A Short Guide to Hibernate 7

Version 7.3.0.CR1

Table of Contents

Preface

Hibernate 6 was a major redesign of the world's most popular and feature-rich ORM solution. The redesign touched almost every subsystem of Hibernate, including the APIs, mapping annotations, and the query language. This new Hibernate was suddenly more powerful, more robust, more portable, and more type safe.

Hibernate 7 builds on this foundation, adds support for JPA 3.2, and introduces Hibernate Data Repositories, an implementation of the Jakarta Data specification. Taken together, these enhancements yield a level of compile-time type safety—and resulting developer productivity—which was previously impossible. Hibernate Data Repositories offers truly seamless integration of the ORM solution with the persistence layer, obsoleting older add-on repository frameworks.

Hibernate ORM and Hibernate Reactive are core components of Quarkus 3, the most exciting new environment for cloud-native development in Java, and Hibernate remains the persistence solution of choice for almost every major Java framework or server.

Unfortunately, the changes in Hibernate 6 also obsoleted much of the information about Hibernate that's available in books, in blog posts, and on stackoverflow.

This guide is an up-to-date, high-level discussion of the current feature set and recommended usage. It does not attempt to cover every feature and should be used in conjunction with other documentation:

- Hibernate's extensive Javadoc,
- the Guide to Hibernate Query Language,
- Introducing Hibernate Data Repositories, and
- the Hibernate User Guide.

> The Hibernate User Guide includes detailed discussions of most aspects of Hibernate. But with so much information to cover, readability is difficult to achieve, and so it's most useful as a reference. Where necessary, we'll provide links to relevant sections of the User Guide.

Chapter 1. Introduction

Hibernate is usually described as a library that makes it easy to map Java classes to relational database tables. But this formulation does no justice to the central role played by the relational data itself. So a better description might be:

> Hibernate makes **relational data** visible to a program written in Java, in a **natural** and **typesafe** form,
>
> 1. making it easy to write complex queries and work with their results,
> 2. letting the program easily synchronize changes made in memory with the database, respecting the ACID properties of transactions, and
> 3. allowing performance optimizations to be made after the basic persistence logic has already been written.

Here the relational data is the focus, along with the importance of type safety. The goal of *object/relational mapping* (ORM) is to eliminate fragile and untypesafe code, and make large programs easier to maintain in the long run.

ORM takes the pain out of persistence by relieving the developer of the need to hand-write tedious, repetitive, and fragile code for flattening graphs of objects to database tables and rebuilding graphs of objects from flat SQL query result sets. Even better, ORM makes it much easier to tune performance later, after the basic persistence logic has already been written.

Actually, the problem that object/relational mapping solves has been mischaracterized for decades, and has very little to do with any so-called "mismatch" between classes and tables. The real problem is:

1. Data is stored in **normalized tables**.
2. The process of efficiently reading data from tables **denormalizes the data with joins**.
3. In an object-oriented program, we often want to work with data in something quite close to its **original normalized form**.

Therefore, the principal task of object/relational mapping is to *renormalize the data after reading it from the database*. This need exists even if the classes in our program are identical to the database tables.

> A perennial question is: should I use ORM, or plain SQL? The answer is usually: *use both*. JPA and Hibernate were designed to work *in conjunction with* handwritten SQL. You see, most programs with nontrivial data access logic will benefit from the use of ORM at least *somewhere*. But if Hibernate is making things more difficult, for some particularly tricky piece of data access logic, the only sensible thing to do is to use something better suited to the problem! Just because you're using Hibernate for persistence doesn't mean you have to use it for *everything*.

Let's underline the important point here: the goal of ORM is *not* to hide SQL or the relational model. After all, Hibernate's query language is nothing more than an object-oriented dialect of ANSI SQL.

> **Is Hibernate a "leaky abstraction"?**
>
> Hibernate—and ORM in general—has been accused of being a *leaky abstraction*, that is, of failing to completely hide the underlying relational database. Is this true? Well, as you can guess from what we've already said, the short answer is: yes, *and that's a good thing.* Of course, whether you consider an abstraction "leaky" depends on what you think it's trying to abstract. Hibernate successfully abstracts many things: variations between dialects of SQL, messy interactions with JDBC, SQL object naming conventions, and so on. But it doesn't even attempt to pretend there's anything other than a relational data model underlying all this. The simple reason is performance. It's just not possible to achieve acceptable performance in data access without acknowledging the nature of the underlying persistent representation. We're old enough to have seen multiple generations of developer relearn this lesson by experience.

Developers often ask about the relationship between Hibernate and JPA, so let's take a short detour into some history.

1.1. Hibernate and JPA

Hibernate was the inspiration behind the *Java* (now *Jakarta*) *Persistence API*, or JPA, and includes a complete implementation of the latest revision of this specification.

> **The early history of Hibernate and JPA**
>
> The Hibernate project began in 2001, when Gavin King's frustration with Entity Beans in EJB 2 boiled over. It quickly overtook other open source and commercial contenders to become the most popular persistence solution for Java, and the book *Hibernate in Action*, written with Christian Bauer, was an influential bestseller.

In 2004, Gavin and Christian joined a tiny startup called JBoss, and other early Hibernate contributors soon followed: Max Rydahl Andersen, Emmanuel Bernard, Steve Ebersole, and Sanne Grinovero.

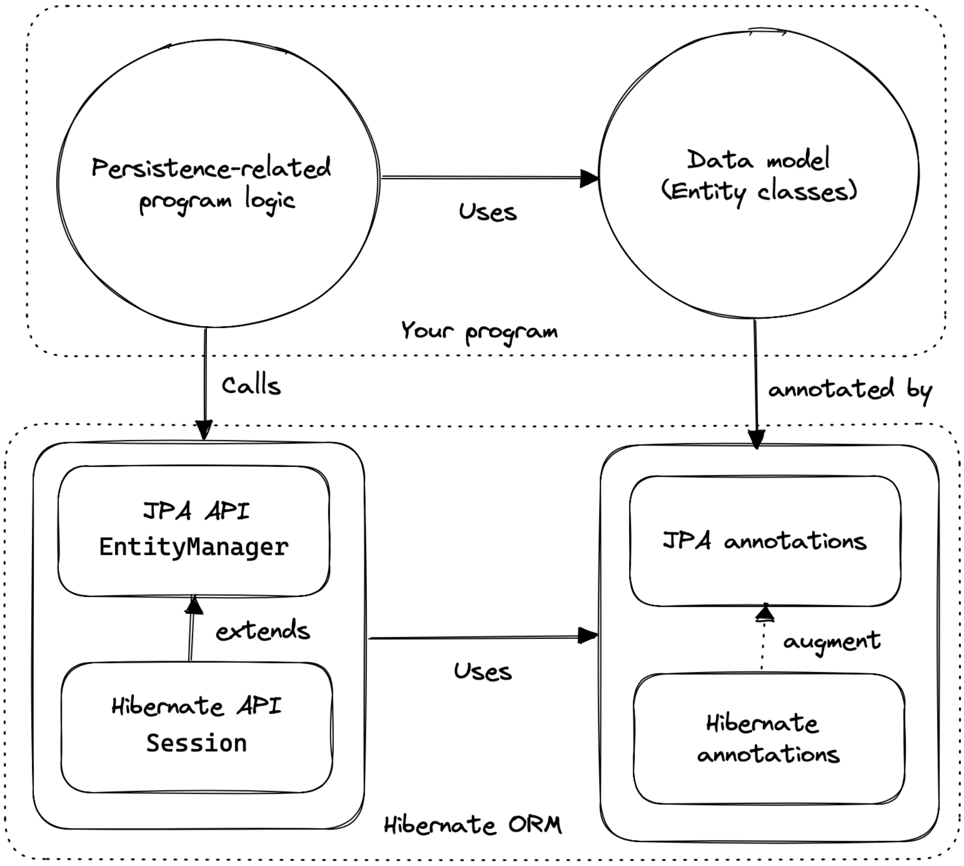
Soon after, Gavin joined the EJB 3 expert group and convinced the group to deprecate Entity Beans in favor of a brand-new persistence API modelled after Hibernate. Later, members of the TopLink team got involved, and the Java Persistence API evolved as a collaboration between—primarily—Sun, JBoss, Oracle, and Sybase, under the leadership of Linda Demichiel.

Over the intervening two decades, *many* talented people have contributed to the development of Hibernate. We're all especially grateful to Steve, who has led the project for many years, since Gavin stepped back to focus in other work.

We can think of the API of Hibernate in terms of three basic elements:

- an implementation of the JPA-defined APIs, most importantly, of the interfaces `EntityManagerFactory` and `EntityManager`, and of the JPA-defined O/R mapping annotations,
- a *native API* exposing the full set of available functionality, centered around the interfaces `SessionFactory`, which extends `EntityManagerFactory`, and `Session`, which extends `EntityManager`, and
- a set of *mapping annotations* which augment the O/R mapping annotations defined by JPA, and which may be used with the JPA-defined interfaces, or with the native API.

Hibernate also offers a range of SPIs for frameworks and libraries which extend or integrate with Hibernate, but we're not interested in any of that stuff here.



As an application developer, you must decide whether to:

- write your program in terms of `Session` and `SessionFactory`, or
- maximize portability to other implementations of JPA by, wherever reasonable, writing code in terms of `EntityManager` and `EntityManagerFactory`, falling back to the native APIs only where necessary.

Whichever path you take, you will use the JPA-defined mapping annotations most of the time, and the Hibernate-defined annotations for more advanced mapping problems.

> You might wonder if it's possible to develop an application using *only* JPA-defined APIs, and, indeed, that's possible in principle. JPA is a great baseline that really nails the basics of the object/relational mapping problem. But without the native APIs, and extended mapping annotations, you miss out on much of the power of Hibernate.

Since Hibernate existed before JPA, and since JPA was modelled on Hibernate, we unfortunately have some competition and duplication in naming between the standard and native APIs. For example:

Table 1.1: Examples of competing APIs with similar naming

Hibernate	JPA
`org.hibernate.annotations.CascadeType`	`javax.persistence.CascadeType`
`org.hibernate.FlushMode`	`javax.persistence.FlushModeType`
`org.hibernate.annotations.FetchMode`	`javax.persistence.FetchType`
`org.hibernate.query.Query`	`javax.persistence.Query`
`org.hibernate.Cache`	`javax.persistence.Cache`
`@org.hibernate.annotations.NamedQuery`	`@javax.persistence.NamedQuery`
`@org.hibernate.annotations.Cache`	`@javax.persistence.Cacheable`
`org.hibernate.relational.SchemaManager`	`jakarta.persistence.SchemaManager`

Typically, the Hibernate-native APIs offer something a little extra that's missing in JPA, so this isn't exactly a *flaw*. But it's something to watch out for.

1.2. Writing Java code with Hibernate

If you're completely new to Hibernate and JPA, you might already be wondering how the persistence-related code is structured.

Well, typically, our persistence-related code comes in two layers:

1. a representation of our data model in Java, which takes the form of a set of annotated entity classes, and
2. a larger number of functions which interact with Hibernate's APIs to perform the persistence operations associated with our various transactions.

The first part, the data or "domain" model, is usually easier to write, but doing a great and very clean job of it will strongly affect your success in the second part.

Most people implement the domain model as a set of what we used to call "Plain Old Java Objects", that is, as simple Java classes with no direct dependencies on technical infrastructure, nor on application logic which deals with request processing, transaction management, communications, or interaction with the database.

> Take your time with this code, and try to produce a Java model that's as close as reasonable to the relational data model. Avoid using exotic or advanced mapping features when they're not really needed. When in the slightest doubt, map a foreign key relationship using `@ManyToOne` with `@OneToMany(mappedBy=…)` in preference to more complicated association mappings.

The second part of the code is much trickier to get right. This code must:

- manage transactions and sessions,
- interact with the database via the Hibernate session,
- publish CDI events and send JMS messages,
- fetch and prepare data needed by the UI, and
- handle failures.

> Responsibility for transaction and session management, and for recovery from certain kinds of failure, is best handled in some sort of framework code.

We're going to come back soon to the thorny question of how this persistence logic should be organized, and how it should fit into the rest of the system.

1.3. Hello, Hibernate

Before we get deeper into the weeds, we'll quickly present a basic example program that will help you get started if you don't already have Hibernate integrated into your project.

We begin with a simple Gradle build file:

`build.gradle`

```
plugins {
    id 'java'
}

group = 'org.example'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    // the GOAT ORM
    implementation 'org.hibernate.orm:hibernate-core:7.3.0.CR1'

    // Hibernate Processor
    annotationProcessor 'org.hibernate.orm:hibernate-processor:7.3.0.CR1'

    // Hibernate Validator
    implementation 'org.hibernate.validator:hibernate-validator:9.0.1.Final'
    implementation 'org.glassfish.expressly:expressly:6.0.0'

    // Agroal connection pool
    runtimeOnly 'org.hibernate.orm:hibernate-agroal:7.3.0.CR1'

    // logging via Log4j
    runtimeOnly 'org.apache.logging.log4j:log4j-core:2.24.3'

    // H2 database
    runtimeOnly 'com.h2database:h2:2.3.232'
}
```

Only the first of these dependencies is absolutely *required* to run Hibernate.

Next, we'll add a logging configuration file for log4j:

`log4j2.properties`

```
rootLogger.level = info
rootLogger.appenderRefs = console
rootLogger.appenderRef.console.ref = console

# SQL statements (set level=debug to enable)
logger.hibernate.name = org.hibernate.SQL
logger.hibernate.level = info
# JDBC parameter binding (set level=trace to enable)
logger.jdbc-bind.name=org.hibernate.orm.jdbc.bind
logger.jdbc-bind.level=info
# JDBC result set extraction (set level=trace to enable)
logger.jdbc-extract.name=org.hibernate.orm.jdbc.extract
logger.jdbc-extract.level=info
# JDBC batching (set level=trace to enable)
logger.batch.name=org.hibernate.orm.jdbc.batch
logger.batch.level=info

# direct log output to the console
appender.console.name = console
appender.console.type = Console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %highlight{[%p]} %m%n
```

Now we need some Java code. We begin with our *entity class*:

`Book.java`

```
package org.hibernate.example;
```

```java
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.validation.constraints.NotNull;

@Entity
class Book {
    @Id
    String isbn;

    @NotNull
    String title;

    Book() {}

    Book(String isbn, String title) {
        this.isbn = isbn;
        this.title = title;
    }
}
```

Finally, let's see code which configures and instantiates Hibernate and asks it to persist and query the entity. Don't worry if this makes no sense at all right now. It's the job of the rest of this Short Guide to make all this crystal clear.

`Main.java`

```java
package org.hibernate.example;

import org.hibernate.jpa.HibernatePersistenceConfiguration;

import static java.lang.System.out;

public class Main {
    public static void main(String[] args) {
        var sessionFactory =
                new HibernatePersistenceConfiguration("Bookshelf")
                        .managedClass(Book.class)
                        // use H2 in-memory database
                        .jdbcUrl("jdbc:h2:mem:db1")
                        .jdbcCredentials("sa", "")
                        // set the Agroal connection pool size
                        .jdbcPoolSize(16)
                        // display SQL in console
                        .showSql(true, true, true)
                        .createEntityManagerFactory();

        // export the inferred database schema
        sessionFactory.getSchemaManager().create(true);

        // persist an entity
        sessionFactory.inTransaction(session -> {
            session.persist(new Book("9781932394153", "Hibernate in Action"));
        });

        // query data using HQL
        sessionFactory.inSession(session -> {
            out.println(session.createSelectionQuery("select isbn||': '||title from Book").getSingleResult());
        });

        // query data using criteria API
        sessionFactory.inSession(session -> {
            var builder = sessionFactory.getCriteriaBuilder();
            var query = builder.createQuery(String.class);
            var book = query.from(Book.class);
            query.select(builder.concat(builder.concat(book.get(Book_.isbn), builder.literal(": ")),
                    book.get(Book_.title)));
            out.println(session.createSelectionQuery(query).getSingleResult());
        });
    }
}
```

In practice, we never access the database directly from a `main()` method. So now let's talk about how to organize persistence logic in a real system. The rest of this chapter is not compulsory. If you're itching for more details about Hibernate itself, you're quite welcome to skip straight to the next chapter, and come back later.

1.4. Entities

A class in the domain model which directly represents a relational database table is called an *entity*. Entity classes are central to object persistence and to object/relational mapping. They're also, typically, central players in the business logic of our application program. Entities represent the *things* in our business domain. This makes them very important objects indeed!

Given how much weight an entity already bears due to its very nature, we need to think carefully before weighing it down with too many additional responsibilities.

What sort of logic belongs in an entity?

There exists an extensive online literature which posits that there are *rich domain models*, where entities have methods implementing interesting business logic, and *anemic domain models*, where the entities are pure data holders, and that a developer should hold an opinion that one or the other of these sorts of domain model is "better".

We do not hold any such opinion, and if you ask us for one, we will most likely suddenly discover somewhere else we need to be.

A more interesting question is not *how much* logic belongs in the entity class, but *what sort* of logic belongs there. We think the answer is that an entity should never implement technical concerns, and should never obtain references to framework objects. Nor should it hold extra mutable state which is not very directly related to its role in representing persistent state. For example:

- an entity may compute totals and averages, even caching them if necessary, enforce its invariants, interact with and construct other entities, and so on, and its annotations express how it maps to database tables,
- but the entity should not call the `EntityManager` or a Jakarta Data repository, build a criteria query, send a JMS message, start a transaction, publish events to the CDI event bus, maintain a stateful queue of events to be published later, or anything of a similar nature.

One way to summarize this is:

> *Entities do business logic; but they don't do orchestration.*

Later, we'll discuss various ways to manage transactions, send event notifications, and query the database. Such code will always be external to the entity itself.

In keeping with our commitment to anti-dogmatism, we would like to add the following important caveat to the discussion in the previous callout.

Active Record

The discussion above expresses our "traditional" approach—which lay behind the design of Hibernate, of JPA, and of Jakarta Data—where entity classes are plain Java objects without dependence on framework code. An alternative approach is the Active Record pattern, as exemplified by Panache. In Active Record, entity types inherit framework objects, and persistence operations are located directly on the entities. You can think of this as merging the roles of entity and DAO/Repository into a single object.

Active Record comes with both upsides and downsides, but we don't mean to exclude it from consideration. We must therefore slightly modify the prescription we've given above: in an Active Record, it's obviously OK to access the `EntityManager` and perform other persistence-related operations, and we therefore expect our Active Record class to look somewhat more "technical" than a trad entity.

But the basic principle remains intact: an entity does not do orchestration, it does not manage transactions, it does not obtain references to *other* sorts of framework object, and it does not hold mutable state unrelated to its persistent state.

For now, we're going to assume that entities are implemented as plain Java classes.

1.5. Stateful and stateless sessions

It should be very clear from the example code above, that the session is also a very important object. It exposes basic operations like `persist()` and `createQuery()`, and so it's our first port of call when we want to interact with the database via Hibernate. In the code we just saw, we’ve used a *stateful session*.

Later, we’ll learn about the idea of a *persistence context*. Oversimplifying for now, you can think of it as a cache of data which has been read in the current transaction. Thus, in the architecture of Hibernate, it’s sometimes called the *first-level cache*. Each stateful session — that is, every Hibernate `Session`, and every JPA `EntityManager` — has its own persistence context.

But stateful sessions have never been the only possibility. The `StatelessSession` interface offers a way to interact with Hibernate *without* going through a persistence context. However, the programming model is somewhat different.

Stateless sessions

Among our biggest regrets is that we didn’t give enough love to `StatelessSession` twenty years ago. Sure, a stateful session is in some sense more powerful, or at least more magical. But with that magic comes a loss of direct control over persistence operations, and some traps for inexperienced users. A significant minority of developers find working with a persistence context frustrating, and they would surely be better served by a stateless session.

- We used to view `StatelessSession` as an API directed toward very specific usage patterns, in particular, batch processing of large numbers of entities. As a result, we left out certain functionality — for example, use of the second-level cache — which didn’t seem relevant to those use cases. This left `StatelessSession` lacking feature parity with `Session`, and it was a mistake. In Hibernate 7, we’ve fixed this mistake. A `StatelessSession` now offers essentially all the functionality of Hibernate except, naturally, the first-level cache.
- Compounding our error, we left `StatelessSession` out of JPA. This meant that a large number of Hibernate users *didn’t even realize this option existed.* We promise to make sure there are stateless sessions in Jakarta Persistence 4.

So, finally, let us state for the record: we messed up here. Hibernate is all about *object/relational mapping;* persistence contexts are something extra on top. You don’t have to use stateful sessions, and you’re not doing anything wrong if you decide to use stateless sessions instead.

As of Hibernate 7, a key decision for any new project is which of these programming models to take as a baseline. Fortunately, the two models aren’t mutually exclusive. This is a friendly competition, where the two APIs are designed to complement each other. Even if we decide to use stateful `Session`s most of the time, we can still use a `StatelessSession` wherever it’s more convenient.

> On the other hand, if you decide to adopt Jakarta Data, the decision is made for you: repositories in Jakarta Data 1.0 are always stateless, and in Hibernate Data Repositories a repository is backed by a `StatelessSession`.

But now we’ve got just a little bit ahead of ourselves. In the next section taking we’re taking a journey which *might* — but definitely doesn’t *necessarily* — end at the idea of a "repository".

1.6. Organizing persistence logic

In a real program, persistence logic like the code shown above is usually interleaved with other sorts of code, including logic:

- implementing the rules of the business domain, or
- for interacting with the user.

Therefore, many developers quickly—even *too quickly*, in our opinion—reach for ways to isolate the persistence logic into some sort of separate architectural layer. We’re going to ask you to suppress this urge for now.

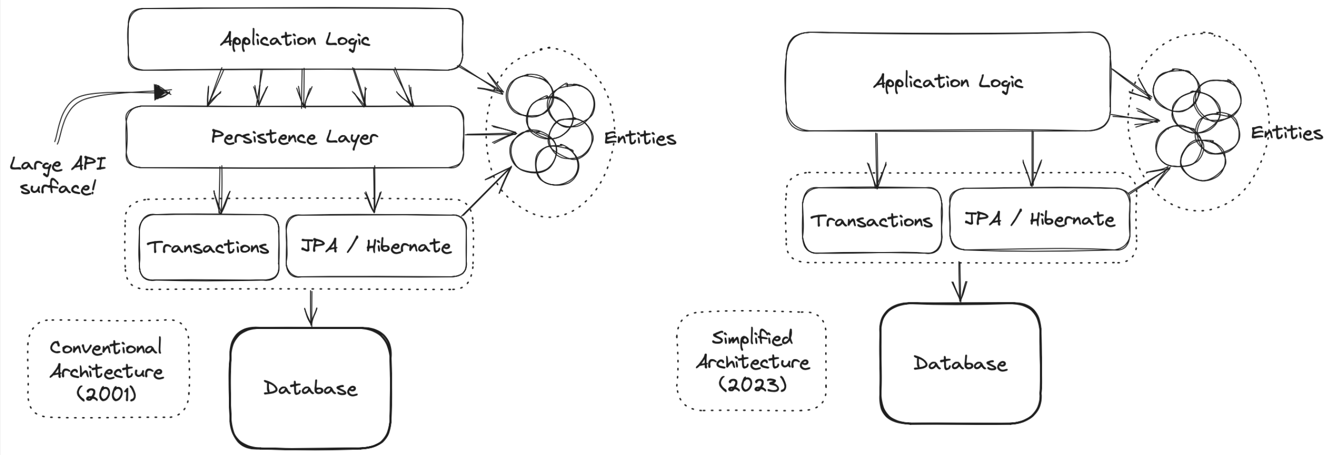
We prefer a *bottom-up* approach to organizing our code. We like to start thinking about methods and functions, not about architectural layers and container-managed objects.

Rethinking the persistence layer

When we wrote *An Introduction to Hibernate 6*, the predecessor of this document, we broke with a long practice of remaining agnostic in debates over application architecture. Into the vacuum created by our agnosticism had poured a deluge of advice which tended to encourage over-engineering and violation of the First Commandment of software engineering: *Don’t Repeat Yourself.* We felt compelled to speak up for a more elementary approach.

Here, we reiterate our preference for design which emerges organically from the code itself, via a process of refactoring and iterative abstraction. The Extract Method refactoring is a far, far more powerful tool than drawing boxes and arrows on whiteboards.

In particular, we hereby give you permission to write code which mixes business logic with persistence logic within the same architectural layer. Every architectural layer comes with a high cost in boilerplate, and in many contexts a separate persistence layer is simply unnecessary. Both of the following architectures represent allowed points within the design space:



In the case that a separate persistence layer *is* helpful, we encourage you to consider the use of Jakarta Data repositories, in preference to older approaches.

To illustrate the sort of approach to code organization that we advocate, let's consider a service which queries the database using HQL or SQL. We might start with something like this, a mix of UI and persistence logic:

```
@Path("/")
@Produces("application/json")
public class BookResource {

    private final SessionFactory sessionfactory = .... ;

    @GET
    @Path("book/{isbn}")
    public Book getBook(String isbn) {
        var book = sessionFactory.fromTransaction(session -> session.find(Book.class, isbn));
        return book == null ? Response.status(404).build() : book;
    }

}
```

Indeed, we might also *finish* with something like that—it's quite hard to identify anything concretely wrong with the code above, and for such a simple case it seems really difficult to justify making this code more complicated by introducing additional objects.

One very nice aspect of this code, which we wish to draw your attention to, is that session and transaction management is handled by generic "framework" code, just as we already recommended above. In this case, we're using the `fromTransaction()` method, which happens to come built in to Hibernate. But you might prefer to use something else, for example:

- in a container environment like Jakarta EE or Quarkus, *container-managed transactions* and *container-managed persistence contexts*, or
- something you write yourself.

The important thing is that calls like `createEntityManager()` and `getTransaction().begin()` don't belong in regular program logic, because it's tricky and tedious to get the error handling correct.

Let's now consider a slightly more complicated case.

```
@Path("/")
@Produces("application/json")
public class BookResource {
    private static final int RESULTS_PER_PAGE = 20;

    private final SessionFactory sessionfactory = .... ;

    @GET
    @Path("books/{titlePattern}/{pageNumber:\\d+}")
    public List<Book> findBooks(String titlePattern, int pageNumber) {
        var page = Page.page(RESULTS_PER_PAGE, pageNumber);
        var books =
                sessionFactory.fromTransaction(session -> {
                    var findBooksByTitle = "from Book where title like ?1 order by title";
                    return session.createSelectionQuery(findBooksByTitle, Book.class)
                            .setParameter(1, titlePattern)
```

```
                            .setPage(page)
                            .getResultList();
                });
        return books.isEmpty() ? Response.status(404).build() : books;
    }

}
```

This is fine, and we won't complain if you prefer to leave the code exactly as it appears above. But there's one thing we could perhaps improve. We love super-short methods with single responsibilities, and there looks to be an opportunity to introduce one here. Let's hit the code with our favorite thing, the Extract Method refactoring. We obtain:

```
static List<Book> findBooksTitled(Session session, String titlePattern, Page page) {
    var findBooksByTitle = "from Book where title like ?1 order by title";
    return session.createSelectionQuery(findBooksByTitle, Book.class)
            .setParameter(1, titlePattern)
            .setPage(page)
            .getResultList();
}
```

This is an example of a *query method*, a function which accepts arguments to the parameters of a HQL or SQL query, and executes the query, returning its results to the caller. And that's all it does; it doesn't orchestrate additional program logic, and it doesn't perform transaction or session management.

It's even better to specify the query string using the `@NamedQuery` annotation, so that Hibernate can validate the query at startup time, that is, when the `SessionFactory` is created, instead of when the query is first executed. Indeed, since we included Hibernate Processor in our Gradle build, the query can even be validated at *compile time*.

We need a place to put the annotation, so let's move our query method to a new class:

```
@CheckHQL // validate named queries at compile time
@NamedQuery(name = "findBooksByTitle",
            query = "from Book where title like :title order by title")
class Queries {

    static List<Book> findBooksTitled(Session session, String titlePattern, Page page) {
        return session.createQuery(Queries_._findBooksByTitle_)  //type safe reference to the named query
                .setParameter("title", titlePattern)
                .setPage(page)
                .getResultList();
    }

}
```

Notice that our query method doesn't attempt to hide the `EntityManager` from its clients. Indeed, the client code is responsible for providing the `EntityManager` or `Session` to the query method.

The client code may:

- obtain an `EntityManager` or `Session` by calling `inTransaction()` or `fromTransaction()`, as we saw above, or,
- in an environment with container-managed transactions, it might obtain it via dependency injection.

Whatever the case, the code which orchestrates a unit of work usually just calls the `Session` or `EntityManager` directly, passing it along to helper methods like our query method if necessary.

```
@GET
@Path("books/{titlePattern}/{pageNumber:\\d+}")
public List<Book> findBooks(String titlePattern, int pageNumber) {
    var page = Page.page(RESULTS_PER_PAGE, pageNumber);
    var books =
            sessionFactory.fromTransaction(session ->
                    // call handwritten query method
                    Queries.findBooksTitled(session, titlePattern, page));
    return books.isEmpty() ? Response.status(404).build() : books;
}
```

You might be thinking that our query method looks a bit boilerplatey. That's true, perhaps, but we're much more concerned that it's still not perfectly typesafe. Indeed, for many years, the lack of compile-time checking for HQL queries and code which binds arguments to query parameters was our number one source of discomfort with Hibernate. Here, the `@CheckHQL` annotation takes care of checking the query itself, but the call to `setParameter()` is still not type safe.

Fortunately, there's now a great solution to both problems. Hibernate Processor is able to fill in the implementation of such query methods for us. This facility is the topic of a whole chapter of this introduction, so for now we'll just leave you with one simple example.

Suppose we simplify `Queries` to just the following:

```
// a sort of proto-repository, this interface is never implemented
interface Queries {
    // a HQL query method with a generated static "implementation"
    @HQL("where title like :title order by title")
    List<Book> findBooksTitled(String title, Page page);
}
```

Then Hibernate Processor automatically produces an implementation of the method annotated `@HQL` in a class named `Queries_`. We can call it just like we were previously calling our handwritten version:

```
@GET
@Path("books/{titlePattern}/{pageNumber:\\d+}")
public List<Book> findBooks(String titlePattern, int pageNumber) {
    var page = Page.page(RESULTS_PER_PAGE, pageNumber);
    var books =
            sessionFactory.fromTransaction(session ->
                    // call the generated query method "implementation"
                    Queries_.findBooksTitled(session, titlePattern, page));
    return books.isEmpty() ? Response.status(404).build() : books;
}
```

In this case, the quantity of code eliminated is pretty trivial. The real value is in improved type safety. We now find out about errors in assignments of arguments to query parameters at compile time.

This is all quite nice so far, but at this point you're probably wondering whether we could use dependency injection to obtain an *instance* of the `Queries` interface, and have this object take care of obtaining its own `Session`. Well, indeed we can. What we need to do is indicate the kind of session the `Queries` interface depends on, by adding a method to retrieve the session. Observe, again, that we're *still* not attempting to hide the `Session` from the client code.

```
// a true repository interface with generated implementation
interface Queries {
    // declare the kind of session backing this repository
    Session session();

    // a HQL query method with a generated implementation
    @HQL("where title like :title order by title")
    List<Book> findBooksTitled(String title, Page page);
}
```

The `Queries` interface is now considered a *repository*, and we may use CDI to inject the repository implementation generated by Hibernate Processor. Also, since I guess we're now working in some sort of container environment, we'll let the container manage transactions for us.

```
@Inject Queries queries;  // inject the repository

@GET
@Path("books/{titlePattern}/{pageNumber:\\d+}")
@Transactional
public List<Book> findBooks(String titlePattern, int pageNumber) {
    var page = Page.page(RESULTS_PER_PAGE, pageNumber);
    var books = queries.findBooksTitled(session, titlePattern, page); // call the repository method
    return books.isEmpty() ? Response.status(404).build() : books;
}
```

Alternatively, if CDI isn't available, we may directly instantiate the generated repository implementation class using `new Queries_(entityManager)`.

> The Jakarta Data specification now formalizes this approach using standard annotations, and our implementation of this specification, Hibernate Data Repositories, is built into Hibernate Processor. You probably already have it available in your program.

> Unlike other repository frameworks, Hibernate Data Repositories offers something that plain JPA simply doesn't have: full compile-time type safety for your queries. To learn more, please refer to Introducing Hibernate Data Repositories.

Why we changed our mind about repositories

At the time we wrote *An Introduction to Hibernate 6*, we were especially frustrated with the limitations of popular frameworks which claimed to simplify the use of JPA by wrapping and hiding the `EntityManager`. In our considered opinion, such frameworks typically made JPA harder to use, sometimes misleading users into misuse of the technology.

The birth of the Jakarta Data specification has obsoleted our arguments against repositories, along with the older frameworks which were the source of our frustration. Jakarta Data—as realized by Hibernate Data Repositories—offers a clean but very flexible way to organize code, along with much better compile-time type safety, without getting in the way of direct use of the `StatelessSession`.

Now that we have a rough picture of what our persistence logic might look like, it's natural to ask how we should test our code.

1.7. Testing persistence logic

When we write tests for our persistence logic, we're going to need:

1. a database, with
2. an instance of the schema mapped by our persistent entities, and
3. a set of test data, in a well-defined state at the beginning of each test.

It might seem obvious that we should test against the same database system that we're going to use in production, and, indeed, we should certainly have at least *some* tests for this configuration. But on the other hand, tests which perform I/O are much slower than tests which don't, and most databases can't be set up to run in-process.

So, since most persistence logic written using Hibernate 6 is *extremely* portable between databases, it often makes good sense to test against an in-memory Java database. (H2 is the one we recommend.)

> We do need to be careful here if our persistence code uses native SQL, or if it uses concurrency-management features like pessimistic locks.

Whether we're testing against our real database, or against an in-memory Java database, we'll need to export the schema at the beginning of a test suite. We *usually* do this when we create the Hibernate `SessionFactory` or JPA `EntityManagerFactory`, and so traditionally we've used a configuration property for this.

The JPA-standard property is `jakarta.persistence.schema-generation.database.action`. For example, if we're using `PersistenceConfiguration` to configure Hibernate, we could write:

```
configuration.property(PersistenceConfiguration.SCHEMAGEN_DATABASE_ACTION,
                       Action.SPEC_ACTION_DROP_AND_CREATE);
```

Alternatively, we may use the new `SchemaManager` API to export the schema, just as we did above. This option is especially convenient when writing tests.

```
sessionFactory.getSchemaManager().create(true);
```

Since executing DDL statements is very slow on many databases, we don't want to do this before every test. Instead, to ensure that each test begins with the test data in a well-defined state, we need to do two things before each test:

1. clean up any mess left behind by the previous test, and then
2. reinitialize the test data.

We may truncate all the tables, leaving an empty database schema, using the `SchemaManager`.

```
sessionFactory.getSchemaManager().truncate();
```

After truncating tables, we might need to initialize our test data. We may specify test data in a SQL script, for example:

/import.sql

```
insert into Books (isbn, title) values ('9781932394153', 'Hibernate in Action')
insert into Books (isbn, title) values ('9781932394887', 'Java Persistence with Hibernate')
```

```
insert into Books (isbn, title) values ('9781617290459', 'Java Persistence with Hibernate, Second Edition')
```

If we name this file `import.sql`, and place it in the root classpath, that's all we need to do.

Otherwise, we need to specify the file in the configuration property `jakarta.persistence.sql-load-script-source`. If we're using `PersistenceConfiguration` to configure Hibernate, we could write:

```
configuration.property(AvailableSettings.JAKARTA_HBM2DDL_LOAD_SCRIPT_SOURCE,
                       "/org/example/test-data.sql");
```

The SQL script will be executed every time `export()` or `truncate()` is called.

> There's another sort of mess a test can leave behind: cached data in the second-level cache. We recommend *disabling* Hibernate's second-level cache for most sorts of testing. Alternatively, if the second-level cache is not disabled, then before each test we should call:
>
> ```
> sessionFactory.getCache().evictAllRegions();
> ```

Now, suppose you've followed our advice, and written your entities and query methods to minimize dependencies on "infrastructure", that is, on libraries other than JPA and Hibernate, on frameworks, on container-managed objects, and even on bits of your own system which are hard to instantiate from scratch. Then testing persistence logic is now straightforward!

You'll need to:

- bootstrap Hibernate and create a `SessionFactory` or `EntityManagerFactory` at the beginning of your test suite (we've already seen how to do that), and
- create a new `Session` or `EntityManager` inside each `@Test` method, using `inTransaction()`, for example.

Actually, some tests might require multiple sessions. But be careful not to leak a session between different tests.

> Another important test we'll need is one which validates our O/R mapping annotations against the actual database schema. This is again the job of the schema management tooling, either:
>
> ```
> configuration.property(PersistenceConfiguration.SCHEMAGEN_DATABASE_ACTION,
>                        Action.ACTION_VALIDATE);
> ```
>
> Or:
>
> ```
> sessionFactory.getSchemaManager().validate();
> ```
>
> This "test" is one which many people like to run even in production, when the system starts up.

1.8. Overview

It's now time to begin our journey toward actually *understanding* the code we saw earlier.

This introduction will guide you through the basic tasks involved in developing a program that uses Hibernate for persistence:

1. configuring and bootstrapping Hibernate, and obtaining an instance of `SessionFactory` or `EntityManagerFactory`,
2. writing a *domain model*, that is, a set of *entity classes* which represent the persistent types in your program, and which map to tables of your database,
3. customizing these mappings when the model maps to a pre-existing relational schema,
4. using the `Session` or `EntityManager` to perform operations which query the database and return entity instances, or which update the data held in the database,
5. using Hibernate Processor to improve compile-time type-safety,
6. writing complex queries using the Hibernate Query Language (HQL) or native SQL, and, finally
7. tuning performance of the data access logic.

Naturally, we'll start at the top of this list, with the least-interesting topic: *configuration*.

Chapter 2. Configuration and bootstrap

We would love to make this section short. Unfortunately, there are several distinct ways to configure and bootstrap Hibernate, and we're going to have to describe at least two of them in detail.

The five basic ways to obtain an instance of Hibernate are shown in the following table:

Using the standard JPA-defined XML, and the operation `Persistence.createEntityManagerFactory()`	Usually chosen when portability between JPA implementations is important.
Using the standard JPA-defined `PersistenceConfiguration` class	Usually chosen when portability between JPA implementations is important, but programmatic control is desired.
Using `HibernatePersistenceConfiguration` or the older `Configuration` class to construct a `SessionFactory`	When portability between JPA implementations is not important, this option adds some convenience and saves a typecast.
Using the more complex APIs defined in `org.hibernate.boot`	Used primarily by framework integrators, this option is outside the scope of this document.
By letting the container take care of the bootstrap process and of injecting the `SessionFactory` or `EntityManagerFactory`	Used in a container environment like WildFly or Quarkus.

Here we'll focus on the first two options.

Hibernate in containers

Actually, the last option is extremely popular, since every major Java application server and microservice framework comes with built-in support for Hibernate. Such container environments typically also feature facilities to automatically manage the lifecycle of an `EntityManager` or `Session` and its association with container-managed transactions.

To learn how to configure Hibernate in such a container environment, you'll need to refer to the documentation of your chosen container. For Quarkus, here's the relevant documentation.

If you're using Hibernate outside of a container environment, you'll need to:

- include Hibernate ORM itself, along with the appropriate JDBC driver, as dependencies of your project, and
- configure Hibernate with information about your database, by specifying configuration properties.

2.1. Including Hibernate in your project build

First, add the following dependency to your project:

```
org.hibernate.orm:hibernate-core:{version}
```

Where `{version}` is the version of Hibernate you're using, `7.3.0.CR1`, for example.

You'll also need to add a dependency for the JDBC driver for your database.

Table 2.2: JDBC driver dependencies

Database	Driver dependency
PostgreSQL or CockroachDB	`org.postgresql:postgresql:{version}`
MySQL or TiDB	`com.mysql:mysql-connector-j:{version}`
MariaDB	`org.mariadb.jdbc:mariadb-java-client:{version}`
DB2	`com.ibm.db2:jcc:{version}`
SQL Server	`com.microsoft.sqlserver:mssql-jdbc:{version}`
Oracle	`com.oracle.database.jdbc:ojdbc17:{version}`

Database	Driver dependency
H2	`com.h2database:h2:{version}`
HSQLDB	`org.hsqldb:hsqldb:{version}`
MongoDB	The JDBC driver is bundled with the dialect mentioned in Optional dependencies
Google Spanner	`com.google.cloud:google-cloud-spanner-jdbc:{version}`

Where `{version}` is the latest version of the JDBC driver for your database.

2.2. Optional dependencies

Optionally, you might also add any of the following additional features:

Table 2.3: Optional dependencies

Optional feature	Dependencies
An SLF4J logging implementation	`org.apache.logging.log4j:log4j-core` or `org.slf4j:slf4j-jdk14`
A JDBC connection pool, for example, Agroal	`org.hibernate.orm:hibernate-agroal`
The Hibernate Processor, especially if you're using Jakarta Data or the JPA criteria query API	`org.hibernate.orm:hibernate-processor`
The Query Validator, for compile-time checking of HQL	`org.hibernate:query-validator`
Hibernate Validator, an implementation of Bean Validation	`org.hibernate.validator:hibernate-validator` and `org.glassfish.expressly:expressly`
Local second-level cache support via JCache and EHCache	`org.hibernate.orm:hibernate-jcache` and `org.ehcache:ehcache`
Local second-level cache support via JCache and Caffeine	`org.hibernate.orm:hibernate-jcache` and `com.github.ben-manes.caffeine:jcache`
Distributed second-level cache support via Infinispan	`org.infinispan:infinispan-hibernate-cache-v60`
A JSON serialization library for working with JSON datatypes, for example, Jackson 2, {jackson3}[Jackson 3] or Yasson	`com.fasterxml.jackson.core:jackson-databind`, `tools.jackson.core:jackson-databind` or `org.eclipse:yasson`
Hibernate Spatial	`org.hibernate.orm:hibernate-spatial`
Envers, for auditing historical data	`org.hibernate.orm:hibernate-envers`
Hibernate JFR, for monitoring via Java Flight Recorder	`org.hibernate.orm:hibernate-jfr`
Hibernate Jandex integration, for entity discovery	`org.hibernate.orm:hibernate-scan-jandex`
Community dialects	`org.hibernate.orm:hibernate-community-dialects`
Third-party dialects	MongoDB: `org.mongodb:mongodb-hibernate:{version}` Google Spanner: `com.google.cloud:google-cloud-spanner-hibernate-dialect:{version}` Where `{version}` is the version of the third-party dialect compatible with the version of Hibernate ORM you are using. See the dialect's own documentation for more information. The compatibility matrix on the Hibernate website may also be of help.

You might also add the Hibernate bytecode enhancer to your Gradle build if you want to use field-level lazy fetching.

2.3. Configuration using JPA XML

Sticking to the JPA-standard approach, we would provide a file named `persistence.xml`, which we usually place in the `META-INF` directory of a *persistence archive*, that is, of the `.jar` file or directory which contains our entity classes.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="2.0">

    <persistence-unit name="org.hibernate.example">

        <class>org.hibernate.example.Book</class>
        <class>org.hibernate.example.Author</class>

        <properties>
            <!-- PostgreSQL -->
            <property name="jakarta.persistence.jdbc.url"
                      value="jdbc:postgresql://localhost/example"/>

            <!-- Credentials -->
            <property name="jakarta.persistence.jdbc.user"
                      value="gavin"/>
            <property name="jakarta.persistence.jdbc.password"
                      value="hibernate"/>

            <!-- Automatic schema export -->
            <property name="jakarta.persistence.schema-generation.database.action"
                      value="drop-and-create"/>

            <!-- SQL statement logging -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>

        </properties>

    </persistence-unit>

</persistence>
```

The `<persistence-unit>` element defines a named *persistence unit*, that is:

- a collection of associated entity types, along with
- a set of default configuration settings, which may be augmented or overridden at runtime.

Each `<class>` element specifies the fully-qualified name of an entity class.

Scanning for entity classes

In some container environments, for example, in any EE container, the `<class>` elements are unnecessary, since the container will scan the archive for annotated classes, and automatically recognize any class annotated `@Entity`.

Each `<property>` element specifies a *configuration property* and its value. Note that:

- the configuration properties in the `jakarta.persistence` namespace are standard properties defined by the JPA spec, and
- properties in the `hibernate` namespace are specific to Hibernate.

We may obtain an `EntityManagerFactory` by calling `Persistence.createEntityManagerFactory()`:

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("org.hibernate.example");
```

If necessary, we may override configuration properties specified in `persistence.xml`:

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("org.hibernate.example",
            Map.of(AvailableSettings.JAKARTA_JDBC_PASSWORD, password));
```

2.4. Programmatic configuration using JPA API

The new `PersistenceConfiguration` class allows full programmatic control over creation of the `EntityManagerFactory`.

```
EntityManagerFactory entityManagerFactory =
        new PersistenceConfiguration("Bookshop")
            .managedClass(Book.class)
            .managedClass(Author.class)
            // PostgreSQL
            .property(PersistenceConfiguration.JDBC_URL, "jdbc:postgresql://localhost/example")
            // Credentials
            .property(PersistenceConfiguration.JDBC_USER, user)
            .property(PersistenceConfiguration.JDBC_PASSWORD, password)
            // Automatic schema export
            .property(PersistenceConfiguration.SCHEMAGEN_DATABASE_ACTION,
                    Action.SPEC_ACTION_DROP_AND_CREATE)
            // SQL statement logging
            .property(JdbcSettings.SHOW_SQL, true)
            .property(JdbcSettings.FORMAT_SQL, true)
            .property(JdbcSettings.HIGHLIGHT_SQL, true)
            // Create a new EntityManagerFactory
            .createEntityManagerFactory();
```

The specification gives JPA implementors like Hibernate explicit permission to extend this class, and so Hibernate offers the `HibernatePersistenceConfiguration`, which lets us obtain a `SessionFactory` without any need for a cast.

```
SessionFactory sessionFactory =
        new HibernatePersistenceConfiguration("Bookshop")
            .managedClass(Book.class)
            .managedClass(Author.class)
            // PostgreSQL
            .jdbcUrl("jdbc:postgresql://localhost/example")
            // Credentials
            .jdbcCredentials(user, password)
            // Automatic schema export
            .schemaToolingAction(Action.SPEC_ACTION_DROP_AND_CREATE)
            // SQL statement logging
            .showSql(true, true, true)
            // Create a new SessionFactory
            .createEntityManagerFactory();
```

Alternatively, the venerable class `Configuration` offers similar functionality.

Advanced configuration options

Actually, these APIs are very simple facades resting on the much more powerful—but also more complex—APIs defined in the package `org.hibernate.boot`. This API is useful if you have very advanced requirements, for example, if you're writing a framework or implementing a container. You'll find more information in the User Guide, and in the package-level documentation of `org.hibernate.boot`.

2.5. Entity discovery

In a Jakarta EE container environment, we don't usually need to list entity and embeddable classes explicitly in `persistence.xml`. Instead, the container scans the persistence unit `jar` file and automatically discovers classes annotated `@Entity`, `@Embeddable`, or `@MappedSuperclass`.

`HibernatePersistenceConfiguration` offers the same functionality if the optional dependency `hibernate-scan-jandex` is available at runtime.

In the following code, entity classes available on the class loader which loaded `Main.class` are automatically discovered.

```
SessionFactory sessionFactory =
        // entities discovered on ClassLoader of Main.class
        new HibernatePersistenceConfiguration("Bookshop", Main.class)
            // PostgreSQL
            .jdbcUrl("jdbc:postgresql://localhost/example")
            // Credentials
            .jdbcCredentials(user, password)
            // Automatic schema export
            .schemaToolingAction(Action.SPEC_ACTION_DROP_AND_CREATE)
            // SQL statement logging
            .showSql(true, true, true)
            // Create a new SessionFactory
            .createEntityManagerFactory();
```

Notice that we were able to remove the calls to `managedClass()`.

2.6. Configuration using Hibernate properties file

If we're using programmatic configuration, but we don't want to put certain configuration properties directly in the Java code, we can specify them in a file named `hibernate.properties`, and place the file in the root classpath.

```
# PostgreSQL
jakarta.persistence.jdbc.url=jdbc:postgresql://localhost/example
# Credentials
jakarta.persistence.jdbc.user=hibernate
jakarta.persistence.jdbc.password=zAh7mY$2MNshzAQ5

# SQL statement logging
hibernate.show_sql=true
hibernate.format_sql=true
hibernate.highlight_sql=true
```

2.7. Basic configuration settings

The `PersistenceConfiguration` class declares `static final` constants holding the names of all configuration properties defined by the specification itself, for example, `JDBC_URL` holds the property name `"jakarta.persistence.jdbc.driver"`.

Similarly, the class `AvailableSettings` enumerates all the configuration properties understood by Hibernate.

Of course, we're not going to cover every useful configuration setting in this chapter. Instead, we'll mention the ones you need to get started, and come back to some other important settings later, especially when we talk about performance tuning.

> Hibernate has many—too many—switches and toggles. Please don't go crazy messing about with these settings; most of them are rarely needed, and many only exist to provide backward compatibility with older versions of Hibernate. With rare exception, the default behavior of every one of these settings was carefully chosen to be *the behavior we recommend*.

The properties you really do need to get started are these three:

Table 2.4: JDBC connection settings

Configuration property name	Purpose
`jakarta.persistence.jdbc.url`	JDBC URL of your database
`jakarta.persistence.jdbc.user` and `jakarta.persistence.jdbc.password`	Your database credentials

> Since Hibernate 6, you don't need to specify `hibernate.dialect`. The correct Hibernate SQL `Dialect` will be determined for you automatically. The only reason to specify this property is if you're using a custom user-written or third-party `Dialect` class.
>
> Similarly, neither `hibernate.connection.driver_class` nor `jakarta.persistence.jdbc.driver` is needed when working with one of the supported databases.

In some environments it's useful to be able to start Hibernate without accessing the database. In this case, we must explicitly specify not only the database platform, but also the version of the database, using the standard JPA configuration properties.

```
# disable use of JDBC database metadata
hibernate.boot.allow_jdbc_metadata_access=false

# explicitly specify database and version
jakarta.persistence.database-product-name=PostgreSQL
jakarta.persistence.database-major-version=15
jakarta.persistence.database-minor-version=7
```

The product name is the value returned by `java.sql.DatabaseMetaData.getDatabaseProductName()`, for example, `PostgreSQL`, `MySQL`, `H2`, `Oracle`, `EnterpriseDB`, `MariaDB`, or `Microsoft SQL Server`.

Table 2.5: Settings needed when database is inaccessible at startup

Configuration property name	Purpose
`hibernate.boot.allow_jdbc_metadata_access`	Set to `false` to disallow access to the database at startup
`jakarta.persistence.database-product-name`	The database product name, according to the JDBC driver
`jakarta.persistence.database-major-version` and `jakarta.persistence.database-minor-version`	The major and minor versions of the database

Pooling JDBC connections is an extremely important performance optimization. You can set the size of Hibernate's built-in connection pool using this property:

Table 2.6: Built-in connection pool size

Configuration property name	Purpose
`hibernate.connection.pool_size`	The size of the connection pool

This configuration property is also respected when you use Agroal, HikariCP, or c3p0 for connection pooling.

> By default, Hibernate uses a simplistic built-in connection pool. This pool is not meant for use in production, and later, when we discuss performance, we'll see how to select a more robust implementation.

Alternatively, in a container environment, you'll need at least one of these properties:

Table 2.7: Transaction management settings

Configuration property name	Purpose
`jakarta.persistence.transactionType`	(Optional, defaults to `JTA`) Determines if transaction management is via JTA or resource-local transactions. Specify `RESOURCE_LOCAL` if JTA should not be used.
`jakarta.persistence.jtaDataSource`	JNDI name of a JTA datasource
`jakarta.persistence.nonJtaDataSource`	JNDI name of a non-JTA datasource

In this case, Hibernate obtains pooled JDBC database connections from a container-managed `DataSource`.

2.8. Automatic schema export

You can have Hibernate infer your database schema from the mapping annotations you've specified in your Java code, and export the schema at initialization time by specifying one or more of the following configuration properties:

Table 2.8: Schema management settings

Configuration property name	Purpose
`jakarta.persistence.schema-generation.database.action`	• If `drop-and-create`, first drop the schema, then export tables, sequences, and constraints, and then populate initial data • If `create`, export tables, sequences, and constraints, without attempting to drop them first, and then populate initial data • If `create-drop`, drop the schema and recreate it on `SessionFactory` startup; additionally, drop the schema on `SessionFactory` shutdown • If `drop`, drop the schema on `SessionFactory` shutdown • If `validate`, validate the database schema without changing it • If `update`, only export what's missing in the schema, and alter incorrect column types • If `populate`, only populate initial data
`jakarta.persistence.create-database-schemas`	(Optional) If `true`, automatically create schemas and catalogs
`jakarta.persistence.schema-generation.create-source`	(Optional) If `metadata-then-script` or `script-then-metadata`, execute an additional SQL script when exported tables and sequences
`jakarta.persistence.schema-generation.create-script-source`	(Optional) The name of a SQL DDL script to be executed
`jakarta.persistence.sql-load-script-source`	(Optional) The name of a SQL DML script to be executed
`hibernate.hbm2ddl.import_files_sql_extractor`	(Optional) If `multi-line`, SQL statements may be split across multiple lines in scripts, and must be `;`-terminated.

This feature is extremely useful for testing.

> The easiest way to pre-initialize a database with test or "reference" data is to place a list of SQL `insert` statements in a file named, for example, `import.sql`, and specify the path to this file using the property `jakarta.persistence.sql-load-script-source`. We've already seen an example of this approach, which is cleaner than writing Java code to instantiate entity instances and calling `persist()` on each of them.

As we mentioned earlier, it can also be useful to control schema export programmatically.

> The `SchemaManager` API allows programmatic control over schema export:
>
> ```
> sessionFactory.getSchemaManager().create(true);
> ```

2.9. Logging the generated SQL

To see the generated SQL as it's sent to the database, you have two options.

One way is to set the property `hibernate.show_sql` to `true`, and Hibernate will log SQL directly to the console. You can make the output much more readable by enabling formatting or highlighting. These settings really help when troubleshooting the generated SQL statements.

Table 2.9: Settings for SQL logging to the console

Configuration property name	Purpose
`hibernate.show_sql`	If `true`, log SQL directly to the console
`hibernate.format_sql`	If `true`, log SQL in a multiline, indented format
`hibernate.highlight_sql`	If `true`, log SQL with syntax highlighting via ANSI escape codes

Alternatively, you can enable `DEBUG`-level logging for the category `org.hibernate.SQL` using your preferred SLF4J logging implementation.

For example, if you're using Log4J 2 (as above in Optional dependencies), add these lines to your `log4j2.properties` file:

```
# SQL execution
```

```
logger.hibernate.name = org.hibernate.SQL
logger.hibernate.level = debug

# JDBC parameter binding
logger.jdbc-bind.name=org.hibernate.orm.jdbc.bind
logger.jdbc-bind.level=trace
# JDBC result set extraction
logger.jdbc-extract.name=org.hibernate.orm.jdbc.extract
logger.jdbc-extract.level=trace

# JDBC batching
logger.jdbc-batch.name=org.hibernate.orm.jdbc.batch
logger.jdbc-batch.level=trace
```

SQL logging respects the settings `hibernate.format_sql` and `hibernate.highlight_sql`, so we don't miss out on the pretty formatting and highlighting.

2.10. Minimizing repetitive mapping information

The following properties are very useful for minimizing the amount of information you'll need to explicitly specify in `@Table` and `@Column` annotations, which we'll discuss below in Object/relational mapping:

Table 2.10: Settings for minimizing explicit mapping information

Configuration property name	Purpose
`hibernate.default_schema`	A default schema name for entities which do not explicitly declare one
`hibernate.default_catalog`	A default catalog name for entities which do not explicitly declare one
`hibernate.physical_naming_strategy`	A `PhysicalNamingStrategy` implementing your database naming standards
`hibernate.implicit_naming_strategy`	An `ImplicitNamingStrategy` which specifies how "logical" names of relational objects should be inferred when no name is specified in annotations

Writing your own `PhysicalNamingStrategy` and/or `ImplicitNamingStrategy` is an especially good way to reduce the clutter of annotations on your entity classes, and to implement your database naming conventions, and so we think you should do it for any nontrivial data model. We'll have more to say about them in Naming strategies.

2.11. Quoting SQL identifiers

By default, Hibernate never quotes a SQL table or column name in generated SQL when the name contains only alphanumeric characters. This behavior is usually much more convenient, especially when working with a legacy schema, since unquoted identifiers aren't case-sensitive, and so Hibernate doesn't need to know or care whether a column is named `NAME`, `name`, or `Name` on the database side. On the other hand, any table or column name containing a punctuation character like `$` is automatically quoted by default.

The following settings enable additional automatic quoting:

Table 2.11: Settings for identifier quoting

Configuration property name	Purpose
`hibernate.auto_quote_keyword`	Automatically quote any identifier which is a SQL keyword
`hibernate.globally_quoted_identifiers`	Automatically quote every identifier

Note that `hibernate.globally_quoted_identifiers` is a synonym for `<delimited-identifiers/>` in `persistence.xml`. We don't recommend the use of global identifier quoting, and in fact these settings are rarely used.

A better alternative is to explicitly quote table and column names where necessary, by writing `@Table(name="\"View\")` or `@Column(name="\"number\"")`. Since that's kinda ugly, Hibernate lets us use a backtick as the quote character instead of the double quote.

2.12. Nationalized character data in SQL Server

By default, SQL Server's `char` and `varchar` types don't accommodate Unicode data. But a Java string may contain any Unicode character. So, if you're working with SQL Server, you might need to force Hibernate to use the `nchar` and `nvarchar` column types.

Table 2.12: Setting the use of nationalized character data

Configuration property name	Purpose
`hibernate.use_nationalized_character_data`	Use `nchar` and `nvarchar` instead of `char` and `varchar`

On the other hand, if only *some* columns store nationalized data, use the `@Nationalized` annotation to indicate fields of your entities which map these columns.

Alternatively, you can configure SQL Server to use the UTF-8 enabled collation `_UTF8`.

2.13. Date and time types and JDBC

By default, Hibernate handles date and time types defined by `java.time` by:

- converting `java.time` types to JDBC date/time types defined in `java.sql` when sending data to the database, and
- reading `java.sql` types from JDBC and then converting them to `java.time` types when retrieving data from the database.

This works best when the database server time zone agrees with JVM system time zone.

We therefore recommend setting things up so that the database server and the JVM agree on the same time zone. **Hint:** when in doubt, UTC is quite a nice time zone.

There are two system configuration properties which influence this behavior:

Table 2.13: Settings for JDBC date/time handling

Configuration property name	Purpose
`hibernate.jdbc.time_zone`	Use an explicit time zone when interacting with JDBC
`hibernate.type.java_time_use_direct_jdbc`	Read and write `java.time` types directly to and from JDBC

You may set `hibernate.jdbc.time_zone` to the time zone of the database server if for some reason the JVM needs to operate in a different time zone. We do not recommend this approach.

On the other hand, we would love to recommend the use of `hibernate.type.java_time_use_direct_jdbc`, but this option is still experimental for now, and does result in some subtle differences in behavior which might affect legacy programs using Hibernate.

Chapter 3. Entities

An *entity* is a Java class which represents data in a relational database table. We say that the entity *maps* or *maps to* the table. Much less commonly, an entity might aggregate data from multiple tables, but we'll get to that later.

An entity has *attributes*—properties or fields—which map to columns of the table. In particular, every entity must have an *identifier* or *id*, which maps to the primary key of the table. The id allows us to uniquely associate a row of the table with an instance of the Java class, at least within a given *persistence context*.

We'll explore the idea of a persistence context later. For now, think of it as a one-to-one mapping between ids and entity instances.

An instance of a Java class cannot outlive the virtual machine to which it belongs. But we may think of an entity instance having a lifecycle which transcends a particular instantiation in memory. By providing its id to Hibernate, we may re-materialize the instance in a new persistence context, as long as the associated row is present in the database. Therefore, the operations `persist()` and `remove()` may be thought of as demarcating the beginning and end of the lifecycle of an entity, at least with respect to persistence.

Thus, an id represents the *persistent identity* of an entity, an identity that outlives a particular instantiation in memory. And this is an important difference between entity class itself and the values of its attributes—the entity has a persistent identity, and a well-defined lifecycle with respect to persistence, whereas a `String` or `List` representing one of its attribute values doesn't.

An entity usually has associations to other entities. Typically, an association between two entities maps to a foreign key in one of the database tables. A group of mutually associated entities is often called a *domain model*, though *data model* is also a perfectly good term.

3.1. Entity classes

An entity must:

- be a non-`final` class,
- with a non-`private` constructor with no parameters.

On the other hand, the entity class may be either concrete or `abstract`, and it may have any number of additional constructors.

> An entity class may be a `static` inner class.

Every entity class must be annotated `@Entity`.

```
@Entity
class Book {
    Book() {}
    ...
}
```

Alternatively, the class may be identified as an entity type by providing an XML-based mapping for the class.

Mapping entities using XML

When XML-based mappings are used, the `<entity>` element is used to declare an entity class:

```
<entity-mappings>
    <package>org.hibernate.example</package>

    <entity class="Book">
        <attributes> ... </attributes>
    </entity>

    ...
</entity-mappings>
```

Since the `orm.xml` mapping file format defined by the JPA specification was modelled closely on the annotation-based mappings, it's usually easy to go back and forth between the two options.

We won't have much more to say about XML-based mappings in this Short Guide, since it's not our preferred way to do things.

"Dynamic" models

We love representing entities as classes because the classes give us a *type-safe* model of our data. But Hibernate also has the ability to represent entities as detyped instances of `java.util.Map`. There's information in the User Guide, if you're curious.

This must sound like a weird feature for a project that places importance on type-safety. Actually, it's a useful capability for a very particular sort of generic code. For example, Hibernate Envers is a great auditing/versioning system for Hibernate entities. Envers makes use of maps to represent its *versioned model* of the data.

3.2. Access types

Each entity class has a default *access type*, either:

- direct *field access*, or
- *property access*.

Hibernate automatically determines the access type from the location of attribute-level annotations. Concretely:

- if a field is annotated `@Id`, field access is used, or
- if a getter method is annotated `@Id`, property access is used.

Back when Hibernate was just a baby, property access was quite popular in the Hibernate community. Today, however, field access is *much* more common.

> The default access type may be specified explicitly using the `@Access` annotation, but we strongly discourage this, since it's ugly and never necessary.

> Mapping annotations should be placed consistently:
>
> - if `@Id` annotates a field, the other mapping annotations should also be applied to fields, or,
> - if `@Id` annotates a getter, the other mapping annotations should be applied to getters.
>
> It is in principle possible to mix field and property access using explicit `@Access` annotations at the attribute level. We don't recommend doing this.

An entity class like `Book`, which does not extend any other entity class, is called a *root entity*. Every root entity must declare an identifier attribute.

3.3. Entity class inheritance

An entity class may `extend` another entity class.

```
@Entity
class AudioBook extends Book {
    AudioBook() {}
    ...
}
```

A subclass entity inherits every persistent attribute of every entity it extends.

A root entity may also extend another class and inherit mapped attributes from the other class. But in this case, the class which declares the mapped attributes must be annotated `@MappedSuperclass`.

```
@MappedSuperclass
class Versioned {
    ...
}

@Entity
class Book extends Versioned {
    ...
}
```

A root entity class must declare an attribute annotated `@Id`, or inherit one from a `@MappedSuperclass`. A subclass entity always inherits the identifier attribute of the root entity. It may not declare its own `@Id` attribute.

3.4. Identifier attributes

An identifier attribute is usually a field:

```
@Entity
class Book {
    Book() {}

    @Id
    Long id;

    ...
}
```

But it may be a property:

```
@Entity
class Book {
    Book() {}

    private Long id;

    @Id
    Long getId() { return id; }
    void setId(Long id) { this.id = id; }

    ...
}
```

An identifier attribute must be annotated `@Id` or `@EmbeddedId`.

Identifier values may be:

- assigned by the application, that is, by your Java code, or
- generated and assigned by Hibernate.

We'll discuss the second option first.

3.5. Generated identifiers

An identifier is often system-generated, in which case it should be annotated `@GeneratedValue`:

```
@Id @GeneratedValue
Long id;
```

System-generated identifiers, or *surrogate keys* make it easier to evolve or refactor the relational data model. If you have the freedom to define the relational schema, we recommend the use of surrogate keys. On the other hand, if, as is more common, you're working with a pre-existing database schema, you might not have the option.

JPA defines the following strategies for generating ids, which are enumerated by `GenerationType`:

Table 3.1: Standard id generation strategies

Strategy	Java type	Implementation
`GenerationType.UUID`	`UUID` or `String`	A Java `UUID`
`GenerationType.IDENTITY`	`Long` or `Integer`	An identity or autoincrement column
`GenerationType.SEQUENCE`	`Long` or `Integer`	A database sequence
`GenerationType.TABLE`	`Long` or `Integer`	A database table

Strategy	Java type	Implementation
`GenerationType.AUTO`	`Long` or `Integer`	Selects `SEQUENCE`, `TABLE`, or `UUID` based on the identifier type and capabilities of the database

For example, this UUID is generated in Java code:

```
@Id @GeneratedValue UUID id;  // AUTO strategy selects UUID based on the field type
```

This id maps to a SQL `identity`, `auto_increment`, or `bigserial` column:

```
@Id @GeneratedValue(strategy = IDENTITY) Long id;
```

The `@SequenceGenerator` and `@TableGenerator` annotations allow further control over `SEQUENCE` and `TABLE` generation respectively.

Consider this sequence generator:

```
@SequenceGenerator(name = "bookSeq", sequenceName = "seq_book", initialValue = 5, allocationSize=10)
```

Values are generated using a database sequence defined as follows:

```
create sequence seq_book start with 5 increment by 10
```

Notice that Hibernate doesn't have to go to the database every time a new identifier is needed. Instead, a given process obtains a block of ids, of size `allocationSize`, and only needs to hit the database each time the block is exhausted. Of course, the downside is that generated identifiers are not contiguous.

> If you let Hibernate export your database schema, the sequence definition will have the right `start with` and `increment` values. But if you're working with a database schema managed outside Hibernate, make sure the `initialValue` and `allocationSize` members of `@SequenceGenerator` match the `start with` and `increment` specified in the DDL.

Any identifier attribute may now make use of the generator named `bookSeq`:

```
@Id
@GeneratedValue(generator = "bookSeq")  // reference to generator defined elsewhere
Long id;
```

Actually, it's extremely common to place the `@SequenceGenerator` annotation on the `@Id` attribute that makes use of it:

```
@Id
@GeneratedValue  // uses the generator defined below
@SequenceGenerator(sequenceName = "seq_book", initialValue = 5, allocationSize=10)
Long id;
```

In this case, the `name` of the `@SequenceGenerator` should not be specified.

We may even place a `@SequenceGenerator` or `@TableGenerator` annotation at the package level:

```
@SequenceGenerator(sequenceName = "id_sequence", initialValue = 5, allocationSize=10)
@TableGenerator(table = "id_table", initialValue = 5, allocationSize=10)
package org.example.entities;
```

Then any entity in this package which specifies `strategy=SEQUENCE` or `strategy=TABLE` without also explicitly specifying a generator `name` will be assigned a generator based on the package-level annotation.

```
@Id
@GeneratedValue(strategy=SEQUENCE)  // uses the sequence generator defined at the package level
Long id;
```

As you can see, JPA provides quite adequate support for the most common strategies for system-generated ids. However, the annotations themselves are a bit more intrusive than they should be, and there's no well-defined way to extend this framework to support custom strategies for id generation. Nor may `@GeneratedValue` be used on a property not annotated `@Id`. Since custom id generation is a rather common requirement, Hibernate provides a very carefully-designed framework for user-defined `Generator`s, which we'll discuss in User-defined generators.

3.6. Natural keys as identifiers

Not every identifier attribute maps to a (system-generated) surrogate key. Primary keys which are meaningful to the user of the system are called *natural keys*.

When the primary key of a table is a natural key, we don't annotate the identifier attribute `@GeneratedValue`, and it's the responsibility of the application code to assign a value to the identifier attribute.

```
@Entity
class Book {
    @Id
    String isbn;

    ...
}
```

Of particular interest are natural keys which comprise more than one database column, and such natural keys are called *composite keys*.

3.7. Composite identifiers

If your database uses composite keys, you'll need more than one identifier attribute. There are two ways to map composite keys in JPA:

- using an `@IdClass`, or
- using an `@EmbeddedId`.

Perhaps the most immediately-natural way to represent this in an entity class is with multiple fields annotated `@Id`, for example:

```
@Entity
@IdClass(BookId.class)
class Book {
    Book() {}

    @Id
    String isbn;

    @Id
    int printing;

    ...
}
```

But this approach comes with a problem: what object can we use to identify a `Book` and pass to methods like `find()` which accept an identifier?

The solution is to write a separate class with fields that match the identifier attributes of the entity. Every such id class must override `equals()` and `hashCode()`. Of course, the easiest way to satisfy these requirements is to declare the id class as a `record`.

```
record BookId(String isbn, int printing) {}
```

The `@IdClass` annotation of the `Book` entity identifies `BookId` as the id class to use for that entity.

This is not our preferred approach. Instead, we recommend that the `BookId` class be declared as an `@Embeddable` type:

```
@Embeddable
record BookId(String isbn, int printing) {}
```

We'll learn more about Embeddable objects below.

Now the entity class may reuse this definition using `@EmbeddedId`, and the `@IdClass` annotation is no longer required:

```
@Entity
class Book {
    Book() {}

    @EmbeddedId
```

```
    BookId bookId;

    ...
}
```

This second approach eliminates some duplicated code.

Either way, we may now use `BookId` to obtain instances of `Book`:

```
Book book = session.find(Book.class, new BookId(isbn, printing));
```

3.8. Version attributes

An entity may have an attribute which is used by Hibernate for optimistic lock verification. A *version attribute* is usually of type `Integer`, `Short`, `Long`, `LocalDateTime`, `OffsetDateTime`, `ZonedDateTime`, or `Instant`.

```
@Version
int version;
```

```
@Version
LocalDateTime lastUpdated;
```

A version attribute is automatically assigned by Hibernate when an entity is made persistent, and automatically incremented or updated each time the entity is updated.

If the version attribute is numeric, then an entity is, by default, assigned the version number `0` when it's first made persistent. It's easy to specify that the initial version should be assigned the number `1` instead:

```
@Version
int version = 1; // the initial version number
```

If the version attribute is an `Instant` or datetime, the value assigned to the attribute is generated in the JVM by default.

> If the datetime value should be generated by the database, the `@Version` annotation may be used in conjunction with a generator annotation, for example:
>
> ```
> @CurrentTimestamp @Version
> LocalDateTime lastUpdated;
> ```

Almost every entity which is frequently updated should have a version attribute.

> If an entity doesn't have a version number, which often happens when mapping legacy data, we can still do optimistic locking. The `@OptimisticLocking` annotation lets us specify that optimistic locks should be checked by validating the values of `ALL` fields, or only the `DIRTY` fields of the entity. And the `@OptimisticLock` annotation lets us selectively exclude certain fields from optimistic locking.

The `@Id` and `@Version` attributes we've already seen are just specialized examples of *basic attributes*.

3.9. Natural id attributes

Even when an entity has a surrogate key, it should always be possible to write down a combination of fields which uniquely identifies an instance of the entity, from the point of view of the user of the system. This combination of fields is its natural key. Above, we considered the case where the natural key coincides with the primary key. Here, the natural key is a second unique key of the entity, distinct from its surrogate primary key.

> If you can't identify a natural key, it might be a sign that you need to think more carefully about some aspect of your data model. If an entity doesn't have a meaningful unique key, then it's impossible to say what event or object it represents in the "real world" outside your program.

Since it's *extremely* common to retrieve an entity based on its natural key, Hibernate has a way to mark the attributes of the entity which make up its natural key. Each attribute must be annotated `@NaturalId`.

```
@Entity
```

```
class Book {
    Book() {}

    @Id @GeneratedValue
    Long id; // the system-generated surrogate key

    @NaturalId
    String isbn; // belongs to the natural key

    @NaturalId
    int printing; // also belongs to the natural key

    ...
}
```

Hibernate automatically generates a `UNIQUE` constraint on the columns mapped by the annotated fields.

> Consider using the natural id attributes to implement `equals()` and `hashCode()`.

In cases where the natural id is defined by multiple attributes, Hibernate also offers the `@NaturalIdClass` annotation which acts similarly to the Jakarta Persistence `@IdClass` annotation for find operations -

```
record BookKey(String isbn, int printing) {}

@Entity
@NaturalIdClass(BookKey.class)
class Book {
    ...
}
```

See the User Guide for more details about natural ids.

The payoff for doing this extra work, as we will see much later, is that we can take advantage of optimized natural id lookups that make use of the second-level cache.

Note that even when you've identified a natural key, we still recommend the use of a generated surrogate key in foreign keys, since this makes your data model *much* easier to change.

3.10. Basic attributes

A *basic* attribute of an entity is a field or property which maps to a single column of the associated database table. The JPA specification defines a quite limited set of basic types:

Table 3.2: JPA-standard basic attribute types

Classification	Package	Types
Primitive types		`boolean`, `int`, `double`, etc
Primitive wrappers	`java.lang`	`Boolean`, `Integer`, `Double`, etc
Strings	`java.lang`	`String`
Arbitrary-precision numeric types	`java.math`	`BigInteger`, `BigDecimal`
UUIDs	`java.util`	`UUID`
Date/time types	`java.time`	`LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetDateTime`, `Instant`, `Year`
Deprecated date/time types 💀	`java.util`	`Date`, `Calendar`
Deprecated JDBC date/time types 💀	`java.sql`	`Date`, `Time`, `Timestamp`
Binary and character arrays		`byte[]`, `char[]`
Binary and character wrapper arrays 💀	`java.lang`	`Byte[]`, `Character[]`

Classification	Package	Types
Enumerated types		Any `enum`
Serializable types		Any type which implements `java.io.Serializable`

We're begging you to use types from the `java.time` package instead of anything which inherits `java.util.Date`.

The use of `Byte[]` and `Character[]` as basic types was deprecated by Jakarta Persistence 3.2. Hibernate does not allow `null` elements in such arrays. Use `byte[]` or `char[]` instead.

Serializing a Java object and storing its binary representation in the database is usually wrong. As we'll soon see in Embeddable objects, Hibernate has much better ways to handle complex Java objects.

Hibernate slightly extends this list with the following types:

Table 3.3: Additional basic attribute types in Hibernate

Classification	Package	Types
Additional date/time types	`java.time`	`Duration`, `ZoneId`, `ZoneOffset`, and even `ZonedDateTime`
JDBC LOB types	`java.sql`	`Blob`, `Clob`, `NClob`
Java class object	`java.lang`	`Class`
Internet addresses	`java.net`	`InetAddress`
Miscellaneous types	`java.util`	`Currency`, `Locale`, `URL`, `TimeZone`

The `@Basic` annotation explicitly specifies that an attribute is basic, but it's often not needed, since attributes are assumed basic by default. On the other hand, if a non-primitively-typed attribute cannot be null, use of `@Basic(optional=false)` is highly recommended.

```
@Basic(optional=false) String firstName;
@Basic(optional=false) String lastName;
String middleName; // may be null
```

Note that primitively-typed attributes are inferred `NOT NULL` by default.

How to make a column `not null` in JPA

There are two standard ways to add a `NOT NULL` constraint to a mapped column in JPA:

- using `@Basic(optional=false)`, or
- using `@Column(nullable=false)`.

You might wonder what the difference is.

Well, it's perhaps not obvious to a casual user of the JPA annotations, but they actually come in two "layers":

- annotations like `@Entity`, `@Id`, and `@Basic` belong to the *logical* layer, the subject of the current chapter—they specify the semantics of your Java domain model, whereas
- annotations like `@Table` and `@Column` belong to the *mapping* layer, the topic of the next chapter—they specify how elements of the domain model map to objects in the relational database.

Information may be inferred from the logical layer down to the mapping layer, but is never inferred in the opposite direction.

Now, the `@Column` annotation, to whom we'll be properly introduced a bit later, belongs to the *mapping* layer, and so its `nullable` member only affects schema generation (resulting in a `not null` constraint in the generated DDL). On the other hand, the `@Basic` annotation belongs to the logical layer, and so an attribute marked `optional=false` is checked by Hibernate before it even writes an entity to the database. Note that:

- `optional=false` implies `nullable=false`, but

- `nullable=false` *does not* imply `optional=false`.

Therefore, we prefer `@Basic(optional=false)` to `@Column(nullable=false)`.

> But wait! An even better solution is to use the `@NotNull` annotation from Bean Validation. Just add Hibernate Validator to your project build, as described in Optional dependencies.

3.11. Enumerated types

We included Java `enum`s on the list above. An enumerated type is considered a sort of basic type, but since most databases don't have a native `ENUM` type, JPA provides a special `@Enumerated` annotation to specify how the enumerated values should be represented in the database:

- by default, an enum is stored as an integer, the value of its `ordinal()` member, but
- if the attribute is annotated `@Enumerated(STRING)`, it will be stored as a string, the value of its `name()` member.

```
//here, an ORDINAL encoding makes sense
@Enumerated
@Basic(optional=false)
DayOfWeek dayOfWeek;

//but usually, a STRING encoding is better
@Enumerated(EnumType.STRING)
@Basic(optional=false)
Status status;
```

The `@EnumeratedValue` annotation allows the column value to be customized:

```
enum Resolution {
    UNRESOLVED(0), FIXED(1), REJECTED(-1);

    @EnumeratedValue // store the code, not the enum ordinal() value
    final int code;

    Resolution(int code) {
        this.code = code;
    }
}
```

Since Hibernate 6, an `enum` annotated `@Enumerated(STRING)` is mapped to:

- a `VARCHAR` column type with a `CHECK` constraint on most databases, or
- an `ENUM` column type on MySQL.

Any other `enum` is mapped to a `TINYINT` column with a `CHECK` constraint.

> JPA picks the wrong default here. In most cases, storing an integer encoding of the `enum` value makes the relational data harder to interpret.
>
> Even considering `DayOfWeek`, the encoding to integers is ambiguous. If you check `java.time.DayOfWeek`, you'll notice that `SUNDAY` is encoded as `6`. But in the country I was born, `SUNDAY` is the *first* day of the week!
>
> So we prefer `@Enumerated(STRING)` for most `enum` attributes.

An interesting special case arises on PostgreSQL and Oracle.

Named enumerated types

Some databases support *named* `ENUM` types, which must be declared using in DDL using:

- `CREATE TYPE … AS ENUM` on PostgreSQL, or
- `CREATE DOMAIN … AS ENUM` on Oracle.

These look like a perfect match for Java `enum`s, which also have names!

Sadly, these `ENUM` types aren't well-integrated with the SQL language, nor well-supported by the JDBC drivers, so Hibernate doesn't use them by default. But if you would like to use a named enumerated type on Postgres or Oracle, just annotate your `enum` attribute like this:

```
@JdbcTypeCode(SqlTypes.NAMED_ENUM)
@Basic(optional=false)
Status status;
```

Alternatively, you may enable the configuration property `hibernate.type.prefer_native_enum_types`.

The limited set of pre-defined basic attribute types can be stretched a bit further by supplying a *converter*.

3.12. Converters

A JPA `AttributeConverter` is responsible for:

- converting a given Java type to one of the types listed above, and/or
- perform any other sort of pre- and post-processing you might need to perform on a basic attribute value before writing and reading it to or from the database.

Converters substantially widen the set of attribute types that can be handled by JPA.

There are two ways to apply a converter:

- the `@Convert` annotation applies an `AttributeConverter` to a particular entity attribute, or
- the `@Converter` annotation (or, alternatively, the `@ConverterRegistration` annotation) registers an `AttributeConverter` for automatic application to all attributes of a given type.

For example, the following converter will be automatically applied to any attribute of type `EnumSet<DayOfWeek>`, and takes care of persisting the `EnumSet<DayOfWeek>` to a column of type `INTEGER`:

```
@Converter(autoApply = true)
public static class EnumSetConverter
        // converts Java values of type EnumSet<DayOfWeek> to integers for storage in an INT column
        implements AttributeConverter<EnumSet<DayOfWeek>,Integer> {
    @Override
    public Integer convertToDatabaseColumn(EnumSet<DayOfWeek> enumSet) {
        int encoded = 0;
        var values = DayOfWeek.values();
        for (int i = 0; i<values.length; i++) {
            if (enumSet.contains(values[i])) {
                encoded |= 1<<i;
            }
        }
        return encoded;
    }

    @Override
    public EnumSet<DayOfWeek> convertToEntityAttribute(Integer encoded) {
        var set = EnumSet.noneOf(DayOfWeek.class);
        var values = DayOfWeek.values();
        for (int i = 0; i<values.length; i++) {
            if (((1<<i) & encoded) != 0) {
                set.add(values[i]);
            }
        }
        return set;
    }
}
```

On the other hand, if we *don't* set `autoapply=true`, then we must explicitly apply the converter using the `@Convert` annotation:

```
@Convert(converter = EnumSetConverter.class)
@Basic(optional = false)
EnumSet<DayOfWeek> daysOfWeek;
```

> Converters are supposed to be used for type conversion. Some enterprising members of the community have noticed that they can be (mis)used to perform other tasks: trimming whitespace, normalizing case, assigning a default value in place of `null`, and so on. Hibernate *tolerates* but does not encourage such (mis)use. In particular, we strongly recommend against defining an `autoApply` converter acting on a basic type.

All this is nice, but it probably won't surprise you that Hibernate goes beyond what is required by JPA.

3.13. Compositional basic types

Hibernate considers a "basic type" to be formed by the marriage of two objects:

- a `JavaType`, which models the semantics of a certain Java class, and
- a `JdbcType`, representing a SQL type which is understood by JDBC.

When mapping a basic attribute, we may explicitly specify a `JavaType`, a `JdbcType`, or both.

JavaType

An instance of `org.hibernate.type.descriptor.java.JavaType` represents a particular Java class. It's able to:

- compare instances of the class to determine if an attribute of that class type is dirty (modified),
- produce a useful hash code for an instance of the class,
- coerce values to other types, and, in particular,
- convert an instance of the class to one of several other equivalent Java representations at the request of its partner `JdbcType`.

For example, `IntegerJavaType` knows how to convert an `Integer` or `int` value to the types `Long`, `BigInteger`, and `String`, among others.

We may explicitly specify a Java type using the `@JavaType` annotation, but for the built-in `JavaType`s this is never necessary.

```
@JavaType(LongJavaType.class)  // not needed, this is the default JavaType for long
long currentTimeMillis;
```

For a user-written `JavaType`, the annotation is more useful:

```
@JavaType(BitSetJavaType.class)
BitSet bitSet;
```

Alternatively, the `@JavaTypeRegistration` annotation may be used to register `BitSetJavaType` as the default `JavaType` for `BitSet`.

JdbcType

An `org.hibernate.type.descriptor.jdbc.JdbcType` is able to read and write a single Java type from and to JDBC.

For example, `VarcharJdbcType` takes care of:

- writing Java strings to JDBC `PreparedStatement`s by calling `setString()`, and
- reading Java strings from JDBC `ResultSet`s using `getString()`.

By pairing `LongJavaType` with `VarcharJdbcType` in holy matrimony, we produce a basic type which maps `Long`s and primitive `longs`s to the SQL type `VARCHAR`.

We may explicitly specify a JDBC type using the `@JdbcType` annotation.

```
@JdbcType(VarcharJdbcType.class)
long currentTimeMillis;
```

Alternatively, we may specify a JDBC type code:

```
@JdbcTypeCode(Types.VARCHAR)
long currentTimeMillis;
```

The `@JdbcTypeRegistration` annotation may be used to register a user-written `JdbcType` as the default for a given SQL type code.

JDBC types and JDBC type codes

The types defined by the JDBC specification are enumerated by the integer type codes in the class `java.sql.Types`. Each JDBC type is an abstraction of a commonly-available type in SQL. For example, `Types.VARCHAR` represents the SQL type `VARCHAR` (or `VARCHAR2` on Oracle).

Since Hibernate understands more SQL types than JDBC, there's an extended list of integer type codes in the class `org.hibernate.type.SqlTypes`. For example, `SqlTypes.GEOMETRY` represents the spatial data type `GEOMETRY`.

AttributeConverter

If a given `JavaType` doesn't know how to convert its instances to the type required by its partner `JdbcType`, we must help it out by providing a JPA `AttributeConverter` to perform the conversion.

For example, to form a basic type using `LongJavaType` and `TimestampJdbcType`, we would provide an `AttributeConverter<Long,Timestamp>`.

```
@JdbcType(TimestampJdbcType.class)
@Convert(converter = LongToTimestampConverter.class)
long currentTimeMillis;
```

Let's abandon our analogy right here, before we start calling this basic type a "throuple".

3.14. Date and time types, and time zones

Dates and times should always be represented using the types defined in `java.time`.

⚠ Never use the legacy types `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, or `java.util.Date`. At our urging, support for these types has even been officially deprecated in JPA 3.2. Eventually, we hope to completely remove support for these types from the JPA spec and from Hibernate.

Some of the types in `java.time` map naturally to an ANSI SQL column type. A source of confusion is that some databases still don't follow the ANSI standard naming here. Also, as you're probably aware, the `DATE` type on Oracle is not an ANSI SQL `DATE`. In fact, Oracle doesn't have `DATE` or `TIME` types—every date or time must be stored as a timestamp.

Table 3.4: Type mappings from `java.time` to ANSI SQL

`java.time` class	ANSI SQL type	MySQL	SQL Server	Oracle
`LocalDate`	`DATE`	`DATE`	`DATE`	`DATE` 💀
`LocalTime`	`TIME`	`TIME`	`TIME`	`TIMESTAMP` 💀
`LocalDateTime`	`TIMSTAMP`	`DATETIME`	`DATETIME2`	`TIMESTAMP`
`OffsetDateTime`, `ZonedDateTime`	`TIMESTAMP WITH TIME ZONE`	`TIMESTAMP` 🙄	`DATETIMEOFFSET`	`TIMESTAMP WITH TIME ZONE`

On the other hand, there are no perfectly natural mappings for `Instant` and `Duration` on most databases. By default:

- `Duration` is mapped to a column of type `NUMERIC(21)` holding the length of the duration in nanoseconds, and
- `Instant` is mapped to a column of type `TIMESTAMP` (`DATETIME` on MySQL).

Fortunately, these mappings can be modified by specifying the `JdbcType`.

For example, if we wanted to store an `Instant` using `TIMESTAMP WITH TIME ZONE` (`TIMESTAMP` on MySQL) instead of `TIMESTAMP`, then we could annotate the field:

```
// store the Instant as a TIMESTAMP WITH TIME ZONE, instead of as a TIMESTAMP
@JdbcTypeCode(SqlTypes.TIMESTAMP_WITH_TIMEZONE)
Instant instant;
```

Alternatively, we could set the configuration property `hibernate.type.preferred_instant_jdbc_type`:

```
// store field of type Instant as TIMESTAMP WITH TIME ZONE, instead of as a TIMESTAMP
```

```
config.setProperty(MappingSettings.PREFERRED_INSTANT_JDBC_TYPE, SqlTypes.TIMESTAMP_WITH_TIMEZONE);
```

We have worked very hard to make sure that Java date and time types work with consistent and correct semantics across all databases supported by Hibernate. In particular, Hibernate is very careful in how it handles time zones.

> Unfortunately, with the notable exception of Oracle, most SQL databases feature embarrassingly poor support for timezones. Even some databases which do supposedly support `TIMESTAMP WITH TIME ZONE` simply covert the datetime to UTC. Here, Hibernate is limited by the capabilities of the databases themselves, and so on many databases, time zone information will not, by default, be preserved for an `OffsetDateTime` or `ZonedDateTime`.

> The still-experimental annotation `@TimeZoneStorage` provides some additional options in case the default behavior falls short.

3.15. Embeddable objects

An embeddable object is a Java class whose state maps to multiple columns of a table, but which doesn't have its own persistent identity. That is, it's a class with mapped attributes, but no `@Id` attribute.

An embeddable object can only be made persistent by assigning it to the attribute of an entity. Since the embeddable object does not have its own persistent identity, its lifecycle with respect to persistence is completely determined by the lifecycle of the entity to which it belongs.

An embeddable class must be annotated `@Embeddable` instead of `@Entity`.

```
@Embeddable
class Name {

    @Basic(optional=false)
    String firstName;

    @Basic(optional=false)
    String lastName;

    String middleName;

    Name() {}

    Name(String firstName, String middleName, String lastName) {
        this.firstName = firstName;
        this.middleName = middleName;
        this.lastName = lastName;
    }

    ...
}
```

An embeddable class must satisfy the same requirements that entity classes satisfy, with the exception that an embeddable class has no `@Id` attribute. In particular, it must have a constructor with no parameters.

Alternatively, an embeddable type may be defined as a Java record type:

```
@Embeddable
record Name(String firstName, String middleName, String lastName) {}
```

In this case, the requirement for a constructor with no parameters is relaxed.

We may now use our `Name` class (or record) as the type of an entity attribute:

```
@Entity
class Author {
    @Id @GeneratedValue
    Long id;

    Name name;

    ...
```

```
}
```

Embeddable types can be nested. That is, an `@Embeddable` class may have an attribute whose type is itself a different `@Embeddable` class.

> JPA provides an `@Embedded` annotation to identify an attribute of an entity that refers to an embeddable type. This annotation is completely optional, and so we don't usually use it.

On the other hand a reference to an embeddable type is *never* polymorphic. One `@Embeddable` class `F` may inherit a second `@Embeddable` class `E`, but an attribute of type `E` will always refer to an instance of that concrete class `E`, never to an instance of `F`.

Usually, embeddable types are stored in a "flattened" format. Their attributes map columns of the table of their parent entity. Later, in Mapping embeddable types to UDTs or to JSON, we'll see a couple of different options.

An attribute of embeddable type represents a relationship between a Java object with a persistent identity, and a Java object with no persistent identity. We can think of it as a whole/part relationship. The embeddable object belongs to the entity, and can't be shared with other entity instances. And it exists for only as long as its parent entity exists.

Next we'll discuss a different kind of relationship: a relationship between Java objects which each have their own distinct persistent identity and persistence lifecycle.

3.16. Associations

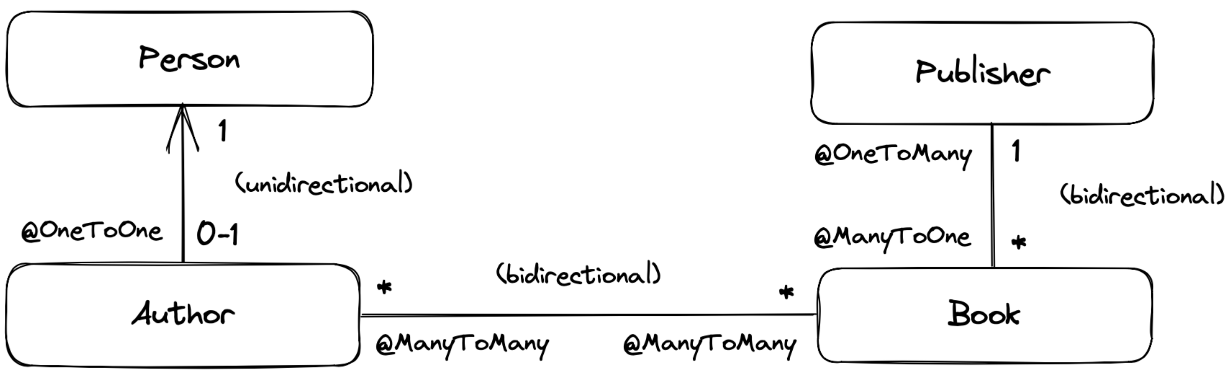
An *association* is a relationship between entities. We usually classify associations based on their *multiplicity*. If `E` and `F` are both entity classes, then:

- a *one-to-one* association relates at most one unique instance `E` with at most one unique instance of `F`,
- a *many-to-one* association relates zero or more instances of `E` with a unique instance of `F`, and
- a *many-to-many* association relates zero or more instances of `E` with zero or more instance of `F`.

An association between entity classes may be either:

- *unidirectional*, navigable from `E` to `F` but not from `F` to `E`, or
- *bidirectional*, and navigable in either direction.

In this example data model, we can see the sorts of associations which are possible:



> An astute observer of the diagram above might notice that the relationship we've presented as a unidirectional one-to-one association could reasonably be represented in Java using subtyping. This is quite normal. A one-to-one association is the usual way we implement subtyping in a fully-normalized relational model. It's related to the `JOINED` inheritance mapping strategy.

There are three annotations for mapping associations: `@ManyToOne`, `@OneToMany`, and `@ManyToMany`. They share some common annotation members:

Table 3.5: Association-defining annotation members

Member	Interpretation	Default value
`cascade`	Persistence operations which should cascade to the associated entity; a list of `CascadeTypes`	`{}`

Member	Interpretation	Default value
`fetch`	Whether the association is eagerly fetched or may be proxied	• `LAZY` for `@OneToMany` and `@ManyToMany` • `EAGER` for `@ManyToOne` 💀 💀 💀
`targetEntity`	The associated entity class	Determined from the attribute type declaration
`optional`	For a `@ManyToOne` or `@OneToOne` association, whether the association can be `null`	`true`
`mappedBy`	For a bidirectional association, an attribute of the associated entity which maps the association	By default, the association is assumed unidirectional

We'll explain the effect of these members as we consider the various types of association mapping.

It's not a requirement to represent *every* foreign key relationship as an association at the Java level. It's perfectly acceptable to replace a `@ManyToOne` mapping with a basic-typed attribute holding an identifier, if it's inconvenient to think of this relationship as an association at the Java level. That said, it's possible to take this idea way too far.

💀 Aggregates 💀

It's come to our attention that a vocal group of people advocate that Java entity classes should be broken up into tiny disconnected islands they call "aggregates". An aggregate—at least as a first approximation—corresponds roughly to what we would usually call a parent/child relationship. Simple examples of aggregates might be `Order`/`Item`, or `Product`/`Part`. According to this way of thinking, there should be no associations *between* aggregates. So the `Item.product` association should be replaced with `productId`, `Part.manufacturer` should be replaced with `manufacturerId`, and so on. (Of course, the word "aggregate" may also be employed in other senses, but this is the sense we're discussing right now.)

In the example we've been using, `Book` would not be permitted to have a collection of entity type `Author`, and should instead hold only the ids of the authors, or perhaps instances of some `BookAuthor` type which duplicates some state of `Author` and is disconnected from the rest of the model.

Let's stipulate that this might be a perfectly natural thing to do in certain contexts, for example, when accessing a document database. But one context where it doesn't usually make sense is when accessing a relational database via Hibernate. The reason is that Hibernate offers rich functionality for optimizing access to associated data, including:

- the second level cache, and
- join, batch, and subselect fetching, whether via HQL, entity graphs, or fetch profiles.

But all this functionality is lost if Hibernate doesn't know it's dealing with an association, inevitably making the application program much more vulnerable to problems with N+1 selects, just as soon as we encounter a business requirement which involves data from more than one aggregate. (Always keep in mind that business requirements change much faster than relational data models!)

To put it mildly: this is not how JPA was ever intended to be used.

It's difficult to respond charitably to most of the arguments in favor of this approach, since most of them don't rise above the level of hand-waving at boxes on drawn on whiteboards. An argument we *can* respond to is the concern that transparent lazy fetching can lead to "accidental" fetching of an association and the potential for N+1 selects. This is a legit concern, and one we worry about too, but where it's really a problem we have a much better solution: just use a `StatelessSession`, or a Jakarta Data repository, where association fetching is always an explicit operation. Indeed, `StatelessSession` even guards against accidental *updates*, since `update()` is always an explicit operation.

Now that we know that associations are actually good and useful, let's see how to model the various kinds of association we might find need to map to a relational data model. We begin with the most common association multiplicity.

3.17. Many-to-one

A many-to-one association is the most basic sort of association we can imagine. It maps completely naturally to a foreign key in the database. Almost all the associations in your domain model are going to be of this form.

> Later, we'll see how to map a many-to-one association to an association table.

The `@ManyToOne` annotation marks the "to one" side of the association, so a unidirectional many-to-one association looks like this:

```
class Book {
```

```java
    @Id @GeneratedValue
    Long id;

    @ManyToOne(fetch=LAZY)
    Publisher publisher;

    ...
}
```

Here, the Book table has a foreign key column holding the identifier of the associated Publisher.

> A very unfortunate misfeature of JPA is that @ManyToOne associations are fetched eagerly by default. This is almost never what we want. Almost all associations should be lazy. The only scenario in which fetch=EAGER makes sense is if we think there's always a *very* high probability that the associated object will be found in the second-level cache. Whenever this isn't the case, remember to explicitly specify fetch=LAZY.

Most of the time, we would like to be able to easily navigate our associations in both directions. We do need a way to get the Publisher of a given Book, but we would also like to be able to obtain all the Books belonging to a given publisher.

To make this association bidirectional, we need to add a collection-valued attribute to the Publisher class, and annotate it @OneToMany.

> Hibernate needs to proxy unfetched associations at runtime. Therefore, the many-valued side must be declared using an interface type like Set or List, and never using a concrete type like HashSet or ArrayList.

To indicate clearly that this is a bidirectional association, and to reuse any mapping information already specified in the Book entity, we must use the mappedBy annotation member to refer back to Book.publisher.

```java
@Entity
class Publisher {
    @Id @GeneratedValue
    Long id;

    @OneToMany(mappedBy="publisher")
    Set<Book> books;

    ...
}
```

The Publisher.books field is called the *unowned* side of the association.

Now, we passionately *hate* the stringly-typed mappedBy reference to the owning side of the association. Thankfully, the Hibernate Processor gives us a way to make it a bit more type safe:

```java
@OneToMany(mappedBy=Book_.PUBLISHER)  // get used to doing it this way!
Set<Book> books;
```

We're going to use this approach for the rest of the Short Guide.

To modify a bidirectional association, we must change the *owning side*.

> Changes made to the unowned side of an association are never synchronized to the database. If we desire to change an association in the database, we must change it from the owning side. Here, we must set Book.publisher.
>
> In fact, it's often necessary to change *both sides* of a bidirectional association. For example, if the collection Publisher.books was stored in the second-level cache, we must also modify the collection, to ensure that the second-level cache remains synchronized with the database.

That said, it's *not* a hard requirement to update the unowned side, at least if you're sure you know what you're doing.

> In principle Hibernate *does* allow you to have a unidirectional one-to-many, that is, a @OneToMany with no matching @ManyToOne on the other side. In practice, this mapping is unnatural, and just doesn't work very well. Avoid it.

Here we've used Set as the type of the collection, but Hibernate also allows the use of List or Collection here, with almost no difference in semantics. In particular, the List may not contain duplicate elements, and its order will not be persistent.

```java
@OneToMany(mappedBy=Book_.PUBLISHER)
```

```
    Collection<Book> books;
```

We'll see how to map a collection with a persistent order much later.

Set, List, or Collection?

A one-to-many association mapped to a foreign key can never contain duplicate elements, so `Set` seems like the most semantically correct Java collection type to use here, and so that's the conventional practice in the Hibernate community.

The catch associated with using a set is that we must carefully ensure that `Book` has a high-quality implementation of `equals()` and `hashCode()`. Now, that's not necessarily a bad thing, since a quality `equals()` is independently useful.

But what if we used `Collection` or `List` instead? Then our code would be much less sensitive to how `equals()` and `hashCode()` were implemented.

In the past, we were perhaps too dogmatic in recommending the use of `Set`. Now? I guess we're happy to let you guys decide. In hindsight, we could have done more to make clear that this was always a viable option.

3.18. One-to-one (first way)

The simplest sort of one-to-one association is almost exactly like a `@ManyToOne` association, except that it maps to a foreign key column with a `UNIQUE` constraint.

> Later, we'll see how to map a one-to-one association to an association table.

A one-to-one association must be annotated `@OneToOne`:

```
@Entity
class Author {
    @Id @GeneratedValue
    Long id;

    @OneToOne(optional=false, fetch=LAZY)
    Person person;

    ...
}
```

Here, the `Author` table has a foreign key column holding the identifier of the associated `Person`.

> A one-to-one association often models a "type of" relationship. In our example, an `Author` is a type of `Person`. An alternative—and often more natural—way to represent "type of" relationships in Java is via entity class inheritance.

We can make this association bidirectional by adding a reference back to the `Author` in the `Person` entity:

```
@Entity
class Person {
    @Id @GeneratedValue
    Long id;

    @OneToOne(mappedBy = Author_.PERSON)
    Author author;

    ...
}
```

`Person.author` is the unowned side, because it's the side marked `mappedBy`.

Lazy fetching for one-to-one associations

Notice that we did not declare the unowned end of the association `fetch=LAZY`. That's because:

1. not every `Person` has an associated `Author`, and

2. the foreign key is held in the table mapped by `Author`, not in the table mapped by `Person`.

Therefore, Hibernate can't tell if the reference from `Person` to `Author` is `null` without fetching the associated `Author`.

On the other hand, if *every* `Person` was an `Author`, that is, if the association were non-`optional`, we would not have to consider the possibility of `null` references, and we would map it like this:

```
@OneToOne(optional=false, mappedBy = Author_.PERSON, fetch=LAZY)
Author author;
```

This is not the only sort of one-to-one association.

3.19. One-to-one (second way)

An arguably more elegant way to represent such a relationship is to share a primary key between the two tables.

To use this approach, the `Author` class must be annotated like this:

```
@Entity
class Author {
    @Id
    Long id;

    @OneToOne(optional=false, fetch=LAZY)
    @MapsId
    Person person;

    ...
}
```

Notice that, compared with the previous mapping:

- the `@Id` attribute is no longer a `@GeneratedValue` and,
- instead, the `author` association is annotated `@MapsId`.

This lets Hibernate know that the association to `Person` is the source of primary key values for `Author`.

Here, there's no extra foreign key column in the `Author` table, since the `id` column holds the identifier of `Person`. That is, the primary key of the `Author` table does double duty as the foreign key referring to the `Person` table.

The `Person` class doesn't change. If the association is bidirectional, we annotate the unowned side `@OneToOne(mappedBy = Author_.PERSON)` just as before.

3.20. Many-to-many

A unidirectional many-to-many association is represented as a collection-valued attribute. It always maps to a separate *association table* in the database.

It tends to happen that a many-to-many association eventually turns out to be an entity in disguise.

> Suppose we start with a nice clean many-to-many association between `Author` and `Book`. Later on, it's quite likely that we'll discover some additional information which comes attached to the association, so that the association table needs some extra columns.
>
> For example, imagine that we needed to report the percentage contribution of each author to a book. That information naturally belongs to the association table. We can't easily store it as an attribute of `Book`, nor as an attribute of `Author`.
>
> When this happens, we need to change our Java model, usually introducing a new entity class which maps the association table directly. In our example, we might call this entity something like `BookAuthorship`, and it would have `@OneToMany` associations to both `Author` and `Book`, along with the `contribution` attribute.
>
> We can evade the disruption occasioned by such "discoveries" by simply avoiding the use of `@ManyToMany` right from the start. There's little downside to representing every—or at least *almost* every—logical many-to-many association using an intermediate entity.

A many-to-many association must be annotated `@ManyToMany`:

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany
    Set<Author> authors;

    ...
}
```

If the association is bidirectional, we add a very similar-looking attribute to `Book`, but this time we must specify `mappedBy` to indicate that this is the unowned side of the association:

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany(mappedBy=Author_.BOOKS)
    Set<Author> authors;

    ...
}
```

Remember, if we wish to the modify the collection we must change the owning side.

We've again used `Set`s to represent the association. As before, we have the option to use `Collection` or `List`. But in this case it *does* make a difference to the semantics of the association.

> A many-to-many association represented as a `Collection` or `List` may contain duplicate elements. However, as before, the order of the elements is not persistent. That is, the collection is a *bag*, not a set.

> We don't usually map collections with `fetch=EAGER`, since that usually leads to poor performance and fetching of unnecessary data. But this is especially clear in the case of many-to-many associations. We don't much employ the word "never" when it comes to object/relational mappings, but here we will: **never** write `@ManyToMany(fetch=EAGER)` unless you're deliberately looking for trouble.

3.21. Collections of basic values and embeddable objects

We've now seen the following kinds of entity attribute:

Kind of entity attribute	Kind of reference	Multiplicity	Examples
Single-valued attribute of basic type	Non-entity	At most one	`@Basic String name`
Single-valued attribute of embeddable type	Non-entity	At most one	`@Embedded Name name`
Single-valued association	Entity	At most one	`@ManyToOne Publisher publisher` `@OneToOne Person person`
Many-valued association	Entity	Zero or more	`@OneToMany Set<Book> books` `@ManyToMany Set<Author> authors`

Scanning this taxonomy, you might ask: does Hibernate have multivalued attributes of basic or embeddable type?

Well, actually, we've already seen that it does, at least in two special cases. So first, lets recall that JPA treats `byte[]` and `char[]` arrays as basic types. Hibernate persists a `byte[]` or `char[]` array to a `VARBINARY` or `VARCHAR` column, respectively.

But in this section we're really concerned with cases *other* than these two special cases. So then, *apart from* `byte[]` *and* `char[]`, does Hibernate have multivalued attributes of basic or embeddable type?

And the answer again is that *it does*. Indeed, there are two different ways to handle such a collection, by mapping it:

- to a column of SQL `ARRAY` type (assuming the database has an `ARRAY` type), or

- to a separate table.

So we may expand our taxonomy with:

Kind of entity attribute	Kind of reference	Multiplicity	Examples
`byte[]` and `char[]` arrays	Non-entity	Zero or more	`byte[] image` `char[] text`
Collection of basic-typed elements	Non-entity	Zero or more	`@Array String[] names` `@ElementCollection Set<String> names`
Collection of embeddable elements	Non-entity	Zero or more	`@ElementCollection Set<Name> names`

There's actually two new kinds of mapping here: `@Array` mappings, and `@ElementCollection` mappings.

> These sorts of mappings are overused.
>
> There *are* situations where we think it's appropriate to use a collection of basic-typed values in our entity class. But such situations are rare. Almost every many-valued relationship should map to a foreign key association between separate tables. And almost every table should be mapped by an entity class.
>
> The features we're about to meet in the next two subsections are used much more often by beginners than they're used by experts. So if you're a beginner, you'll save yourself same hassle by staying away from these features for now.

We'll talk about `@Array` mappings first.

3.22. Collections mapped to SQL arrays

Let's consider a calendar event which repeats on certain days of the week. We might represent this in our `Event` entity as an attribute of type `DayOfWeek[]` or `List<DayOfWeek>`. Since the number of elements of this array or list is upper bounded by 7, this is a reasonable case for the use of an `ARRAY`-typed column. It's hard to see much value in storing this collection in a separate table.

Learning to not hate SQL arrays

For a long time, we thought arrays were a kind of weird and warty thing to add to the relational model, but recently we've come to realize that this view was overly closed-minded. Indeed, we might choose to view SQL `ARRAY` types as a generalization of `VARCHAR` and `VARBINARY` to generic "element" types. And from this point of view, SQL arrays look quite attractive, at least for certain problems. If we're comfortable mapping `byte[]` to `VARBINARY(255)`, why would we shy away from mapping `DayOfWeek[]` to `TINYINT ARRAY[7]`?

Unfortunately, JPA doesn't define a standard way to map SQL arrays, but here's how we can do it in Hibernate:

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @Array(length=7)
    DayOfWeek[] daysOfWeek;  // stored as a SQL ARRAY type
    ...
}
```

The `@Array` annotation is optional—it lets us specify an upper bound on the length of the `ARRAY` column. By writing `@Array(length=7)` here, we specified that DDL should be generated with the column type `TINYINT ARRAY[7]`.

Just for fun, we used an enumerated type in the code above, but the array element time may be almost any basic type. For example, the Java array types `String[]`, `UUID[]`, `double[]`, `BigDecimal[]`, `LocalDate[]`, and `OffsetDateTime[]` are all allowed, mapping to the SQL types `VARCHAR(n) ARRAY`, `UUID ARRAY`, `FLOAT(53) ARRAY`, `NUMERIC(p,s) ARRAY`, `DATE ARRAY`, and `TIMESTAMP(p) WITH TIME ZONE ARRAY`, respectively.

> Now for the gotcha: not every database has a SQL `ARRAY` type, and some that *do* have an `ARRAY` type don't allow it to be used as a column type.
>
> In particular, neither DB2 nor SQL Server have array-typed columns. On these databases, Hibernate falls back to something much worse: it uses Java serialization to encode the array to a binary representation, and stores the binary stream in a `VARBINARY` column. Quite clearly, this is terrible. You can ask Hibernate to do something *slightly* less terrible by

> annotating the attribute `@JdbcTypeCode(SqlTypes.JSON)`, so that the array is serialized to JSON instead of binary format. But at this point it's better to just admit defeat and use an `@ElementCollection` instead.

Alternatively, we could store this array or list in a separate table.

3.23. Collections mapped to a separate table

JPA *does* define a standard way to map a collection to an auxiliary table: the `@ElementCollection` annotation.

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @ElementCollection
    DayOfWeek[] daysOfWeek;  // stored in a dedicated table
    ...
}
```

Actually, we shouldn't use an array here, since array types can't be proxied, and so the JPA specification doesn't even say they're supported. Instead, we should use `Set`, `List`, or `Map`.

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @ElementCollection
    List<DayOfWeek> daysOfWeek;  // stored in a dedicated table
    ...
}
```

Here, each collection element is stored as a separate row of the auxiliary table. By default, this table has the following definition:

```
create table Event_daysOfWeek (
    Event_id bigint not null,
    daysOfWeek tinyint check (daysOfWeek between 0 and 6),
    daysOfWeek_ORDER integer not null,
    primary key (Event_id, daysOfWeek_ORDER)
)
```

Which is fine, but it's still a mapping we prefer to avoid.

> `@ElementCollection` is one of our least-favorite features of JPA. Even the name of the annotation is bad.
>
> The code above results in a table with three columns:
>
> - a foreign key of the `Event` table,
> - a `TINYINT` encoding the `enum`, and
> - an `INTEGER` encoding the ordering of elements in the array.
>
> Instead of a surrogate primary key, it has a composite key comprising the foreign key of `Event` and the order column.
>
> When—inevitably—we find that we need to add a fourth column to that table, our Java code must change completely. Most likely, we'll realize that we need to add a separate entity after all. So this mapping isn't very robust in the face of minor changes to our data model.

There's much more we could say about "element collections", but we won't say it, because we don't want to hand you the gun you'll shoot your foot with.

3.24. Summary of annotations

Let's pause to remember the annotations we've met so far.

Table 3.8: Declaring entities and embeddable types

Annotation	Purpose	JPA-standard
`@Entity`	Declare an entity class	✔
`@MappedSuperclass`	Declare a non-entity class with mapped attributes inherited by an entity	✔
`@Embeddable`	Declare an embeddable type	✔
`@IdClass`	Declare the identifier class for an entity with multiple `@Id` attributes	✔

Table 3.9: Declaring basic and embedded attributes

Annotation	Purpose		JPA-standard
`@Id`	Declare a basic-typed identifier attribute		✔
`@Version`	Declare a version attribute		✔
`@Basic`	Declare a basic attribute	Default	✔
`@EmbeddedId`	Declare an embeddable-typed identifier attribute		✔
`@Embedded`	Declare an embeddable-typed attribute	Inferred	✔
`@Enumerated`	Declare an `enum`-typed attribute and specify how it is encoded	Inferred	✔
`@Array`	Declare that an attribute maps to a SQL `ARRAY`, and specify the length	Inferred	✖
`@ElementCollection`	Declare that a collection is mapped to a dedicated table		✔

Table 3.10: Converters and compositional basic types

Annotation	Purpose	JPA-standard
`@Converter`	Register an `AttributeConverter`	✔
`@Convert`	Apply a converter to an attribute	✔
`@JavaType`	Explicitly specify an implementation of `JavaType` for a basic attribute	✖
`@JdbcType`	Explicitly specify an implementation of `JdbcType` for a basic attribute	✖
`@JdbcTypeCode`	Explicitly specify a JDBC type code used to determine the `JdbcType` for a basic attribute	✖
`@JavaTypeRegistration`	Register a `JavaType` for a given Java type	✖
`@JdbcTypeRegistration`	Register a `JdbcType` for a given JDBC type code	✖

Table 3.11: System-generated identifiers

Annotation	Purpose	JPA-standard
`@GeneratedValue`	Specify that an identifier is system-generated	✔
`@SequenceGenerator`	Define an id generator backed by a database sequence	✔
`@TableGenerator`	Define an id generated backed by a database table	✔
`@IdGeneratorType`	Declare an annotation that associates a custom `Generator` with each `@Id` attribute it annotates	✖

Annotation	Purpose	JPA-standard
`@ValueGenerationType`	Declare an annotation that associates a custom `Generator` with each `@Basic` attribute it annotates	✖

Table 3.12: Declaring entity associations

Annotation	Purpose	JPA-standard
`@ManyToOne`	Declare the single-valued side of a many-to-one association (the owning side)	✔
`@OneToMany`	Declare the many-valued side of a many-to-one association (the unowned side)	✔
`@ManyToMany`	Declare either side of a many-to-many association	✔
`@OneToOne`	Declare either side of a one-to-one association	✔
`@MapsId`	Declare that the owning side of a `@OneToOne` association maps the primary key column	✔

Phew! That's already a lot of annotations, and we have not even started with the annotations for O/R mapping!

3.25. `equals()` and `hashCode()`

Entity classes should override `equals()` and `hashCode()`, especially when associations are represented as sets.

People new to Hibernate or JPA are often confused by exactly which fields should be included in the `hashCode()`. And people with more experience often argue quite religiously that one or another approach is the only right way. The truth is, there's no unique right way to do it, but there are some constraints. So please keep the following principles in mind:

- You should not include a mutable field in the hashcode, since that would require rehashing every collection containing the entity whenever the field is mutated.
- It's not completely wrong to include a generated identifier (surrogate key) in the hashcode, but since the identifier is not generated until the entity instance is made persistent, you must take great care to not add it to any hashed collection before the identifier is generated. We therefore advise against including any database-generated field in the hashcode.

It's OK to include any immutable, non-generated field in the hashcode.

> We therefore recommend identifying a natural key for each entity, that is, a combination of fields that uniquely identifies an instance of the entity, from the perspective of the data model of the program. The natural key should correspond to a unique constraint on the database, and to the fields which are included in `equals()` and `hashCode()`.

In this example, the `equals()` and `hashCode()` methods agree with the `@NaturalId` annotation:

```
@Entity
class Book {

    @Id @GeneratedValue
    Long id;

    @NaturalId
    @Basic(optional=false)
    String isbn;

    String getIsbn() {
        return isbn;
    }

    ...

    @Override
    public boolean equals(Object other) {
        return other instanceof Book                  // check type with instanceof, not getClass()
            && ((Book) other).getIsbn().equals(isbn); // compare natural ids
    }
    @Override
```

```java
    public int hashCode() {
        return isbn.hashCode();  // hashcode based on the natural id
    }
}
```

That said, an implementation of `equals()` and `hashCode()` based on the generated identifier of the entity can work *if you're careful*.

> Your implementation of `equals()` must be written to accommodate the possibility that the object passed to the `equals()` might be a proxy. Therefore, you should use `instanceof`, not `getClass()` to check the type of the argument, and should access fields of the passed entity via its accessor methods.

Chapter 4. Object/relational mapping

Given a domain model—that is, a collection of entity classes decorated with all the fancy annotations we just met in the previous chapter—Hibernate will happily go away and infer a complete relational schema, and even export it to your database if you ask politely.

The resulting schema will be entirely sane and reasonable, though if you look closely, you'll find some flaws. For example, by default, every `VARCHAR` column will have the same length, `VARCHAR(255)`.

But the process I just described—which we call *top down* mapping—simply doesn't fit the most common scenario for the use of O/R mapping. It's only rarely that the Java classes precede the relational schema. Usually, *we already have a relational schema*, and we're constructing our domain model around the schema. This is called *bottom up* mapping.

> Developers often refer to a pre-existing relational database as "legacy" data. This tends to conjure images of bad old "legacy apps" written in COBOL or something. But legacy data is valuable, and learning to work with it is important.

Especially when mapping bottom up, we often need to customize the inferred object/relational mappings. This is a somewhat tedious topic, and so we don't want to spend too many words on it. Instead, we'll quickly skim the most important mapping annotations.

Hibernate SQL case convention

Computers have had lowercase letters for rather a long time now. Most developers learned long ago that text written in MixedCase, camelCase, or even snake_case is easier to read than text written in SHOUTYCASE. This is just as true of SQL as it is of any other language.

Therefore, for over twenty years, the convention on the Hibernate project has been that:

- query language identifiers are written in `lowercase`,
- table names are written in `MixedCase`, and
- column names are written in `camelCase`.

That is to say, we simply adopted Java's excellent conventions and applied them to SQL.

Now, there's no way we can force you to follow this convention, even if we wished to. Hell, you can easily write a `PhysicalNamingStrategy` which makes table and column names ALL UGLY AND SHOUTY LIKE THIS IF YOU PREFER. But, *by default*, it's the convention Hibernate follows, and it's frankly a pretty reasonable one.

4.1. Mapping entity inheritance hierarchies

In Entity class inheritance we saw that entity classes may exist within an inheritance hierarchy. There's three basic strategies for mapping an entity hierarchy to relational tables. Let's put them in a table, so we can more easily compare the points of difference between them.

Table 4.1: Entity inheritance mapping strategies

Strategy	Mapping	Polymorphic queries	Constraints	Normalization	When to use it
`SINGLE_TABLE`	Map every class in the hierarchy to the same table, and uses the value of a *discriminator column* to determine which concrete class each row represents.	To retrieve instances of a given class, we only need to query the one table.	Attributes declared by subclasses map to columns without `NOT NULL` constraints, and so their non-nullability is enforced via a `CHECK` constraint. Any association may have a `FOREIGN KEY` constraint. 🤓	Subclass data is denormalized. 🧐	Works well when subclasses declare few or no additional attributes.

Strategy	Mapping	Polymorphic queries	Constraints	Normalization	When to use it
`JOINED`	Map every class in the hierarchy to a separate table, but each table only maps the attributes declared by the class itself. Optionally, a discriminator column may be used.	To retrieve instances of a given class, we must `JOIN` the table mapped by the class with: • all tables mapped by its superclasses and • all tables mapped by its subclasses.	Any attribute may map to a column with a `NOT NULL` constraint. 🤓 Any association may have a `FOREIGN KEY` constraint. 🤓	The tables are normalized. 🤓	The best option when we care a lot about constraints and normalization.
`TABLE_PER_CLASS`	Map every concrete class in the hierarchy to a separate table, but denormalize all inherited attributes into the table.	To retrieve instances of a given class, we must take a `UNION` over the table mapped by the class and the tables mapped by its subclasses.	Associations targeting a superclass cannot have a corresponding `FOREIGN KEY` constraint in the database. 💀 💀 Any attribute may map to a column with a `NOT NULL` constraint. 🤓	Superclass data is denormalized. 🧐	Not very popular. From a certain point of view, competes with `@MappedSuperclass`.

The three mapping strategies are enumerated by `InheritanceType`. We specify an inheritance mapping strategy using the `@Inheritance` annotation.

For mappings with a *discriminator column*, we should:

- specify the discriminator column name and type by annotating the root entity `@DiscriminatorColumn`, and
- specify the values of this discriminator by annotating each entity in the hierarchy `@DiscriminatorValue`.

For single table inheritance we always need a discriminator:

```
@Entity
@DiscriminatorColumn(discriminatorType=CHAR, name="kind")
@DiscriminatorValue('P')
class Person { ... }

@Entity
@DiscriminatorValue('A')
class Author { ... }
```

We don't need to explicitly specify `@Inheritance(strategy=SINGLE_TABLE)`, since that's the default.

For `JOINED` inheritance we don't need a discriminator:

```
@Entity
@Inheritance(strategy=JOINED)
class Person { ... }

@Entity
class Author { ... }
```

> However, we can add a discriminator column if we like, and in that case the generated SQL for polymorphic queries will be slightly simpler.

Similarly, for `TABLE_PER_CLASS` inheritance we have:

```
@Entity
@Inheritance(strategy=TABLE_PER_CLASS)
```

```
class Person { ... }

@Entity
class Author { ... }
```

Hibernate doesn't allow discriminator columns for `TABLE_PER_CLASS` inheritance mappings, since they would make no sense, and offer no advantage.

Notice that in this last case, a polymorphic association like:

```
@ManyToOne Person person;
```

is a bad idea, since it's impossible to create a foreign key constraint that targets both mapped tables.

It's quite easy to overuse inheritance. We've occasionally seen extreme cases where a `JOINED` inheritance hierarchy includes *hundreds* of entities, spanning hundreds of tables. Efficiently querying such a hierarchy is an almost impossible task for Hibernate. Fortunately, a `JOINED` inheritance relationship can always be remodeled as a one-to-one association, allowing much more efficient queries. In general, a single entity inheritance hierarchy should never span more than a few tables, including secondary tables.

4.2. Mapping to tables

The following annotations specify exactly how elements of the domain model map to tables of the relational model:

Table 4.2: Annotations for mapping tables

Annotation	Purpose
`@Table`	Map an entity class to its primary table
`@SecondaryTable`	Define a secondary table for an entity class
`@JoinTable`	Map a many-to-many or many-to-one association to its association table
`@CollectionTable`	Map an `@ElementCollection` to its table

The first two annotations are used to map an entity to its *primary table* and, optionally, one or more *secondary tables*.

4.3. Mapping entities to tables

By default, an entity maps to a single table, which may be specified using `@Table`:

```
@Entity
@Table(name="People")
class Person { ... }
```

However, the `@SecondaryTable` annotation allows us to spread its attributes across multiple *secondary tables*.

```
@Entity
@Table(name="Books")
@SecondaryTable(name="Editions")
class Book { ... }
```

The `@Table` annotation can do more than just specify a name:

Table 4.3: `@Table` annotation members

Annotation member	Purpose
`name`	The name of the mapped table
`schema` 💀	The schema to which the table belongs

Annotation member	Purpose
`catalog` 💀	The catalog to which the table belongs
`uniqueConstraints`	One or more `@UniqueConstraint` annotations declaring multi-column unique constraints
`indexes`	One or more `@Index` annotations each declaring an index
`check`	One or more `@CheckConstraint` annotations declaring multi-column check constraints
`comment`	A DDL comment

> It only makes sense to explicitly specify the `schema` in annotations if the domain model is spread across multiple schemas.
>
> Otherwise, it's a bad idea to hardcode the schema (or catalog) in a `@Table` annotation. Instead:
>
> - set the configuration property `hibernate.default_schema` (or `hibernate.default_catalog`), or
> - simply specify the schema in the JDBC connection URL.

The `@SecondaryTable` annotation is even more interesting:

Table 4.4: `@SecondaryTable` annotation members

Annotation member	Purpose
`name`	The name of the mapped table
`schema` 💀	The schema to which the table belongs
`catalog` 💀	The catalog to which the table belongs
`uniqueConstraints`	One or more `@UniqueConstraint` annotations declaring multi-column unique constraints
`indexes`	One or more `@Index` annotations each declaring an index
`pkJoinColumns`	One or more `@PrimaryKeyJoinColumn` annotations, specifying primary key column mappings
`foreignKey`	A `@ForeignKey` annotation specifying the name of the `FOREIGN KEY` constraint on the `@PrimaryKeyJoinColumns`
`check`	One or more `@CheckConstraint` annotations declaring multi-column check constraints
`comment`	A DDL comment

> Using `@SecondaryTable` on a subclass in a `SINGLE_TABLE` entity inheritance hierarchy gives us a sort of mix of `SINGLE_TABLE` with `JOINED` inheritance.

4.4. Mapping associations to tables

The `@JoinTable` annotation specifies an *association table*, that is, a table holding foreign keys of both associated entities. This annotation is usually used with `@ManyToMany` associations:

```
@Entity
class Book {
    ...

    @ManyToMany
    @JoinTable(name="BooksAuthors")
    Set<Author> authors;

    ...
}
```

But it's even possible to use it to map a `@ManyToOne` or `@OneToOne` association to an association table.

```
@Entity
class Book {
    ...

    @ManyToOne(fetch=LAZY)
    @JoinTable(name="BookPublisher")
    Publisher publisher;

    ...
}
```

Here, there should be a `UNIQUE` constraint on one of the columns of the association table.

```
@Entity
class Author {
    ...

    @OneToOne(optional=false, fetch=LAZY)
    @JoinTable(name="AuthorPerson")
    Person author;

    ...
}
```

Here, there should be a `UNIQUE` constraint on *both* columns of the association table.

Table 4.5: `@JoinTable` annotation members

Annotation member	Purpose
`name`	The name of the mapped association table
`schema` 💀	The schema to which the table belongs
`catalog` 💀	The catalog to which the table belongs
`uniqueConstraints`	One or more `@UniqueConstraint` annotations declaring multi-column unique constraints
`indexes`	One or more `@Index` annotations each declaring an index
`joinColumns`	One or more `@JoinColumn` annotations, specifying foreign key column mappings to the table of the owning side
`inverseJoinColumns`	One or more `@JoinColumn` annotations, specifying foreign key column mappings to the table of the unowned side
`foreignKey`	A `@ForeignKey` annotation specifying the name of the `FOREIGN KEY` constraint on the `joinColumns`s
`inverseForeignKey`	A `@ForeignKey` annotation specifying the name of the `FOREIGN KEY` constraint on the `inverseJoinColumns`s
`check`	One or more `@CheckConstraint` annotations declaring multi-column check constraints
`comment`	A DDL comment

To better understand these annotations, we must first discuss column mappings in general.

4.5. Mapping to columns

These annotations specify how elements of the domain model map to columns of tables in the relational model:

Table 4.6: Annotations for mapping columns

Annotation	Purpose
`@Column`	Map an attribute to a column
`@JoinColumn`	Map an association to a foreign key column

Annotation	Purpose
`@PrimaryKeyJoinColumn`	Map the primary key used to join a secondary table with its primary, or a subclass table in `JOINED` inheritance with its root class table
`@OrderColumn`	Specifies a column that should be used to maintain the order of a `List`.
`@MapKeyColumn`	Specified a column that should be used to persist the keys of a `Map`.

We'll come back to the last two annotations much later, in Ordered and sorted collections and map keys.

We use the `@Column` annotation to map basic attributes.

4.6. Mapping basic attributes to columns

The `@Column` annotation is not only useful for specifying the column name.

Table 4.7: `@Column` annotation members

Annotation member	Purpose
`name`	The name of the mapped column
`table`	The name of the table to which this column belongs
`length`	The length of a `VARCHAR`, `CHAR`, or `VARBINARY` column type
`precision`	The decimal digits of precision of a `FLOAT`, `DECIMAL`, or `NUMERIC` type
`scale`	The scale of a `DECIMAL` or `NUMERIC` column type, the digits of precision that occur to the right of the decimal point
`secondPrecision`	The digits of precision occurring to the right of the decimal point in the seconds field of a `TIME`, or `TIMESTAMP` column type
`unique`	Whether the column has a `UNIQUE` constraint
`nullable`	Whether the column has a `NOT NULL` constraint
`insertable`	Whether the column should appear in generated SQL `INSERT` statements
`updatable`	Whether the column should appear in generated SQL `UPDATE` statements
`columnDefinition` 💀	A DDL fragment that should be used to declare the column
`check`	One or more `@CheckConstraint` annotations declaring single-column check constraints
`comment`	A DDL comment

> We no longer recommend the use of `columnDefinition` since it results in unportable DDL. Hibernate has much better ways to customize the generated DDL using techniques that result in portable behavior across different databases.

Here we see four different ways to use the `@Column` annotation:

```
@Entity
@Table(name="Books")
@SecondaryTable(name="Editions")
class Book {
    @Id @GeneratedValue
    @Column(name="bookId") // customize column name
    Long id;

    @Column(length=100, nullable=false) // declare column as VARCHAR(100) NOT NULL
    String title;

    @Column(length=17, unique=true, nullable=false) // declare column as VARCHAR(17) NOT NULL UNIQUE
```

```
    String isbn;

    @Column(table="Editions", updatable=false) // column belongs to the secondary table, and is never updated
    int edition;
}
```

We don't use `@Column` to map associations.

4.7. Mapping associations to foreign key columns

The `@JoinColumn` annotation is used to customize a foreign key column.

Table 4.8: `@JoinColumn` annotation members

Annotation member	Purpose
`name`	The name of the mapped foreign key column
`table`	The name of the table to which this column belongs
`referencedColumnName`	The name of the column to which the mapped foreign key column refers
`unique`	Whether the column has a `UNIQUE` constraint
`nullable`	Whether the column has a `NOT NULL` constraint
`insertable`	Whether the column should appear in generated SQL `INSERT` statements
`updatable`	Whether the column should appear in generated SQL `UPDATE` statements
`columnDefinition` 💀	A DDL fragment that should be used to declare the column
`foreignKey`	A `@ForeignKey` annotation specifying the name of the `FOREIGN KEY` constraint
`check`	One or more `@CheckConstraint` annotations declaring single-column check constraints
`comment`	A DDL comment

A foreign key column doesn't necessarily have to refer to the primary key of the referenced table. It's quite acceptable for the foreign key to refer to any other unique key of the referenced entity, even to a unique key of a secondary table.

Here we see how to use `@JoinColumn` to define a `@ManyToOne` association mapping a foreign key column which refers to the `@NaturalId` of `Book`:

```
@Entity
@Table(name="Items")
class Item {
    ...

    @ManyToOne(optional=false)  // implies nullable=false
    @JoinColumn(name = "bookIsbn", referencedColumnName = "isbn",  // a reference to a non-PK column
                foreignKey = @ForeignKey(name="ItemsToBooksBySsn")) // supply a name for the FK constraint
    Book book;

    ...
}
```

In case this is confusing:

- `bookIsbn` is the name of the foreign key column in the `Items` table,
- it refers to a unique key `isbn` in the `Books` table, and
- it has a foreign key constraint named `ItemsToBooksBySsn`.

Note that the `foreignKey` member is completely optional and only affects DDL generation.

> If you don't supply an explicit name using `@ForeignKey`, Hibernate will generate a quite ugly name. The reason for this is

> that the maximum length of foreign key names on some databases is extremely constrained, and we need to avoid collisions. To be fair, this is perfectly fine if you're only using the generated DDL for testing. You can customize the generated constraint names by writing your own `ImplicitNamingStrategy`.

For composite foreign keys we might have multiple `@JoinColumn` annotations:

```
@Entity
@Table(name="Items")
class Item {
    ...

    @ManyToOne(optional=false)
    @JoinColumn(name = "bookIsbn", referencedColumnName = "isbn")
    @JoinColumn(name = "bookPrinting", referencedColumnName = "printing")
    Book book;

    ...
}
```

If we need to specify the `@ForeignKey`, this starts to get a bit messy:

```
@Entity
@Table(name="Items")
class Item {
    ...

    @ManyToOne(optional=false)
    @JoinColumns(value = {@JoinColumn(name = "bookIsbn", referencedColumnName = "isbn"),
                          @JoinColumn(name = "bookPrinting", referencedColumnName = "printing")},
                 foreignKey = @ForeignKey(name="ItemsToBooksBySsn"))
    Book book;

    ...
}
```

For associations mapped to a `@JoinTable`, fetching the association requires two joins, and so we must declare the `@JoinColumns` inside the `@JoinTable` annotation:

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany
    @JoinTable(joinColumns=@JoinColumn(name="bookId"),
               inverseJoinColumns=@joinColumn(name="authorId"),
               foreignKey=@ForeignKey(name="BooksToAuthors"))
    Set<Author> authors;

    ...
}
```

Again, the `foreignKey` member is optional.

> For mapping a `@OneToOne` association to a primary key with `@MapsId`, Hibernate lets us use either `@JoinColumn` or `@PrimaryKeyJoinColumn`.
>
> ```
> @Entity
> class Author {
>     @Id
>     Long id;
>
>     @OneToOne(optional=false, fetch=LAZY)
>     @MapsId
>     @PrimaryKeyJoinColumn(name="personId")
>     Person author;
> ```

```
    ...
}
```

Arguably, the use of `@PrimaryKeyJoinColumn` is clearer.

4.8. Mapping primary key joins between tables

The `@PrimaryKeyJoinColumn` is a special-purpose annotation for mapping:

- the primary key column of a `@SecondaryTable`—which is also a foreign key referencing the primary table, or
- the primary key column of the primary table mapped by a subclass in a `JOINED` inheritance hierarchy—which is also a foreign key referencing the primary table mapped by the root entity.

Table 4.9: `@PrimaryKeyJoinColumn` annotation members

Annotation member	Purpose
`name`	The name of the mapped foreign key column
`referencedColumnName`	The name of the column to which the mapped foreign key column refers
`columnDefinition` 💀	A DDL fragment that should be used to declare the column
`foreignKey`	A `@ForeignKey` annotation specifying the name of the `FOREIGN KEY` constraint

When mapping a subclass table primary key, we place the `@PrimaryKeyJoinColumn` annotation on the entity class:

```
@Entity
@Table(name="People")
@Inheritance(strategy=JOINED)
class Person { ... }

@Entity
@Table(name="Authors")
@PrimaryKeyJoinColumn(name="personId") // the primary key of the Authors table
class Author { ... }
```

But to map a secondary table primary key, the `@PrimaryKeyJoinColumn` annotation must occur inside the `@SecondaryTable` annotation:

```
@Entity
@Table(name="Books")
@SecondaryTable(name="Editions",
                pkJoinColumns = @PrimaryKeyJoinColumn(name="bookId")) // the primary key of the Editions table
class Book {
    @Id @GeneratedValue
    @Column(name="bookId") // the name of the primary key of the Books table
    Long id;

    ...
}
```

4.9. Column lengths and adaptive column types

Hibernate automatically adjusts the column type used in generated DDL based on the column length specified by the `@Column` annotation. So we don't usually need to explicitly specify that a column should be of type `TEXT` or `CLOB`—or worry about the parade of `TINYTEXT`, `MEDIUMTEXT`, `TEXT`, `LONGTEXT` types on MySQL—because Hibernate automatically selects one of those types if required to accommodate a string of the `length` we specify.

The constant values defined in the class `Length` are very helpful here:

Table 4.10: Predefined column lengths

Constant	Value	Description
`DEFAULT`	255	The default length of a `VARCHAR` or `VARBINARY` column when none is explicitly specified

Constant	Value	Description
`LONG`	32600	The largest column length for a `VARCHAR` or `VARBINARY` that is allowed on every database Hibernate supports
`LONG16`	32767	The maximum length that can be represented using 16 bits (but this length is too large for a `VARCHAR` or `VARBINARY` column on for some database)
`LONG32`	2147483647	The maximum length for a Java string

We can use these constants in the `@Column` annotation:

```
@Column(length=LONG)
String text;

@Column(length=LONG32)
byte[] binaryData;
```

This is usually all you need to do to make use of large object types in Hibernate.

4.10. LOBs

JPA provides a `@Lob` annotation which specifies that a field should be persisted as a `BLOB` or `CLOB`.

Semantics of the `@Lob` annotation

What the spec actually says is that the field should be persisted

> *…as a large object to a database-supported large object type.*

It's quite unclear what this means, and the spec goes on to say that

> *…the treatment of the* `Lob` *annotation is provider-dependent…*

which doesn't help much.

Hibernate interprets this annotation in what we think is the most reasonable way. In Hibernate, an attribute annotated `@Lob` will be written to JDBC using the `setClob()` or `setBlob()` method of `PreparedStatement`, and will be read from JDBC using the `getClob()` or `getBlob()` method of `ResultSet`.

Now, the use of these JDBC methods is usually unnecessary! JDBC drivers are perfectly capable of converting between `String` and `CLOB` or between `byte[]` and `BLOB`. So unless you specifically need to use these JDBC LOB APIs, you *don't* need the `@Lob` annotation.

Instead, as we just saw in Column lengths and adaptive column types, all you need is to specify a large enough column `length` to accommodate the data you plan to write to that column.

You should usually write this:

```
@Column(length=LONG32) // good, correct column type inferred
String text;
```

instead of this:

```
@Lob // almost always unnecessary
String text;
```

This is particularly true for PostgreSQL.

Unfortunately, the driver for PostgreSQL doesn't allow `BYTEA` or `TEXT` columns to be read via the JDBC LOB APIs.

This limitation of the Postgres driver has resulted in a whole cottage industry of bloggers and stackoverflow question-answerers recommending convoluted ways to hack the Hibernate `Dialect` for Postgres to allow an attribute annotated `@Lob` to be written using `setString()` and read using `getString()`.

But simply removing the `@Lob` annotation has exactly the same effect.

Conclusion:

- on PostgreSQL, `@Lob` always means the `OID` type,
- `@Lob` should never be used to map columns of type `BYTEA` or `TEXT`, and
- please don't believe everything you read on stackoverflow.

Finally, as an alternative, Hibernate lets you declare an attribute of type `java.sql.Blob` or `java.sql.Clob`.

```
@Entity
class Book {
    ...
    Clob text;
    Blob coverArt;
    ....
}
```

The advantage is that a `java.sql.Clob` or `java.sql.Blob` can in principle index up to 2^{63} characters or bytes, much more data than you can fit in a Java `String` or `byte[]` array (or in your computer).

To assign a value to these fields, we'll need to use a `LobHelper`. We can get one from the `Session`:

```
LobHelper helper = session.getLobHelper();
book.text = helper.createClob(text);
book.coverArt = helper.createBlob(image);
```

In principle, the `Blob` and `Clob` objects provide efficient ways to read or stream LOB data from the server.

```
Book book = session.find(Book.class, bookId);
String text = book.text.getSubString(1, textLength);
InputStream bytes = book.coverArt.getBinaryStream();
```

Of course, the behavior here depends very much on the JDBC driver, and so we really can't promise that this is a sensible thing to do on your database.

4.11. Mapping embeddable types to UDTs or to JSON

There's a couple of alternative ways to represent an embeddable type on the database side.

Embeddables as UDTs

First, a really nice option, at least in the case of Java record types, and for databases which support *user-defined types* (UDTs), is to define a UDT which represents the record type. Hibernate 6 makes this really easy. Just annotate the record type, or the attribute which holds a reference to it, with the new `@Struct` annotation:

```
@Embeddable
@Struct(name="PersonName")
record Name(String firstName, String middleName, String lastName) {}
```

```
@Entity
class Person {
    ...
    Name name;
    ...
}
```

This results in the following UDT:

```
create type PersonName as (firstName varchar(255), middleName varchar(255), lastName varchar(255))
```

And the name column of the Author table will have the type PersonName.

Embeddables to JSON

A second option that's available is to map the embeddable type to a JSON (or JSONB) column. Now, this isn't something we would exactly *recommend* if you're defining a data model from scratch, but it's at least useful for mapping pre-existing tables with JSON-typed columns. Since embeddable types are nestable, we can map some JSON formats this way, and even query JSON properties using HQL.

> At this time, JSON arrays are not supported!

To map an attribute of embeddable type to JSON, we must annotate the attribute @JdbcTypeCode(SqlTypes.JSON), instead of annotating the embeddable type. But the embeddable type Name should still be annotated @Embeddable if we want to query its attributes using HQL.

```
@Embeddable
record Name(String firstName, String middleName, String lastName) {}
```

```
@Entity
class Person {
    ...
    @JdbcTypeCode(SqlTypes.JSON)
    Name name;
    ...
}
```

We also need to add Jackson or an implementation of JSONB—for example, Yasson—to our runtime classpath. To use Jackson 2 we could add this line to our Gradle build:

```
runtimeOnly 'com.fasterxml.jackson.core:jackson-databind:{jacksonVersion}'
```

To use Jackson 3 we could add this line to our Gradle build:

```
runtimeOnly 'tools.jackson.core:jackson-databind:{jackson3Version}'
```

Now the name column of the Author table will have the type jsonb, and Hibernate will automatically use Jackson to serialize a Name to and from JSON format.

4.12. Summary of SQL column type mappings

So, as we've seen, there are quite a few annotations that affect the mapping of Java types to SQL column types in DDL. Here we summarize the ones we've just seen in the second half of this chapter, along with some we already mentioned in earlier chapters.

Table 4.11: Annotations for mapping SQL column types

Annotation	Interpretation
@Enumerated, @EnumeratedValue	Specify how an enum type should be persisted
@Nationalized	Use a nationalized character type: NCHAR, NVARCHAR, or NCLOB
@Lob 💀	Use JDBC LOB APIs to read and write the annotated attribute
@Array	Map a collection to a SQL ARRAY type of the specified length
@Struct	Map an embeddable to a SQL UDT with the given name
@TimeZoneStorage	Specify how the time zone information should be persisted
@JdbcType or @JdbcTypeCode	Use an implementation of JdbcType to map an arbitrary SQL type
@Collate	Specify a collation for a column

In addition, there are some configuration properties which have a *global* effect on how basic types map to SQL column types:

Table 4.12: Type mapping settings

Configuration property name	Purpose
`hibernate.use_nationalized_character_data`	Enable use of nationalized character types by default
`hibernate.type.preferred_boolean_jdbc_type`	Specify the default SQL column type for storing a `boolean`
`hibernate.type.preferred_uuid_jdbc_type`	Specify the default SQL column type for storing a `UUID`
`hibernate.type.preferred_duration_jdbc_type`	Specify the default SQL column type for storing a `Duration`
`hibernate.type.preferred_instant_jdbc_type`	Specify the default SQL column type for storing an `Instant`
`hibernate.timezone.default_storage`	Specify the default strategy for storing time zone information
`hibernate.type.prefer_native_enum_types`	Use named enum types on PostgreSQL and Oracle

Earlier, we saw how to use these settings to control the default mappings for `Instant` and `Duration`.

> These are *global* settings and thus quite clumsy. We recommend against messing with any of these settings unless you have a really good reason for it.

There's one more topic we would like to cover in this chapter.

4.13. Mapping to formulas

Hibernate lets us map an attribute of an entity to a SQL formula involving columns of the mapped table. Thus, the attribute is a sort of "derived" value.

Table 4.13: Annotations for mapping formulas

Annotation	Purpose
`@Formula`	Map an attribute to a SQL formula
`@JoinFormula`	Map an association to a SQL formula
`@DiscriminatorFormula`	Use a SQL formula as the discriminator in single table inheritance.

For example:

```
@Entity
class Order {
    ...
    @Column(name = "sub_total", scale=2, precision=8)
    BigDecimal subTotal;

    @Column(name = "tax", scale=4, precision=4)
    BigDecimal taxRate;

    @Formula("sub_total * (1.0 + tax)")
    BigDecimal totalWithTax;
    ...
}
```

The formula is evaluated every time the entity is read from the database.

4.14. Derived Identity

An entity has a *derived identity* if it inherits part of its primary key from an associated "parent" entity. We've already met a kind of degenerate case of *derived identity* when we talked about one-to-one associations with a shared primary key.

But a `@ManyToOne` association may also form part of a derived identity. That is to say, there could be a foreign key column or columns included as part of the composite primary key. There's three different ways to represent this situation on the Java side of things:

- using `@IdClass` without `@MapsId`,
- using `@IdClass` with `@MapsId`, or
- using `@EmbeddedId` with `@MapsId`.

Let's suppose we have a `Parent` entity class defined as follows:

```
@Entity
class Parent {
    @Id
    Long parentId;

    ...
}
```

The `parentId` field holds the primary key of the `Parent` table, which will also form part of the composite primary key of every `Child` belonging to the `Parent`.

First way

In the first, slightly simpler approach, we define an `@IdClass` to represent the primary key of `Child`:

```
class DerivedId {
    Long parent;
    String childId;

    // constructors, equals, hashcode, etc
    ...
}
```

And a `Child` entity class with a `@ManyToOne` association annotated `@Id`:

```
@Entity
@IdClass(DerivedId.class)
class Child {
    @Id
    String childId;

    @Id @ManyToOne
    @JoinColumn(name="parentId")
    Parent parent;

    ...
}
```

Then the primary key of the `Child` table comprises the columns `(childId,parentId)`.

Second way

This is fine, but sometimes it's nice to have a field for each element of the primary key. We may use the `@MapsId` annotation we met earlier:

```
@Entity
@IdClass(DerivedId.class)
class Child {
    @Id
    Long parentId;
    @Id
    String childId;

    @ManyToOne
    @MapsId(Child_.PARENT_ID) // typesafe reference to Child.parentId
    @JoinColumn(name="parentId")
    Parent parent;

    ...
}
```

We're using the approach we saw previously to refer to the `parentId` property of `Child` in a typesafe way.

Note that we must place column mapping information on the association annotated `@MapsId`, not on the `@Id` field.

We must slightly modify our `@IdClass` so that field names align:

```
class DerivedId {
    Long parentId;
    String childId;

    // constructors, equals, hashcode, etc
    ...
}
```

Third way

The third alternative is to redefine our `@IdClass` as an `@Embeddable`. We don't actually need to change the `DerivedId` class, but we do need to add the annotation.

```
@Embeddable
class DerivedId {
    Long parentId;
    String childId;

    // constructors, equals, hashcode, etc
    ...
}
```

Then we may use `@EmbeddedId` in `Child`:

```
@Entity
class Child {
    @EmbeddedId
    DerivedId id;

    @ManyToOne
    @MapsId(DerivedId_.PARENT_ID) // typesafe reference to DerivedId.parentId
    @JoinColumn(name="parentId")
    Parent parent;

    ...
}
```

The choice between `@IdClass` and `@EmbeddedId` boils down to taste. The `@EmbeddedId` is perhaps a little DRYer.

4.15. Adding constraints

Database constraints are important. Even if you're sure that your program has no bugs 🧐, it's probably not the only program with access to the database. Constraints help ensure that different programs (and human administrators) play nicely with each other.

Hibernate adds certain constraints to generated DDL automatically: primary key constraints, foreign key constraints, and some unique constraints. But it's common to need to:

- add additional unique constraints,
- add check constraints, or
- customize the name of a foreign key constraint.

We've already seen how to use `@ForeignKey` to specify the name of a foreign key constraint.

There are two ways to add a unique constraint to a table:

- using `@Column(unique=true)` to indicate a single-column unique key, or
- using the `@UniqueConstraint` annotation to define a uniqueness constraint on a combination of columns.

```
@Entity
@Table(uniqueConstraints=@UniqueConstraint(columnNames={"title", "year", "publisher_id"}))
class Book { ... }
```

This annotation looks a bit ugly perhaps, but it's actually useful even as documentation.

The `@Check` annotation adds a check constraint to the table.

```
@Entity
@Check(name="ValidISBN", constraints="length(isbn)=13")
class Book { ... }
```

The `@Check` annotation is commonly used at the field level:

```
@Id @Check(constraints="length(isbn)=13")
String isbn;
```

Chapter 5. Interacting with the database

To interact with the database, that is, to execute queries, or to insert, update, or delete data, we need an instance of one of the following objects:

- a JPA `EntityManager`,
- a Hibernate `Session`, or
- a Hibernate `StatelessSession`.

The `Session` interface extends `EntityManager`, and so the only difference between the two interfaces is that `Session` offers a few more operations.

> Actually, in Hibernate, every `EntityManager` is a `Session`, and you can narrow it like this:
>
> ```
> Session session = entityManager.unwrap(Session.class);
> ```

An instance of `Session` (or of `EntityManager`) is a *stateful session*. It mediates the interaction between your program and the database via operations on a *persistence context*.

In this chapter, we're not going to talk much about `StatelessSession`. We'll come back to this very useful API when we talk about performance. What you need to know for now is that a stateless session doesn't have a persistence context.

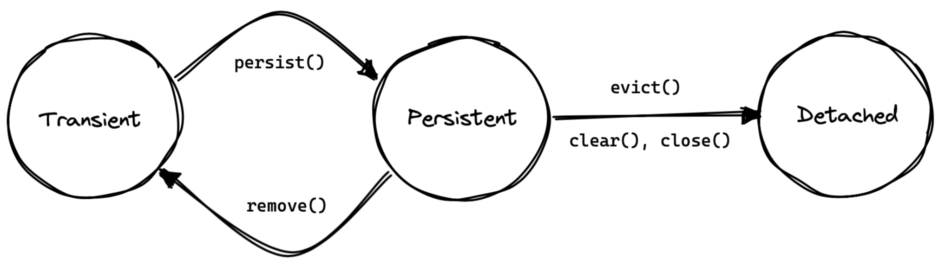
> Still, we should let you know that some people prefer to use `StatelessSession` everywhere. It's a simpler programming model, and lets the developer interact with the database more *directly*.
>
> Stateful sessions certainly have their advantages, but they're more difficult to reason about, and when something goes wrong, the error messages can be more difficult to understand.

5.1. Persistence contexts

A persistence context is a sort of cache; we sometimes call it the "first-level cache", to distinguish it from the second-level cache. For every entity instance read from the database within the scope of a persistence context, and for every new entity made persistent within the scope of the persistence context, the context holds a unique mapping from the identifier of the entity instance to the instance itself.

Thus, an entity instance may be in one of three states with respect to a given persistence context:

1. *transient* — never persistent, and not associated with the persistence context,
2. *persistent* — currently associated with the persistence context, or
3. *detached* — previously persistent in another session, but not currently associated with *this* persistence context.



At any given moment, an instance may be associated with at most one persistence context.

The lifetime of a persistence context usually corresponds to the lifetime of a transaction, though it's possible to have a persistence context that spans several database-level transactions that form a single logical unit of work.

> A persistence context—that is, a `Session` or `EntityManager`—absolutely positively **must not be shared between multiple threads or between concurrent transactions.**
>
> If you accidentally leak a session across threads, you will suffer.

Container-managed persistence contexts

In a container environment, the lifecycle of a persistence context scoped to the transaction will usually be managed for you.

There are several reasons we like persistence contexts.

1. They help avoid *data aliasing*: if we modify an entity in one section of code, then other code executing within the same persistence context will see our modification.
2. They enable *automatic dirty checking*: after modifying an entity, we don't need to perform any explicit operation to ask Hibernate to propagate that change back to the database. Instead, the change will be automatically synchronized with the database when the session is flushed.
3. They can improve performance by avoiding a trip to the database when a given entity instance is requested repeatedly in a given unit of work.
4. They make it possible to *transparently batch* together multiple database operations.

A persistence context also allows us to detect circularities when performing operations on graphs of entities. (Even in a stateless session, we need some sort of temporary cache of the entity instances we've visited while executing a query.)

On the other hand, stateful sessions come with some very important restrictions, since:

- persistence contexts aren't threadsafe, and can't be shared across threads, and
- a persistence context can't be reused across unrelated transactions, since that would break the isolation and atomicity of the transactions.

Furthermore, a persistence context holds a hard references to all its entities, preventing them from being garbage collected. Thus, the session must be discarded once a unit of work is complete.

> If you don't completely understand the previous passage, go back and re-read it until you do. A great deal of human suffering has resulted from users mismanaging the lifecycle of the Hibernate `Session` or JPA `EntityManager`.

We'll conclude by noting that whether a persistence context helps or harms the performance of a given unit of work depends greatly on the nature of the unit of work. For this reason Hibernate provides both stateful and stateless sessions.

5.2. Creating a session

Sticking with standard JPA-defined APIs, we saw how to obtain an `EntityManagerFactory` in Configuration using JPA XML. It's quite unsurprising that we may use this object to create an `EntityManager`:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
```

When we're finished with the `EntityManager`, we should explicitly clean it up:

```
entityManager.close();
```

On the other hand, if we're starting from a `SessionFactory`, as described in Programmatic configuration using JPA API, we may use:

```
Session session = sessionFactory.openSession();
```

But we still need to clean up:

```
session.close();
```

Injecting the `EntityManager`

If you're writing code for some sort of container environment, you'll probably obtain the `EntityManager` by some sort of dependency injection. For example, in Java (or Jakarta) EE you would write:

```
@PersistenceContext EntityManager entityManager;
```

In Quarkus, injection is handled by CDI:

```
@Inject EntityManager entityManager;
```

Outside a container environment, we'll also have to write code to manage database transactions.

5.3. Managing transactions

Using JPA-standard APIs, the `EntityTransaction` interface allows us to control database transactions. The idiom we recommend is the following:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction tx = entityManager.getTransaction();
try {
    tx.begin();
    //do some work
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx.isActive()) tx.rollback();
    throw e;
}
finally {
    entityManager.close();
}
```

But this code is extremely tedious, so there's a cleaner option:

```
entityManagerFactory.runInTransaction(entityManager -> {
    // do the work
    ...
});
```

When we need to return a value from within the anonymous function, we use `callInTransaction()` instead of `runInTransaction()`.

Using Hibernate's native APIs we can write something very similar:

```
sessionFactory.inTransaction(session -> {
    //do the work
    ...
});
```

Container-managed transactions

In a container environment, the container itself is usually responsible for managing transactions. In Java EE or Quarkus, you'll probably indicate the boundaries of the transaction using the `@Transactional` annotation.

The `EntityTransaction` interface provides a standard way to set the transaction timeout:

```
entityManager.getTransaction().setTimeout(30); // 30 seconds
```

`EntityTransaction` also provides a way to set the transaction to rollback-only mode:

```
entityManager.getTransaction().setRollbackOnly();
```

A transaction in rollback-only mode will be rolled back when it completes.

> Hibernate is not a software transactional memory. When a transaction rolls back, Hibernate makes no attempt to roll back the state of objects held in memory to their state at the beginning of the transaction. After a transaction rollback, the persistence context must be discarded, and the state of its entities must be assumed inconsistent with the state held by the database.

The interface `Transaction` extends `EntityTransaction` with some additional operations, including the ability to register transaction completion callbacks.

5.4. Operations on the persistence context

Of course, the main reason we need an `EntityManager` is to do stuff to the database. The following important operations let us interact with the persistence context and schedule modifications to the data:

Table 5.1: Methods for modifying data and managing the persistence context

Method name and parameters	Effect
`persist(Object)`	Make a transient object persistent and schedule a SQL `insert` statement for later execution
`remove(Object)`	Make a persistent object transient and schedule a SQL `delete` statement for later execution
`merge(Object)`	Copy the state of a given detached object to a corresponding managed persistent instance and return the persistent object
`detach(Object)`	Disassociate a persistent object from a session without affecting the database
`clear()`	Empty the persistence context and detach all its entities
`flush()`	Detect changes made to persistent objects association with the session and synchronize the database state with the state of the session by executing SQL `insert`, `update`, and `delete` statements

Notice that `persist()` and `remove()` have no immediate effect on the database, and instead simply schedule a command for later execution. Also notice that there's no `update()` operation for a stateful session. Modifications are automatically detected when the session is flushed.

On the other hand, except for `getReference()`, the following operations all result in immediate access to the database:

Table 5.2: Methods for reading and locking data

Method name and parameters	Effect
`find(Class,Object)`	Obtain a persistent object given its type and its id
`find(Class,Object,LockModeType)`	Obtain a persistent object given its type and its id, requesting the given optimistic or pessimistic lock mode
`find(EntityGraph,Object)`	Obtain a persistent object given its id and an `EntityGraph` specifying its type and associations which should be eagerly fetched
`getReference(Class,id)`	Obtain a reference to a persistent object given its type and its id, without actually loading its state from the database
`getReference(Object)`	Obtain a reference to a persistent object with the same identity as the given detached instance, without actually loading its state from the database
`refresh(Object)`	Refresh the persistent state of an object using a new SQL `select` to retrieve its current state from the database
`refresh(Object,LockModeType)`	Refresh the persistent state of an object using a new SQL `select` to retrieve its current state from the database, requesting the given optimistic or pessimistic lock mode
`lock(Object,LockModeType)`	Obtain an optimistic or pessimistic lock on a persistent object

Any of these operations might throw an exception. Now, if an exception occurs while interacting with the database, there's no good way to resynchronize the state of the current persistence context with the state held in database tables.

Therefore, a session is considered to be unusable after any of its methods throws an exception.

> The persistence context is fragile. If you receive an exception from Hibernate, you should immediately close and discard the current session. Open a new session if you need to, but throw the bad one away first.

One very important kind of exception which can happen when data is shared between concurrent units of work is an *optimistic lock failure*. Optimistic locks are verified by checking versions. A version check is included in the `where` clause of every SQL `update` or `delete` statement for a versioned entity. If a version check fails—that is, if no rows are updated—Hibernate infers that the entity was updated in some other unit of work and throws an `OptimisticLockException` to indicate that the current session is working with stale data. As with other exceptions, this loss of synchronization between the persistence context and the database means that we must discard the current session.

Some of these operations listed above require slightly more care than others. When you call `detach()`, `clear()`, `flush()`, or `refresh()`, you've already strayed from the narrow path. You didn't stray far—and you probably had a good reason for going there—but you're in territory where Hibernate just has to assume you know what you're doing. If you start to feel that this terrain is bogging you down, consider using a stateless session.

Four of these operations accept *options*, allowing influence over their behavior.

Method name and parameters	Effect
`find(Class,Object,FindOption…)`	Obtain a persistent object given its type and its id, using the specified options
`find(EntityGraph,Object,FindOption…)`	Obtain a persistent object given its id and an `EntityGraph` specifying its type and associations which should be eagerly fetched, using the specified options
`refresh(Object,LockModeType,RefreshOption…)`	Refresh the persistent state of an object using a new SQL `select` to retrieve its current state from the database, requesting the given optimistic or pessimistic lock mode, using the specified options
`lock(Object,LockModeType,LockOption…)`	Obtain an optimistic or pessimistic lock on a persistent object, using the specified options

For example, JPA provides the `Timeout` class which is a `FindOption`, a `RefreshOption`, and a `LockOption`.

```
var book = entityManger.find(Book.class, isbn, Timeout.ms(100), CacheStoreMode.BYPASS);
```

Finally, the Hibernate `Session` offers the following method, which is capable of efficiently loading multiple entity instances in parallel:

Method name and parameters	Effect
`findMultiple(Class,List<Object>,FindOption…)`	Obtain a list of persistent objects given their type and their ids, using the specified options
`findMultiple(EntityGraph,List<Object>,FindOption…)`	Obtain a list of persistent objects given their ids and an `EntityGraph` specifying their type and associations which should be eagerly fetched, using the specified options

The following code results in a single SQL `select` statement:

```
List<Book> books = session.findMultiple(Book.class, bookIds);
```

As discussed earlier, Hibernate offers the ability to map a natural id and perform load operations using that natural id. This is accomplished using the `KeyType#NATURAL` `FindOption` -

```
var bookKey = new BookKey(...);
var book = session.find(Book.class, bookKey, NATURAL);
var books = session.findMultiple(Book.class, List.of(bookKey), NATURAL);
```

When loading by natural id, the type of value accepted depends on the type of natural id. For single-attribute natural ids, whether defined by a basic or embedded type, the attribute type should be used. For multi-attribute natural ids, Hibernate will accept a number of forms:

- If a `@NaturalIdClass` is defined, an instance of the natural id class may be used.
- An array of the individual attribute values, ordered alphabetically by name, may be used.
- A `Map` of the individual attribute values, keyed by the attribute name, may be used.

Each of the operations we've seen so far affects a single entity instance passed as an argument. But there's a way to set things up so that an operation will propagate to associated entities.

See the User Guide for more details about loading by natural ids.

5.5. Cascading persistence operations

It's quite often the case that the lifecycle of a *child* entity is completely dependent on the lifecycle of some *parent*. This is especially common for many-to-one and one-to-one associations, though it's very rare for many-to-many associations.

For example, it's quite common to make an `Order` and all its `Item`s persistent in the same transaction, or to delete a `Project` and its `Files`s at once. This sort of relationship is sometimes called a *whole/part*-type relationship.

Cascading is a convenience which allows us to propagate one of the operations listed in Operations on the persistence context from a parent to its children. To set up cascading, we specify the `cascade` member of one of the association mapping annotations, usually `@OneToMany` or `@OneToOne`.

```
@Entity
class Order {
    ...
    @OneToMany(mappedby=Item_.ORDER,
               // cascade persist(), remove(), and refresh() from Order to Item
               cascade={PERSIST,REMOVE,REFRESH},
               // also remove() orphaned Items
               orphanRemoval=true)
    private Set<Item> items;
    ...
}
```

Orphan removal indicates that an `Item` should be automatically deleted if it is removed from the set of items belonging to its parent `Order`.

5.6. Proxies and lazy fetching

Our data model is a set of interconnected entities, and in Java our whole dataset would be represented as an enormous interconnected graph of objects. It's possible that this graph is disconnected, but more likely it's connected, or composed of a relatively small number of connected subgraphs.

Therefore, when we retrieve an object belonging to this graph from the database and instantiate it in memory, we simply can't recursively retrieve and instantiate all its associated entities. Quite aside from the waste of memory on the VM side, this process would involve a huge number of round trips to the database server, or a massive multidimensional cartesian product of tables, or both. Instead, we're forced to cut the graph somewhere.

Hibernate solves this problem using *proxies* and *lazy fetching*. A proxy is an object that masquerades as a real entity or collection, but doesn't actually hold any state, because that state has not yet been fetched from the database. When you call a method of the proxy, Hibernate will detect the call and fetch the state from the database before allowing the invocation to proceed to the real entity object or collection.

Now for the gotchas:

1. Hibernate will only do this for an entity which is currently associated with a persistence context. Once the session ends, and the persistence context is cleaned up, the proxy is no longer fetchable, and instead its methods throw the hated `LazyInitializationException`.
2. For a polymorphic association, Hibernate does not know the concrete type of the referenced entity when the proxy is instantiated, and so operations like `instanceof` and typecasts do not work correctly when applied to a proxy.
3. A round trip to the database to fetch the state of a single entity instance is just about *the least efficient* way to access data. It almost inevitably leads to the infamous *N+1 selects* problem we'll discuss later when we talk about how to optimize association fetching.

> The `@ConcreteProxy` annotation solves gotcha 2, but at the cost of performance (extra joins), and so its use is not generally recommended, except in very special circumstances.

> We're getting a bit ahead of ourselves here, but let's quickly mention the general strategy we recommend to navigate past these gotchas:
>
> - All associations should be set `fetch=LAZY` to avoid fetching extra data when it's not needed. As we mentioned earlier, this setting is not the default for `@ManyToOne` associations, and must be specified explicitly.
> - But strive to avoid writing code which triggers lazy fetching. Instead, fetch all the data you'll need upfront at the beginning of a unit of work, using one of the techniques described in Association fetching, usually, using *join fetch* in HQL or an `EntityGraph`.

It's important to know that some operations which may be performed with an unfetched proxy *don't* require fetching its state from the database. First, we're always allowed to obtain its identifier:

```
var pubId = entityManager.find(Book.class, bookId).getPublisher().getId(); // does not fetch publisher
```

Second, we may create an association to a proxy:

```
book.setPublisher(entityManager.getReference(Publisher.class, pubId)); // does not fetch publisher
```

Sometimes it's useful to test whether a proxy or collection has been fetched from the database. JPA lets us do this using the `PersistenceUnitUtil`:

```
boolean authorsFetched = entityManagerFactory.getPersistenceUnitUtil().isLoaded(book.getAuthors());
```

Hibernate has a slightly easier way to do it:

```
boolean authorsFetched = Hibernate.isInitialized(book.getAuthors());
```

Similarly, `PersistenceUnitUtil.load()` force-fetches a proxy or collection:

```
Book book = session.find(Book.class, bookId);  // fetch just the Book, leaving authors unfetched
entityManagerFactory.getPersistenceUnitUtil().load(book.getAuthors());
```

Again, `Hibernate.initialize()` is slightly more convenient:

```
Book book = session.find(Book.class, bookId);  // fetch just the Book, leaving authors unfetched
Hibernate.initialize(book.getAuthors());  // fetch the Authors
```

On the other hand, the above code is very inefficient, requiring two trips to the database to obtain data that could in principle be retrieved with just one query.

The static methods of the `Hibernate` class let us do a lot more, and it's worth getting a bit familiar with them. Of particular interest are the operations which let us work with unfetched collections without fetching their state from the database. For example, consider this code:

```
Book book = session.find(Book.class, bookId);  // fetch just the Book, leaving authors unfetched
Author authorRef = session.getReference(Author.class, authorId);  // obtain an unfetched proxy
boolean isByAuthor = Hibernate.contains(book.getAuthors(), authorRef); // no fetching
```

This code fragment leaves both the set `book.authors` and the proxy `authorRef` unfetched.

It's clear from the discussion above that we need a way to request that an association be *eagerly* fetched using a database `join`, thus protecting ourselves from the infamous N+1 selects. One way to do this is by passing an `EntityGraph` to `find()`.

5.7. Entity graphs and eager fetching

When an association is mapped `fetch=LAZY`, it won't, by default, be fetched when we call the `find()` method. We may request that an association be fetched eagerly (immediately) by passing an `EntityGraph` to `find()`.

```
var graph = entityManager.createEntityGraph(Book.class);
graph.addSubgraph(Book_.publisher);
Book book = entityManager.find(graph, bookId);
```

This code adds a `left outer join` to our SQL query, fetching the associated `Publisher` along with the `Book`.

We may even attach additional nodes to our `EntityGraph`:

```
var graph = session.createEntityGraph(Book.class);
graph.addSubgraph(Book_.publisher);
graph.addPluralSubgraph(Book_.authors).addSubgraph(Author_.person);
Book book = entityManager.find(graph, bookId);
```

This results in a SQL query with *four* `left outer join`s.

> In the code examples above, The classes `Book_` and `Author_` are generated by Hibernate Processor, as we saw earlier. They let us refer to attributes of our model in a completely type-safe way. We'll use them again, below, when we talk about Criteria queries.

JPA specifies that any given `EntityGraph` may be interpreted in two different ways.

- A *fetch graph* specifies exactly the associations that should be eagerly loaded. Any association not belonging to the entity graph is proxied and loaded lazily only if required.
- A *load graph* specifies that the associations in the entity graph are to be fetched in addition to the associations mapped `fetch=EAGER`.

An `EntityGraph` passed directly to `find()` is always interpreted as a load graph.

> You're right, the names make no sense. But don't worry, if you take our advice, and map your associations `fetch=LAZY`, there's no difference between a "fetch" graph and a "load" graph, so the names don't matter.

> JPA even specifies a way to define named entity graphs using annotations. But the annotation-based API is so verbose that it's just not worth using.

5.8. Controlling lookup by id

As we will soon see in the next chapter, we can do almost anything via HQL, criteria, or native SQL queries. But when we already know the identifier of the entity we need, a query can feel like overkill. And queries don't make efficient use of the second level cache.

We met the `find()` and `findMultiple()` methods earlier. These are the most basic ways to perform a *lookup* by id. But they can't quite do everything. Therefore, Hibernate has some APIs that streamline certain more complicated lookups:

Table 5.5: Operations for lookup by id

Method name	Purpose
`byId()`	Lets us specify association fetching via an `EntityGraph`, as we saw; also lets us specify some additional options, including how the lookup interacts with the second level cache, and whether the entity should be loaded in read-only mode
`byMultipleIds()`	Lets us load a *batch* of ids at the same time

> Since the introduction of `FindOption` in JPA 3.2, `byId()` is now much less useful and deprecated. Instead, use `find()` and `findMultiple()` as discussed earlier.

Batch loading is very useful when we need to retrieve multiple instances of the same entity class by id:

```
var graph = session.createEntityGraph(Book.class);
graph.addSubgraph(Book_.publisher);

List<Book> books =
        session.byMultipleIds(Book.class)
            .withFetchGraph(graph)  // control association fetching
            .withBatchSize(20)      // specify an explicit batch size
            .with(CacheMode.GET)    // control interaction with the cache
            .multiLoad(bookIds);
```

The given list of `bookIds` will be broken into batches, and each batch will be fetched from the database in a single `select`. If we don't specify the batch size explicitly, a batch size will be chosen automatically.

We also have some operations for working with lookups by natural id:

Method name	Purpose
`bySimpleNaturalId()`	For an entity with just one attribute is annotated `@NaturalId`
`byNaturalId()`	For an entity with multiple attributes are annotated `@NaturalId`
`byMultipleNaturalId()`	Lets us load a *batch* of natural ids at the same time

Again, see the User Guide for more details about loading by natural ids.

Here's how we can retrieve an entity by its composite natural id:

```
Book book =
```

```java
        session.byNaturalId(Book.class)
            .using(Book_.isbn, isbn)
            .using(Book_.printing, printing)
            .load();
```

Notice that this code fragment is completely typesafe, again thanks to Hibernate Processor.

5.9. Controlling state retrieval during merge

The `merge()` operation is usually used when some interaction with a user or automated system spans multiple transactions, with each transaction having its own persistence context:

1. an entity `e` is retrieved in one persistence context, and then the current transaction ends, resulting in destruction of the persistence context and in the entity becoming detached, then
2. `e` is modified in some way while detached, and then
3. finally, the modification is made persistent by merging the detached instance in a second transaction, with a new persistence context, by calling `session.merge(e)`.

In step 3, the original entity instance `e` remains detached, but `merge()` returns a distinct instance `f` representing the same row of the database and associated with the new persistence context. That is, `merge()` trades a detached instance for a persistent instance representing the same row.

To determine the nature of the modification held in `e` and to guarantee correct semantics with respect to optimistic locking, Hibernate first `select`s the current persistent state of the entity from the database before applying the modification to `f`.

Cascade merge allows multiple modifications held in a graph of entity instances to be made persistent in one call to `merge()`. When `merge()` is used with cascading — that is, when the `merge()` operation is applied to a root entity with associations mapped `cascade=MERGE` — Hibernate issues a single `select` statement to retrieve the current database state of the root entity and all its associated entities. This behavior avoids the use of N+1 select statements for state retrieval during cascade merge but, in certain circumstances, the query might be suboptimal. On the other hand, this query does not occur if the root entity was already loaded into the persistence context before `merge()` is called.

Therefore, when using the `EntityManager` API, we may gain control over the way state is loaded before a `merge()` just by calling `find()` before calling `merge()`.

```java
Book book = ... ;
var graph = entityManager.createEntityGraph(Book.class);
graph.addSubgraph(Book_.chapters);  // Book.chapters mapped cascade=MERGE
entityManager.find(graph, book.getIsbn());  // force loading of the book
entityManager.merge(book);  // merge the detached objet
```

The `Session` interface provides a streamlined alternative:

```java
Book book = ... ;
var graph = session.createEntityGraph(Book.class);
graph.addSubgraph(Book_.chapters);
session.merge(book, graph);  // equivalent to find() then merge()
```

When merging multiple root entities, `findMultiple()` may be used instead of `find()`.

```java
List<Book> books = ... ;
var isbns = books.stream().map(Book::getIsbn).toList();
var graph = session.createEntityGraph(Book.class);
graph.addSubgraph(Book_.chapters);  // Book.chapters mapped cascade=MERGE
session.findMultiple(graph, isbns);  // force loading of the books
for (var book in books) {
    session.merge(book);
}
```

In some cases, `merge()` is much less efficient than the `upsert()` operation of `StatelessSession`.

5.10. Flushing the session

From time to time, a *flush* operation is triggered, and the session synchronizes dirty state held in memory—that is, modifications to the state of entities associated with the persistence context—with persistent state held in the database. Of course, it does this by executing SQL

INSERT, UPDATE, and DELETE statements.

By default, a flush is triggered:

- when the current transaction commits, for example, when Transaction.commit() is called,
- before execution of a query whose result would be affected by the synchronization of dirty state held in memory, or
- when the program directly calls flush().

In the following code, the flush occurs when the transaction commits:

```
session.getTransaction().begin();
session.persist(author);
var books =
        // new Author does not affect results of query for Books
        session.createSelectionQuery("from Book")
                // no need to flush
                .getResultList();
// flush occurs here, just before transaction commits
session.getTransaction().commit();
```

But in this code, the flush occurs when the query is executed:

```
session.getTransaction().begin();
session.persist(book);
var books =
        // new Book would affect results of query for Books
        session.createSelectionQuery("from Book")
                // flush occurs here, just before query is executed
                .getResultList();
// changes were already flushed to database, nothing to flush
session.getTransaction().commit();
```

It's always possible to call flush() explicitly:

```
session.getTransaction().begin();
session.persist(author);
session.flush(); // explicit flush
var books =
        session.createSelectionQuery("from Book")
                // nothing to flush
                .getResultList();
// nothing to flush
session.getTransaction().commit();
```

> Notice that SQL statements are not usually executed synchronously by methods of the Session interface like persist() and remove(). If synchronous execution of SQL is desired, the StatelessSession allows this.

This behavior can be controlled by explicitly setting the flush mode. For example, to disable flushes that occur before query execution, call:

```
entityManager.setFlushMode(FlushModeType.COMMIT);
```

Hibernate allows greater control over the flush mode than JPA:

```
session.setHibernateFlushMode(FlushMode.MANUAL);
```

Since flushing is a somewhat expensive operation (the session must dirty-check every entity in the persistence context), setting the flush mode to COMMIT can occasionally be a useful optimization. But take care—in this mode, queries might return stale data:

```
session.getTransaction().begin();
session.setFlushMode(FlushModeType.COMMIT); // disable AUTO-flush
session.persist(book);
var books =
        // flushing on query execution disabled
        session.createSelectionQuery("from Book")
                // no flush, query returns stale results
                .getResultList();
```

```
// flush occurs here, just before transaction commits
session.getTransaction().commit();
```

Table 5.7: Flush modes

Hibernate `FlushMode`	**JPA** `FlushModeType`	**Interpretation**
`MANUAL`		Never flush automatically
`COMMIT`	`COMMIT`	Flush before transaction commit
`AUTO`	`AUTO`	Flush before transaction commit, and before execution of a query whose results might be affected by modifications held in memory
`ALWAYS`		Flush before transaction commit, and before execution of every query

A second way to reduce the cost of flushing is to load entities in *read-only* mode:

- `Session.setDefaultReadOnly(true)` specifies that all entities loaded by a given session should be loaded in read-only mode by default,
- `SelectionQuery.setReadOnly(true)` specifies that every entity returned by a given query should be loaded in read-only mode, and
- `Session.setReadOnly(Object, true)` specifies that a given entity already loaded by the session should be switched to read-only mode.

Hibernate's `ReadOnlyMode` is a custom `FindOption`:

```
var book = entityManager.find(Book.class, isbn, ReadOnlyMode.READ_ONLY);
```

It's not necessary to dirty-check an entity instance in read-only mode.

5.11. Lifecycle callbacks and entity listeners

The annotations `@PrePersist`, `@PreRemove`, `@PreUpdate`, `@PostPersist`, `@PostRemove`, `@PostUpdate`, and `@PostLoad` allow an entity to respond to persistence lifecycle operations and maintain its transient internal state. For example:

```
@Entity
class Order {
    ...
    transient double total;

    @PostLoad
    void computeTotal() {
        total = items.stream().mapToDouble(i -> i.price * i.quantity).sum();
    }

    ...
}
```

If we need to interact with technical objects, we can place the lifecycle callback on a separate class, called an *entity listener*. The `@EntityListeners` annotation specifies the listeners for a given entity class:

```
@Entity
@EntityListeners(OrderEvents.class)
class Order { ... }
```

An entity listener may inject CDI beans:

```
// entity listener class
class OrderEvents {
    @Inject
    Event<NewOrder> newOrderEvent;

    @PostPersist
    void newOrder(Order order) {
        // send a CDI event
        newOrderEvent.fire(new NewOrder(order));
    }
```

```
}
```

A single entity listener class may even be a generic listener that receives lifecycle callbacks for multiple different entity classes.

> The venerable `Interceptor` interface is a more powerful alternative to entity listeners. `Interceptor` and its friend `CustomEntityDirtinessStrategy` allow advanced users to augment the built-in handling of managed entities with custom behavior. These interfaces are very useful if you're building your own persistence framework with Hibernate as the foundation.

We're about to see one way `Interceptor` can be used.

5.12. Transient vs detached

Sometimes, Hibernate needs to be able to distinguish whether an entity instance is:

- a brand-new transient object the client just instantiated using `new`, or
- a detached object, which previously belonged to a persistence context.

This is a bit of a problem, since there's no good and efficient way for Hibernate to just tag an entity with a Post-it saying "I've seen you before".

Therefore, Hibernate uses heuristics. The two most useful heuristics are:

1. If the entity has a generated identifier, the value of the id field is inspected: if the value currently assigned to the id field is the default value for the type of the field, then the object is transient; otherwise, the object is detached.
2. If the entity has a version, the value of the version field is inspected: if the value currently assigned to the version field is the default value, or a negative number, then the object is transient; otherwise, the object is detached.

If the entity has neither a generated id, nor a version, Hibernate usually falls back to just doing something reasonable. In extreme cases a `SELECT` query will be issued to determine whether a matching row exists in the database.

> These heuristics aren't perfect. It's quite easy to confuse Hibernate by assigning a value to the id field or version field, making a new transient instance look like it's detached. We therefore strongly discourage assigning values to fields annotated `@GeneratedValue` or `@Version` before passing an entity to Hibernate.

If the heuristics ever happen cause a real problem, you may implement your own Post-it tagging via `Interceptor.isTransient()`.

5.13. Interacting directly with JDBC

From time to time we run into the need to write some code that calls JDBC directly. The `EntityManager` now offers a convenient way to do this:

```
entityManager.runWithConnection((Connection connection) -> {
    try (var callable = connection.prepareCall("{call myproc(?)}")) {
        callable.setLong(1, argument);
        callable.execute();
    }
});
```

To return a value, use `callWithConnection()` instead of `runWithConnection()`.

The Hibernate `Session` has an older, slightly simpler API:

```
session.doWork(connection -> {
    try (var callable = connection.prepareCall("{call myproc(?)}")) {
        callable.setLong(1, argument);
        callable.execute();
    }
});
```

If the work returns a value, use `doReturningWork()` instead of `doWork()`.

The `Connection` passed to the work is the same connection being used by the session, and so any work performed using that connection occurs in the same transaction context.

> In a container environment where transactions and database connections are managed by the container, this might not

> be the easiest way to obtain the JDBC connection.

5.14. What to do when things go wrong

Object/relational mapping has been called the "Vietnam of computer science". The person who made this analogy is American, and so one supposes that he meant to imply some kind of unwinnable war. This is quite ironic, since at the very moment he made this comment, Hibernate was already on the brink of winning the war.

Today, Vietnam is a peaceful country with exploding per-capita GDP, and ORM is a solved problem. That said, Hibernate is complex, and ORM still presents many pitfalls for the inexperienced, even occasionally for the experienced. Sometimes things go wrong.

In this section we'll quickly sketch some general strategies for avoiding "quagmires".

- Understand SQL and the relational model. Know the capabilities of your RDBMS. Work closely with the DBA if you're lucky enough to have one. Hibernate is not about "transparent persistence" for Java objects. It's about making two excellent technologies work smoothly together.
- Log the SQL executed by Hibernate. You cannot know that your persistence logic is correct until you've actually inspected the SQL that's being executed. Even when everything seems to be "working", there might be a lurking N+1 selects monster.
- Be careful when modifying bidirectional associations. In principle, you should update *both ends* of the association. But Hibernate doesn't strictly enforce that, since there are situations where such a rule would be too heavy-handed. Whatever the case, it's up to you to maintain consistency across your model.
- Never leak a persistence context across threads or concurrent transactions. Have a strategy or framework to guarantee this never happens.
- When running queries that return large result sets, take care to consider the size of the session cache. Consider using a stateless session.
- Think carefully about the semantics of the second-level cache, and how the caching policies impact transaction isolation.
- Avoid fancy bells and whistles you don't need. Hibernate is incredibly feature-rich, and that's a good thing, because it serves the needs of a huge number of users, many of whom have one or two very specialized needs. But nobody has *all* those specialized needs. In all probability, you have none of them. Write your domain model in the simplest way that's reasonable, using the simplest mapping strategies that make sense.
- When something isn't behaving as you expect, *simplify*. Isolate the problem. Find the absolute minimum test case which reproduces the behavior, *before* asking for help online. Most of the time, the mere act of isolating the problem will suggest an obvious solution.
- If you're new to Hibernate, avoid frameworks and libraries that "wrap" JPA. You need an excellent understanding of Hibernate and JPA *first*, before introducing unnecessary additional moving parts. If there's any one criticism of Hibernate and ORM that sometimes *does* ring true, it's that it takes you too far from direct control over JDBC. An additional layer just takes you even further. If you insist you really do need this extra layer, we beg you to consider Hibernate Data Repositories instead of older third-party solutions.
- Avoid copy/pasting code from random bloggers or stackoverflow reply guys. Many of the suggestions you'll find online just aren't the simplest solution, and many aren't correct for Hibernate 6 and 7. Instead, *understand* what you're doing; study the Javadoc of the APIs you're using; read the JPA specification; follow the advice we give in this document; go direct to the Hibernate team on Zulip. (Sure, we can be a bit cantankerous at times, but we *do* always want you to be successful.)
- Always consider other options. You don't have to use Hibernate for *everything*.

Chapter 6. Executing queries

Hibernate features three complementary ways to write queries:

- the *Hibernate Query Language*, an extremely powerful superset of JPQL, which abstracts most of the features of modern dialects of SQL,
- the JPA *criteria query* API, along with extensions, allowing almost any HQL query to be constructed programmatically via a typesafe API, and, of course
- for when all else fails, *native SQL* queries.

In addition, Hibernate 7 provides a convenient new way to programmatically customize a query before executing it.

6.1. HQL queries

A full discussion of the query language would require almost as much text as the rest of this Short Guide. Fortunately, HQL is already described in exhaustive (and exhausting) detail in *A Guide to Hibernate Query Language*. It doesn't make sense to repeat that information here.

Here we want to see how to execute a query via the `Session` or `EntityManager` API. The method we call depends on what kind of query it is:

- *selection queries* return a result list, but do not modify the data, but
- *mutation queries* modify data, and return the number of modified rows.

Selection queries usually start with the keyword `select` or `from`, whereas mutation queries begin with the keyword `insert`, `update`, or `delete`.

Table 6.1: Executing HQL

Kind	`Session` **method**	`EntityManager` **method**	`Query` **execution method**
Selection	`createSelectionQuery(String,Class)`	`createQuery(String,Class)`	`getResultList()`, `getSingleResult()`, or `getSingleResultOrNull()`
Mutation	`createMutationQuery(String)`	`createQuery(String)`	`executeUpdate()`

So for the `Session` API we would write:

```
List<Book> matchingBooks =
        session.createSelectionQuery("from Book where title like :titleSearchPattern", Book.class)
            .setParameter("titleSearchPattern", titleSearchPattern)
            .getResultList();
```

Or, if we're sticking to the JPA-standard APIs:

```
List<Book> matchingBooks =
        entityManager.createQuery("select b from Book b where b.title like :titleSearchPattern", Book.class)
            .setParameter("titleSearchPattern", titleSearchPattern)
            .getResultList();
```

The main difference between `createSelectionQuery()` and `createQuery()` is that `createSelectionQuery()` throws an exception if passed a query string that begins with `insert`, `delete`, or `update`.

We've been using `getResultList()` because we've been expecting our queries to return multiple results. If we're expecting a query to return a single result, we can use `getSingleResult()`.

```
Book book =
        session.createSelectionQuery("from Book where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .getSingleResult();
```

Or, if we're expecting it to return at most one result, we can use `getSingleResultOrNull()`.

```
Book bookOrNull =
        session.createSelectionQuery("from Book where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .getSingleResultOrNull();
```

The difference, of course, is that `getSingleResult()` throws an exception if there's no matching row in the database, whereas `getSingleResultOrNull()` just returns `null`.

To execute a `MutationQuery`, we use `executeUpdate()`, which returns the number of entities affected by the `insert`, `update`, or `delete`.

6.2. Query parameters

Queries are often parameterized.

- In the query above, `:titleSearchPattern` is called a *named parameter*.
- Alternatively, we may label a parameter by a number. Such a parameter is called an *ordinal parameter*.

We may easily rewrite our query to use an ordinal parameter:

```
List<Book> matchingBooks =
        session.createSelectionQuery("from Book where title like ?1", Book.class)
            .setParameter(1, titleSearchPattern)
            .getResultList();
```

When a query has multiple parameters, named parameters tend to be easier to read, even if slightly more verbose.

> *Never* concatenate user input with HQL and pass the concatenated string to `createSelectionQuery()`. This would open up the possibility for an attacker to execute arbitrary code on your database server.

The `setParameter()` methods specify arguments to query parameters.

> The two-argument forms of `setParameter()` are perfect for most purposes, but *very occasionally* it's necessary to resolve an ambiguity in the interpretation of the argument value by explicitly specifying the type of the argument. The best way to identify the type is via a reference to a JPA metamodel `Type`. There are two ways to do this:
>
> - by passing the `Type` as a third argument to `setParameter()`, or
> - by packaging the argument and its `Type` in a `TypedParameterValue`.
>
> For example, we may pass a static metamodel reference to `setParameter()`.
>
> ```
> session.createSelectionQuery("from Person where address = :address")
>         .setParameter("address" address, Person_.address.getType())
>         .getResultList();
> ```

6.3. Auto-flush

By default, Hibernate dirty checks entities in the persistence context before executing a query, in order to determine if there are changes which have not yet been flushed to the database, but which might affect the results of the query. If there are unflushed changes, then Hibernate goes ahead and executes an automatic flush before executing the query. That way, the query won't return stale results which fail to reflect changes made to data within the current unit of work. But if there are many entities association with the persistence context, then this can be an expensive operation.

To disable this behavior, set the query flush mode to `NO_FLUSH`:

```
Book bookOrNull =
        session.createSelectionQuery("from Book where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .setQueryFlushMode(QueryFlushMode.NO_FLUSH)
            .getSingleResult();
```

Or, especially if you're using JPA-standard APIs, use `FlushModeType.COMMIT`:

```
Book bookOrNull =
        session.createSelectionQuery("from Book where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .setFlushMode(FlushModeType.COMMIT)
            .getSingleResult();
```

> Setting the flush mode to `NO_FLUSH`, `COMMIT`, or `MANUAL` might cause the query to return stale results.

Occasionally we need to build a query at runtime, from a set of optional conditions. For this, JPA offers an API which allows programmatic construction of a query.

6.4. Criteria queries

Imagine we're implementing some sort of search screen, where the user of our system is offered several different ways to constrain the query result set. For example, we might let them search for books by title and/or the author name. Of course, we could construct a HQL query by string concatenation, but this is a bit fragile, so it's quite nice to have an alternative.

HQL is implemented in terms of criteria objects

Actually, since Hibernate 6, every HQL query is compiled to a criteria query before being translated to SQL. This ensures that the semantics of HQL and criteria queries are identical.

First we need an object for building criteria queries. Using the JPA-standard APIs, this would be a `CriteriaBuilder`, and we get it from the `EntityManagerFactory`:

```
CriteriaBuilder builder = entityManagerFactory.getCriteriaBuilder();
```

But if we have a `SessionFactory`, we get something much better, a `HibernateCriteriaBuilder`:

```
HibernateCriteriaBuilder builder = sessionFactory.getCriteriaBuilder();
```

The `HibernateCriteriaBuilder` extends `CriteriaBuilder` and adds many operations that JPQL doesn't have.

If you're using `EntityManagerFactory`, don't despair, you have two perfectly good ways to obtain the `HibernateCriteriaBuilder` associated with that factory. Either:

```
HibernateCriteriaBuilder builder =
        entityManagerFactory.unwrap(SessionFactory.class).getCriteriaBuilder();
```

Or simply:

```
HibernateCriteriaBuilder builder =
        (HibernateCriteriaBuilder) entityManagerFactory.getCriteriaBuilder();
```

We're ready to create a criteria query.

```
CriteriaQuery<Book> query = builder.createQuery(Book.class);
Root<Book> book = query.from(Book.class);
Predicate where = builder.conjunction();
if (titlePattern != null) {
    where = builder.and(where, builder.like(book.get(Book_.title), titlePattern));
}
if (namePattern != null) {
    Join<Book,Author> author = book.join(Book_.author);
    where = builder.and(where, builder.like(author.get(Author_.name), namePattern));
}
query.select(book).where(where)
    .orderBy(builder.asc(book.get(Book_.title)));
```

Here, as before, the classes `Book_` and `Author_` are generated by Hibernate Processor.

Notice that we didn't bother treating `titlePattern` and `namePattern` as parameters. That's safe because, by default, Hibernate automatically and transparently treats strings passed to the `CriteriaBuilder` as JDBC parameters.

Execution of a criteria query works almost exactly like execution of HQL.

Table 6.2: Executing criteria queries

Kind	`Session` method	`EntityManager` method	`Query` execution method
Selection	`createSelectionQuery(CriteriaQuery)`	`createQuery(CriteriaQuery)`	`getResultList()`, `getSingleResult()`, or `getSingleResultOrNull()`
Mutation	`createMutationQuery(CriteriaUpdate)` or `createMutationQuery(CriteriaDelete)`	`createQuery(CriteriaUpdate)` or `createQuery(CriteriaDelete)`	`executeUpdate()`

For example:

```
List<Book> matchingBooks =
        session.createSelectionQuery(query)
            .getResultList();
```

Update, insert, and delete queries work similarly:

```
CriteriaDelete<Book> delete = builder.createCriteriaDelete(Book.class);
Root<Book> book = delete.from(Book.class);
delete.where(builder.lt(builder.year(book.get(Book_.publicationDate)), 2000));
session.createMutationQuery(delete).executeUpdate();
```

> It's even possible to transform a HQL query string to a criteria query, and modify the query programmatically before execution:
>
> ```
> HibernateCriteriaBuilder builder = sessionFactory.getCriteriaBuilder();
> var query = builder.createQuery("from Book where year(publicationDate) > 2000", Book.class);
> var root = query.getRoot(0, Book.class);
> query.where(builder.like(root.get(Book_.title), builder.literal("Hibernate%")));
> query.orderBy(builder.asc(root.get(Book_.title)), builder.desc(root.get(Book_.isbn)));
> List<Book> matchingBooks = session.createSelectionQuery(query).getResultList();
> ```
>
> This is starting to get a bit messy. In Hibernate 7, we can often use `Restriction` instead.

Do you find some of the code above a bit too verbose? We do.

6.5. A more comfortable way to write criteria queries

Actually, what makes the JPA criteria API less ergonomic than it should be is the need to call all operations of the `CriteriaBuilder` as instance methods, instead of having them as `static` functions. The reason it works this way is that each JPA provider has its own implementation of `CriteriaBuilder`.

The helper class `CriteriaDefinition` can reduce the verbosity of criteria queries by eliminating the need to explicitly qualify calls to the methods of `CriteriaBuilder`. Our previous example would look like this:

```
CriteriaQuery<Book> query =
        new CriteriaDefinition(entityManagerFactory, Book.class) {{
            select(book);
            if (titlePattern != null) {
                restrict(like(book.get(Book_.title), titlePattern));
            }
            if (namePattern != null) {
                var author = book.join(Book_.author);
                restrict(like(author.get(Author_.name), namePattern));
            }
            orderBy(asc(book.get(Book_.title)));
        }};
```

When all else fails, and sometimes even before that, we're left with the option of writing a query in SQL.

6.6. Native SQL queries

HQL is a powerful language which helps reduce the verbosity of SQL, and significantly increases portability of queries between databases. But ultimately, the true value of ORM is not in avoiding SQL, but in alleviating the pain involved in dealing with SQL result sets once we get them back to our Java program. As we said right up front, Hibernate's generated SQL is meant to be used in conjunction with handwritten SQL, and native SQL queries are one of the facilities we provide to make that easy.

Table 6.3: Executing SQL

Kind	`Session` **method**	`EntityManager` **method**	`Query` **execution method**
Selection	`createNativeQuery(String,Class)`	`createNativeQuery(String,Class)`	`getResultList()`, `getSingleResult()`, or `getSingleResultOrNull()`
Mutation	`createNativeMutationQuery(String)`	`createNativeQuery(String)`	`executeUpdate()`
Stored procedure	`createStoredProcedureCall(String)`	`createStoredProcedureQuery(String)`	`execute()`

For the most simple cases, Hibernate can infer the shape of the result set:

```
Book book =
        session.createNativeQuery("select * from Books where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .getSingleResult();

String title =
        session.createNativeQuery("select title from Books where isbn = ?1", String.class)
            .setParameter(1, isbn)
            .getSingleResult();
```

However, in general, there isn't enough information in the JDBC `ResultSetMetaData` to infer the mapping of columns to entity objects. So for more complicated cases, you'll need to use the `@SqlResultSetMapping` annotation to define a named mapping, and pass the name to `createNativeQuery()`. This gets fairly messy, so we don't want to hurt your eyes by showing you an example of it.

By default, Hibernate doesn't flush the session before execution of a native query. That's because the session is unaware of which modifications held in memory would affect the results of the query.

So if there are any unflushed changes to `Book`s, this query might return stale data:

```
List<Book> books =
        session.createNativeQuery("select * from Books", Book.class)
            .getResultList();
```

There's two ways to ensure the persistence context is flushed before this query is executed.

Either, we could simply force a flush by calling `flush()` or by setting the flush mode to `ALWAYS`:

```
List<Book> books =
        session.createNativeQuery("select * from Books", Book.class)
            .setHibernateFlushMode(ALWAYS)
            .getResultList();
```

Or, alternatively, we could tell Hibernate which modified state affects the results of the query:

```
List<Book> books =
        session.createNativeQuery("select * from Books", Book.class)
            .addSynchronizedEntityClass(Book.class)
            .getResultList();
```

> You can call stored procedures using `createStoredProcedureQuery()` or `createStoredProcedureCall()`.

6.7. Restrictions and ordering

We've already seen how the JPA Criteria Query API can be used to construct a query completely programmatically. The Criteria API is powerful, but for the most common scenarios it's at least arguably overkill. The `CriteriaDefinition` class helps a bit, but it doesn't completely eliminate the verbosity of programmatic query definition.

In Hibernate 7, there's a new option, a very ergonomic API for programmatically adding restrictions or ordering to an existing query before executing it. (Actually, the ordering part of this was introduced in Hibernate 6.5.) This new API:

- isn't part of the Criteria Query API, and so we don't need a `CriteriaQuery` object to make use of it,
- *does* make use of the JPA static metamodel for type safety,
- works with both HQL and Criteria queries, and
- is optimized for the case of a query which returns its single root entity.

```
var selection =
        SelectionSpecification.create(Book.class,
            // an optional base query, written in HQL:
                "from Book where year(publicationDate) > 2000");

// add programmatic restrictions:
if (titlePattern != null)
    selection.restrict(Restriction.like(Book_.title, namePattern));
if (isbns != null && !isbns.isEmpty())
    selection.restrict(Restriction.in(Book_.isbn, isbns));

// add programmatic ordering:
if (orderByTitle) selection.sort(Order.asc(Book_.title));
if (orderByIsbn) selection.sort(Order.asc(Book_.isbn));

// add programmatic association fetching:
if (fetchPublisher) selection.fetch(Path.from(Book.class).to(Book_.publisher));

// execute the query in the given session:
List<Book> matchingBooks = selection.createQuery(session).getResultList();
```

Notice that:

- The `Restriction` interface has static methods for constructing a variety of different kinds of restriction in a completely typesafe way.
- Similarly, the `Order` interface has a variety of static methods for constructing different kinds of sorting criteria.

We need the following methods of `SelectionSpecification`:

Table 6.4: Methods for query restriction and ordering

Method name	Purpose
`restrict()`	Add a `Restriction` on the query results
`sort()`, `resort()`	Specify how the query results should be ordered
`fetch()`	Add a fetched association `Path`
`augment()`	Add a custom function which directly manipulates the select query

Two of these operations are also available for a `MutationSpecification`:

Table 6.5: Methods for mutation restriction

Method name	Purpose
`restrict()`	Add a `Restriction` on the records to be updated
`augment()`	Add a custom function which directly manipulates the update or delete query

Alternatively, `Restriction` and `Order` can be used with generated query or finder methods, and even with Jakarta Data repositories.

The interface `Path` may be used to express restrictions on fields of an embedded or associated entity class. It may even be used for association fetching.

```
List<Book> booksForPublisher =
        SelectionSpecification.create(Book.class)
                .restrict(Path.from(Book.class).to(Book_.publisher).to(Publisher_.name)
                                .equalTo(publisherName))
                .fetch(Path.from(Book.class).to(Book_.publisher))
                .createQuery(session)
                .getResultList();
```

Specifications aren't for everything, however.

> `SelectionSpecification` (similar to its friend `MutationSpecification`) may be used in cases where a query returns a single "root" entity, possibly with some fetched associations. It's not useful in cases where a query should return multiple entities, a projection of entity fields, or an aggregation. For such cases, the full Criteria API is appropriate.

Finally, the `augment()` method deserves its own subsection.

6.8. Augmentation

When `Restriction`, `Path`, and `Order` aren't expressive enough, we can *augment* the query by manipulating its representation as a criteria:

```
var books =
        SelectionSpecification.create(Book.class)
              .augment((builder, query, book) ->
                      // augment the query via JPA Criteria API
                      query.where(builder.like(book.get(Book_.title), titlePattern)))
                          .orderBy(builder.asc(book.get(Book_.isbn)))
              .createQuery(session)
              .getResultList();
```

For really advanced cases, `augment()` works quite nicely with `CriteriaDefinition`.

```
var books =
        SelectionSpecification.create(Book.class)
              .augment((builder, query, book) ->
                  // eliminate explicit references to 'builder'
                  new CriteriaDefinition<>(query) {{
                      where(like(entity.get(BasicEntity_.title), titlePattern),
                            greaterThan(book.get(Book_.pages), minPages));
                      orderBy(asc(book.get(Book_.isbn)));
                  }}
              )
              .createQuery(session)
              .getResultList();
```

However, we emphasize that this API shines in cases where complex manipulations are *not* required. For complicated queries involving multiple entities, or with aggregation and projection, you're best off heading straight to the `CriteriaBuilder`.

Programmatic restrictions, and especially programmatic ordering, are often used together with pagination.

6.9. Limits and pagination

If a query might return more results than we can handle at one time, we may specify:

- a *limit* on the maximum number of rows returned, and,
- optionally, an *offset*, the first row of an ordered result set to return.

> The offset is used to paginate query results.

There's two ways to add a limit or offset to a HQL or native SQL query:

- using the syntax of the query language itself, for example, `offset 10 rows fetch next 20 rows only`, or
- using the methods `setFirstResult()` and `setMaxResults()` of the `SelectionQuery` interface.

If the limit or offset is parameterized, the second option is simpler. For example, this:

```
List<Book> books =
        session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
            .setParameter(1, titlePattern)
            .setMaxResults(MAX_RESULTS)
            .getResultList();
```

is simpler than:

```
// a worse way to do pagination
List<Book> books =
        session.createSelectionQuery("from Book where title like ?1 order by title fetch first ?2 rows only", Book
.class)
            .setParameter(1, titlePattern)
            .setParameter(2, MAX_RESULTS)
            .getResultList();
```

Hibernate's `SelectionQuery` has a slightly different way to paginate the query results:

```
List<Book> books =
        session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
            .setParameter(1, titlePattern)
            .setPage(Page.first(MAX_RESULTS))
            .getResultList();
```

The `getResultCount()` method is useful for displaying the number of pages of results:

```
SelectionQuery<Book> query =
        session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
            .setParameter(1, titlePattern);
long results = query.getResultCount();
long pages = results / MAX_RESULTS + (results % MAX_RESULTS == 0 ? 0 : 1);
List<Book> books = query.setMaxResults(MAX_RESULTS).getResultList();
```

Table 6.6: Methods for query limits, pagination, and ordering

Method name	Purpose	JPA-standard
`setMaxResults()`	Set a limit on the number of results returned by a query	✔
`setFirstResult()`	Set an offset on the results returned by a query	✔
`setPage()`	Set the limit and offset by specifying a `Page` object	✖
`getResultCount()`	Determine how many results the query would return in the absence of any limit or offset	✖

It's quite common for pagination to be combined with the need to order query results by a field that's determined at runtime. The `Order` class we just met above provides the ability to specify that the query results should be ordered by one or more fields of the entity type returned by the query:

```
List<Book> books =
        session.createSelectionQuery("from Book where title like ?1", Book.class)
            .setParameter(1, titlePattern)
            .setOrder(List.of(Order.asc(Book_.title), Order.asc(Book_.isbn)))
            .setMaxResults(MAX_RESULTS)
            .getResultList();
```

The approach to pagination we've just seen is sometimes called *offset-based pagination*. Since Hibernate 6.5, there's an alternative approach, which offers some advantages, though it's a little more difficult to use.

6.10. Key-based pagination

Key-based pagination aims to reduce the likelihood of missed or duplicate results when data is modified between page requests. It's most easily illustrated with an example:

```java
String QUERY = "from Book where publicationDate > :minDate";

// obtain the first page of results
KeyedResultList<Book> first =
        session.createSelectionQuery(QUERY, Book.class)
                .setParameter("minDate", minDate)
                .getKeyedResultList(Page.first(25)
                        .keyedBy(Order.asc(Book_.isbn)));
List<Book> firstPage = first.getResultList();
...

if (!firstPage.isLastPage()) {
    // obtain the second page of results
    KeyedResultList<Book> second =
            session.createSelectionQuery(QUERY, Book.class)
                    .setParameter("minDate", minDate))
                    .getKeyedResultList(firstPage.getNextPage());
    List<Book> secondPage = second.getResultList();
    ...
}
```

The "key" in key-based pagination refers to a unique key of the result set which determines a total order on the query results. In this example, `Book.isbn` is the key.

Since this code is a little bit fiddly, key-based pagination works best with generated query or finder methods.

6.11. Representing projection lists

A *projection list* is the list of things that a query returns, that is, the list of expressions in the `select` clause. Since Java has no tuple types, representing query projection lists in Java has always been a problem for JPA and Hibernate. Traditionally, we've just used `Object[]` most of the time:

```java
var results =
        session.createSelectionQuery("select isbn, title from Book", Object[].class)
            .getResultList();

for (var result : results) {
    var isbn = (String) result[0];
    var title = (String) result[1];
    ...
}
```

This is really a bit ugly. Java's `record` types now offer an interesting alternative:

```java
record IsbnTitle(String isbn, String title) {}

var results =
        session.createSelectionQuery("select isbn, title from Book", IsbnTitle.class)
            .getResultList();

for (var result : results) {
    var isbn = result.isbn();
    var title = result.title();
    ...
}
```

Notice that we're able to declare the `record` right before the line which executes the query.

This works just as well with queries written in SQL:

```java
record BookInfo(String isbn, String title, int pages) {}

List<BookInfo> resultList =
        session.createNativeQuery("select title, isbn, pages from Book", BookInfo.class)
            .getResultList();
```

Now, this approach is only *superficially* more typesafe, since the query itself is not checked statically, and so we can't say it's objectively better. But perhaps you find it more aesthetically pleasing. And if we're going to be passing query results around the system, the use of a `record` type is *much* better.

The criteria query API offers a much more satisfying solution to the problem. Consider the following code:

```
var builder = sessionFactory.getCriteriaBuilder();
var query = builder.createTupleQuery();
var book = query.from(Book.class);
var bookTitle = book.get(Book_.title);
var bookIsbn = book.get(Book_.isbn);
var bookPrice = book.get(Book_.price);
query.select(builder.tuple(bookTitle, bookIsbn, bookPrice));
var resultList = session.createSelectionQuery(query).getResultList();
for (var result : resultList) {
    String title = result.get(bookTitle);
    String isbn = result.get(bookIsbn);
    BigDecimal price = result.get(bookPrice);
    ...
}
```

This code is manifestly completely typesafe, and much better than we can hope to do with HQL.

6.12. Named queries

The `@NamedQuery` annotation lets us define a HQL query that is compiled and checked as part of the bootstrap process. This means we find out about errors in our queries earlier, instead of waiting until the query is actually executed. We can place the `@NamedQuery` annotation on any class, even on an entity class.

```
@NamedQuery(name = "10BooksByTitle",
            query = "from Book where title like :titlePattern order by title fetch first 10 rows only")
class BookQueries {}
```

We have to make sure that the class with the `@NamedQuery` annotation will be scanned by Hibernate, either:

- by adding <class>org.hibernate.example.BookQueries</class> to `persistence.xml`, or
- by calling `persistenceConfiguration.managedClass(BookQueries.class)`.

> Unfortunately, JPA's `@NamedQuery` annotation can't be placed on a package descriptor. Therefore, Hibernate provides a very similar annotation, `@org.hibernate.annotations.NamedQuery` which *can* be specified at the package level. If we declare a named query at the package level, we must call:
>
> ```
> configuration.addPackage("org.hibernate.example")
> ```
>
> so that Hibernate knows where to find it.

The `@NamedNativeQuery` annotation lets us do the same for native SQL queries. There's much less advantage to using `@NamedNativeQuery`, because there is very little that Hibernate can do to validate the correctness of a query written in the native SQL dialect of your database.

Table 6.7: Executing named queries

Kind	`Session` method	`EntityManager` method	`Query` execution method
Selection	`createNamedSelectionQuery(String,Class)`	`createNamedQuery(TypedQueryReference)`, `createNamedQuery(String,Class)`	`getResultList()`, `getSingleResult()`, `getSingleResultOrNull()`
Mutation	`createNamedMutationQuery(String)`	`createNamedQuery(TypedQueryReference)`, `createNamedQuery(String)`	`executeUpdate()`

We execute our named query like this:

```
List<Book> books =
        entityManager.createQuery(BookQueries_._10BooksByTitle_)
            .setParameter("titlePattern", titlePattern)
            .getResultList()
```

Here, `BookQueries_._10BooksByTitle_` is an element of the JPA static metamodel of type `TypedQueryReference<Book>`, generated by Hibernate Processor.

Note that the code which executes the named query is not aware of whether the query was written in HQL or in native SQL, making it slightly easier to change and optimize the query later.

It's nice to have our queries checked at startup time. It's even better to have them checked at compile time. In Organizing persistence logic, we mentioned that the Hibernate Processor can do that for us, with the help of the `@CheckHQL` annotation, and we presented that as a reason to use `@NamedQuery`.

> Actually, Hibernate even has a separate Query Validator capable of performing compile-time validation of HQL query strings that occur as arguments to `createQuery()` and friends. If we use the Query Validator, there's not much advantage to the use of named queries.

We're going to learn more about Hibernate Processor in the next chapter.

Chapter 7. Compile-time tooling

The *static metamodel generator* is a standard part of JPA. We've actually already seen its handiwork in the code examples earlier: it's the author of the class `Book_`, which contains the static metamodel of the entity class `Book`.

Hibernate comes with an annotation processor which does much more than just this. It's capable of automatically generating:

- JPA metamodel classes, as we've already seen,
- Jakarta Data metamodel classes,
- static query methods and finder methods, and
- implementations of repository interfaces, including Jakarta Data repositories.

Hibernate Processor

Hibernate Processor, the annotation processor formerly known as the Metamodel Generator, began its life as a code generator for what JPA calls a *static metamodel*. That is, it produces a typed model of the persistent classes in our program, giving us a type safe way to refer to their attributes in Java code. In particular, it lets us specify entity graphs and criteria queries in a completely type-safe way.

The history behind this thing is quite interesting. Back when Java's annotation processing API was brand spankin' new, the static metamodel for JPA was proposed by Gavin King for inclusion in JPA 2.0, as a way to achieve type safety in the nascent criteria query API. It's fair to say that, back in 2010, this API was not a runaway success. Tools did not, at the time, feature robust support for annotation processors. And all the explicit generic types made user code quite verbose and difficult to read. (The need for an explicit reference to a `CriteriaBuilder` instance also contributed verbosity to the criteria API.) For years, Gavin counted this as one of his more embarrassing missteps.

But time has been kind to the static metamodel. By now, all Java compilers, build tools, and IDEs have robust support for annotation processing, and Java's local type inference (the `var` keyword) eliminates the verbose generic types. JPA's `CriteriaBuilder` and `EntityGraph` APIs are still not quite perfect, but the imperfections aren't related to static type safety or annotation processing. The static metamodel itself is undeniably useful and elegant.

And it turns out that there was quite a lot of unlocked potential there. Since Hibernate 6.3 the Processor has started taking on a much bigger role. Today, it even contains a complete implementation of the Jakarta Data specification.

Now, you still don't have to use the Hibernate Processor with Hibernate—the APIs we just mentioned still also accept plain strings—but we find that it works well with Gradle and integrates smoothly with our IDE, and the advantage in type-safety is compelling.

We've already seen how to set up the annotation processor in the Gradle build we saw earlier. For more details on how to integrate the Hibernate Processor, check out the Static Metamodel Generator section in the User Guide.

7.1. The static metamodel

We've already seen several ways to use the JPA static metamodel. Metamodel references are useful for expressing, in a completely type-safe way:

- Criteria queries,
- eager fetching via an entity graph,
- dynamic restriction and sorting, and
- references to named queries and named entity graphs.

Here's an example of the sort of code that's generated for an entity class, as mandated by the JPA specification:

Generated Code

```
@StaticMetamodel(Book.class)
public abstract class Book_ {

    /**
     * @see org.example.Book#isbn
     **/
    public static volatile SingularAttribute<Book, String> isbn;

    /**
     * @see org.example.Book#text
     **/
```

```java
    public static volatile SingularAttribute<Book, String> text;

    /**
     * @see org.example.Book#title
     **/
    public static volatile SingularAttribute<Book, String> title;

    /**
     * @see org.example.Book#type
     **/
    public static volatile SingularAttribute<Book, Type> type;

    /**
     * @see org.example.Book#publicationDate
     **/
    public static volatile SingularAttribute<Book, LocalDate> publicationDate;

    /**
     * @see org.example.Book#publisher
     **/
    public static volatile SingularAttribute<Book, Publisher> publisher;

    /**
     * @see org.example.Book#authors
     **/
    public static volatile SetAttribute<Book, Author> authors;

    public static final String ISBN = "isbn";
    public static final String TEXT = "text";
    public static final String TITLE = "title";
    public static final String TYPE = "type";
    public static final String PUBLICATION_DATE = "publicationDate";
    public static final String PUBLISHER = "publisher";
    public static final String AUTHORS = "authors";

}
```

For each attribute of the entity, the `Book_` class has:

1. a `String`-valued constant like `TITLE` , and
2. a typesafe reference like `title` to a metamodel object of type `Attribute`.

> Hibernate Processor allows *statically-typed* access to elements of the JPA `Metamodel`. But the `Metamodel` is also accessible in a "reflective" way, via the `EntityManagerFactory`.
>
> ```java
> EntityType<Book> book = entityManagerFactory.getMetamodel().entity(Book.class);
> SingularAttribute<Book,Long> id = book.getDeclaredId(Long.class)
> ```
>
> This is very useful for writing generic code in frameworks or libraries. For example, you could use it to create your own criteria query API.

The JPA static metamodel for an entity also contains members representing the named queries and named entity graphs declared by `@NamedQuery`, `@NamedNativeQuery`, and `@NamedEntityGraph` annotations of the entity class.

For example, if we had:

```java
@CheckHQL // validate named queries at compile time
@NamedQuery(name = "findBooksByTitle",
            query = "from Book where title like :title order by title")
@Entity
class Book { ... }
```

Then we may execute the query as follows:

```java
var books =
        entityManager.createNamedQuery(Queries_._findBooksByTitle_)
                .setParameter("title", titlePattern)
                .setPage(page)
```

```
                    .getResultList();
```

Notice that no typecast was required here, since the generated code embeds the return type of the query as a type argument of the JPA `TypedQueryReference`:

```
/**
 * @see #_findBooksByTitle_
 **/
public static final String QUERY_FIND_BOOKS_BY_TITLE = "findBooksByTitle";


/**
 * The query named {@value QUERY_FIND_BOOKS_BY_TITLE}
 * <pre>
 * from Book where title like :title order by title
 * </pre>
 *
 * @see org.example.Book
 **/
public static volatile TypedQueryReference<Book> _findBooksByTitle_;
```

> Actually, Hibernate Processor doesn't require that such annotations be applied to the entity class itself, as we already saw earlier.

We've already been using metamodel references like `Book_.authors` and `Book.AUTHORS` in the previous chapters. So now let's see what else Hibernate Processor can do for us.

7.2. Finder methods, query methods, and repositories

Automatic generation of *finder methods* and *query methods* is a relatively new feature of Hibernate Processor—originally introduced as an experiment—which ultimately grew into a whole new way to use Hibernate.

We're going to meet three different kinds of generated method:

- a *named query method* has its signature and implementation generated directly from a `@NamedQuery` annotation,
- a *query method* has a signature that's explicitly declared, and a generated implementation which executes a HQL or SQL query specified via a `@HQL` or `@SQL` annotation, and
- a *finder method* annotated `@Find` has a signature that's explicitly declared, and a generated implementation inferred from the parameter list.

We're also going to see two ways that these methods can be called:

- as static methods of a generated abstract class, or
- as instance methods of an interface with a generated implementation which may even be injected.

Back in Organizing persistence logic, we walked you through a few different ways to organize your code with the help of Hibernate Processor. That journey terminated at the idea of a repository, but we emphasized that you aren't required to stay all the way to the end of the line. Repositories are a sweet spot for many users, but they might not be your sweet spot, and that's OK. Hibernate Processor is perfectly happy to generate `static` implementations of `@HQL`, `@SQL`, and `@Find` methods, eliminating the need to inject or instantiate a repository object.

Hibernate Processor and Jakarta Data

The functionality we're about to describe was developed before Jakarta Data took on its current shape, and directly triggered the apocalypse which lead to the final form of the specification. Therefore, there's massive overlap between the functionality described in this chapter, and the functionality available via the Jakarta Data annotations. On the other hand, Jakarta Data can't do *everything* described below, and in particular it doesn't yet come with built-in support for stateful persistence contexts or reactive sessions.

We've therefore opted *not* to rewrite this chapter in a Jakarta Data-centric way, and instead refer you to Introducing Hibernate Data Repositories for information about the standard Jakarta Data APIs.

As Jakarta Data matures, even more of this functionality might be made obsolete, at least in the form described here. We're working hard to make that happen.

> The functionality described in the rest of this chapter depends on the use of the annotations described in Entities.

> Hibernate Processor is not currently able to generate finder methods and query methods for entities declared completely in XML, and it's not able to validate HQL which queries such entities. (On the other hand, the O/R mappings may be specified in XML, since they're not needed by the Processor.)

To whet our appetites, let's see how it works for a `@NamedQuery`.

7.3. Named queries and Hibernate Processor

The very simplest way to generate a query method is to put a `@NamedQuery` annotation anywhere we like, with a `name` beginning with the magical character `#`.

Let's just stick it on the `Book` class:

```
@CheckHQL // validate the query at compile time
@NamedQuery(name = "#findByTitleAndType",
            query = "select book from Book book where book.title like :title and book.type = :type")
@Entity
public class Book { ... }
```

Now the Processor adds the following method declaration to the metamodel class `Book_`.

Generated Code

```
/**
 * Execute named query {@value #QUERY_FIND_BY_TITLE_AND_TYPE} defined by annotation of {@link Book}.
 **/
public static List<Book> findByTitleAndType(@Nonnull EntityManager entityManager, String title, Type type) {
    return entityManager.createNamedQuery(QUERY_FIND_BY_TITLE_AND_TYPE)
            .setParameter("title", title)
            .setParameter("type", type)
            .getResultList();
}
```

We can easily call this method from wherever we like, as long as we have access to an `EntityManager`:

```
List<Book> books =
        Book_.findByTitleAndType(entityManager, titlePattern, Type.BOOK);
```

Now, this is quite nice, but it's a bit inflexible in various ways, and so this probably *isn't* the best way to generate a query method.

7.4. Generated query methods

The principal problem with generating the query method straight from the `@NamedQuery` annotation is that it doesn't let us explicitly specify the return type or parameter list. In the case we just saw, Hibernate Processor does a reasonable job of inferring the query return type and parameter types, but we're often going to need a bit more control.

The solution is to write down the signature of the query method *explicitly*, as an abstract method in Java. We'll need a place to put this method, and since our `Book` entity isn't an abstract class, we'll just introduce a new interface for this purpose:

```
interface Queries {
    @HQL("where title like :title and type = :type")
    List<Book> findBooksByTitleAndType(String title, String type);
}
```

Instead of `@NamedQuery`, which is a type-level annotation, we specify the HQL query using the new `@HQL` annotation, which we place directly on the query method. This results in the following generated code in the `Queries_` class:

Generated Code

```
@StaticMetamodel(Queries.class)
public abstract class Queries_ {

    /**
     * Execute the query {@value #FIND_BOOKS_BY_TITLE_AND_TYPE_String_Type}.
     *
     * @see org.example.Queries#findBooksByTitleAndType(String,Type)
     **/
```

```
    public static List<Book> findBooksByTitleAndType(@Nonnull EntityManager entityManager, String title, Type type) {
        return entityManager.createQuery(FIND_BOOKS_BY_TITLE_AND_TYPE_String_Type, Book.class)
                .setParameter("title", title)
                .setParameter("type", type)
                .getResultList();
    }

    static final String FIND_BOOKS_BY_TITLE_AND_TYPE_String_Type =
            "where title like :title and type = :type";

}
```

Notice that the signature differs just slightly from the one we wrote down in the `Queries` interface: the Processor has prepended a parameter accepting `EntityManager` to the parameter list.

If we want to explicitly specify the name and type of this parameter, we may declare it explicitly:

```
interface Queries {
    @HQL("where title like :title and type = :type")
    List<Book> findBooksByTitleAndType(StatelessSession session, String title, String type);
}
```

Hibernate Processor defaults to using `EntityManager` as the session type, but other types are allowed:

- `Session`,
- `StatelessSession`, or
- `Mutiny.Session` or `Mutiny.StatelessSession` from Hibernate Reactive.

The real value of all this is in the checks which can now be done at compile time. Hibernate Processor verifies that the parameters of our abstract method declaration match the parameters of the HQL query, for example:

- for a named parameter `:alice`, there must be a method parameter named `alice` with exactly the same type, or
- for an ordinal parameter `?2`, the second method parameter must have exactly the same type.

The query must also be syntactically legal and semantically well-typed, that is, the entities, attributes, and functions referenced in the query must actually exist and have compatible types. Hibernate Processor determines this by inspecting the annotations of the entity classes at compile time.

> The `@CheckHQL` annotation which instructs Hibernate to validate named queries is *not* necessary for query methods annotated `@HQL`.

The `@HQL` annotation has a friend named `@SQL` which lets us specify a query written in native SQL instead of in HQL. In this case there's a lot less the Processor can do to check that the query is legal and well-typed.

We imagine you're wondering whether a `static` method is really the right thing to use here.

7.5. Generating query methods as instance methods

One thing not to like about what we've just seen is that we can't transparently replace a generated `static` function of the `Queries_` class with an improved handwritten implementation without impacting clients. Now, if our query is only called in one place, which is quite common, this isn't going to be a big issue, and so we're inclined to think the `static` function is fine.

But if this function is called from many places, it's probably better to promote it to an instance method of some class or interface. Fortunately, this is straightforward.

All we need to do is add an abstract getter method for the session object to our `Queries` interface. (And remove the session from the method parameter list.) We may call this method anything we like:

```
interface Queries {
    EntityManager entityManager();

    @HQL("where title like :title and type = :type")
    List<Book> findBooksByTitleAndType(String title, String type);
}
```

Here we've used `EntityManager` as the session type, but other types are allowed, as we saw above.

Now Hibernate Processor does something a bit different:

Generated Code

```
@StaticMetamodel(Queries.class)
public class Queries_ implements Queries {

    private final @Nonnull EntityManager entityManager;

    public Queries_(@Nonnull EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    public @Nonnull EntityManager entityManager() {
        return entityManager;
    }

    /**
     * Execute the query {@value #FIND_BOOKS_BY_TITLE_AND_TYPE_String_Type}.
     *
     * @see org.example.Queries#findBooksByTitleAndType(String,Type)
     **/
    @Override
    public List<Book> findBooksByTitleAndType(String title, Type type) {
        return entityManager.createQuery(FIND_BOOKS_BY_TITLE_AND_TYPE_String_Type, Book.class)
                .setParameter("title", title)
                .setParameter("type", type)
                .getResultList();
    }

    static final String FIND_BOOKS_BY_TITLE_AND_TYPE_String_Type =
            "where title like :title and type = :type";

}
```

The generated class `Queries_` now implements the `Queries` interface, and the generated query method implements our abstract method directly.

Of course, the protocol for calling the query method has to change:

```
Queries queries = new Queries_(entityManager);
List<Book> books = queries.findByTitleAndType(titlePattern, Type.BOOK);
```

If we ever need to swap out the generated query method with one we write by hand, without impacting clients, all we need to do is replace the abstract method with a `default` method of the `Queries` interface. For example:

```
interface Queries {
    EntityManager entityManager();

    // handwritten method replacing previous generated implementation
    default List<Book> findBooksByTitleAndType(String title, String type) {
        entityManager()
                .createQuery("where title like :title and type = :type", Book.class)
                        .setParameter("title", title)
                        .setParameter("type", type)
                        .setFlushMode(COMMIT)
                        .setMaxResults(100)
                        .getResultList();
    }
}
```

What if we would like to inject a `Queries` object instead of calling its constructor directly?

As you recall, we don't think these things really need to be container-managed objects. But if you *want* them to be—if you're allergic to calling constructors, for some reason—then:

- placing `jakarta.inject` on the build path will cause an `@Inject` annotation to be added to the constructor of `Queries_`, and
- placing `jakarta.enterprise.context` on the build path will cause a `@Dependent` annotation to be added to the `Queries_` class.

Thus, the generated implementation of `Queries` will be a perfectly functional CDI bean with no extra work to be done.

Is the `Queries` interface starting to look a lot like a DAO-style repository object? Well, perhaps. You can certainly *decide to use* this facility to create a `BookRepository` if that's what you prefer. But unlike a repository, our `Queries` interface:

- doesn't attempt to hide the `EntityManager` from its clients,
- doesn't implement or extend any framework-provided interface or abstract class, at least not unless you want to create such a framework yourself, and
- isn't restricted to service a particular entity class.

We can have as many or as few interfaces with query methods as we like. There's no one-one-correspondence between these interfaces and entity types. This approach is so flexible that we don't even really know what to call these "interfaces with query methods".

7.6. Generated finder methods

At this point, one usually begins to question whether it's even necessary to write a query at all. Would it be possible to just infer the query from the method signature?

In some simple cases it's indeed possible, and this is the purpose of *finder methods*. A finder method is a method annotated `@Find`. For example:

```
@Find
Book getBook(String isbn);
```

A finder method may have multiple parameters:

```
@Find
List<Book> getBooksByTitle(String title, Type type);
```

The name of the finder method is arbitrary and carries no semantics. But:

- the return type determines the entity class to be queried, and
- the parameters of the method must match the fields of the entity class *exactly*, by both name and type.

Considering our first example, `Book` has a persistent field `String isbn`, so this finder method is legal. If there were no field named `isbn` in `Book`, or if it had a different type, this method declaration would be rejected with a meaningful error at compile time. Similarly, the second example is legal, since `Book` has fields `String title` and `Type type`.

You might notice that our solution to this problem is very different from the approach taken by others. In DAO-style repository frameworks, you're asked to encode the semantics of the finder method into the *name of the method*. This idea came to Java from Ruby, and we think it doesn't belong here. It's completely unnatural in Java, and by almost any measure other than *counting characters* it's objectively worse than just writing the query in a string literal. At least string literals accommodate whitespace and punctuation characters. Oh and, you know, it's pretty useful to be able to rename a finder method *without changing its semantics*. 🙄

The code generated for this finder method depends on what kind of fields match the method parameters:

`@Id` field	Uses `EntityManager.find()`
All `@NaturalId` fields	Uses `Session.byNaturalId()`
Other persistent fields, or a mix of field types	Uses a criteria query

The generated code also depends on what kind of session we have, since the capabilities of stateless sessions, and of reactive sessions, differ slightly from the capabilities of regular stateful sessions.

With `EntityManager` as the session type, we obtain:

```java
/**
 * Find {@link Book} by {@link Book#isbn isbn}.
 *
 * @see org.example.Dao#getBook(String)
 **/
@Override
public Book getBook(@Nonnull String isbn) {
    return entityManager.find(Book.class, isbn);
}

/**
 * Find {@link Book} by {@link Book#title title} and {@link Book#type type}.
 *
 * @see org.example.Dao#getBooksByTitle(String,Type)
 **/
@Override
public List<Book> getBooksByTitle(String title, Type type) {
    var builder = entityManager.getEntityManagerFactory().getCriteriaBuilder();
    var query = builder.createQuery(Book.class);
    var entity = query.from(Book.class);
    query.where(
            title==null
                ? entity.get(Book_.title).isNull()
                : builder.equal(entity.get(Book_.title), title),
            type==null
                ? entity.get(Book_.type).isNull()
                : builder.equal(entity.get(Book_.type), type)
    );
    return entityManager.createQuery(query).getResultList();
}
```

It's even possible to match a parameter of a finder method against a property of an associated entity or embeddable. The natural syntax would be a parameter declaration like `String publisher.name`, but because that's not legal Java, we can write it as `String publisher$name`, taking advantage of a legal Java identifier character that nobody ever uses for anything else:

```java
@Find
List<Book> getBooksByPublisherName(String publisher$name);
```

The `@Pattern` annotation may be applied to a parameter of type `String`, indicating that the argument is a wildcarded pattern which will be compared using `like`.

```java
@Find
List<Book> getBooksByTitle(@Pattern String title, Type type);
```

Even better, a parameter may be of type `Range<T>`, where `T` is the type of the matching field.

```java
@Find
List<Book> getBooksByTitle(Range<String> title, Type type);
```

The `Range` interface has a variety of `static` methods the caller may use to construct different kinds of ranges. For example, `Range.pattern()` constructs a `Range` representing a pattern.

```java
List<Book> books =
        // returns books with titles beginning with "hibernate"
        queries.getBooksByTitle(Range.prefix("hibernate", false), type);
```

A finder method may specify fetch profiles, for example:

```java
@Find(namedFetchProfiles=Book_.FETCH_WITH_AUTHORS)
Book getBookWithAuthors(String isbn);
```

This lets us declare which associations of `Book` should be pre-fetched by annotating the `Book` class.

7.7. Paging, ordering, and restrictions

Optionally, a query method—or a finder method which returns multiple results—may have additional "magic" parameters which do not map to query parameters:

Parameter type	Purpose	Example argument
`Page`	Specifies a page of query results	`Page.first(20)`
`Order<? super E>`	Specifies an entity attribute to order by, if `E` is the entity type returned by the query	`Order.asc(Book_.title)`
`List<Order? super E>` (or varargs)	Specifies entity attributes to order by, if `E` is the entity type returned by the query	`List.of(Order.asc(Book_.title), Order.asc(Book_.isbn))`
`Order<Object[]>`	Specifies a column to order by, if the query returns a projection list	`Order.asc(1)`
`List<Object[]>` (or varargs)	Specifies columns to order by, if the query returns a projection list	`List.of(Order.asc(1), Order.desc(2))`
`Restriction<? super E>`	Specifies a restriction used to filter query results	`Restriction.startsWith("Hibernate")`

Thus, if we redefine our earlier query method as follows:

```
interface Queries {
    @HQL("from Book where title like :title and type = :type")
    List<Book> findBooksByTitleAndType(String title, Type type,
                                       Page page, Order<? super Book>... order);
}
```

Then we can call it like this:

```
List<Book> books =
        Queries_.findBooksByTitleAndType(entityManager, titlePattern, Type.BOOK,
                Page.page(RESULTS_PER_PAGE, page), Order.asc(Book_.isbn));
```

Alternatively, we could have written this query method as a finder method:

```
interface Queries {
    @Find
    List<Book> getBooksByTitle(String title, Type type,
                               Page page, Order<? super Book>... order);
}
```

Similarly, we may define a query method which accepts an arbitrary `Restriction`:

```
interface Queries {
    @Find
    List<Book> findBooks(Restriction<? super Book> restriction, Order<? super Book>... order);
}
```

As we saw earlier, the `Restriction` interface has a variety of `static` methods for constructing restrictions.

```
List<Book> books =
        // returns books with titles beginning with "hibernate", sorted by title
        queries.findBooks(Restriction.startsWith(Book_.title, "hibernate", false),
                          Order.asc(Book_.title));
```

This gives some dynamic control over query execution. We'll see below that it's even possible for the caller to gain direct control over the `Query` object.

7.8. Key-based pagination

A generated query or finder method can make use of key-based pagination.

```
@Query("where publicationDate > :minDate")
KeyedResultList<Book> booksFromDate(Session session, LocalDate minDate, KeyedPage<Book> page);
```

Note that this method:

- accepts a `KeyedPage`, and
- returns `KeyedResultList`.

Such a method may be used like this:

```
// obtain the first page of results
KeyedResultList<Book> first =
        Queries_.booksFromDate(session, minDate,
                Page.first(25).keyedBy(Order.asc(Book_.isbn)));
List<Book> firstPage = first.getResultList();
...

if (!firstPage.isLastPage()) {
    // obtain the second page of results
    KeyedResultList<Book> second =
            Queries_.booksFromDate(session, minDate,
                    firstPage.getNextPage());
    List<Book> secondPage = second.getResultList();
    ...
}
```

7.9. Query and finder method return types

A query method doesn't need to return `List`. It might return a single `Book`.

```
@HQL("where isbn = :isbn")
Book findBookByIsbn(String isbn);
```

For a query with a projection list, `Object[]` or `List<Object[]>` is permitted:

```
@HQL("select isbn, title from Book where isbn = :isbn")
Object[] findBookAttributesByIsbn(String isbn);
```

But when there's just one item in the `select` list, the type of that item should be used:

```
@HQL("select title from Book where isbn = :isbn")
String getBookTitleByIsbn(String isbn);
```

```
@HQL("select local datetime")
LocalDateTime getServerDateTime();
```

A query which returns a selection list may have a query method which repackages the result as a record, as we saw in Representing projection lists.

```
record IsbnTitle(String isbn, String title) {}

@HQL("select isbn, title from Book")
List<IsbnTitle> listIsbnAndTitleForEachBook(Page page);
```

A query method might even return `TypedQuery` or `SelectionQuery`:

```
@HQL("where title like :title")
SelectionQuery<Book> findBooksByTitle(String title);
```

This is extremely useful at times, since it allows the client to further manipulate the query:

```
List<Book> books =
        Queries_.findBooksByTitle(entityManager, titlePattern)
            .setOrder(Order.asc(Book_.title))                   // order the results
```

```
                .setPage(Page.page(RESULTS_PER_PAGE, page))          // return the given page of results
                .setFlushMode(FlushModeType.COMMIT)                  // don't flush session before query execution
                .setReadOnly(true)                                   // load the entities in read-only mode
                .setCacheStoreMode(CacheStoreMode.BYPASS)            // don't cache the results
                .setComment("Hello world!")                          // add a comment to the generated SQL
                .getResultList();
```

An `insert`, `update`, or `delete` query must return `int`, `boolean`, or `void`.

```
@HQL("delete from Book")
int deleteAllBooks();
```

```
@HQL("update Book set discontinued = true where discontinued = false and isbn = :isbn")
boolean discontinueBook(String isbn);
```

```
@HQL("update Book set discontinued = true where isbn = :isbn")
void discontinueBook(String isbn);
```

On the other hand, finder methods are currently much more limited. A finder method must return an entity type like `Book`, or a list of the entity type, `List<Book>`, for example.

> As you might expect, for a reactive session, all query methods and finder methods must return `Uni`.

7.10. An alternative approach

What if you just don't like the ideas we've presented in this chapter, preferring to call the `Session` or `EntityManager` directly, but you still want compile-time validation for HQL? Or what if you *do* like the ideas, but you're working on a huge existing codebase full of code you don't want to change?

Well, there's a solution for you, too. The Query Validator is a separate annotation processor that's capable of type-checking HQL strings, not only in annotations, but even when they occur as arguments to `createQuery()`, `createSelectionQuery()`, or `createMutationQuery()`. It's even able to check calls to `setParameter()`, with some restrictions.

The Query Validator works in `javac`, Gradle, Maven, and the Eclipse Java Compiler.

> Unlike Hibernate Processor, which is a completely bog-standard Java annotation processor based on only standard Java APIs, the Query Validator makes use of internal compiler APIs in `javac` and `ecj`. This means it can't be guaranteed to work in every Java compiler. The current release is known to work in JDK 11 and above, though JDK 15 or above is preferred.

Chapter 8. Tuning and performance

Once you have a program up and running using Hibernate to access the database, it's inevitable that you'll find places where performance is disappointing or unacceptable.

Fortunately, most performance problems are relatively easy to solve with the tools that Hibernate makes available to you, as long as you keep a couple of simple principles in mind.

First and most important: the reason you're using Hibernate is that it makes things easier. If, for a certain problem, it's making things *harder*, stop using it. Solve this problem with a different tool instead.

> Just because you're using Hibernate in your program doesn't mean you have to use it *everywhere*.

Second: there are two main potential sources of performance bottlenecks in a program that uses Hibernate:

- too many round trips to the database, and
- memory consumption associated with the first-level (session) cache.

So performance tuning primarily involves reducing the number of accesses to the database, and/or controlling the size of the session cache.

But before we get to those more advanced topics, we should start by tuning the connection pool.

8.1. Tuning the connection pool

The connection pool built in to Hibernate is suitable for testing, but isn't intended for use in production. Instead, Hibernate supports several different connection pools, including our favorite, Agroal.

Hibernate will automatically make use of `AgroalConnectionProvider` if the module `org.hibernate.orm:hibernate-agroal` is available at runtime. So just add it as a runtime dependency, and you're all set.

Well, actually, that's a bit fragile, since Hibernate silently falls back to using the default connection pool if Agroal happens to be missing at runtime. Perhaps it's better to set this configuration property:

Configuration property name	Purpose
`hibernate.connection.provider_class`	Explicitly specify a connection pool, for example, `agroal`, `hikaricp` or `c3p0`.

> You can set `hibernate.connection.provider_class` to `agroal` so that Hibernate fails at startup if Agroal is missing.

To properly configure Agroal, you'll need to set some extra configuration properties, in addition to the settings we already saw in Basic configuration settings. Properties with the prefix `hibernate.agroal` are passed through to Agroal:

```
# configure Agroal connection pool
hibernate.agroal.maxSize 20
hibernate.agroal.minSize 10
hibernate.agroal.acquisitionTimeout PT1s
hibernate.agroal.reapTimeout PT10s
```

There are many to choose from, as enumerated by `AgroalSettings`:

Table 8.2: Settings for configuring Agroal

Configuration property name	Purpose
`hibernate.agroal.maxSize`	The maximum number of connections present on the pool
`hibernate.agroal.minSize`	The minimum number of connections present on the pool
`hibernate.agroal.initialSize`	The number of connections added to the pool when it is started
`hibernate.agroal.maxLifetime`	The maximum amount of time a connection can live, after which it is removed from the pool
`hibernate.agroal.acquisitionTimeout`	The maximum amount of time a thread can wait for a connection, after which an exception is thrown instead

Configuration property name	Purpose
`hibernate.agroal.reapTimeout`	The duration for eviction of idle connections
`hibernate.agroal.leakTimeout`	The duration of time a connection can be held without causing a leak to be reported
`hibernate.agroal.idleValidationTimeout`	A foreground validation is executed if a connection has been idle on the pool for longer than this duration
`hibernate.agroal.validationTimeout`	The interval between background validation checks
`hibernate.agroal.initialSql`	A SQL command to be executed when a connection is created

The following settings are common to all connection pools supported by Hibernate:

Table 8.3: Common settings for connection pools

`hibernate.connection.pool_size`	The size of the connection pool
`hibernate.connection.autocommit`	The default autocommit mode
`hibernate.connection.isolation`	The default transaction isolation level

A popular alternative to Agroal is HikariCP. Integration with HikariCP is provided by the module `org.hibernate.orm:hibernate-hikaricp`. Its settings are enumerated by `HikariCPSettings`.

Container-managed datasources

In a container environment, you usually don't need to configure a connection pool through Hibernate. Instead, you'll use a container-managed datasource, as we saw in Basic configuration settings.

A related important setting is the default JDBC fetch size.

8.2. JDBC fetch size

The *JDBC fetch size* controls the maximum number of rows the JDBC driver fetches from the database in one round trip. In Hibernate we usually limit query result sets using pagination, and so we almost always prefer that the JDBC driver fetch the whole query result set in one trip. Most JDBC drivers accommodate this usage pattern by *not* setting a default fetch size. However, there are a couple of exceptions to this and for the offending drivers you should probably override the default fetch size using the following configuration property.

Table 8.4: Default JDBC fetch size

`hibernate.jdbc.fetch_size`	The default JDBC fetch size

The default fetch size can be overridden for a given query by calling `setFetchSize()`, but this is rarely necessary.

The Oracle JDBC driver defaults to a JDBC fetch size of 10. You should *always* set `hibernate.jdbc.fetch_size` explicitly if you're using Oracle, or, even better, specify the parameter `defaultRowPrefetch` in the JDBC connection URL.

The MySQL JDBC driver ignores the fetch size by default. The JDBC connection property `useCursorFetch=true` enables the use of server-side cursors, and with this setting the driver respects the fetch size.

8.3. Enabling statement batching

An easy way to improve performance of some transactions, with almost no work at all, is to turn on automatic DML statement batching. Batching only helps in cases where a program executes many inserts, updates, or deletes against the same table in a single transaction.

All we need to do is set a single property:

Table 8.5: Enabling JDBC batching

Configuration property name	Purpose	Alternative
`hibernate.jdbc.batch_size`	Maximum batch size for SQL statement batching	`setJdbcBatchSize()`

To confirm that statement batching is working, enable `TRACE`-level logging for the category `org.hibernate.orm.jdbc.batch`.

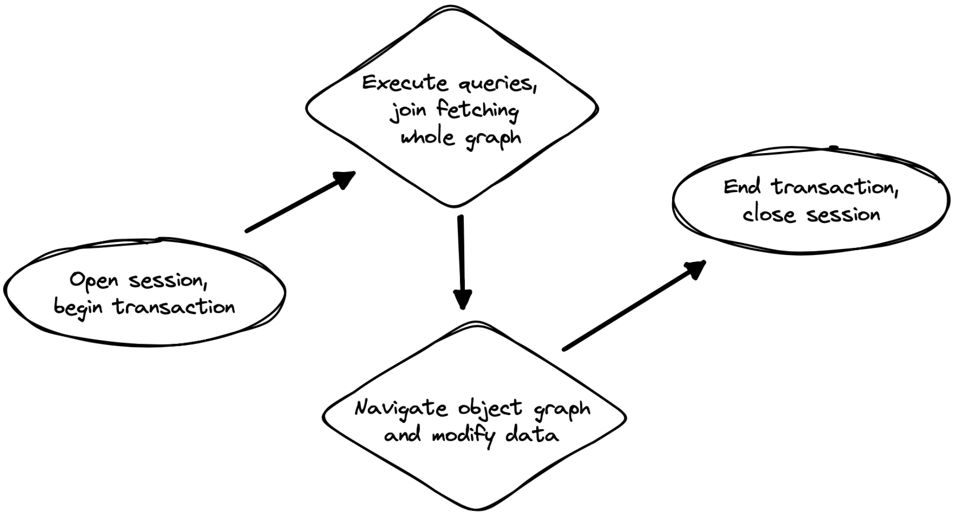
That said, batching is rarely the most convenient or most efficient way to update or delete many rows at once.

> Even better than DML statement batching is the use of HQL `update` or `delete` queries, or even native SQL that calls a stored procedure!

8.4. Association fetching

Achieving high performance in ORM means minimizing the number of round trips to the database. This goal should be uppermost in your mind whenever you're writing data access code with Hibernate. The most fundamental rule of thumb in ORM is:

- explicitly specify all the data you're going to need right at the start of a session/transaction, and fetch it immediately in one or two queries,
- and only then start navigating associations between persistent entities.



Without question, the most common cause of poorly-performing data access code in Java programs is the problem of *N+1 selects*. Here, a list of *N* rows is retrieved from the database in an initial query, and then associated instances of a related entity are fetched using *N* subsequent queries.

> This isn't a bug or limitation of Hibernate; this problem even affects typical handwritten JDBC code behind DAOs. Only you, the developer, can solve this problem, because only you know ahead of time what data you're going to need in a given unit of work. But that's OK. Hibernate gives you all the tools you need.

In this section we're going to discuss different ways to avoid such "chatty" interaction with the database.

Hibernate provides several strategies for efficiently fetching associations and avoiding *N+1* selects:

- *outer join fetching*—where an association is fetched using a `left outer join`,
- *batch fetching*—where an association is fetched using a subsequent `select` with a batch of primary keys, and
- *subselect fetching*—where an association is fetched using a subsequent `select` with keys re-queried in a subselect.

Of these, you should almost always use outer join fetching. But let's consider the alternatives first.

8.5. Batch fetching and subselect fetching

Consider the following code:

```
List<Book> books =
        session.createSelectionQuery("from Book order by isbn", Book.class)
            .getResultList();
books.forEach(book -> book.getAuthors().forEach(author -> out.println(book.title + " by " + author.name)));
```

This code is *very* inefficient, resulting, by default, in the execution of *N+1* `select` statements, where *N* is the number of `Books`.

Let's see how we can improve on that.

SQL for batch fetching

With batch fetching enabled, Hibernate might execute the following SQL on PostgreSQL:

```
/* initial query for Books */
select b1_0.isbn,b1_0.price,b1_0.published,b1_0.publisher_id,b1_0.title
from Book b1_0
order by b1_0.isbn

/* first batch of associated Authors */
select a1_0.books_isbn,a1_1.id,a1_1.bio,a1_1.name
from Book_Author a1_0
    join Author a1_1 on a1_1.id=a1_0.authors_id
where a1_0.books_isbn = any (?)

/* second batch of associated Authors */
select a1_0.books_isbn,a1_1.id,a1_1.bio,a1_1.name
from Book_Author a1_0
    join Author a1_1 on a1_1.id=a1_0.authors_id
where a1_0.books_isbn = any (?)
```

The first `select` statement queries and retrieves `Books`. The second and third queries fetch the associated `Authors` in batches. The number of batches required depends on the configured *batch size*. Here, two batches were required, so two SQL statements were executed.

> The SQL for batch fetching looks slightly different depending on the database. Here, on PostgreSQL, Hibernate passes a batch of primary key values as a SQL `ARRAY`.

SQL for subselect fetching

On the other hand, with subselect fetching, Hibernate would execute this SQL:

```
/* initial query for Books */
select b1_0.isbn,b1_0.price,b1_0.published,b1_0.publisher_id,b1_0.title
from Book b1_0
order by b1_0.isbn

/* fetch all associated Authors */
select a1_0.books_isbn,a1_1.id,a1_1.bio,a1_1.name
from Book_Author a1_0
    join Author a1_1 on a1_1.id=a1_0.authors_id
where a1_0.books_isbn in (select b1_0.isbn from Book b1_0)
```

Notice that the first query is re-executed in a subselect in the second query. The execution of the subselect is likely to be relatively inexpensive, since the data should already be cached by the database. Clever, huh?

Enabling the use of batch or subselect fetching

Both batch fetching and subselect fetching are disabled by default, but we may enable one or the other globally using properties.

Table 8.6: Configuration settings to enable batch and subselect fetching

Configuration property name	Property value	Alternatives
`hibernate.default_batch_fetch_size`	A sensible batch size >1 to enable batch fetching	`@BatchSize()`, `setFetchBatchSize()`
`hibernate.use_subselect_fetch`	`true` to enable subselect fetching	`@Fetch(SUBSELECT)`, `setSubselectFetchingEnabled()`

Alternatively, we can enable one or the other in a given session:

```
session.setFetchBatchSize(5);
session.setSubselectFetchingEnabled(true);
```

We may request subselect fetching more selectively by annotating a collection or many-valued association with the `@Fetch` annotation.

```
@ManyToMany @Fetch(SUBSELECT)
Set<Author> authors;
```

Note that `@Fetch(SUBSELECT)` has the same effect as `@Fetch(SELECT)`, except after execution of a HQL or criteria query. But after query execution, `@Fetch(SUBSELECT)` is able to much more efficiently fetch associations.

Later, we'll see how we can use fetch profiles to do this even more selectively.

That's all there is to it. Too easy, right?

Sadly, that's not the end of the story. While batch fetching might *mitigate* problems involving N+1 selects, it won't solve them. The truly correct solution is to fetch associations using joins. Batch fetching (or subselect fetching) can only be the *best* solution in rare cases where outer join fetching would result in a cartesian product and a huge result set.

But batch fetching and subselect fetching have one important characteristic in common: they can be performed *lazily*. This is, in principle, pretty convenient. When we query data, and then navigate an object graph, lazy fetching saves us the effort of planning ahead. It turns out that this is a convenience we're going to have to surrender.

8.6. Join fetching

Outer join fetching is usually the best way to fetch associations, and it's what we use most of the time. Unfortunately, by its very nature, join fetching simply can't be lazy. So to make use of join fetching, we must plan ahead. Our general advice is:

Avoid the use of lazy fetching, which is often the source of N+1 selects.

Now, we're not saying that associations should be mapped for eager fetching by default! That would be a terrible idea, resulting in simple session operations that fetch almost the entire database. Therefore:

Most associations should be mapped for lazy fetching by default.

It sounds as if this tip is in contradiction to the previous one, but it's not. It's saying that you must explicitly specify eager fetching for associations precisely when and where they are needed.

If we need eager join fetching in some particular transaction, we have four different ways to specify that.

Passing a JPA `EntityGraph`	We've already seen this in Entity graphs and eager fetching
Specifying a named *fetch profile*	We'll discuss this approach later in Named fetch profiles
Using `left join fetch` in HQL/JPQL	See *A Guide to Hibernate Query Language* for details
Using `From.fetch()` in a criteria query	Same semantics as `join fetch` in HQL

Typically, a query is the most convenient option. Here's how we can ask for join fetching in HQL:

```
List<Book> booksWithJoinFetchedAuthors =
        session.createSelectionQuery("from Book join fetch authors order by isbn")
            .getResultList();
```

And this is the same query, written using the criteria API:

```
var builder = sessionFactory.getCriteriaBuilder();
var query = builder.createQuery(Book.class);
var book = query.from(Book.class);
book.fetch(Book_.authors);
query.select(book);
query.orderBy(builder.asc(book.get(Book_.isbn)));
List<Book> booksWithJoinFetchedAuthors =
        session.createSelectionQuery(query).getResultList();
```

Either way, a single SQL `select` statement is executed:

```
select b1_0.isbn,a1_0.books_isbn,a1_1.id,a1_1.bio,a1_1.name,b1_0.price,b1_0.published,b1_0.publisher_id,b1_0.title
from Book b1_0
    join (Book_Author a1_0 join Author a1_1 on a1_1.id=a1_0.authors_id)
        on b1_0.isbn=a1_0.books_isbn
order by b1_0.isbn
```

Much better!

Join fetching, despite its non-lazy nature, is clearly more efficient than either batch or subselect fetching, and this is the source of our recommendation to avoid the use of lazy fetching.

> There's one interesting case where join fetching becomes inefficient: when we fetch two many-valued associations *in parallel*. Imagine we wanted to fetch both `Author.books` and `Author.royaltyStatements` in some unit of work. Joining both collections in a single query would result in a cartesian product of tables, and a large SQL result set. Subselect fetching comes to the rescue here, allowing us to fetch `books` using a join, and `royaltyStatements` using a single subsequent `select`.

Of course, an alternative way to avoid many round trips to the database is to cache the data we need in the Java client. If we're expecting to find the associated data in a local cache, we probably don't need join fetching at all.

> But what if we can't be *certain* that all associated data will be in the cache? In that case, we might be able to reduce the cost of cache misses by enabling batch fetching.

8.7. The second-level cache

A classic way to reduce the number of accesses to the database is to use a second-level cache, allowing data cached in memory to be shared between sessions.

By nature, a second-level cache tends to undermine the ACID properties of transaction processing in a relational database. We *don't* use a distributed transaction with two-phase commit to ensure that changes to the cache and database happen atomically. So a second-level cache is often by far the easiest way to improve the performance of a system, but only at the cost of making it much more difficult to reason about concurrency. And so the cache is a potential source of bugs which are difficult to isolate and reproduce.

Therefore, by default, an entity is not eligible for storage in the second-level cache. We must explicitly mark each entity that will be stored in the second-level cache with the `@Cache` annotation from `org.hibernate.annotations`.

But that's still not enough. Hibernate does not itself contain an implementation of a second-level cache, so it's also necessary to configure an external *cache provider*.

> Caching is disabled by default. To minimize the risk of data loss, we force you to stop and think before any entity goes into the cache.

Hibernate segments the second-level cache into named *regions*, one for each:

- mapped entity hierarchy or
- collection role.

For example, there might be separate cache regions for `Author`, `Book`, `Author.books`, and `Book.authors`.

Each region is permitted its own policies for expiry, persistence, and replication. These policies must be configured externally to Hibernate.

The appropriate policies depend on the kind of data an entity represents. For example, a program might have different caching policies for "reference" data, for transactional data, and for data used for analytics. Ordinarily, the implementation of those policies is the responsibility of the underlying cache implementation.

> The second-level cache is never aware of any changes to data which are made externally to Hibernate. Updates made via direct JDBC—or by some other program—are never visible in the second-level cache. When such updates occur, we might need to explicitly invalidate cached data. Alternatively, in cases where the program is able to tolerate somewhat stale data, an expiry policy might be an acceptable solution.

8.8. Specifying which data is cached

By default, no data is eligible for storage in the second-level cache.

An entity hierarchy or collection role may be assigned a region using the `@Cache` annotation. If no region name is explicitly specified, the region name is just the name of the entity class or collection role.

```java
@Entity
@Cache(usage=NONSTRICT_READ_WRITE, region="Publishers")
class Publisher {
    ...

    @Cache(usage=READ_WRITE, region="PublishedBooks")
    @OneToMany(mappedBy=Book_.PUBLISHER)
    Set<Book> books;

    ...
}
```

The cache defined by a `@Cache` annotation is automatically utilized by Hibernate to:

- retrieve an entity by id when `find()` is called, or
- to resolve an association by id.

> The `@Cache` annotation must be specified on the *root class* of an entity inheritance hierarchy. It's an error to place it on a subclass entity.

The `@Cache` annotation always specifies a `CacheConcurrencyStrategy`, a policy governing access to the second-level cache by concurrent transactions.

Table 8.8: Cache concurrency

Concurrency policy	**Interpretation**	**Explanation**
`READ_ONLY`	• Immutable data • Read-only access	Indicates that the cached object is immutable, and is never updated. If an entity with this cache concurrency is updated, an exception is thrown. This is the simplest, safest, and best-performing cache concurrency strategy. It's particularly suitable for so-called "reference" data.
`NONSTRICT_READ_WRITE`	• Concurrent updates are extremely improbable • Read/write access with no locking	Indicates that the cached object is sometimes updated, but that it's extremely unlikely that two transactions will attempt to update the same item of data at the same time. This strategy does not use locks. When an item is updated, the cache is invalidated both before and after completion of the updating transaction. But without locking, it's impossible to completely rule out the possibility of a second transaction storing or retrieving stale data in or from the cache during the completion process of the first transaction.
`READ_WRITE`	• Concurrent updates are possible but not common • Read/write access using soft locks	Indicates a non-vanishing likelihood that two concurrent transactions attempt to update the same item of data simultaneously. This strategy uses "soft" locks to prevent concurrent transactions from retrieving or storing a stale item from or in the cache during the transaction completion process. A soft lock is simply a marker entry placed in the cache while the updating transaction completes. • A second transaction may not read the item from the cache while the soft lock is present, and instead simply proceeds to read the item directly from the database, exactly as if a regular cache miss had occurred. • Similarly, the soft lock also prevents this second transaction from storing a stale item to the cache when it returns from its round trip to the database with something that might not quite be the latest version.
`TRANSACTIONAL`	• Concurrent updates are frequent • Transactional access	Indicates that concurrent writes are common, and the only way to maintain synchronization between the second-level cache and the database is via the use of a fully transactional cache provider. In this case, the cache and the database must cooperate via JTA or the XA protocol, and Hibernate itself takes on little responsibility for maintaining the integrity of the cache.

Which policies make sense may also depend on the underlying second-level cache implementation.

> JPA has a similar annotation, named `@Cacheable`. Unfortunately, it's almost useless to us, since:
>
> - it provides no way to specify any information about the nature of the cached entity and how its cache should be managed, and
> - it may not be used to annotate associations, and so we can't even use it to mark collection roles as eligible for storage in the second-level cache.

8.9. Caching by natural id

If our entity has a natural id, we can enable an additional cache, which holds cross-references from natural id to primary id, by annotating the entity `@NaturalIdCache`. By default, the natural id cache is stored in a dedicated region of the second-level cache, separate from the cached entity data.

```
@Entity
@Cache(usage=READ_WRITE, region="Book")
@NaturalIdCache(region="BookIsbn")
class Book {
    ...
    @NaturalId
    String isbn;

    @NaturalId
    int printing;
    ...
}
```

This cache is utilized when the entity is retrieved using one of the operations of `Session` which performs lookup by natural id.

> Since the natural id cache doesn't contain the actual state of the entity, it doesn't make sense to annotate an entity `@NaturalIdCache` unless it's already eligible for storage in the second-level cache, that is, unless it's also annotated `@Cache`.

It's worth noticing that, unlike the primary identifier of an entity, a natural id might be mutable.

We must now consider a subtlety that often arises when we have to deal with so-called "reference data", that is, data which fits easily in memory, and doesn't change much.

8.10. Caching and association fetching

Let's consider again our `Publisher` class:

```
@Cache(usage=NONSTRICT_READ_WRITE, region="Publishers")
@Entity
class Publisher { ... }
```

Data about publishers doesn't change very often, and there aren't so many of them. Suppose we've set everything up so that the publishers are almost *always* available in the second-level cache.

Then in this case we need to think carefully about associations of type `Publisher`.

```
@ManyToOne
Publisher publisher;
```

There's no need for this association to be lazily fetched, since we're expecting it to be available in memory, so we won't set it `fetch=LAZY`. But on the other hand, if we leave it marked for eager fetching then, by default, Hibernate will often fetch it using a join. This places completely unnecessary load on the database.

The solution is the `@Fetch` annotation:

```
@ManyToOne @Fetch(SELECT)
Publisher publisher;
```

By annotating the association `@Fetch(SELECT)`, we suppress join fetching, giving Hibernate a chance to find the associated `Publisher` in the cache.

Therefore, we arrive at this rule of thumb:

> Many-to-one associations to "reference data", or to any other data that will almost always be available in the cache, should be mapped `EAGER`,`SELECT`.
>
> Other associations, as we've already made clear, should be `LAZY`.

Once we've marked an entity or collection as eligible for storage in the second-level cache, we still need to set up an actual cache.

8.11. Configuring the second-level cache provider

Configuring a second-level cache provider is a rather involved topic, and quite outside the scope of this document. But in case it helps, we often test Hibernate with the following configuration, which uses EHCache as the cache implementation, as above in Optional dependencies:

Table 8.9: EHCache configuration

Configuration property name	Property value
`hibernate.cache.region.factory_class`	`jcache`
`hibernate.javax.cache.uri`	`/ehcache.xml`

If you're using EHCache, you'll also need to include an `ehcache.xml` file that explicitly configures the behavior of each cache region belonging to your entities and collections. You'll find more information about configuring EHCache here.

We may use any other implementation of JCache, such as Caffeine. JCache automatically selects whichever implementation it finds on the classpath. If there are multiple implementations on the classpath, we must disambiguate using:

Table 8.10: Disambiguating the JCache implementation

Configuration property name	Property value
`hibernate.javax.cache.provider`	The implementation of `javax.cache.spi.CachingProvider`, for example: `org.ehcache.jsr107.EhcacheCachingProvider` — for EHCache `com.github.benmanes.caffeine.jcache.spi.CaffeineCachingProvider` — for Caffeine

Alternatively, to use Infinispan as the cache implementation, the following settings are required:

Table 8.11: Infinispan provider configuration

Configuration property name	Property value
`hibernate.cache.region.factory_class`	`infinispan`
`hibernate.cache.infinispan.cfg`	Path to infinispan configuration file, for example: `org/infinispan/hibernate/cache/commons/builder/infinispan-configs.xml` — for a distributed cache `org/infinispan/hibernate/cache/commons/builder/infinispan-configs-local.xml` — to test with local cache

Infinispan is usually used when distributed caching is required. There's more about using Infinispan with Hibernate here.

Finally, there's a way to globally disable the second-level cache:

Table 8.12: Setting to disable caching

Configuration property name	Property value
`hibernate.cache.use_second_level_cache`	`true` to enable caching, or `false` to disable it

When `hibernate.cache.region.factory_class` is set, this property defaults to `true`.

> This setting lets us easily disable the second-level cache completely when troubleshooting or profiling performance.

You can find much more information about the second-level cache in the User Guide.

8.12. Caching query result sets

The caches we've described above are only used to optimize lookups by id or by natural id. Hibernate also has a way to cache the result sets of queries, though this is only rarely an efficient thing to do.

The query cache must be enabled explicitly:

Table 8.13: Setting to enable the query cache

Configuration property name	Property value
`hibernate.cache.use_query_cache`	`true` to enable the query cache

To cache the results of a query, call `SelectionQuery.setCacheable(true)`:

```
session.createQuery("from Product where discontinued = false")
    .setCacheable(true)
    .getResultList();
```

By default, the query result set is stored in a cache region named `default-query-results-region`. Since different queries should have different caching policies, it's common to explicitly specify a region name:

```
session.createQuery("from Product where discontinued = false")
    .setCacheable(true)
    .setCacheRegion("ProductCatalog")
    .getResultList();
```

A result set is cached together with a *logical timestamp*. By "logical", we mean that it doesn't actually increase linearly with time, and in particular it's not the system time.

When a `Product` is updated, Hibernate *does not* go through the query cache and invalidate every cached result set that's affected by the change. Instead, there's a special region of the cache which holds a logical timestamp of the most-recent update to each table. This is called the *update timestamps cache*, and it's kept in the region `default-update-timestamps-region`.

> It's *your responsibility* to ensure that this cache region is configured with appropriate policies. In particular, update timestamps should never expire or be evicted.

When a query result set is read from the cache, Hibernate compares its timestamp with the timestamp of each of the tables that affect the results of the query, and *only* returns the result set if the result set isn't stale. If the result set *is* stale, Hibernate goes ahead and re-executes the query against the database and updates the cached result set.

As is generally the case with any second-level cache, the query cache can break the ACID properties of transactions.

8.13. Second-level cache management

For the most part, the second-level cache is transparent. Program logic which interacts with the Hibernate session is unaware of the cache, and is not impacted by changes to caching policies.

At worst, interaction with the cache may be controlled by specifying of an explicit `CacheMode`:

```
session.setCacheMode(CacheMode.IGNORE);
```

Or, using JPA-standard APIs:

```
entityManager.setCacheRetrieveMode(CacheRetrieveMode.BYPASS);
entityManager.setCacheStoreMode(CacheStoreMode.BYPASS);
```

The JPA-defined cache modes come in two flavors: `CacheRetrieveMode` and `CacheStoreMode`.

Table 8.14: JPA-defined cache retrieval modes

Mode	Interpretation
`CacheRetrieveMode.USE`	Read data from the cache if available
`CacheRetrieveMode.BYPASS`	Don't read data from the cache; go direct to the database

We might select `CacheRetrieveMode.BYPASS` if we're concerned about the possibility of reading stale data from the cache.

Table 8.15: JPA-defined cache storage modes

Mode	Interpretation
`CacheStoreMode.USE`	Write data to the cache when read from the database or when modified; do not update already-cached items when reading
`CacheStoreMode.REFRESH`	Write data to the cache when read from the database or when modified; always update cached items when reading
`CacheStoreMode.BYPASS`	Don't write data to the cache

We should select `CacheStoreMode.BYPASS` if we're querying data that doesn't need to be cached.

It's a good idea to set the `CacheStoreMode` to `BYPASS` just before running a query which returns a large result set full of data that we don't expect to need again soon. This saves work, and prevents the newly-read data from pushing out the previously cached data.

In JPA we would use this idiom:

```
entityManager.setCacheStoreMode(CacheStoreMode.BYPASS);
List<Publisher> allpubs =
        entityManager.createQuery("from Publisher", Publisher.class)
            .getResultList();
entityManager.setCacheStoreMode(CacheStoreMode.USE);
```

But Hibernate has a better way:

```
List<Publisher> allpubs =
        session.createSelectionQuery("from Publisher", Publisher.class)
            .setCacheStoreMode(CacheStoreMode.BYPASS)
            .getResultList();
```

A Hibernate `CacheMode` packages a `CacheRetrieveMode` with a `CacheStoreMode`.

Table 8.16: Hibernate cache modes and JPA equivalents

Hibernate `CacheMode`	Equivalent JPA modes
`NORMAL`	`CacheRetrieveMode.USE`, `CacheStoreMode.USE`
`IGNORE`	`CacheRetrieveMode.BYPASS`, `CacheStoreMode.BYPASS`
`GET`	`CacheRetrieveMode.USE`, `CacheStoreMode.BYPASS`
`PUT`	`CacheRetrieveMode.BYPASS`, `CacheStoreMode.USE`
`REFRESH`	`CacheRetrieveMode.REFRESH`, `CacheStoreMode.BYPASS`

There's no particular reason to prefer Hibernate's `CacheMode` over the JPA equivalents. This enumeration only exists because Hibernate had cache modes long before they were added to JPA.

For "reference" data, that is, for data which is expected to always be found in the second-level cache, it's a good idea to *prime* the cache at startup. There's a really easy way to do this: just execute a query immediately after obtaining the `EntityManager` or `SessionFactory`.

```
SessionFactory sessionFactory =
        setupHibernate(new Configuration())
            .buildSessionFactory();
// prime the second-level cache
sessionFactory.inSession(session -> {
    session.createSelectionQuery("from Country")
        .setReadOnly(true)
        .getResultList();
    session.createSelectionQuery("from Product where discontinued = false")
        .setReadOnly(true)
        .getResultList();
});
```

Very occasionally, it's necessary or advantageous to control the cache explicitly. For example, we might need to evict some data that we know to be stale because it was updated:

- via direct JDBC, or
- by some other program.

The `Cache` interface allows programmatic eviction of cached items.

```
sessionFactory.getCache().evictEntityData(Book.class, bookId);
```

Second-level cache management via the `Cache` interface is not transaction-aware. None of the operations of `Cache` respect any isolation or transactional semantics associated with the underlying caches. In particular, eviction via the methods of this interface causes an immediate "hard" removal outside any current transaction and/or locking scheme.

Ordinarily, however, Hibernate automatically evicts or updates cached data after modifications, and, in addition, cached data which is unused will eventually be expired according to the configured policies.

This is quite different to what happens with the first-level cache.

8.14. Session cache management

Entity instances aren't automatically evicted from the session cache when they're no longer needed. Instead, they stay pinned in memory until the session they belong to is discarded by your program. So if you query many entities from within a single persistence context, you might experience very poor performance. In such cases, it becomes necessary to manage the session cache manually.

The methods `detach()` and `clear()` allow you to remove entities from the session cache, making them available for garbage collection. Since most sessions are rather short-lived, you won't need these operations very often. And if you find yourself thinking you *do* need them in a certain situation, you should strongly consider an alternative solution: a *stateless session*.

The very best way to avoid having too many entities pinned in the session cache is to not load them from the database in the first place. Many operations can be performed more efficiently on the database server. This is especially true for bulk operations.

To update many entities at once, use an `update` statement:

```
session.createMutationQuery(
        """
        update Book as b
            set b.totalSales =
                coalesce((select sum(quantity) from OrderItem where book = b), 0)
        """
        ).executeUpdate();
```

To insert many entities at once, use `insert … select`:

```
session.createMutationQuery(
        """
        insert into OrderQueue (id, isbn, quantity, customerId)
            select o.id, i.quantity, i.book.isbn, o.customer.id
```

```
            from Order o join o.items i
            where o.status = PENDING
        """
        ).executeUpdate();
```

For more complicated operations, consider using a stored procedure.

8.15. Stateless sessions

An arguably-underappreciated feature of Hibernate is the `StatelessSession` interface, which provides a command-oriented, more bare-metal approach to interacting with the database.

You may obtain a stateless session from the `SessionFactory`:

```
StatelessSession ss = getSessionFactory().openStatelessSession();
```

A stateless session:

- doesn't have a first-level cache (persistence context), and
- doesn't implement transactional write-behind or automatic dirty checking, so all operations are executed immediately when they're explicitly called.

For a stateless session, we're always working with detached objects. Thus, the programming model is a bit different:

Table 8.17: Important methods of the `StatelessSession`

Method name and parameters	Effect
`get(Class, Object)`	Obtain a detached object, given its type and its id, by executing a `select`
`fetch(Object)`	Fetch an association of a detached object
`refresh(Object)`	Refresh the state of a detached object by executing a `select`
`insert(Object)`	Immediately `insert` the state of the given transient object into the database
`update(Object)`	Immediately `update` the state of the given detached object in the database
`delete(Object)`	Immediately `delete` the state of the given detached object from the database
`upsert(Object)`	Immediately `insert` or `update` the state of the given detached object using a SQL `merge into` statement

The operations of a stateless session have no corresponding `CascadeType`s, and so these operations never cascade to associated entity instances.

There's no `flush()` operation, and so `update()` is always explicit.

In certain circumstances, this makes stateless sessions easier to work with and simpler to reason about, but with the caveat that a stateless session is much more vulnerable to data aliasing effects, since it's easy to get two non-identical Java objects which both represent the same row of a database table.

Consider the following fragments:

```
var b1 = statelessSession.get(Book.class, isbn);
var b2 = statelessSession.get(Book.class, isbn);
assert b1 == b2;  // fails
```

```
var b1 = statelessSession.get(Book.class, isbn);
var b2 = statelessSession.get(Book.class, isbn);
statelessSession.fetch(b1.publisher);
statelessSession.fetch(b2.publisher);
assert b1.publisher == b2.publisher;  // fails
```

In a stateful session, entity instances are canonicalized by primary key, and so we don't usually have two different objects representing a single row. No such canonicalization exists across invocations of a stateless session. In a stateless session, both the assertions fail.

> If we use `fetch()` in a stateless session, we can very easily obtain two objects representing the same database row!

But there are also some advantages to this model. In particular, the absence of a persistence context means that we can safely perform bulk-processing tasks without allocating huge quantities of memory. Use of a `StatelessSession` alleviates the need to call:

- `clear()` or `detach()` to perform first-level cache management, and
- `setCacheMode()` to bypass interaction with the second-level cache.

> Stateless sessions can be useful, but for bulk operations on huge datasets, Hibernate can't possibly compete with stored procedures!

8.16. Optimistic and pessimistic locking

Finally, an aspect of behavior under load that we didn't mention above is row-level data contention. When many transactions try to read and update the same data, the program might become unresponsive with lock escalation, deadlocks, and lock acquisition timeout errors.

There's two basic approaches to data concurrency in Hibernate:

- optimistic locking using `@Version` columns, and
- database-level pessimistic locking using the SQL `for update` syntax (or equivalent).

In the Hibernate community it's *much* more common to use optimistic locking, and Hibernate makes that incredibly easy.

> Where possible, in a multiuser system, avoid holding a pessimistic lock across a user interaction. Indeed, the usual practice is to avoid having transactions that span user interactions. For multiuser systems, optimistic locking is king.

That said, there *is* also a place for pessimistic locks, which can sometimes reduce the probability of transaction rollbacks.

Therefore, the `find()`, `lock()`, and `refresh()` methods of the session accept an optional `LockMode`. Here's the simplest way to execute a `select … for update` in Hibernate:

```
Book book = session.find(Book.class, isbn, LockMode.PESSIMISTIC_WRITE);
```

We can also specify a `LockMode` for a query.

A lock mode can be used to request a pessimistic lock, or to customize the behavior of optimistic locking:

Table 8.18: Optimistic and pessimistic lock modes

`LockMode` **type**	**Meaning**
`READ`	An optimistic lock obtained implicitly whenever an entity is read from the database using `select`
`OPTIMISTIC`	An optimistic lock obtained when an entity is read from the database, and verified using a `select` to check the version when the transaction completes
`OPTIMISTIC_FORCE_INCREMENT`	An optimistic lock obtained when an entity is read from the database, and enforced using an `update` to increment the version when the transaction completes
`WRITE`	A pessimistic lock obtained implicitly whenever an entity is written to the database using `update` or `insert`
`PESSIMISTIC_READ`	A pessimistic `for share` lock
`PESSIMISTIC_WRITE`	A pessimistic `for update` lock
`PESSIMISTIC_FORCE_INCREMENT`	A pessimistic lock enforced using an immediate `update` to increment the version
`NONE`	No lock; assigned when an entity is read from the second-level cache

Note that an `OPTIMISTIC` lock is always verified at the end of the transaction, even when the entity has not been modified. This is slightly different to what most people mean when they talk about an "optimistic lock". It's never necessary to request an `OPTIMISTIC` lock on a modified entity, since the version number is always verified when a SQL `update` is executed.

> JPA has its own `LockModeType`, which enumerates most of the same modes. However, JPA's `LockModeType.READ` is a

> synonym for `OPTIMISTIC`—it's not the same as Hibernate's `LockMode.READ`. Similarly, `LockModeType.WRITE` is a synonym for `OPTIMISTIC_FORCE_INCREMENT` and is not the same as `LockMode.WRITE`.

A pessimistic lock request may be combined with an explicit `Timeout`.

```
session.lock(book, LockMode.PESSIMISTIC_WRITE, Timeout.seconds(2))
```

The interface `Timeouts` defines some special instances of `Timeout` which may be used to request use of `for update nowait` or `for update skip locked` on databases which support these options.

```
session.lock(book, LockMode.PESSIMISTIC_WRITE, Timeouts.NO_WAIT)
```

8.17. Collecting statistics

We may ask Hibernate to collect statistics about its activity by setting this configuration property:

Configuration property name	Property value
`hibernate.generate_statistics`	`true` to enable collection of statistics

The statistics are exposed by the `Statistics` object:

```
long failedVersionChecks =
        sessionFactory.getStatistics()
            .getOptimisticFailureCount();

long publisherCacheMissCount =
        sessionFactory.getStatistics()
            .getEntityStatistics(Publisher.class.getName())
                .getCacheMissCount()
```

Hibernate's statistics enable observability. Both Micrometer and SmallRye Metrics are capable of exposing these metrics.

8.18. Using Java Flight Recorder

Hibernate JFR is a separate module which reports events to Java Flight Recorder. This is different to reporting aggregated metrics via a tool like Micrometer, since JFR records information about the timing and duration of each discrete event, along with a stack trace. If anything, the information reported by JFR is a little *too* detailed to make it really useful for performance tuning—it's perhaps more useful for *troubleshooting*.

No special configuration is required to use Hibernate JFR. Just include `org.hibernate.orm:hibernate-jfr` as a runtime dependency. In particular, you *don't* need to enable `hibernate.generate_statistics`.

8.19. Tracking down slow queries

When a poorly-performing SQL query is discovered in production, it can sometimes be hard to track down exactly where in the Java code the query originates. Hibernate offers two configuration properties that can make it easier to identify a slow query and find its source.

Table 8.20: Settings for tracking slow queries

Configuration property name	Purpose	Property value
`hibernate.log_slow_query`	Log slow queries at the `INFO` level	The minimum execution time, in milliseconds, which characterizes a "slow" query
`hibernate.use_sql_comments`	Prepend comments to the executed SQL	`true` or `false`

When `hibernate.use_sql_comments` is enabled, the text of the HQL query is prepended as a comment to the generated SQL, which usually makes it easy to find the HQL in the Java code.

The comment text may be customized:

- by calling `Query.setComment(comment)` or `Query.setHint(AvailableHints.HINT_COMMENT,comment)`, or

- via the `@NamedQuery` annotation.

> Once you've identified a slow query, one of the best ways to make it faster is to *actually go and talk to someone who is an expert at making queries go fast*. These people are called "database administrators", and if you're reading this document you probably aren't one. Database administrators know lots of stuff that Java developers don't. So if you're lucky enough to have a DBA about, you don't need to Dunning-Kruger your way out of a slow query.

An expertly-defined index might be all you need to fix a slow query.

8.20. Adding indexes

The `@Index` annotation may be used to add an index to a table:

```
@Entity
@Table(indexes=@Index(columnList="title, year, publisher_id"))
class Book { ... }
```

It's even possible to specify an ordering for an indexed column, or that the index should be case-insensitive:

```
@Entity
@Table(indexes=@Index(columnList="(lower(title)), year desc, publisher_id"))
class Book { ... }
```

This lets us create a customized index for a particular query.

Note that SQL expressions like `lower(title)` must be enclosed in parentheses in the `columnList` of the index definition.

> It's not clear that information about indexes belongs in annotations of Java code. Indexes are usually maintained and modified by a database administrator, ideally by an expert in tuning the performance of one particular RDBMS. So it might be better to keep the definition of indexes in a SQL DDL script that your DBA can easily read and modify. Remember, we can ask Hibernate to execute a DDL script using the property `javax.persistence.schema-generation.create-script-source`.

8.21. Dealing with denormalized data

A typical relational database table in a well-normalized schema has a relatively small number of columns, and so there's little to be gained by selectively querying columns and populating only certain fields of an entity class.

But occasionally, we hear from someone asking how to map a table with a hundred columns or more! This situation can arise when:

- data is intentionally denormalized for performance,
- the results of a complicated analytic query are exposed via a view, or
- someone has done something crazy and wrong.

Let's suppose that we're *not* dealing with the last possibility. Then we would like to be able to query the monster table without returning all of its columns. At first glance, Hibernate doesn't offer a perfect bottled solution to this problem. This first impression is misleading. Actually, Hibernate features more than one way to deal with this situation, and the real problem is deciding between the ways. We could:

1. map multiple entity classes to the same table or view, being careful about "overlaps" where a mutable column is mapped to more than one of the entities,
2. use HQL or native SQL queries returning results into record types instead of retrieving entity instances, or
3. use the bytecode enhancer and `@LazyGroup` for attribute-level lazy fetching.

Some other ORM solutions push the third option as the recommended way to handle huge tables, but this has never been the preference of the Hibernate team or Hibernate community. It's much more typesafe to use one of the first two options.

8.22. Reactive programming with Hibernate

Finally, many systems which require high scalability now make use of reactive programming and reactive streams. Hibernate Reactive brings O/R mapping to the world of reactive programming. You can learn much more about Hibernate Reactive from its Reference Documentation.

> Hibernate Reactive may be used alongside vanilla Hibernate in the same program, and can reuse the same entity classes. This means you can use the reactive programming model exactly where you need it—perhaps only in one or two places in

> your system. You don't need to rewrite your whole program using reactive streams.

Chapter 9. Advanced Topics

In the last chapter of this Introduction, we turn to some topics that don't really belong in an introduction. Here we consider some problems, and solutions, that you're probably not going to run into immediately if you're new to Hibernate. But we do want you to know *about* them, so that when the time comes, you'll know what tool to reach for.

9.1. Filters

Filters are one of the nicest and under-usedest features of Hibernate, and we're quite proud of them. A filter is a named, globally-defined, parameterized restriction on the data that is visible in a given session.

Examples of well-defined filters might include:

- a filter that restricts the data visible to a given user according to row-level permissions,
- a filter which hides data which has been soft-deleted,
- in a versioned database, a filter that displays versions which were current at a given instant in the past, or
- a filter that restricts to data associated with a certain geographical region.

A filter must be declared somewhere. A package descriptor is as good a place as any for a `@FilterDef`:

```
@FilterDef(name = "ByRegion",
           parameters = @ParamDef(name = "region", type = String.class))
package org.hibernate.example;
```

This filter has one parameter. Fancier filters might in principle have multiple parameters, though we admit this must be quite rare.

> If you add annotations to a package descriptor, and you're using `Configuration` to configure Hibernate, make sure you call `Configuration.addPackage()` to let Hibernate know that the package descriptor is annotated.

Typically, but not necessarily, a `@FilterDef` specifies a default restriction:

```
@FilterDef(name = "ByRegion",
           parameters = @ParamDef(name = "region", type = String.class),
           defaultCondition = "region = :region")
package org.hibernate.example;
```

Note that filter restrictions are always written in the native SQL dialect of the database, *not* in HQL.

The restriction must contain a reference to the parameter of the filter, specified using the usual syntax for named parameters.

Any entity or collection which is affected by a filter must be annotated `@Filter`:

```
@Entity
@Filter(name = example_.BY_REGION)
class User {

    @Id String username;

    String region;

    ...
}
```

Here, as usual, `example_.BY_REGION` is generated by Hibernate Processor, and is just a constant with the value `"ByRegion"`.

If the `@Filter` annotation does not explicitly specify a restriction, the default restriction given by the `@FilterDef` will be applied to the entity. But an entity is free to override the default condition.

```
@Entity
@Filter(name = example_.FILTER_BY_REGION, condition = "name = :region")
class Region {

    @Id String name;

    ...
}
```

Note that the restriction specified by the `condition` or `defaultCondition` is a native SQL expression.

Table 9.1: Annotations for defining filters

Annotation	Purpose
`@FilterDef`	Defines a filter and declares its name (exactly one per filter)
`@Filter`	Specifies how a filter applies to a given entity or collection (many per filter)

> A filter `condition` may not specify joins to other tables, but it may contain a subquery.
>
> ```
> @Filter(name="notDeleted"
>         condition="(select r.deletionTimestamp from Record r where r.id = record_id) is not null")
> ```
>
> Only unqualified column names like `record_id` in this example are interpreted as belonging to the table of the filtered entity.

By default, a new session comes with every filter disabled. A filter may be explicitly enabled in a given session by calling `enableFilter()` and assigning arguments to the parameters of the filter using the returned instance of `Filter`. You should do this right at the *start* of the session.

```
sessionFactory.inTransaction(session -> {
    session.enableFilter(example_.FILTER_BY_REGION)
        .setParameter("region", "es")
        .validate();

    ...
});
```

Now, any queries executed within the session will have the filter restriction applied. Collections annotated `@Filter` will also have their members correctly filtered.

> On the other hand, filters are not applied to `@ManyToOne` associations, nor to `find()`. This is completely by design and is not in any way a bug.

More than one filter may be enabled in a given session.

Alternatively, since Hibernate 6.5, a filter may be declared as `autoEnabled` in every session. In this case, the argument to a filter parameter must be obtained from a `Supplier`.

```
@FilterDef(name = "ByRegion",
           autoEnabled = true,
           parameters = @ParamDef(name = "region", type = String.class,
                                  resolver = RegionSupplier.class),
           defaultCondition = "region = :region")
package org.hibernate.example;
```

It's not necessary to call `enableFilter()` for a filter declared `autoEnabled = true`.

> When we only need to filter rows by a static condition with no parameters, we don't need a filter, since `@SQLRestriction` provides a much simpler way to do that.

We've mentioned that a filter can be used to implement versioning, and to provide *historical* views of the data. Being such a general-purpose construct, filters provide a lot of flexibility here. But if you're after a more focused/opinionated solution to this problem, you should definitely check out Envers.

Using Envers for auditing historical data

Envers is an add-on to Hibernate ORM which keeps a historical record of each versioned entity in a separate *audit table*, and allows past revisions of the data to be viewed and queried. A full introduction to Envers would require a whole chapter, so we'll just give you a quick taste here.

First, we must mark an entity as versioned, using the `@Audited` annotation:

```
@Audited @Entity
@Table(name="CurrentDocument")
@AuditTable("DocumentRevision")
class Document { ... }
```

> The `@AuditTable` annotation is optional, and it's better to set either `org.hibernate.envers.audit_table_prefix` or `org.hibernate.envers.audit_table_suffix` and let the audit table name be inferred.

The `AuditReader` interface exposes operations for retrieving and querying historical revisions. It's really easy to get hold of one of these:

```
AuditReader reader = AuditReaderFactory.get(entityManager);
```

Envers tracks revisions of the data via a global *revision number*. We may easily find the revision number which was current at a given instant:

```
Number revision = reader.getRevisionNumberForDate(datetime);
```

We can use the revision number to ask for the version of our entity associated with the given revision number:

```
Document doc = reader.find(Document.class, id, revision);
```

Alternatively, we can directly ask for the version which was current at a given instant:

```
Document doc = reader.find(Document.class, id, datetime);
```

We can even execute queries to obtain lists of entities current at the given revision number:

```
List documents =
        reader.createQuery()
            .forEntitiesAtRevision(Document.class, revision)
            .getResultList();
```

For much more information, see the User Guide.

Historically, filters where often used to implement soft-delete. But, since 6.4, Hibernate now comes with soft-delete built in.

9.2. Soft-delete

Even when we don't need complete historical versioning, we often prefer to "delete" a row by marking it as obsolete using a SQL `update`, rather than by executing an actual SQL `delete` and removing the row from the database completely.

The `@SoftDelete` annotation controls how this works:

```
@Entity
@SoftDelete(columnName = "deleted",
            converter = TrueFalseConverter.class)
class Draft {

    ...
}
```

The `columnName` specifies a column holding the deletion status, and the `converter` is responsible for converting a Java `Boolean` to the type of that column. In this example, `TrueFalseConverter` sets the column to the character `'F'` initially, and to `'T'` when the row is deleted. Any JPA `AttributeConverter` for the Java `Boolean` type may be used here. Built-in options include `NumericBooleanConverter` and `YesNoConverter`.

Much more information about soft delete is available in the User Guide.

Another feature that you *could* use filters for, but now don't need to, is multi-tenancy.

9.3. Multi-tenancy

A *multi-tenant* database is one where the data is segregated by *tenant*. We don't need to actually define what a "tenant" really represents here; all we care about at this level of abstraction is that each tenant may be distinguished by a unique identifier. And that there's a well-defined *current tenant* in each session.

We may specify the current tenant when we open a session:

```
var session =
        sessionFactory.withOptions()
            .tenantIdentifier(tenantId)
            .openSession();
```

Or, when using JPA-standard APIs:

```
var entityManager =
        entityManagerFactory.createEntityManager(Map.of(HibernateHints.HINT_TENANT_ID, tenantId));
```

However, since we often don't have this level of control over creation of the session, it's more common to supply an implementation of `CurrentTenantIdentifierResolver` to Hibernate.

To make use of multi-tenancy, we'll usually need to set at least one of these configuration properties:

Table 9.2: Multi-tenancy configuration

Configuration property name	Purpose
`hibernate.tenant_identifier_resolver`	Specifies the `CurrentTenantIdentifierResolver`
`hibernate.multi_tenant.schema_mapper`	Specifies the `TenantSchemaMapper` for schema-based multi-tenancy
`hibernate.multi_tenant.credentials_mapper`	Specifies the `TenantCredentialsMapper` for schema-based or discriminator-based multi-tenancy
`hibernate.multi_tenant_connection_provider`	Specifies the `MultiTenantConnectionProvider` for database-based multi-tenancy

There are three common approaches to multi-tenancy:

1. each tenant has its own database,
2. each tenant has its own schema, or
3. tenants share tables in a single schema, and rows are tagged with the tenant id.

Database-based multi-tenancy

The first option is to give each tenant its own database. That is, we'll use a separate source of JDBC connections for each tenant.

The interface `MultiTenantConnectionProvider` is responsible for obtaining an appropriate `Connection` for a given tenant. Typically, we'll provide a custom implementation of this interface:

- from time to time, Hibernate will ask for a connection, passing the id of the current tenant, and then we must create an appropriate connection or obtain one from a pool, and return it to Hibernate, and
- later, Hibernate will release the connection and ask us to destroy it or return it to the appropriate pool.

> Check out `DataSourceBasedMultiTenantConnectionProviderImpl` for inspiration. If your source of JDBC connections is a set of JNDI-bound `DataSource`s, you might even be able to use this implementation directly.

Schema-based multi-tenancy

The second option is to keep all the data for different tenants in the same database, giving each tenant a different named database schema with its own set of tables.

- If all tenants connect to the database using the same credentials, we must supply a `TenantSchemaMapper` which is responsible for mapping tenant ids to schema names.
- If each tenant should use distinct credentials to connect to the database, and if your JDBC `DataSource` supports this (not all do!) then we can use a `TenantCredentialsMapper` to supply the username and password of each tenant. If each database user has a different default schema, we might not need a `TenantSchemaMapper`.

Discriminator-based multi-tenancy

The third option is quite different. In this case we store data from all tenants in the same tables, but each table has a dedicated column holding the tenant id mapped by each of our entities.

```
@Entity
class Account {
    @Id String id;
    @TenantId String tenantId;

    ...
}
```

The `@TenantId` annotation is used to indicate an attribute of an entity which holds the tenant id. Within a given session, our data is automatically filtered so that only rows tagged with the tenant id of the current tenant are visible in that session.

> Native SQL queries are *not* automatically filtered by tenant id; you'll have to do that part yourself.

9.4. Read-only replicas

A similar but distinct problem is accessing data held in a read-only replica of the main production database. One way to handle this problem is to simply instantiate two instances of `SessionFactory`:

- read-only transactions use a `SessionFactory` configured to access the read-only replica, while
- other transactions use the `SessionFactory` configured to read from and write to the main database.

> The second-level cache doesn't play well with replication, and so a `SessionFactory` with access to a read-only replica should be configured with the second-level cache disabled.

Alternatively, Hibernate 7.2 introduces experimental support for accessing replicas via a single instance of `SessionFactory`. A `Session` which accesses a read-only replica must be created in a special read-only mode:

```
Session readOnlySession =
        factory.withOptions()
                .readOnly(true)
                .initialCacheMode(CacheMode.IGNORE)
                .openSession();
```

There are now two possibilities.

- Some JDBC drivers (MySQL) are able to automatically direct read-only sessions to the read-only replica. In this case, there's no more work to do, since Hibernate will automatically call `Connection.setReadOnly(true)` to signal to the JDBC driver that the read-only replica may be used.
- Other drivers (Postgres, Oracle) don't feature any special support for read-only replicas, and in this case we need to supply our own custom `ConnectionProvider` or `MultiTenantConnectionProvider` and implement `getReadOnlyConnection()` to return a connection to the read-only replica.

Notice that we created the read-only session with `CacheMode.IGNORE`, indicating that access to the read-only replica should bypass the second-level cache. There's two different phenomena we need to consider here:

1. A read-only replica might contain stale data which has already been updated or deleted from the main database and invalidated in the second-level cache. A read-only session might read this stale data and recache it, exposing the stale data to subsequent sessions. Use of `CacheMode.GET` in a read-only session prevents this phenomenon.
2. A session might read data which has not yet been replicated from the main database and add it to the second-level cache. If a read-only session reads this data from the cache, it might fail to resolve references to other data which has not yet been replicated. Use of `CacheMode.PUT` in a read-only session prevents this phenomenon.

Use of `CacheMode.IGNORE` in a read-only session prevents both phenomena.

> The first issue is objectively more serious than the second, and the second issue can often be avoided by careful programming by someone *who really understands what they're doing*. We do not, therefore, require the use of `CacheMode.IGNORE`, but we strongly encourage the use of *at least* `CacheMode.GET` in every read-only session.

9.5. Using custom-written SQL

We've already discussed how to run queries written in SQL, but occasionally that's not enough. Sometimes—but much less often than you might expect—we would like to customize the SQL used by Hibernate to perform basic CRUD operations for an entity or collection.

For this we can use `@SQLInsert` and friends:

```
@Entity
@SQLInsert(sql = "insert into person (name, id, valid) values (?, ?, true)",
           verify = Expectation.RowCount.class)
@SQLUpdate(sql = "update person set name = ? where id = ?")
@SQLDelete(sql = "update person set valid = false where id = ?")
@SQLSelect(sql = "select id, name from person where id = ? and valid = true")
public static class Person { ... }
```

Table 9.3: Annotations for overriding generated SQL

Annotation	Purpose
`@SQLSelect`	Overrides a generated SQL `select` statement
`@SQLInsert`	Overrides a generated SQL `insert` statement
`@SQLUpdate`	Overrides a generated SQL `update` statement
`@SQDelete`	Overrides a generated SQL `delete` statement for a single row
`@SQDeleteAll`	Overrides a generated SQL `delete` statement for multiple rows
`@SQLRestriction`	Adds a restriction to generated SQL
`@SQLOrder`	Adds an ordering to generated SQL

> If the custom SQL should be executed via a `CallableStatement`, just specify `callable=true`.

Any SQL statement specified by one of these annotations must have exactly the number of JDBC parameters that Hibernate expects, that is, one for each column mapped by the entity, in the exact order Hibernate expects. In particular, the primary key columns must come last.

However, the `@Column` annotation does lend some flexibility here:

- if a column should not be written as part of the custom `insert` statement, and has no corresponding JDBC parameter in the custom SQL, map it `@Column(insertable=false)`, or
- if a column should not be written as part of the custom `update` statement, and has no corresponding JDBC parameter in the custom SQL, map it `@Column(updatable=false)`.

The `verify` member of these annotations specifies a class implementing `Expectation`, allowing customized logic for checking the success of an operation executed via JDBC. There are three built-in implementations:

- `Expectation.None`, which performs no checks,
- `Expectation.RowCount`, which is what Hibernate usually uses when executing its own generated SQL,
- and `Expectation.OutParameter`, which is useful for checking an output parameter of a stored procedure.

You can write your own implementation of `Expectation` if none of these options is suitable.

> If you need custom SQL, but are targeting multiple dialects of SQL, you can use the annotations defined in `DialectOverride`. For example, this annotation lets us override the custom `insert` statement just for PostgreSQL:
>
> ```
> @DialectOverride.SQLInsert(dialect = PostgreSQLDialect.class,
>     override = @SQLInsert(sql="insert into person (name,id) values (?,gen_random_uuid())"))
> ```
>
> It's even possible to override the custom SQL for specific *versions* of a database.

Sometimes a custom `insert` or `update` statement assigns a value to a mapped column which is calculated when the statement is executed on the database. For example, the value might be obtained by calling a SQL function:

```
@SQLInsert(sql = "insert into person (name, id) values (?, gen_random_uuid())")
```

But the entity instance which represents the row being inserted or updated won't be automatically populated with that value. And so our persistence context loses synchronization with the database. In situations like this, we may use the `@Generated` annotation to tell Hibernate to reread the state of the entity after each `insert` or `update`.

9.6. Handling database-generated columns

Sometimes, a column value is assigned or mutated by events that happen in the database, and aren't visible to Hibernate. For example:

- a table might have a column value populated by a trigger,
- a mapped column might have a default value defined in DDL, or
- a custom SQL `insert` or `update` statement might assign a value to a mapped column, as we saw in the previous subsection.

One way to deal with this situation is to explicitly call `refresh()` at appropriate moments, forcing the session to reread the state of the entity. But this is annoying.

The `@Generated` annotation relieves us of the burden of explicitly calling `refresh()`. It specifies that the value of the annotated entity attribute is generated by the database, and that the generated value should be automatically retrieved using a SQL `returning` clause, or separate `select` after it is generated.

A useful example is the following mapping:

```
@Entity
class Entity {
    @Generated @Id
    @ColumnDefault("gen_random_uuid()")
    UUID id;
}
```

The generated DDL is:

```
create table Entity (
    id uuid default gen_random_uuid() not null,
    primary key (uuid)
)
```

So here the value of `id` is defined by the column default clause, by calling the PostgreSQL function `gen_random_uuid()`.

When a column value is generated during updates, use `@Generated(event=UPDATE)`. When a value is generated by both inserts *and* updates, use `@Generated(event={INSERT,UPDATE})`.

> For columns which should be generated using a SQL `generated always as` clause, prefer the `@GeneratedColumn` annotation, so that Hibernate automatically generates the correct DDL.

Actually, the `@Generated` and `@GeneratedColumn` annotations are defined in terms of a more generic and user-extensible framework for handling attribute values generated in Java, or by the database. So let's drop down a layer, and see how that works.

9.7. User-defined generators

JPA doesn't define a standard way to extend the set of id generation strategies, but Hibernate does:

- the `Generator` hierarchy of interfaces in the package `org.hibernate.generator` lets you define new generators, and
- the `@IdGeneratorType` meta-annotation from the package `org.hibernate.annotations` lets you write an annotation which associates a `Generator` type with identifier attributes.

Furthermore, the `@ValueGenerationType` meta-annotation lets you write an annotation which associates a `Generator` type with a non-`@Id` attribute.

> These APIs were new in Hibernate 6, and supersede the classic `IdentifierGenerator` interface and `@GenericGenerator` annotation from older versions of Hibernate. However, the older APIs are still available and custom `IdentifierGenerator`s written for older versions of Hibernate continue to work in Hibernate 7.

Hibernate has a range of built-in generators which are defined in terms of this new framework.

Table 9.4: Built-in generators

Annotation	Implementation	Purpose
`@Generated`	`GeneratedGeneration`	Generically handles database-generated values
`@GeneratedColumn`	`GeneratedAlwaysGeneration`	Handles values generated using `generated always`
`@CurrentTimestamp`	`CurrentTimestampGeneration`	Generic support for database or in-memory generation of creation or update timestamps
`@CreationTimestamp`	`CurrentTimestampGeneration`	A timestamp generated when an entity is first made persistent
`@UpdateTimestamp`	`CurrentTimestampGeneration`	A timestamp generated when an entity is made persistent, and regenerated every time the entity is modified
`@UuidGenerator`	`UuidGenerator`	A more flexible generator for RFC 4122 UUIDs

Furthermore, support for JPA's standard id generation strategies is also defined in terms of this framework.

As an example, let's look at how `@UuidGenerator` is defined:

```
@IdGeneratorType(org.hibernate.id.uuid.UuidGenerator.class)
@ValueGenerationType(generatedBy = org.hibernate.id.uuid.UuidGenerator.class)
@Retention(RUNTIME)
@Target({ FIELD, METHOD })
public @interface UuidGenerator { ... }
```

`@UuidGenerator` is meta-annotated both `@IdGeneratorType` and `@ValueGenerationType` because it may be used to generate both ids and values of regular attributes. Either way, this `Generator` class does the hard work:

```
public class UuidGenerator
        // this generator produced values before SQL is executed
        implements BeforeExecutionGenerator {

    // constructors accept an instance of the @UuidGenerator
    // annotation, allowing the generator to be "configured"

    // called to create an id generator
    public UuidGenerator(
            org.hibernate.annotations.UuidGenerator config,
            Member idMember,
            GeneratorCreationContext creationContext) {
        this(config, idMember);
    }

    // called to create a generator for a regular attribute
    public UuidGenerator(
            org.hibernate.annotations.UuidGenerator config,
            Member member,
            GeneratorCreationContext creationContext) {
        this(config, idMember);
    }

    ...

    @Override
    public EnumSet<EventType> getEventTypes() {
        // UUIDs are only assigned on insert, and never regenerated
        return INSERT_ONLY;
    }

    @Override
    public Object generate(SharedSessionContractImplementor session, Object owner, Object currentValue, EventType
eventType) {
        // actually generate a UUID and transform it to the required type
        return valueTransformer.transform( generator.generateUuid( session ) );
    }
```

```
}
```

You can find out more about custom generators from the Javadoc for `@IdGeneratorType` and for `org.hibernate.generator`.

9.8. Naming strategies

When working with a pre-existing relational schema, it's usual to find that the column and table naming conventions used in the schema don't match Java's naming conventions.

Of course, the `@Table` and `@Column` annotations let us explicitly specify a mapped table or column name. But we would prefer to avoid scattering these annotations across our whole domain model.

Therefore, Hibernate lets us define a mapping between Java naming conventions, and the naming conventions of the relational schema. Such a mapping is called a *naming strategy*.

First, we need to understand how Hibernate assigns and processes names.

- *Logical naming* is the process of applying naming rules to determine the *logical names* of objects which were not explicitly assigned names in the O/R mapping. That is, when there's no `@Table` or `@Column` annotation.
- *Physical naming* is the process of applying additional rules to transform a logical name into an actual "physical" name that will be used in the database. For example, the rules might include things like using standardized abbreviations, or trimming the length of identifiers.

Thus, there's two flavors of naming strategy, with slightly different responsibilities. Hibernate comes with default implementations of these interfaces:

Flavor	Default implementation
An `ImplicitNamingStrategy` is responsible for assigning a logical name when none is specified by an annotation	A default strategy which implements the rules defined by JPA
A `PhysicalNamingStrategy` is responsible for transforming a logical name and producing the name used in the database	A trivial implementation which does no processing

We happen to not much like the naming rules defined by JPA, which specify that mixed case and camel case identifiers should be concatenated using underscores. We bet you could easily come up with a much better `ImplicitNamingStrategy` than that! (Hint: it should always produce legit mixed case identifiers.)

The popular `PhysicalNamingStrategySnakeCaseImpl` produces snake case identifiers.

Custom naming strategies may be enabled using the configuration properties we already mentioned without much explanation back in Minimizing repetitive mapping information.

Table 9.6: Naming strategy configuration

Configuration property name	Purpose
`hibernate.implicit_naming_strategy`	Specifies the `ImplicitNamingStrategy`
`hibernate.physical_naming_strategy`	Specifies the `PhysicalNamingStrategy`

9.9. Spatial datatypes

Hibernate Spatial augments the built-in basic types with a set of Java mappings for OGC spatial types.

- Geolatte-geom defines a set of Java types implementing the OGC spatial types, and codecs for translating to and from database-native spatial datatypes.
- Hibernate Spatial itself supplies integration with Hibernate.

To use Hibernate Spatial, we must add it as a dependency, as described in Optional dependencies.

Then we may immediately use Geolatte-geom and JTS types in our entities. No special annotations are needed:

```
import org.locationtech.jts.geom.Point;
import jakarta.persistence.*;
```

```java
@Entity
class Event {
    Event() {}

    Event(String name, Point location) {
        this.name = name;
        this.location = location;
    }

    @Id @GeneratedValue
    Long id;

    String name;

    Point location;

}
```

The generated DDL uses `geometry` as the type of the column mapped by `location`:

```sql
create table Event (
    id bigint not null,
    location geometry,
    name varchar(255),
    primary key (id)
)
```

Hibernate Spatial lets us work with spatial types just as we would with any of the built-in basic attribute types.

```java
var geometryFactory = new GeometryFactory();
...

Point point = geometryFactory.createPoint(new Coordinate(10, 5));
session.persist(new Event("Hibernate ORM presentation", point));
```

But what makes this powerful is that we may write some very fancy queries involving functions of spatial types:

```java
Polygon triangle =
        geometryFactory.createPolygon(
                new Coordinate[] {
                        new Coordinate(9, 4),
                        new Coordinate(11, 4),
                        new Coordinate(11, 20),
                        new Coordinate(9, 4)
                }
        );
Point event =
        session.createQuery("select location from Event where within(location, :zone) = true", Point.class)
                .setParameter("zone", triangle)
                .getSingleResult();
```

Here, `within()` is one of the functions for testing spatial relations defined by the OpenGIS specification. Other such functions include `touches()`, `intersects()`, `distance()`, `boundary()`, etc. Not every spatial relation function is supported on every database. A matrix of support for spatial relation functions may be found in the User Guide.

> If you want to play with spatial functions on H2, run the following code first:
>
> ```java
> sessionFactory.inTransaction(session -> {
>     session.doWork(connection -> {
>         try (var statement = connection.createStatement()) {
>             statement.execute("create alias if not exists h2gis_spatial for
> \"org.h2gis.functions.factory.H2GISFunctions.load\"");
>             statement.execute("call h2gis_spatial()");
>         }
>     });
> } );
> ```

9.10. Ordered and sorted collections and map keys

Java lists and maps don't map very naturally to foreign key relationships between tables, and so we tend to avoid using them to represent associations between our entity classes. But if you feel like you *really* need a collection with a fancier structure than `Set`, Hibernate does have options.

> For more detail about the use of these annotations, please refer to this post on the Hibernate blog.

The following options let us map the index of a `List` or key of a `Map` to a column, and are used with:

- `@ElementCollection`, or
- on the owning side of an association.

They should not be used on the unowned (that is, `mappedBy`) side of an association.

Table 9.7: Annotations for mapping lists and maps

Annotation	Purpose	JPA-standard
`@OrderColumn`	Specifies the column used to maintain the order of a list	✔
`@ListIndexBase`	The column value for the first element of the list (zero by default)	✖
`@MapKeyColumn`	Specifies the column used to persist the keys of a map (used when the key is of basic type)	✔
`@MapKeyJoinColumn`	Specifies the column used to persist the keys of a map (used when the key is an entity)	✔

The name of the `@OrderColumn` or `@MapKeyColumn` may be defaulted, for example:

```
@ManyToMany
@OrderColumn // order of list is persistent
List<Author> authors = new ArrayList<>();
```

But it's usually better to specify the column name explicitly:

```
@ElementCollection
@OrderColumn(name="tag_order")
@ListIndexBase(1) // order column and base value
List<String> tags;
```

Such mappings can get pretty complicated:

```
@ElementCollection
@CollectionTable(name = "author_bios",                 // table name
        joinColumns = @JoinColumn(name = "book_isbn")) // column holding foreign key of owner
@Column(name="bio")                                    // column holding map values
@MapKeyJoinColumn(name="author_ssn")                   // column holding map keys
Map<Author,String> biographies;
```

As you can imagine, we think you should use such mappings very sparingly, if at all.

For a `Map` representing an unowned `@OneToMany` association, the column holding the key of the map must also be mapped on the owning side, usually by an attribute of the target entity. In this case we use a different annotation:

Table 9.8: Annotation for mapping an entity attribute to a map key

Annotation	Purpose	JPA-standard
`@MapKey`	Specifies an attribute of the target entity which acts as the key of the map	✔

Note that `@MapKey` specifies a field or property name, not a column name.

```
@OneToMany(mappedBy = Book_.PUBLISHER)
@MapKey(name = Book_.TITLE) // the key of the map is the title of the book
Map<String,Book> booksByTitle = new HashMap<>();
```

In fact, `@MapKey` may also be used for owned collections.

Now, let's introduce a little distinction:

- an *ordered collection* is one with an ordering maintained in the database, and
- a *sorted collection* is one which is sorted in Java code.

These annotations allow us to specify how the elements of a collection should be ordered as they are read from the database:

Table 9.9: Annotations for ordered collections

Annotation	Purpose	JPA-standard
`@OrderBy`	Specifies a fragment of JPQL used to order the collection	✔
`@SQLOrder`	Specifies a fragment of SQL used to order the collection	✖

On the other hand, the following annotations specify how a collection should be sorted in memory, and are used for collections of type `SortedSet` or `SortedMap`:

Table 9.10: Annotations for sorted collections

Annotation	Purpose	JPA-standard
`@SortNatural`	Specifies that the elements of a collection are `Comparable`	✖
`@SortComparator`	Specifies a `Comparator` used to sort the collection	✖

Under the covers, Hibernate uses a `TreeSet` or `TreeMap` to maintain the collection in sorted order.

The unowned (`mappedBy`) side of a bidirectional association is not responsible for specifying column mappings. So it's wrong in principle to use `@OrderColumn` or `@MapKeyColumn` on the unowned side of an association mapping. But for unowned collections, we may use `@OrderBy` or `@MapKey` instead. That is:

- You can use `@OrderColumn` or `@MapKeyColumn` with an `@ElementCollection`, owned `@ManyToMany`, or owned `@OneToMany`.
- But use `@OrderBy` or `@MapKey` when it's an *unowned* `@ManyToMany` or `@OneToMany`.

9.11. Any mappings

An `@Any` mapping is a sort of polymorphic many-to-one association where the target entity types are not related by the usual entity inheritance. The target type is distinguished using a discriminator value stored on the *referring* side of the relationship.

This is quite different to discriminated inheritance where the discriminator is held in the tables mapped by the referenced entity hierarchy.

For example, consider an `Order` entity containing `Payment` information, where a `Payment` might be a `CashPayment` or a `CreditCardPayment`:

```
interface Payment { ... }

@Entity
class CashPayment { ... }

@Entity
class CreditCardPayment { ... }
```

In this example, `Payment` is not be declared as an entity type, and is not annotated `@Entity`. It might even be an interface, or at most just a mapped superclass, of `CashPayment` and `CreditCardPayment`. So in terms of the object/relational mappings, `CashPayment` and `CreditCardPayment` would not be considered to participate in the same entity inheritance hierarchy.

On the other hand, `CashPayment` and `CreditCardPayment` do have the same identifier type. This is important.

An `@Any` mapping would store the discriminator value identifying the concrete type of `Payment` along with the state of the associated `Order`, instead of storing it in the table mapped by `Payment`.

```
@Entity
class Order {
    ...
```

```
    @Any
    @AnyKeyJavaClass(UUID.class)   //the foreign key type
    @JoinColumn(name="payment_id") // the foreign key column
    @Column(name="payment_type")   // the discriminator column
    // map from discriminator values to target entity types
    @AnyDiscriminatorValue(discriminator="CASH", entity=CashPayment.class)
    @AnyDiscriminatorValue(discriminator="CREDIT", entity=CreditCardPayment.class)
    Payment payment;

    ...
}
```

It's reasonable to think of the "foreign key" in an `@Any` mapping as a composite value made up of the foreign key and discriminator taken together. Note, however, that this composite foreign key is only conceptual and cannot be declared as a physical constraint on the relational database table.

There are a number of annotations which are useful to express this sort of complicated and unnatural mapping:

Table 9.11: Annotations for `@Any` mappings

Annotations	Purpose
`@Any`	Declares that an attribute is a discriminated polymorphic association mapping
`@AnyDiscriminator`	Specify the Java type of the discriminator
`@JdbcType` or `@JdbcTypeCode`	Specify the JDBC type of the discriminator
`@AnyDiscriminatorValue`	Specifies how discriminator values map to entity types
`@Column` or `@Formula`	Specify the column or formula in which the discriminator value is stored
`@AnyKeyJavaType` or `@AnyKeyJavaClass`	Specify the Java type of the foreign key (that is, of the ids of the target entities)
`@AnyKeyJdbcType` or `@AnyKeyJdbcTypeCode`	Specify the JDBC type of the foreign key
`@JoinColumn`	Specifies the foreign key column

Of course, `@Any` mappings are disfavored, except in extremely special cases, since it's much more difficult to enforce referential integrity at the database level.

There's also currently some limitations around querying `@Any` associations in HQL. This is allowed:

```
from Order ord
    join CashPayment cash
        on id(ord.payment) = cash.id
```

Polymorphic association joins for `@Any` mappings are not currently implemented.

Further information may be found in the User Guide.

9.12. Selective column lists in inserts and updates

By default, Hibernate generates `insert` and `update` statements for each entity during boostrap, and reuses the same `insert` statement every time an instance of the entity is made persistent, and the same `update` statement every time an instance of the entity is modified.

This means that:

- if an attribute is `null` when the entity is made persistent, its mapped column is redundantly included in the SQL `insert`, and
- worse, if a certain attribute is unmodified when other attributes are changed, the column mapped by that attribute is redundantly included in the SQL `update`.

Most of the time, this just isn't an issue worth worrying about. The cost of interacting with the database is *usually* dominated by the cost of a round trip, not by the number of columns in the `insert` or `update`. But in cases where it does become important, there are two ways to be more selective about which columns are included in the SQL.

The JPA-standard way is to indicate statically which columns are eligible for inclusion via the `@Column` annotation. For example, if an entity is always created with an immutable `creationDate`, and with no `completionDate`, then we would write:

```
@Column(updatable=false) LocalDate creationDate;
@Column(insertable=false) LocalDate completionDate;
```

This approach works quite well in many cases, but often breaks down for entities with more than a handful of updatable columns.

An alternative solution is to ask Hibernate to generate SQL dynamically each time an `insert` or `update` is executed. We do this by annotating the entity class.

Table 9.12: Annotations for dynamic SQL generation

Annotation	Purpose
`@DynamicInsert`	Specifies that an `insert` statement should be generated each time an entity is made persistent
`@DynamicUpdate`	Specifies that an `update` statement should be generated each time an entity is modified

It's important to realize that, while `@DynamicInsert` has no impact on semantics, the more useful `@DynamicUpdate` annotation *does* have a subtle side effect.

The wrinkle is that if an entity has no version property, `@DynamicUpdate` opens the possibility of two optimistic transactions concurrently reading and selectively updating a given instance of the entity. In principle, this might lead to a row with inconsistent column values after both optimistic transactions commit successfully.

Of course, this consideration doesn't arise for entities with a `@Version` attribute.

But there's a solution! Well-designed relational schemas should have *constraints* to ensure data integrity. That's true no matter what measures we take to preserve integrity in our program logic. We may ask Hibernate to add a `check` constraint to our table using the `@Check` annotation. Check constraints and foreign key constraints can help ensure that a row never contains inconsistent column values.

9.13. Using the bytecode enhancer

Hibernate's bytecode enhancer enables the following features:

- *attribute-level lazy fetching* for basic attributes annotated `@Basic(fetch=LAZY)` and for lazy non-polymorphic associations,
- *interception-based*—instead of the usual *snapshot-based*—detection of modifications.

To use the bytecode enhancer, we must add the Hibernate plugin to our gradle build:

```
plugins {
    id "org.hibernate.orm" version "7.3.0.CR1"
}

hibernate {
  enhancement {}
}
```

Some online documentation (including previous versions of the present one) suggest to use `hibernate { enhancement }`, which will *not* work as it is interpreted by Gradle as a (pointless) getter call instead of actual configuration. That form will result in bytecode enhancement NOT happening (unfortunately silently). To enable bytecode enhancement, make sure to always use the block form (with `{}`).

Consider this field:

```
@Entity
class Book {
    ...

    @Basic(optional = false, fetch = LAZY)
    @Column(length = LONG32)
```

```
    String fullText;

    ...
}
```

The `fullText` field maps to a `clob` or `text` column, depending on the SQL dialect. Since it's expensive to retrieve the full book-length text, we've mapped the field `fetch=LAZY`, telling Hibernate not to read the field until it's actually used.

- *Without* the bytecode enhancer, this instruction is ignored, and the field is always fetched immediately, as part of the initial `select` that retrieves the `Book` entity.
- *With* bytecode enhancement, Hibernate is able to detect access to the field, and lazy fetching is possible.

> By default, Hibernate fetches all lazy fields of a given entity at once, in a single `select`, when any one of them is accessed. Using the `@LazyGroup` annotation, it's possible to assign fields to distinct "fetch groups", so that different lazy fields may be fetched independently.

Similarly, interception lets us implement lazy fetching for non-polymorphic associations without the need for a separate proxy object. However, if an association is polymorphic, that is, if the target entity type has subclasses, then a proxy is still required.

Interception-based change detection is a nice performance optimization with a slight cost in terms of correctness.

- *Without* the bytecode enhancer, Hibernate keeps a snapshot of the state of each entity after reading from or writing to the database. When the session flushes, the snapshot state is compared to the current state of the entity to determine if the entity has been modified. Maintaining these snapshots does have an impact on performance.
- *With* bytecode enhancement, we may avoid this cost by intercepting writes to the field and recording these modifications as they happen.

This optimization isn't *completely* transparent, however.

> Interception-based change detection is less accurate than snapshot-based dirty checking. For example, consider this attribute:
>
> ```
> byte[] image;
> ```
>
> Interception is able to detect writes to the `image` field, that is, replacement of the whole array. It's not able to detect modifications made directly to the *elements* of the array, and so such modifications may be lost.

> Hibernate's extended bytecode enhancement feature has been deprecated, primarily because it relies on assumptions and behaviors that often require a broader runtime scope than what Hibernate alone can reliably provide, similar to container-based environments such as Quarkus or WildFly. Applications which make use of this feature should instead use proper object-oriented encapsulation, exposing managed state via getters and setters.

9.14. Named fetch profiles

We've already seen two different ways to override the default fetching strategy for an association:

- JPA entity graphs, and
- the `join fetch` clause in HQL, or, equivalently, the method `From.fetch()` in the criteria query API.

A third way is to define a named fetch profile. First, we must declare the profile, by annotating a class or package `@FetchProfile`:

```
@FetchProfile(name = "EagerBook")
@Entity
class Book { ... }
```

Note that even though we've placed this annotation on the `Book` entity, a fetch profile—unlike an entity graph—isn't "rooted" at any particular entity.

We may specify association fetching strategies using the `fetchOverrides` member of the `@FetchProfile` annotation, but frankly it looks so messy that we're embarrassed to show it to you here.

> Similarly, a JPA entity graph may be defined using `@NamedEntityGraph`. But the format of this annotation is *even worse* than `@FetchProfile(fetchOverrides=…)`, so we can't recommend it. 💀

A better way is to annotate an association with the fetch profiles it should be fetched in:

```
@FetchProfile(name = "EagerBook")
@Entity
class Book {
    ...

    @ManyToOne(fetch = LAZY)
    @FetchProfileOverride(profile = Book_.PROFILE_EAGER_BOOK, mode = JOIN)
    Publisher publisher;

    @ManyToMany
    @FetchProfileOverride(profile = Book_.PROFILE_EAGER_BOOK, mode = JOIN)
    Set<Author> authors;

    ...
}
```

```
@Entity
class Author {
    ...

    @OneToOne
    @FetchProfileOverride(profile = Book_.PROFILE_EAGER_BOOK, mode = JOIN)
    Person person;

    ...
}
```

Here, once again, `Book_.PROFILE_EAGER_BOOK` is generated by Hibernate Processor, and is just a constant with the value `"EagerBook"`.

For collections, we may even request subselect fetching:

```
@FetchProfile(name = "EagerBook")
@FetchProfile(name = "BookWithAuthorsBySubselect")
@Entity
class Book {
    ...

    @OneToOne
    @FetchProfileOverride(profile = Book_.PROFILE_EAGER_BOOK, mode = JOIN)
    Person person;

    @ManyToMany
    @FetchProfileOverride(profile = Book_.PROFILE_EAGER_BOOK, mode = JOIN)
    @FetchProfileOverride(profile = Book_.BOOK_WITH_AUTHORS_BY_SUBSELECT,
                          mode = SUBSELECT)
    Set<Author> authors;

    ...
}
```

We may define as many different fetch profiles as we like.

Table 9.13: Annotations for defining fetch profiles

Annotation	Purpose
`@FetchProfile`	Declares a named fetch profile, optionally including a list of `@FetchOverride`s
`@FetchProfile.FetchOverride`	Declares a fetch strategy override as part of the `@FetchProfile` declaration
`@FetchProfileOverride`	Specifies the fetch strategy for the annotated association, in a given fetch profile

A fetch profile must be explicitly enabled for a given session by passing the name of the profile to `enableFetchProfile()`:

```
session.enableFetchProfile(Book_.PROFILE_EAGER_BOOK);
```

```
Book eagerBook = session.find(Book.class, bookId);
```

Alternatively, an instance of `EnabledFetchProfile` may be obtained in a type safe way from the static metamodel, and applied to the session:

```
Book_._EagerBook.enable(session);
Book eagerBook = session.find(Book.class, bookId);
```

Even better, the `EnabledFetchProfile` may be passed as a `FindOption`:

```
Book eagerBook = entityManager.find(Book.class, bookId, Book_._EagerBook);
```

So why or when might we prefer named fetch profiles to entity graphs? Well, it's really hard to say. It's nice that this feature *exists*, and if you love it, that's great. But Hibernate offers alternatives that we think are more compelling most of the time.

The one and only advantage unique to fetch profiles is that they let us very selectively request subselect fetching. We can't do that with entity graphs, and we can't do it with HQL.

> There's a special built-in fetch profile named `org.hibernate.defaultProfile` which is defined as the profile with `@FetchProfileOverride(mode=JOIN)` applied to every eager `@ManyToOne` or `@OneToOne` association. If you enable this profile:
>
> ```
> session.enableFetchProfile("org.hibernate.defaultProfile");
> ```
>
> Then `outer join`s for such associations will *automatically* be added to every HQL or criteria query. This is nice if you can't be bothered typing out those `join fetch`es explicitly. And in principle it even helps partially mitigate the problem of JPA having specified the wrong default for the `fetch` member of `@ManyToOne`.

Chapter 10. Credits

The full list of contributors to Hibernate ORM can be found on the GitHub repository.

The following contributors were involved in this documentation:

- Gavin King