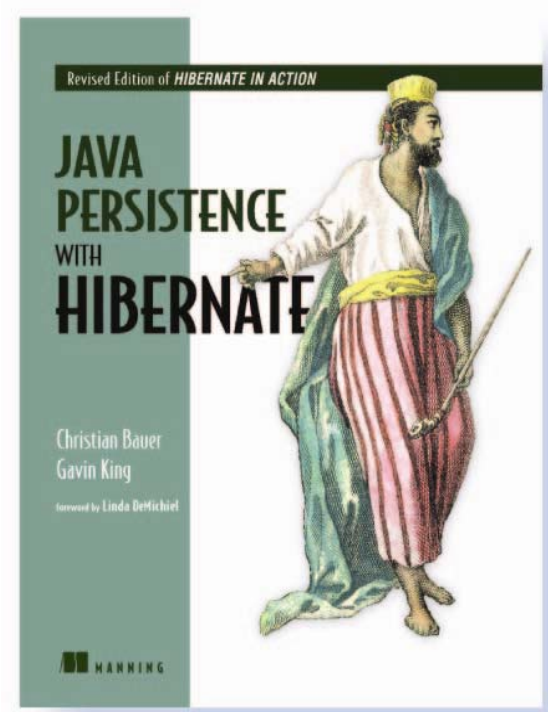
An Excerpt
Java Persistence with Hibernate
By Christian Bauer and Gavin King



Excerpt of Chapter 8:

Legacy Databases and custom SQL

Java Persistence with Hiberate

by Christian Bauer
and Gavin King

Chapter 8

Copyright 2006 Manning Publications

Excerpt from Chapter 8 of *Java Persistence with Hibernate*
(2nd Edition of *Hibernate in Action)*

8 Legacy databases and custom SQL

This chapter covers

- Legacy database integration and tricky mappings
- Customization of SQL statements
- Improving the SQL schema with custom DDL

Excerpt from Chapter 8 of *Java Persistence with Hibernate*
(2nd Edition of *Hibernate in Action*)

Many examples presented in this chapter are about "difficult" mappings. The first time you'll likely have problems creating a mapping is with a legacy database schema that can't be modified. We discuss typical issues you encounter in such a scenario and how you can bend and twist your mapping metadata instead of changing your application or database schema.

We also show you how you can override the SQL Hibernate generates automatically. This includes SQL queries, DML (create, update, delete) operations, as well as Hibernate's automatic DDL-generation feature. You'll see how to map stored procedures and user-defined SQL functions, and how to apply the right integrity rules in your database schema. This section will be especially useful if your DBA needs full control (or if you're a DBA and want to optimize Hibernate at the SQL level).

As you can see, the topics in this chapter are diverse; you don't have to read them all at once. You can consider a large part of this chapter to be reference material and come back when you face a particular issue.

8.1 Integrating legacy databases

In this section, we hope to cover all the things you may encounter when you have to deal with an existing legacy database or (and this is often synonymous) a weird or broken schema. If your development process is top-down, however, you may want to skip this section. Furthermore, we recommend that you first read all chapters about class, collection, and association mappings before you attempt to reverse-engineer a complex legacy schema.

We have to warn you: When your application inherits an existing legacy database schema, you should usually make as few changes to the existing schema as possible. Every change that you make to the schema could break other existing applications that access the database. Possibly expensive migration of existing data is also something you need to evaluate. In general, it isn't possible to build a new application and make no changes to the existing data model—a new application usually means additional business requirements that naturally require evolution of the database schema.

We'll therefore consider two types of problems: problems that relate to the changing business requirements (which generally can't be solved without schema changes) and problems that relate only to how you wish to represent the same business problem in your new application (these can usually, but not always, be solved without database schema changes). It should be clear that the first kind of problem is usually visible by looking at just the logical data model. The second

more often relates to the implementation of the logical data model as a physical database schema.

If you accept this observation, you'll see that the kinds of problems that require schema changes are those that necessitate addition of new entities, refactoring of existing entities, addition of new attributes to existing entities, and modification to the associations between entities. The problems that can be solved without schema changes usually involve inconvenient table or column definitions for a particular entity. In this section, we'll concentrate on these kinds of problems.

We assume that you've tried to reverse-engineer your existing schema with the Hibernate toolset, as described in chapter 2, section 2.3, "Reverse engineering a legacy database." The concepts and solutions discussed in the following sections assume that you have basic object/relational mapping in place and that you need to make additional changes to get it working. Alternatively, you can try to write the mapping completely by hand without the reverse-engineering tools.

Let's start with the most obvious problem: legacy primary keys.

8.1.1 Handling primary keys

We've already mentioned that we think natural primary keys can be a bad idea. Natural keys often make it difficult to refactor the data model when business requirements change. They may even, in extreme cases, impact performance. Unfortunately, many legacy schemas use (natural) composite keys heavily and, for the reason we discourage the use of composite keys, it may be difficult to change the legacy schema to use noncomposite natural or surrogate keys.

Therefore, Hibernate supports the use of natural keys. If the natural key is a composite key, support is via the `<composite-id>` mapping. Let's map both a composite and a noncomposite natural primary key.

Mapping a natural key

If you encountered a `USERS` table in a legacy schema, it's likely that `USERNAME` is the actual primary key. In this case, you have no surrogate identifier that is automatically generated. Instead, you enable the `assigned` identifier generator strategy to indicate to Hibernate that the identifier is a natural key assigned by the application before the object is saved:

```
<class name="User" table="USERS">
    <id name="username" column="USERNAME" length="16">
        <generator class="assigned"/>
    </id>

    ...
</class>
```

The code to save a new `User` is as follows:

```
User user = new User();
user.setUsername("johndoe"); // Assign a primary key value
user.setFirstname("John");
user.setLastname("Doe");
session.saveOrUpdate(user); // Will result in an INSERT
// System.out.println( session.getIdentifier(user) );
session.flush();
```

How does Hibernate know that `saveOrUpdate()` requires an `INSERT` and not an `UPDATE`? It doesn't, so a trick is needed: Hibernate queries the `USERS` table for the given username, and if it's found, Hibernate updates the row. If it isn't found, insertion of a new row is required and done. This is certainly not the best solution, because it triggers an additional hit on the database.

Several strategies avoid the `SELECT`:

- Add a `<version>` or a `<timestamp>` mapping, and a property, to your entity. Hibernate manages both values internally for optimistic concurrency control (discussed later in the book). As a side effect, an empty timestamp or a `0` or `NULL` version indicates that an instance is new and has to be inserted, not updated.
- Implement a Hibernate `Interceptor`, and hook it into your `Session`. This extension interface allows you to implement the method `isTransient()` with any custom procedure you may need to distinguish old and new objects.

On the other hand, if you're happy to use `save()` and `update()` explicitly instead of `saveOrUpdate()`, Hibernate doesn't have to distinguish between transient and detached instances—you do this by selecting the right method to call. (This issue is, in practice, the only reason to not use `saveOrUpdate()` all the time, by the way.)

Mapping natural primary keys with JPA annotations is straightforward:

```
@Id
private String username;
```

If no identifier generator is declared, Hibernate assumes that it has to apply the regular select-to-determine-state-unless-versioned strategy and expects the application to take care of the primary key value assignment. You can again avoid the `SELECT` by extending your application with an interceptor or by adding a version-control property (version number or timestamp).

Composite natural keys extend on the same ideas.

Mapping a composite natural key

Suppose that the primary key of the `USERS` table consists of a `USERNAME` and `DEPARTMENT_NR`. You can add a property named `departmentNr` to the `User` class and create the following mapping:

```
<class name="User" table="USERS">

    <composite-id>
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentNr"
                      column="DEPARTMENT_NR"/>
    </composite-id>

    ...
</class>
```

The code to save a new `User` looks like this:

```
User user = new User();

// Assign a primary key value
user.setUsername("johndoe");
user.setDepartmentNr(42);

// Set property values
user.setFirstname("John");
user.setLastname("Doe");

session.saveOrUpdate(user);
session.flush();
```

Again, keep in mind that Hibernate executes a `SELECT` to determine what `saveOrUpdate()` should do—unless you enable versioning control or a custom `Interceptor`. But what object can/should you use as the identifier when you call `load()` or `get()`? Well, it's possible to use an instance of the `User` class, for example:

```
User user = new User();

// Assign a primary key value
user.setUsername("johndoe");
user.setDepartmentNr(42);

// Load the persistent state into user
session.load(User.class, user);
```

In this code snippet, `User` acts as its own identifier class. It's more elegant to define a separate composite identifier class that declares just the key properties. Call this class `UserId`:

```
public class UserId implements Serializable {
    private String username;
```

```
    private Integer departmentNr;

    public UserId(String username, Integer departmentNr) {
        this.username = username;
        this.departmentNr = departmentNr;
    }

    // Getters...

    public int hashCode() {
        int result;
            result = username.hashCode();
            result = 29 * result + departmentNr.hashCode();
            return result;
    }

    public boolean equals(Object other) {
        if (other==null) return false;
        if ( !(other instanceof UserId) ) return false;
        UserId that = (UserId) other;
        return this.username.equals(that.username) &&
            this.departmentNr.equals(that.departmentNr);
    }
}
```

It's critical that you implement `equals()` and `hashCode()` correctly, because Hibernate relies on these methods for cache lookups. Identifier classes are also expected to implement `Serializable`.

You now remove the `username` and `departmentNr` properties from `User` and add a `userId` property. Create the following mapping:

```
<class name="User" table="USERS">

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentNr"
                      column="DEPARTMENT_NR"/>
    </composite-id>

    ...
</class>
```

Save a new instance of `User` with this code:

```
UserId id = new UserId("johndoe", 42);

User user = new User();

// Assign a primary key value
user.setUserId(id);

// Set property values
```

```
user.setFirstname("John");
user.setLastname("Doe");

session.saveOrUpdate(user);
session.flush();
```

Again, a SELECT is needed for `saveOrUpdate()` to work. The following code shows how to load an instance:

```
UserId id = new UserId("johndoe", 42);

User user = (User) session.load(User.class, id);
```

Now, suppose that the DEPARTMENT_NR is a foreign key referencing the DEPARTMENT table, and that you wish to represent this association in the Java domain model as a many-to-one association.

Foreign keys in composite primary keys

We recommend that you map a foreign key column that is also part of a composite primary key with a regular `<many-to-one>` element, and disable any Hibernate inserts or updates of this column with `insert="false" update="false"`, as follows:

```
<class name="User" table="USER">

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentId"
                      column="DEPARTMENT_ID"/>
    </composite-id>

    <many-to-one name="department"
                 class="Department"
                 column="DEPARTMENT_ID"
                 insert="false" update="false"/>
    ...
</class>
```

Hibernate now ignores the `department` property when updating or inserting a `User`, but you can of course read it with `johndoe.getDepartment()`. The relationship between a `User` and `Department` is now managed through the `departmentId` property of the `UserId` composite key class:

```
UserId id = new UserId("johndoe", department.getId() );

User user = new User();

// Assign a primary key value
user.setUserId(id);
```

```
// Set property values
user.setFirstname("John");
user.setLastname("Doe");
user.setDepartment(department);

session.saveOrUpdate(user);
session.flush();
```

Only the identifier value of the department has any effect on the persistent state; the `setDepartment(department)` call is done for consistency: Otherwise, you'd have to refresh the object from the database to get the department set after the flush. (In practice you can move all these details into the constructor of your composite identifier class.)

An alternative approach is a `<key-many-to-one>`:

```
<class name="User" table="USER">

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-many-to-one name="department"
                         class="Department"
                         column="DEPARTMENT_ID"/>
    </composite-id>

    ...
</class>
```

However, it's usually inconvenient to have an association in a composite identifier class, so this approach isn't recommended except in special circumstances. The `<key-many-to-one>` construct also has limitations in queries: You can't restrict a query result in HQL or Criteria across a `<key-many-to-one>` join (although it's possible these features will be implemented in a later Hibernate version).

Foreign keys to composite primary keys

Because `USERS` has a composite primary key, any referencing foreign key is also composite. For example, the association from `Item` to `User` (the seller) is now mapped with a composite foreign key.

Hibernate can hide this detail from the Java code with the following association mapping from `Item` to `User`:

```
<many-to-one name="seller" class="User">
    <column name="USERNAME"/>
    <column name="DEPARTMENT_ID"/>
</many-to-one>
```

Any collection owned by the `User` class also has a composite foreign key—for example, the inverse association, `items`, sold by this user:

```
<set name="itemsForAuction" inverse="true">
    <key>
        <column name="USERNAME"/>
        <column name="DEPARTMENT_ID"/>
    </key>
    <one-to-many class="Item"/>
</set>
```

Note that the order in which columns are listed is important and should match the order in which they appear in the `<composite-id>` element of the primary key mapping of `User`.

This completes our discussion of the basic composite key mapping technique in Hibernate. Mapping composite keys with annotations is almost the same, but as always, small differences are important.

Composite keys with annotations

The JPA specification covers strategies for handling composite keys. You have three options:

- Encapsulate the identifier properties in a separate class and mark it `@Embeddable`, like a regular component. Include a property of this component type in your entity class, and map it with `@Id` for an application-assigned strategy.
- Encapsulate the identifier properties in a separate class without any annotations on it. Include a property of this type in your entity class, and map it with `@EmbeddedId`.
- Encapsulate the identifier properties in a separate class. Now—and this is different that what you usually do in native Hibernate—duplicate all the identifier properties in the entity class. Then, annotate the entity class with `@IdClass` and specify the name of your encapsulated identifier class.

The first option is straightforward. You need to make the `UserId` class from the previous section embeddable:

```
@Embeddable
public class UserId implements Serializable {
    private String username;
    private String departmentNr;

    ...
}
```

As for all component mappings, you can define extra mapping attributes on the fields (or getter methods) of this class. To map the composite key of `User`, set the generation strategy to application assigned by omitting the `@GeneratedValue` annotation:

```
@Id
@AttributeOverrides({
    @AttributeOverride(name   = "username",
                       column = @Column(name="USERNAME") ),
    @AttributeOverride(name   = "departmentNr",
                       column = @Column(name="DEP_NR") )
})
private UserId userId;
```

Just as you did with regular component mappings earlier in the book, you can override particular attribute mappings of the component class, if you like.

The second composite-key mapping strategy doesn't require that you mark up the `UserId` primary key class. Hence, no `@Embeddable` and no other annotation on that class is needed. In the owning entity, you map the composite identifier property with `@EmbeddedId`, again, with optional overrides:

```
@EmbeddedId
@AttributeOverrides({
    @AttributeOverride(name   = "username",
                       column = @Column(name="USERNAME") ),
    @AttributeOverride(name   = "departmentNr",
                       column = @Column(name="DEP_NR") )
})
private UserId userId;
```

In a JPA XML descriptor, this mapping looks as follows:

```
<embeddable class="auction.model.UserId" access ="PROPERTY">
    <attributes>
        <basic name="username">
            <column name="UNAME"/>
        </basic>
        <basic name="departmentNr">
            <column name="DEPARTMENT_NR"/>
        </basic>
    </attributes>
</embeddable>

<entity class="auction.model.User" access="FIELD">
    <attributes>
        <embedded-id name="userId">
            <attribute-override name="username">
                <column name="USERNAME"/>
            </attribute-override>
```

```
            <attribute-override name="departmentNr">
                <column name="DEP_NR"/>
            </attribute-override>
        </embedded-id>
        ...
    </attributes>
</entity>
```

The third composite-key mapping strategy is a bit more difficult to understand, especially for experienced Hibernate users. First, you encapsulate all identifier attributes in a separate class—as in the previous strategy, no extra annotations on that class are needed. Now you duplicate all the identifier properties in the entity class:

```
@Entity
@Table(name = "USERS")
@IdClass(UserId.class)
public class User {

    @Id
    private String username;

    @Id
    private String departmentNr;

    // Accessor methods, etc.
    ...
}
```

Hibernate inspects the `@IdClass` and singles out all the duplicate properties (by comparing name and type) as identifier properties and as part of the primary key. All primary key properties are annotated with `@Id`, and depending on the position of these elements (field or getter method), the entity defaults to field or property access.

Note that this last strategy is also available in Hibernate XML mappings; however, it's somewhat obscure:

```
<composite-id class="UserId" mapped="true">
    <key-property name="username"
                  column="USERNAME"/>

    <key-property name="departmentNr"
                  column="DEP_NR"/>
</composite-id>
```

You omit the identifier property name of the entity (because there is none), so Hibernate handles the identifier internally. With `mapped="true"`, you enable the last JPA mapping strategy, so all key properties are now expected to be present in both the `User` and the `UserId` classes.

This composite identifier mapping strategy looks as follows if you use JPA XML descriptors:

```
<entity class="auction.model.User" access="FIELD">
    <id-class class="auction.model.UserId"/>
    <attributes>
        <id name="username"/>
        <id name="departmentNr"/>
    </attributes>
</entity>
```

Because we didn't find a compelling case for this last strategy defined in Java Persistence, we have to assume that it was added to the specification to support some legacy behavior (EJB 2.x entity beans).

Composite foreign keys are also possible with annotations. Let's first map the association from `Item` to `User`:

```
@ManyToOne
@JoinColumns({
  @JoinColumn(name="USERNAME", referencedColumnName = "USERNAME"),
  @JoinColumn(name="DEP_NR", referencedColumnName = "DEP_NR")
})
private User seller;
```

The primary difference between a regular `@ManyToOne` and this mapping is the number of columns involved—again, the order is important and should be the same as the order of the primary key columns. However, if you declare the `referencedColumnName` for each column, order isn't important, and both the source and target tables of the foreign key constraint can have different column names.

The inverse mapping from `User` to `Item` with a collection is even more straightforward:

```
@OneToMany(mappedBy = "seller")
private Set<Item> itemsForAuction = new HashSet<Item>();
```

This inverse side needs the `mappedBy` attribute, as usual for bidirectional associations. Because this is the inverse side, it doesn't need any column declarations.

In legacy schemas, a foreign key often doesn't reference a primary key.

Foreign key referencing nonprimary keys

Usually, a foreign key constraint references a primary key. A foreign key constraint is an integrity rule that guarantees that the referenced table has one row with a key value that matches the key value in the referencing table and given row. Note that a foreign key constraint can be self-referencing; in other words, a column with a foreign key constraint can reference the primary key column of the same table. (The `PARENT_CATEGORY_ID` in the CaveatEmptor `CATEGORY` table is one example.)

Legacy schemas sometimes have foreign key constraints that don't follow the simple "FK references PK" rule. Sometimes a foreign key references a nonprimary key: a simple unique column, a natural nonprimary key. Let's assume that in CaveatEmptor, you need to handle a legacy natural key column called `CUSTOMER_NR` on the `USERS` table:

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID">...</id>

    <property name="customerNr"
              column="CUSTOMER_NR"
              not-null="true"
              unique="true"/>

</class>
```

The only thing that is probably new to you in this mapping is the `unique` attribute. This is one of the SQL customization options in Hibernate; it's not used at runtime (Hibernate doesn't do any uniqueness validation) but to export the database schema with `hbm2ddl`. If you have an existing schema with a natural key, you assume that it's unique. For completeness, you can and should repeat such important constraints in your mapping metadata—maybe you'll use it one day to export a fresh schema.

Equivalent to the XML mapping, you can declare a column as unique in JPA annotations:

```
@Column(name = "CUSTOMER_NR", nullable = false, unique=true)
private int customerNr;
```

The next issue you may discover in the legacy schema is that the `ITEM` table has a foreign key column, `SELLER_NR`. In an ideal world, you would expect this foreign key to reference the primary key, `USER_ID`, of the `USERS` table. However, in a legacy schema, it may reference the natural unique key, `CUSTOMER_NR`. You need to map it with a property reference:

```
<class name="Item" table="ITEM">

    <id name="id" column="ITEM_ID">...</id>

    <many-to-one name="seller" column="SELLER_NR"
                 property-ref="customerNr"/>

</class>
```

You'll encounter the `property-ref` attribute in more exotic Hibernate mappings. It's used to tell Hibernate that "this is a mirror of the named property." In the previous example, Hibernate now knows the target of the foreign key reference. One

further thing to note is that `property-ref` requires the target property to be unique, so `unique="true"`, as shown earlier, is needed for this mapping.

If you try to map this association with JPA annotations, you may look for an equivalent to the `property-ref` attribute. You map the association with an explicit reference to the natural key column, `CUSTOMER_NR`:

```
@ManyToOne
@JoinColumn(name="SELLER_NR", referencedColumnName = "CUSTOMER_NR")
private User seller;
```

Hibernate now knows that the referenced target column is a natural key and manages the foreign key relationship accordingly.

To complete this example, you make this association mapping between the two classes bidirectional, with a mapping of an `itemsForAuction` collection on the `User` class. First, here it is in XML:

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID">...</id>

    <property name="customerNr" column="CUSTOMER_NR" unique="true"/>

    <set name="itemsForAuction" inverse="true">
        <key column="SELLER_NR" property-ref="customerNr"/>
        <one-to-many class="Item"/>
    </set>

</class>
```

Again the foreign key column in `ITEM` is mapped with a property reference to `customerNr`. In annotations, this is a lot easier to map as an inverse side:

```
@OneToMany(mappedBy = "seller")
private Set<Item> itemsForAuction = new HashSet<Item>();
```

Composite foreign key referencing nonprimary keys

Some legacy schemas are even more complicated than the one discussed before: A foreign key might be a composite key and, by design, reference a composite natural nonprimary key!

Let's assume that `USERS` has a natural composite key that includes the `FIRSTNAME`, `LASTNAME`, and `BIRTHDAY` columns. A foreign key may reference this natural key, as shown in figure 8.1.

To map this, you need to group several properties under the same name—otherwise you can't name the composite in a `property-ref`. Apply the `<properties>` element to group the mappings:

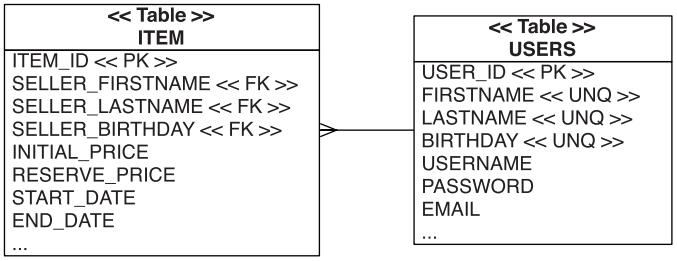


Figure 8.1 A composite foreign key references a composite primary key.

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID">...</id>

    <properties name="nameAndBirthday" unique="true" update="false">
        <property name="firstname" column="FIRSTNAME"/>
        <property name="lastname" column="LASTNAME"/>
        <property name="birthday" column="BIRTHDAY" type="date"/>
    </properties>

    <set name="itemsForAuction" inverse="true">
        <key property-ref="nameAndBirthday">
            <column name="SELLER_FIRSTNAME"/>
            <column name="SELLER_LASTNAME"/>
            <column name="SELLER_BIRTHDAY"/>
        </key>
        <one-to-many class="Item"/>
    </set>

</class>
```

As you can see, the `<properties>` element is useful not only to give several properties a name, but also to define a multicolumn `unique` constraint or to make several properties immutable. For the association mappings, the order of columns is again important:

```
<class name="Item" table="ITEM">

    <id name="id" column="ITEM_ID">...</id>

    <many-to-one name="seller" property-ref="nameAndBirthday">
        <column name="SELLER_FIRSTNAME"/>
        <column name="SELLER_LASTNAME"/>
        <column name="SELLER_BIRTHDAY"/>
    </many-to-one>

</class>
```

Fortunately, it's often straightforward to clean up such a schema by refactoring foreign keys to reference primary keys—if you can make changes to the database that don't disturb other applications sharing the data.

This completes our exploration of natural, composite, and foreign key-related problems you may have to deal with when you try to map a legacy schema. Let's move on to other interesting special mapping strategies.

Sometimes you can't make any changes to a legacy database—not even creating tables or views. Hibernate can map classes, properties, and even parts of associations to a simple SQL statement or expression. We call these kinds of mappings formula mappings.

8.1.2 Arbitrary join conditions with formulas

Mapping a Java artifact to an SQL expression is useful for more than integrating a legacy schema. You created two formula mappings already: The first, "Using derived properties," in chapter 4, section 4.4.1, was a simple derived read-only property mapping. The second formula calculated the discriminator in an inheritance mapping; see chapter 5, section 5.1.3, "Table per class hierarchy."

You'll now apply formulas for a more exotic purposes. Keep in mind that some of the mappings you'll see now are complex, and you may be better prepared to understand them after reading all the chapters in part 2 of this book.

Understanding the use case

You now map a literal join condition between two entities. This sounds more complex than it is in practice. Look at the two classes shown in figure 8.2.

A particular `Item` may have several `Bids`—this is a one-to-many association. But it isn't the only association between the two classes; the other, a unidirectional

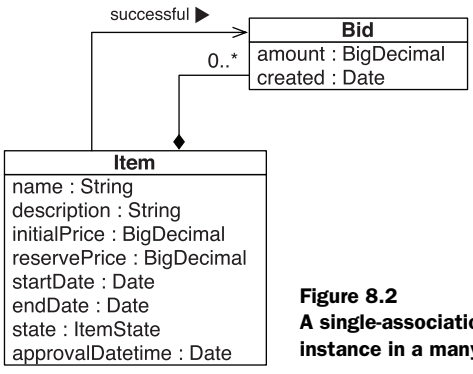


Figure 8.2
A single-association that references an instance in a many-association

one-to-one, is needed to single out one particular `Bid` instance as the winning bid. You map the first association because you'd like to be able to get all the bids for an auctioned item by calling `anItem.getBids()`. The second association allows you to call `anItem.getSuccessfulBid()`. Logically, one of the elements in the collection is also the successful bid object referenced by `getSuccessfulBid()`.

The first association is clearly a bidirectional one-to-many/many-to-one association, with a foreign key `ITEM_ID` in the `BID` table. (If you haven't mapped this before, look at chapter 6, section 6.4, "Mapping a parent/children relationship.")

The one-to-one association is more difficult; you can map it several ways. The most natural is a uniquely constrained foreign key in the `ITEM` table referencing a row in the `BID` table—the winning row, for example a `SUCCESSFUL_BID_ID` column.

Legacy schemas often need a mapping that isn't a simple foreign key relationship.

Mapping a formula join condition

Imagine that each row in the `BID` table has a flag column to mark the winning bid, as shown in figure 8.3. One `BID` row has the flag set to `true`, and all other rows for this auction item are naturally `false`. Chances are good that you won't find a constraint or an integrity rule for this relationship in a legacy schema, but we ignore this for now and focus on the mapping to Java classes.

To make this mapping even more interesting, assume that the legacy schema didn't use the SQL `BOOLEAN` datatype but a `CHAR(1)` field and the values `T` (for true) and `F` (for false) to simulate the boolean switch. Your goal is to map this flag column to a `successfulBid` property of the `Item` class. To map this as an object reference, you need a literal join condition, because there is no foreign key Hibernate can use for a join. In other words, for each `ITEM` row, you need to join a row from the `BID` table that has the `SUCCESSFUL` flag set to `T`. If there is no such row, the `anItem.getSuccessfulBid()` call returns `null`.

Let's first map the `Bid` class and a `successful` boolean property to the `SUCCESSFUL` database column:

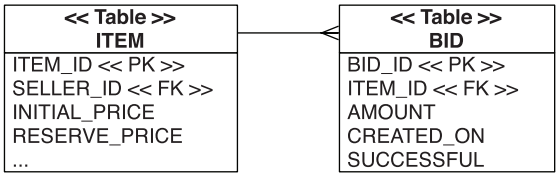


Figure 8.3
The winning bid is marked with the `SUCCESSFUL` column flag.

```
<class name="Bid" table="BID">

    <id name="id" column="BID_ID"...

    <property name="amount"
              ...

    <properties name="successfulReference">

        <property name="successful"
                  column="SUCCESSFUL"
                  type="true_false"/>
                  ...

        <many-to-one name="item"
                     class="Item"
                     column="ITEM_ID"/>
                     ...
    </properties>

    <many-to-one name="bidder"
                 class="User"
                 column="BIDDER_ID"/>
                 ...
</class>
```

The `type="true_false"` attribute creates a mapping between a Java `boolean` primitive (or its wrapper) property and a simple `CHAR(1)` column with `T/F` literal values—it's a built-in Hibernate mapping type. You again group several properties with `<properties>` under a name that you can reference in other mappings. What is new here is that you can group a `<many-to-one>`, not only basic properties.

The real trick is happening on the other side, for the mapping of the `successfulBid` property of the `Item` class:

```
<class name="Item" table="ITEM">

    <id name="id" column="ITEM_ID"...

    <property name="initialPrice"
        ...

    <one-to-one name="successfulBid"
                property-ref="successfulReference">
        <formula>'T'</formula>
        <formula>ITEM_ID</formula>
    </one-to-one>

    <set name="bids" inverse="true">
        <key column="ITEM_ID”/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

Ignore the <set> association mapping in this example; this is the regular one-to-many association between Item and Bid, bidirectional, on the ITEM_ID foreign key column in BID.

NOTE *Isn't <one-to-one> used for primary key associations?* Usually, a <one-to-one> mapping is a primary key relationship between two entities, when rows in both entity tables share the same primary key value. However, by using a formula with a property-ref, you can apply it to a foreign key relationship. In the example shown in this section, you could replace the <one-to-one> element with <many-to-one>, and it would still work.

The interesting part is the <one-to-one> mapping and how it relies on a property-ref and literal formula values as a join condition when you work with the association.

Working with the association

The full SQL query for retrieval of an auction item and its successful bid looks like this:

```
select
    i.ITEM_ID,
    i.INITIAL_PRICE,
    ...
    b.BID_ID,
    b.AMOUNT,
    b.SUCCESSFUL,
    b.BIDDER_ID,
    ...
from
    ITEM i
left outer join
    BID b
        on 'T' = b.SUCCESSFUL
        and i.ITEM_ID = b.ITEM_ID
where
    i.ITEM_ID = ?
```

When you load an Item, Hibernate now joins a row from the BID table by applying a join condition that involves the columns of the successfulReference property. Because this is a grouped property, you can declare individual expressions for each of the columns involved, in the right order. The first one, 'T', is a literal, as you can see from the quotes. Hibernate now includes 'T' = SUCCESSFUL in the join condition when it tries to find out whether there is a successful row in the BID table. The second expression isn't a literal but a column name (no quotes).

Hence, another join condition is appended: `i.ITEM_ID = b.ITEM_ID`. You can expand this and add more join conditions if you need additional restrictions.

Note that an outer join is generated because the item in question may not have a successful bid, so `NULL` is returned for each `b.*` column. You can now call `anItem.getSuccessfulBid()` to get a reference to the successful bid (or `null` if none exists).

Finally, with or without database constraints, you can't just implement an `item.setSuccessfulBid()` method that only sets the value on a private field in the `Item` instance. You have to implement a small procedure in this setter method that takes care of this special relationship and the flag property on the bids:

```
public class Item {
    ...

    private Bid successfulBid;
    private Set<Bid> bids = new HashSet<Bid>();

    public Bid getSuccessfulBid() {
        return successfulBid;
    }

    public void setSuccessfulBid(Bid successfulBid) {
        if (successfulBid != null) {

            for (Bid bid : bids)
                bid.setSuccessful(false);

            successfulBid.setSuccessful(true);
            this.successfulBid = successfulBid;
        }
    }

}
```

When `setSuccessfulBid()` is called, you set all bids to not successful. Doing so may trigger the loading of the collection—a price you have to pay with this strategy. Then, the new successful bid is marked and set as an instance variable. Setting the flag updates the `SUCCESSFUL` column in the `BID` table when you save the objects. To complete this (and to fix the legacy schema), your database-level constraints need to do the same as this method. (We'll come back to constraints later in this chapter.)

One of the things to remember about this literal join condition mapping is that it can be applied in many other situations, not only for successful or default relationships. Whenever you need some arbitrary join condition appended to your queries, a formula is the right choice. For example, you could use it in a

<many-to-many> mapping to create a literal join condition from the association table to the entity table(s).

Unfortunately, at the time of writing, Hibernate Annotations doesn't support arbitrary join conditions expressed with formulas. The grouping of properties under a reference name also wasn't possible. We expect that these features will closely resemble the XML mapping, once they're available.

Another issue you may encounter in a legacy schema is that it doesn't integrate nicely with your class granularity. Our usual recommendation to have more classes than tables may not work, and you may have to do the opposite and join arbitrary tables into one class.

8.1.3 Joining arbitrary tables

We've already shown the <join> mapping element in an inheritance mapping in chapter 5; see section 5.1.5, "Mixing inheritance strategies." It helped to break out properties of a particular subclass into a separate table, out of the primary inheritance hierarchy table. This generic functionality has more uses—however, we have to warn you that <join> can also be a bad idea. Any properly designed system should have more classes than tables. Splitting a single class into separate tables is something you should do only when you need to merge several tables in a legacy schema into a single class.

Moving properties into a secondary table

Suppose that in CaveatEmptor, you aren't keeping a user's address information with the user's main information in the USERS table, mapped as a component, but in a separate table. This is shown in figure 8.4. Note that each BILLING_ADDRESS has a foreign key USER_ID, which is in turn the primary key of the BILLING_ADDRESS table.

To map this in XML, you need to group the properties of the Address in a <join> element:

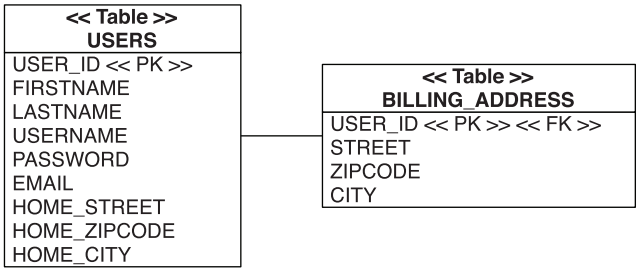


Figure 8.4
Breaking out the billing address data into a secondary table

```
<class name="User" table="USERS">
    <id>...

    <join table="BILLING_ADDRESS" optional="true">
        <key column="USER_ID"/>
        <component name="billingAddress" class="Address">
            <property   name="street"
                        type="string"
                        column="STREET"
                        length="255"/>
            <property   name="zipcode"
                        type="string"
                        column="ZIPCODE"
                        length="16"/>
            <property   name="city"
                        type="string"
                        column="CITY"
                        length="255"/>
        </component>
    </join>

</class>
```

You don't have to join a component; you can as well join individual properties or even a `<many-to-one>` (we did this in the previous chapter for optional entity associations). By setting `optional="true"`, you indicate that the component property may also be `null` for a `User` with no `billingAddress`, and that no row should then be inserted into the secondary table. Hibernate also executes an outer join instead of an inner join to retrieve the row from the secondary table. If you declared `fetch="select"` on the `<join>` mapping, a secondary select would be used for that purpose.

The notion of a secondary table is also included in the Java Persistence specification. First, you have to declare a secondary table (or several) for a particular entity:

```
@Entity
@Table(name = "USERS")
@SecondaryTable(
    name = "BILLING_ADDRESS",
    pkJoinColumns = {
        @PrimaryKeyJoinColumn(name="USER_ID")
    }
)
public class User {
    ...
}
```

Each secondary table needs a name and a join condition. In this example, a foreign key column references the primary key column of the `USERS` table, just like earlier in the XML mapping. (This is the default join condition, so you can only declare the secondary table name, and nothing else). You can probably see that the syntax of annotations is starting to become an issue and code is more difficult to read. The good news is that you won't have to use secondary tables often.

The actual component property, `billingAddress`, is mapped as a regular `@Embedded` class, just like a regular component. However, you need to override each component property column and assign it to the secondary table, in the `User` class:

```
@Embedded
@AttributeOverrides( {
    @AttributeOverride(
        name   = "street",
        column = @Column(name="STREET",
                         table = "BILLING_ADDRESS")
    ),
    @AttributeOverride(
        name   = "zipcode",
        column = @Column(name="ZIPCODE",
                         table = "BILLING_ADDRESS")
    ),
    @AttributeOverride(
        name   = "city",
        column = @Column(name="CITY",
                         table = "BILLING_ADDRESS")
    )
})
private Address billingAddress;
```

This is no longer easily readable, but it's the price you pay for mapping flexibility with declarative metadata in annotations. Or, you can use a JPA XML descriptor:

```
<entity class="auction.model.User" access="FIELD">
    <table name="USERS"/>
    <secondary-table name="BILLING_ADDRESS">
        <primary-key-join-column
            referenced-column-name="USER_ID"/>
    </secondary-table>
    <attributes>
        ...
        <embedded name="billingAddress">
            <attribute-override name="street">
                <column name="STREET" table="BILLING_ADDRESS"/>
            </attribute-override>
            <attribute-override name="zipcode">
```

```
                <column name="ZIPCODE" table="BILLING_ADDRESS"/>
            </attribute-override>
            <attribute-override name="city">
                <column name="CITY" table="BILLING_ADDRESS"/>
            </attribute-override>
        </embedded>
    </attributes>
</entity>
```

Another, even more exotic use case for the `<join>` element is inverse joined properties or components.

Inverse joined properties

Let's assume that in CaveatEmptor you have a legacy table called `DAILY_BILLING`. This table contains all the open payments, executed in a nightly batch, for any auctions. The table has a foreign key column to `ITEM`, as you can see in figure 8.5.

Each payment includes a `TOTAL` column with the amount of money that will be billed. In CaveatEmptor, it would be convenient if you could access the price of a particular auction by calling `anItem.getBillingTotal()`.

You can map the column from the `DAILY_BILLING` table into the `Item` class. However, you never insert or update it from this side; it's read-only. For that reason, you map it inverse—a simple mirror of the (supposed, you don't map it here) other side that takes care of maintaining the column value:

```
<class name="Item" table="ITEM">
    <id>...

    <join table="DAILY_BILLING" optional="true" inverse="true">
        <key column="ITEM_ID"/>
        <property   name="billingTotal"
                    type="big_decimal"
                    column="TOTAL"/>
    </join>

</class>
```

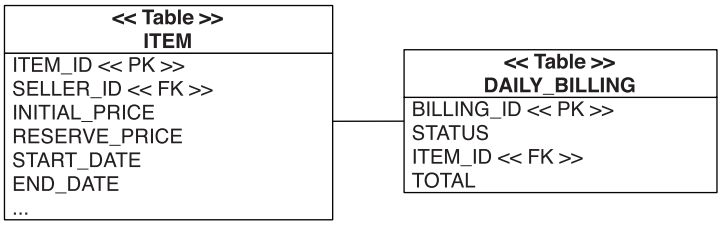


Figure 8.5 The daily billing summary references an item and contains the total sum.

Note that an alternative solution for this problem is a derived property using a formula expression and a correlated subquery:

```
<property name="billingTotal"
          type="big_decimal"
          formula="( select db.TOTAL from DAILY_BILLING db
                     where db.ITEM_ID = ITEM_ID )"/>
```

The main difference is the SQL `SELECT` used to load an `ITEM`: The first solution defaults to an outer join, with an optional second `SELECT` if you enable `<join fetch="select">`. The derived property results in an embedded subselect in the select clause of the original query. At the time of writing, inverse join mappings aren't supported with annotations, but you can use a Hibernate annotation for formulas.

As you can probably guess from the examples, `<join>` mappings come in handy in many situations. They're even more powerful if combined with formulas, but we hope you won't have to use this combination often.

One further problem that often arises in the context of working with legacy data are database triggers.

8.1.4 Working with triggers

There are some reasons for using triggers even in a brand-new database, so legacy data isn't the only scenerio in which they can cause problems. Triggers and object state management with an ORM software are almost always an issue, because triggers may run at inconvenient times or may modify data that isn't synchronized with the in-memory state.

Triggers that run on INSERT

Suppose the `ITEM` table has a `CREATED` column, mapped to a `created` property of type `Date`, that is initialized by a trigger that executes automatically on insertion. The following mapping is appropriate:

```
<property name="created"
          type="timestamp"
          column="CREATED"
          insert="false"
          update="false"/>
```

Notice that you map this property `insert="false" update="false"` to indicate that it isn't to be included in SQL `INSERT`s or `UPDATE`s by Hibernate.

After saving a new `Item`, Hibernate isn't aware of the value assigned to this column by the trigger, because it occurred after the `INSERT` of the item row. If you

need the generated value in the application, you must explicitly tell Hibernate to reload the object with an SQL `SELECT`. For example:

```
Item item = new Item();
...
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

session.save(item);
session.flush(); // Force the INSERT to occur
session.refresh(item); // Reload the object with a SELECT

System.out.println( item.getCreated() );

tx.commit();
session.close();
```

Most problems involving triggers may be solved in this way, using an explicit `flush()` to force immediate execution of the trigger, perhaps followed by a call to `refresh()` to retrieve the result of the trigger.

Before you add `refresh()` calls to your application, we have to tell you that the primary goal of the previous section was to show you when to use `refresh()`. Many Hibernate beginners don't understand its real purpose and often use it incorrectly. A more formal definition of `refresh()` is "refresh an in-memory instance in persistent state with the current values present in the database."

For the example shown, a database trigger filling a column value after insertion, a much simpler technique can be used:

```
<property name="created"
          type="timestamp"
          column="CREATED"
          generated="insert"
          insert="false"
          update="false"/>
```

With annotations, use a Hibernate extension:

```
@Temporal(TemporalType.TIMESTAMP)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
@Column(name = "CREATED", insertable = false, updatable = false)
private Date created;
```

We have already discussed the `generated` attribute in detail in chapter 4, section 4.4.1.3, "Generated and default property values." With `generated="insert"`, Hibernate automatically executes a `SELECT` after insertion, to retrieve the updated state.

There is one further problem to be aware of when your database executes triggers: reassociation of a detached object graph and triggers that run on each `UPDATE`.

Triggers that run on UPDATE

Before we discuss the problem of `ON UPDATE` triggers in combination with reattachment of objects, we need to point out an additional setting for the `generated` attribute:

```
<version name="version"
         column="OBJ_VERSION"
         generated="always"/>
...
<timestamp name="lastModified"
           column="LAST_MODIFIED"
           generated="always"/>
...
<property name="lastModified"
          type="timestamp"
          column="LAST_MODIFIED"
          generated="always"
          insert="false"
          update="false"/>
```

With annotations, the equivalent mappings are as follows:

```
@Version
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "OBJ_VERSION")
private int version;

@Version
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "LAST_MODIFIED")
private Date lastModified;

@Temporal(TemporalType.TIMESTAMP)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "LAST_MODIFIED", insertable = false, updatable = false)
private Date lastModified;
```

With `always`, you enable Hibernate's automatic refreshing not only for insertion but also for updating of a row. In other words, whenever a version, timestamp, or any property value is generated by a trigger that runs on `UPDATE` SQL statements,

you need to enable this option. Again, refer to our earlier discussion of generated properties in section 4.4.1.

Let's look at the second issue you may run into if you have triggers running on updates. Because no snapshot is available when a detached object is reattached to a new `Session` (with `update()` or `saveOrUpdate()`), Hibernate may execute unnecessary SQL `UPDATE` statements to ensure that the database state is synchronized with the persistence context state. This may cause an `UPDATE` trigger to fire inconveniently. You avoid this behavior by enabling `select-before-update` in the mapping for the class that is persisted to the table with the trigger. If the `ITEM` table has an update trigger, add the following attribute to your mapping:

```
<class name="Item"
       table="ITEM"
       select-before-update="true">
    ...
</class>
```

This setting forces Hibernate to retrieve a snapshot of the current database state using an SQL `SELECT`, enabling the subsequent `UPDATE` to be avoided if the state of the in-memory `Item` is the same. You trade the inconvenient `UPDATE` for an additional `SELECT`.

A Hibernate annotation enables the same behavior:

```
@Entity
@org.hibernate.annotations.Entity(selectBeforeUpdate = true)
public class Item { ... }
```

Before you try to map a legacy scheme, note that the `SELECT` before an update only retrieves the state of the entity instance in question. No collections or associated instances are eagerly fetched, and no prefetching optimization is active. If you start enabling `selectBeforeUpdate` for many entities in your system, you'll probably find that the performance issues introduced by the nonoptimized selects are problematic. A better strategy uses merging instead of reattachment. Hibernate can then apply some optimizations (outer joins) when retrieving database snapshots. We'll talk about the differences between reattachment and merging later in the book in more detail.

Let's summarize our discussion of legacy data models: Hibernate offers several strategies to deal with (natural) composite keys and inconvenient columns easily. Before you try to map a legacy schema, our recommendation is to carefully examine whether a schema change is possible. In our experience, many developers immediately dismiss database schema changes as too complex and time-consuming and look for a Hibernate solution. This sometimes isn't justified, and you

should consider schema evolution a natural part of your schema's lifecycle. If tables change, then a data export, some transformation, and an import may solve the problem. One day of work may save many days in the long run.

Legacy schemas often also require customization of the SQL generated by Hibernate, be it for data manipulation (DML) or schema definition (DDL).

8.2 Customizing SQL

SQL started its life in the 1970s but wasn't (ANSI) standardized until 1986. Although each update of the SQL standard has seen new (and many controversial) features, every DBMS product that supports SQL does so in its own unique way. The burden of portability is again on the database application developers. This is where Hibernate helps: Its built-in query mechanisms, HQL and the `Criteria` API, produce SQL that depends on the configured database dialect. All other automatically generated SQL (for example, when a collection has to be retrieved on demand) is also produced with the help of dialects. With a simple switch of the dialect, you can run your application on a different DBMS.

To support this portability, Hibernate has to handle three kinds of operations:

- Every data-retrieval operation results in `SELECT` statements being executed. Many variations are possible; for example, database products may use a different syntax for the join operation or how a result can be limited to a particular number of rows.
- Every data modification requires the execution of Data Manipulation Language (DML) statements, such as `UPDATE`, `INSERT`, and `DELETE`. DML often isn't as complex as data retrieval, but it still has product-specific variations.
- A database schema must be created or altered before DML and data retrieval can be executed. You use Data Definition Language (DDL) to work on the database catalog; it includes statements such as `CREATE`, `ALTER`, and `DROP`. DDL is almost completely vendor specific, but most products have at least a similar syntax structure.

Another term we use often is CRUD, for create, read, update, and delete. Hibernate generates all this SQL for you, for all CRUD operations and schema definition. The translation is based on an `org.hibernate.dialect.Dialect` implementation—Hibernate comes bundled with dialects for all popular SQL database management systems. We encourage you to look at the source code of the dialect you're using; it's not difficult to read. Once you're more experienced with

Hibernate, you may even want to extend a dialect or write your own. For example, to register a custom SQL function for use in HQL selects, you'd extend an existing dialect with a new subclass and add the registration code—again, check the existing source code to find out more about the flexibility of the dialect system.

On the other hand, you sometimes need more control than Hibernate APIs (or HQL) provide, when you need to work on a lower level of abstraction. With Hibernate you can override or completely replace all CRUD SQL statements that will be executed. You can customize and extend all DDL SQL statements that define your schema, if you rely on Hibernate's automatic schema-export tool (you don't have to).

Furthermore Hibernate allows you to get a plain JDBC `Connection` object at all times through `session.connection()`. You should use this feature as a last resort, when nothing else works or anything else would be more difficult than plain JDBC. With the newest Hibernate versions, this is fortunately exceedingly rare, because more and more features for typical stateless JDBC operations (bulk updates and deletes, for example) are built-in, and many extension points for custom SQL already exist.

This custom SQL, both DML and DDL, is the topic of this section. We start with custom DML for create, read, update, and delete operations. Later, we integrate stored database procedures to do the same work. Finally, we look at DDL customization for the automatic generation of a database schema and how you can create a schema that represents a good starting point for the optimization work of a DBA.

Note that at the time of writing this detailed customization of automatically generated SQL isn't available in annotations; hence, we use XML metadata exclusively in the following examples. We expect that a future version of Hibernate Annotations will include better support for SQL customization.

8.2.1 Writing custom CRUD statements

The first custom SQL you'll write is used to load entities and collections. (Most of the following code examples show almost the same SQL Hibernate executes by default, without much customization—this helps you to understand the mapping technique more quickly.)

Loading entities and collections with custom SQL

For each entity class that requires a custom SQL operation to load an instance, you define a `<loader>` reference to a named query:

```
<class name="User" table="USERS">
    <id name="id" column="USER_ID"...
```

```
    <loader query-ref="loadUser"/>
    ...
</class>
```

The `loadUser` query can now be defined anywhere in your mapping metadata, separate and encapsulated from its use. This is an example of a simple query that retrieves the data for a `User` entity instance:

```
<sql-query name="loadUser">
    <return alias="u" class="User"/>
    select
        us.USER_ID      as {u.id},
        us.FIRSTNAME    as {u.firstname},
        us.LASTNAME     as {u.lastname},
        us.USERNAME     as {u.username},
        us."PASSWORD"   as {u.password},
        us.EMAIL        as {u.email},
        us.RANKING      as {u.ranking},
        us.IS_ADMIN     as {u.admin},
        us.CREATED      as {u.created},
        us.HOME_STREET  as {u.homeAddress.street},
        us.HOME_ZIPCODE as {u.homeAddress.zipcode},
        us.HOME_CITY    as {u.homeAddress.city},
        us.DEFAULT_BILLING_DETAILS_ID as {u.defaultBillingDetails}
    from
        USERS us
    where
        us.USER_ID = ?
</sql-query>
```

As you can see, the mapping from column names to entity properties uses a simple aliasing. In a named loader query for an entity, you have to `SELECT` the following columns and properties:

- The primary key columns and primary key property or properties, if a composite primary key is used.
- All scalar properties, which must be initialized from their respective column(s).
- All composite properties which must be initialized. You can address the individual scalar elements with the following aliasing syntax: `{entity-alias.componentProperty.scalarProperty}`.
- All foreign key columns, which must be retrieved and mapped to the respective `many-to-one` property. See the `DEFAULT_BILLING_DETAILS_ID` example in the previous snippet.

- All scalar properties, composite properties, and many-to-one entity references that are inside a `<join>` element. You use an inner join to the secondary table if all the joined properties are never `NULL`; otherwise, an outer join is appropriate. (Note that this isn't shown in the example.)
- If you enable lazy loading for scalar properties, through bytecode instrumentation, you don't need to load the lazy properties. See chapter 13, section 13.1.6, "Lazy loading with interception."

The `{propertyName}` aliases as shown in the previous example are not absolutely necessary. If the name of a column in the result is the same as the name of a mapped column, Hibernate can automatically bind them together.

You can even call a mapped query by name in your application with `session.getNamedQuery("loadUser")`. Many more things are possible with custom SQL queries, but we'll focus on basic SQL customization for CRUD in this section. We come back to other relevant APIs in chapter 15, section 15.2, "Using native SQL queries."

Let's assume that you also want to customize the SQL that is used to load a collection—for example, the `items` sold by a `User`. First, declare a loader reference in the collection mapping:

```
<set name="items" inverse="true">
    <key column="SELLER_ID" not-null="true"/>
    <one-to-many class="Item"/>
    <loader query-ref="loadItemsForUser"/>
</set>
```

The named query `loadItemsForUser` looks almost the same as the entity loader:

```
<sql-query name="loadItemsForUser">
    <load-collection alias="i" role="User.items"/>
    select
        {i.*}
    from
        ITEM i
    where
        i.SELLER_ID = :id
</sql-query>
```

There are two major differences: One is the `<load-collection>` mapping from an alias to a collection role; it should be self-explanatory. What is new in this query is an automatic mapping from the SQL table alias `ITEM i` to the properties of all items with `{i.*}`. You created a connection between the two by using the same alias: the symbol `i`. Furthermore, you're now using a named parameter, `:id`,

instead of a simple positional parameter with a question mark. You can use whatever syntax you prefer.

Sometimes, loading an entity instance and a collection is better done in a single query, with an outer join (the entity may have an empty collection, so you can't use an inner join). If you want to apply this eager fetch, don't declare a loader references for the collection. The entity loader takes care of the collection retrieval:

```
<sql-query name="loadUser">
    <return alias="u" class="User"/>
    <return-join alias="i" property="u.items"/>
    select
        {u.*}, {i.*}
    from
        USERS u
    left outer join ITEM i
        on u.USER_ID = i.SELLER_ID
    where
        u.USER_ID = ?
</sql-query>
```

Note how you use the `<return-join>` element to bind an alias to a collection property of the entity, effectively linking both aliases together. Further note that this technique also works if you'd like to eager-fetch one-to-one and many-to-one associated entities in the original query. In this case, you may want an inner join if the associated entity is mandatory (the foreign key can't be `NULL`) or an outer join if the target is optional. You can retrieve many single-ended associations eagerly in one query; however, if you (outer-) join more than one collection, you create a Cartesian product, effectively multiplying all collection rows. This can generate huge results that may be slower than two queries. You'll meet this limitation again when we discuss fetching strategies in chapter 13.

As mentioned earlier, you'll see more SQL options for object loading later in the book. We now discuss customization of insert, update, and delete operations, to complete the CRUD basics.

Custom insert, update, and delete

Hibernate produces all trivial CRUD SQL at startup. It caches the SQL statements internally for future use, thus avoiding any runtime cost of SQL generation for the most common operations. You've seen how you can override the R of CRUD, so let's do the same for CUD.

For each entity or collection, you can define custom CUD SQL statements inside the `<sql-insert>`, `<sql-delete>`, and `<sql-update>` element, respectively:

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID"...

    ...

    <join table="BILLING_ADDRESS" optional="true">
        <key column="USER_ID"/>
        <component name="billingAddress" class="Address">
            <property ...
        </component>

        <sql-insert>
            insert into BILLING_ADDRESS
                            (STREET, ZIPCODE, CITY, USER_ID)
            values (?, ?, ?, ?)
        </sql-insert>

        <sql-update>...</sql-update>

        <sql-delete>...</sql-delete>

    </join>

    <sql-insert>
        insert into USERS (FIRSTNAME, LASTNAME, USERNAME,
                           "PASSWORD", EMAIL, RANKING, IS_ADMIN,
                           CREATED, DEFAULT_BILLING_DETAILS_ID,
                           HOME_STREET, HOME_ZIPCODE, HOME_CITY,
                           USER_ID)
        values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    </sql-insert>

    <sql-update>...</sql-update>

    <sql-delete>...</sql-delete>

</class>
```

This mapping example may look complicated, but it's really simple. You have two tables in a single mapping: the primary table for the entity, `USERS`, and the secondary table `BILLING_ADDRESS` from your legacy mapping earlier in this chapter. Whenever you have secondary tables for an entity, you have to include them in any custom SQL—hence the `<sql-insert>`, `<sql-delete>`, and `<sql-update>` elements in both the `<class>` and the `<join>` sections of the mapping.

The next issue is the binding of arguments for the statements. For CUD SQL customization, only positional parameters are supported at the time of writing. But what is the right order for the parameters? There is an internal order to how Hibernate binds arguments to SQL parameters. The easiest way to figure out the right SQL statement and parameter order is to let Hibernate generate one for

you. Remove your custom SQL from the mapping file, enable DEBUG logging for the org.hibernate.persister.entity package, and watch (or search) the Hibernate startup log for lines similar to these:

```
AbstractEntityPersister - Insert 0: insert into USERS (FIRSTNAME,
 LASTNAME, USERNAME, "PASSWORD", EMAIL, RANKING, IS_ADMIN,
 CREATED, DEFAULT_BILLING_DETAILS_ID, HOME_STREET, HOME_ZIPCODE,
 HOME_CITY, USER_ID) values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
AbstractEntityPersister - Update 0: update USERS set
 FIRSTNAME=?, LASTNAME=?, "PASSWORD"=?, EMAIL=?, RANKING=?,
 IS_ADMIN=?, DEFAULT_BILLING_DETAILS_ID=?, HOME_STREET=?,
 HOME_ZIPCODE=?, HOME_CITY=? where USER_ID=?
...
```

You can now copy the statements you want to customize into your mapping file and make the necessary changes. For more information on logging in Hibernate, refer to "Enabling logging statistics" in chapter 2, in section 2.1.3.

You've now mapped CRUD operations to custom SQL statements. On the other hand, dynamic SQL isn't the only way how you can retrieve and manipulate data. Predefined and compiled procedures stored in the database can also be mapped to CRUD operations for entities and collections.

8.2.2 Integrating stored procedures and functions

Stored procedures are common in database application development. Moving code closer to the data and executing it inside the database has distinct advantages.

First, you don't have to duplicate functionality and logic in each program that accesses the data. A different point of view is that a lot of business logic shouldn't be duplicated, so it can be applied all the time. This includes procedures that guarantee the integrity of the data: for example, constraints that are too complex to be implemented declaratively. You'll usually also find triggers in a database that has procedural integrity rules.

Stored procedures have advantages for all processing on large amounts of data, such as reporting and statistical analysis. You should always try to avoid moving large data sets on your network and between your database and application servers, so a stored procedure is a natural choice for mass data operations. Or, you can implement a complex data-retrieval operation that assembles data with several queries before it passes the final result to the application client.

On the other hand, you'll often see (legacy) systems that implement even the most basic CRUD operations with a stored procedure. As a variation of this, systems that don't allow any direct SQL DML, but only stored procedure calls, also had (and sometimes still have) their place.

You may start integrating existing stored procedures for CRUD or for mass data operations, or you may begin writing your own stored procedure first.

Writing a procedure

Programming languages for stored procedures are usually proprietary. Oracle PL/SQL, a procedural dialect of SQL, is very popular (and available with variations in other database products). Some databases even support stored procedures written in Java. Standardizing Java stored procedures was part of the SQLJ effort, which, unfortunately, hasn't been successful.

You'll use the most common stored procedure systems in this section: Oracle databases and PL/SQL. It turns out that stored procedures in Oracle, like so many other things, are always different than you expect; we'll tell you whenever something requires extra attention.

A stored procedure in PL/SQL has to be created in the database catalog as source code and then compiled. Let's first write a stored procedure that can load all `User` entities that match a particular criterion:

```
<database-object>
    <create>
        create or replace procedure SELECT_USERS_BY_RANK
        (
         OUT_RESULT out SYS_REFCURSOR,
         IN_RANK    in  int
        ) as
        begin
         open OUT_RESULT for
         select
             us.USER_ID         as USER_ID,
             us.FIRSTNAME       as FIRSTNAME,
             us.LASTNAME        as LASTNAME,
             us.USERNAME        as USERNAME,
             us."PASSWORD"      as PASSWD,
             us.EMAIL           as EMAIL,
             us.RANKING         as RANKING,
             us.IS_ADMIN        as IS_ADMIN,
             us.CREATED         as CREATED,
             us.HOME_STREET     as HOME_STREET,
             us.HOME_ZIPCODE    as HOME_ZIPCODE,
             us.HOME_CITY       as HOME_CITY,
             ba.STREET          as BILLING_STREET,
             ba.ZIPCODE         as BILLING_ZIPCODE,
             ba.CITY            as BILLING_CITY,
             us.DEFAULT_BILLING_DETAILS_ID
                                as DEFAULT_BILLING_DETAILS_ID
         from
             USERS us
```

```
            left outer join
                BILLING_ADDRESS ba
                    on us.USER_ID = ba.USER_ID
               where
                   us.RANKING >= IN_RANK;
           end;
      </create>
      <drop>
          drop procedure SELECT_USERS_BY_RANK
      </drop>
  </database-object>
```

You embed the DDL for the stored procedure in a `<database-object>` element for creation and removal. That way, Hibernate automatically creates and drops the procedure when the database schema is created and updated with the `hbm2ddl` tool. You could also execute the DDL by hand on your database catalog. Keeping it in your mapping files (in whatever location seems appropriate, such as in `MyStoredProcedures.hbm.xml`) is a good choice if you're working on a nonlegacy system with no existing stored procedures. We'll come back to other options for the `<database-object>` mapping later in this chapter.

As before, the stored procedure code in the example is straightforward: a join query against the base tables (primary and secondary tables for the `User` class) and a restriction by `RANKING`, an input argument to the procedure.

You must observe a few rules for stored procedures mapped in Hibernate. Stored procedures support `IN` and `OUT` parameters. If you use stored procedures with Oracle's own JDBC drivers, Hibernate requires that the first parameter of the stored procedure is an `OUT`; and for stored procedures that are supposed to be used for queries, the query result is supposed to be returned in this parameter. In Oracle 9 or newer, the type of the `OUT` parameter has to be a `SYS_REFCURSOR`. In older versions of Oracle, you must define your own reference cursor type first, called `REF CURSOR`—examples can be found in Oracle product documentation. All other major database management systems (and drivers for the Oracle DBMS not from Oracle) are JDBC-compliant, and you can return a result directly in the stored procedure without using an `OUT` parameter. For example, a similar procedure in Microsoft SQL Server would look as follows:

```
create procedure SELECT_USERS_BY_RANK
  @IN_RANK int
  as
    select
       us.USER_ID         as USER_ID,
       us.FIRSTNAME       as FIRSTNAME,
       us.LASTNAME        as LASTNAME,
```

```
    ...
  from
    USERS us
  where us.RANKING >= @IN_RANK
```

Let's map this stored procedure to a named query in Hibernate.

Querying with a procedure

A stored procedure for querying is mapped as a regular named query, with some minor differences:

```
<sql-query name="loadUsersByRank" callable="true">
    <return alias="u" class="User">
        <return-property name="id"          column="USER_ID"/>
        <return-property name="firstname"   column="FIRSTNAME"/>
        <return-property name="lastname"    column="LASTNAME"/>
        <return-property name="username"    column="USERNAME"/>
        <return-property name="password"    column="PASSWD"/>
        <return-property name="email"       column="EMAIL"/>
        <return-property name="ranking"     column="RANKING"/>
        <return-property name="admin"       column="IS_ADMIN"/>
        <return-property name="created"     column="CREATED"/>
        <return-property name="homeAddress">
            <return-column name="HOME_STREET"/>
            <return-column name="HOME_ZIPCODE"/>
            <return-column name="HOME_CITY"/>
        </return-property>
        <return-property name="billingAddress">
            <return-column name="BILLING_STREET"/>
            <return-column name="BILLING_ZIPCODE"/>
            <return-column name="BILLING_CITY"/>
        </return-property>
        <return-property name="defaultBillingDetails"
                         column="DEFAULT_BILLING_DETAILS_ID"/>
   </return>
   { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

The first difference, compared to a regular SQL query mapping, is the `callable="true"` attribute. This enables support for callable statements in Hibernate and correct handling of the output of the stored procedure. The following mappings bind the column names returned in the procedures result to the properties of a `User` object. One special case needs extra consideration: If multicolumn properties, including components (`homeAddress`), are present in the class, you need to map their columns in the right order. For example, the `homeAddress` property is mapped as a `<component>` with three properties, each to its own

column. Hence, the stored procedure mapping includes three columns bound to the `homeAddress` property.

The `call` of the stored procedure prepares one `OUT` (the question mark) and a named input parameter. If you aren't using the Oracle JDBC drivers (other drivers or a different DBMS), you don't need to reserve the first `OUT` parameter; the result can be returned directly from the stored procedure.

Look at the regular class mapping of the `User` class. Notice that the column names returned by the procedure in this example are the same as the column names you already mapped. You can omit the binding of each property and let Hibernate take care of the mapping automatically:

```
<sql-query name="loadUsersByRank" callable="true">
    <return class="User"/>
   { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

The responsibility for returning the correct columns, for all properties and foreign key associations of the class with the same names as in the regular mappings, is now moved into the stored procedure code. Because you have aliases in the stored procedure already (`select ... us.FIRSTNAME as FIRSTNAME...`), this is straightforward. Or, if only some of the columns returned in the result of the procedure have different names than the ones you mapped already as your properties, you only need to declare these:

```
<sql-query name="loadUsersByRank" callable="true">
    <return class="User">
        <return-property name="firstname"   column="FNAME"/>
        <return-property name="lastname"    column="LNAME"/>
   </return>
   { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

Finally, let's look at the `call` of the stored procedure. The syntax you're using here, `{ call PROCEDURE() }`, is defined in the SQL standard and portable. A nonportable syntax that works for Oracle is `begin PROCEDURE(); end;`. It's recommended that you always use the portable syntax. The procedure has two parameters. As explained, the first is reserved as an output parameter, so you use a positional parameter symbol (`?`). Hibernate takes care of this parameter if you configured a dialect for an Oracle JDBC driver. The second is an input parameter you have to supply when executing the call. You can either use only positional parameters or mix named and positional parameters. We prefer named parameters for readability.

Querying with this stored procedure in the application looks like any other named query execution:

```
Query q = session.getNamedQuery("loadUsersByRank");
q.setParameter("rank", 12);
List result = q.list();
```

At the time of writing, mapped stored procedures can be enabled as named queries, as you did in this section, or as loaders for an entity, similar to the `loadUser` example you mapped earlier.

Stored procedures can not only query and load data, but also manipulate data. The first use case for this is mass data operations, executed in the database tier. You shouldn't map this in Hibernate but should execute it with plain JDBC: `session.connection().prepareCallableStatement();` and so on. The data-manipulation operations you can map in Hibernate are the creation, deletion, and update of an entity object.

Mapping CUD to a procedure

Earlier, you mapped `<sql-insert>`, `<sql-delete>`, and `<sql-update>` elements for a class to custom SQL statements. If you'd like to use stored procedures for these operations, change the mapping to callable statements:

```
<class name="User">
    ...

    <sql-update callable="true" check="none">
        { call UPDATE_USER(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?) }
    </sql-update>

</class>
```

With the current version of Hibernate, you have the same problem as before: the binding of values to the positional parameters. First, the stored procedure must have the same number of input parameters as expected by Hibernate (enable the SQL log as shown earlier to get a generated statement you can copy and paste). The parameters again must be in the same order as expected by Hibernate.

Consider the `check="none"` attribute. For correct (and, if you enabled it) optimistic locking, Hibernate needs to know whether this custom update operation was successful. Usually, for dynamically generated SQL, Hibernate looks at the number of updated rows returned from an operation. If the operation didn't or couldn't update any rows, an optimistic locking failure occurs. If you write your own custom SQL operation, you can customize this behavior as well.

With `check="none"`, Hibernate expects your custom procedure to deal internally with failed updates (for example, by doing a version check of the row that

needs to be updated) and expects your procedure to throw an exception if something goes wrong. In Oracle, such a procedure is as follows:

```
<database-object>
    <create>
       create or replace procedure UPDATE_USER
        (IN_FIRSTNAME  in varchar,
         IN_LASTNAME   in varchar,
         IN_PASSWORD   in varchar,
         ...
        )
      as
        rowcount INTEGER;
      begin

         update USERS set
           FIRSTNAME   = IN_FIRSTNAME,
           LASTNAME    = IN_LASTNAME,
           "PASSWORD"  = IN_PASSWORD,
         where
           OBJ_VERSION = ...;

         rowcount := SQL%ROWCOUNT;
         if rowcount != 1 then
             RAISE_APPLICATION_ERROR( -20001, 'Version check failed');
         end if;

      end;

    </create>
    <drop>
        drop procedure UPDATE_USER
    </drop>
</database-object>
```

The SQL error is caught by Hibernate and converted into an optimistic locking exception you can then handle in application code. Other options for the check attribute are as follows:

- If you enable `check="count"`, Hibernate checks the number of modified rows using the plain JDBC API. This is the default and used when you write dynamic SQL without stored procedures.
- If you enable `check="param"`, Hibernate reserves an `OUT` parameter to get the return value of the stored procedure call. You need to add an additional question mark to your call and, in your stored procedure, return the row count of your DML operation on this (first) `OUT` parameter. Hibernate then validates the number of modified rows for you.

Mappings for insertion and deletion are similar; all of these must declare how optimistic lock checking is performed. You can copy a template from the Hibernate startup log to get the correct order and number of parameters.

Finally, you can also map stored functions in Hibernate. They have slightly different semantics and use cases.

Mapping stored functions

A stored function only has input parameters—no output parameters. However, it can return a value. For example, a stored function can return the rank of a user:

```
<database-object>
    <create>
        create or replace function GET_USER_RANK
         (IN_USER_ID int)
        return int is
         RANK int;
        begin
            select
                RANKING
            into
                RANK
            from
                USERS
            where
                USER_ID = IN_USER_ID;

            return RANK;
        end;
    </create>
    <drop>
        drop function GET_USER_RANK
    </drop>
</database-object>
```

This function returns a scalar number. The primary use case for stored functions that return scalars is embedding a call in regular SQL or HQL queries. For example, you can retrieve all users who have a higher rank than a given user:

```
String q = "from User u where u.ranking > get_user_rank(:userId)";
List result = session.createQuery(q)
                .setParameter("userId", 123)
                .list();
```

This query is in HQL; thanks to the pass-through functionality for function calls in the `WHERE` clause (not in any other clause though), you can call any stored function in your database directly. The return type of the function should match the

operation: in this case, the greater-than comparison with the `ranking` property, which is also numeric.

If your function returns a resultset cursor, as in previous sections, you can even map it as a named query and let Hibernate marshal the resultset into an object graph.

Finally, remember that stored procedures and functions, especially in legacy databases, sometimes can't be mapped in Hibernate; in such cases you have to fall back to plain JDBC. Sometimes you can wrap a legacy stored procedure with another stored procedure that has the parameter interface expected by Hibernate. There are too many varieties and special cases to be covered in a generic mapping tool. However, future versions of Hibernate will improve mapping capabilities—we expect better handling of parameters (no more counting of question marks) and support for arbitrary input and output arguments to be available in the near future.

You've now completed customization of runtime SQL queries and DML. Let's switch perspective and customize the SQL used for the creation and modification of the database schema, the DDL.