SAMPLE CHAPTER

JAVA PERSISTENCE WITH HIBERNATE

SECOND EDITION

Christian Bauer
Gavin King
Gary Gregory

FOREWORD BY Linda DeMichiel

MANNING

Java Persistence with Hibernate

by Christian Bauer

Gavin King

Gary Gregory

Chapter 14

brief contents

14 Creating and executing queries

In this chapter

- The basic query APIs
- Creating and preparing queries
- Optimizing query execution

If you've been using handwritten SQL for a number of years, you may be concerned that ORM will take away some of the expressiveness and flexibility you're used to. This isn't the case with Hibernate and Java Persistence.

With Hibernate's and Java Persistence's powerful query facilities, you can express almost everything you commonly (or even uncommonly) need to express in SQL, but in object-oriented terms—using classes and properties of classes. Moreover, you can always fall back to SQL strings and let Hibernate do the heavy lifting of handling the query result. For additional SQL resources, consult our reference section.

Major new features in JPA 2

- A type-safe criteria API for the programmatic creation of queries is now available.
- You can now declare up front the type of a query result with the new `TypedQuery` interface.
- You can programmatically save a `Query` (JPQL, criteria, or native SQL) for later use as a named query.
- In addition to being able to set query parameters, hints, maximum results, and flush and lock modes, JPA 2 extends the `Query` API with various getter methods for obtaining the current settings.
- JPA now standardizes several query hints (timeout, cache usage).

In this chapter, we show you how to create and execute queries with JPA and the Hibernate API. The queries are as simple as possible so you can focus on the creation and execution API without unfamiliar languages possibly distracting. The next chapter will cover query languages.

Common to all APIs, a query must be prepared in application code before execution. There are three distinct steps:

1. Create the query, with any arbitrary selection, restriction, and projection of data that you want to retrieve.
2. Prepare the query: bind runtime arguments to query parameters, set hints, and set paging options. You can reuse the query with changing settings.
3. Execute the prepared query against the database and retrieve the data. You can control how the query is executed and how data should be retrieved into memory (all at once or piecemeal, for example).

Depending on the query options you use, your starting point for query creation is either the `EntityManager` or the native `Session` API. First up is creating the query.

14.1 Creating queries

JPA represents a query with a `javax.persistence.Query` or `javax.persistence.TypedQuery` instance. You create queries with the `EntityManager#createQuery()` method and its variants. You can write the query in the Java Persistence Query Language (JPQL), construct it with the `CriteriaBuilder` and `CriteriaQuery` APIs, or use plain SQL. (There is also `javax.persistence.StoredProcedureQuery`, covered in section 17.4.)

Hibernate has its own, older API to represent queries: `org.hibernate.Query` and `org.hibernate.SQLQuery`. We talk more about these in a moment. Let's start with the JPA standard interfaces and query languages.

14.1.1 The JPA query interfaces

Say you want to retrieve all `Item` entity instances from the database. With JPQL, this simple query string looks quite a bit like the SQL you know:

```
Query query = em.createQuery("select i from Item i");
```

The JPA provider returns a fresh `Query`; so far, Hibernate hasn't sent any SQL to the database. Remember that further preparation and execution of the query are separate steps.

JPQL is compact and will be familiar to anyone with SQL experience. Instead of table and column names, JPQL relies on entity class and property names. Except for these class and property names, JPQL is case-insensitive, so it doesn't matter whether you write `SeLEct` or `select`.

JPQL (and SQL) query strings can be simple Java literals in your code, as you saw in the previous example. Alternatively, especially in larger applications, you can move the query strings out of your data-access code and into annotations or XML. A query is then accessed by name with `EntityManager#createNamedQuery()`. We discuss externalized queries separately later in this chapter; there are many options to consider.

A significant disadvantage of JPQL surfaces as problems during refactoring of the domain model: if you rename the `Item` class, your JPQL query will break. (Some IDEs can detect and refactor JPQL strings, though.)

JPA and query languages: HQL vs. JPQL

Before JPA existed (and even today, in some documentation), the query language in Hibernate was called HQL. The differences between JPQL and HQL are insignificant now. Whenever you provide a query string to any query interface in Hibernate, either with the `EntityManager` or `Session`, it's a JPQL/HQL string. The same engine parses the query internally. The fundamental syntax and semantics are the same, although Hibernate, as always, supports some special constructs that aren't standardized in JPA. We'll tell you when a particular keyword or clause in an example only works in Hibernate. To simplify your life, think *JPQL* whenever you see *HQL*.

You can make query construction with `CriteriaBuilder` and `CriteriaQuery` APIs completely type-safe. JPA also calls this *query by criteria*:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
// Also available on EntityManagerFactory:
// CriteriaBuilder cb = entityManagerFactory.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery();
criteria.select(criteria.from(Item.class));

Query query = em.createQuery(criteria);
```

First you get a `CriteriaBuilder` from your `EntityManager` by calling `getCriteriaBuilder()`. If you don't have an `EntityManager` ready, perhaps because you want to create the query independently from a particular persistence context, you may obtain the `CriteriaBuilder` from the usually globally shared `EntityManagerFactory`.

You then use the builder to create any number of `CriteriaQuery` instances. Each `CriteriaQuery` has at least one root class specified with `from()`; in the last example, that's `Item.class`. This is called *selection*; we'll talk more about it in the next chapter. The shown query returns all `Item` instances from the database.

The `CriteriaQuery` API will appear seamless in your application, without string manipulation. It's the best choice when you can't fully specify the query at development time and the application must create it dynamically at runtime. Imagine that you have to implement a search mask in your application, with many check boxes, input fields, and switches the user can enable. You must dynamically create a database query from the user's chosen search options. With JPQL and string concatenation, such code would be difficult to write and maintain.

You can write strongly typed `CriteriaQuery` calls, without strings, using the static JPA metamodel. This means your queries will be safe and included in refactoring operations, as already shown in the section "Using a static metamodel" in chapter 3.

Creating a detached criteria query

You always need an `EntityManager` or `EntityManagerFactory` to get the JPA `CriteriaBuilder`. With the older native `org.hibernate.Criteria` API, you only need access to the root entity class to create a detached query, as shown in section 16.3.

If you need to use features specific to your database product, your only choice is native SQL. You can directly execute SQL in JPA and let Hibernate handle the result for you, with the `EntityManager#createNativeQuery()` method:

```
Query query = em.createNativeQuery(
    "select * from ITEM", Item.class
);
```

After execution of this SQL query, Hibernate reads the `java.sql.ResultSet` and creates a `List` of managed `Item` entity instances. Of course, all columns necessary to construct an `Item` must be available in the result, and an error is thrown if your SQL query doesn't return them properly.

In practice, the majority of the queries in your application will be trivial—easily expressed in JPQL or with a `CriteriaQuery`. Then, possibly during optimization, you'll find a handful of complex and performance-critical queries. You may have to use special and proprietary SQL keywords to control the optimizer of your DBMS product. Most developers then write SQL instead of JPQL and move such complex queries into an XML file, where, with the help of a DBA, you change them independently from Java code. Hibernate can still handle the query result for you; hence you integrate SQL into your JPA application. There is nothing wrong with using SQL in Hibernate; don't let some kind of ORM "purity" get in your way. When you have a special case, don't try to hide it, but rather expose and document it properly so the next engineer will understand what's going on.

In certain cases, it's useful to specify the type of data returned from a query.

14.1.2 Typed query results

Let's say you want to retrieve only a single `Item` with a query, given its identifier value:

```
Query query = em.createQuery(
    "select i from Item i where i.id = :id"
).setParameter("id", ITEM_ID);

Item result = (Item) query.getSingleResult();
```

In this example, you see a preview of parameter binding and query execution. The important bit is the return value of the `getSingleResult()` method. It's `java.lang.Object`, and you have to cast it to an `Item`.

If you provide the class of your return value when creating the query, you can skip the cast. This is the job of the `javax.persistence.TypedQuery` interface:

```
TypedQuery<Item> query = em.createQuery(
    "select i from Item i where i.id = :id", Item.class
).setParameter("id", ITEM_ID);

Item result = query.getSingleResult();     <—— No cast needed
```

Query by criteria also supports the `TypedQuery` interface:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(cb.equal(i.get("id"), ITEM_ID));

TypedQuery<Item> query = em.createQuery(criteria);

Item result = query.getSingleResult();     <—— No cast needed
```

Note that this `CriteriaQuery` isn't completely type-safe: the `Item#id` property is addressed with a string in `get("id")`. In chapter 3's "Using a static metamodel," you saw how you can make such queries completely type-safe with static metamodel classes.

Hibernate is older than even the first version of JPA, so it also has its own query APIs.

Hibernate Feature

14.1.3 Hibernate's query interfaces

Hibernate's own query representations are `org.hibernate.Query` and `org.hibernate.SQLQuery`. As usual, they offer more than is standardized in JPA, at the cost of portability. They're also much older than JPA, so there is some feature duplication.

Your starting point for Hibernate's query API is the `Session`:

```
Session session = em.unwrap(Session.class);
org.hibernate.Query query = session.createQuery("select i from Item i");
// Proprietary API: query.setResultTransformer(...);
```

You write the query string in standard JPQL. Compared with `javax.persistence.Query`, the `org.hibernate.Query` API has some additional proprietary methods that are only available in Hibernate. You see more of the API later in this and the following chapters.

Hibernate also has its own SQL result-mapping facility, with `org.hibernate.SQLQuery`:

```
Session session = em.unwrap(Session.class);

org.hibernate.SQLQuery query = session.createSQLQuery(
    "select {i.*} from ITEM {i}"
).addEntity("i", Item.class);
```

This example relies on placeholders in the SQL string to map columns of the `java.sql.ResultSet` to entity properties. We'll talk more about integration of SQL queries with this proprietary and the standard JPA result mapping in section 17.2.

Hibernate also has an older, proprietary `org.hibernate.Criteria` query API:

```
Session session = em.unwrap(Session.class);

org.hibernate.Criteria query = session.createCriteria(Item.class);
query.add(org.hibernate.criterion.Restrictions.eq("id", ITEM_ID));

Item result = (Item) query.uniqueResult();
```

You can also access the proprietary Hibernate query API given a `javax.persistence.Query`, by unwrapping an `org.hibernate.jpa.HibernateQuery` first:

```
javax.persistence.Query query = em.createQuery(
    // ...
);

org.hibernate.Query hibernateQuery =
    query.unwrap(org.hibernate.jpa.HibernateQuery.class)
        .getHibernateQuery();

hibernateQuery.getQueryString();
hibernateQuery.getReturnAliases();
// ... other proprietary API calls
```

We focus on the standard API and later show you some rarely needed advanced options only available with Hibernate's API, such as *scrolling with cursors* and *query by example*.

After writing your query, and before executing it, you typically want to further prepare the query by setting parameters applicable to a particular execution.

14.2 Preparing queries

A query has several aspects: it defines what data should be loaded from the database and the restrictions that apply, such as the identifier of an `Item` or the name of a `User`. When you write a query, you shouldn't code these arguments into the query string using string concatenation. You should use parameter placeholders instead and then bind the argument values before execution. This allows you to reuse the query with different argument values while keeping you're safe from SQL injection attacks.

Depending on your user interface, you frequently also need *paging*. You limit the number of rows returned from the database by your query. For example, you may want to return only result rows 1 to 20 because you can only show so much data on each screen, then a bit later you want rows 21 to 40, and so on.

Let's start with parameter binding.

14.2.1 Protecting against SQL injection attacks

Without runtime parameter binding, you're forced to write bad code:

```
String searchString = getValueEnteredByUser();     <—— Never do this!

Query query = em.createQuery(
    "select i from Item i where i.name = '" + searchString + "'"
);
```

You should never write code like this, because a malicious user could craft a search string to execute code on the database you didn't expect or want—that is, by entering the value of `searchString` in a search dialog box as `foo' and callSomeStoredProcedure() and 'bar' = 'bar`.

As you can see, the original `searchString` is no longer a simple search for a string but also executes a stored procedure in the database! The quote characters aren't escaped; hence the call to the stored procedure is another valid expression in the query. If you write a query like this, you open a major security hole in your application by allowing the execution of arbitrary code on your database. This is an *SQL injection* attack. Never pass unchecked values from user input to the database! Fortunately, a simple mechanism prevents this mistake.

The JDBC API includes functionality for safely binding values to SQL parameters. It knows exactly what characters in the parameter value to escape so the previous vulnerability doesn't exist. For example, the database driver escapes the single-quote characters in the given `searchString` and no longer treats them as control characters but as a part of the search string value. Furthermore, when you use parameters, the database can efficiently cache precompiled prepared statements, improving performance significantly.

There are two approaches to parameter binding: *named* and *positional* parameters. JPA support both options, but you can't use both at the same time for a particular query.

14.2.2 Binding named parameters

With named parameters, you can rewrite the query in the previous section as follows:

```
String searchString = // ...

Query query = em.createQuery(
    "select i from Item i where i.name = :itemName"
).setParameter("itemName", searchString);
```

The colon followed by a parameter name indicates a named parameter, here `itemName`. In a second step, you bind a value to the `itemName` parameter. This code is cleaner, much safer, and performs better, because you can reuse a single compiled SQL statement if only parameter values change.

You can get a `Set` of `Parameters` from a `Query`, either to obtain more information about each parameter (such as name or required Java type) or to verify that you've bound all parameters properly before execution:

```
for (Parameter<?> parameter : query.getParameters()) {
    assertTrue(query.isBound(parameter));
}
```

The `setParameter()` method is a generic operation that can bind all types of arguments. It only needs a little help for temporal types:

```
Date tomorrowDate = // ...

Query query = em.createQuery(
    "select i from Item i where i.auctionEnd > :endDate"
).setParameter("endDate", tomorrowDate, TemporalType.TIMESTAMP);
```

Hibernate needs to know whether you want only the date or time or the full timestamp bound.

For convenience, an entity instance can also be passed to the `setParameter()` method:

```
Item someItem = // ...

Query query = em.createQuery(
    "select b from Bid b where b.item = :item"
).setParameter("item", someItem);
```

Hibernate binds the identifier value of the given `Item`. You later see that `b.item` is a shortcut for `b.item.id`.

For criteria queries, there is a long way and a short way to bind parameters:

```
String searchString = // ...

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);

Query query = em.createQuery(
    criteria.select(i).where(
        cb.equal(
            i.get("name"),
            cb.parameter(String.class, "itemName")
        )
    )
).setParameter("itemName", searchString);
```

Here you put the `itemName` parameter placeholder of type `String` into the `CriteriaQuery` and then bind a value to it as usual with the `Query#setParameter()` method.

Alternatively, with a `ParameterExpression`, you don't have to name the placeholder, and the binding of the argument is type-safe (you can't bind an `Integer` to a `ParameterExpression<String>`):

```
String searchString = // ...

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery criteria = cb.createQuery(Item.class);
Root<Item> i = criteria.from(Item.class);

ParameterExpression<String> itemNameParameter =
    cb.parameter(String.class);

Query query = em.createQuery(
    criteria.select(i).where(
        cb.equal(
            i.get("name"),
            itemNameParameter
        )
    )
).setParameter(itemNameParameter, searchString);
```

A rarely used and less safe option for value binding is positional query parameters.

14.2.3 *Using positional parameters*

If you prefer, you can use positional parameters instead of named parameters:

```
Query query = em.createQuery(
    "select i from Item i where i.name like ?1 and i.auctionEnd > ?2"
);
query.setParameter(1, searchString);
query.setParameter(2, tomorrowDate, TemporalType.TIMESTAMP);
```

In this example, the positional parameter markers are indexed `?1` and `?2`. You may know this type of parameter placeholder from JDBC, but without the numbers and only the question marks. JPA requires that you enumerate the placeholders, starting with 1.

> **NOTE** In Hibernate both styles work, so be careful! Hibernate will warn you about a brittle query if you use JDBC-style positional parameters with only question marks.

Our recommendation is to avoid positional parameters. They may be more convenient if you build complex queries programmatically, but the `CriteriaQuery` API is a much better alternative for that purpose.

After binding parameters to your query, you may want to enable *pagination* if you can't display all results at once.

14.2.4 Paging through large result sets

A commonly used technique to process large result sets is *paging*. Users may see the result of their search request (for example, for specific items) as a page. This page shows a limited subset (say, 10 items) at a time, and users can navigate to the next and previous pages manually to view the rest of the result.

The `Query` interface supports paging of the query result. In this query, the requested page starts in the middle of the result set:

```
Query query = em.createQuery("select i from Item i");
query.setFirstResult(40).setMaxResults(10);
```

Starting from the fortieth row, you retrieve the next 10 rows. The call to `setFirstResults(40)` starts the result set at row 40. The call to `setMaxResults(10)` limits the query result set to 10 rows returned by the database. Because there is no standard way to express paging in SQL, Hibernate knows the tricks to make this work efficiently on your particular DBMS.

It's crucially important to remember that paging operates at the SQL level, on result rows. Limiting a result to 10 rows isn't necessarily the same as limiting the result to 10 instances of `Item`! In section 15.4.5, you'll see some queries with *dynamic fetching* that can't be combined with row-based paging at the SQL level, and we'll discuss this issue again.

You can even *add* this flexible paging option to an SQL query:

```
Query query = em.createNativeQuery("select * from ITEM");
query.setFirstResult(40).setMaxResults(10);
```

Hibernate will rewrite your SQL query to include the necessary keywords and clauses for limiting the number of returned rows to the page you specified.

In practice, you frequently combine paging with a special count-query. If you show a page of items, you also let the user know the total count of items. In addition, you need this information to decide whether there are more pages to show and whether the user can click to the next page. This usually requires two slightly different queries: for example, `select i from Item i` combined with `setMaxResults()` and `setFirstResult()` would retrieve a page of items, and `select count(i) from Item i` would retrieve the total number of items available.

Hibernate Feature

Maintaining two almost identical queries is overhead you should avoid. A popular trick is to write only one query but execute it with a database cursor first to get the total result count:

```
Query query = em.createQuery("select i from Item i");

org.hibernate.Query hibernateQuery =                              ◁ 1 Unwraps API
    query.unwrap(org.hibernate.jpa.HibernateQuery.class).getHibernateQuery();

org.hibernate.ScrollableResults cursor =                          ◁ 2 Executes query with cursor
        hibernateQuery.scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

cursor.last();                                    ◁ 3 Counts rows
int count = cursor.getRowNumber()+1;

cursor.close();                                   ◁ 4 Closes cursor

query.setFirstResult(40).setMaxResults(10);       ◁ 5 Gets arbitrary data
```

1. Unwrap the Hibernate API to use scrollable cursors.
2. Execute the query with a database cursor; this doesn't retrieve the result set into memory.
3. Jump to the last row of the result in the database, and then get the row number. Because row numbers are zero-based, add 1 to get the total count of rows.
4. You must close the database cursor.
5. Execute the query again, and retrieve an arbitrary page of data.

There is one significant problem with this convenient strategy: your JDBC driver and/or DBMS may not support database cursors. Even worse, cursors seem to work, but the data is silently retrieved into application memory; the cursor isn't operating directly on the database. Oracle and MySQL drivers are known to be problematic, and we have more to say about scrolling and cursors in the next section. Later in this book, in section 19.2, we'll further discuss paging strategies in an application environment.

Your query is now ready for execution.

14.3 Executing queries

Once you've created and prepared a `Query`, you're ready to execute it and retrieve the result into memory. Retrieving the entire result set into memory in one go is the most common way to execute a query; we call this *listing*. Some other options are available that we also discuss next, such as *scrolling* and *iterating*.

14.3.1 Listing all results

The `getResultList()` method executes the `Query` and returns the results as a `java.util.List`:

```
Query query = em.createQuery("select i from Item i");
List<Item> items = query.getResultList();
```

Hibernate executes one or several SQL `SELECT` statements immediately, depending on your fetch plan. If you map any associations or collections with `FetchType.EAGER`, Hibernate must fetch them in addition to the data you want retrieved with your query. All data is loaded into memory, and any entity instances that Hibernate retrieves are in persistent state and managed by the persistence context.

Of course, the persistence context doesn't manage scalar projection results. The following query returns a `List` of `String`s:

```
Query query = em.createQuery("select i.name from Item i");
List<String> itemNames = query.getResultList();
```

With some queries, you know the result is only a single result—for example, if you want only the highest `Bid` or only one `Item`.

14.3.2 Getting a single result

You may execute a query that returns a single result with the `getSingleResult()` method:

```
TypedQuery<Item> query = em.createQuery(
    "select i from Item i where i.id = :id", Item.class
).setParameter("id", ITEM_ID);

Item item = query.getSingleResult();
```

The call to `getSingleResult()` returns an `Item` instance. This also works for scalar results:

```
TypedQuery<String> query = em.createQuery(
    "select i.name from Item i where i.id = :id", String.class
).setParameter("id", ITEM_ID);

String itemName = query.getSingleResult();
```

Now, the ugly bits: if there are no results, `getSingleResult()` throws a `NoResult-Exception`. This query tries to find an item with a nonexistent identifier:

```
try {
    TypedQuery<Item> query = em.createQuery(
        "select i from Item i where i.id = :id", Item.class
    ).setParameter("id", 1234l);

    Item item = query.getSingleResult();
    // ...

} catch (NoResultException ex) {
    // ...
}
```

You'd expect a `null` for this type of perfectly benign query. This is rather tragic, because it forces you to guard this code with a `try/catch` block. In fact, it forces you to *always* wrap a call of `getSingleResult()`, because you can't know whether the row(s) will be present.

If there's more than one result, `getSingleResult()` throws a `NonUniqueResultException`. This usually happens with this kind of query:

```
try {
    Query query = em.createQuery(
        "select i from Item i where name like '%a%'"
    );

    Item item = (Item) query.getSingleResult();
    // ...

} catch (NonUniqueResultException ex) {
    // ...
}
```

Retrieving all results into memory is the most common way to execute a query. Hibernate supports some other methods that you may find interesting if you want to optimize a query's memory consumption and execution behavior.

Hibernate Feature

14.3.3 Scrolling with database cursors

Plain JDBC provides a feature called *scrollable result sets*. This technique uses a cursor that the database management system holds. The cursor points to a particular row in the result of a query, and the application can move the cursor forward and backward. You can even directly jump to a row with the cursor.

One of the situations where you should scroll through the results of a query instead of loading them all into memory involves result sets that are too large to fit into memory. Usually you try to restrict the result further by tightening the conditions in the query. Sometimes this isn't possible, maybe because you need all the data but want to retrieve it in several steps. We'll show such a batch-processing routine in section 20.1.

JPA doesn't standardize scrolling through results with database cursors, so you need the `org.hibernate.ScrollableResults` interface available on the proprietary `org.hibernate.Query`:

```
Session session = em.unwrap(Session.class);

org.hibernate.Query query = session.createQuery(          <- 1 Creates query
    "select i from Item i order by i.id asc"
);                                                          2 Opens cursor

org.hibernate.ScrollableResults cursor =                  <-
        query.scroll(org.hibernate.ScrollMode.SCROLL_INSENSITIVE);

cursor.setRowNumber(2);                         <- 3 Jumps to third row
```

```
Item item = (Item) cursor.get(0);          <-- 4 Gets column value

cursor.close();                            <-- 5 Closes cursor
```

Start by creating an `org.hibernate.Query` ❶ and opening a cursor ❷. You then ignore the first two result rows, jump to the third row ❸, and get that row's first "column" value ❹. There are no columns in JPQL, so this is the first projection element: here, `i` in the `select` clause. More examples of projection are available in the next chapter. *Always* close the cursor ❺ before you end the database transaction!

As mentioned earlier in this chapter, you can also `unwrap()` the Hibernate query API from a regular `javax.persistence.Query` you've constructed with `CriteriaBuilder`. A proprietary `org.hibernate.Criteria` query can also be executed with scrolling instead of `list()`; the returned `ScrollableResults` cursor works the same.

The `ScrollMode` constants of the Hibernate API are equivalent to the constants in plain JDBC. In the previous example, `ScrollMode.SCROLL_INSENSITIVE` means the cursor isn't sensitive to changes made in the database, effectively guaranteeing that no dirty reads, unrepeatable reads, or phantom reads can slip into your result set while you scroll. Other available modes are `SCROLL_SENSITIVE` and `FORWARD_ONLY`. A sensitive cursor exposes you to committed modified data while the cursor is open; and with a forward-only cursor, you can't jump to an absolute position in the result. Note that the Hibernate persistence context cache still provides repeatable read for entity instances even with a sensitive cursor, so this setting can only affect modified scalar values you project in the result set.

Be aware that some JDBC drivers don't support scrolling with database cursors properly, although it might seem to work. With MySQL drivers, for example, the drivers always retrieve the entire result set of a query into memory immediately; hence you only scroll through the result set in application memory. To get real row-by-row streaming of the result, you have to set the JDBC fetch size of the query to `Integer.MIN_VALUE` (as explained in section 14.5.4) and only use `ScrollMode.FORWARD_ONLY`. Check the behavior and documentation of your DBMS and JDBC driver before using cursors.

An important limitation of scrolling with a database cursor is that it can't be combined with dynamic fetching with the `join fetch` clause in JPQL. Join fetching works with potentially several rows at a time, so you can't retrieve data row by row. Hibernate will throw an exception if you try to `scroll()` a query with dynamic fetch clauses.

Another alternative to retrieving all data at once is *iteration*.

Hibernate Feature

14.3.4 Iterating through a result

Let's say you know that most of the entity instances your query will retrieve are already present in memory. They may be in the persistence context or in the second-level shared cache (see section 20.2). In such a case, it might make sense to *iterate* the query result with the proprietary `org.hibernate.Query` API:

```
Session session = em.unwrap(Session.class);

org.hibernate.Query query = session.createQuery(
    "select i from Item i"
);

Iterator<Item> it = query.iterate(); // select ID from ITEM
while (it.hasNext()) {
    Item next = it.next(); // select * from ITEM where ID = ?
    // ...
}

Hibernate.close(it);
```

Iterator must be closed, either when the Session is closed or manually

When you call `query.iterate()`, Hibernate executes your query and sends an SQL `SELECT` to the database. But Hibernate slightly modifies the query and, instead of retrieving all columns from the `ITEM` table, only retrieves the identifier/primary key values.

Then, every time you call `next()` on the `Iterator`, an additional SQL query is triggered and the rest of the `ITEM` row is loaded. Obviously, this will cause an *n+1 selects problem* unless Hibernate can avoid the additional queries on `next()`. This will be the case if Hibernate can find the item's data in either the persistence context cache or the second-level cache.

The `Iterator` returned by `iterate()` must be closed. Hibernate closes it automatically when the `EntityManager` or `Session` is closed. If your iteration procedure exceeds the maximum number of open cursors in your database, you can close the `Iterator` manually with `Hibernate.close(iterator)`.

Iteration is rarely useful, considering that in the example all auction items would have to be in the caches to make this routine perform well. Like scrolling with a cursor, you can't combine it with dynamic fetching and `join fetch` clauses; Hibernate will throw an exception if you try.

So far, the code examples have all embedded query string literals in Java code. This isn't unreasonable for simple queries, but when you begin considering complex queries that must be split over multiple lines, it gets a bit unwieldy. Instead, you can give each query a name and move it into annotations or XML files.

14.4 Naming and externalizing queries

Externalizing query strings lets you store all queries related to a particular persistent class (or a set of classes) with the other metadata for that class. Alternatively, you can bundle your queries into an XML file, independent of any Java class. This technique is often preferred in larger applications; hundreds of queries are easier to maintain in a few well-known places rather than scattered throughout the code base in various classes accessing the database. You reference and access an externalized query by its name.

14.4.1 Calling a named query

The `EntityManager#getNamedQuery()` method obtains a `Query` instance for a named query:

```
Query query = em.createNamedQuery("findItems");
```

You can also obtain a `TypedQuery` instance for a named query:

```
TypedQuery<Item> query = em.createNamedQuery("findItemById", Item.class);
```

Hibernate's query API also supports accessing named queries:

```
org.hibernate.Query query = session.getNamedQuery("findItems");
```

Named queries are global—that is, the name of a query is a unique identifier for a particular persistence unit or `org.hibernate.SessionFactory`. How and where they're defined, in XML files or annotations, is no concern of your application code. On startup, Hibernate loads named JPQL queries from XML files and/or annotations and parses them to validate their syntax. (This is useful during development, but you may want to disable this validation in production, for a faster bootstrap, with the persistence unit configuration property `hibernate.query.startup_check`.)

Even the query language you use to write a named query doesn't matter. It can be JPQL or SQL.

14.4.2 Defining queries in XML metadata

You can place a named query in any JPA `<entity-mappings>` element in your orm.xml metadata. In larger applications, we recommend isolating and separating all named queries into their own file. Alternatively, you may want the same XML mapping file to define the queries and a particular class.

The `<named-query>` element defines a named JPQL query:

PATH: /model/src/main/resources/querying/ExternalizedQueries.xml

```
<entity-mappings
   version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
           http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

    <named-query name="findItems">
        <query><![CDATA[
            select i from Item i
        ]]></query>
    </named-query>

</entity-mappings>
```

You should wrap the query text into a CDATA instruction so any characters in your query string that may accidentally be considered XML (such as the *less than* operator) don't confuse the XML parser. We omit CDATA from most other examples for clarity.

Named queries don't have to be written in JPQL. They may even be native SQL queries—and your Java code doesn't need to know the difference:

PATH: /model/src/main/resources/querying/ExternalizedQueries.xml

```
<named-native-query name="findItemsSQL"
                    result-class="org.jpwh.model.querying.Item">
    <query>select * from ITEM</query>
</named-native-query>
```

This is useful if you think you may want to optimize your queries later by fine-tuning the SQL. It's also a good solution if you have to port a legacy application to JPA/Hibernate, where SQL code can be isolated from the hand-coded JDBC routines. With named queries, you can easily port the queries one by one to mapping files.

Hibernate Feature

Hibernate also has its own, nonstandard facility for externalized queries in Hibernate XML metadata files:

PATH: /model/src/main/resources/querying/ExternalizedQueries.hbm.xml

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <query name="findItemsOrderByAuctionEndHibernate">
        select i from Item i order by i.auctionEnd asc
    </query>

    <sql-query name="findItemsSQLHibernate">
        <return class="org.jpwh.model.querying.Item"/>
        select * from ITEM order by NAME asc
    </sql-query>

</hibernate-mapping>
```

You'll see more examples of externalized and especially custom SQL queries later in chapter 17.

If you don't like XML files, you can bundle and name your queries in Java annotation metadata.

14.4.3 Defining queries with annotations

JPA supports named queries with the @NamedQuery and @NamedNativeQuery annotations. You can only place these annotations on a mapped class. Note that the query name must again be globally unique in all cases; no class or package name is automatically prefixed to the query name:

```
@NamedQueries({
    @NamedQuery(
        name = "findItemById",
        query = "select i from Item i where i.id = :id"
    )
})
@Entity
public class Item {
    // ...
}
```

The class is annotated with an `@NamedQueries` containing an array of `@NamedQuery`. A single query can be declared directly; you don't need to wrap it in `@NamedQueries`. If you have an SQL instead of a JPQL query, use the `@NamedNativeQuery` annotation. There are many options to consider for mapping SQL result sets, so we'll show you how this works later, in a dedicated section in chapter 17.

Hibernate Feature

Unfortunately, JPA's named query annotations only work when they're on a mapped class. You can't put them into a package-info.java metadata file. Hibernate has some proprietary annotations for that purpose:

PATH: /model/src/main/java/org/jpwh/model/querying/package-info.java

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByName",
        query = "select i from Item i order by i.name asc"
    )
})
package org.jpwh.model.querying;
```

If neither XML files nor annotations seem to be the right place for defining your named queries, you might want to construct them programmatically.

14.4.4 Defining named queries programmatically

You can "save" a `Query` as a named query with the `EntityManagerFactory#addNamedQuery()` method:

```
Query findItemsQuery = em.createQuery("select i from Item i");
em.getEntityManagerFactory().addNamedQuery(
    "savedFindItemsQuery", findItemsQuery
);

Query query =
                                                       Later, with the same
em.createNamedQuery("savedFindItemsQuery");    <---    EntityManagerFactory
```

This registers your query with the persistence unit, the EntityManagerFactory, and make it reusable as a named query. The saved Query doesn't have to be a JPQL statement; you can also save a criteria or native SQL query. Typically, you register your queries once, on startup of your application.

We leave it to you to decide whether you want to use the named query feature. But we consider query strings in the application code (unless they're in annotations) to be the second choice; you should always externalize query strings if possible. In practice, XML files are probably the most versatile option.

Finally, for some queries, you may need extra settings and hints.

14.5 Query hints

In this section, we introduce some additional query options from the JPA standard and some proprietary Hibernate settings. As the name implies, you probably won't need them right away, so you can skip this section if you like and read it later as a reference.

All the examples use the same query, shown here:

```
String queryString = "select i from Item i";
```

In general, you can set a hint on a Query with the setHint() method. All the other query APIs, such as TypedQuery and StoredProcedureQuery, also have this method. If the persistence provider doesn't support a hint, the provider will ignore it silently.

JPA standardizes the hints shown in table 14.1.

Table 14.1 Standardized JPA query hints

Name	Value	Description
javax.persistence.query.timeout	(Milliseconds)	Sets the timeout for query execution. This hint is also available as a constant on org.hibernate.annotations.QueryHints.TIMEOUT_JPA.
javax.persistence.cache.retrieveMode	USE | BYPASS	Controls whether Hibernate tries to read data from the second-level shared cache when marshaling a query result, or bypasses the cache and only reads data from the query result.
javax.persistence.cache.storeMode	USE | BYPASS |REFRESH	Controls whether Hibernate stores data in the second-level shared cache when marshaling a query result.

Hibernate Feature

Hibernate has its own vendor-specific hints for queries, also available as constants on `org.hibernate.annotations.QueryHints`; see table 14.2.

Table 14.2 Hibernate query hints

Name	Value	Description
`org.hibernate.flushMode`	`org.hibernate.FlushMode` (Enum)	Controls whether and when the persistence context should be flushed before execution of the query
`org.hibernate.readOnly`	`true` | `false`	Enables or disables dirty checking for the managed entity instances returned by the query
`org.hibernate.fetchSize`	(JDBC fetch size)	Calls the JDBC `PreparedStatement#setFetchSize()` method before executing the query, an optimization hint for the database driver
`org.hibernate.comment`	(SQL comment string)	A comment to prepend to the SQL, useful for (database) logging

Second-level shared caching (especially query caching) is a complex issue, so we'll dedicate section 20.2 to it. You should read that section before enabling the shared cache: setting caching to "enabled" for a query will have no effect.

Some of the other hints also deserve a longer explanation.

14.5.1 Setting a timeout

You can control how long to let a query run by setting a *timeout*:

```
Query query = em.createQuery(queryString)
    .setHint("javax.persistence.query.timeout", 60000);     ◁——— 1 minute
```

With Hibernate, this method has the same semantics and consequences as the `setQueryTimeout()` method on the JDBC `Statement` API.

Note that a JDBC driver doesn't necessarily cancel the query precisely when the timeout occurs. The JDBC specification says, "Once the data source has had an opportunity to process the request to terminate the running command, a `SQLException` will be thrown to the client …." Hence, there is room for interpretation as to when exactly the data source has an opportunity to terminate the command. It might only be after the execution completes. You may want to test this with your DBMS product and driver.

You can also specify this timeout hint as a global default property in persistence.xml as a property when creating the `EntityManagerFactory` or as a named query option. The `Query#setHint()` method then overrides this global default for a particular query.

14.5.2 Setting the flush mode

Let's assume that you make modifications to persistent entity instances before executing a query. For example, you modify the `name` of managed `Item` instances. These modifications are only present in memory, so Hibernate by default *flushes* the persistence context and all changes to the database before executing your query. This guarantees that the query runs on current data and that no conflict between the query result and the in-memory instances can occur.

This may be impractical at times, if you execute a sequence that consists of many query-modify-query-modify operations, and each query is retrieving a different data set than the one before. In other words, you sometimes know you don't need to flush your modifications to the database before executing a query, because conflicting results aren't a problem. Note that the persistence context provides repeatable read for entity instances, so only scalar results of a query are a problem anyway.

You can disable flushing of the persistence context before a query with the `org.hibernate.flushMode` hint on a `Query` and the value `org.hibernate.FlushMode.COMMIT`. Fortunately, JPA has a standard `setFlushMode()` method on the `EntityManager` and `Query` API, and `FlushModeType.COMMIT` is also standardized. So, if you want to disable flushing only before a particular query, use the standard API:

```
Query query = em.createQuery(queryString)
    .setFlushMode(FlushModeType.COMMIT);
```

With the flush mode set to `COMMIT`, Hibernate won't flush the persistence context before executing the query. The default is `AUTO`.

14.5.3 Setting read-only mode

In section 10.2.8, we talked about how you can reduce memory consumption and prevent long dirty-checking cycles. You can tell Hibernate that it should consider all entity instances returned by a query as read-only (although not detached) with a hint:

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.READ_ONLY,
        true
    );
```

All `Item` instances returned by this query are in persistent state, but Hibernate doesn't enable snapshot for automatic dirty checking in the persistence context. Hibernate doesn't persist any modifications automatically unless you disable read-only mode with `session.setReadOnly(item, false)`.

14.5.4 Setting a fetch size

The *fetch size* is an optimization hint for the database driver:

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.FETCH_SIZE,
        50
    );
```

This hint may not result in any performance improvement if the driver doesn't implement this functionality. If it does, it can improve the communication between the JDBC client and the database by retrieving many rows in one batch when the client (Hibernate) operates on a query result (that is, on a `ResultSet`).

14.5.5 Setting an SQL comment

When you optimize an application, you often have to read complex SQL logs. We highly recommend that you enable the property `hibernate.use_sql_comments` in your persistence.xml configuration. Hibernate will then add an auto-generated comment to each SQL statement it writes to the logs.

You can set a custom comment for a particular `Query` with a hint:

```
Query query = em.createQuery(queryString)
    .setHint(
        org.hibernate.annotations.QueryHints.COMMENT,
        "Custom SQL comment"
    );
```

The hints you've been setting so far are all related to Hibernate or JDBC handling. Many developers (and DBAs) consider a query hint to be something completely different. In SQL, a query hint is an instruction in the SQL statement for the optimizer of the DBMS. For example, if the developer or DBA thinks the execution plan selected by the database optimizer for a particular SQL statement isn't the fastest, they use a hint to force a different execution plan. Hibernate and Java Persistence don't support arbitrary SQL hints with an API; you'll have to fall back to native SQL and write your own SQL statement—you can of course execute that statement with the provided APIs.

On the other hand, with some DBMS products, you can control the optimizer with an SQL comment at the beginning of an SQL statement. In that case, use the comment hint as shown in the last example.

In all previous examples, you've set the query hint directly on the `Query` instance. If you have externalized and named queries, you must set hints in annotations or XML.

14.5.6 Named query hints

All the query hints set earlier with `setHint()` can also be set in XML metadata in a `<named-query>` or `<named-native-query>` element:

```
<entity-mappings
   version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
            http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

    <named-query name="findItems">
        <query><![CDATA[
            select i from Item i
        ]]></query>
        <hint name="javax.persistence.query.timeout" value="60000"/>
        <hint name="org.hibernate.comment" value="Custom SQL comment"/>
    </named-query>

</entity-mappings>
```

You can set hints on named queries defined in annotations:

```
@NamedQueries({
    @NamedQuery(
        name = "findItemByName",
        query = "select i from Item i where i.name like :name",
        hints = {
            @QueryHint(
                name = org.hibernate.annotations.QueryHints.TIMEOUT_JPA,
                value = "60000"),
            @QueryHint(
                name = org.hibernate.annotations.QueryHints.COMMENT,
                value = "Custom SQL comment")
        }
    )
})
```

Hibernate Feature

Hints can be set on named queries in Hibernate annotations in a package-info.java file:

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemBuyNowPriceGreaterThan",
        query = "select i from Item i where i.buyNowPrice > :price",
        timeout = 60,                                                  <—— Seconds!
        comment = "Custom SQL comment"
    )
})

package org.jpwh.model.querying;
```

Hibernate Feature

In addition, of course, hints can be set on named queries externalized into a Hibernate XML metadata file:

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="http://www.hibernate.org/xsd/orm/hbm">

    <query name="findItemsOrderByAuctionEndHibernateWithHints"
           cache-mode="ignore"
           comment="Custom SQL comment"
           fetch-size="50"
           read-only="true"
           timeout="60">
        select i from Item i order by i.auctionEnd asc
    </query>

</hibernate-mapping>
```

14.6 Summary

- You learned how to create and execute queries. To create queries, you use the JPA query interfaces and process typed query results. You also saw Hibernate's own query interfaces.
- You then looked at how to prepare queries, taking care to protect against SQL injection attacks. You learned how to use bound and positional parameters, and you paged through large result sets.
- Instead of embedding JPQL in Java sources, you can name and externalize queries. You saw how to call a named query and different ways to define queries: in XML metadata, with annotations, and programmatically.
- We discussed the query hints you can give Hibernate: setting a timeout, flush mode, read-only mode, fetch size, and SQL comment. As with JPQL, you saw how to name and externalize query hints.

JAVA/HIBERNATE

Java Persistence with Hibernate SECOND EDITION

Bauer • King • Gregory

Persistence—the ability of data to outlive an instance of a program—is central to modern applications. Hibernate, the most popular Java persistence tool, offers automatic and transparent object/relational mapping, making it a snap to work with SQL databases in Java applications.

Java Persistence with Hibernate explores Hibernate by developing an application that ties together hundreds of individual examples. You'll immediately dig into the rich programming model of Hibernate, working through mappings, queries, fetching strategies, transactions, conversations, caching, and more. Along the way you'll find a well-illustrated discussion of best practices in database design and optimization techniques. In this revised edition, authors Christian Bauer, Gavin King, and Gary Gregory cover Hibernate 5 in detail with the Java Persistence 2.1 standard (JSR 338). All examples have been updated for the latest Hibernate and Java EE specification versions.

What's Inside

- Object/relational mapping concepts
- Efficient database application design
- Comprehensive Hibernate and Java Persistence reference
- Integration of Java Persistence with EJB, CDI, JSF, and JAX-RS
- Unmatched breadth and depth

The book assumes a working knowledge of Java.

Christian Bauer is a member of the Hibernate developer team and a trainer and consultant. Gavin King is the founder of the Hibernate project and a member of the Java Persistence expert group (JSR 220). Gary Gregory is a principal software engineer working on application servers and legacy integration.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/java-persistence-with-hibernate-second-edition

Free eBook
SEE INSERT

"The most comprehensive book about Hibernate Persistence ... works well both as a tutorial and as a reference."
—Sergio Fernandez Gonzalez
Accenture Software

"The essential guidebook for navigating the intricacies of Hibernate."
—José Diaz, OptumHealth

"An excellent update to a classic and essential book."
—Jerry Goodnough
Cognitive Medical Systems

"The must-have reference for every Hibernate user."
—Stephan Heffner
SPIEGEL-Verlag Rudolf Augstein
GmbH & Co. KG

ISBN 13: 978-1-61729-045-9
ISBN 10: 1-61729-045-9
55999
9 781617 290459

MANNING $59.99 / Can $68.99 [INCLUDING eBOOK]