Hibernate.cfg.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<!-- Database connection settings -->
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:./db1</property>
<property name="connection.username">sa</property>
<property name="connection.password"/>
<property name="dialect">org.hibernate.dialect.H2Dialect</property>
<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>
<!-- Drop and re-create the database schema on startup -->
<property name="hbm2ddl.auto">create-drop</property>
<mapping class="chapter01.hibernate.Message"/>
</session-factory>
</hibernate-configuration>
```

connection.driver.class This is the fully qualified name of the JDBC driver class for the session factory.
connection.url This is the JDBC URL used to connect to the database.
connection.username The connection’s username, surprisingly enough.
connection.password Another surprise – the connection’s password. In an uninitialized H2 database, “sa” and an empty password are sufficient.
dialect This property tells Hibernate how to write SQL for the specific database.
show_sql This property sets Hibernate to echo its generated SQL statements to a specified logger.
hbm2ddl.auto This property tells Hibernate whether it should manage the database schema; in this case, we’re telling it to create on initialization and drop the database when it’s done.

PersistenceTest.java

```
package chapter01.hibernate;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;
import java.util.List;
import static org.testng.Assert.assertEquals;
public class PersistenceTest {
private SessionFactory factory = null;
@BeforeClass
public void setup() {
StandardServiceRegistry registry =
new StandardServiceRegistryBuilder()
.configure()
.build();
factory = new MetadataSources(registry)
.buildMetadata()
.buildSessionFactory();
}
public Message saveMessage(String text) {
Message message = new Message(text);
try (Session session = factory.openSession()) {
Transaction tx = session.beginTransaction();
session.persist(message);
tx.commit();
}
return message;
}
@Test
public void readMessage() {
Message savedMessage = saveMessage("Hello, World");
List<Message> list;
try (Session session = factory.openSession()) {
list = session
.createQuery("from Message", Message.class)
.list();
}
assertEquals(list.size(), 1);
for (Message m : list) {
System.out.println(m);
}
assertEquals(list.get(0), savedMessage);
}
}
```

Reading Data

Hibernate Query Language (HQL)

```
private Person findPerson(Session session, String name) {
```

```
Query<Person> query = session.createQuery(
"from Person p where p.name=:name",
Person.class
);
query.setParameter("name", name);
Person person = query.uniqueResult();
return person;
}
```

A Method to Create or Return a Specific Person

```
private Person savePerson(Session session, String name) {
Person person = findPerson(session, name);
if (person == null) {
person = new Person();
person.setName(name);
session.save(person);
}
return person;
}
```

Or

```
try (Session session = factory.openSession()) {
Transaction tx = session.beginTransaction();
createData(session, "J. C. Smell", "Gene Showrama", "Java", 6);
createData(session, "J. C. Smell", "Scottball Most", "Java", 7);
createData(session, "J. C. Smell", "Drew Lombardo", "Java", 8);
tx.commit();
}
```

Updating Data

```
try (Session session = factory.openSession()) {
Transaction tx = session.beginTransaction();
Query<Ranking> query = session.createQuery(
"from Ranking r "
+ "where r.subject.name=:name "
+ "and r.skill.name=:skill", Ranking.class);
query.setParameter("name", subject);
query.setParameter("skill", skill);
int average = (int) stats.getAverage();
tx.commit();
return average;
```

Or

```
try (Session session = factory.openSession()) {
Transaction tx = session.beginTransaction();
Query<Ranking> query = session.createQuery(
"from Ranking r "
+ "where r.subject.name=:subject and "
+ "r.observer.name=:observer and "
+ "r.skill.name=:skill", Ranking.class);
query.setParameter("subject", "J. C. Smell");
query.setParameter("observer", "Gene Showrama");
query.setParameter("skill", "Java");
ranking.setRanking(9);
tx.commit();
```

Removing Data

```
Query<Ranking> query = session.createQuery(
"from Ranking r "
+ "where r.subject.name=:subject and "
+ "r.observer.name=:observer and "
+ " r.skill.name=:skill", Ranking.class);
query.setParameter("subject", subject);
query.setParameter("observer", observer);
query.setParameter("skill", skill);
Ranking ranking = query.uniqueResult();
```

The Persistence LifeCycle
A Typical Identifier Field

```
@Id
public Long id;
```

Вы должны использовать аннотацию @GeneratedValue, если вы не хотите назначать значения идентификатора
самим собой!
Существует пять различных возможностей генерации: идентичность, последовательность, таблица, авто и
никто. Генерация удостоверений опирается на естественную последовательность таблиц. Это запрашивается в
Аннотация @GeneratedValue с использованием параметра GenerationType.IDENTITY

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

public Long id;

Entities and Associations
When only one of the pair of entities contains a reference to the other, the association is unidirectional. If the association is mutual, then it is referred to as being bidirectional.

Saving Entities
public Serializable save(Object object)
public Serializable save(String entityName, Object object)

Loading Entities
public <T> T load(Class<T> theClass, Object id)
public Object load(String entityName, Object id)
public void load(Object object, Object id)

Merging Entities
Merging is performed when you desire to have a detached entity changed to persistent state again, with the detached entity's changes migrated to (or overriding) the database. The method signatures for the merge operations are the following:
Object merge(Object object)
Object merge(String entityName, Object object)

Merging is the inverse of refresh(), which overrides the detached entity's values with the values from the database.

Refreshing Entities
Hibernate provides a mechanism to refresh persistent objects from their database representation, overwriting the values that the in-memory object might have. Use one of the refresh() methods on the Session interface to refresh an instance of a persistent
object, as follows:
public void refresh(Object object)

public void refresh(Object object, LockMode lockMode)

Updating Entities
public void flush() throws HibernateException
public void setHibernateFlushMode(FlushMode flushMode)
public FlushMode getHibernateFlushMode()
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.

Deleting Entities
public void delete(Object object)
public void delete (String entityName, Object object)

Cascading Operations
The cascade types
supported by the Java Persistence API are as follows:
• PERSIST
• MERGE
• REFRESH
• REMOVE
• DETACH
• ALL

• CascadeType.PERSIST means that save() or persist() operations cascade to related entities; for our Email and Message example, if Email's @OneToOne annotation includes PERSIST, saving the Email would save the Message as well.

• CascadeType.MERGE means that related entities are merged into managed state when the owning entity is merged.
• CascadeType.REFRESH does the same thing for the refresh() operation.
• CascadeType.REMOVE removes all related entities association with this setting when the owning entity is deleted.
• CascadeType.DETACH detaches all related entities if a manual detach were to occur.
• CascadeType.ALL is shorthand for all of the cascade operations.

The cascade configuration option accepts an array of CascadeType references; thus, to include only refreshes and merges in the cascade operation for a one-to-one relationship, you might see the following:

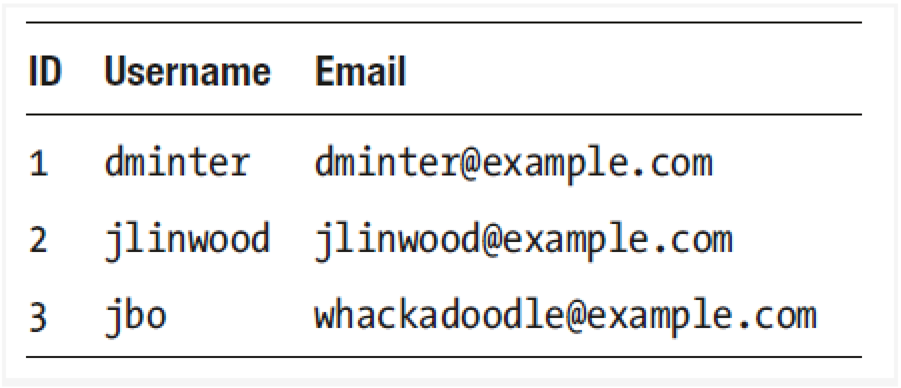
```
@OneToOne(cascade={CascadeType.REFRESH, CascadeType.MERGE})
EntityType otherSide;
```

Lazy Loading, Proxies, and Collection Wrappers

Mapping

The One-to-One Association

A one-to-one association between classes can be represented in a variety of ways. At its simplest, the properties of both classes are maintained in the same table. For example, a one-to-one association between a User and an Email class might be represented as a single table, as in Table



ID	Username	Email
1	dminter	dminter@example.com
2	jlinwood	jlinwood@example.com
3	jbo	whackadoodle@example.com

Alternatively, the entities can be maintained in distinct tables with identical primary keys (as shown here) or with a key maintained from one of the entities into the other, as in Tables 5-6 and 5-7.

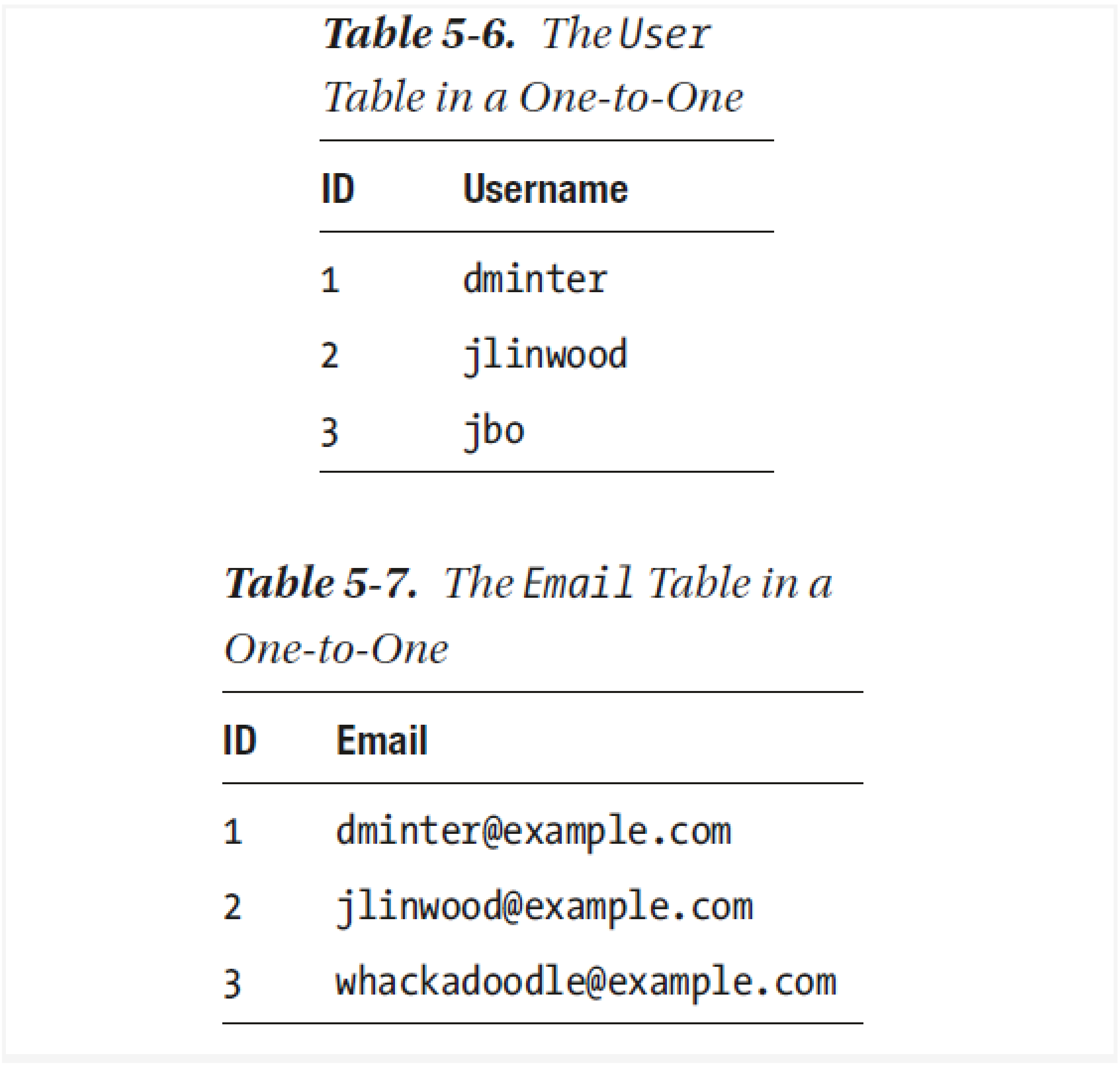


Table 5-6. *The User Table in a One-to-One*

ID	Username
1	dminter
2	jlinwood
3	jbo

Table 5-7. *The Email Table in a One-to-One*

ID	Email
1	dminter@example.com
2	jlinwood@example.com
3	whackadoodle@example.com

The One-to-Many and Many-to-One Associations

A one-to-many association (or from the perspective of the other class, a many-to-one association) can most simply be represented by the use of a foreign key, with no additional constraints. The relationship can also be maintained by the use of a link table. This will maintain a foreign key into each of the associated tables, which will itself form the primary key of the link table. A link table is effectively mandatory for many-to-many relationships, but for relationships that have a cardinality of one on one side of the relationship, link tables tend to be used when the relationship has a state that isn't reflected in the objects themselves (like where in a List something might be), or when the objects should not have an explicit reference to another entity.

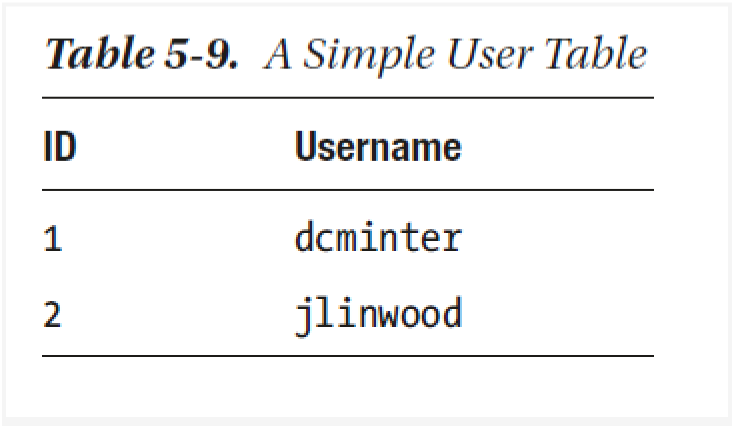


Table 5-9. A Simple User Table

ID	Username
1	dcminter
2	jlinwood

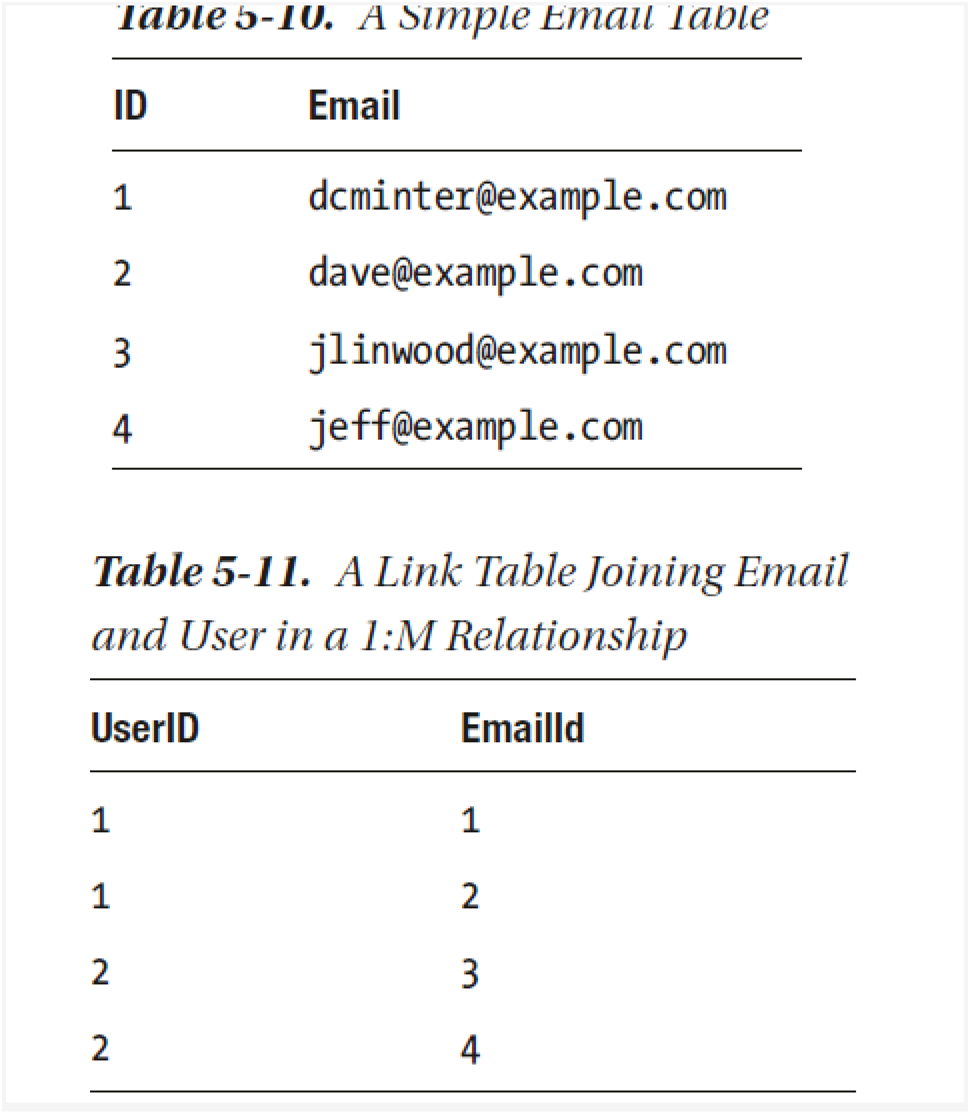


Table 5-10. A Simple Email Table

ID	Email
1	dcminter@example.com
2	dave@example.com
3	jlinwood@example.com
4	jeff@example.com

Table 5-11. A Link Table Joining Email and User in a 1:M Relationship

UserID	EmailId
1	1
1	2
2	3
2	4

The Many-to-Many Association

As noted at the end of the previous section, if a unique constraint is not applied to the “one” end of the relationship when using a link table, it becomes a limited sort of many-to-many relationship. All of the possible combinations of User and Email can be represented, but it is not possible for the same user to have the same email address entity associated twice, because that would require the compound primary key to be duplicated.

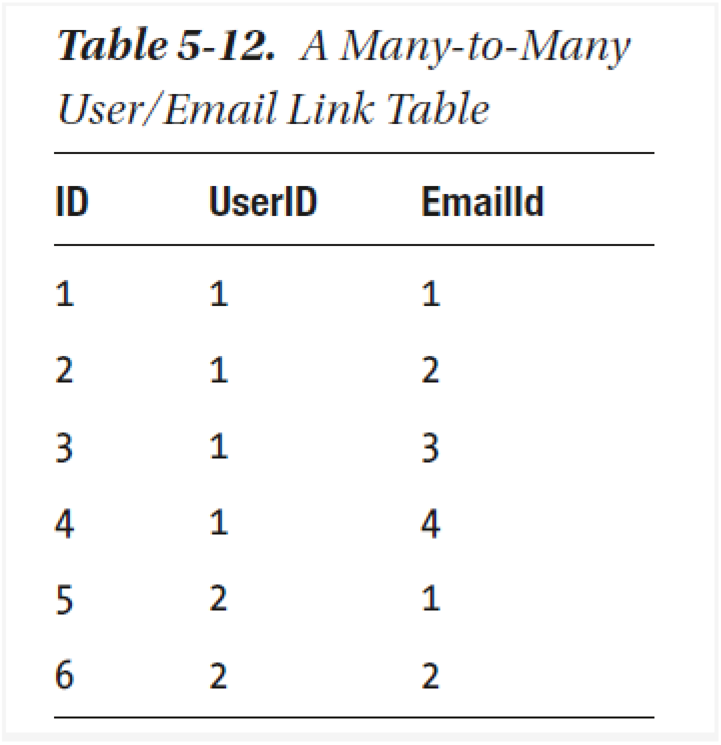


Table 5-12. *A Many-to-Many User/Email Link Table*

ID	UserID	EmailId
1	1	1
2	1	2
3	1	3
4	1	4
5	2	1
6	2	2

Mapping with Annotations
The most common annotations, just for reference’s sake, are @Entity, @Id, and @Column; other common ones we’ll encounter often are @GenerationStrategy (associated with @Id) and the association-related annotations like @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany.

Entity Beans with @Entity
The first step is to annotate the Book class as a JPA 2 entity bean. We add the @Entity annotation to the Book class, as follows:

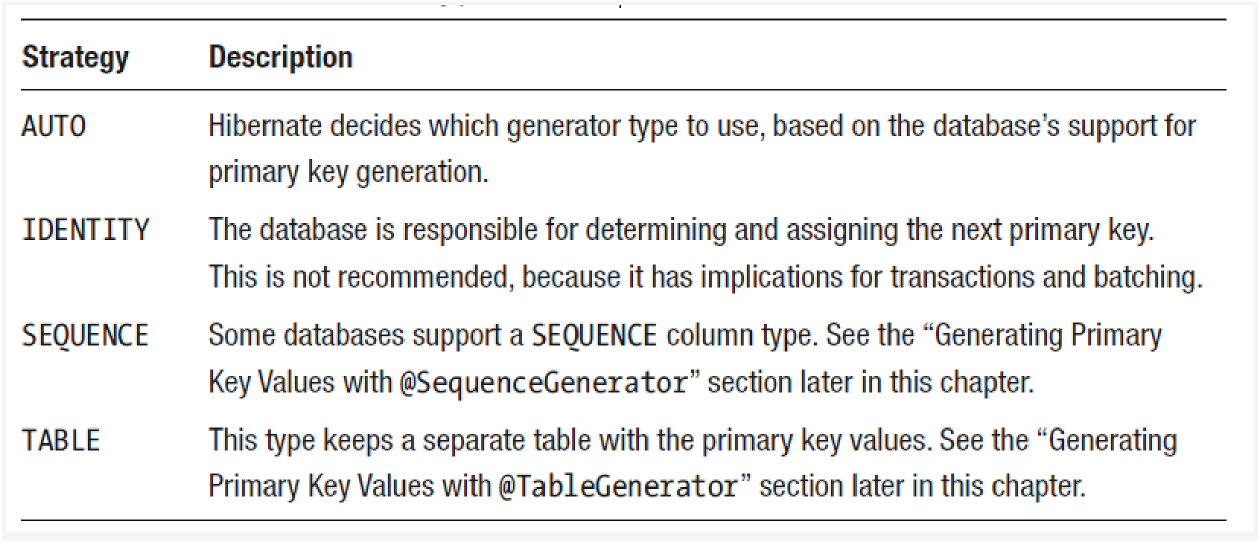
```
@Entity
```

```
public class Book
// Declaration of instance variables goes here
public Book() {
}
```

Primary Keys with @Id and @GeneratedValue
Each entity bean has to have a primary key, which you annotate on the class with the @Id annotation. Typically, the primary key will be a single field, though it can also be a composite of multiple fields.

```
@Id
int id;
```

The GenerationType Options



Strategy	Description
AUTO	Hibernate decides which generator type to use, based on the database's support for primary key generation.
IDENTITY	The database is responsible for determining and assigning the next primary key. This is not recommended, because it has implications for transactions and batching.
SEQUENCE	Some databases support a SEQUENCE column type. See the "Generating Primary Key Values with @SequenceGenerator" section later in this chapter.
TABLE	This type keeps a separate table with the primary key values. See the "Generating Primary Key Values with @TableGenerator" section later in this chapter.

Generating Primary Key Values with @SequenceGenerator
мы можем объявить свойство первичного ключа как генерируется последовательностью базы данных. Последовательность — это объект базы данных, который можно использовать как источник значений первичного ключа. Это похоже на использование типа столбца идентификаторов, за исключением что последовательность не зависит от какой-либо конкретной таблицы и поэтому может использоваться несколько таблиц. Полезна ли возможность использования таблицы последовательностей для нескольких идентификаторов? Это зависит.

Генератор последовательности лучше всего работает при интенсивном доступе, потому что он выделяет идентификаторы в блоках, поэтому при выделении новых ключей нет шансов на столкновение, поэтому во многих случаях это отличный выбор. Однако это не гарантирует, что вы получите серию предсказуемых идентификаторов, потому что после того, как блок был выделен, блок чисел больше не доступен для любого другого поколения идентификаторов.

```
@Id
@SequenceGenerator(name="seq1",sequenceName="HIB_SEQ")
@GeneratedValue(strategy=SEQUENCE,generator="seq1")
int id;
```

Generating Primary Key Values with @TableGenerator
The @TableGenerator annotation is used in a very similar way to the @SequenceGenerator annotation, but because @TableGenerator manipulates a standard database table to obtain its primary key values, instead of using a vendor-specific sequence object, it is guaranteed to be portable between database platforms. Генерирует таблицу

```
@Id
@TableGenerator(name="tablegen",
table="ID_TABLE",
pkColumnName="ID",
valueColumnName="NEXT_ID")
@GeneratedValue(strategy=TABLE,generator="tablegen")
int id;
```

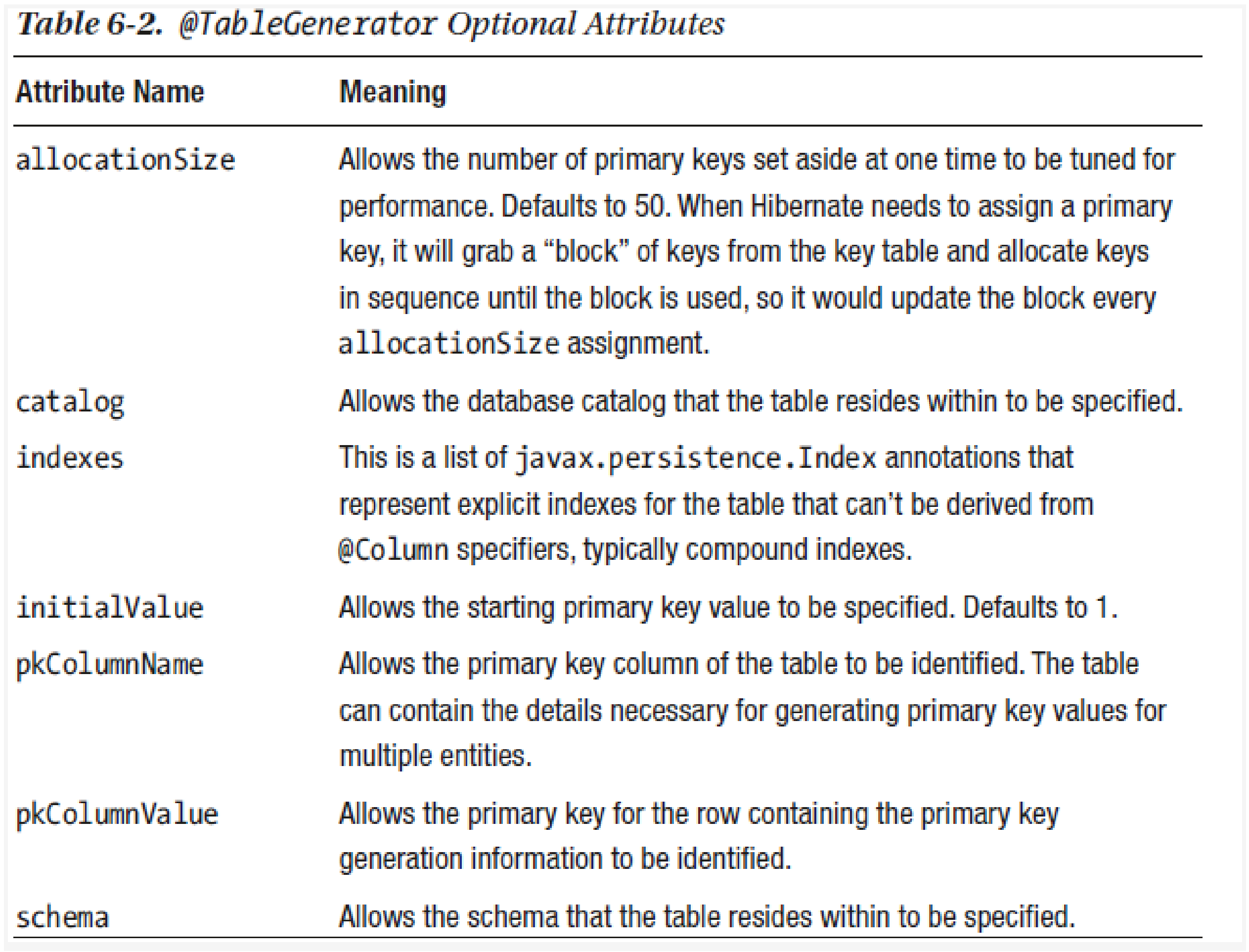
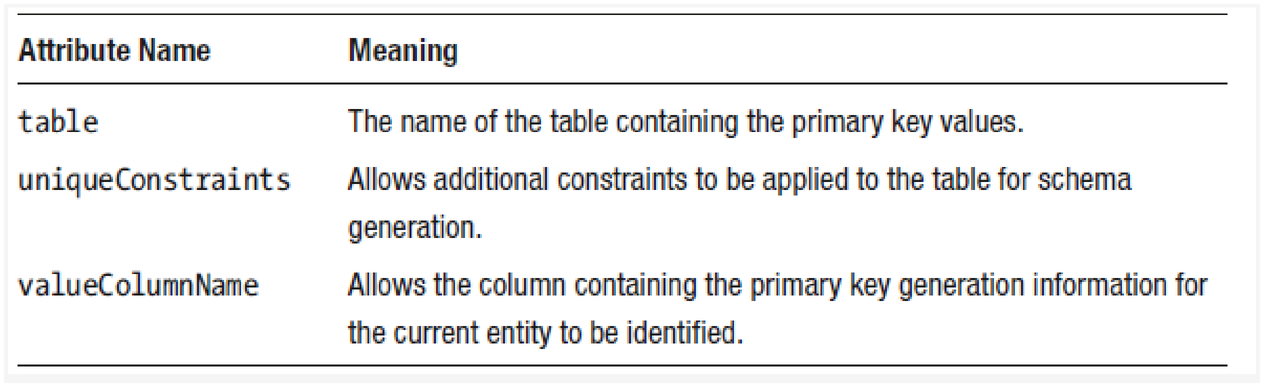


Table 6-2. @TableGenerator Optional Attributes

Attribute Name	Meaning
`allocationSize`	Allows the number of primary keys set aside at one time to be tuned for performance. Defaults to 50. When Hibernate needs to assign a primary key, it will grab a "block" of keys from the key table and allocate keys in sequence until the block is used, so it would update the block every `allocationSize` assignment.
`catalog`	Allows the database catalog that the table resides within to be specified.
`indexes`	This is a list of `javax.persistence.Index` annotations that represent explicit indexes for the table that can't be derived from `@Column` specifiers, typically compound indexes.
`initialValue`	Allows the starting primary key value to be specified. Defaults to 1.
`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.



Attribute Name	Meaning
`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.

Compound Primary Keys with @Id, @IdClass, or @EmbeddedId
Your three strategies for using this primary key class once it has been created are as
follows:
1. Mark it as @Embeddable and add it to your entity class as if it were a normal attribute, marked with @Id.

2. Add it to your entity class as if it were a normal attribute, marked
with @EmbeddableId.
3. 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.

```
@Embeddable
public class ISBN implements Serializable {
@Column(name = "group_number")
int group;
int publisher;
int title;
int checkDigit;
public ISBN() {
}
```

Our last example uses the @IdClass, which is very similar to the
@EmbeddedId
example. With @IdClass, the entity has fields that match the id class'
definition, all
marked with @Id in the entity; the key class is used for Session methods like
get() and
load(), which require single objects to look up classes by key.

```
@Entity
@IdClass(IdClassBook.EmbeddedISBN.class)
public class IdClassBook {
```

The @IdClass annotation uses the fully scoped name for the EmbeddedISBN
class,
whose simple name matches the EmbeddedISBN class from
EmbeddedPKBook. This is an
example of the scoping of the class name in question; the fully qualified
names of the key
classes are different even though the simple names are not.

The key class' fields must match the entity to which they apply; you can't use different names or types here.

Database Table Mapping with @Table and @SecondaryTable
The table name can be customized further, and other database-related attributes can be configured via the @Table annotation. This annotation allows you to specify many of the details of the table that will be used to persist the entity in the database.
@Table(name="ORDER_HISTORY").
Аннотация @SecondaryTable предоставляет способ моделирования компонента управления данными, который
сохраняется в нескольких разных таблицах базы данных.

```
@Entity
@Table(
name = "customer",
uniqueConstraints = {@UniqueConstraint(columnNames = "name")}
)
@SecondaryTable(name = "customer_details")
public class Customer {
```

Persisting Basic Types with @Basic
By default, properties and instance variables in your POJO are persistent; Hibernate will store their values for you. The simplest mappings are therefore for the "basic" types. These include primitives, primitive wrappers, arrays of primitives or wrappers, enumerations, and any types that implement Serializable but are not themselves mapped entities.
This is EAGER by default, but can be set to LAZY to permit loading on access of the value.

Omitting Persistence with @Transient
Some fields, such as calculated values, may be used at runtime only, and they should be discarded from objects as when the entities are persisted into the database. The JPA specification provides the @Transient annotation for these

transient fields. The @Transient annotation does not have any attributes – you just add it to the instance variable or the getter method as appropriate for the entity bean's property access strategy.

```
@Transient
Date publicationDate;
```

Mapping Properties and Fields with @Column

The @Column annotation is used to specify the details of the column to which a field or property will be mapped.

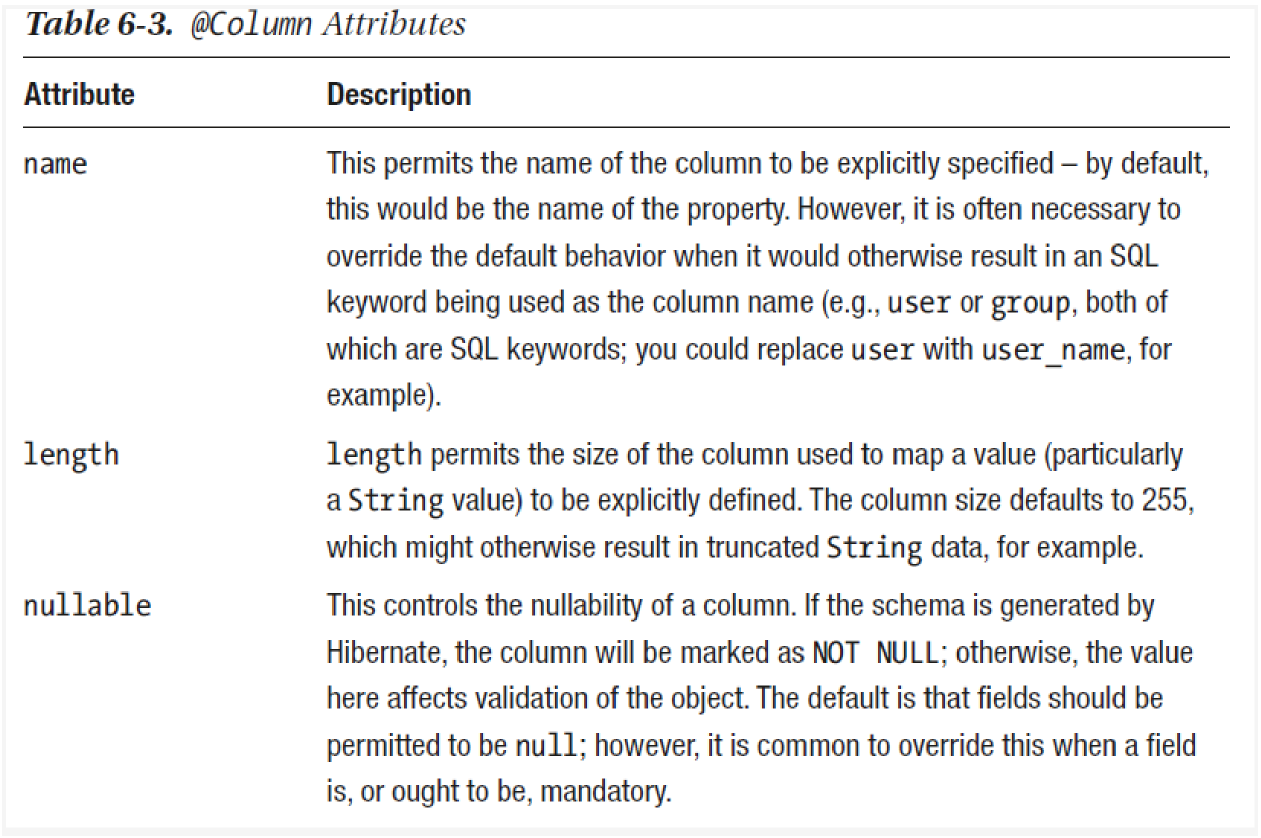
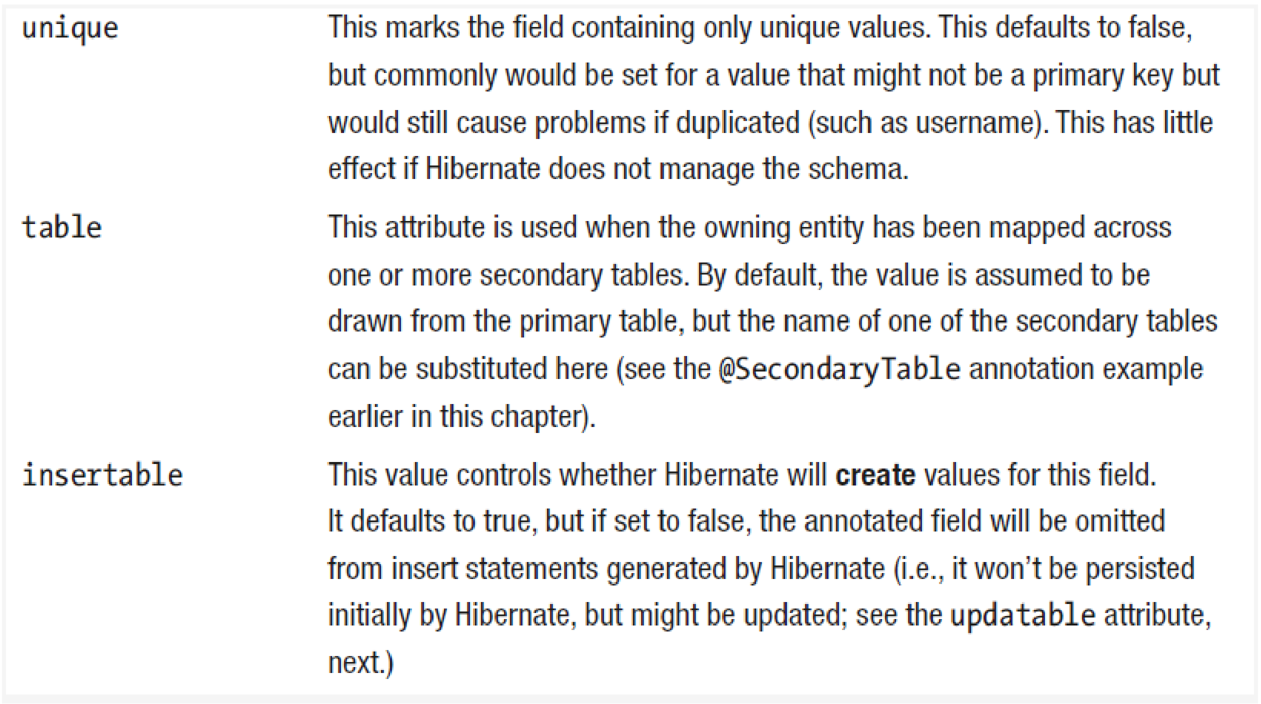
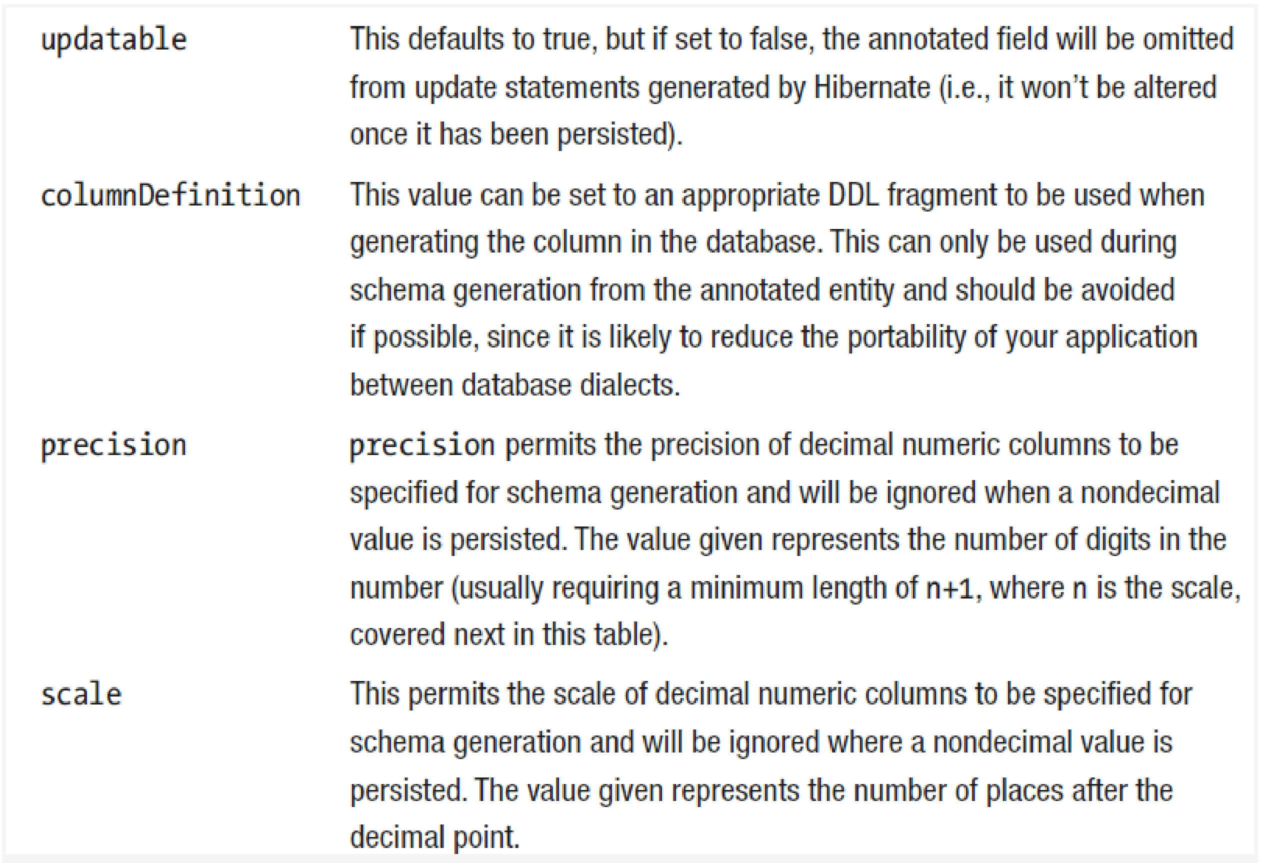


Table 6-3. *@Column Attributes*

Attribute	Description
`name`	This permits the name of the column to be explicitly specified – by default, this would be the name of the property. However, it is often necessary to override the default behavior when it would otherwise result in an SQL keyword being used as the column name (e.g., `user` or `group`, both of which are SQL keywords; you could replace `user` with `user_name`, for example).
`length`	`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, which might otherwise result in truncated `String` data, for example.
`nullable`	This controls the nullability of a column. If the schema is generated by Hibernate, the column will be marked as `NOT NULL`; otherwise, the value here affects validation of the object. The default is that fields should be permitted to be `null`; however, it is common to override this when a field is, or ought to be, mandatory.



`unique`	This marks the field containing only unique values. This defaults to false, but commonly would be set for a value that might not be a primary key but would still cause problems if duplicated (such as username). This has little effect if Hibernate does not manage the schema.
`table`	This attribute 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 (see the `@SecondaryTable` annotation example earlier in this chapter).
`insertable`	This value controls whether Hibernate will **create** values for this field. It 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 initially by Hibernate, but might be updated; see the `updatable` attribute, next.)



`updatable`	This 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`	This value 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`	`precision` permits the precision of decimal numeric columns to be specified for schema generation and will be ignored when a nondecimal 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, covered next in this table).
`scale`	This permits the scale of decimal numeric columns to be specified for schema generation and will be ignored where a nondecimal value is persisted. The value given represents the number of places after the decimal point.

```
@Column(name="working_title",length=200,nullable=false)
String title;
```

Modeling Entity Relationships

Mapping an Embedded (Component) One-to-One Association

```
@Embedded
@AttributeOverrides({
@AttributeOverride(name="address",column=@Column(name="ADDR")),
@AttributeOverride(name="country",column=@Column(name="NATION"))
})
AuthorAddress address;
```

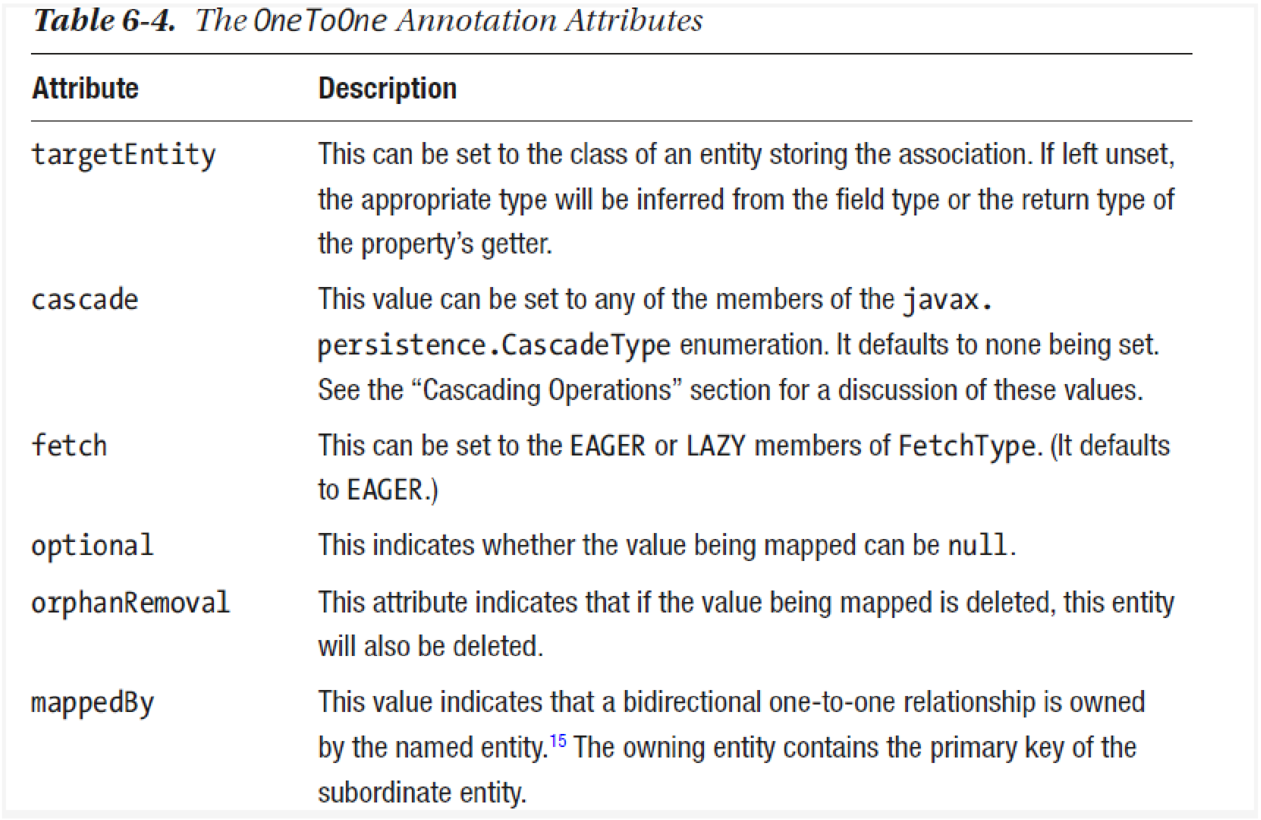


Table 6-4. The OneToOne Annotation Attributes

Attribute	Description
`targetEntity`	This 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`	This value can be set to any of the members of the `javax.persistence.CascadeType` enumeration. It defaults to none being set. See the "Cascading Operations" section for a discussion of these values.
`fetch`	This can be set to the `EAGER` or `LAZY` members of `FetchType`. (It defaults to `EAGER`.)
`optional`	This indicates whether the value being mapped can be `null`.
`orphanRemoval`	This attribute indicates that if the value being mapped is deleted, this entity will also be deleted.
`mappedBy`	This value indicates that a bidirectional one-to-one relationship is owned by the named entity.[15] The owning entity contains the primary key of the subordinate entity.

Mapping a Many-to-One or One-to-Many Association

```
@OneToMany(cascade = ALL,mappedBy = "publisher")
Set<Book> books;
```

The many-to-one end of this relationship is expressed in similar terms to the
one-to-many
end, as shown here:

```
@ManyToOne
@JoinColumn(name = "publisher_id")
Publisher publisher;
```

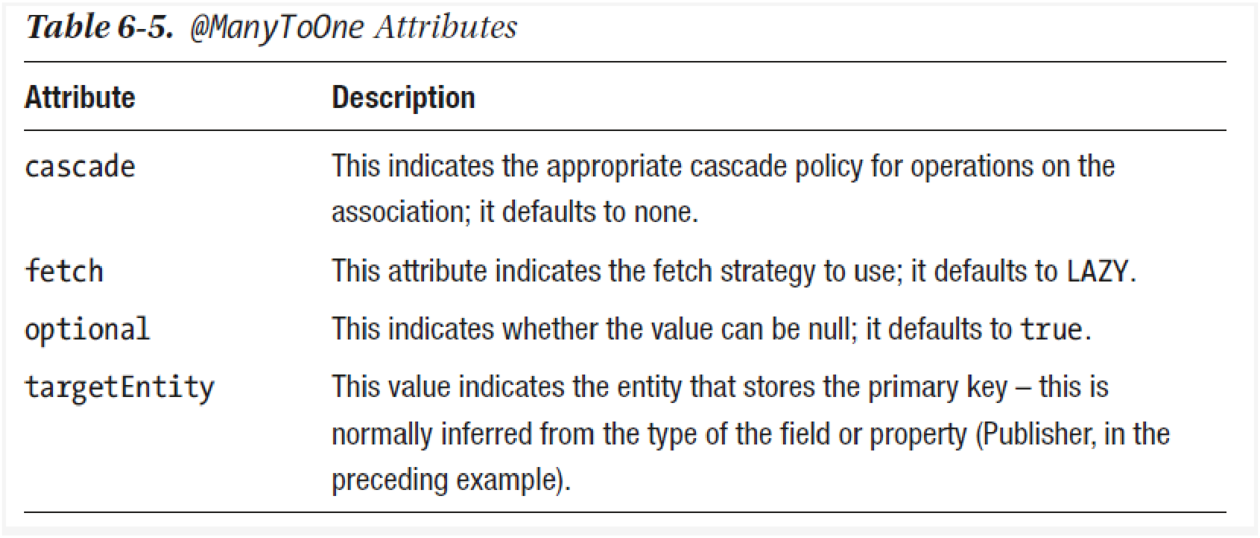


Table 6-5. @ManyToOne Attributes

Attribute	Description
`cascade`	This indicates the appropriate cascade policy for operations on the association; it defaults to none.
`fetch`	This attribute indicates the fetch strategy to use; it defaults to LAZY.
`optional`	This indicates whether the value can be null; it defaults to `true`.
`targetEntity`	This value indicates the entity that stores the primary key – this is normally inferred from the type of the field or property (Publisher, in the preceding example).

We have also supplied the optional @JoinColumn attribute to name the foreign key column required by the association something other than the default (publisher) – this is not necessary, but it illustrates the use of the annotation.

```
@OneToMany(cascade = ALL)
@JoinTable
Set<Book> books;
```

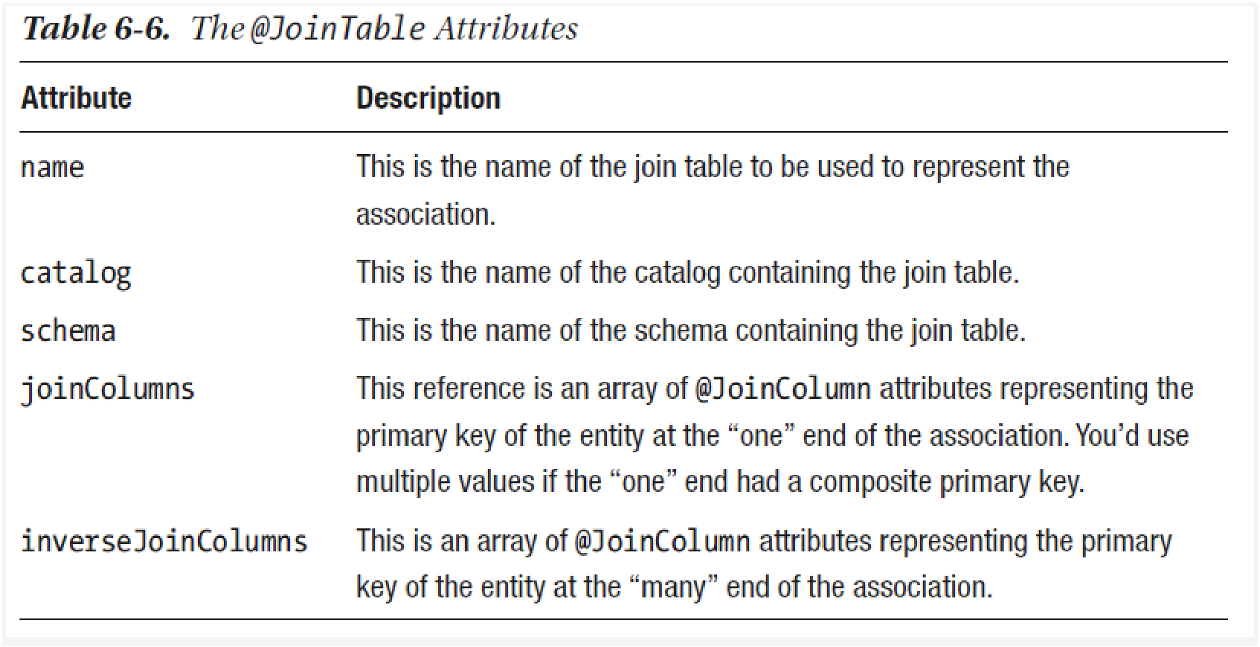


Table 6-6. The @JoinTable Attributes

Attribute	Description
`name`	This is the name of the join table to be used to represent the association.
`catalog`	This is the name of the catalog containing the join table.
`schema`	This is the name of the schema containing the join table.
`joinColumns`	This reference is an array of `@JoinColumn` attributes representing the primary key of the entity at the "one" end of the association. You'd use multiple values if the "one" end had a composite primary key.
`inverseJoinColumns`	This 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")
)
Set<Book> books;
```

Mapping a Many-to-Many Association

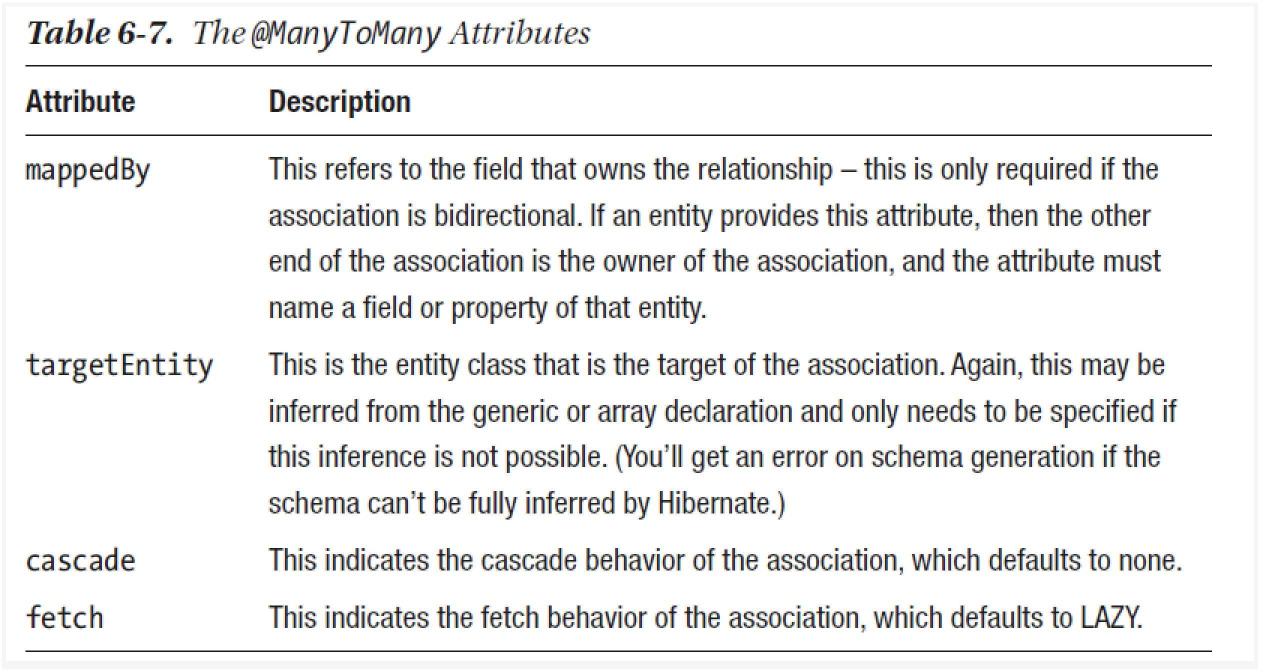


Table 6-7. *The @ManyToMany Attributes*

Attribute	Description
`mappedBy`	This refers to 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`	This 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 inference is not possible. (You'll get an error on schema generation if the schema can't be fully inferred by Hibernate.)
`cascade`	This indicates the cascade behavior of the association, which defaults to none.
`fetch`	This indicates the fetch behavior of the association, which defaults to LAZY.

```
@ManyToMany(cascade = ALL)
Set<Author> authors;
@ManyToMany(mappedBy = "authors")
Set<Book> books;
```

Cascading Operations
following rough correspondence with operations on entities:
• 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.

```
@OneToOne(cascade=CascadeType.ALL)
Address address;
```

Collection Ordering - упорядочивание

```
@OneToMany(cascade = ALL, mappedBy = "publisher")
@OrderBy("name ASC")
List<Book> books;
```

Inheritance

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

Single Table

The single-table approach manages one database table for the superclass and all its subtypes.

```
@Entity(name="SingleBook")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Book {
```

Joined Table

An alternative to the monolithic single-table approach is the otherwise similar joined-table
approach. Here, a discriminator column is used, but the fields of the various derived types are stored in distinct tables.

```
@Entity(name="JoinedBook")
@Inheritance(strategy = InheritanceType.JOINED)
public class Book {
```

Table per Class

Finally, there is the table-per-class approach, in which all of the fields of each type in the inheritance hierarchy are stored in distinct tables.

```
@Entity(name="PerClassBook")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Book {
```

Other JPA 2 Persistence Annotations

Temporal Data

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. The annotation accepts a single value attribute from the javax.persistence. TemporalType enumeration. This offers three possible values: DATE, TIME, and TIMESTAMP. These correspond, respectively, to java.sql.Date, java.sql.Time, and java.sql.Timestamp. The table column is given the appropriate data type at schema generation time.

```
@Temporal(TemporalType.TIME)
java.util.Date startingTime;
```

Element Collections

In addition to mapping collections using one-to-many mappings, JPA 2 introduced an @ElementCollection annotation for mapping collections of basic or embeddable classes, like List or Set.

```
@ElementCollection
List<String> passwordHints;
```

Large Objects
A persistent property or field can be marked for persistence as a database-supported large object type by applying the @Lob annotation.

```
@Lob
String title; // a very, very long title indeed
```

Mapped Superclasses
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.

```
@MappedSuperclass
public class Book {
```

Ordering Collections with @OrderColumn
While @OrderBy allows data to be ordered once it has been retrieved from the database, JPA 2 also provides an annotation that allows the ordering of appropriate collection types (e.g., List) to be maintained in the database, as opposed to being ordered at retrieval; it does so by maintaining an order column to represent that order. Here’s an example:

```
@OneToMany
@OrderColumn(
name="employeeNumber"
)
List<Employee> employees;
```

Named Queries (HQL or JPQL)
@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.

```
@Entity
@NamedQuery(
name="findAuthorsByName",
query="from Author where name = :author"
)
public class Author {
```

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.21 Listing 6-22 gives an example of this.
Listing 6-22. A package-info.java File

```
@javax.annotations.NamedQuery(
name="findBooksByAuthor",
query="from Book b where b.author.name = :author"
)
package chapter06.annotations;
```

И теперь его вызов
Hibernate's Session allows named queries to be accessed directly, as shown in Listing 6-23. Listing 6-23. Invoking a Named Query via the Session

```
Query query = session.getNamedQuery("findBooksByAuthor", Book.class);
query.setParameter("author", "Dave");
List<Book> booksByDave = query.list();
System.out.println("There is/are " + booksByDave.size()
+ " books by Dave in the catalog");
```

Named Native Queries (SQL)
The @NamedNativeQuery annotation is declared in almost exactly the same manner as the @NamedQuery annotation. The following block of code shows a simple example of the declaration of a named native query:

```
@NamedNativeQuery(
name="nativeFindAuthorNames",
query="select name from author"
)
```

Configuring the Annotated Classes

Once you have an annotated class, you will need to provide the class to your application’s Hibernate configuration, just as if it were an XML mapping.

```
<hibernate-configuration>
<session-factory>
<!-- Database connection settings -->
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:./db6</property>
<property name="connection.username">sa</property>
<property name="connection.password"/>
<property name="dialect">org.hibernate.dialect.H2Dialect</property>
<!-- set up c3p0 for use -->
<property name="c3p0.max_size">10</property>
<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>
<!-- Drop and re-create the database schema on startup -->
<property name="hbm2ddl.auto">create</property>
<mapping class="chapter06.primarykey.after.Book"/>
<mapping class="chapter06.compoundpk.CPKBook"/>
<mapping class="chapter06.compoundpk.EmbeddedPKBook"/>
<mapping class="chapter06.compoundpk.IdClassBook"/>
<mapping class="chapter06.twotables.Customer"/>
</session-factory>
</hibernate-configuration>
```

You can also add an annotated class to your Hibernate configuration programmatically. The annotation toolset comes with an org.hibernate.cfg. AnnotationConfiguration object that extends the base Hibernate Configuration object for adding mappings. The methods on AnnotationConfiguration for adding annotated classes to the configuration are as follows:

```
addAnnotatedClass(Class persistentClass) throws MappingException
addAnnotatedClasses(List<Class> classes)
```

addPackage(String packageName) throws MappingException

Hibernate-Specific Persistence Annotations

@Immutable

The @org.hibernate.annotations.Immutable annotation marks an entity as being, well, immutable. This is useful for situations in which your entity represents reference data – things like lists of states, genders, or other rarely mutated data.

Natural IDs

The first part of this chapter spent a lot of pages discussing primary keys, including generated values. Generated values are referred to as “artificial primary keys” and are very much recommended23 as a sort of shorthand reference for a given row. This might be annotated with @Column(unique=true, nullable=false, updatable=false), However, there’s also the concept of a “natural ID,” which provides another convenient way to refer to an entity, apart from an artificial or composite primary key.

```
@Entity
public class SimpleNaturalIdEmployee {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
Integer id;
@NaturalId
Integer badge;
```

JPA Integration and Lifecycle Events

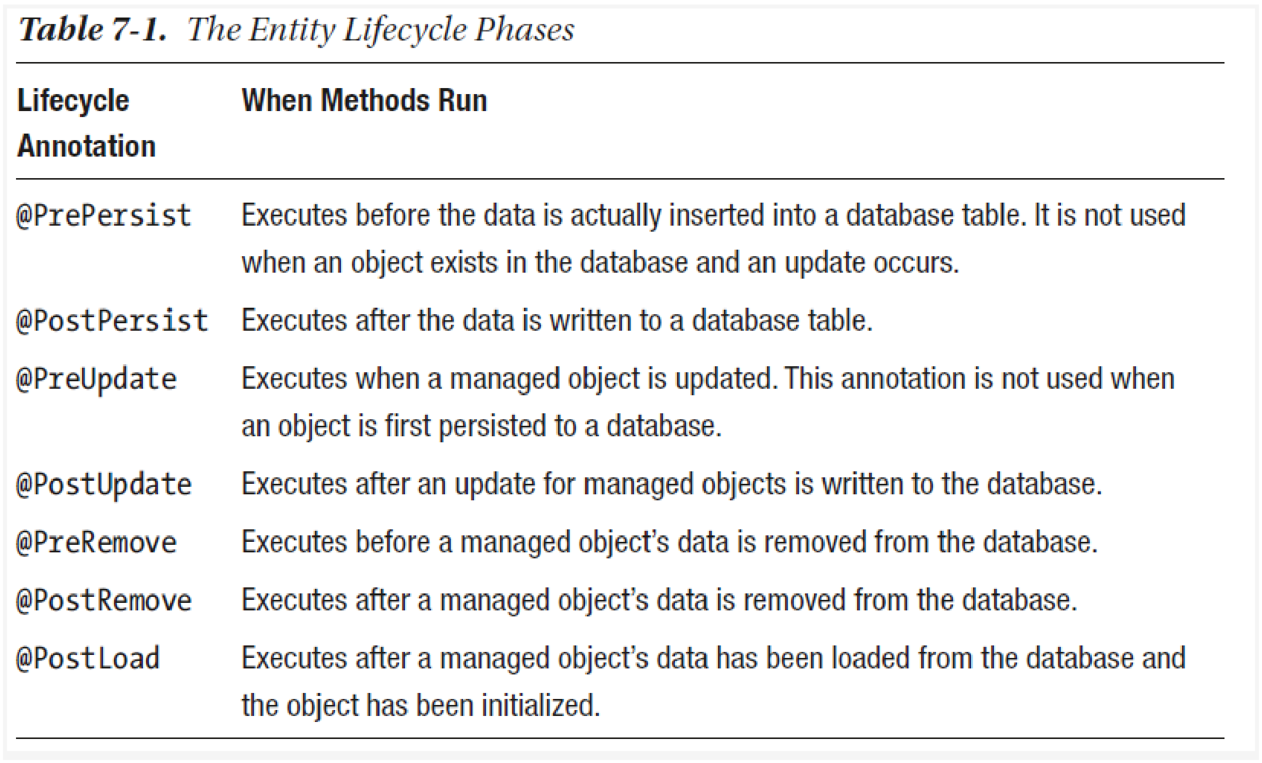


Table 7-1. The Entity Lifecycle Phases

Lifecycle Annotation	When Methods Run
`@PrePersist`	Executes before the data is actually inserted into a database table. It is not used when an object exists in the database and an update occurs.
`@PostPersist`	Executes after the data is written to a database table.
`@PreUpdate`	Executes when a managed object is updated. This annotation is not used when an object is first persisted to a database.
`@PostUpdate`	Executes after an update for managed objects is written to the database.
`@PreRemove`	Executes before a managed object's data is removed from the database.
`@PostRemove`	Executes after a managed object's data is removed from the database.
`@PostLoad`	Executes after a managed object's data has been loaded from the database and the object has been initialized.

Data validation

```
ValidatedPerson person=ValidatedPerson.builder()
.age(15)
.fname("Johnny")
.lname("McYoungster")
.build();
```

Using the Session

Table 8-1. Session Methods for Create, Read, Update, Delete

Method	Description
`save()`	Saves an object to the database. This should not be called for an object that has already been saved to the database.
`saveOrUpdate()`	Saves an object to the database or updates the database if the object already exists. This 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.
`merge()`	Merges the fields of a nonpersistent object into the appropriate persistent object (determined by ID). If no such object exists in the database, then one is created and saved.
`persist()`	Reassociates an object with the session so that changes made to the object will be persisted.
`get()`	Retrieves a specific object from the database by the object's identifier.
`getEntityName()`	Retrieves the entity name (this will usually be the same as the fully qualified class name of the POJO).
`getIdentifier()`	Determines the identifier – the object(s) representing the primary key – for a specific object associated with the session.
`load()`	Loads an object from the database by the object's identifier (you should use the `get()` methods if you are not certain that the object is in the database and you don't want to trap an exception).
`refresh()`	Refreshes the state of an associated object from the database.
`update()`	Updates the database with changes to an object.
`delete()`	Deletes an object from the database.
`createFilter()`	Creates a filter (a selection criterion) to narrow operations on the database.
`enableFilter()`	Enables a named filter in queries produced by `createFilter()`.

Table 8-1. (continued)

Method	Description
`disableFilter()`	Disables a named filter.
`getEnabledFilter()`	Retrieves a currently enabled filter object.
`createQuery()`	Creates a Hibernate query to be applied to the database.
`getNamedQuery()`	Retrieves a query from the mapping file.
`cancelQuery()`	Cancels execution of any query currently in progress from another thread. This does not necessarily dictate what resources are freed or when; the database might still attempt to fulfill the query despite cancellation, for example.
`createCriteria()`	Creates a criteria object for narrowing search results.

Table 8-2. *Session Methods for Transactions and Locking*

Method	Description
`beginTransaction()`	Begins a transaction.
`getTransaction()`	Retrieves the current transaction object. This does not return `null` when no transaction is in progress. Instead, the `active` property of the returned object is `false`.
`lock()`	Gets a database lock for an object (or can be used like `merge()` if `LockMode.NONE` is given). In effect, this method checks the status of an object in the database as compared to the object in memory.

Table 8-3. *Session Methods for Managing Resources*

Method	Description
`contains()`	Determines whether a specific object is associated with the database.
`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.

Method	Description
`evict()`	Disassociates an object from the session so that subsequent changes to it will not be persisted.
`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. This still takes place in the context of a transaction, however, so its usefulness might be limited depending on the kinds of transactions in use.
`isOpen()`	Determines whether the session has been closed.
`isDirty()`	Determines whether the session is synchronized with the database; it will be `true` if the session has not written changes in memory to the database tables.
`getCacheMode()`	Determines the caching mode currently employed.
`setCacheMode()`	Changes the caching mode currently employed.
`getCurrentLockMode()`	Determines the locking mode currently employed for a specific object. (It can be set with the `lock()` method, e.g., among many other options.)
`setFlushMode()`	Determines the approach to flushing currently used. The options are to flush after every operation, flush when needed, never flush, or flush only on commit.
`setReadOnly()`	Marks a persistent object as read-only (or as writable). There are minor performance benefits from marking an object as read-only, but changes to its state will be ignored until it is marked as writable.
`close()`	Closes the session and, hence, the underlying database connection; releases other resources (such as the cache). You must not perform operations on the `Session` object after calling `close()`.
`getSessionFactory()`	Retrieves a reference to the `SessionFactory` object that created the current `Session` instance.

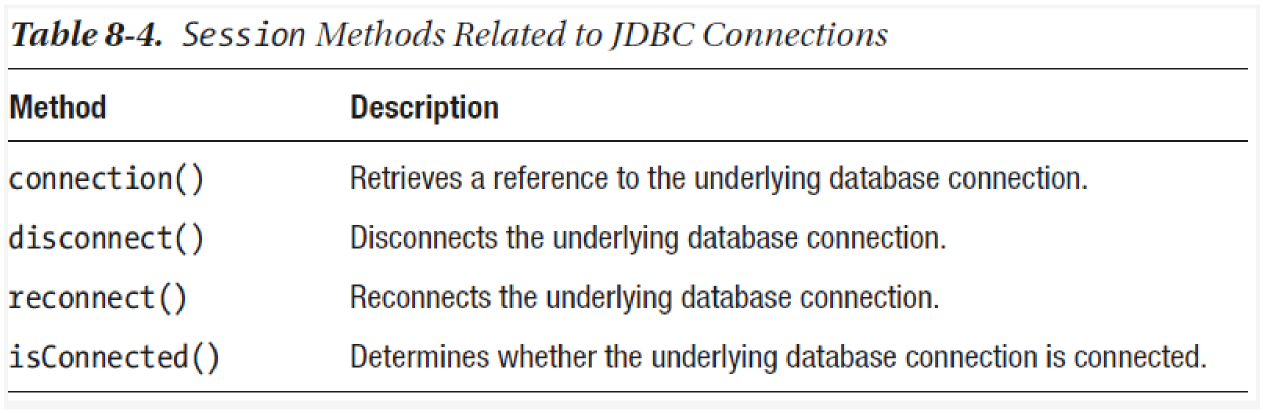


Table 8-4. Session Methods Related to JDBC Connections

Method	Description
`connection()`	Retrieves a reference to the underlying database connection.
`disconnect()`	Disconnects the underlying database connection.
`reconnect()`	Reconnects the underlying database connection.
`isConnected()`	Determines whether the underlying database connection is connected.

Transactions and Locking

```
try(Session session = factory.openSession()) {
session.beginTransaction();
// Normal session usage here?
session.getTransaction().commit();
} catch (HibernateException e) {
Transaction tx = session.getTransaction();
if (tx.isActive()) tx.rollback();
}
```

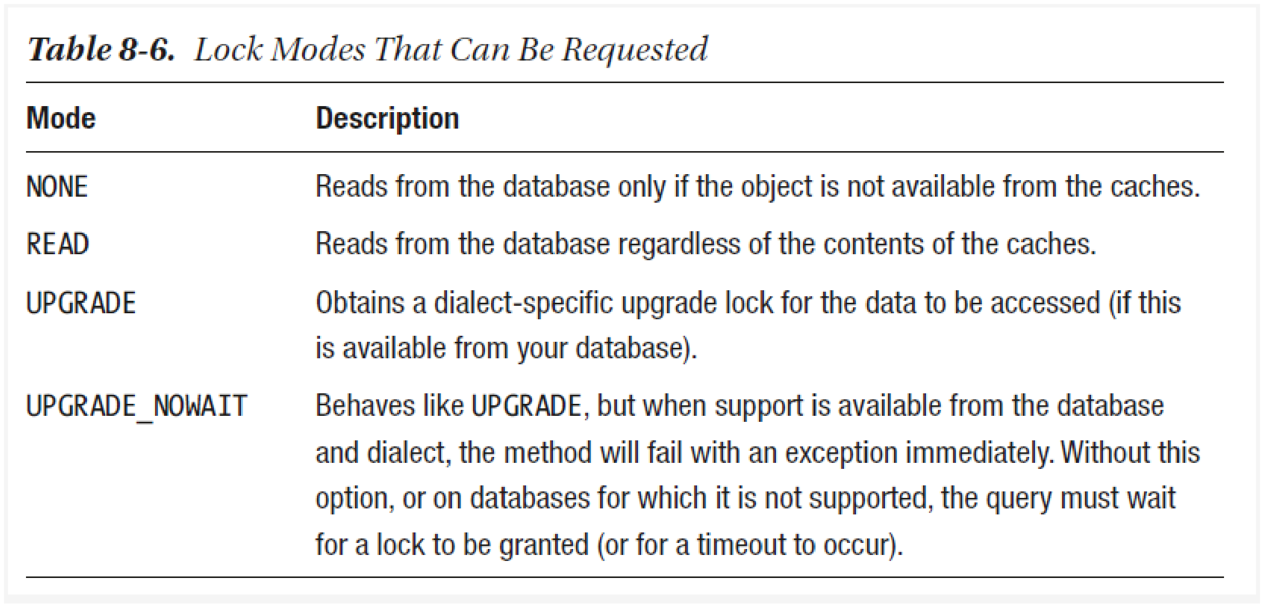


Table 8-6. Lock Modes That Can Be Requested

Mode	Description
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 an 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).

Caching

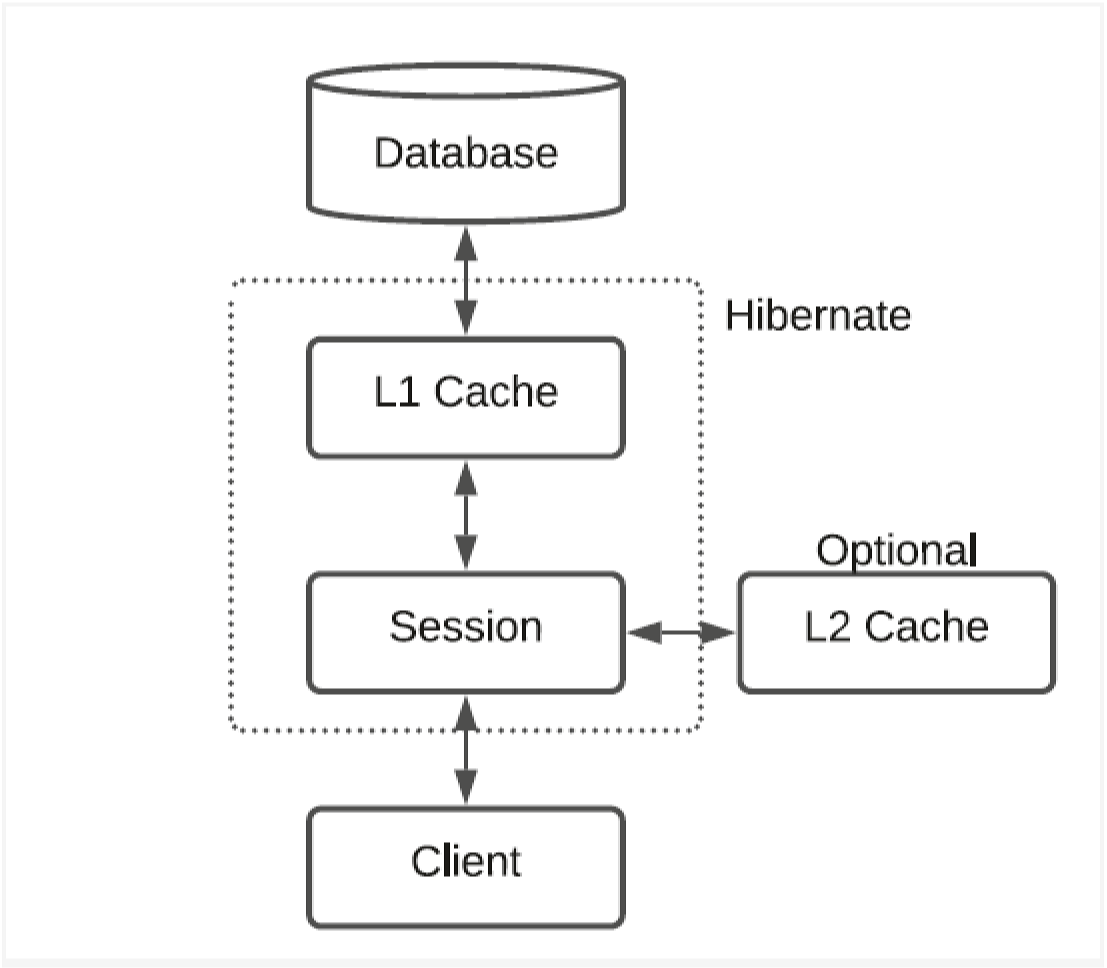


Table 8-7. *CacheMode Options*

Mode	Description
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, in case another `Session` has cached them somehow).

@Entity

```
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
@Data
public class Supplier implements Serializable {
```

Threads

Having considered the caches available to a Hibernate application, you may now be concerned about the risk of a conventional Java deadlock if two threads of execution were to contend for the same object in the Hibernate session cache.

```
try(Session session=SessionUtil.getSession()) {
Transaction tx=session.beginTransaction();
operationOne(session);
operationTwo(session);
operationThree(session);
tx.commit();
}
```

Searches and HQL Queries

HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. It is a superset of the JPQL, the Java Persistence Query Language; a JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries.

UPDATE

```
UPDATE [VERSIONED]
[FROM] path [[AS] alias] [, ...]
SET property = value [, ...]
[WHERE logicalExpression]
```

```
Query query=session.createQuery(
"update Person p set p.creditscore=:creditscore where p.name=:name"
);
query.setInteger("creditscore", 612);
query.setString("name", "John Q. Public");
int modifications=query.executeUpdate();
```

```
DELETE
DELETE
[FROM] path [[AS] alias]
[WHERE logicalExpression]
```

```
Query query=session.createQuery(
"delete from Person p where p.accountstatus=:status
");
query.setString("status", "toBePurged");
int rowsDeleted=query.executeUpdate();
```

```
INSERT
INSERT
INTO path ( property [, ...])
select
```

```
Query query=session.createQuery(
"insert into purged_users(id, name, status) "+
"select id, name, status from users where status=:status"
);
query.setString("status", "toBePurged");
int rowsCopied=query.executeUpdate();
```

```
SELECT
[SELECT [DISTINCT] property [, ...]]
FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]
WHERE logicalExpression
GROUP BY property [, ...]
HAVING logicalExpression
ORDER BY property [ASC | DESC] [, ...]
```

Named Queries (может оформляться в файле)

```
@NamedQuery(name = "supplier.findAll", query = "from Supplier s")
```

Или

```
@NamedQueries({
@NamedQuery(name = "supplier.findAll",
query = "from Supplier s"),
@NamedQuery(name = "supplier.findByName",
query = "from Supplier s where s.name=:name"),
@NamedQuery(name = "supplier.averagePrice",
query = "select p.supplier.id, avg(p.price) " +
"from Product p " +
"GROUP BY p.supplier.id"),
})
```

Using Restrictions with HQL

As with SQL, you use the where clause to select results that match your query's
expressions. HQL provides many different expressions that you can use to construct a
query. In the HQL language grammar, there are many possible expressions,7 including
these:

• Logic operators: OR, AND, NOT
• Equality operators: = (for "equals"), <>, !=, ^= (which mean "not equal")
• Comparison operators: <, >, ⇐, >=, like, not like, between, not between
• Math operators: +, -, *, /
• Concatenation operator: ||
• Cases: Case when <logical expression> then <unary expression> else <unary expression> end
• Collection expressions: some, exists, all, any

Using Named Parameters

Hibernate supports named parameters in its HQL queries.

```
Query<Product> query = session.createQuery(
"from Product where price >= :price",
Product.class);
```

Paging Through the Result Set

```
try (Session session = SessionUtil.getSession()) {
Transaction tx = session.beginTransaction();
session.createQuery("delete from Product").executeUpdate();
session.createQuery("delete from Supplier").executeUpdate();
session.save(supplier);
}
tx.commit();

try (Session session = SessionUtil.getSession()) {
Query<String> query = session.createQuery(
"select s.name from Supplier s order by s.name",
String.class);
query.setFirstResult(4);
query.setMaxResults(4);
List<String> suppliers = query.list();
String list = suppliers
.stream()
.collect(Collectors.joining(","));
assertEquals(list,
"supplier 04,supplier 05,supplier 06,supplier 07");
}
```

Using Native SQL

Filtering the Results of Searches

To use filters with annotations, you will need to use the @FilterDef, @ParamDef, and @Filter annotations. The @FilterDef annotation defines the filter and belongs to either the class or the package. To define a filter on a class, add a @FilterDef annotation alongside the @Entity annotation.

```
@Entity
@FilterDef(name = "byStatus", parameters = @ParamDef(name = "status",
type = "boolean"))
@Filter(name = "byStatus", condition = "status = :status")
public class User {
```

Using Filters in Your Application

Your application programmatically determines which filters to activate or deactivate for a given Hibernate session. Each Session can have a different set of filters with different parameter values. By default, sessions do not have any active filters – you must explicitly enable filters programmatically for each session. The Session interface contains several methods for working with filters, as follows:

```
public Filter enableFilter(String filterName)
public Filter getEnabledFilter(String filterName)
public void disableFilter(String filterName)
```

The org.hibernate.Filter interface has six methods. You are unlikely to use validate(); Hibernate uses that method when it processes the filters. The other five methods are as follows:

```
public Filter setParameter(String name, Object value)
public Filter setParameterList(String name, Collection values)
public Filter setParameterList(String name, Object[] values)
public String getName()
public FilterDefinition getFilterDefinition()
```

пример

```
@Entity
@Data
@NoArgsConstructor
@FilterDefs({
```

```
@FilterDef(
name = "byStatus",
parameters = @ParamDef(name = "status", type = "boolean")),
@FilterDef(
name = "byGroup",
parameters = @ParamDef(name = "group", type = "string")),
@FilterDef(
name = "userEndsWith1")
})
@Filters({
@Filter(name = "byStatus", condition = "active = :status"),
@Filter(name = "byGroup",
condition =
":group in (select ug.groups from user_groups ug where ug.user_id = id)"),
@Filter(name = "userEndsWith1", condition = "name like '%1'")
})
public class User {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
Integer id;
@Column(unique = true)
String name;
boolean active;
@ElementCollection
Set<String> groups;
```