Column(name="CAR\_ID")

@GeneratedValue(strategy= GenerationType.AUTO)

**private int** id;

**private** String name = **null**;

@OneToOne (cascade= CascadeType.ALL)

@JoinColumn(name="ENGINE\_ID")

**private** Engine engine = **null**;

...

The Engine class is much simpler.

@Entity

@Table(name="ENGINE\_ONE2ONE\_ANN")

**public class Engine** {

@Id

@Column(name="ENGINE\_ID")

@GeneratedValue(strategy= GenerationType.AUTO)

**private int** id = 0;

@OneToOne(mappedBy="car")

**private** Car car = **null**;

...

Finally, to test above

private void persistV2() {

// Create an Engine

Engine e = newEngine();

e.setId(1);

e.setMake("V8 Series");

e.setModel("DTS");

e.setSize("1.6 V8 GAS");

*// Create a Car*

Car car = **new** Car();

car.setId(1);

car.setName("Cadillac ATS Sedan");

car.setColor("White");

*// Associate both*

car.setEngine(e);

*// Now persist the car using the save method.*

*// Note: The Engine gets saved automatically because of the cascade attribute*

*// defined in the mapping*

session.save(car);

...

}

**One-to-Many Associations:**

***Bidirectional:***

create table UNIVERSITY (

   university\_id BIGINT NOT NULL AUTO\_INCREMENT,

   name VARCHAR(30) NOT NULL,

   country  VARCHAR(30) NOT NULL,

   PRIMARY KEY (university\_id)

);

create table STUDENT (

   student\_id BIGINT NOT NULL AUTO\_INCREMENT,

   university\_id BIGINT NOT NULL,

   first\_name VARCHAR(30) NOT NULL,

   last\_name  VARCHAR(30) NOT NULL,

   section    VARCHAR(30) NOT NULL,

   PRIMARY KEY (student\_id),

   CONSTRAINT student\_university FOREIGN KEY (university\_id) REFERENCES UNIVERSITY (university\_id) ON UPDATE CASCADE ON DELETE CASCADE

);

@Entity

@Table(name = "UNIVERSITY")

public class University {

    @Id

    @GeneratedValue

    @Column(name = "UNIVERSITY\_ID")

    private long id;

    @Column(name = "NAME")

    private String name;

    @Column(name = "COUNTRY")

    private String country;

    @OneToMany(mappedBy = "university", cascade = CascadeType.ALL)

    private List<Student> students;

. . .

}

@OneToMany on list property here denotes that one University can have multiple students. mappedBy says that it’s the inverse side of relationship. Also note the cascade attribute, which means the dependent object(Student) will be persisted/updated/deleted automatically on subsequent persist/update/delete on University object. No need to perform operation separately on Student.

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @ManyToOne(optional = false)

    @JoinColumn(name = "UNIVERSITY\_ID")

    private University university;

. . .

}

@JoinColumn says that Student table will contain a separate column UNIVERSITY\_ID which will eventually act as a foreign key reference to primary key of University table. @ManyToOne says that multiple Student tuples can refer to same University Tuples(Multiple students can register in same university).Additionally , with optional=false we make sure that no Student tuple can exist without a University tuple.

Test Program –

Student student1 = new Student("Sam", "Disilva", "Maths");

Student student2 = new Student("Joshua", "Brill", "Science");

Student student3 = new Student("Peter", "Pan", "Physics");

University university = new University("CAMBRIDGE", "ENGLAND");

List<Student> allStudents = new ArrayList<Student>();

student1.setUniversity(university);

student2.setUniversity(university);

student3.setUniversity(university);

allStudents.add(student1);

allStudents.add(student2);

allStudents.add(student3);

university.setStudents(allStudents);

session.persist(university);

//Note - Students will be presisted automatically, thanks to CASCADE.ALL defined on students property of University class.

***Unidirectional:***

create table UNIVERSITY (

   university\_id BIGINT NOT NULL AUTO\_INCREMENT,

   name VARCHAR(30) NOT NULL,

   country  VARCHAR(30) NOT NULL,

   PRIMARY KEY (university\_id)

);

create table STUDENT (

   student\_id BIGINT NOT NULL AUTO\_INCREMENT,

   university\_id BIGINT NOT NULL,

   first\_name VARCHAR(30) NOT NULL,

   last\_name  VARCHAR(30) NOT NULL,

   section    VARCHAR(30) NOT NULL,

   PRIMARY KEY (student\_id),

   CONSTRAINT student\_university FOREIGN KEY (university\_id) REFERENCES UNIVERSITY (university\_id) ON UPDATE CASCADE ON DELETE CASCADE

);

@Entity

@Table(name = "UNIVERSITY")

public class University {

    @Id

    @GeneratedValue

    @Column(name = "UNIVERSITY\_ID")

    private long id;

    @Column(name = "NAME")

    private String name;

    @Column(name = "COUNTRY")

    private String country;

. . .

}

@OneToMany on list property here denotes that one University can have multiple students. mappedBy says that it’s the inverse side of relationship. Also note the cascade attribute, which means the dependent object(Student) will be persisted/updated/deleted automatically on subsequent persist/update/delete on University object. No need to perform operation separately on Student.

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @ManyToOne(optional = false)

    @JoinColumn(name = "UNIVERSITY\_ID")

    private University university;

. . .

}

@*JoinColumn* says that there is a column UNIVERSITY\_ID in Student table which will refer (foreign key) to primary key of the University table. @*ManyToOne* says that multiple Student tuples can refer to same University Tuples (Multiple students can register in same university).Additionally , with *optional*=false we make sure that no Student tuple can exist without a University tuple.

Test Program –

Student student1 = new Student("Sam", "Disilva", "Maths");

Student student2 = new Student("Joshua", "Brill", "Science");

Student student3 = new Student("Peter", "Pan", "Physics");

University university = new University("CAMBRIDGE", "ENGLAND");

student1.setUniversity(university);

student2.setUniversity(university);

student3.setUniversity(university);

session.persist(university);

session.persist(student1);

session.persist(student2);

session.persist(student3);

//As unidirectional so all object has to be saved

**Many-to-Many Associations:-**

***Bidirectional:***

Only change in this relationship (ManyToMany Bidirectional) and [ManyToMany Unidirectional](http://websystique.com/hibernate/hibernate-many-to-many-unidirectional-annotation-example/) is that, in the Subject class we have added following property.

@ManyToMany(mappedBy="subjects")

private List<Student> students = new ArrayList<Student>();

Example

create table STUDENT (

   student\_id BIGINT NOT NULL AUTO\_INCREMENT,

   first\_name VARCHAR(30) NOT NULL,

   last\_name  VARCHAR(30) NOT NULL,

   PRIMARY KEY (student\_id)

);

create table SUBJECT (

   subject\_id BIGINT NOT NULL AUTO\_INCREMENT,

   name VARCHAR(30) NOT NULL,

   PRIMARY KEY (subject\_id)

);

CREATE TABLE STUDENT\_SUBJECT (

    student\_id BIGINT NOT NULL,

    subject\_id BIGINT NOT NULL,

    PRIMARY KEY (student\_id, subject\_id),

    CONSTRAINT FK\_STUDENT FOREIGN KEY (student\_id) REFERENCES STUDENT (student\_id),

    CONSTRAINT FK\_SUBJECT FOREIGN KEY (subject\_id) REFERENCES SUBJECT (subject\_id)

);

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @ManyToMany(cascade = CascadeType.ALL)

    @JoinTable(name = "STUDENT\_SUBJECT",

        joinColumns = { @JoinColumn(name = "STUDENT\_ID") },

        inverseJoinColumns = { @JoinColumn(name = "SUBJECT\_ID") })

    private List<Subject> subjects = new ArrayList<Subject>();

. . .

}

@*ManyToMany* indicates that there is a Many-to-Many relationship between Student and subject. A Student can enroll for multiple subjects, and a subject can have multiple students enrolled. Notice cascade = CascadeType.ALL, with cascading while persisting (update/delete) Student tuples, subjects tuples will also be persisted (updated/deleted).

**@*JoinTable*** indicates that there is a link table which joins two tables via containing there keys. This annotation is mainly used on the owning side of the relationship. *joinColumns* refers to the column name of owning side(STUDENT\_ID of STUDENT), and *inverseJoinColumns* refers to the column of inverse side of relationship(SUBJECT\_ID of SUBJECT).Primary key of this joined table is combination of STUDENT\_ID & SUBJECT\_ID.

@Entity

@Table(name = "SUBJECT")

public class Subject {

    @Id

    @GeneratedValue

    @Column(name = "SUBJECT\_ID")

    private long id;

    @Column(name = "NAME")

    private String name;

    @ManyToMany(mappedBy="subjects")

    private List<Student> students = new ArrayList<Student>();

. . .

}

*mappedBy* attribute tells that this is the inverse side of relationship which is managed by “subjects” property of Student annotated with @*JoinColumn*.

One important remark : In case of \*Many\* association, always override hashcode and equals method which are looked by hibernate when holding entities into collections.

Test program –

Student student1 = new Student("Sam","Disilva");

Student student2 = new Student("Joshua", "Brill");

Subject subject1 = new Subject("Economics");

Subject subject2 = new Subject("Politics");

Subject subject3 = new Subject("Foreign Affairs");

//Student1 have 3 subjects

student1.getSubjects().add(subject1);

student1.getSubjects().add(subject2);

student1.getSubjects().add(subject3);

//Student2 have 2 subjects

student2.getSubjects().add(subject1);

student2.getSubjects().add(subject2);

session.persist(student1);

session.persist(student2);

***Unidirectional:***

For unidirectional many-to-Many association, everything remains as bidirectional, only change is that list of students would not be there in subject class as in bidirectional example.

**Advanced Concept**

**Hibernate Types**

Hibernate uses Java’s reflection to find out the type of the property. Although this option of omitting the types works out fine, the preferred and recommended option is to set the types on the properties implicitly.

**Entity and Value Types**

Hibernate provides two categories of value types, basic types and components:

*Basic types*

The string, boolean, int, long, double, timestamp, and other types fall under this category.

*Components*

Sometimes we wish to have a type defined based on more than one field. The component type defines a set of fields as a specific type. Components are quite handy for splitting table data into varied objects. Components help to organize objects according to the object model rather than depending on a table.

\*\*\*\*Go through Hibernate documentation for this….

**Custom Types**

In addition to these types, Hibernate also provides excellent support for creating our own type.

\*\*\*\*Go through Hibernate documentation for this….

**Caching**

When it comes to performance tuning, caching strategy tops the list. We can improve the performance of any data-intensive application by introducing caching mechanisms. Hibernate supports caching of persistent objects using first-level and second-level caching methods.

**First-Level Caching**

The first level is simply the transactional cache associated with the Session object, which is available during the lifespan of that session or in the conversation only. This caching is provided by default by the framework.

The first-level cache maintained by the session itself, thus a network roundtrip to the database is avoided. The session cache is keyed with the class type, and hence you may have to take extra care when trying to override the existing instance.

**Second-Level Caching**

The second-level cache is globally available via the SessionFactory class. So, any data present in this cache is made available to the entire application.

Hibernate supports a few open source cache implementations, such as EhCache and InfiniSpan.

EhCache is the default second-level cache provider in Hibernate.

\*\*\*\*Go through Hibernate & EhCache documentation for this….

**Caching Queries**

Not only can we cache objects, we can cache queries too. If you have some queries that will be invoked quite often, it is advisable to cache them. To use this functionality, set the hibernate.cache.use\_query\_cache attribute to true.

We need to do one more thing in our code: set the cacheable property on the Query to true by invoking the Query.setCacheable() method.

**Inheritance Strategies**

When we think of object-oriented programming, one principle that comes immediately to mind is inheritance. We always think in terms of *has-a* or *is-a* relationships when modeling real-world problems. Unfortunately, relational databases do not understand the is-a inheritance relationship, although we can get away with using primary and foreign keys for has-a inheritance support. Hibernate overcomes this problem by providing three different strategies to support inheritance persistence, each of which we’ll explore in the following sections.

1. **Table-per-Class Strategy**

The table-per-class strategy, defines one table for all the object hierarchies. This is a simple strategy, as a single table suffices to store the application’s data needs. We differentiate different classes by employing a *discriminator* column.

A discriminator column tags the data for each class separately. In the case of the Employee and Executive object model, there would be two rows in the same table with an additional column indicating each row as an employee or an executive.

Create the parent class Employee with the following annotations:

@Entity(name="INHERITANCE\_S1\_EMPLOYEE\_ANN")

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name="DISCRIMINATOR"

discriminatoryType=DicriminatorType.STRING)

@DiscriminatorValue(value="EMPLOYEE")

public class Employee {

@Id

@Column(name="EMPLOYEE\_ID")

private int id = 0;

...

}

We define the inheritance strategy by annotating our entity with the *@Inheritance* annotation. This annotation accepts a strategy via the strategy variable; in this case, it’s a SINGLE\_TABLE strategy. The InheritanceType also has TABLE\_PER\_CLASS and JOINED strategies. Don’t be tempted to set an InheritanceType.TABLE\_PER\_CLASS value when using the table-per-class strategy—we must set SINGLE\_TABLE only. The TABLE\_PER\_CLASS is set for the table-per-*concrete*-class strategy.

In addition to the inheritance strategy, we need to define the discriminator column (as

discussed in the previous section) via the @DiscriminatorColumn annotation. This annotation

describes the name and type of the discriminator column. The @Discrimina

torValue sets the static value on the entity (EMPLOYEE, in this case).

The subclass EMPLOYEE’s annotations are simple:

@Entity

@DiscriminatorValue(value="EXECUTIVE")

public class Executive extends Employee {

private String role = null;

...

}

Test class

public void test() {

Employee emp = new Employee("Barry Bumbles");

session.save(emp);

Executive ex = new Executive("Harry Dumbles");

ex.setRole("Director");

session.save(ex);

...

}

1. **Table-per-Subclass Strategy**

In the previous section, we saw that the table-per-class strategy persisted all rows to a single table, differentiating each of the rows using a discriminating column. Instead of having one humongous table to store the object graphs, we have the option of a separate *table for each class*. This strategy is called the *table-per-subclass* inheritance persistence strategy.

In this strategy, all the subclasses (including the parent class if the parent class is not an abstract class) will have their own table persistence.

@Entity(name="INHERITANCE\_S2\_EMPLOYEE\_ANN")

@Inheritance(strategy= InheritanceType.JOINED)

public class Employee {

@Id

@Column(name="EMPLOYEE\_ID")

private int id = 0;

...

}

We set the strategy to table-per-subclass by setting the InheritanceType to the JOINED value. On the child class, we need to declare its primary join column using @PrimaryKeyJoinColumn, which is the foreign key.

@Entity(name="INHERITANCE\_S2\_EXECUTIVE\_ANN")

@PrimaryKeyJoinColumn(name="EMPLOYEE\_ID")

public class Executive extends Employee

{

...

}

Table structure for this will look like below

// EMPLOYEE table

CREATE TABLE inheritance\_s2\_employee (

EMPLOYEE\_ID int(11) NOT NULL AUTO\_INCREMENT,

NAME varchar(255) DEFAULT NULL,

PRIMARY KEY (EMPLOYEE\_ID)

)

// EXECUTIVE table

CREATE TABLE inheritance\_s2\_executive (

EMPLOYEE\_ID int(11) NOT NULL,

ROLE varchar(255) DEFAULT NULL,

PRIMARY KEY (EMPLOYEE\_ID),

CONSTRAINT FK\_EMP FOREIGN KEY (EMPLOYEE\_ID)

REFERENCES inheritance\_s2\_employee (EMPLOYEE\_ID)

)

1. **Table-per-Concrete-Class Strategy**

In the table-per-concrete-class strategy, the object hierarchy is persisted to the individual table for each concrete class. Any properties of the superclass will be copied to the child class’s related tables, thus making this strategy uncommon.

Although the Person class doesn’t have its own persistence table, we may still have to annotate it with @Entity and set its inheritance strategy as TABLE\_PER\_CLASS.

@Entity

@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)

public abstract class Person {

@Id

@GeneratedValue

@Column(name="EMPLOYEE\_ID")

private int id;

private String name;

...

}

// Employee class

@Entity(name="INHERITANCE\_S3\_EMPLOYEE\_ANN")

public class Employee extends Person{

private String role = null;

...

}

// Executive class

@Entity(name = "INHERITANCE\_S3\_EXECUTIVE\_ANN")

public class Executive extends Person {

private double bonus = 0.0;

...

}

When you run the test client, the data is populated in both tables as expected. In addition to their own properties, the name property (defined in the parent class) will be duplicated in both the tables too. This is one disadvantage of using the table-per-concrete-class strategy.

**Hibernate Query Language –**

**Fetching All Rows:**

Eg.

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

List<TravelReview> reviews = query.list();

Here, TravelReview is the java class name not the database table name.

**Pagination:**

Eg.

Query query =

session.createQuery("from TravelReview");

query.setMaxResults(100);

query.setFirstResult(10);

List<TravelReview> reviews = query.list();

The setMaxResults method will let Hibernate know that we are interested in seeing only 100 records.

By using the setFirstResult method on the query instance, it sets the starting point to the record to be fetched.

**Retrieving a Unique Record:**

When we know there’s one and only one record for a query criteria, we could use the API’s uniqueResult method.

Query query = session.createQuery("from TravelReview where title='London'");

TravelReview review = (TravelReview) query.uniqueResult();

**Named Parameters:**

Query query = session.createQuery( "from TravelReview where title=:titleId and id=:reviewId");

query.setString("titleId", "London");

query.setInteger("reviewId",1);

**Using the IN option:**

We may need to fetch data with criteria matching a selective list. We need to use HQL’s IN option to use this feature. This is equivalent to using SQL’s IN operator.

We need a list satisfying our criteria, which we set up using standard Java collection classes. In the next example, we use ArrayList to populate this selective list. We then can use the query’s *setParameterList* method, which accepts the list of items from which our query should fetch:

private void getTravelReviewWithQueryParamList() {

{

...

// Define a list and populate with our criteria

List titleList = new ArrayList();

titleList.add("London");

titleList.add("Venice");

// Construct the query

Query query =

session.createQuery("from TravelReview where title in (:titleList)");

// Notice how we've set the named parameter referring to our titleList?

query.setParameterList("titleList", titleList);;

List<TravelReview> reviews = query.list();

...

}

The Hibernate runtime transforms this query into its SQL equivalent:

SELECT \* from TRAVEL\_REVIEW where title in('London','Hyderabad',..)

**Positional Parameters:**

Instead of using the named placeholders (prefixing with *:<name>*) as just shown, you can use positional placeholders. Positional placeholders eliminate the string binding of the name/value pairs in a query, instead using integers as their positions. In this case, you use question marks (*?*) as your placeholders. So, the same code from earlier can be rewritten as:

Query query = session.createQuery("from TravelReview where title=? and id=?");

// title at 0th place

query.setString(0, "London");

// id at 1st place

query.setInteger(1,1);

**Aliases:**

Sometimes we wish to give a name to the object we are querying. The names given to tables are called aliases, and they are especially helpful when we’re constructing joins or subqueries.

The preceding HQL query can be rewritten as shown here:

// The tr is the alias to the object

Query query = session.createQuery(

"from TravelReview as tr where tr.title=:title and tr.id=:id"

);

Note that the *as* keyword is optional.

**Iterators:**

The query.list method returns a list, which in turn returns an iterator. Iterators form an integral part of the Java collection’s toolkit, providing the functionality of iterating through a list.

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

Iterator queryIter = query.list().iterator();

while(queryIter.hasNext()){

TravelReview tr = (TravelReview)queryIter.next();

System.out.println("Travel Review:" + tr);

}

**Select Query:**

The SELECT operator in HQL works along the same lines as in SQL(no surprise there!). Should we wish to fetch selected columns from the database (instead of a whole row), we need to use the SELECT keyword. It is not case sensitive, so select is the same as SELECT.

// The tr is the alias to the object

Query query = session.createQuery("SELECT tr.review from TravelReview as tr");

// Each review is a long description in String format

List<String> reviews = query.list();

System.out.println("City Review:");

// Loop through all of the result columns

for (String review : reviews) {

System.out.println("\t" + review);

}

Or for multiple columns:

String SELECT\_QUERY\_MULTIPLE\_COLUMNS = "SELECT tr.title, tr.review from TravelReview as tr";

Query query = session.createQuery(SELECT\_QUERY\_MULTIPLE\_COLUMNS);

Iterator reviews = query.list().iterator();

while(reviews.hasNext()){

Object[] r = (Object[])reviews.next();

System.out.print("Title:"+r[0]+"\t");

System.out.println("Review:"+r[1]);

}

As a more efficient alternative, we can use another feature in Hibernate’s toolkit—turning the results into a domain object. For example, let’s say we have a City instance composed of title and description. We need to create an instance on every row of data we fetch as follows:

String QUERY = "SELECT new City(tr.title, tr.review ) from TravelReview as tr";

// Obtain the cities

List<City> cities = session.createQuery(QUERY).list();

for (City city : cities) {

System.out.println("City: "+city);

}

**Aggregate Functions:**

Hibernate supports *aggregate* functions such as avg, min, max, and count(\*) that are equivalent to what SQL provides.

// Fetching the max ticket price

List review = session.createQuery("select max(ticket\_price) from TravelFlight")

.list();

// Getting the average age of a planet from a galaxy table

List review = session.createQuery("select avg(planet\_age) from Galaxy").list();

**Updates and Deletes:**

In earlier chapters, we have persisted or updated entities using the session’s save (or saveOrUpdate) or delete methods.

While we can certainly use them, there’s an alternative way to update and delete data: by using the query’s *executeUpdate* method. The method expects a query string with bind parameters,

// To update the record

String UPDATE\_QUERY="update TravelReview set review=:review where id=2";

Query query = session.createQuery(UPDATE\_QUERY);

query.setParameter("review", "The city with charm.The city you will never forget");

int success = query.executeUpdate();

The preceding update query uses a bind parameter to set the review for a record whose id=2. The executeUpdate method updates the table and returns an integer indicating if the update was successful. We can use the same method to execute a delete statement:

// To delete a record

String DELETE\_QUERY="delete TravelReview where id=6";

Query query = session.createQuery(DELETE\_QUERY);

int success = query.executeUpdate();

**Criterias:**

Hibernate provides an alternative way of filtering by introducing *criterias*. A criteria is a mechanism through which we set our query conditions on the entity itself. Hibernate provides us a Criteria class along with another class, Restrictions, through which we set the filtering conditions.

Criteria criteria = session.createCriteria(TravelReview.class);

List review = criteria.add(Restrictions.eq("title", "London")).list();

System.out.println("Using equals: " + review);

First, we create a Criteria instance from the session and then add Restrictions to it. Restrictions has a few static methods—such as eq (equals to), ne (not equals to), and like—which are self-explanatory.

We can add restrictions to the criteria by chaining them

List reviews = session.createCriteria(TravelReview.class)

.add(Restrictions.eq("author", "John Jones"))

.add(Restrictions.between("date",fromDate,toDate))

.add(Restrictions.ne("title","New York")).list();

Should we wish to retrieve only a few columns, we can use the **Projections** class to do so. For example, the following code retrieves the title column from our table:

// Selecting all title columns

List review = session.createCriteria(TravelReview.class)

.setProjection(Projections.property("title"))

.list();

// Getting row count

review = session.createCriteria(TravelReview.class)

.setProjection(Projections.rowCount())

.list();

// Fetching number of titles

review = session.createCriteria(TravelReview.class)

.setProjection(Projections.count("title"))

.list();

The Projections class has a few other static methods, such as avg (average), row Count (SQL’s count(\*) equivalent), count (column count), and max and min (maximum and minimum values).

**Named Queries:**

All the while, we have been using the queries in the code itself. Hard coding the queries isn’t a good practice. We have two ways to remove this constraint. We can use the @*NamedQuery* annotation to bind the queries on the entity at a class level, or we can declare them in our mapping files. In both methods, the named queries are retrieved from the session.

When using the annotation route, we need to add the @NamedQuery annotation to our TravelReview entity. The @NamedQuery annotation accepts a name and the query itself.

@Entity(name="TRAVEL\_REVIEW")

@NamedQuery(name = "GET\_TRAVEL\_REVIEWS",

query = "from com.madhusudhan.jh.hql.TravelReview")

public class TravelReview implements Serializable {

...

}

Once we have defined the named query via the annotation, it can be retrieved from the session during runtime. We must use the name attribute to reference the specific named query, as shown in this example:

private void usingNamedQueries() {

...

// Fetch the predefined named query

Query query = session.getNamedQuery("GET\_TRAVEL\_REVIEWS");

List reviews = query.list();

}

You can also bind multiple queries to an entity by adding each @NamedQuery to a parent @NamedQueries annotation, as shown here:

@Entity(name = "TRAVEL\_REVIEW")

@NamedQueries(

value = {

@NamedQuery(name = "GET\_TRAVEL\_REVIEWS", query = "from TravelReview"),

@NamedQuery(name = "GET\_TRAVEL\_REVIEWS\_FOR\_TITLE",

query = "from TravelReview where id=:title")

}

)

public class TravelReview implements Serializable { ... }

@NamedQueries accepts a value argument, which is made up of an array of @NamedQuery definitions.

**Native SQL:**

Hibernate also provides us with a feature to execute native SQL queries. The session.createSQLQuery method returns a SQLQuery object, similar to how create Query returns a Query object. This class extends the Query class that we have seen in earlier sections.

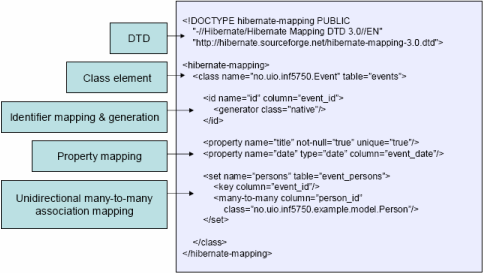
SQLQuery query = session.createSQLQuery("select \* from NATIVESQL\_EMPLOYEE");

List result = query.list();

Extra notes-

**What is the need for Hibernate xml mapping file?**

Hibernate mapping file tells Hibernate which tables and columns to use to load and store objects. Typical mapping file look as follows:

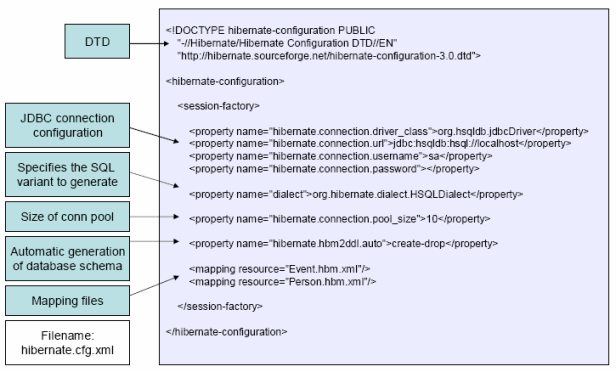
  
  
**8.What are the most common methods of Hibernate configuration?**

The most common methods of Hibernate configuration are:

* Programmatic configuration
* XML configuration (hibernate.cfg.xml)

**9.What are the important tags of hibernate.cfg.xml?**

Following are the important tags of hibernate.cfg.xml:



What are the Core interfaces are of Hibernate framework?

The five core interfaces are used in just about every Hibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

* Session interface
* SessionFactory interface
* Configuration interface
* Transaction interface
* Query and Criteria interfaces

**N+1 SELECTs Problem**

**Example 1**

**Problem (in Simple Words)**

Let’s assume that you’re writing code that’d track the price of mobile phones. Now, let’s say you have a collection of objects representing different Mobile phone vendors (MobileVendor), and each vendor has a collection of objects representing the PhoneModels they offer.

To put it simple, there’s exists a one-to-many relationship between MobileVendor:PhoneModel.

**MobileVendor Class**

|  |  |
| --- | --- |
| 1  2  3  4  5 | Class MobileVendor{          longvendor\_id;          PhoneModel[] phoneModels;          ...   } |

Okay, so you want to print out all the details of phone models. A naive O/R implementation would SELECT all mobile vendors and then do N additional SELECTs for getting the information of PhoneModel for each vendor.

|  |  |
| --- | --- |
| 1  2  3  4  5 | -- Get all Mobile Vendors   SELECT\* FROMMobileVendor;    -- For each MobileVendor, get PhoneModel details   SELECT\* FROMPhoneModel WHEREMobileVendor.vendorId=? |

As you see, the N+1 problem can happen if the first query populates the primary object and the second query populates all the child objects for each of the unique primary objects returned.

**Example 2**

Consider our [simple Book and Shelf relation](http://learningviacode.blogspot.in/2012/07/lazyextra.html). Every shelf has a collection of Books. As we have not applied any optimization strategies, when we request for a Shelf object from the database, a Shelf proxy is loaded. Then when we access a scalar property (like code) the Shelf scalar properties get loaded using a select query. The Book set is represented by a proxy. Now if we try to access the collection a second fetch is needed.  
Consider the below code

**publicstatic** void testFetchSelect() {

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Shelf shelf1 = (Shelf) session.load(Shelf.class, 1);

**System**.out.println("The Shelf class is " + shelf1.getClass());

**System**.out.println("The shelf1 code is " + shelf1.getCode());

**System**.out.println("The Collection class is " + shelf1.getAllBooks().getClass());

**System**.out.println("The number of books in shelf1 is " + shelf1.getAllBooks().size());

transaction.commit();

session.close();

}

The above code loads a Shelf entity and displays the number of books on the shelf.  
The output logs are as below:

The Shelf class is**com.collection.smart.Shelf$$EnhancerByCGLIB$$c94b6717**

Hibernate:

/\* load com.collection.smart.Shelf \*/

**select**

**shelf0\_.ID as ID1\_0\_,**

**shelf0\_.CODE as CODE1\_0\_**

**from**

**SHELF shelf0\_**

**where**

**shelf0\_.ID=?**

The shelf1 code is SH001

The Collection class **is org.hibernate.collection.PersistentSet**

Hibernate:

/\* load one-to-many com.collection.smart.Shelf.allBooks \*/

**select**

**allbooks0\_.SHELF\_ID as SHELF3\_1\_,**

**allbooks0\_.ID as ID1\_,**

**allbooks0\_.ID as ID0\_0\_,**

**allbooks0\_.Name as Name0\_0\_,**

**allbooks0\_.shelf\_id as shelf3\_0\_0\_**

**from**

**BOOK allbooks0\_**

**where**

**allbooks0\_.SHELF\_ID=?**

The number of books in shelf1 is 2

As can be seen initially a Book proxy was loaded. On trying to display the code via a getter, the first select query was fired.   
The Collection is not loaded yet. it is represented in memory by a Hibernate class [PersistentSet.](http://docs.jboss.org/hibernate/core/3.2/api/org/hibernate/collection/PersistentSet.html) On trying to read the book count, a second select query if fired which now loads all the books in the collection thereby initializing it.  
**Thus a total of 2 queries were fired.**One to load the Entity and the second to load the collection.  
Now consider the below code.

**publicstatic** void testNPlus1Select() {

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

@SuppressWarnings("unchecked")

**List**<Shelf>shelfs = session.createQuery("from Shelf shelf where shelf.id = 1

or shelf.id = 2").list();

**for** (Shelf shelf : shelfs) {

Set<Book> books = shelf.getAllBooks();

**System**.out.println("shelf class is " + shelf.getClass());

**System**.out.println("Total books is " + books.size());

}

transaction.commit();

session.close();

}

The code executes a query that will return a list of shelves. Two shelves at most. Each Shelf internally has a Collection proxy. If we access the collection, than a second query will be fired to return the set of books associated with the shelf entity. The logs indicate the same:

Testing Query select

Hibernate:

/\*

from

Shelf shelf

where

shelf.id = 1

or shelf.id = 2 \*/

**select**

**shelf0\_.ID as ID1\_,**

**shelf0\_.CODE as CODE1\_**

**from**

**SHELF shelf0\_**

**where**

**shelf0\_.ID=1**

**or shelf0\_.ID=2**

shelf class is class com.collection.smart.Shelf

Hibernate:

/\* load one-to-many com.collection.smart.Shelf.allBooks \*/

**select**

**allbooks0\_.SHELF\_ID as SHELF3\_1\_,**

**allbooks0\_.ID as ID1\_,**

**allbooks0\_.ID as ID0\_0\_,**

**allbooks0\_.Name as Name0\_0\_,**

**allbooks0\_.shelf\_id as shelf3\_0\_0\_**

**from**

**BOOK allbooks0\_**

**where**

**allbooks0\_.SHELF\_ID=?**

Total books is 2

shelf class is class com.collection.smart.Shelf

Hibernate:

/\* load one-to-many com.collection.smart.Shelf.allBooks \*/

**select**

**allbooks0\_.SHELF\_ID as SHELF3\_1\_,**

**allbooks0\_.ID as ID1\_,**

**allbooks0\_.ID as ID0\_0\_,**

**allbooks0\_.Name as Name0\_0\_,**

**allbooks0\_.shelf\_id as shelf3\_0\_0\_**

**from**

**BOOK allbooks0\_**

**where**

**allbooks0\_.SHELF\_ID=?**

Total books is 2

As can be seen a total of 3 queries were fired. The first query loaded 2 shelves. Than **for each shelf one additional select query was fired. Thus for n shelves, n select book queries would be fired. These n queries + the initial selection query is known as the N+1 query problem.**  
In upcoming posts we shall look at the alternatives available in entity configuration to avoid this problem.

**Hibernate – fetching strategies**

Hibernate has few fetching strategies to optimize the Hibernate generated select statement, so that it can be as efficient as possible. The fetching strategy is declared in the mapping relationship to define how Hibernate fetch its related collections and entities.

**Fetching Strategies**

There are four fetching strategies

1. fetch-”join” = Disable the lazy loading, always load all the collections and entities.  
2. fetch-”select” (default) = Lazy load all the collections and entities.  
3. batch-size=”N” = Fetching up to ‘N’ collections or entities, \*Not record\*.  
4. fetch-”subselect” = Group its collection into a sub select statement.

For detail explanation, you can check on the [Hibernate documentation](https://www.hibernate.org/315.html).

**Fetching strategies examples**

Here’s a “one-to-many relationship” example for the fetching strategies demonstration. A stock is belong to many stock daily records.

*Example to declare fetch strategies in XML file*

...

**<hibernate-mapping>**

**<class**name="com.mkyong.common.Stock"table="stock"**>**

**<set**name="stockDailyRecords"cascade="all"inverse="true"

table="stock\_daily\_record"batch-size="10"fetch="select"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

**</class>**

**</hibernate-mapping>**

*Example to declare fetch strategies in annotation*

...

@Entity

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

**publicclass** Stock **implements**Serializable{

...

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

@Cascade(CascadeType.ALL)

@Fetch(FetchMode.SELECT)

@BatchSize(size =10)

**public** Set<StockDailyRecord>getStockDailyRecords(){

**returnthis**.stockDailyRecords;

}

...

}

Let explore how fetch strategies affect the Hibernate generated SQL statement.

**1. fetch=”select” or @Fetch(FetchMode.SELECT)**

This is the default fetching strategy. it enabled the lazy loading of all it’s related collections. Let see the example…

*//call select from stock*

Stock stock=(Stock)session.get(Stock.**class**, 114);

Set sets =stock.getStockDailyRecords();

*//call select from stock\_daily\_record*

**for**(Iteratoriter=sets.iterator();iter.hasNext();){

StockDailyRecordsdr=(StockDailyRecord)iter.next();

System.out.println(sdr.getDailyRecordId());

System.out.println(sdr.getDate());

}

*Output*

Hibernate:

**select** ...from mkyong.stock

where stock0\_.STOCK\_ID=?

Hibernate:

**select** ...from mkyong.stock\_daily\_record

where stockdaily0\_.STOCK\_ID=?

Hibernate generated two select statements

1. Select statement to retrieve the Stock records -**session.get(Stock.class, 114)**  
2. Select its related collections – **sets.iterator()**

**2. fetch=”join” or @Fetch(FetchMode.JOIN)**

The “join” fetching strategy will disabled the lazy loading of all it’s related collections. Let see the example…

*//call select from stock and stock\_daily\_record*

Stock stock=(Stock)session.get(Stock.**class**, 114);

Set sets =stock.getStockDailyRecords();

*//no extra select*

**for**(Iteratoriter=sets.iterator();iter.hasNext();){

StockDailyRecordsdr=(StockDailyRecord)iter.next();

System.out.println(sdr.getDailyRecordId());

System.out.println(sdr.getDate());

}

*Output*

Hibernate:

**select** ...

from

mkyong.stock stock0\_

left outer **join**

mkyong.stock\_daily\_record stockdaily1\_

on stock0\_.STOCK\_ID=stockdaily1\_.STOCK\_ID

where

stock0\_.STOCK\_ID=?

Hibernate generated only one select statement, it retrieve all its related collections when the Stock is initialized. -**session.get(Stock.class, 114)**

1. Select statement to retrieve the Stock records and outer join its related collections.

**3. batch-size=”10″ or @BatchSize(size = 10)**

This ‘batch size’ fetching strategy is always misunderstanding by many Hibernate developers. Let see the \*misunderstand\* concept here…

Stock stock=(Stock)session.get(Stock.**class**, 114);

Set sets =stock.getStockDailyRecords();

**for**(Iteratoriter=sets.iterator();iter.hasNext();){

StockDailyRecordsdr=(StockDailyRecord)iter.next();

System.out.println(sdr.getDailyRecordId());

System.out.println(sdr.getDate());

}

What is your expected result, is this per-fetch 10 records from collection? See the output  
*Output*

Hibernate:

**select** ...from mkyong.stock

where stock0\_.STOCK\_ID=?

Hibernate:

**select** ...from mkyong.stock\_daily\_record

where stockdaily0\_.STOCK\_ID=?

The batch-size did nothing here, it is not how batch-size work. See this statement.

The batch-size fetching strategy is not define how many records inside in the collections are loaded. Instead, it defines how many collections should be loaded.

— Repeat N times until you remember this statement —

**Another example**

Let see another example, you want to print out all the stock records and its related stock daily records (collections) one by one.

List<Stock> list =session.createQuery("from Stock").list();

**for**(Stock stock: list){

Set sets =stock.getStockDailyRecords();

**for**(Iteratoriter=sets.iterator();iter.hasNext();){

StockDailyRecordsdr=(StockDailyRecord)iter.next();

System.out.println(sdr.getDailyRecordId());

System.out.println(sdr.getDate());

}

}

**No batch-size fetching strategy**

*Output*

Hibernate:

**select** ...

frommkyong.stock stock0\_

Hibernate:

**select** ...

frommkyong.stock\_daily\_record stockdaily0\_

where stockdaily0\_.STOCK\_ID=?

Hibernate:

**select** ...

frommkyong.stock\_daily\_record stockdaily0\_

where stockdaily0\_.STOCK\_ID=?

Keep repeat the **select** statements....depend how many stock records **in** your table.

If you have 20 stock records in the database, the Hibernate’s default fetching strategies will generate 20+1 select statements and hit the database.

1. Select statement to retrieve all the Stock records.  
2. Select its related collection  
3. Select its related collection  
4. Select its related collection  
….  
21. Select its related collection

The generated queries are not efficient and caused a serious performance issue.

**Enabled the batch-size=’10′ fetching strategy**

Let see another example with batch-size=’10′ is enabled.  
*Output*

Hibernate:

**select** ...

frommkyong.stock stock0\_

Hibernate:

**select** ...

frommkyong.stock\_daily\_record stockdaily0\_

where

stockdaily0\_.STOCK\_ID**in(**

?, ?, ?, ?, ?, ?, ?, ?, ?, ?

**)**

Now, Hibernate will per-fetch the collections, with a select \*in\* statement. If you have 20 stock records, it will generate 3 select statements.

1. Select statement to retrieve all the Stock records.  
2. Select In statement to per-fetch its related collections (10 collections a time)  
3. Select In statement to per-fetch its related collections (next 10 collections a time)

With batch-size enabled, it simplify the select statements from 21 select statements to 3 select statements.

**4. fetch=”subselect” or @Fetch(FetchMode.SUBSELECT)**

This fetching strategy is enable all its related collection in a sub select statement. Let see the same query again..

List<Stock> list =session.createQuery("from Stock").list();

**for**(Stock stock: list){

Set sets =stock.getStockDailyRecords();

**for**(Iteratoriter=sets.iterator();iter.hasNext();){

StockDailyRecordsdr=(StockDailyRecord)iter.next();

System.out.println(sdr.getDailyRecordId());

System.out.println(sdr.getDate());

}

}

*Output*

Hibernate:

**select** ...

frommkyong.stock stock0\_

Hibernate:

**select** ...

from

mkyong.stock\_daily\_record stockdaily0\_

where

stockdaily0\_.STOCK\_ID**in(**

**select**

stock0\_.STOCK\_ID

from

mkyong.stock stock0\_

**)**

With “subselect” enabled, it will create two select statements.

1. Select statement to retrieve all the Stock records.  
   2. Select all its related collections in a sub select query.

### Name some important interfaces of Hibernate framework?

Some of the important interfaces of Hibernate framework are:

* 1. **SessionFactory (org.hibernate.SessionFactory)**: SessionFactory is an immutable thread-safe cache of compiled mappings for a single database. We need to initialize SessionFactory once and then we can cache and reuse it. SessionFactory instance is used to get the Session objects for database operations.
  2. **Session (org.hibernate.Session)**: Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps JDBC java.sql.Connectionand works as a factory for org.hibernate.Transaction. We should open session only when it’s required and close it as soon as we are done using it. Session object is the interface between java application code and hibernate framework and provide methods for CRUD operations.
  3. **Transaction (org.hibernate.Transaction)**: Transaction is a single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC or JTA transaction. Aorg.hibernate.Session might span multiple org.hibernate.Transaction in some cases.

### What is Hibernate SessionFactory and how to configure it?

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

The internal state of a SessionFactory is immutable. Once it is created this internal state is set. This internal state includes all of the metadata about Object/Relational Mapping.

SessionFactory also provide methods to get the Class metadata and Statistics instance to get the stats of query executions, second level cache details etc.

### Hibernate SessionFactory is thread safe?

Internal state of SessionFactory is immutable, so it’s thread safe. Multiple threads can access it simultaneously to get Session instances.

### What is Hibernate Session and how to get it?

Hibernate Session is the interface between java application layer and hibernate. This is the core interface used to perform database operations. Lifecycle of a session is bound by the beginning and end of a transaction.

Session provide methods to perform create, read, update and delete operations for a persistent object. We can execute HQL queries, SQL native queries and create criteria using Session object.

### Hibernate Session is thread safe?

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

### What is difference between openSession and getCurrentSession?

Hibernate SessionFactorygetCurrentSession() method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file. Since this session object belongs to the hibernate context, we don’t need to close it. Once the session factory is closed, this session object gets closed.

|  |  |
| --- | --- |
| 1 | <propertyname="hibernate.current\_session\_context\_class">thread</property> |

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

### What is difference between Hibernate Session get() and load() method?

Hibernate session comes with different methods to load data from database. get and load are most used methods, at first look they seems similar but there are some differences between them.

* 1. get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.
  2. Since load() throws exception when data is not found, we should use it only when we know data exists.
  3. We should use get() when we want to make sure data exists in the database.

### What is hibernate caching? Explain Hibernate first level cache?

As the name suggests, hibernate caches query data to make our application faster. Hibernate Cache can be very useful in gaining fast application performance if used correctly. The idea behind cache is to reduce the number of database queries, hence reducing the throughput time of the application.

Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.  
Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.

### What are different states of an entity bean?

An entity bean instance can exist is one of the three states.

* 1. **Transient**: When an object is never persisted or associated with any session, it’s in transient state. Transient instances may be made persistent by calling save(), persist() or saveOrUpdate(). Persistent instances may be made transient by calling delete().
  2. **Persistent**: When an object is associated with a unique session, it’s in persistent state. Any instance returned by a get() or load() method is persistent.
  3. **Detached**: When an object is previously persistent but not associated with any session, it’s in detached state. Detached instances may be made persistent by calling update(), saveOrUpdate(), lock() or replicate(). The state of a transient or detached instance may also be made persistent as a new persistent instance by calling merge().

### What is use of Hibernate Session merge() call?

Hibernate merge can be used to update existing values, however this method create a copy from the passed entity object and return it. The returned object is part of persistent context and tracked for any changes, passed object is not tracked. For example program, read [Hibernate merge](http://www.journaldev.com/3481/hibernate-save-vs-saveorupdate-vs-persist-vs-merge-vs-update-explanation-with-examples).

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

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

Hibernate persist is similar to save with transaction. I feel it’s better than save because we can’t use it outside the boundary of transaction, so all the object mappings are preserved. Also persist doesn’t return the generated id immediately, so data persistence happens when needed.

Hibernate saveOrUpdate results into insert or update queries based on the provided data. If the data is present in the database, update query is executed. We can use saveOrUpdate() without transaction also, but again you will face the issues with mapped objects not getting saved if session is not flushed. For example usage of these methods, read [Hibernate save vs persist](http://www.journaldev.com/3481/hibernate-save-vs-saveorupdate-vs-persist-vs-merge-vs-update-explanation-with-examples).

### What will happen if we don’t have no-args constructor in Entity bean?

Hibernate uses [Reflection API](http://www.journaldev.com/1789/java-reflection-tutorial-for-classes-methods-fields-constructors-annotations-and-much-more) to create instance of Entity beans, usually when you call get() or load() methods. The method Class.newInstance() is used for this and it requires no-args constructor. So if you won’t have no-args constructor in entity beans, hibernate will fail to instantiate it and you will getHibernateException.

### What is difference between sorted collection and ordered collection, which one is better?

When we use Collection API sorting algorithms to sort a collection, it’s called sorted list. For small collections, it’s not much of an overhead but for larger collections it can lead to slow performance and OutOfMemory errors. Also the entity beans should implement Comparable or Comparator interface for it to work, read more at [java object list sorting](http://www.journaldev.com/780/java-comparable-and-comparator-example-to-sort-objects).

If we are using Hibernate framework to load collection data from database, we can use it’s Criteria API to use “order by” clause to get ordered list. Below code snippet shows you how to get it.

|  |  |
| --- | --- |
| 1  2 | List<Employee>empList = session.createCriteria(Employee.class)                          .addOrder(Order.desc("id")).list(); |

Ordered list is better than sorted list because the actual sorting is done at database level, that is fast and doesn’t cause memory issues.

### How to implement Joins in Hibernate?

There are various ways to implement joins in hibernate.

* + Using associations such as one-to-one, one-to-many etc.
  + Using JOIN in the HQL query. There is another form “join fetch” to load associated data simultaneously, no lazy loading.
  + We can fire native sql query and use join keyword.

### What is Query Cache in Hibernate?

Hibernate implements a cache region for queries resultset that integrates closely with the hibernate second-level cache.

This is an optional feature and requires additional steps in code. This is only useful for queries that are run frequently with the same parameters. First of all we need to configure below property in hibernate configuration file.

|  |  |
| --- | --- |
| 1 | <propertyname="hibernate.cache.use\_query\_cache">true</property> |

And in code, we need to use setCacheable(true) method of Query, quick example looks like below.

|  |  |
| --- | --- |
| 1  2  3 | Query query = session.createQuery("from Employee");  query.setCacheable(true);  query.setCacheRegion("ALL\_EMP"); |

### What are the benefits of Named SQL Query?

Hibernate Named Query helps us in grouping queries at a central location rather than letting them scattered all over the code.  
Hibernate Named Query syntax is checked when the hibernate session factory is created, thus making the application fail fast in case of any error in the named queries.  
Hibernate Named Query is global, means once defined it can be used throughout the application.

However one of the major disadvantage of Named query is that it’s hard to debug, because we need to find out the location where it’s defined.

### What is Hibernate Proxy and how it helps in lazy loading?

Hibernate uses proxy object to support lazy loading. Basically when you load data from tables, hibernate doesn’t load all the mapped objects. As soon as you reference a child or lookup object via getter methods, if the linked entity is not in the session cache, then the proxy code will go to the database and load the linked object. It uses javassist to effectively and dynamically generate sub-classed implementations of your entity objects.

### Different types of cascading?

1. None: No Cascading, it’s not a type but when we don’t define any cascading then no operations in parent affects the child.
2. ALL: Cascades save, delete, update, evict, lock, replicate, merge, persist. Basically everything
3. SAVE\_UPDATE: Cascades save and update, available only in hibernate.
4. DELETE: Corresponds to the Hibernate native DELETE action, only in hibernate.
5. DETATCH, MERGE, PERSIST, REFRESH and REMOVE – for similar operations
6. LOCK: Corresponds to the Hibernate native LOCK action.
7. REPLICATE: Corresponds to the Hibernate native REPLICATE action