1. Hibernate also takes a configuration-by-exception approach for annotations so Hibernate uses the name of the POJO class as the default value of the database table the object maps to.
2. Hibernate supports Java Management Extensions (JMX), J2EE Connector Architecture (JCA), and Java Naming and Directory Interface (JNDI) Java language standards. Using JMX, you can configure Hibernate while it is running. In addition, Hibernate does not replace JDBC as a database connectivity layer—Hibernate sits on a level above JDBC.
3. Every relational database/JDBC driver behaves slightly differently, so Hibernate created dialects to abstract away the differences which specify in the Hibernate configuration file. These dialects define the SQL variant and the specific database features to use for each vendor’s database.
4. Hibernate distribution uses two bytecode libraries – javassist and CGLib.
5. To sum up, there are three classes that you need to use: AnnotationConfiguration, SessionFactory and Session.

• Use the AnnotationConfiguration class to read (and to set) configuration details (using configure() method).

• Use the AnnotationConfiguration object to create a SessionFactory object.

• Use the SessionFactory object to create Session objects as needed.

1. A typical application will have one AnnotationConfiguration object, which will only be used in initialization. There will be one SessionFactory object that will exist throughout the life cycle of the application. Application will ask this SessionFactory object for a Session any time it needs to work with the database. After making changes and persist it within one session, and then close down the Session object.
2. The SessionFactory is a heavyweight object, and your application should use one Hibernate SessionFactory object for each discrete database instance that it interacts with. The SessionFactory object is also thread-safe, so you can reuse the session factory in applications with multiple threads – such as most web applications.
3. The Session objects are lightweight. Each thread in application should get a separate Session object from the session factory.
4. Following is the list of all the properties. Mainly properties have their default values:
   * + **hibernate.c3p0.acquire\_increment** The C3P0 database connection pool improves performance of Hibernate applications by managing database connections. Instead of connecting to the database every time a connection is asked for, the connection pool keeps a collection of open database connections for the application to use. There are several C3P0 configuration properties for Hibernate. After the connection pool is completely utilized, determines how many new connections are added to the pool.
     + **hibernate.c3p0.idle\_test\_period** Determines how long to wait before a connection is validated.
     + **hibernate.c3p0.max\_size** The maximum size of the connection pool for C3PO.
     + **hibernate.c3p0.max\_statements** The upper limit for the SQL statement cache for C3PO.
     + **hibernate.c3p0.min\_size** The minimum size of the connection pool for C3PO.
     + **hibernate.c3p0.timeout** The timeout for C3PO (in seconds).
     + **hibernate.cache.provider\_class** Specifies a class that implements the org.hibernate.cache.CacheProviderinterface.
     + **hibernate.cache.query\_cache\_factory** Specifies a class that implements the org.hibernate.cache.QueryCacheFactoryinterface for getting QueryCacheobjects.
     + **hibernate.cache.region\_prefix** The prefix to use for the name of the cache.
     + **hibernate.cache.use\_minimal\_puts** Configures the cache to favor minimal puts over minimal gets.
     + **hibernate.cache.use\_query\_cache** Specifies whether to use the query cache.
     + **hibernate.cache.use\_second\_level\_cache** Determines whether to use the Hibernate second-level cache.
     + **hibernate.cglib.use\_reflection\_optimizer** Instead of using slower standard Java reflection, uses the CGLibcode generation library to optimize access to business object properties. The application may be slower at startup if this is enabled, but with faster runtime performance.
     + **hibernate.connection.autocommit** Allows autocommitmode to be used for the JDBC connection (not usually a good idea).
     + **hibernate.connection.datasource** The DataSourcename for a container-managed datasource.
     + **hibernate.connection.driver\_class** The JDBC driver class.
     + **hibernate.connection.isolation** The transaction isolation level for the JDBC connection.
     + **hibernate.connection....................** Passes any JDBC property you like to the JDBC connection—for instance, hibernate.connection.debuglevel=infowould pass a JDBC property called debuglevel.
     + **hibernate.connection.password** The database password.
     + **hibernate.connection.pool\_size** Limits the number of connections waiting in the Hibernate database connection pool.
     + **hibernate.connection.provider\_class** The class that implements Hibernate’s ConnectionProvider interface.
     + **hibernate.connection.url** The JDBC URL to the database instance.
     + **hibernate.connection.username** The database username.
     + **hibernate.default\_catalog** The default database catalog name that Hibernate uses to generate SQL for unqualified table names.
     + **hibernate.default\_schema** The default database owner name that Hibernate uses to generate SQL for unqualified table names.
     + **hibernate.dialect** The SQL dialect to use for Hibernate; varies by database. See this chapter’s “SQL Dialects” section.
     + **hibernate.generate\_statistics** Determines whether statistics are collected.
     + **hibernate.hbm2ddl.auto** Automatically creates, updates, or drops the database schema on startup and shut down. There are three possible values: create, create-drop, and update. Be careful with create-drop!
     + **hibernate.jdbc.batch\_size** The maximum batch size for updates.
     + **hibernate.jdbc.batch\_versioned\_data** Determines whether Hibernate batches versioned data, which depends on your JDBC driver properly implementing row counts for batch updates. Hibernate uses the row count to determine whether the update is successful.
     + **hibernate.jdbc.factory\_class** The class name of a custom implementation of the org.hibernate.jdbc.Batcherinterface for controlling JDBC prepared statements.
     + **hibernate.jdbc.fetch\_size** Determines how many rows the JDBC connection will try to buffer with every fetch. This is a balance between memory and minimizing database network traffic.
     + **hibernate.jdbc.use\_get\_generated\_keys** Determines Hibernate’s behavior with respect to generated keys. If this property is set to true, and if the database driver supports the JDBC 3.0 generated keys API, Hibernate will retrieve generated keys from the statement after it executes an SQL query.
     + **hibernate.jdbc.use\_scrollable\_resultset** Determines whether Hibernate will use JDBC scrollable result sets for a user-provided JDBC connection.
     + **hibernate.jdbc.use\_streams\_for\_binary** Determines whether binary data is read or written over JDBC as streams.
     + **hibernate.jndi.class** The InitialContextclass for JNDI.
     + **hibernate.jndi.<JNDIpropertyname>** Passes any JNDI property you like to the JNDI InitialContext.
     + **hibernate.jndi.url** Provides the URL for JNDI.
     + **hibernate.max\_fetch\_depth** Determines how deep Hibernate will go to fetch the results of an outer join. Used by Hibernate’s outer join loader.
     + **hibernate.order\_updates** Orders SQL update statements by each primary key.
     + **hibernate.proxool** Prefix for the Proxool database connection pool.
     + **hibernate.proxool.existing\_pool** Configures Proxool with an existing pool.
     + **hibernate.proxool.pool\_alias** The alias to use for any of the configured Proxool pools previously mentioned.
     + **hibernate.proxool.properties** Path to a Proxool properties file.
     + **hibernate.proxool.xml** Path to a Proxool XML configuration file.
     + **hibernate.query.factory\_class** Specifies an HQL query factory class name.
     + **hibernate.query.substitutions** Any possible SQL token substitutions that Hibernate should use.
     + **hibernate.session\_factory\_name** If set, causes the Hibernate session factory to bind to this JNDI name.
     + **hibernate.show\_sql** Logs the generated SQL commands.
     + **hibernate.sql\_exception\_converter** Specifies which SQLExceptionConverterto use to convert SQLExceptionsinto JDBCExceptions.
     + **hibernate.transaction.auto\_close\_session** Automatically closes the session after a transaction.
     + **hibernate.transaction.factory\_class** Specifies a class that implements the org.hibernate.transaction.TransactionFactoryinterface.
     + **hibernate.transaction.flush\_before\_completion** Automatically flushes before completion.
     + **hibernate.transaction.manager\_lookup\_class** Specifies a class that implements the org.hibernate.transaction.TransactionManagerLookup interface.
     + **hibernate.use\_identifier\_rollback** Determines whether Hibernate uses identifier rollback.
     + **hibernate.use\_sql\_comments** Generates SQL with comments.
     + **hibernate.wrap\_result\_sets** Turns on JDBC result set wrapping with column names.
     + **hibernate.xml.output\_stylesheet** Specifies an XSLT stylesheet for Hibernate’s XML data binder. Requires xalan.jar.jta.UserTransactionThe JNDI name for the UserTransaction object.
5. There are two ways to declare Mapped Classes:
6. add the annotated classes directly to an instance of AnnotationConfiguration, use one of the following methods:

* addClass(Class): Takes a Java class name, which is then inspected for Hibernate annotations and added as a mapped class
* addJar(File): Adds any annotated classes in the specified JAR file to the AnnotationConfiguration object

1. you can instead use the <mapping> element in the hibernate.cfg.xml XML configuration file. For annotated classes, the <mapping> element has two possible attributes—jar and class.
2. A custom naming strategy must implement the org.hibernate.cfg.NamingStrategy interface or extend one of the two provided naming strategy classes, org.hibernate.cfg.DefaultNamingStrategy or org.hibernate.cfg.ImprovedNamingStrategy. Once you have created a naming strategy, pass an instance of it to the AnnotationConfiguration object’s setNamingStrategy() method. For example for using the ImprovedNamingStrategy naming strategy:

AnnotationConfiguration conf = new AnnotationConfiguration()

conf.setNamingStrategy(ImprovedNamingStrategy.INSTANCE);

1. The default naming strategy simply returns the unqualified Java class name as the database table name. For instance, the table name for the Java class com.hibernatebook.AccessGroups would be AccessGroups. The column name would be the same as the property name, and the collection table would have the same name as the property.
2. The improved naming strategy adds underscores in place of uppercase letters in mixed-case table and column names, and then lowercases the name. For instance, the same com.hibernatebook.AccessGroups Java class would correspond to a database table named access\_groups.
3. Hibernate can obtain a data source through a JNDI lookup(in case of Container Managed Data Source). We must use the hibernate.connection.datasource property to specify the JNDI name and then we may set the optional hibernate.jndi.url and hibernate.jndi.class properties to specify the location of the container’s JNDI provider and the class name of the container’s implementation of the JNDI InitialContextFactory interface.
4. A session from the SessionFactory object can be created using one of the four openSession() methods. The no-argument openSession() method opens a session, with the database connection and interceptor specified in the SessionFactory’s original configuration. You can explicitly pass a JDBC connection to use, a Hibernate interceptor, or both as arguments to the remaining openSession() methods.
5. SessionFactory.close() method releases all the resource information used by the session factory and made available to the Session objects. When the session factory closes, it destroys the cache for the entity persisters and collection persisters, and also destroys the query cache and the timestamps cache. Then the session factory closes the JDBC connection provider and removes the current instance from its JNDI object factory binding.
6. If we don’t want to provide the configuration properties in a file, we can apply them directly using the -D flag. e.g.

java -classpath ...

-Dhibernate.connection.driver\_class=org.hsqldb.jdbcDriver

...

It is useful when running tools and utilities on an ad hoc basis.

1. Each class that will be persisted by Hibernate is required to have a default constructor with at least package scope. They should have get and set methods for all of the attributes that are to be persisted.
2. A surrogate key is an arbitrary value (usually numeric), with the data type depending on the number of objects expected (e.g., 32-bit, 64-bit, etc.).
3. An object that is mapped to Hibernate can be in any one of three different states: **transient, persistent, or detached**.
4. **Transient** objects exist in memory. Hibernate does not manage transient objects or persist changes to transient objects. To persist the changes to a transient object, you would have to ask the session to save the transient object to the database, at which point Hibernate assigns the object an identifier.
5. **Persistent** objects exist in the database, and Hibernate manages the persistence for persistent objects. If fields or properties change on a persistent object, Hibernate will keep the database representation up-to-date.
6. **Detached** objects have a representation in the database, but changes to the object will not be reflected in the database, and vice versa. A detached object can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session’s evict() method. One reason you might consider doing this would be to read an object out of the database, modify the properties of the object in memory, and then store the results some place other than your database. In order to persist changes made to a detached object, the application must reattach it to a valid Hibernate session.
7. A detached instance can be associated with a new Hibernate session when your application calls one of the load(), refresh(), merge(), update(), or save() methods on the new session with a reference to the detached object. After the call, the detached object would be a persistent object managed by the new Hibernate session. Versions prior to Hibernate 3 had support for the Lifecycle and Validatable interfaces. These allowed your objects to listen for save, update, delete, load, and validate events using methods on the object. In Hibernate 3, this functionality moved into events and interceptors, and the old interfaces were removed.
8. If no primary key field or property is available , then Hibernate itself can manage the identifier value internally.
9. While Hibernate lets us specify that changes to one association will result in changes to the database, it does not allow us to cause changes to one end of the association(or mapping/relations) to be automatically reflected in the other end in the Java POJOs.

openSession();

beginTransaction();

Email email = new Email("Test Email");

Message message = new Message("Test Message");

email.setMessage(message);

save(email,message);

System.out.println("Stored...");

System.out.println(email);

System.out.println(email.getMessage());

System.out.println(message);

System.out.println(message.getEmail());

Serializable emailPrimaryKey = session.getIdentifier(email);

Serializable messagePrimaryKey = session.getIdentifier(message);

endTransaction();

closeSession();

System.out.println();

openSession();

beginTransaction();

email = (Email)session.get(Email.class,emailPrimaryKey);

message = (Message)session.get(Message.class,messagePrimaryKey);

System.out.println("Retrieved...");

System.out.println(email);

System.out.println(email.getMessage());

System.out.println(message);

System.out.println(message.getEmail());

endTransaction();

closeSession();

If you run the code from Listing 4-4, you will see the following output:

Stored...

Test Email

Test Message

Test Message

null

Retrieved...

Test Email

Test Message

Test Message

Test Emails

When the entities are initially stored, the Message object’s reference to its associated Email is null, even after Hibernate has stored the data. The entity in memory is not updated to reflect the change to the Email entity. However, after we have closed the session, opened a new one, and loaded the entities from the database, the entity has been updated. Because the session has been closed, the session is forced to reload the entities from the database when we request them by primary key. Because the Email entity is the owner of the association, the association exists in the database purely in the form of a foreign key relationship from the Email table onto the Message table’s primary key. When we altered the Email entity and saved it, this foreign key relationship was therefore updated. So, when we reload the entities, the Message entity’s association details are (correctly) obtained from the same foreign key.

1. If we alter this code to make the association in the Message entity instead of the Email entity, but leave the Email entity the owner of the association, we will see the reverse effect, as follows:

Email email = new Email("Test Email");

Message message = new Message("Test Message");

//email.setMessage(message);

message.setEmail(email);

Because we have not made the association in the Email entity (the owner), the foreign key of the Email table is not pointed at the Message table. When we reload the entities, we do not see the “automatic” association behavior—quite the opposite:

Stored...

Test Email

null

Test Message

Test Email

Retrieved...

Test Email

null

Test Message

null

1. Table shows how you can select the side of the relationship that should be made the owner of a bidirectional association. Remember that to make an association the owner, you must mark the other end as inverse="true" (the choice of terminology is poor, but entrenched).

|  |  |
| --- | --- |
| Type of Association | Options |
| One-to-one | Either end can be made the owner, but one (and only one) of them should be—if you don’t specify this, you will end up with a circular dependency. |
| One-to-many | The many end must be made the owner of the association |
| Many-to-one | This is the same as the one-to-many relationship, viewed from the opposite perspective, so the same rule applies—the many end must be made the owner of the association. |
| Many-to-many | Either end of the association can be made the owner |

1. Both save() methods take a transient object reference (which must not be null) as an argument.

public **Serializable** save(Object object) throws HibernateException

public **Serializable** save(String entityName,Object object) throws HibernateException

1. Requesting a persistent object again from the same Hibernate session returns the same Java instance of a class, which means that you can compare the objects using the standard Java == equality syntax. If, however, you request a persistent object from more than one Hibernate session, Hibernate will provide distinct instances from each session, and the == operator will return false if you compare these object instances.
2. The load() methods will throw an exception if the unique id(Serializable Id) is not found in the database, whereas the get() methods will merely return a null reference.
3. If we use SQL to update the database, Hibernate will not be aware that the representation changed. So we should call the refresh() method.
4. If a property changes on a persistent object, the associated Hibernate session will queue the change for persistence to the database using SQL. You can also determine whether the session is dirty and changes need to be committed. The flush() method forces Hibernate to flush the session, as follows:

public void flush() throws HibernateException

You can determine if the session is dirty with the isDirty() method, as follows:

public boolean isDirty() throws HibernateException

You can also instruct Hibernate to use a flushing mode for the session with the setFlushMode() method. The getFlushMode() method returns the flush mode for the current session, as follows:

public void setFlushMode(FlushMode flushMode)

public FlushMode getFlushMode()

The possible flush modes are the following:

• ALWAYS: Every query flushes the session before the query is executed. This is going to be very slow.

• AUTO: Hibernate manages the query flushing to guarantee that the data returned by a query is up-to-date.

• COMMIT: Hibernate flushes the session on transaction commits.

• MANUAL: Your application needs to manage the session flushing with the flush() method. Hibernate never flushes the session itself.

By default, Hibernate uses the AUTO flush mode.

1. If we set the cascade attribute to delete or all, the delete will be cascaded to all of the associated objects but bulk deletes (delete using HQL query) do not cause cascade operations to be carried out. If cascade behavior is needed, we will need to carry out the appropriate deletions ourself, or use the session’s delete() method.
2. All of the basic life cycle operations discussed in this chapter have associated cascade values, as follows:

• create

• merge

• delete

• save-update

• evict

• replicate

• lock

• refresh

These values can be concatenated in a comma-separated list to allow cascading for any combination of these operations. When all operations should be cascaded, Hibernate provides a shortcut value named all that tells Hibernate to cascade all of these operations from the parent to each child object (for that relationship), except for delete-orphan which is Use delete-orphan to remove a child object from the database when you remove the child from the parent’s collection. This cascading type only works on one-to-many associations.

1. Lazy loading is the default using XML mappings, but not when using annotations.
2. Maintaining the mapping information as external XML files allows the mapping information to be changed to reflect business changes or schema alterations without forcing you to rebuild the application as a whole.
3. The entity class must have a no-argument constructor that is visible with at least protected scope. Hibernate supports package scope as the minimum, but you lose portability to other containers if you take advantage of this. Other JPA 2 rules for an entity bean class are that the class must not be final, and that the entity bean class must be concrete.
4. If the annotation is applied to a field, then field access will be used while if annotation is applied to the getter for the field, then property access will be used.
5. By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to use but we can override this by also applying the @GeneratedValue annotation. This takes a pair of attributes: strategy and generator.
6. There are four different types of primary key generators on GeneratorType, as follows:

• AUTO: Hibernate decides which generator type to use, based on the database’s support for primary key generation. This is the default value.

• IDENTITY: The database is responsible for determining and assigning the next primary key.

• SEQUENCE: Some databases support a SEQUENCE column type. Here’s an example:

@Id

@SequenceGenerator(name="seq1",sequenceName="HIB\_SEQ")

@GeneratedValue(strategy=SEQUENCE,generator="seq1")

public int getId() {

return id;

}

Here, a sequence generation annotation named seq1 refers to the database sequence object called HIB\_SEQ. Only the sequence generator name is mandatory—the other attributes will take sensible default values.

• TABLE: This type keeps a separate table with the primary key values. E.g.

@Id

@TableGenerator(name="tablegen", table="ID\_TABLE", pkColumnName="ID", valueColumnName="NEXT\_ID")

@GeneratedValue(strategy=TABLE,generator="tablegen")

public int getId() {

return id;

}

These available values for the strategy attribute do not exactly match the values for Hibernate’s primary key generators for XML mapping. If you use Hibernate-specific primary key generation strategies, you will risk forfeiting portability of your application to other JPA 2 environments.

1. For the optimal portability and optimal performance, we should avoid the use of a table generator, but instead use the AUTO configuration.
2. The optional attributes for TableGenerator are as follows:

• allocationSize: Allows the number of primary keys set aside at one time to be tuned for performance.

• catalog: Allows the catalog that the table resides within to be specified.

• initialValue: Allows the starting primary key value to be specified.

• pkColumnName: Allows the primary key column of the table to be identified. The table can contain the details necessary for generating primary key values for multiple entities.

• pkColumnValue: Allows the primary key for the row containing the primary key generation information to be identified.

• schema: Allows the schema that the table resides within to be specified.

• table: The name of the table containing the primary key values.

• uniqueConstraints: Allows additional constraints to be applied to the table for schema generation.

• valueColumnName: Allows the column containing the primary key generation information for the current entity to be identified.

1. For composite primary keys, we must create a public serializable class with a default constructor to represent this primary key. This class must implement hashCode() and equals() methods to allow the Hibernate code to test for primary key collisions .We can use any of three strategies for using this primary key class once it has been created are as follows:
2. • Mark it (pk class) as @Embeddable and add to your entity class a normal property for it, marked with @Id. This is the most natural approach. ). It allows you to treat the compound primary key as a single property, and it permits the reuse of the @Embeddable class in other tables.

• Add to your entity class a normal property for it, marked with @EmbeddableId. This is most natural approach. Here, the primary key class cannot be used in other tables since it is not an @Embeddable entity, but it does allow us to treat the key as a single attribute of the Account class

1. • Add properties to your entity class for all of its fields, mark them with @Id, and mark your entity class with @IdClass, supplying the class of your primary key class. This approach allows us to map the compound primary key class using properties of the entity itself corresponding to the names of the properties in the primary key class. The names must correspond (there is no mechanism for overriding this), and the primary key class must honor the same obligations as with the other two techniques. The only advantage to this approach is its ability to “hide” the use of the primary key class from the interface of the enclosing entity. The @IdClass annotation takes a value parameter of Class type. The fields that correspond to the properties of the primary key class to be used must all be annotated with @Id.
2. The @Table annotation provides four attributes, allowing you to override the name of the table, its catalog, and its schema, and enforce unique constraints on columns in the table. While @SecondaryTable annotation provides a way to model an entity bean that is persisted across several different database tables.
3. When the field or property is not a primitive, it can be stored and retrieved as a null value. This default behavior can be overridden by applying the @Basic.This annotation takes two optional attributes. The first attribute is named optional and takes a boolean. Defaulting to true, this can be set to false to provide a hint to schema generation that the associated column should be created NOT NULL. The second is named fetch and takes a member of the enumeration FetchType. This is EAGER by default, but can be set to LAZY to permit loading on access of the value.
4. Some fields, such as calculated values, may be used at run time only and they should be discarded from objects as they are persisted into the database. The EJB 3 specification provides the @Transient annotation for these transient fields. The @Transient annotation does not have any attributes.
5. One of the important differences between using annotations with Hibernate and using XML mapping documents. With annotations, Hibernate will default to persisting all of the fields on a mapped object. When using XML mapping documents, Hibernate requires you to tell it explicitly which fields will be persisted.
6. @Column has the following attributes:

* **name** permits the name of the column to be explicitly specified—by default, this would be the name of the property.
* **length** permits the size of the column used to map a value (particularly a String value) to be explicitly defined. The column size defaults to 255.
* **nullable** permits the column to be marked NOT NULL when the schema is generated. The default is that fields should be permitted to be null.
* **unique** permits the column to be marked as containing only unique values. This defaults to false.
* **table** is used when the owning entity has been mapped across one or more secondary tables. By default, the value is assumed to be drawn from the primary table, but the name of one of the secondary tables can be substituted here.
* **insertable** defaults to true, but if set to false, the annotated field will be omitted from insert statements generated by Hibernate (i.e., it won’t be persisted).
* **updatable** defaults to true, but if set to false, the annotated field will be omitted from update statements generated by Hibernate (i.e., it won’t be altered once it has been persisted).
* **columnDefinition** can be set to an appropriate DDL fragment to be used when generating the column in the database. This can only be used during schema generation from the annotated entity, and should be avoided if possible, since it is likely to reduce the portability of your application between database dialects.
* **precision** permits the precision of decimal numeric columns to be specified for schema generation, and will be ignored when a non-decimal value is persisted.The value given represents the number of digits in the number (usually requiring a minimum length of n+1, where n is the scale).
* **scale** permits the scale of decimal numeric columns to be specified for schema generation and will be ignored where a non-decimal value is persisted. Thevalue given represents the number of places after the decimal point.

1. When all the fields of one entity are maintained within the same table as another, the enclosed entity is referred to in Hibernate as a **component**. The JPA 2 standard refers to such an entity as being **embedded**. An embeddable entity can only use the @Basic, @Column, @Lob, @Temporal, and @Enumerated annotations. It cannot maintain its own primary key with the @Id tag because its primary key is the primary key of the enclosing entity. The @Embeddable annotation itself is purely a marker annotation, and takes no additional attributes.
2. The @Embedded annotation draws its column information from the embedded type, but permits the overriding of a specific column or columns with the @AttributeOverride and @AttributeOverrides tags. E.g. AuthorAddress with columns named ADDR and NATION.

@Embedded

@AttributeOverrides({

@AttributeOverride(name="address",column=@Column(name="ADDR")),

@AttributeOverride(name="country",column=@Column(name="NATION"))

})

public AuthorAddress getAddress() {

return this.address;

}

1. One-to-one association can have a bidirectional relationship with a. One side will need to own the relationship and be responsible for updating a join column with a foreign key to the other side. The non-owning side will need to use the mappedBy attribute to indicate the entity that owns the relationship.The @OneToOne annotation permits the following optional attributes to be specified:

**targetEntity** can be set to the class of an entity storing the association. If left unset, the appropriate type will be inferred from the field type, or the return type of the property’s getter.

**cascade** can be set to any of the members of the javax.persistence.CascadeType enumeration. It defaults to none being set.

**fetch** can be set to the EAGER or LAZY members of FetchType.

**optional** indicates whether the value being mapped can be null.

**orphanRemoval** indicates that if the value being mapped is deleted, this entity will also be deleted.

**mappedBy** indicates that a bidirectional one-to-one relationship is owned by the named entity. The owning entity contains the primary key of the subordinate entity.

1. Following are the list of all Cascade type:

• **ALL** requires all operations to be cascaded to dependent entities. This is the same as including **MERGE, PERSIST, REFRESH, DETACH, and REMOVE.**

• **MERGE** cascades updates to the entity’s state in the database **(i.e. UPDATE ...).**

• **PERSIST** cascades the initial storing of the entity’s state in the database **(i.e. INSERT. . .).**

• **REFRESH** cascades the updating of the entity’s state from the database **(i.e. SELECT . . .).**

**• DETACH** cascades the removal of the entity from the managed persistence context.

• **REMOVE** cascades deletion of the entity from the database **(i.e., DELETE . . .).**

• If no cascade type is specified, no operations will be cascaded through the association.

1. An ordered collection can be persisted in Hibernate or JPA 2 using the @OrderColumn annotation to maintain the order of the collection. You can also order the collection at retrieval time by means of the @OrderBy annotation. The value of the @OrderBy annotation is an ordered list of the field names to sort by, each one optionally appended with ASC (for ascending order, as in the preceding code) or DESC (for descending order). If neither ASC nor DESC is appended to one of the field names, the order will default to ascending. @OrderBy can be applied to any collection-valued association.
2. The @OneToMany annotation can be applied to a field or property value for a collection or an array representing the mapped “many” end of the association.

* The **mappedBy** attribute is mandatory on a bidirectional association and optional (being implicit) on a unidirectional association.
* **cascade** is optional, taking a member of the javax.persistence.CascadeType enumeration and dictating the cascade behavior of the mapped entity.
* **targetEntity** is optional, as it can usually be deduced from the type of the field or property
* **fetch** is optional, allowing lazy or eager fetching to be specified as a member of the javax.persistence.FetchType enumeration.

1. The @ManyToOne annotation takes a similar set of attributes to @OneToMany. The following list describes the attributes, all of which are optional.

**cascade** indicates the appropriate cascade policy for operations on the association; it defaults to none.

**fetch** indicates the fetch strategy to use; it defaults to LAZY.

**optional** indicates whether the value can be null; it defaults to true.

**targetEntity** indicates the entity that stores the primary key—this is normally inferred from the type of the field or property.

1. When a unidirectional one-to-many association is to be formed, it is possible to express the relationship using a link table. The @JoinTable annotation provides attributes that allow various aspects of the link table to be controlled. These attributes are as follows:

**name** is the name of the join table to be used to represent the association.

**catalog** is the name of the catalog containing the join table.

**schema** is the name of the schema containing the join table.

**joinColumns** is an array of @JoinColumn attributes representing the primary key of the entity at the “one” end of the association.

**inverseJoinColumns** is an array of @JoinColumn attributes representing the primary key of the entity at the “many” end of the association.

@OneToMany(cascade = ALL)

@JoinTable(

name="PublishedBooks",

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

inverseJoinColumns = @JoinColumn( name = "book\_id")

)

public Set<Book> getBooks() {

return books;

}

1. @ManyToMany takes the following attributes:

* **mappedBy** is the field that owns the relationship—this is only required if the association is bidirectional. If an entity provides this attribute, then the other end of the association is the owner of the association, and the attribute must name a field or property of that entity.
* **targetEntity** is the entity class that is the target of the association. Again, this may be inferred from the generic or array declaration, and only needs to be specified if this is not possible.
* **cascade** indicates the cascade behavior of the association, which defaults to none.
* **fetch** indicates the fetch behavior of the association, which defaults to LAZY.

1. The JPA 2 standard and Hibernate both support three approaches to mapping inheritance hierarchies into the database. These are as follows:

• Single table (SINGLE\_TABLE): One table for each class hierarchy.

• Joined (JOINED): One table for each subclass (including interfaces and abstract classes)

• Table-per-class (TABLE\_PER\_CLASS): One table for each concrete class implementation

Persistent entities that are related by inheritance must be marked up with the @Inheritance annotation. This takes a single strategy attribute, which is set to one of three javax.persistence.InheritanceType enumeration values corresponding to these approaches.

1. In case of Single Table approach, to determine the appropriate type to instantiate when retrieving entities from the database, an @DiscriminatorColumn annotation should be provided in the root (and only in the root) of the persistent hierarchy. This defines a column containing a value that distinguishes between each of the types used.The attributes permitted by the @DiscriminatorColumn annotation are as follows:

**name** is the name of the discriminator column.

**discriminatorType** is the type of value to be stored in the column as selected from the javax.persistence.DiscriminatorType enumeration of STRING, CHAR or INTEGER.

**columnDefinition** is a fragment of DDL defining the column type. Using this is liable to reduce the portability of your code across databases.

**length** is the column length of STRING discriminator types. It is ignored for CHAR and INTEGER types.

In the following example, we specify that an INTEGER discriminator type should be stored in the column named DISCRIMINATOR. Rows representing Book entities will have a value of 1 in this column, whereas the rows representing ComputerBook entities should have a value of 2 in the same column.

@Entity

@Inheritance(strategy = SINGLE\_TABLE)

@DiscriminatorColumn(name="DISCRIMINATOR", discriminatorType=INTEGER)

@DiscriminatorValue("1")

public class Book {

...

}

and

@Entity

@DiscriminatorValue("2")

public class ComputerBook extends Book {

...

}

1. @DiscriminatorColumn annotation is not applicable to Table-per-class while applicable to others only.
2. Fields or properties of an entity that have java.util.Date or java.util.Calendar types represent temporal data. By default, these will be stored in a column with the TIMESTAMP data type, but this default behavior can be overridden with the @Temporal annotation which accepts a single value attribute from the javax.persistence.TemporalType enumeration. This offers three possible values: DATE, TIME, and TIMESTAMP.
3. @ElementCollection annotation can be used for mapping collections of basic or embeddable classes. There are two attributes on the @ElementCollection annotation: targetClass and fetch. The targetClass attribute tells Hibernate which class is stored in the collection. If you use generics on your collection, you do not need to specify targetClass because Hibernate will infer the correct class. The fetch attribute takes a member of the enumeration, FetchType. This is EAGER by default, but can be set to LAZY to permit loading when the value is accessed.
4. A persistent property or field can be marked for persistence as a database-supported large object type by applying the @Lob annotation. The annotation takes no attributes. String and character based types will be stored in an appropriate character-based type. All other objects will be stored in a BLOB.
5. A special case of inheritance occurs when the root of the hierarchy is not itself a persistent entity, but various classes derived from it are. Such a class can be abstract or concrete. The @MappedSuperclass annotation allows you to take advantage of this circumstance. The class marked with @MappedSuperclass is not an entity, and is not queryable (it cannot be passed to methods that expect an entity in the Session or EntityManager objects). It cannot be the target of an association.
6. @NamedQuery and @NamedQueries allow one or more Hibernate Query Language or Java Persistence Query Language (JPQL) queries to be associated with an entity. The required attributes are as follows:

**name** is the name by which the query is retrieved.

**query** is the JPQL (or HQL) query associated with the name.

**hints** is taking a QueryHint annotation name/value pair, which allows caching mode, timeout value, and a variety of other platform-specific tweaks to be .

If a query has no natural association with any of the entity declarations, it is possible to make the @NamedQuery annotation at the package level. There is no natural place to put a package-level annotation, so Java annotations allow for a specific file, called package-info.java, to contain them. Here is an example:

@javax.annotations.NamedQuery(

name="findAuthorsByName",

query="from Author where name = :author"

)

package com.hibernatebook.annotations;

1. Hibernate also allows the database’s native query language (usually a dialect of SQL) to be used in place of HQL or JPQL. The @NamedNativeQuery annotation is declared in almost exactly the same manner as the @NamedQuery annotation.
2. The Hibernate-specific @Entity annotation extends the basic details of the @javax.persistence.Entity annotation. It allows the following additional attributes to be specified:

**dynamicInsert** is used to flag that insert statements should be generated at run time (not at startup), allowing only the altered columns to be inserted. By default this is disabled.

**dynamicUpdate** is used to flag that update statements should be generated at runtime, allowing only the altered columns to be updated. By default this is disabled.

**mutable** is true by default, but if set to false, it allows the persistence engine to cache the values read from the database, and the persistence engine will make no attempt to update them in response to changes (changes that should not be made if this flag is set to false).

**optimisticLock** allows an optimistic lock strategy to be selected from the OptimisticLockType enumeration values of ALL, DIRTY, NONE, and VERSION. This defaults to VERSION.

**persister** allows a persister class other than the default Hibernate one to be selected for the entity.

**polymorphism** allows the polymorphism strategy to be selected from the PolymorphismType enumeration values of EXPLICIT and IMPLICIT. This defaults to IMPLICIT.

**selectBeforeUpdate** allows the user to request that a SELECT be performed to retrieve the entity before any potential update.

1. The Hibernate-specific @Sort annotation allows a collection managed by Hibernate to be sorted by a standard Java comparator. The following code gives an example.

@javac.persistence.OneToMany

@org.hibernate.annotations.Sort(

type=org.hibernate.annotations.SortType.COMPARATOR,

comparator=EmployeeComparator.class

)

public Set<Employee> getEmployees() {

return this.employees;

}

1. The Hibernate-specific @Table annotation supplements the standard table annotation and allows additional index hints to be provided to Hibernate. The following code gives an example.

// Standard persistence annotations:

@javax.persistence.Entity

@javax.persistence.Table(name="FOO")

// Hibernate-specific table annotation:

@Table(

appliesTo="FOO", indexes = {

@Index(name="FOO\_FROM\_TO\_IDX",columnNames={"FIRST","LAST"}),

@Index(name="FOO\_EMPLOYEE\_IDX",columnNames={"EMPLOYEE\_NUM"}))

public class Foo {

...

}

1. The contents of a collection that will be retrieved from the database can be restricted with a Hibernatespecific @Where annotation from the @org.hibernate.annotations package. This simply adds a Where clause to the query that will be used to obtain the entities contained within the collection. Here’s an example:

@javax.persistence.OneToMany

@org.hibernate.annotations.Where(clause="grade > 2")

public Set<Employee> getEmployees() {

return this.employees;

}

1. For alternative Key Generation Strategies, Hibernate supplies the @GenericGenerator .The attributes that can be supplied to the annotation are as follows:

**name** is mandatory, and is used to identify the generic generator in the @GeneratedValue annotation.

**strategy** is mandatory, and determines the generator type to be used. This can be a standard Hibernate generator type or the name of a class implementing the org.hibernate.id.IdentifierGenerator interface.

**parameters** is a list of @Parameter annotations defining any parameter values required by the generator strategy.

The available standard Hibernate strategies are increment, identity, sequence, hilo, seqhilo, uuid, guid, native, assigned, select, and foreign. For example, the non-standard uuid strategy for a primary

1. Hibernate types fall into three broad categories: entities, components, and values.
2. An entity can also be a dynamic map (actually a Map of Maps). These are mapped against the database in the same way as a POJO, but with the default entity mode of the SessionFactory set to dynamic-map. The advantage of POJOs over the dynamic-map approach is that compile-time type safety is retained. Conversely, dynamic maps are a quick way to get up and running when building prototypes. It is also possible to represent your entities as Dom4J Document objects. This is a useful feature when importing and exporting data from a preexisting Hibernate database.
3. In addition to these standard types, you can create your own. Your user type class should implement either the org.hibernate.usertype.UserType interface or the org.hibernate.usertype.CompositeUserType interface.
4. The required order and cardinality of the child elements of <hibernate-mapping> are as follows:

(meta\*, typedef\*, import\*, (class | subclass | joined-subclass | union-subclass)\*,

(query | sql-query)\*, filter-def\*).

\* represent represent zero or more occurence

1. The default implicit polymorphism behavior in Hibernate returns instances of the class if superclasses or implemented interfaces are named in the query, and returns subclasses if the class itself is named in the query.
2. Proxy in hibernate specifies a class or an interface to use as the proxy for lazy initialization. Hibernate uses runtime-generated proxies by default, but you can specify your own implementation of org.hibernate.HibernateProxy in their place.
3. The <class> element have an <id> or a <composite-id> element. The child elements of the <id> element are as follows: (meta\*, column\*, type?, generator?) where ‘?’ is zero or one occurrence and <generator> element defines how to generate a new primary key for a new instance of the class. The generator takes a class attribute, which defines the mechanism to be used. The class should be an implementation of org.hibernate.id.IdentifierGenerator.
4. Hibernate have the option of using the special **assigned** generator type. This allows you to explicitly set the identifier for the entities that you will be persisting—Hibernate will not then attempt to assign any identifier value to such an entity. If you use this technique, you will not be able to use the saveOrUpdate() method on a transient entity—instead, you will have to call the appropriate save() or update() method explicitly.
5. The child elements of the <property> element are as follows: (meta\*, (column | formula)\*, type?). Any element accepting a column attribute, as is the case for the <property> element, will also accept <column> elements in its place.
6. The <component> element is used to map classes that will be represented as extra columns within a table describing some other class.
7. If your class represents data using a class derived from the List interface, but you do not want to maintain an index column to keep track of the order of items, you can optionally use the bag collection mapping to achieve this. The order in which the items are stored and retrieved from a bag is completely ignored.
8. Session.saveOrUpdate() method is slightly less efficient than the save() method since it may need to perform a SELECT statement to check whether the object already exists, but it will not fail if the object has already been saved.
9. Session.getTransaction() does not return **null** when no transaction is in progress. Instead, the active property of the returned object is **false**.
10. Session.clear() clears the session of all loaded instances and cancels any saves,updates, or deletions that have not been completed. Retains any iterators that are in use while flush() flushes all pending changes into the database—all saves, updates, and deletions will be carried out; essentially, this synchronizes the session with the database.
11. Session.evict() disassociates an object from the session so that subsequent changes to it will not be persisted.
12. Session.isDirty() Determines whether the session is synchronized with the database.
13. The isolation levels permitted by JDBC and Hibernate are as follows:

**0 None:** Anything is permitted; the database or driver does not support transactions.

**1 Read Uncommitted**: Dirty, nonrepeatable, and phantom reads are permitted.

**2 Read Committed**: Nonrepeatable reads and phantom reads are permitted.

**4 Repeatable Read**: Phantom reads are permitted.

**8 Serializable**: The rule must be obeyed absolutely i.e. Acids rules

A dirty read may see the in-progress changes of an uncommitted transaction. A nonrepeatable read sees different data for the same query at the beginning of the transaction and again at the end, and get a different answer both times without making any updates. A phantom read sees different numbers of rows for the same query without making any updates.

1. If we want to treat one particular transaction at a high level of isolation (usually Serializable), while permitting lower degrees of isolation for others. To do so, we need to obtain the JDBC connection directly, alter the isolation level, begin the transaction, roll back or clean.
2. Locking can be acquired for the momentary operation on the data only, or it can be retained until the end of the transaction. The former is called optimistic locking and the latter is called pessimistic locking. The Read Uncommitted isolation level always acquires optimistic locks, whereas the Serializable isolation level will only acquire pessimistic locks.
3. Lock Modes are as follows:

**NONE** Reads from the database only if the object is not available from the caches.

**READ** Reads from the database regardless of the contents of the caches.

**UPGRADE** Obtains a dialect-specific upgrade lock for the data to be accessed (if this is available from your database).

**UPGRADE\_NOWAIT** Behaves like UPGRADE, but when support is available from the database and dialect. The method will fail with a locking exception immediately. Without this option, or on databases for which it is not supported, the query must wait for a lock to be granted(or for a timeout to occur).

An additional lock mode, WRITE, is acquired by Hibernate automatically when it has written to a row within the current transaction. This mode cannot be set explicitly, but calls to getLockMode may return it.

1. There are two caches available to the Hibernate session: the compulsory L1 cache, through which all requests must pass, and the optional level-two (L2) cache. The L1 cache will always be consulted before any attempt is made to locate an object in the L2 cache. The L1 cache ensures that within a session, requests for a given object from a database will always return the same object instance, thus preventing data from conflicting and preventing Hibernate from trying to load an object multiple times. Items in the L1 cache can be individually discarded by invoking the evict() method on the session for the object that you wish to discard. To discard all items in the L1 cache, invoke the clear() method. The L2 cache is external to Hibernate.
2. The type of access to the L2 cache can be configured on a per-session basis by selecting a CacheMode option and applying it with the setCacheMode() method.

**NORMAL** Data is read from and written to the cache as necessary.

**GET** Data is never added to the cache (although cache entries are invalidated when updated by the session).

**PUT** Data is never read from the cache, but cache entries will be updated as they are read from the database by the session.

**REFRESH** This is the same as PUT, but the use\_minimal\_puts Hibernate configuration option will be ignored if it has been set.

**IGNORE** Data is never read from or written to the cache (except that cache entries will still be invalidated when they are updated by the session).

1. A database management system (DBMS) can detect deadlock by two inter dependent tables automatically, at which point the transaction of one or more of the offending processes will be aborted by the database. The resulting deadlock error will be received and handled by Hibernate as a normal HibernateException.
2. An HQL **INSERT** cannot be used to directly insert arbitrary entities—it can only be used to insert entities constructed from information obtained from SELECT queries.
3. If FETCH ALL PROPERTIES is used in SELECT statement , then lazy loading semantics will be ignored, and all the immediate properties of the retrieved object(s) will be actively loaded (this does not apply recursively).
4. In the HQL language grammar, there are the following possible expressions:

• Logic operators: OR, AND, NOT

• Equality operators: =, <>, !=, ^=

• Comparison operators: <, >, <=, >=, like, not like, between, not between

• Math operators: +, -, \*, /

• Concatenation operator: ||

• Collection expressions: some, exists, all, any

• JDBC query parameter: ?

• Date and time SQL-92 functional operators: current\_time(), current\_date(), current\_timestamp()

• SQL functions (supported by the database): length(), upper(), lower(), ltrim(), rtrim(), etc

1. There are two methods on the Query interface for paging: setFirstResult() and setMaxResults(), just as with the Criteria interface. The setFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the setMaxResults() method.
2. The executeUpdate() method returns an int that contains the number of rows affected by the update or delete. Be careful when you use bulk delete with objects that are in relationships. Hibernate will not know that you removed the underlying data in the database, and you can get foreign key integrity errors.
3. We can embed the SQL query into annotations using @NamedNativeQuery and @SqlResultSetMapping:

@NamedNativeQuery(name="com.hibernatebook.queries.Product.SQLpricing",

query="select product.price from Product as product where product.price > 25.0",

resultSetMapping="SQLPricingMapping")

@SqlResultSetMapping(name="SQLPricingMapping", columns=@ColumnResult(name="price"))

1. The one of the restriction is the SQL restriction sqlRestriction(). This restriction allows you to directly specify SQL in the Criteria API. This is useful if you need to use SQL clauses that Hibernate does not support through the Criteria API. Your application’s code does not need to know the name of the table your class uses—use {alias} to signify the class’s table.
2. Hibernate Criteria provides a result transformer for distinct entities, org.hibernate.transform.DistinctRootEntityResultTransformer, which ensures that no duplicates will be in your query’s result set. Rather than using SELECT DISTINCT with SQL, the distinct result transformer compares each of your results using their default hashCode() methods, and only adds those results with unique hash codes to your result set. This may or may not be the result you would expect from an otherwise equivalent SQL DISTINCT query, so be careful with this. An additional performance note: the comparison is done in Hibernate’s Java code, not at the database, so non-unique results will still be transported across the network.
3. To use projections, org.hibernate.criterion.Projection object is added to Criteria object with the setProjection() method. Aggregate functions available through the Projections factory class include the following:

• avg(String propertyName): Gives the average of a property’s value

• count(String propertyName): Counts the number of times a property occurs

• countDistinct(String propertyName): Counts the number of unique values the -property contains

• max(String propertyName): Calculates the maximum value of the property values

• min(String propertyName): Calculates the minimum value of the property values

• sum(String propertyName): Calculates the sum total of the property values

• rowCount(): Calculates the row count

Another use of projections is to retrieve individual properties, rather than entities. Use the property() method on the Projections class to create a Projection for a property.

1. Query-By-Example(QBE): For instance, if we have a user database, we can construct an instance of a user object, set the property values for type and creation date, and then use the Criteria API to run a QBE query. Hibernate will return a result set containing all user objects that match the property values that were set. Behind the scenes, Hibernate inspects the Example object and constructs an SQL fragment that corresponds to the properties on the Example object. Hibernate Example object exclude zero-valued properties with the excludeZeroes() method. We can exclude properties by name with the excludeProperty() method, or exclude nothing (compare for null values and zeroes exactly as they appear in the Example object) with the excludeNone().
2. The advantage of using Hibernate filters is that you can programmatically turn filters on or off in your application code, and your filters are defined in your Hibernate mapping documents for easy maintainability. The major disadvantage of filters is that you cannot create new filters at run time. Instead, any filters your application requires need to be specified in the proper Hibernate annotations or mapping documents. Although this may sound somewhat limiting, the fact that filters can be parameterized makes them pretty flexible.
3. Filters can be defined like:

@Entity

@FilterDef(name="latePaymentFilter", parameters=@ParamDef( name="dueDate", type="date" ) )

@Filter(name="latePaymentFilter", condition=":dueDate = paymentDate")

1. By default, Hibernate will throw an exception if you load a value that doesn't exist in a many-to-one relationship and then try and use it. Hibernate will create a proxy object to represent that value, and if it doesn't exist, it will throw an exception. We can override this default behavior with the @NotFound annotation on the many-to-one relationship, on the same getter method.
2. Optimistic Locking are of following types:

* Versioning: is essentially a type of optimistic locking. When any changes to an entity are stored, a version column is updated to reflect the fact that the entity has changed. When a subsequent user tries to commit changes to the same entity, the original version number will be compared against the current value—if they differ, the commit will be rejected. The <class> element’s optimistic-lock attribute can be used to override the default versioning based optimistic locking strategy.
* Dirty checking: offers an alternative form of optimistic locking. Here, the values of the entities are themselves checked to see if they have changed since the entity was originally obtained. With dirty checking only those fields that have changed since the persistent entity was obtained will be.
* All checking: all the fields comprising the entity will be checked for changes. If the fields being checked have changed prior to the commit, then the commit will fail.

1. Hibernate session can be accessed in one more way: as a map of name/value pairs. This mode is accessed by calling the getSession() method with a parameter of EntityMode.MAP but composite primary key id classes object should be POJO in Hibernate.
2. Hibernate 3 actually implements most of its functionality as event listeners. For example, listener for save-updatecan be defined like:

<listener type="save-or-update" class="com.hibernatebook.advanced.BookingSaveOrUpdateEventListener"/>

Alternatively, a programmatic registration of the same event would be given thus:

Configuration config = new Configuration();

config.setListener("save-update", new BookingSaveOrUpdateEventListener());

1. Interceptors are privy to a blow-by-blow account of what is going on as Hibernate carries out its duties. While we can listen in, we can only make limited changes to the way in which Hibernate actually behaves. Occasionally, we will find that it is necessary to persist a POJO that has no default constructor and source code is not available so Interceptors can be used for applying default values for the parameters of the POJO’s non-default constructor.
2. **LazyInitializationException occurs when you try to access the lazily loaded attributes of a detached entity—typically when the session that loaded it has been closed.** For this, Spring provides a mechanism to implement the OpenSessionInView pattern of behavior. This ensures that the Session object is retained until processing of the view is complete. Only then is it closed—it must be closed at some point to ensure that your web applications don’t leak a Session for every user request! The effect is that with either an OpenSessionInViewInterceptor in your Spring configuration file or an OpenSessionInViewFilter configured in your web.xml file, you can access lazily loaded attributes of entities acquired from your DAOs without any risk of the dreaded LazyInitializationException.
3. We dictate whether a single Session object will be used for the entire duration of the user request. Setting this to true is the most efficient approach, and therefore the default; however, it has potential side effects, particularly if declarative transactions are not in use. When set to false, individual sessions will be acquired for each DAO operation:

<bean name="openSessionInViewInterceptor"

class="org.springframework.orm.hibernate3.support.OpenSessionInViewInterceptor">

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

<property name="singleSession" value="true"/>

</bean>

1. Hibernate exceptions are now thrown as unchecked exceptions from version 3 onwards.