

Java Persistence

with Spring Data
and Hibernate

Cătălin Tudose
Christian Bauer
Gavin King
Gary Gregory



MANNING



MEAP Edition
Manning Early Access Program
Java Persistence with Spring Data and Hibernate

Version 4

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Java Persistence with Spring Data and Hibernate*.

To get the most benefit from this book, you will need to have some established skills in programming: knowledge of object-oriented programming concepts; basic understanding of main relational database concepts (tables, relationships, primary keys, foreign keys); intermediate Java 8+ language skills (Java 11+ is recommended); basic understanding of the Apache Maven tool; basic understanding of UML and how to read simple diagrams; basic skills to open a Java program in an IDE, edit it, and launch it in execution; basic skills of how to work with the MySQL database management system.

For some particular dedicated chapters, you will need to have the basic understanding of some other techniques. We include here: the purpose of the Spring and Spring Boot frameworks; the basic understanding of the Representational State Transfer (REST) software architecture; the basic understanding of the dependency injection principle.

The book topic is Java persistence with the help of Spring Data and Hibernate. You will learn how to write code that makes the connection between the object-oriented programming world and the relational and non-relational databases world, how to persist, retrieve, update and delete the information from a database using Java programs and the Spring Data and Hibernate frameworks.

The reader will be able to use different software frameworks and tools to access relational and non-relational databases. He will focus more on the logic of the program and less on directly managing the database.

By reading this book, you'll discover the features and architecture of Spring Data and Hibernate and you'll learn how to develop safe and flexible Java applications to access both relational and non-relational databases.

It is a big book for a big library, and I hope you find it as useful to read as I did to write it. Please post any questions, comments, or suggestions you have about the book at [liveBook's Discussion Forum](#). Your feedback is essential in developing the best book possible.

-Cătălin Tudose, PhD

brief contents

PART 1: GETTING STARTED WITH ORM

- 1 Understanding object/relational persistence*
- 2 Starting a project*
- 3 Domain models and metadata*
- 4 Working with Spring Data JPA*

PART 2: MAPPING STRATEGIES

- 5 Mapping persistent classes*
- 6 Mapping value types*
- 7 Mapping inheritance*
- 8 Mapping collections and entity associations*
- 9 Advanced entity association mappings*

PART 3: TRANSACTIONAL DATA PROCESSING

- 10 Managing data*
- 11 Transactions and concurrency*
- 12 Fetch plans, strategies, and profiles*
- 13 Filtering data*

PART 4: BUILDING JAVA PERSISTENCE APPLICATIONS WITH SPRING

- 14 Integrating JPA and Hibernate with Spring*
- 15 Working with Spring Data JDBC*

16 Working with Spring Data REST

**PART 5: JAVA PERSISTENCE WITH NON-RELATIONAL
DATABASES**

17 Working with Spring Data MongoDB

18 Working with Hibernate OGM

**PART 6: WRITING QUERIES AND TESTING JAVA PERSISTENCE
APPLICATIONS**

19 Querying JPA with QueryDSL

20 Testing Java Persistence applications

1

Understanding object/relational persistence

This chapter covers

- Persisting with SQL databases in Java applications
- Analyzing the object/relational paradigm mismatch
- Introducing ORM, JPA, Hibernate, and Spring Data

Working with Java means in most cases working with objects, as the central figure of object-oriented programs. While the program is executed, it may create and manage in-memory objects, but they are getting lost when it comes to the end. Frequently, at least some part of the information they contain will be needed after a while. Saving this information means to persist it prior to stopping the execution of the program.

When an object is instantiated running an application, it becomes transient – its existence happens only inside that application. Persisting an object means making it available for the long term, even if the program terminated or stopped.

This book is about JPA, Hibernate, and Spring Data; our focus is on using Hibernate as a provider of the Jakarta Persistence API (formerly Java Persistence API) and Spring Data as a Spring-based programming model for data access. We cover basic and advanced features and describe some ways to develop new applications using Java Persistence. Often, these recommendations aren't specific to Hibernate or Spring Data. Sometimes they're our own ideas about the *best* ways to do things when working with persistent data, explained in the context of Hibernate and Spring Data.

The approach to managing persistent data may be a key design decision in many software projects. Persistence has always been a hot topic of debate in the Java community. Is persistence a problem that is already solved by SQL and extensions such as stored procedures, or is it a more pervasive problem that must be addressed by special Java

frameworks? Should we hand-code even the most primitive CRUD (create, read, update, delete) operations in SQL and JDBC, or should this work be handed to an intermediary layer? How do we achieve portability if every database management system has its own SQL dialect? Should we abandon SQL completely and adopt a different database technology, such as object database systems or NoSQL systems? The debate may never end, but a solution called *object/relational mapping* (ORM) now has wide acceptance. This is due in large part to the Hibernate, an open-source ORM service implementation, and Spring Data, an umbrella project from the Spring family whose purpose is to unify and facilitate access to different kinds of persistence stores, including here relational database systems and NoSQL databases.

Before we can get started with Hibernate and Spring Data, you need to understand the core problems of object persistence and ORM. This chapter explains why you need tools like Hibernate and Spring Data and specifications such as the *Jakarta Persistence API* (JPA).

First, we define persistent data management in the context of software applications and discuss the relationship of SQL, JDBC, and Java, the underlying technologies and standards that Hibernate and Spring Data build on. We then discuss the so-called *object/relational paradigm mismatch* and the generic problems we encounter in object-oriented software development with SQL databases. These problems make it clear that we need tools and patterns to minimize the time we have to spend on the persistence-related code in our applications.

The best way to learn Hibernate and Spring Data isn't necessarily linear. We understand that you may want to try Hibernate or Spring Data right away. If this is how you'd like to proceed, skip to the next chapter and set up a project with the "Hello World" example. We recommend that you return here at some point as you go through this book; that way, you'll be prepared and have the background concepts you need for the rest of the material.

1.1 What is persistence?

Most of the applications require persistent data. Persistence is one of the fundamental concepts in application development. If an information system didn't preserve data when it was powered off, the system would be of little practical use. *Object persistence* means individual objects can outlive the application process; they can be saved to a data store and be re-created at a later point in time. When we talk about persistence in Java, we're generally talking about mapping and storing object instances in a database using SQL. We start by taking a brief look at the technology and how it's used in Java. Armed with this information, we then continue our discussion of persistence and how it's implemented in object-oriented applications.

1.1.1 Relational databases

You, like most other software engineers, have probably worked with SQL and relational databases; many of us handle such systems every day. Relational database management systems have SQL-based application programming interfaces; hence, we call today's relational database products *SQL database management systems* (DBMS) or, when we're talking about particular systems, *SQL databases*.

Relational technology is a well-known technology, and this alone is sufficient reason for many organizations to choose it. Relational databases are strengthened because they're an incredibly flexible and robust approach to data management. Due to the well-researched theoretical foundation of the relational data model, relational databases can guarantee and protect the integrity of the stored data, among other desirable characteristics. You may be familiar with E.F. Codd's five-decades-old introduction of the relational model, *A Relational Model of Data for Large Shared Data Banks* (Codd, 1970). A more recent compendium worth reading, with a focus on SQL, is C. J. Date's *SQL and Relational Theory* (Date, 2015).

Relational DBMSs aren't specific to Java, nor is an SQL database specific to a particular application. This important principle is known as *data independence*. In other words, in most cases, *data lives longer than an application does*. Relational technology provides a way of sharing data among different applications, or among different parts of the same overall system (the data entry application and the reporting application, for example). Relational technology is a common denominator of many disparate systems and technology platforms. Hence, the relational data model is often the foundation for the common enterprise-wide representation of business entities.

Before we go into more detail about the practical aspects of SQL databases, we have to mention an important issue: although marketed as relational, a database system providing only an SQL data language interface isn't really relational and in many ways isn't even close to the original concept. Naturally, this has led to confusion. SQL practitioners blame the relational data model for shortcomings in the SQL language, and relational data management experts blame the SQL standard for being a weak implementation of the relational model and ideals. We highlight some significant aspects of this issue throughout this book, but generally, we focus on the practical aspects. If you're interested in more background material, we highly recommend *Fundamentals of Database Systems* by Ramez Elmasri and Shamkant B. Navathe (Elmasri, 2016) for the theory and concepts of relational database systems.

1.1.2 Understanding SQL

To use JPA, Hibernate, and Spring Data effectively, you must start with a solid understanding of the relational model and SQL. You need to understand the relational model and the information model and topics such as normalization to guarantee the integrity of your data, and you'll need to use your knowledge of SQL to tune the performance of your application – they are all prerequisites for reading this book. Hibernate and Spring Data simplify many repetitive coding tasks, but your knowledge of persistence technology must extend beyond the frameworks themselves if you want to take advantage of the full power of modern SQL databases. To dig deeper, consult the bibliography at the end of this book.

You've probably used SQL for many years and are familiar with the basic operations and statements written in this language. Still, we know from our own experience that SQL is sometimes hard to remember, and some terms vary in usage.

You should be comfortable with them, so let's briefly review some of the SQL terms used in this book. You use SQL as a *data definition language* (DDL), including the syntax for *creating*, *altering*, and *dropping* artifacts such as tables and constraints in the catalog of the DBMS. When this *schema* is ready, you use SQL as a *data manipulation language* (DML) to

perform operations on data, including *insertions*, *updates*, and *deletions*. You retrieve data by executing *data query language* (DQL) statements with *restrictions*, *projections*, and *Cartesian products*. For efficient reporting, you use SQL to *join*, *aggregate*, and *group* data as necessary. You can even nest SQL statements inside each other—a technique that uses *subselects*. When your business requirements change, you'll have to modify the database schema again with DDL statements after data has been stored; this is known as *schema evolution*. You may also use SQL as a *data control language* (DCL), to *grant* and *revoke* access to the database or parts of it.

If you're an SQL veteran and you want to know more about optimization and how SQL is executed, get a copy of the excellent book *SQL Tuning*, by Dan Tow (Tow, 2003). For a look at the practical side of SQL through the lens of how not to use SQL, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*, by Bill Karwin (Karwin, 2010) is a good resource.

Although the SQL database is one part of ORM, the other part, of course, consists of the data in your Java application that needs to be persisted to and loaded from the database.

1.1.3 Using SQL in Java

When you work with an SQL database in a Java application, you issue SQL statements to the database via the Java Database Connectivity (JDBC) API. Whether the SQL was written by hand and embedded in the Java code or generated on the fly by Java code, you use the JDBC API to bind arguments when preparing query parameters, executing the query, scrolling through the query result, retrieving values from the result set, and so on. These are low-level data access tasks; as application engineers, we're more interested in the business problem that requires this data access. What we'd really like to write is code that saves and retrieves instances of our classes, relieving us of this low-level labor.

Because these data access tasks are often so tedious, we have to ask: are the relational data model and (especially) SQL the right choices for persistence in object-oriented applications? We answer this question unequivocally: yes! There are many reasons why SQL databases dominate the computing industry—relational database management systems are the only proven generic data management technology, and they're almost always a *requirement* in Java projects.

Note that we aren't claiming that relational technology is *always* the best solution. Many data management requirements warrant a completely different approach. For example, internet-scale distributed systems (web search engines, content distribution networks, peer-to-peer sharing, instant messaging) have to deal with exceptional transaction volumes. Many of these systems don't require that after a data update completes, all processes see the same updated data (strong transactional consistency). Users might be happy with weak consistency; after an update, there might be a window of inconsistency before all processes see the updated data. Some scientific applications work with enormous but very specialized datasets. Such systems and their unique challenges typically require equally unique and often custom-made persistence solutions. Generic data management tools such as ACID-compliant transactional SQL databases, JDBC, Hibernate, and Spring Data would play only a minor role for these types of systems.

Relational systems at Internet scale

To understand why relational systems, and the data-integrity guarantees associated with them, are difficult to scale, we recommend that you first familiarize yourself with the *CAP theorem*. According to this rule, a distributed system cannot be *consistent*, *available*, and *tolerant against partition failures* all at the same time.

A system may guarantee that all nodes will see the same data at the same time and that data read and write requests are always answered. But when a part of the system fails due to a host, network, or data center problem, you must either give up strong consistency or 100% availability. In practice, this means you need a strategy that detects partition failures and restores either consistency or availability to a certain degree (for example, by making some part of the system temporarily unavailable for data synchronization to occur in the background). Often it depends on the data, the user, or the operation whether strong consistency is necessary.

In this book, we'll think of the problems of data storage and sharing in the context of an object-oriented application that uses a *domain model*. Instead of directly working with the rows and columns of a `java.sql.ResultSet`, the business logic of an application interacts with the application-specific object-oriented domain model. If the SQL database schema of an online auction system has `ITEM` and `BID` tables, for example, the Java application defines corresponding `Item` and `Bid` classes. Instead of reading and writing the value of a particular row and column with the `ResultSet` API, the application loads and stores instances of `Item` and `Bid` classes.

At runtime, the application, therefore, operates with instances of these classes. Each instance of a `Bid` has a reference to an auction `Item`, and each `Item` may have a collection of references to `Bid` instances. The business logic isn't executed in the database (as an SQL stored procedure); it's implemented in Java and executed in the application tier. This allows business logic to use sophisticated object-oriented concepts such as inheritance and polymorphism. For example, we could use well-known design patterns such as *Strategy*, *Mediator*, and *Composite* (see *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, 1994)), all of which depend on polymorphic method calls.

Now a warning: not all Java applications are designed this way, nor should they be. Simple applications may be much better off without a domain model. Use the JDBC `ResultSet` if that's all you need. Call existing stored procedures, and read their SQL result sets, too. Many applications need to execute procedures that modify large sets of data, close to the data. You might implement some reporting functionality with plain SQL queries and render the result directly onscreen. SQL and the JDBC API are perfectly serviceable for dealing with tabular data representations, and the JDBC `RowSet` makes CRUD operations even easier. Working with such a representation of persistent data is straightforward and well understood.

But in the case of applications with nontrivial business logic, the domain model approach helps to improve code reuse and maintainability significantly. In practice, *both* strategies are common and needed.

For several decades, developers have spoken of a *paradigm mismatch*. The *paradigms* referred to are object modeling and relational modeling, or, more practically, object-oriented

But in the case of applications with nontrivial business logic, the domain model approach helps to improve code reuse and maintainability significantly. In practice, *both* strategies are common and needed.

For several decades, developers have spoken of a *paradigm mismatch*. The *paradigms* referred to are object modeling and relational modeling, or, more practically, object-oriented programming and SQL. This mismatch explains why every enterprise project expends so much effort on persistence-related concerns.

With this conception, you can begin to see the problems—some well understood and some less well understood—that an application that combines both data representations must solve: an object-oriented domain model and a persistent relational model. Let’s take a closer look at this so-called paradigm mismatch.

1.2 The paradigm mismatch

The object/relational paradigm mismatch can be broken into several parts, which we examine one at a time. Let’s start our exploration with a simple example that is problem-free. As we build on it, you’ll see the mismatch begins to appear.

Suppose you have to design and implement an online e-commerce application. In this application, you need a class to represent information about a user of the system, and you need another class to represent information about the user’s billing details, as shown in figure 1.1.

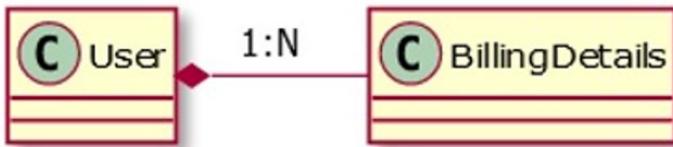


Figure 1.1 A simple UML diagram of the User and BillingDetails entities.

In this diagram, you can see that a `User` has many `BillingDetails`. This is a composition, indicated by the full diamond. A composition is that type of association where an object (`BillingDetails` in our case) cannot conceptually exist without the container (`User` in our case). You can navigate the relationship between the classes in both directions; this means you can iterate through collections or call methods to get to the “other” side of the relationship. The classes representing these entities may be extremely simple:

```

Path: Ch01/e-commerce/src/com/manning/javapersistence/ch01/User.java

public class User {
    private String username;
    private String address;
    private Set<BillingDetails> billingDetails = new HashSet<>();

    // Constructor, accessor methods (getters/setters), business methods
}
Path: Ch01/e-commerce/src/com/manning/javapersistence/ch01/BillingDetails.java

public class BillingDetails {
    private String account;
    private String bankname;
    private User user;

    // Constructor, accessor methods (getters/setters), business methods
}

```

Note that you're only interested in the state of the entities concerning persistence, so we've omitted the implementation of constructors, accessor methods, and business methods.

It's easy to come up with an SQL schema design for this case:

```

CREATE TABLE USERS (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    ADDRESS VARCHAR(255) NOT NULL
);

CREATE TABLE BILLINGDETAILS (
    ACCOUNT VARCHAR(15) NOT NULL PRIMARY KEY,
    BANKNAME VARCHAR(255) NOT NULL,
    USERNAME VARCHAR(15) NOT NULL,
    FOREIGN KEY (USERNAME) REFERENCES USERS(USERNAME)
);

```

The foreign key-constrained column `USERNAME` in `BILLINGDETAILS` represents the relationship between the two entities. For this simple domain model, the object/relational mismatch is barely in evidence; it's straightforward to write JDBC code to insert, update, and delete information about users and billing details.

Now let's see what happens when you consider something a little more realistic. The paradigm mismatch will be visible when you add more entities and entity relationships to your application.

1.2.1 The problem of granularity

The most obvious problem with the current implementation is that you've designed an address as a simple `String` value. In most systems, it's necessary to store street, city, state, country, and ZIP code information separately. Of course, you could add these properties directly to the `User` class, but because, likely, other classes in the system will also carry address information, it makes more sense to create an `Address` class to reuse it. Figure 1.2 shows the updated model.

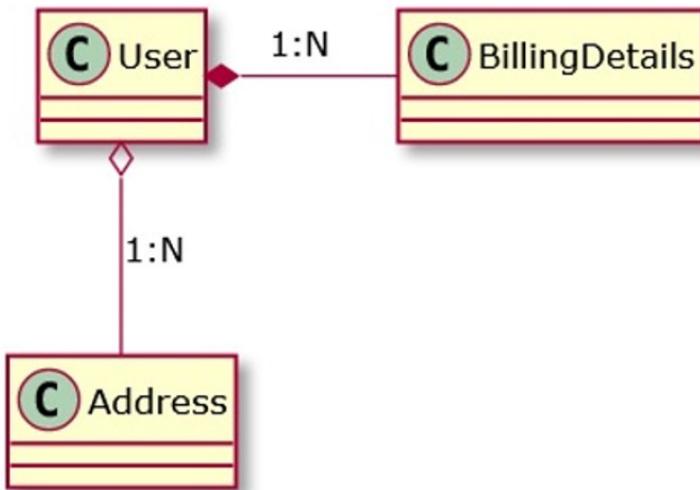


Figure 1.2 The User has an Address. **Figure 1.5** Many-to-many association between User and BillingDetails

Should you also add an ADDRESS table? Not necessarily; it's common to keep address information in the USERS table, in individual columns. This design is likely to perform better because a table join isn't needed if you want to retrieve the user and address in a single query. The nicest solution may be to create a new SQL data type to represent addresses and to add a single column of that new type in the USERS table instead of several new columns.

You have the choice of adding either several columns or a single column (of a new SQL data type). This is clearly a problem of *granularity*. Broadly speaking, granularity refers to the relative size of the types you're working with.

Let's return to the example. Adding a new data type to the database catalog, to store Address Java instances in a single column, sounds like the best approach:

```

CREATE TABLE USERS (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    ADDRESS ADDRESS NOT NULL
);
  
```

A new Address type (class) in Java and a new ADDRESS SQL data type should guarantee interoperability. But you'll find various problems if you check the support for user-defined data types (UDTs) in today's SQL database management systems.

UDT support is one of several so-called *object-relational extensions* to traditional SQL. This term alone is confusing because it means the database management system has (or is supposed to support) a sophisticated data type system. Unfortunately, UDT support is a somewhat obscure feature of most SQL DBMSs and certainly isn't portable between different products. Furthermore, the SQL standard supports user-defined data types, but poorly.

This limitation isn't the fault of the relational data model. You can consider the failure to standardize such an important piece of functionality as a result of the object-relational database wars between vendors in the mid-1990s. Today, most engineers accept that SQL products have limited-type systems—no questions asked. Even with a sophisticated UDT system in your SQL DBMS, you would still likely duplicate the type declarations, writing the new type in Java and again in SQL. Attempts to find a better solution for the Java space, such as SQLJ, unfortunately, have not had much success. DBMS products rarely support deploying and executing Java classes directly on the database, and if support is available, it's typically limited to very basic functionality in everyday usage.

For these and whatever other reasons, the use of UDTs or Java types in an SQL database isn't common practice in the industry at this time, and it's unlikely that you'll encounter a legacy schema that makes extensive use of UDTs. You therefore can't and won't store instances of your new `Address` class in a single new column that has the same data type as the Java layer.

The pragmatic solution for this problem has several columns of built-in vendor-defined SQL types (such as boolean, numeric, and string data types). You usually define the `USERS` table as follows:

```
CREATE TABLE USERS (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    ADDRESS_STREET VARCHAR(255) NOT NULL,
    ADDRESS_ZIPCODE VARCHAR(5) NOT NULL,
    ADDRESS_CITY VARCHAR(255) NOT NULL
);
```

Classes in the Java domain model come in a range of different levels of granularity: from coarse-grained entity classes like `User` to finer-grained classes like `Address`, down to simple `SwissZipCode` extending `AbstractNumericZipCode` (or whatever your desired level of abstraction is). In contrast, just two levels of type granularity are visible in the SQL database: relation types created by you, like `USERS` and `BILLINGDETAILS`, and built-in data types such as `VARCHAR`, `BIGINT`, or `TIMESTAMP`.

Many simple persistence mechanisms fail to recognize this mismatch and so end up forcing the less flexible representation of SQL products on the object-oriented model, effectively flattening it.

It turns out that the granularity problem isn't especially difficult to solve, even if it's visible in so many existing systems. We describe the solution to this problem in section 4.1.

A much more difficult and interesting problem arises when we consider domain models that rely on *inheritance*, a feature of object-oriented design you may use to bill the users of your e-commerce application in new and interesting ways.

1.2.2 The problem of inheritance

In Java, you implement type inheritance using super-classes and subclasses. To illustrate why this can present a mismatch problem, let's modify your e-commerce application so that you now can accept not only bank account billing, but also credit cards. The most natural way to reflect this change in the model is to use inheritance for the `BillingDetails` superclass, along with several concrete subclasses: `CreditCard`, `BankAccount`. Each of these

subclasses defines slightly different data (and completely different functionality that acts on that data). The UML class diagram in figure 1.3 illustrates this model.

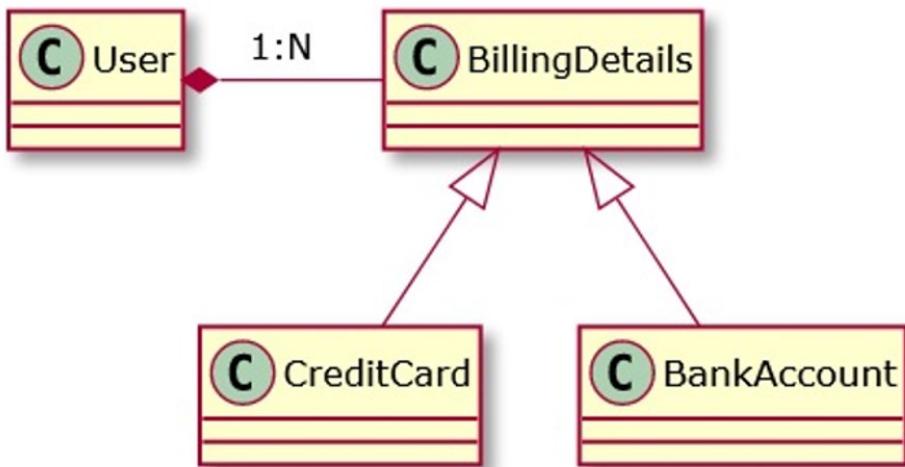


Figure 1.3 Using inheritance for different billing strategies **Figure 1.5 Many-to-many association between User and BillingDetails**

What changes must you make to support this updated Java class structure? Can you create a table `CREDITCARD` that *extends* `BILLINGDETAILS`? SQL database products don't generally implement table inheritance (or even data type inheritance), and if they do implement it, they don't follow a standard syntax.

We haven't finished with inheritance. As soon as we introduce inheritance into the model, we have the possibility of *polymorphism*.

The `User` class has an association with the `BillingDetails` superclass. This is a *polymorphic association*. At runtime, a `User` instance may reference an instance of any of the subclasses of `BillingDetails`. Similarly, you want to be able to write *polymorphic queries* that refer to the `BillingDetails` class and have the query return instances of its subclasses.

SQL databases also lack an obvious way (or at least a standardized way) to represent a polymorphic association. A foreign key constraint refers to exactly one target table; it isn't straightforward to define a foreign key that refers to multiple tables.

The result of this mismatch of subtypes is that the inheritance structure in a model must be persisted in an SQL database that doesn't offer an inheritance mechanism. In chapter 6, we discuss how ORM solutions such as Hibernate solve the problem of persisting a class hierarchy to an SQL database table or tables, and how polymorphic behavior can be implemented. Fortunately, this problem is now well understood in the community, and most solutions support approximately the same functionality.

The next aspect of the object/relational mismatch problem is the issue of *object identity*.

1.2.3 The problem of identity

You probably noticed that the example defined `USERNAME` as the primary key of the `USERS` table. Was that a good choice? How do you handle identical objects in Java?

Although the problem of identity may not be obvious at first, you'll encounter it often in your growing and expanding e-commerce system, such as when you need to check whether two instances are identical. There are three ways to tackle this problem: two in the Java world and one in your SQL database. As expected, they work together only with some help.

Java defines two different notions of *sameness*:

- Instance identity (roughly equivalent to a memory location, checked with `a == b`)
- Instance equality, as determined by the implementation of the `equals()` method (also called *equality by value*)

On the other hand, the identity of a database row is expressed as a comparison of primary key values. As you'll see in section 9.1.2, neither `equals()` nor `==` is always equivalent to a comparison of primary key values. It's common for several non-identical instances in Java to simultaneously represent the same row of the database—for example, in concurrently running application threads. Furthermore, some subtle difficulties are involved in implementing `equals()` correctly for a persistent class and understanding when this might be necessary.

Let's use an example to discuss another problem related to database identity. In the table definition for `USERS`, `USERNAME` is the primary key. Unfortunately, this decision makes it difficult to change a user's name; you need to update not only the row in `USERS` but also the foreign key values in (many) rows of `BILLINGDETAILS`. To solve this problem, later in this book we recommend that you use *surrogate keys* whenever you can't find a good natural key. We also discuss what makes a good primary key. A surrogate key column is a primary key column with no meaning to the application user—in other words, a key that isn't presented to the application user. Its only purpose is to identify data inside the application.

For example, you may change your table definitions to look like this:

```

CREATE TABLE USERS (
    ID BIGINT NOT NULL PRIMARY KEY,
    USERNAME VARCHAR(15) NOT NULL UNIQUE,
    ...
);
CREATE TABLE BILLINGDETAILS (
    ID BIGINT NOT NULL PRIMARY KEY,
    ACCOUNT VARCHAR(15) NOT NULL,
    BANKNAME VARCHAR(255) NOT NULL,
    USER_ID BIGINT NOT NULL,
    FOREIGN KEY (USER_ID) REFERENCES USERS(ID)
);

```

The `ID` columns contain system-generated values. These columns were introduced purely for the benefit of the data model, so how (if at all) should they be represented in the Java domain model? We discuss this question in section 4.2, and we find a solution with ORM.

In the context of persistence, identity is closely related to how the system handles caching and transactions. Different persistence solutions have chosen different strategies, and this has been an area of confusion. We cover all these interesting topics—and show how they’re related—in section 9.1.

So far, the skeleton e-commerce application you’ve designed has exposed the paradigm mismatch problems with mapping granularity, subtypes, and identity. We need to discuss further the important concept of *associations*: how the relationships between entities are mapped and handled. Is the foreign key constraint in the database all you need?

1.2.4 Problems relating to associations

In your domain model, associations represent the relationships between entities. The `User`, `Address`, and `BillingDetails` classes are all associated; but unlike `Address`, `BillingDetails` stands on its own. `BillingDetails` instances are stored in their own table. Association mapping and the management of entity associations are central concepts in any object persistence solution.

Object-oriented languages represent associations using *object references*; but in the relational world, a *foreign key-constrained column* represents an association, with copies of key values. The constraint is a rule that guarantees the integrity of the association. There are substantial differences between the two mechanisms.

Object references are inherently directional; the association is from one instance to the other. They’re pointers. If an association between instances should be navigable in both directions, you must define the association *twice*, once in each of the associated classes. The UML class diagram in figure 1.4 illustrates this model, with a one-to-many association.



Figure 1.4 One-to-many association between `User` and `BillingDetails`

You've already seen this in the domain model classes:

```
Path: Ch01/e-commerce/src/com/manning/javapersistence/ch01/User.java
```

```
public class User {
    private Set<BillingDetails> billingDetails;
}
```

```
Path: Ch01/e-commerce/src/com/manning/javapersistence/ch01/BillingDetails.java
```

```
public class BillingDetails {  
}
```

Navigation in a particular direction has no meaning for a relational data model because you can create data associations with *join* and *projection* operators. The challenge is to map a completely open data model, which is independent of the application that works with the data, to an application-dependent navigational model—a constrained view of the associations needed by this particular application.

Java associations can have *many-to-many* multiplicity. The UML class diagram in figure 1.4 illustrates this model.



Figure 1.5 Many-to-many association between User and BillingDetails

The classes could look like this:

```
Path: Ch01/e-commerce/src/com/manning/javapersistence/ch01/User.java

public class User {
    private Set<BillingDetails> billingDetails;
}

Path: Ch01/e-commerce/src/com/manning/javapersistence/ch01/BillingDetails.java

public class BillingDetails {
    private Set<User> users;
}
```

But the foreign key declaration on the `BILLINGDETAILS` table is a *many-to-one* association: each bank account is linked to a particular user. Each user may have multiple linked bank accounts.

If you wish to represent a *many-to-many* association in an SQL database, you must introduce a new table, usually called a *link table*. In most cases, this table doesn't appear anywhere in the domain model. For this example, if you consider the relationship between the user and the billing information to be *many-to-many*, you define the link table as follows:

```
CREATE TABLE USER_BILLINGDETAILS (
    USER_ID BIGINT,
    BILLINGDETAILS_ID BIGINT,
    PRIMARY KEY (USER_ID, BILLINGDETAILS_ID),
    FOREIGN KEY (USER_ID) REFERENCES USERS(ID),
    FOREIGN KEY (BILLINGDETAILS_ID) REFERENCES BILLINGDETAILS(ID)
);
```

You no longer need the `USER_ID` foreign key column and constraint on the `BILLINGDETAILS` table; this additional table now manages the links between the two entities. We discuss association and collection mappings in detail in chapter 7.

So far, the issues we've considered are mainly *structural*: you can see them by considering a purely static view of the system. Perhaps the most difficult problem in object persistence is a *dynamic* problem: how data is accessed at runtime.

1.2.5 The problem of data navigation

There is a fundamental difference in how you access data in Java code and within a relational database. In Java, when you access a user's billing information, you call `someUser.getBillingDetails().iterator().next()` or something similar. Starting from Java 8, you may call `someUser.getBillingDetails().stream()`.

`filter(someCondition).map(someMapping).forEach(billingDetails->{doSomething(billingDetails)}).` This is the most natural way to access object-oriented data, and it's often described as *walking the object network*. You navigate from one instance to another, even iterating collections, following prepared pointers between classes. Unfortunately, this isn't an efficient way to retrieve data from an SQL database.

The single most important thing you can do to improve the performance of data access code is to *minimize the number of requests to the database*. The most obvious way to do this

is to minimize the number of SQL queries. (Of course, other, more sophisticated, ways—such as extensive caching—follow as a second step.)

Therefore, efficient access to relational data with SQL usually requires joins between the tables of interest. The number of tables included in the join when retrieving data determines the depth of the object network you can navigate in memory. For example, if you need to retrieve a `User` and aren't interested in the user's billing information, you can write this simple query:

```
SELECT * FROM USERS WHERE ID = 123
```

On the other hand, if you need to retrieve a `User` and then subsequently visit each of the associated `BillingDetails` instances (let's say, to list all the user's bank accounts), you write a different query:

```
SELECT * FROM USERS, BILLINGDETAILS
WHERE USERS.ID = 123 AND
BILLINGDETAILS.ID = USERS.ID
```

As you can see, to use joins efficiently you need to know what portion of the object network you plan to access when you retrieve the initial instance *before* you start navigating the object network! Careful, though: if you retrieve too much data (probably more than you might need), you're wasting memory in the application tier. You may also overwhelm the SQL database with huge *Cartesian product* result sets. Imagine retrieving not only users and bank accounts in one query, but also all orders paid from each bank account, the products in each order, and so on.

Any object persistence solution provides functionality for fetching the data of associated instances only when the association is first accessed in the Java code. This is known as *lazy loading*: retrieving data on-demand only. This piecemeal style of data access is fundamentally inefficient in the context of an SQL database because it requires executing one statement for each node or collection of the object network that is accessed. This is the dreaded *n+1 selects* problem. In our example, you will need one `select` to retrieve a `User` and then subsequently *n* `selects` for each of the *n* associated `BillingDetails` instances.

This mismatch in the way you access data in Java code and within a relational database is perhaps the single most common source of performance problems in Java information systems. Avoiding the *Cartesian product* and *n+1 selects* problems is still a problem for many Java programmers. Hibernate provides sophisticated features for efficiently and transparently fetching networks of objects from the database to the application accessing them. We discuss these features in chapter 11.

We now have quite a list of object/relational mismatch problems: the problem of granularity, the problem of inheritance, the problem of identity, the problem of associations, and the problem of data navigation. It can be costly (in time and effort) to find solutions, as you may know from experience. It will take us a large part of this book to provide a complete answer to these questions and to demonstrate ORM as a viable solution. Let's get started with an overview of ORM, the Java Persistence standard, and the Hibernate and Spring Data projects.

1.3 ORM, JPA, Hibernate, and Spring Data

In a nutshell, object/relational mapping (ORM) is the automated (and transparent) persistence of objects in a Java application to the tables in an RDBMS (Relational Database Management System), using metadata that describes the mapping between the classes of the application and the schema of the SQL database. In essence, ORM works by transforming (reversibly) data from one representation to another. Your program using ORM will provide the meta-information about how to map the objects from the memory to the database and the effective transformation will be fulfilled by ORM.

An advantage of ORM that some people may think about is that it shields developers from messy SQL. This view holds that object-oriented developers can't be expected to go deep into SQL or relational databases. On the contrary, we believe that Java developers must have a sufficient level of familiarity with—and appreciation of—relational modeling and SQL to work with Hibernate and Spring Data. ORM is an advanced technique used by developers who have already done it the hard way.

JPA (Jakarta Persistence API, formerly Java Persistence API) is a specification defining an API that takes care of managing the persistence of objects and object-relational mappings. Hibernate is the most popular implementation of this specification. So, JPA will tell what to do to persist objects, while Hibernate will tell how to do it. Spring Data Commons, as part of the Spring Data family, provides the core Spring framework concepts that support all Spring Data modules. Spring Data JPA, another project from the Spring Data family, is an additional layer on top of JPA implementations (as Hibernate). Not only that Spring Data JPA can use all capabilities of JPA, but it adds its own capabilities like the creation of database queries generated from method names. We'll go into many details across this book, but if you would like to have an overall view at this time, you can quickly jump to figure 4.1.

To use Hibernate effectively, you must be able to view and interpret the SQL statements it issues and understand their performance implications. To take advantage of the benefits of Spring Data, you must be able to anticipate how the boilerplate code and the generated queries are created.

The JPA specification defines the following:

- A facility for specifying mapping metadata—how persistent classes and their properties relate to the database schema. JPA relies heavily on Java annotations in domain model classes, but you can also write mappings in XML files.
- APIs for performing basic CRUD operations on instances of persistent classes, most prominently `javax.persistence.EntityManager` to store and load data.
- A language and APIs for specifying queries that refer to classes and properties of classes. This language is the Jakarta Persistence Query Language (JPQL) and looks similar to SQL. The standardized API allows for the programmatic creation of *criteria queries* without string manipulation.
- How the persistence engine interacts with transactional instances to perform dirty checking, association fetching, and other optimization functions. The JPA specification covers some basic caching strategies.

Hibernate implements JPA and supports all the standardized mappings, queries, and programming interfaces.

Let's look at some of the benefits of Hibernate:

- *Productivity*—Hibernate eliminates much of the repetitive work (more than you'd expect) and lets you concentrate on the business problem. No matter which application-development strategy you prefer—top-down, starting with a domain model, or bottom-up, starting with an existing database schema—Hibernate, used together with the appropriate tools, will significantly *reduce development time*.
- *Maintainability*—Automated ORM with Hibernate reduces lines of code (LOC), making the system *more understandable* and *easier to refactor*. Hibernate provides a buffer between the domain model and the SQL schema, isolating each model from minor changes to the other.
- *Performance*—Although hand-coded persistence might be faster in the same sense that assembly code can be faster than Java code, automated solutions like Hibernate allow the use of many optimizations *at all times*. One example is the efficient and easily tunable caching in the application tier. This means developers can spend more energy hand-optimizing the few remaining real bottlenecks instead of prematurely optimizing everything.
- *Vendor independence*—Hibernate can help mitigate some of the risks associated with vendor lock-in. Even if you plan never to change your DBMS product, ORM tools that support several different DBMSs enable *a certain level of portability*. Also, DBMS independence helps in development scenarios where *engineers use a lightweight local database* but deploy for testing and production on a different system.

Spring Data makes the implementation of the persistence layer even more efficient. Spring Data JPA, one of the projects of the family, sits on top of the JPA layer. Spring Data JDBC, another project of the family, sits on top of JDBC.

Let's look at some of the benefits of Spring Data:

- *Shared infrastructure* – *Spring Data Commons*, part of the umbrella Spring Data project, provides a metadata model for persisting Java classes and technology-neutral repository interfaces. It provides its capabilities to the other Spring Data projects.
- *Removes DAO implementations*—JPA implementations use the DAO (*Data Access Object*) pattern. This pattern starts with the idea of an abstract interface to a database and maps application calls to the persistence layer while hiding the details of the database. Spring Data JPA makes it possible to fully remove DAO implementations, so the code will be shorter.
- *Automatic class creation*—Using Spring Data JPA, a DAO interface needs to extend the JPA specific Repository interface – `JpaRepository`. Spring Data JPA will automatically create an implementation for this interface, the programmer will not have to take care of this.

- *Default implementations for methods*—Spring Data JPA will generate default implementations for each method defined by its repository interfaces. Basic CRUD operations do not need to be implemented any longer. This reduces the boilerplate code, speeds up development, and removes the possibility of introducing bugs.
- *Generated queries*—You may define a method on your repository interface following a naming pattern. No need to write your queries by hand, Spring Data JPA will parse the method name and will create a query for it.
- *Close to the database if needed*—Spring Data JDBC can communicate directly to the database and avoids the “magic” of Spring Data JPA. It allows you to interact with the database through JDBC but removes the boilerplate code using the Spring framework facilities.

This chapter has focused on understanding the object/relational persistence and the problems generated by the object/relational paradigm mismatch.

Chapter 2 will look at some of the persistence alternatives from a Java application: JPA, Hibernate native, and Spring Data JPA.

1.4 Summary

- With *object persistence*, individual objects can outlive their application process, be saved to a data store, and be re-created later. The object/relational mismatch comes into play when the datastore is an SQL-based relational database management system. For instance, a network of objects can't be saved to a database table; it must be disassembled and persisted to columns of portable SQL data types. A good solution to this problem is object/relational mapping (ORM).
- ORM isn't a silver bullet for all persistence tasks; its job is to relieve the developer of 95% of object persistence work, such as writing complex SQL statements with many table joins and copying values from JDBC result sets to objects or graphs of objects.
- A full-featured ORM middleware solution may provide database portability, certain optimization techniques like caching, and other viable functions that aren't easy to hand-code in a limited time with SQL and JDBC. An ORM solution implies, in the Java world, the JPA specification and a JPA implementation – Hibernate being the most popular nowadays.
- Spring Data may come on top of the JPA implementations and simplifies, even more, the data persistence process. It is an umbrella project that adheres to the Spring framework principles and comes with an even simpler approach, including here removing the DAO pattern, automatic code generation, and automatic queries generation.

2

Starting a project

This chapter covers

- Overviewing the Hibernate and Spring Data projects
- Developing a “Hello World” with Jakarta Persistence API, Hibernate, and Spring Data
- Examining the configuration and integration options

In this chapter, we’ll start with Jakarta Persistence API (JPA), Hibernate, and Spring Data using a step-by-step example. We’ll see the persistence APIs and how to benefit from using either standardized JPA, native Hibernate, or Spring Data. We first offer a tour through JPA, Hibernate, and Spring Data with a straightforward “Hello World” application. We remind that JPA (Jakarta Persistence API, formerly Java Persistence API), is the specification defining an API that takes care of managing the persistence of objects and object-relational mappings – it will tell what to do to persist objects. Hibernate, the most popular implementation of this specification, will tell how to effectively make the persistence. Spring Data makes the implementation of the persistence layer even more efficient and is an umbrella project that adheres to the Spring framework principles and comes with an even simpler approach.

2.1 Introducing Hibernate

Object-relational mapping (ORM) is a programming technique making the connection between the incompatible worlds of object-oriented systems and relational databases.

Hibernate is an ambitious project that aims to provide a complete solution to the problem of managing persistent data in Java. Today, Hibernate is not only an ORM service but also a collection of data management tools extending well beyond ORM.

The Hibernate project suite includes the following:

- *Hibernate ORM*—Hibernate ORM consists of a core, a base service for persistence with SQL databases, and a native proprietary API. Hibernate ORM is the foundation for

several of the other projects and is the oldest Hibernate project. We can use Hibernate ORM on its own, independent of any framework or any particular runtime environment with all JDKs. As long as a data source is accessible, we can configure it for Hibernate, and it works.

- *Hibernate EntityManager*—This is Hibernate’s implementation of the standard Jakarta Persistence APIs, an optional module we can stack on top of Hibernate ORM. Hibernate’s native features are a superset of the JPA persistence features in every respect.
- *Hibernate Validator*—Hibernate provides the reference implementation of the Bean Validation (JSR 303) specification. Independent of other Hibernate projects, it provides declarative validation for our domain model (or any other) classes.
- *Hibernate Envers*—Envers is dedicated to audit logging and keeping multiple versions of data in the SQL database. This helps add data history and audit trails to the application, similar to version control systems you might already be familiar with, such as Subversion and Git.
- *Hibernate Search*—Hibernate Search keeps an index of the domain model data up to date in an Apache Lucene database. It lets us query this database with a powerful and naturally integrated API. Many projects use Hibernate Search in addition to Hibernate ORM, adding full-text search capabilities. If you have a free text search form in your application’s user interface, and you want happy users, work with Hibernate Search. Hibernate Search isn’t covered in this book; you can find more information in *Hibernate Search in Action* by Emmanuel Bernard (Bernard, 2008).
- *Hibernate OGM*—This Hibernate project is an object/grid mapper. It provides JPA support for NoSQL solutions, reusing the Hibernate core engine but persisting mapped entities into a key/value-, document-, or graph-oriented data store.
- *Hibernate Reactive*—Hibernate Reactive is a reactive API for Hibernate ORM, interacting with a database in a non-blocking manner. It supports non-blocking database drivers. Hibernate Reactive isn’t covered in this book.

The Hibernate source code is freely downloadable from <https://github.com/hibernate>.

2.2 Introducing Spring Data

Spring Data is a family of projects belonging to the Spring framework whose purpose is to simplify the access to both relational and NoSQL databases.

- *Spring Data Commons*—*Spring Data Commons*, part of the umbrella Spring Data project, provides a metadata model for persisting Java classes and technology-neutral repository interfaces.

- *Spring Data JPA*—Spring Data JPA deals with the implementation of JPA-based repositories. It provides improved support for JPA-based data access layers by reducing the boilerplate code and creating implementations for the repository interfaces.
- *Spring Data JDBC*—Spring Data JDBC deals with the implementation of JDBC-based repositories. It provides improved support for JDBC-based data access layers. It does not offer a series of JPA capabilities as caching or lazy loading, resulting in a simpler and limited ORM.
- *Spring Data REST* – *Spring Data REST deals with exporting Spring Data repositories as RESTful resources.*
- *Spring Data MongoDB*—Spring Data MongoDB deals with access to the MongoDB document database. It relies on the repository style data access layer and the POJO programming model.
- *Spring Data Redis*—Spring Data Redis deals with access to the Redis key-value database. It relies on freeing the developer from managing the infrastructure and providing high-level and low-level abstractions for access to the datastore. Spring Data Redis isn't covered in this book.

The Spring Data source code (together with other Spring projects) is freely downloadable from <https://github.com/spring-projects>.

Let's get started with our first JPA, Hibernate, and Spring Data project.

2.3 “Hello World” with JPA

In this section, we'll write our first JPA application, which stores a message in the database and then retrieves it. The machine we are running the code on has MySQL Release 8.0 installed. To install MySQL Release 8.0, you may follow the instructions from the official documentation: <https://dev.mysql.com/doc/refman/8.0/en/installing.html>.

To be able to execute the examples from the source code, you need first to run the Ch02.sql script (figure 2.1). You may open MySQL Workbench, then go to File -> Open SQL Script, choose the SQL file and run it. The examples are using a MySQL server with the default credentials: username `root` and no password.

In the “Hello World” application, we want to store messages in the database and load them from the database. Hibernate applications define persistent classes that are mapped to database tables. We define these classes based on our analysis of the business domain; hence, they're a model of the domain. This example consists of one class and its mapping. You'll notice that we write examples as executable tests, with assertions that verify the correct outcome of each operation. We've tested all the examples in this book, so we can be sure they work properly.

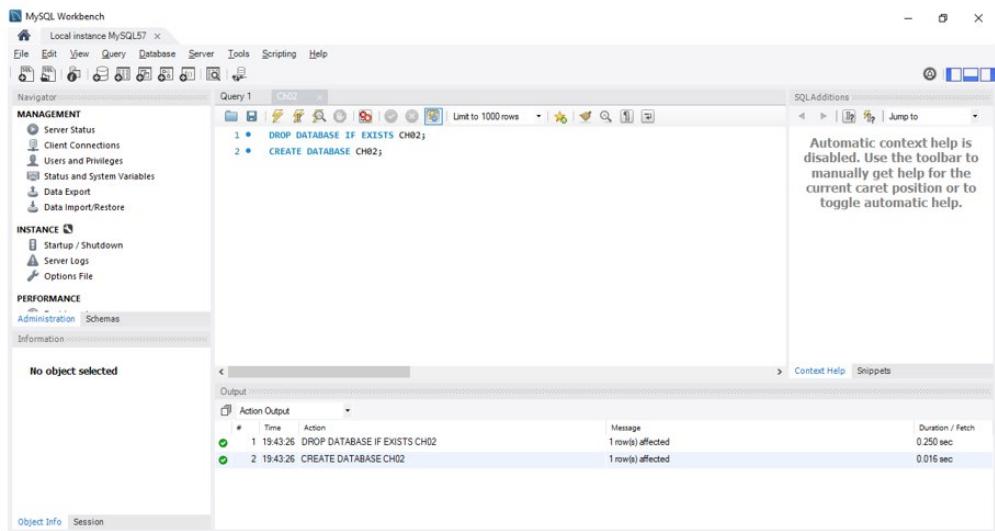


Figure 2.1 The creation of the MySQL database running the Ch02.sql script

Let's start by installing and configuring JPA, Hibernate, and the other needed dependencies. We use Apache Maven as the project build tool, as we do for all the examples in this book. For basic Maven concepts and how to set up Maven, see appendix A.

We declare these dependencies:

Listing 2.1 The Maven dependencies on Hibernate, JUnit Jupiter, and MySQL

Path: Ch02/helloworld/pom.xml

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.4.17.Final</version>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.20</version>
</dependency>
```

The hibernate-entitymanager module includes transitive dependencies on other modules we'll need, such as hibernate-core and the JPA interface stubs.

We also need the `junit-jupiter-engine` dependency, to run the tests with the help of JUnit 5, and the `mysql-connector-java` dependency, the official JDBC driver for MySQL.

Our starting point in JPA is the *persistence unit*. A persistence unit is a pairing of our domain model class mappings with a database connection, plus some other configuration settings. Every application has at least one persistence unit; some applications have several if they're talking to several (logical or physical) databases. Hence, our first step is setting up a persistence unit in our application's configuration.

2.3.1 Configuring a persistence unit

The standard configuration file for persistence units is located on the classpath in `META-INF/persistence.xml`. Create the following configuration file for the "Hello World" application:

Listing 2.2 The `persistence.xml` configuration file

```
Path: Ch02/helloworld/src/main/resources/META-INF/persistence.xml

<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
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

    <persistence-unit name="ch02.ex01"> #A
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider> #B
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/> #C
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/CH02"/> #D
            <property name="javax.persistence.jdbc.user" value="root"/> #E
            <property name="javax.persistence.jdbc.password" value="" /> #F

            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/> #G
            <property name="hibernate.show_sql" value="true"/> #H
            <property name="hibernate.format_sql" value="true"/> #I

            <property name="hibernate.hbm2ddl.auto" value="create"/> #J
        </properties>
    </persistence-unit>

</persistence>
```

#A The `persistence.xml` file configures at least one persistence unit; each unit must have a unique name.

#B As JPA is only a specification, we need to indicate the vendor-specific `PersistenceProvider` implementation of the API. The persistence we define will be backed by a Hibernate provider.

#C We indicate the JDBC properties - the driver.

#D The URL of the database.

#E The username.

#F No password for the access. The machine we are running the programs on has MySQL 8 installed and the access credentials are the ones from `persistence.xml`. You should modify the credentials to correspond to the ones on your machine.

```
#G The Hibernate dialect is MySQL8, as the database to interact with is MySQL Release 8.0.
#H While executing, show the SQL code.
#I Hibernate will format the SQL nicely and generate comments into the SQL string so we know why Hibernate
    executed the SQL statement.
#J Every time the program is executed, the database will be created from scratch. This is ideal for automated testing,
    when we want to work with a clean database for every test run.
```

Let's see what a simple persistent class looks like, how the mapping is created, and some of the things we can do with instances of the persistent class in JPA.

2.3.2 Writing a persistent class

The objective of this example is to store messages in a database and retrieve them for display. The application has a simple persistent class, `Message`:

Listing 2.3 The Message class

```
Path: Ch02/helloworld/src/main/java/com/manning/javapersistence/ch02/Message.java

@Entity                                     #A
public class Message {                     

    @Id                                       #B
    @GeneratedValue                         #C
    private Long id;                        

    private String text;                     #D

    public String getText() {                #D
        return text;                        #D
    }                                         #D

    public void setText(String text) {       #D
        this.text = text;                   #D
    }                                         #D

}
```

#A Every persistent entity class must have at least the `@Entity` annotation. Hibernate maps this class to a table called `MESSAGE`.

#B Every persistent entity class must have an identifier attribute annotated with `@Id`. Hibernate maps this attribute to a column named `id`.

#C Someone must generate identifier values; this annotation enables automatic generation of `ids`.

#D We usually implement regular attributes of a persistent class with private fields and public getter/setter method pairs. Hibernate maps this attribute to a column called `text`.

The identifier attribute of a persistent class allows the application to access the database identity—the primary key value—of a persistent instance. If two instances of `Message` have the same identifier value, they represent the same row in the database.

This example uses `Long` for the type of identifier attribute, but this isn't a requirement. Hibernate allows virtually anything for the identifier type, as we'll see later in the book.

You may have noticed that the `text` attribute of the `Message` class has JavaBeans-style property accessor methods. The class also has a (default) constructor with no parameters.

The persistent classes we show in the examples will usually look something like this. Note that we don't need to implement any particular interface or extend any special superclass.

Instances of the `Message` class can be managed (made persistent) by Hibernate, but they don't have to be. Because the `Message` object doesn't implement any persistence-specific classes or interfaces, we can use it just like any other Java class:

```
Message msg = new Message();
msg.setText("Hello!");
System.out.println(msg.getText());
```

It may look like we're trying to be cute here; in fact, we're demonstrating an important feature that distinguishes Hibernate from some other persistence solutions. We can use the persistent class in any execution context—*no special container is needed*.

We don't have to use annotations to map a persistent class. Later we'll show other mapping options, such as the JPA `orm.xml` mapping file, and native `hbm.xml` mapping files, and when they're a better solution than source annotations, which are the most frequently used approach nowadays.

The `Message` class is now ready. We can store instances in our database and write queries to load them again into application memory.

2.3.3 Storing and loading messages

What you really came here to see is JPA with Hibernate, so let's save a new `Message` to the database.

Listing 2.4 The HelloWorldJPATest class

Path: Ch02/helloworld/src/test/java/com/manning/javapersistence/ch02/HelloWorldJPATest.java

```

public class HelloWorldJPATest {

    @Test
    public void storeLoadMessage() {

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("ch02.ex01");          #A

        try {
            EntityManager em = emf.createEntityManager();                  #B
            em.getTransaction().begin();                                    #C

            Message message = new Message();                             #D
            message.setText("Hello World!");                            #D

            em.persist(message);                                       #E

            em.getTransaction().commit();                                #F
            //INSERT into MESSAGE (ID, TEXT) values (1, 'Hello World!')

            em.getTransaction().begin();                                #G

            List<Message> messages =
                em.createQuery("select m from Message m").getResultList(); #H
            //SELECT * from MESSAGE

            messages.get(messages.size() - 1).setText("Hello World from JPA!"); #I
            //UPDATE MESSAGE set TEXT = 'Hello World from JPA!' where ID = 1

            assertEquals(1, messages.size());                           #K
            assertEquals("Hello World from JPA!",                      #K
                        messages.get(0).getText());                         #L
        });

        em.close();                                                 #M

    } finally {
        emf.close();                                              #N
    }
}

```

#A First, we need an EntityManagerFactory to talk to the database. This API represents the persistence unit; most applications have one EntityManagerFactory for one configured persistence unit. Once it starts, the application should create the EntityManagerFactory; the factory is thread-safe, and all code in the application that accesses the database should share it.

#B Begin a new session with the database by creating an EntityManager. This is the context for all persistence operations.

```

#C Get access to the standard transaction API and begin a transaction on this thread of execution.
#D Create a new instance of the mapped domain model class Message and set its text property.
#E Enlist the transient instance with the persistence context; we make it persistent. Hibernate now knows that we
    wish to store that data, but it doesn't necessarily call the database immediately.
#F Commit the transaction. Hibernate automatically checks the persistence context and executes the necessary SQL
    INSERT statement. To help you understand how Hibernate works, we show the automatically generated and
    executed SQL statements in source code comments when they occur. Hibernate inserts a row in the MESSAGE
    table, with an automatically generated value for the ID primary key column, and the TEXT value.
#G Every interaction with the database should occur within transaction boundaries, even if we're only reading data -
    so, we start a new transaction. Any potential failure appearing from now on will not impact the previously
    committed transaction.
#H Execute a query to retrieve all instances of Message from the database.
#I We can change the value of a property. Hibernate detects this automatically because the loaded Message is still
    attached to the persistence context it was loaded in.
#J On commit, Hibernate checks the persistence context for dirty state and executes the SQL UPDATE automatically
    to synchronize in-memory with the database state.
#K We check the size of the list of messages retrieved from the database.
#L We check the message we persisted is in the database. We use the JUnit 5 assertAll method, which always
    checks all the assertions that are passed to it, even if some of them fail. The two assertions that we verify are
    conceptually related.
#M As we created an EntityManager, we must close it.
#N As we created an EntityManagerFactory, we must close it.

```

The query language you've seen in this example isn't SQL, it's the Jakarta Persistence Query Language (JPQL). Although there is syntactically no difference in this trivial example, the `Message` in the query string doesn't refer to the database table name, but to the persistent class name. For this reason, the `Message` class name in the query is case-sensitive. If we map the class to a different table, the query will still work.

Also, notice how Hibernate detects the modification to the `text` property of the message and automatically updates the database. This is the automatic dirty-checking feature of JPA in action. It saves the effort of explicitly asking the persistence manager to update the database when we modify the state of an instance inside a transaction.

Figure 2.2 shows the result of checking the existence of the record we inserted and updated on the side of the database. We remind that the machine we are running the code on has MySQL Release 8.0 installed and we have created a database named CH02 by running the Ch02.sql script from the source code.

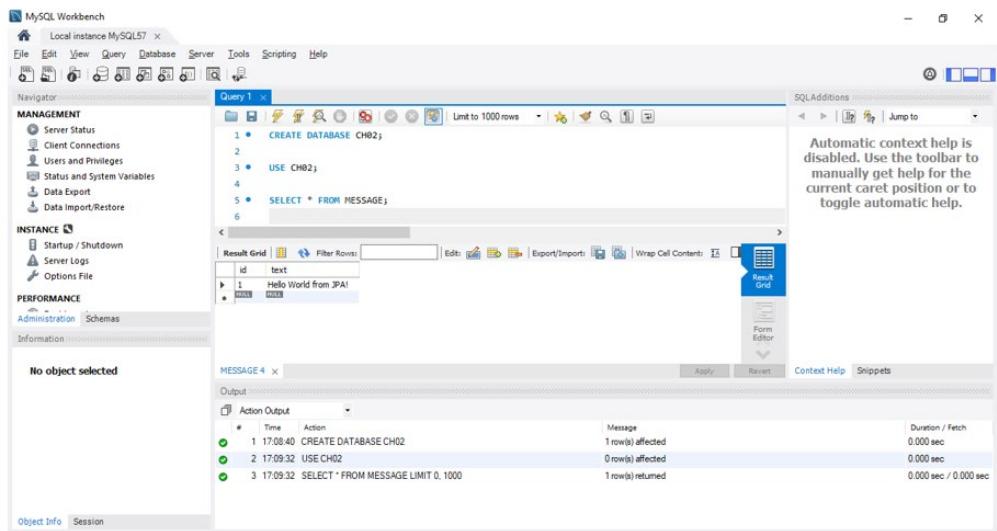


Figure 2.2 The result of checking the existence of the inserted and updated record on the database side

You've now completed your first JPA and Hibernate application. Let's now have a quick look at the native Hibernate bootstrap and configuration API.

2.4 Native Hibernate configuration

Although basic (and extensive) configuration is standardized in JPA, we can't access all the configuration features of Hibernate with properties in persistence.xml. Note that most applications, even quite sophisticated ones, don't need such special configuration options and hence don't have to access the bootstrap API we show in this section. If you aren't sure, you can skip this section and come back to it later, when you need to extend Hibernate type adapters, add custom SQL functions, and so on.

This paragraph analyzes the usage of native Hibernate. This means that we'll no longer use the JPA dependencies and classes, but directly the Hibernate dependencies and API. We remind that JPA is only a specification that may use different implementations (Hibernate for example, but also one of the alternatives, as MyBatis or EclipseLink) through the same API. Hibernate, as implementation, provides its own dependencies and classes. While JPA usage provides more flexibility, we'll see throughout the book that accessing Hibernate implementation directly allows for more features that are not covered by JPA standard (and we'll emphasize this at the right moments).

The native equivalent of the standard JPA EntityManagerFactory is the org.hibernate.SessionFactory. We have usually one per application, and it's the same pairing of class mappings with database connection configuration.

To configure the native Hibernate, we may use a hibernate.properties Java properties file, or a hibernate.cfg.xml XML file. We chose the second option, and the configuration contains

database and session-related options. This XML file is generally placed under the folder `src/main/resource` or `src/test/resource`. As we need the information for the Hibernate configuration in the tests, we chose the second location alternative.

Listing 2.5 The hibernate.cfg.xml configuration file

Path: Ch02/helloworld/src/test/resources/hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN" "http://www.hibernate.org/dtd/hibernate-
    configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">#A
            com.mysql.cj.jdbc.Driver #B
        </property> #C
        <property name="hibernate.connection.url">#C
            jdbc:mysql://localhost:3306/CH02 #D
        </property> #D
        <property name="hibernate.connection.username">root</property> #E
        <property name="hibernate.connection.password"></property> #F
        <property name="hibernate.connection.pool_size">50</property> #G
        <property name="show_sql">true</property> #H
        <property name="hibernate.hbm2ddl.auto">create</property> #I
    </session-factory>
</hibernate-configuration>
```

#A We use the tags to indicate the fact that we are configuring Hibernate.

#B More exactly, we are configuring the `SessionFactory` object. `SessionFactory` is an interface and we need one `SessionFactory` to interact with one database.

#C We indicate the JDBC properties - the driver.

#D The URL of the database.

#E The username.

#F No password to access it. The machine we are running the programs on has MySQL 8 installed and the access credentials are the ones from `hibernate.cfg.xml`. You should modify the credentials to correspond to the ones on your machine.

#G We limit the number of connections waiting in the Hibernate database connection pool to 50.

#H While executing, the SQL code is shown.

#I Every time the program is executed, the database will be created from scratch. This is ideal for automated testing, when we want to work with a clean database for every test run.

Let's save a `Message` to the database using native Hibernate.

Listing 2.6 The HelloWorldHibernateTest class

Path:

```

Ch02/helloworld/src/test/java/com/manning/javapersistence/ch02/HelloWorldHibernateTe
st.java

public class HelloWorldHibernateTest {

    private static SessionFactory createSessionFactory() {
        Configuration configuration = new Configuration();
        configuration.configure().addAnnotatedClass(Message.class);          #A
        ServiceRegistry serviceRegistry = new
            StandardServiceRegistryBuilder().                           #B
            applySettings(configuration.getProperties()).build();      #C
        return configuration.buildSessionFactory(serviceRegistry);       #D
    }

    @Test
    public void storeLoadMessage() {

        try (SessionFactory sessionFactory = createSessionFactory();           #E
             Session session = sessionFactory.openSession()) {               #F

            session.beginTransaction();                                       #G

            Message message = new Message();                                #H
            message.setText("Hello World from Hibernate!");                 #H

            session.persist(message);                                       #I

            session.getTransaction().commit();                                #J
            // INSERT into MESSAGE (ID, TEXT)
            // values (1, 'Hello World from Hibernate!')
            session.beginTransaction();                                     #K

            CriteriaQuery<Message> criteriaQuery =
                session.getCriteriaBuilder().createQuery(Message.class);   #L
                #L
            criteriaQuery.from(Message.class);                            #M

            List<Message> messages =                                         #N
                session.createQuery(criteriaQuery).getResultList();        #N
            // SELECT * from MESSAGE

            session.getTransaction().commit();                                #O

            assertEquals(1, messages.size());                                #P
            assertEquals("Hello World from Hibernate!",                      #P
                         messages.get(0).getText());                          #Q
        }
    }
}

```

#A To create a SessionFactory, first, we need to create a configuration.

#B We need to call the configure method on it and to add Message to it as annotated class. The execution of the configure method will load the content of the default hibernate.cfg.xml file.

#C The builder pattern helps us create the immutable service registry and configure it by applying settings with chained method calls. A `ServiceRegistry` hosts and manages services that need access to the `SessionFactory`. Services are classes that provide pluggable implementations of different types of functionality to Hibernate.

#D We build a `SessionFactory` using the configuration and the service registry we have previously created.

#E The `SessionFactory` created with the `createSessionFactory` method we have previously defined is passed as an argument to a `try` with resources, as `SessionFactory` implements the `AutoCloseable` interface.

#F Similarly, we begin a new session with the database by creating a `Session`, which also implements the `AutoCloseable` interface. This is our context for all persistence operations.

#G Get access to the standard transaction API and begin a transaction on this thread of execution.

#H Create a new instance of the mapped domain model class `Message` and set its `text` property.

#I Enlist the transient instance with the persistence context; we make it persistent. Hibernate now knows that we wish to store that data, but it doesn't necessarily call the database immediately. The native Hibernate API is pretty similar to the standard JPA and most methods have the same name.

#J Synchronize the session with the database and close the current session on commit of the transaction automatically.

#K Begin another transaction. Every interaction with the database should occur within transaction boundaries, even if we're only reading data.

#L Create an instance of `CriteriaQuery` by calling the `CriteriaBuilder createQuery()` method. A `CriteriaBuilder` is used to construct criteria queries, compound selections, expressions, predicates, orderings. A `CriteriaQuery` defines functionality that is specific to top-level queries. `CriteriaBuilder` and `CriteriaQuery` belong to the Criteria API, which allows us to build a query programmatically.

#M Create and add a query root corresponding to the given `Message` entity.

#N Call the `getResultSet()` method of the query object to get the results. The query that is created and executed will be `SELECT * FROM MESSAGE`.

#O Commit the transaction.

#P Check the size of the list of messages retrieved from the database.

#Q Check that the message we persisted is in the database. We use the JUnit 5 `assertAll` method, which always checks all the assertions that are passed to it, even if some of them fail. The two assertions that we verify are conceptually related.

Figure 2.3 shows the result of checking the existence of the record we inserted on the side of the database by using native Hibernate. We remind that the machine we are running the code on has MySQL Release 8.0 installed and we have already created a database named CH02 by running the Ch02.sql script from the source code.

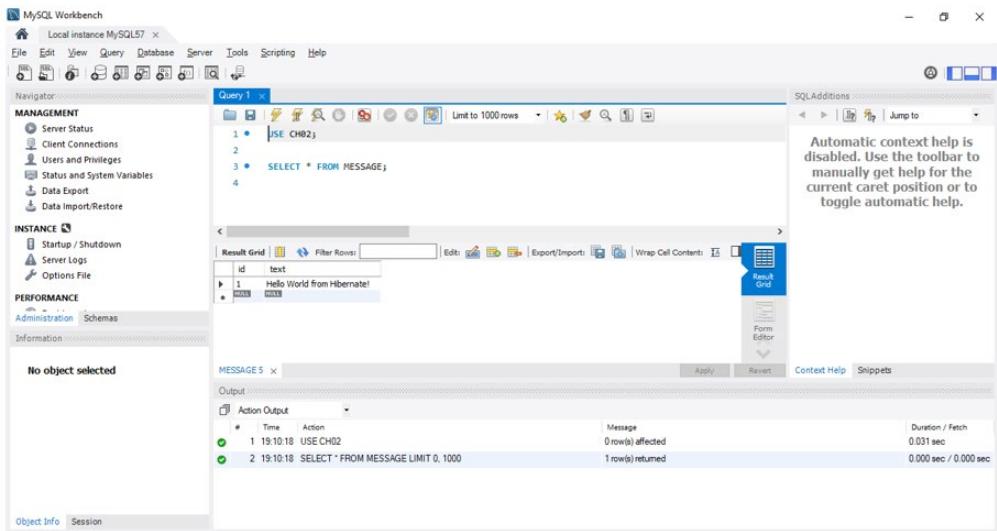


Figure 2.3 The result of checking the existence of the inserted record on the database side

Most of the examples in this book don't use the `SessionFactory` or `Session` API. From time to time, when a particular feature is only available in Hibernate, we show how to `unwrap()` the native interface.

2.5 Switching between JPA and Hibernate

It is possible that we are working with JPA and we need to access the Hibernate API. Or, vice-versa, we work with Hibernate native and we need to create an `EntityManagerFactory` from the Hibernate configuration.

To obtain a `SessionFactory` from an `EntityManagerFactory`, we'll have to `unwrap` the first one from the second one.

Listing 2.7 Obtaining a SessionFactory from an EntityManagerFactory

```
Path: Ch02/helloworld/src/test/java/com/manning/javapersistence/ch02/HelloWorldJPAToHibernateTest.java

private static SessionFactory getSessionFactory
    (EntityManagerFactory entityManagerFactory) {
    return entityManagerFactory.unwrap(SessionFactory.class);
}
```

Starting with JPA version 2.0, we may get access to the APIs of the underlying implementations. The `EntityManagerFactory` (and also the `EntityManager`) declare an `unwrap` method that will return objects belonging to the classes of the JPA implementation.

When using the Hibernate implementation, we may get the corresponding `SessionFactory` or `Session` objects and start using them just as demonstrated in listing 2.6. When a particular feature is only available in Hibernate, we may switch to it using the `unwrap` method.

We may be interested in the reverse operation: create an `EntityManagerFactory` from an initial Hibernate configuration.

Listing 2.8 Obtaining an EntityManagerFactory from a Hibernate configuration

Path:

Ch02/helloworld/src/test/java/com/manning/javapersistence/ch02/HelloWorldHibernateToJPATest.java

```
private static EntityManagerFactory createEntityManagerFactory() {
    Configuration configuration = new Configuration();                      #A
    configuration.configure().addAnnotatedClass(Message.class);               #B

    Map<String, String> properties = new HashMap<>();                      #C
    Enumeration<?> propertyNames =                                         #D
        configuration.getProperties().propertyNames();                         #D
    while (propertyNames.hasMoreElements()) {                                    #E
        String element = (String) propertyNames.nextElement();                 #E
        properties.put(element,                                         #E
            configuration.getProperties().getProperty(element));             #E
    }

    return Persistence.createEntityManagerFactory("ch02.ex01", properties); #F
}
```

#A Create a new Hibernate configuration.

#B Call the `configure` method, which adds the content of the default `hibernate.cfg.xml` file to the configuration, then we explicitly add `Message` as annotated class.

#C Create a new hash map to be filled in with the existing properties.

#D Get all property names from the Hibernate configuration.

#E Add the property names one by one to the previously created map.

#F We return a new `EntityManagerFactory`, providing to it the `ch02.ex01` persistence unit name and the previously created map of properties.

2.6 “Hello World” with Spring Data JPA

In this section, we'll write our first Spring Data JPA application, which stores a message in the database and then retrieves it.

We add the Spring dependencies on the side of the Apache Maven configuration.

Listing 2.9 The Maven dependencies on Spring

Path: Ch02/helloworld/pom.xml

```
<dependency> #A
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>2.3.3.RELEASE</version>
</dependency> #A
<dependency> #B
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.2.5.RELEASE</version>
</dependency> #B
```

#A The `spring-data-jpa` module provides the repository support for JPA and includes transitive dependencies on other modules we'll need, such as `spring-core` and `spring-context`.

#B We also need the `spring-test` dependency, to run the tests with the help of the Spring extension.

The standard configuration file for Spring Data JPA is a Java class to create and setup the beans needed by Spring Data. The configuration can be done using either an XML file or Java code and we chose this second alternative. Create the following configuration file for the "Hello World" application:

Listing 2.10 The SpringDataConfiguration class

Path:

Ch02/helloworld/src/test/java/com/manning/javapersistence/ch02/configuration/SpringDataConfiguration.java

```
@EnableJpaRepositories("com.manning.javapersistence.ch02.repositories")      #A
public class SpringDataConfiguration {
    @Bean
    public DataSource dataSource() {                                              #B
        DriverManagerDataSource dataSource = new DriverManagerDataSource(); #B
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");       #C
        dataSource.setUrl("jdbc:mysql://localhost:3306/CH02");           #D
        dataSource.setUsername("root");                                     #E
        dataSource.setPassword("");                                       #F
        return dataSource;                                               #B
    }

    @Bean
    public JpaTransactionManager transactionManager(EntityManagerFactory emf) { #G
        return new JpaTransactionManager(emf);                                #G
    }

    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {                               #H
        HibernateJpaVendorAdapter jpaVendorAdapter = new
            HibernateJpaVendorAdapter();                                     #H
        jpaVendorAdapter.setDatabase(Database.MYSQL);                      #I
        jpaVendorAdapter.setShowSql(true);                                    #J
        return jpaVendorAdapter;                                            #H
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() { #K
        LocalContainerEntityManagerFactoryBean
            localContainerEntityManagerFactoryBean =
                new LocalContainerEntityManagerFactoryBean();                 #K
        localContainerEntityManagerFactoryBean.setDataSource(dataSource()); #L
        Properties properties = new Properties();                           #M
        properties.put("hibernate.hbm2ddl.auto", "create");               #M
        localContainerEntityManagerFactoryBean.
            setJpaProperties(properties);                                     #M
        localContainerEntityManagerFactoryBean.
            setJpaVendorAdapter(jpaVendorAdapter());                         #N
        localContainerEntityManagerFactoryBean.
            setPackagesToScan("com.manning.javapersistence.ch02");        #O
        return localContainerEntityManagerFactoryBean;                     #K
    }
}
```

#A The `@EnableJpaRepositories` annotation will scan the package of the annotated configuration class for Spring Data repositories.

#B We create a data source bean.

#C We indicate the JDBC properties - the driver.

#D The URL of the database.

#E The username.

```

#F No password for access. The machine we are running the programs on has MySQL 8 installed and the access
credentials are the ones from this configuration. You should modify the credentials to correspond to the ones on
your machine.
#g We create a transaction manager bean based on an entity manager factory. Every interaction with the database
should occur within transaction boundaries and Spring Data needs a transaction manager bean.
#H We create a JPA vendor adapter bean that is needed by JPA to interact with Hibernate.
#I We configure this vendor adapter to access a MySQL database.
#J Show the SQL code while it is executed.
#K We create a LocalContainerEntityManagerFactoryBean – this is a factory bean that produces an
EntityManagerFactory following the JPA standard container bootstrap contract.
#L Set the data source.
#M Set the database to be created from scratch every time the program is executed.
#N Set the vendor adapter.
#O Set the packages to scan for entity classes. As the Message entity is located in
com.manning.javapersistence.ch02, we set this package to be scanned.

```

Spring Data JPA provides support for JPA-based data access layers by reducing the boilerplate code and creating implementations for the repository interfaces. We need only to define our own repository interface, to extend one of the Spring Data interfaces.

Listing 2.11 The MessageRepository interface

Path:

```

Ch02/helloworld/src/main/java/com/manning/javapersistence/ch02/repositories/MessageR
epository.java

public interface MessageRepository extends CrudRepository<Message, Long> {
}

```

The `MessageRepository` interface extends `CrudRepository<Message, Long>`. This means that it is a repository of `Message` entities, having a `Long` identifier. Remember, the `Message` class has an `id` field annotated as `@Id` of type `Long`. We can directly call methods as `save`, `findAll` or `findById`, inherited from `CrudRepository` and we can use them without any other additional information, to execute the usual operations against a database. Spring Data JPA will create a proxy class implementing the `MessageRepository` interface and implement its methods (figure 2.4).

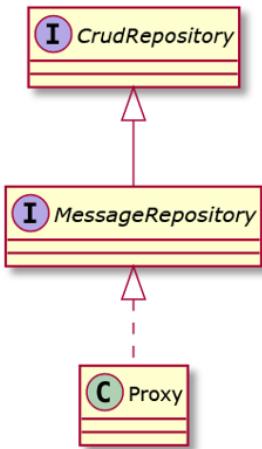


Figure 2.4 The Spring Data JPA Proxy class implements the `MessageRepository` interface

Let's save a `Message` to the database using Spring Data JPA.

Listing 2.12 The `HelloWorldSpringDataJPATest` class

```

Path: Ch02/helloworld/src/test/java/com/manning/javapersistence/ch02/HelloWorldSpringDataJPATest.java

@ExtendWith(SpringExtension.class)                                     #A
@ContextConfiguration(classes = {SpringDataConfiguration.class})      #B
public class HelloWorldSpringDataJPATest {

    @Autowired
    private MessageRepository messageRepository;                         #C
    #C

    @Test
    public void storeLoadMessage() {
        Message message = new Message();                                 #D
        message.setText("Hello World from Spring Data JPA!");          #D

        messageRepository.save(message);                                  #E

        List<Message> messages = (List<Message>)messageRepository.findAll();#F

        assertEquals(1, messages.size());                                #G
        assertEquals("Hello World from Spring Data JPA!",               #G
                    messages.get(0).getText());                            #H
    }
}
  
```

#A We extend the test using `SpringExtension`. This extension is used to integrate the Spring test context with the JUnit 5 Jupiter test.

#B The Spring test context is configured using the beans defined in the previously presented `SpringDataConfiguration` class.

#C A `MessageRepository` bean is injected by Spring through auto-wiring. This is possible as the `com.manning.javapersistence.ch02.repositories` package where `MessageRepository` is located was used as the argument of the `@EnableJpaRepositories` annotation in listing 2.8. If we call `messageRepository.getClass()`, we'll see that it is something like `com.sun.proxy.$Proxy41` – a proxy generated by Spring Data, as explained in figure 2.4.

#D Create a new instance of the mapped domain model class `Message` and set its `text` property.

#E Persist the `message` object. The `save` method is inherited from the `CrudRepository` interface and its body will be generated by Spring Data JPA when the proxy class is created. It will simply save a `Message` entity to the database.

#F Retrieve the messages from repository. The `findAll` method is inherited from the `CrudRepository` interface and its body will be generated by Spring Data JPA when the proxy class is created. It will simply return all entities belonging to the `Message` class.

#G Check the size of the list of messages retrieved from the database and that the message we persisted is in the database #H. We use the JUnit 5 `assertAll` method, which always checks all the assertions that are passed to it, even if some of them fail. The two assertions that we verify are conceptually related.

You notice that the Spring Data JPA test is considerably shorter than the ones using JPA or Hibernate native. This is because the boilerplate code has been removed – no more explicit object creation or explicit control of the transactions. The repository object is injected and it provides the generated methods of the proxy class. The burden is heavier now on the configuration side – but this should be done only once per application.

Figure 2.5 shows the result of checking the existence of the record we inserted on the side of the database by using Spring Data JPA. We remind that the machine we are running the code on has MySQL Release 8.0 installed and we have already created a database named CH02 by running the Ch02.sql script from the source code.

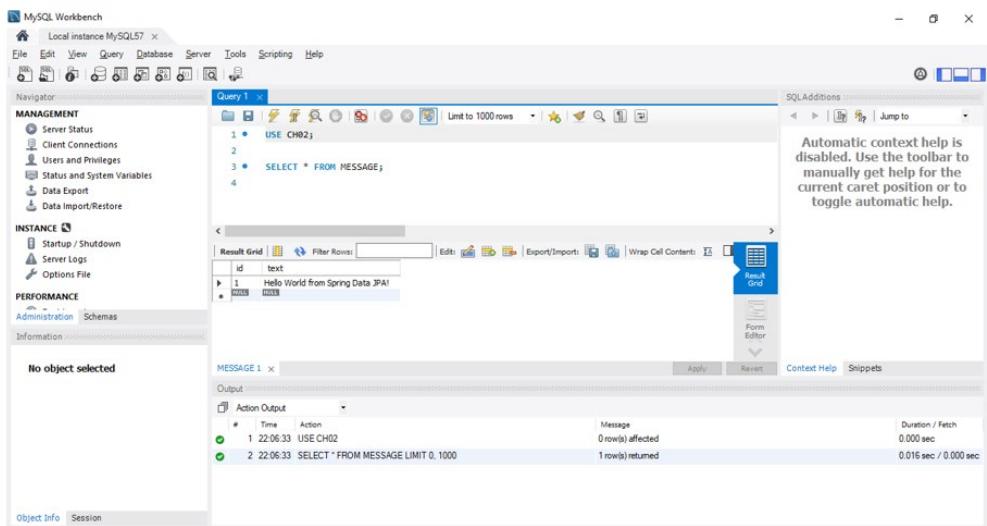


Figure 2.5 The result of checking the existence of the inserted record on the database side

2.7 Comparing the approaches of persisting entities

We have followed the implementation of a simple application interacting with a database and that uses, alternatively, JPA, Hibernate native, and Spring Data JPA. Our purpose was to analyze each approach and how the needed configuration and the code to be written varies. Table 2.1 summarizes the characteristics of these approaches.

Table 2.1 Comparison of working with JPA, Hibernate native, and Spring Data JPA

Framework	Characteristics
JPA	<p>Uses the general JPA API and requires a persistence provider.</p> <p>We may switch between persistence providers from the configuration.</p> <p>Requires explicit management of the EntityManagerFactory, EntityManager, and transactions.</p> <p>The way the configuration is done and the amount of code to be written is similar to the Hibernate native approach.</p> <p>We may switch to the JPA approach by constructing an EntityManagerFactory from a Hibernate native configuration.</p>
Native Hibernate	<p>Uses the specific Hibernate native API. You are locked to using this chosen framework.</p> <p>Builds its configuration starting from the default Hibernate configuration files (hibernate.cfg.xml or hibernate.properties).</p> <p>Requires explicit management of the SessionFactory, Session, and transactions.</p> <p>The way the configuration is done and the amount of code to be written are similar to the JPA approach.</p> <p>We may switch to the Hibernate native approach by unwrapping a SessionFactory from an EntityManagerFactory or a Session from an EntityManager.</p>
Spring Data JPA	<p>Needs the additional Spring Data dependencies in the project.</p> <p>The configuration part will also take care of the creation of the beans needed for the project, including the transaction manager.</p> <p>The repository interface needs only to be declared, and Spring Data will create an implementation for it as a proxy class with generated methods that interact with the database.</p> <p>The needed repository is injected and not explicitly created by the programmer.</p> <p>The amount of code to be written is the shortest of all these approaches, as the configuration part has taken most of the burden.</p>

Considering the performances of each approach, we reference the article *Object-Relational Mapping Using JPA, Hibernate and Spring Data JPA* by Cătălin Tudose and Carmen Odubășteanu (Tudose, 2021).

To analyze the running times, a batch of insert, update, select and delete operations was executed using the three approaches, progressively increasing the number of records from 1000 to 50000. Tests were made on Windows 10 Enterprise, running on a 4 cores Intel i7-5500U processor at 2.40GHz and 8 GB RAM. The results may be examined in tables 2.2-2.5 and figures 2.6-2.9.

Table 2.2 Insert execution times by framework (times in ms)

Number of records	Hibernate	JPA	Spring Data JPA
1000	1138	1127	2288
5000	3187	3307	8410
10000	5145	5341	14565
20000	8591	8488	26313
30000	11146	11859	37579
40000	13011	13300	48913
50000	16512	16463	59629

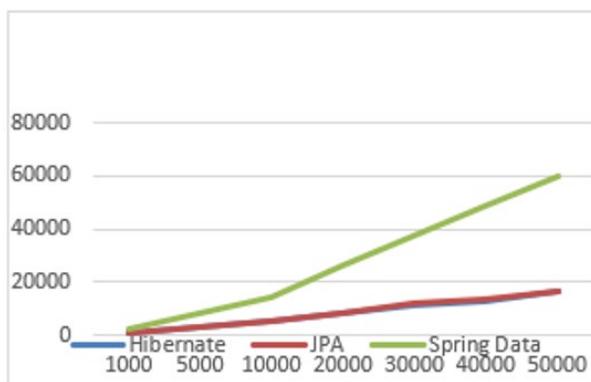
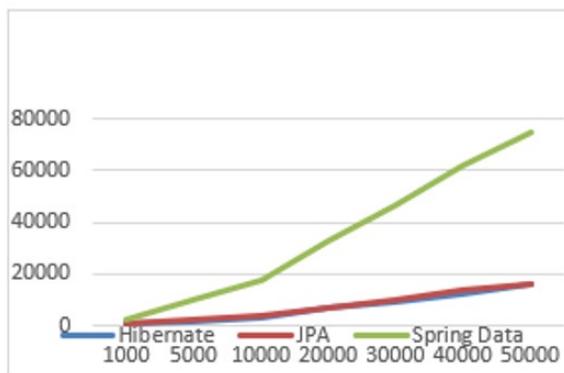
**Figure 2.6 Insert execution times by framework (times in ms)**

Table 2.3 Update execution times by framework (times in ms)

Number of records	Hibernate	JPA	Spring Data JPA
1000	706	759	2683
5000	2081	2256	10211
10000	3596	3958	17594
20000	6669	6776	33090
30000	9352	9696	46341
40000	12720	13614	61599
50000	16276	16355	75071

**Figure 2.7 Update execution times by framework (times in ms)****Table 2.4 Select execution times by framework (times in ms)**

Number of records	Hibernate	JPA	Spring Data JPA
1000	1138	1127	2288
5000	3187	3307	8410
10000	5145	5341	14565
20000	8591	8488	26313
30000	11146	11859	37579
40000	13011	13300	48913
50000	16512	16463	59629

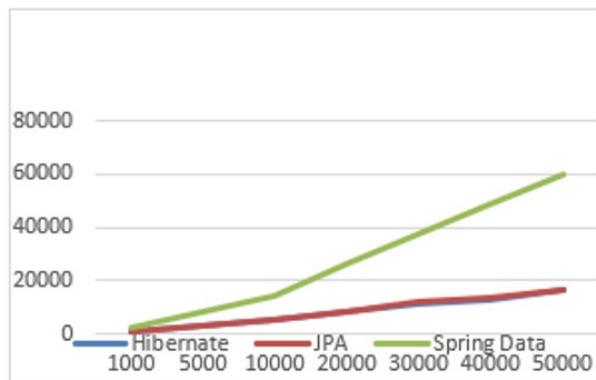


Figure 2.8 Select execution times by framework (times in ms)

Table 2.5 Delete execution times by framework (times in ms)

Number of records	Hibernate	JPA	Spring Data JPA
1000	584	551	2430
5000	1537	1628	9685
10000	2992	2763	17930
20000	5344	5129	32906
30000	7478	7852	47400
40000	10061	10493	62422
50000	12857	12768	79799

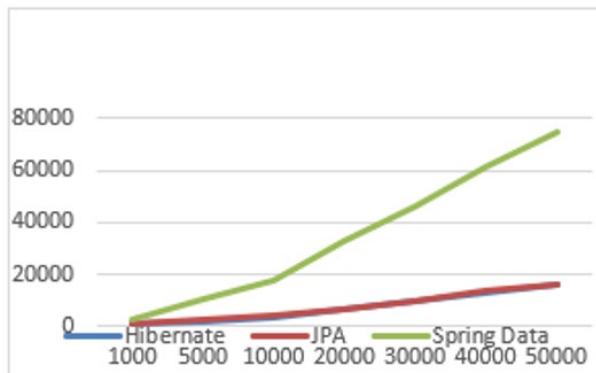


Figure 2.9 Delete execution times by framework (times in ms)

The three approaches provide different performances. Hibernate and JPA go head to head, the graphics of their times almost overlap for all four operations (insert, update, select and delete). Even if JPA comes with its own API on top of Hibernate, this additional layer introduces no overhead.

The execution times of Spring Data JPA insertions start from about 2 times more than Hibernate and JPA for 1000 records to about 3.5 times more for 50000 records. The overhead of the Spring Data JPA framework is considerable.

For Hibernate and JPA, the update and delete execution times decrease in comparison with the insert execution times. On the contrary, the Spring Data JPA update and delete execution times increase in comparison with the insert execution times.

For Hibernate and JPA, the select times grow very slowly with the number of rows. The Spring Data JPA select execution times strongly grow with the number of rows.

Using Spring Data JPA is mainly justified in particular situations: the project already uses the Spring framework and needs to rely on its existing paradigm (e.g. inversion of control, automatically managed transactions); there is a strong need to decrease the amount of code and thus shorten the development time – nowadays, it is cheaper to acquire more computing power than to acquire more developers.

This chapter has focused on alternatives for working with a database from a Java application: JPA, Hibernate native, and Spring Data JPA and provided introductory examples for each of them.

Chapter 3 will start a more complex example and will deal in more depth with domain models and metadata.

2.8 Summary

- We implemented our first Java persistence project using three alternatives: JPA, Hibernate native, and Spring Data JPA.
- We created a persistent class and its mapping with annotations.
- Using JPA, we implemented the configuration and bootstrap of a persistence unit, and how to create the `EntityManagerFactory` entry point. Then we called the `EntityManager` to interact with the database, storing and loading an instance of the persistent domain model class.
- We demonstrated some of the native Hibernate bootstrap and configuration options, as well as the equivalent basic Hibernate APIs, `SessionFactory`, and `Session`.
- We examined how we can switch between the JPA approach and the Hibernate approach.
- We implemented the configuration of a Spring Data JPA application, created the repository interface, then used it to store and load an instance of the persistent domain model class.
- We finally compared and contrasted these three approaches: JPA, Hibernate native, and Spring Data JPA.

3

Domain models and metadata

This chapter covers

- Discovering the `CaveatEmptor` example application
- Implementing the domain model
- Examining object/relational mapping metadata options

The “Hello World” example in the previous chapter introduced you to Hibernate and Spring Data; certainly, it isn’t useful for understanding the requirements of real-world applications with complex data models. For the rest of the book, we use a much more sophisticated example application—`CaveatEmptor`, an online auction system—to demonstrate Jakarta Persistence, Hibernate, and later Spring Data. (*Caveat emptor* means “Let the buyer beware”.)

Major new features in JPA 2

A JPA persistence provider now integrates automatically with a Bean Validation provider. When data is stored, the provider automatically validates constraints on persistent classes.

The `Metamodel` API has been added. You can obtain the names, properties, and mapping metadata of the classes in a persistence unit.

We’ll start our discussion of the application by introducing a layered application architecture. Then, you’ll learn how to identify the business entities of a problem domain. You’ll create a conceptual model of these entities and their attributes, called a *domain model*, and you’ll implement it in Java by creating persistent classes. We’ll spend some time exploring exactly what these Java classes should look like and where they fit within a typical layered application architecture. We’ll also look at the persistence capabilities of the classes and how

this aspect influences the design and implementation. We'll add *Bean Validation*, which helps to automatically verify the integrity of the domain model data not only for persistent information but also for the business logic.

We'll then explore mapping metadata options—the ways you tell Hibernate how the persistent classes and their properties relate to database tables and columns. This can be as simple as adding annotations directly in the Java source code of the classes or writing XML documents that you eventually deploy along with the compiled Java classes that Hibernate accesses at runtime. After reading this chapter, you'll know how to design the persistent parts of your domain model in complex real-world projects and what mapping metadata option you'll primarily prefer and use. Let's start with the example application.

3.1 The example CaveatEmptor application

The CaveatEmptor example is an online auction application that demonstrates ORM techniques, Hibernate, and Spring Data functionality. We won't pay much attention to the user interface in this book (it could be web-based or a rich client); we'll concentrate instead on the data access code.

To understand the design issues involved in ORM, let's pretend the Caveat-Emptor application doesn't yet exist and that we're building it from scratch. Let's start by looking at the architecture.

3.1.1 A layered architecture

With any nontrivial application, it usually makes sense to organize classes by concern. Persistence is one concern; others include presentation, workflow, and business logic. A typical object-oriented architecture includes layers of code that represent the concerns.

Cross-cutting concerns

There are also so-called cross-cutting concerns, which may be implemented generically—by framework code, for example. Typical cross-cutting concerns include logging, authorization, and transaction demarcation.

A layered architecture defines interfaces between code that implements the various concerns, allowing changes to be made to the way one concern is implemented without significant disruption to code in the other layers. Layering determines the kinds of inter-layer dependencies that occur. The rules are as follows:

- Layers communicate from top to bottom. A layer is dependent only on the interface of the layer directly below it.
- Each layer is unaware of any other layers except for the layer just below it and eventually for the layer above if it receives explicit requests from it.

Different systems group concerns differently, so they define different layers. The typical, proven, high-level application architecture uses three layers: one each for presentation, business logic, and persistence, as shown in figure 3.1.

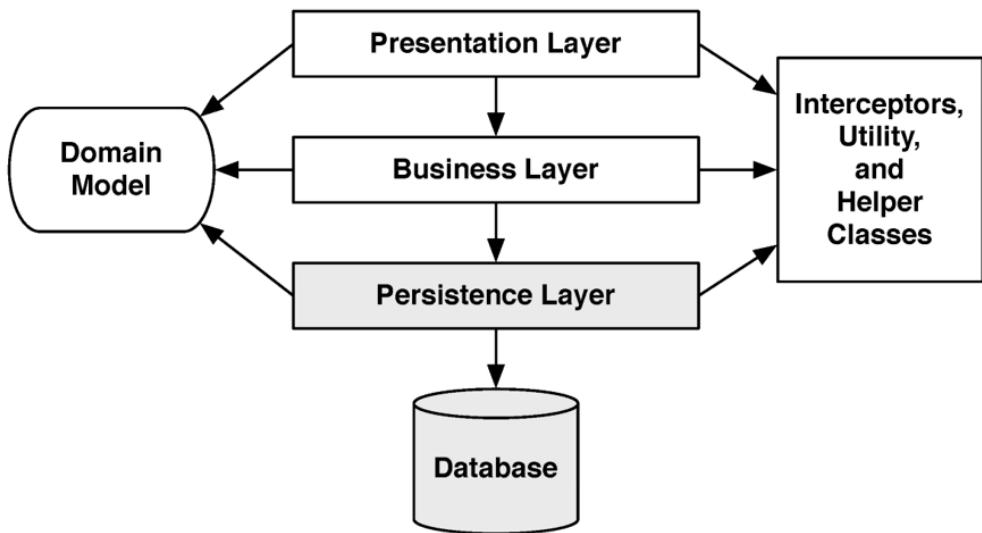


Figure 3.1 A persistence layer is the basis of a layered architecture.

- *Presentation layer*—The user interface logic is topmost. Code responsible for the presentation and control of page and screen navigation is in the presentation layer. The user interface code may directly access business entities of the shared *domain model* and render them on the screen, along with controls to execute actions. In some architectures, business entity instances might not be directly accessible by user interface code: for example, if the presentation layer isn't running on the same machine as the rest of the system. In such cases, the presentation layer may require its own special data-transfer model, representing only a transmittable subset of the domain model. A good example is working with the browser to interact with the application.
- *Business layer*— The business layer is generally responsible for implementing any business rules or system requirements that are part of the problem domain. This layer usually includes some kind of controlling component—code that knows when to invoke which business rule. In some systems, this layer has its own internal representation of the business domain entities. Alternatively, it relies on a domain model implementation, shared with the other layers of the application. A good example is a code responsible for calculus and business logic.
- *Persistence layer*—The persistence layer is a group of classes and components responsible for storing data to, and retrieving it from, one or more data stores. This layer needs a model of the business domain entities for which you'd like to keep a persistent state. The persistence layer is where the bulk of JPA, Hibernate, and Spring Data use takes place.

- *Database*—The database is usually external. It's the actual, persistent representation of the system state. If an SQL database is used, the database includes a schema and possibly stored procedures for the execution of business logic close to the data. The database is the place where data is persisted for the long term.
- *Helper and utility classes*—Every application has a set of infrastructural helper or utility classes that are used in every layer of the application. We may include here general-purpose classes or cross-cutting concern classes (for logging, security, caching). These shared infrastructural elements don't form a layer because they don't obey the rules for inter-layer dependency in a layered architecture.

Now that you have a high-level architecture, you can focus on the business problem.

3.1.2 Analyzing the business domain

At this stage, you, with the help of domain experts, analyze the business problems your software system needs to solve, identifying the relevant main entities and their interactions. The motivating goal behind the analysis and design of a domain model is to capture the essence of the business information for the application's purpose.

Entities are usually notions understood by users of the system: payment, customer, order, item, bid, and so forth. Some entities may be abstractions of less concrete things the user thinks about, such as a pricing algorithm, but even these are usually understandable to the user. You can find all these entities in the conceptual view of the business, sometimes called an *information model*.

From this business model, engineers and architects of object-oriented software create an object-oriented model, still at the conceptual level (no Java code). This model may be as simple as a mental image existing only in the mind of the developer, or it may be as elaborated as a UML class diagram. Figure 3.2 shows a simple model expressed in UML.



Figure 3.2 A class diagram of a typical online auction model

This model contains entities that you're bound to find in any typical e-commerce system: category, item, and user, represented in the class diagram above. This model of the problem domain represents all the entities and their relationships (and perhaps their attributes). We call this kind of object-oriented model of entities from the problem domain, encompassing only those entities that are of interest to the user, a *domain model*. It's an abstract view of the real world.

Instead of an object-oriented model, engineers and architects may start the application design with a data model. This is possibly expressed with an entity-relationship diagram and will contain the `CATEGORY`, `ITEM`, and `USER` entities, together with the relationships between them. We usually say that, concerning persistence, there is little difference between the two;

they're merely different starting points. In the end, what modeling language you use is secondary; we're most interested in the structure and relationships of the business entities. We care about the rules that have to be applied to guarantee the integrity of data (for example, the multiplicity of relationships included in the model) and the code procedures used to manipulate the data (usually not included in the model).

In the next section, we complete our analysis of the CaveatEmptor problem domain. The resulting domain model will be the central theme of this book.

3.1.3 The CaveatEmptor domain model

The CaveatEmptor site auctions many different kinds of items, from electronic equipment to airline tickets. Auctions proceed according to the English auction strategy: users continue to place bids on an item until the bid period for that item expires, and the highest bidder wins.

In any store, goods are categorized by type and grouped with similar goods into sections and onto shelves. The auction catalog requires some kind of hierarchy of item categories so that a buyer can browse these categories or arbitrarily search by category and item attributes. Lists of items appear in the category browser and search result screens. Selecting an item from a list takes the buyer to an item-detail view where an item may have images attached to it.

An auction consists of a sequence of bids, and one is the winning bid. User details include name, address, and billing information.

The result of this analysis, the high-level overview of the domain model, is shown in figure 3.3. Let's briefly discuss some interesting features of this model.

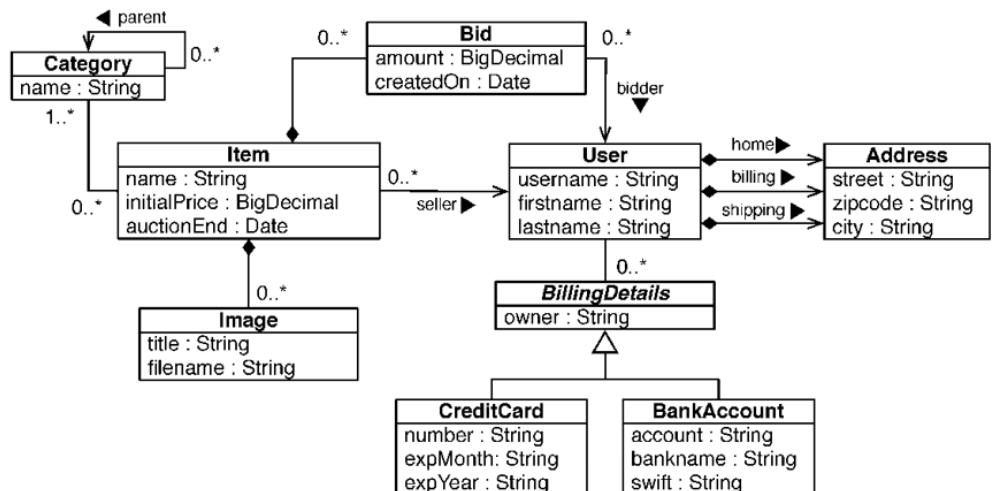


Figure 3.3 Persistent classes of the CaveatEmptor domain model and their relationships – detailed below

Each item can be auctioned only once, so you don't need to make `Item` distinct from any auction entities. Instead, you have a single auction item entity named `Item`. Thus, `Bid` is associated directly with `Item`. You model the `Address` information of a `User` as a separate class. A `User` may have three addresses for home, billing, and shipping. You do allow the user to have many `BillingDetails`. Subclasses of an abstract class represent the various billing strategies (allowing future extension).

The application may nest a `Category` inside another `Category`, and so on. A recursive association, from the `Category` entity to itself, expresses this relationship. Note that a single `Category` may have multiple child categories but at most one parent. Each `Item` belongs to at least one `Category`.

This representation isn't the *complete* domain model but only classes for which you need persistence capabilities. You'd like to store and load instances of `Category`, `Item`, `User`, and so on. We have simplified this high-level overview a little; we may introduce additional classes later or make modifications to them when needed for more complex examples.

Certainly, the entities in a domain model should encapsulate state and behavior. For example, the `User` entity should define the name and address of a customer *and* the logic required to calculate the shipping costs for items (to this particular customer).

There might be other classes in the domain model that have only transient runtime instances. Consider a `WinningBidStrategy` class encapsulating the fact that the highest bidder wins an auction. This might be called by the business layer (controller) code when checking the state of an auction. At some point, you might have to figure out how tax for sold items is calculated or how the system may approve a new user account. We don't consider such business rules or domain model behavior to be unimportant; rather, this concern is mostly orthogonal to the problem of persistence.

Now that you have a (rudimentary) application design with a domain model, the next step is to implement it in Java.

ORM without a domain model

Object persistence with full ORM is most suitable for applications based on a rich domain model. If your application doesn't implement complex business rules or complex interactions between entities (or if you have few entities), you may not need a domain model. Many simple and some not-so-simple problems are perfectly suited to table-oriented solutions, where the application is designed around the database data model instead of around an object-oriented domain model, often with logic executed in the database (stored procedures). Another aspect to consider is the learning curve: once you're proficient with Hibernate and Spring Data, you'll use them for all applications, even as a simple SQL query generator and result mapper. If you're just learning ORM, a trivial use case may not justify your invested time and overhead.

3.2 Implementing the domain model

You'll start with an issue that any implementation must deal with: the separation of concerns – which layer is concerned with what responsibility. The domain model implementation is usually a central, organizing component; it's reused heavily whenever you implement new

application functionality. For this reason, you should be prepared to go to some lengths to ensure that concerns other than business aspects don't leak into the domain model implementation.

3.2.1 Addressing leakage of concerns

When concerns such as persistence, transaction management, or authorization start to appear in the domain model classes, this is an example of leakage of concerns. The domain model implementation is such an important piece of code that it shouldn't depend on orthogonal APIs. For example, code in the domain model shouldn't call the database, not directly and not through an intermediate abstraction. This allows you to reuse the domain model classes virtually anywhere:

- The presentation layer can access instances and attributes of domain model entities when rendering views. The user may use the front-end (e.g., a browser) to interact with the application. This concern should be separate from the concerns of the other layers.
- The controller components in the business layer can also access the state of domain model entities and call methods of the entities. Here it is where calculus and business logic are executed. This concern should be separate from the concerns of the other layers.
- The persistence layer can load and store instances of domain model entities from and to the database, preserving their state. Here it is where the information is persisted for a long time. Similarly, this concern should be separate from the concerns of the other layers.

Most important, preventing leakage of concerns makes it easy to unit-test the domain model without the need for a particular runtime environment or container or the need for mocking any service dependencies. You can write unit tests that verify the correct behavior of your domain model classes without any special test harness. (We aren't talking about performance and integration tests as "load from the database" and "store in the database", but "calculate the shipping cost and tax" behavior.)

The Jakarta EE standard solves the problem of leaky concerns with metadata as annotations within your code or externalized as XML descriptors. This approach allows the runtime container to implement some predefined cross-cutting concerns—security, concurrency, persistence, transactions, and remoteness—in a generic way by intercepting calls to application components.

Hibernate isn't a Jakarta EE runtime environment, and it's not an application server. It's an implementation of the ORM technique.

JPA defines the *entity class* as the primary programming artifact. This programming model enables transparent persistence, and a JPA provider such as Hibernate also offers automated persistence.

3.2.2 Transparent and automated persistence

We use *transparent* to mean a complete separation of concerns between the persistent classes of the domain model and the persistence layer. The persistent classes are unaware

of—and have no dependency on—the persistence mechanism. From inside the persistent classes, there is no reference to the outside persistence mechanism. We use *automatic* to refer to a persistence solution (your annotated domain, the layer, and mechanism) that relieves you of handling low-level mechanical details, such as writing most SQL statements and working with the JDBC API. As a real-world use case, let's analyze how transparent and automated persistence is reflected at the level of the `Item` class.

The `Item` class of the `CaveatEmptor` domain model shouldn't have any runtime dependency on any Jakarta Persistence or Hibernate API. Furthermore:

- JPA doesn't require that any special superclasses or interfaces be inherited or implemented by persistent classes. Nor are any special classes used to implement attributes and associations.
- You can reuse persistent classes outside the context of persistence, in unit tests, or in the presentation layer, for example. You can create instances in any runtime environment with the regular Java `new` operator, preserving testability and reusability.
- In a system with transparent persistence, instances of entities aren't aware of the underlying data store; they need not even be aware that they're being persisted or retrieved. JPA externalizes persistence concerns to a generic persistence manager API.
- Hence, most of your code, and certainly your complex business logic, doesn't have to concern itself with the current state of a domain model entity instance in a single thread of execution.

We regard transparency as a requirement because it makes an application easier to build and maintain. Transparent persistence should be one of the primary goals of any ORM solution. Clearly, no automated persistence solution is completely transparent: every automated persistence layer, including JPA and Hibernate, imposes some requirements on the persistent classes. For example, JPA requires that collection--valued attributes be typed to an interface such as `java.util.Set` or `java.util.List` and not to an actual implementation such as `java.util.HashSet` (this is a good practice anyway). Or, a JPA entity class has to have a special attribute, called the *database identifier* (which is also less of a restriction but usually convenient).

You now know why the persistence mechanism should have minimal impact on how you implement a domain model and that transparent and automated persistence is required. Our preferred programming model to achieve this is POJO.

POJO

POJO is the acronym for Plain Old Java Objects. Martin Fowler, Rebecca Parsons, and Josh Mackenzie coined this term in 2000.

At the beginning of the 2000s, many developers started talking about POJO, a back-to-basics approach that essentially revives JavaBeans, a component model for UI development,

and reapplies it to the other layers of a system. Several revisions of the EJB and JPA specifications brought us new lightweight entities, and it would be appropriate to call them *persistence-capable JavaBeans*. Java engineers often use all these terms as synonyms for the same basic design approach.

You shouldn't be too concerned about what terms we use in this book; the ultimate goal is to apply the persistence aspect as transparently as possible to Java classes. Almost any Java class can be persistence-capable if you follow some simple practices. Let's see how this looks in code.

To be able to execute the examples from the source code, you need first to run the Ch03.sql script. The examples are using a MySQL server with the default credentials: username root and no password.

3.2.3 Writing persistence-capable classes

Working with fine-grained and rich domain models is a major Hibernate objective. This is a reason we work with POJOs. In general, using fine-grained objects means more classes than tables.

A persistence-capable plain-old Java class declares attributes, which represent state, and business methods, which define behavior. Some attributes represent associations to other persistence-capable classes.

A POJO implementation of the `User` entity of the domain model is shown in the following listing (example 1 from the `domainmodel` folder source code). Let's walk through the code.

Listing 3.1 POJO implementation of the User class

```
Path: Ch03/domainmodel/src/main/java/com/manning/javapersistence/ch03/ex01/User.java
public class User {

    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

The class can be abstract and, if needed, extend a non-persistent class or implement an interface. It must be a top-level class, not nested within another class. The persistence-capable class and any of its methods *shouldn't* be final (a requirement of the JPA specification). Hibernate is not so strict, and it will allow declaring final classes as entities or entities with final methods that access persistent fields. However, this is not a good practice, as this will prevent Hibernate from using the proxy pattern for performance improvement. In general, you should follow the JPA requirements if you would like your application to remain portable between different JPA providers.

Unlike the JavaBeans specification, which requires no specific constructor, **Hibernate (and JPA) require a constructor with no arguments for every persistent class**. Alternatively, you might not write a constructor at all; Hibernate will then use the Java default constructor. Hibernate calls classes using the Java reflection API on such a no-argument constructor to create instances. The constructor may not be public, but it has to be at least package-visible if Hibernate will use runtime-generated proxies for performance optimization.

The properties of the POJO implement the attributes of the business entities—for example, the `username` of `User`. You usually implement properties as private or protected member fields, together with public or protected property accessor methods: for each field, a method for retrieving its value and a method for setting the value. These methods are known as the *getter* and *setter*, respectively. The example POJO in listing 3.1 declares getter and setter methods for the `username` property.

The JavaBean specification defines the guidelines for naming accessor methods; this allows generic tools like Hibernate to easily discover and manipulate property values. A getter method name begins with `get`, followed by the name of the property (the first letter in uppercase); a setter method name begins with `set` and similarly is followed by the name of the property. You may begin getter methods for Boolean properties with `is` instead of `get`.

Hibernate doesn't require accessor methods. You can choose how the state of an instance of your persistent classes should be persisted. Hibernate will either directly access fields or call accessor methods. Your class design isn't disturbed much by these considerations. You can make some accessor methods non-public or completely remove them—then configure Hibernate to rely on field access for these properties.

Should property fields and accessor methods be private, protected, or package visible?

Typically, you do not allow direct access to the internal state of your class, so you don't make attribute fields public. If you make fields or methods private, you're effectively declaring that nobody should ever access them; only you're allowed to do that (or a service like Hibernate). This is a definitive statement. There are often good reasons for someone to access your “private” internals—usually to fix one of your bugs—and you only make people angry if they have to fall back to reflection access in an emergency. Instead, you might assume or know that the engineer who comes after you has access to your code and knows what he's doing.

Although trivial accessor methods are common, one of the reasons we like to use JavaBeans-style accessor methods is that they provide encapsulation: you can change the hidden internal implementation of an attribute without any changes to the public interface. If you configure Hibernate to access attributes through methods, you abstract the internal data structure of the class—the instance variables—from the design of the database.

For example, if your database stores the name of a user as a single `NAME` column, but your `User` class has `firstname` and `lastname` fields, you can add the following persistent `name` property to the class (example 2 from the `domainmodel` folder source code).

Listing 3.2 POJO implementation of the User class with logic in accessor methods

Path: Ch03/domainmodel/src/main/java/com/manning/javapersistence/ch03/ex02/User.java

```
public class User {

    private String firstname;
    private String lastname;

    public String getName() {
        return firstname + ' ' + lastname;
    }

    public void setName(String name) {
        StringTokenizer tokenizer = new StringTokenizer(name);
        firstname = tokenizer.nextToken();
        lastname = tokenizer.nextToken();
    }

}
```

Later, you'll see that a custom-type converter in the persistence service is a better way to handle many of these kinds of situations. It helps to have several options.

Another issue to consider is *dirty checking*. Hibernate automatically detects state changes to synchronize the updated state with the database. It's usually safe to return a different instance from the getter method than the instance passed by Hibernate to the setter. Hibernate compares them by value—not by object identity—to determine whether the attribute's persistent state needs to be updated. For example, the following getter method doesn't result in unnecessary SQL UPDATES:

```
public String getFirstname() {
    return new String(firstname);
}
```

There is an important comment to make here, and this refers to *dirty checking* when persisting collections. If we have the `Item` entity and this one has a `Set<Bid>` field accessed through the `setBids` setter, then this code:

```
item.setBids(bids);
em.persist(item);
item.setBids(bids);
```

will result in an unnecessary SQL UPDATE. This happens because Hibernate has its own collection implementations as `PersistentSet`, `PersistentList`, or `PersistentMap`. Providing setters for an entire collection is not a good practice anyway.

How does Hibernate handle exceptions when your accessor methods throw them? If Hibernate uses accessor methods when loading and storing instances and a `RuntimeException` (unchecked) is thrown, the current transaction is rolled back, and the exception is yours to handle in the code that called the Jakarta Persistence (or Hibernate native) API. If you throw a checked application exception, Hibernate wraps the exception into a `RuntimeException`.

Next, we'll focus on the relationships between entities and associations between persistent classes.

3.2.4 Implementing POJO associations

You'll now see how to associate and create different kinds of relationships between objects: one-to-many, many-to-one, and bidirectional relationships. We'll look at the scaffolding code needed to create these associations, how to simplify relationship management, and how to enforce the integrity of these relationships.

You create properties to express associations between classes, and you (typically) call accessor methods to navigate from instance to instance at runtime. Let's consider the associations defined by the `Item` and `Bid` persistent classes, as shown in figure 3.4.

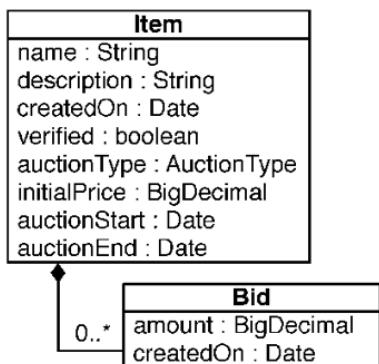


Figure 3.4 Associations between the `Item` and `Bid` classes

We left out the association-related attributes, `Item#bids`, and `Bid#item`. These properties and the methods that manipulate their values are called *scaffolding code*.

This is what the scaffolding code for the `Bid` class looks like:

```

Path: Ch03/domainmodel/src/main/java/com/manning/javapersistence/ch03/ex03/Bid.java
public class Bid {

    private Item item;

    public Item getItem() {
        return item;
    }

    public void setItem(Item item) {
        this.item = item;
    }
}
  
```

The `item` property allows navigation from a `Bid` to the related `Item`. This is an association with *many-to-one* multiplicity; users can make many bids for each item. Here is the `Item` class's scaffolding code:

```
Path: Ch03/domainmodel/src/main/java/com/manning/javapersistence/ch03/ex03/Item.java

public class Item {
    private Set<Bid> bids = new HashSet<>();

    public Set<Bid> getBids() {
        return Collections.unmodifiableSet(bids);
    }
}
```

This association between the two classes allows *bidirectional* navigation: the *many-to-one* is from this perspective a *one-to-many* multiplicity. One item can have many bids – they are of the same type but were generated during the auction by different users and with different amounts, as shown in table 3.1.

Table 3.1 One Item has many Bids, generated during the auction

Item	Bid	User	Amount
1	1	John	100
1	2	Mike	120
1	3	John	140

The scaffolding code for the `bids` property uses a collection interface type, `java.util.Set`. JPA requires interfaces for collection-typed properties, where you must use `java.util.Set`, `java.util.List`, or `java.util.Collection` rather than `HashSet`, for example. It's good practice to program to collection interfaces anyway, rather than concrete implementations, so this restriction shouldn't bother you.

You choose a `Set` and initialize the field to a new `HashSet` because the application disallows duplicate bids. This is good practice because you avoid any `NullPointerExceptions` when someone is accessing the property of a new `Item` – this one will have an empty set of bids. The JPA provider is also required to set a non-empty value on any mapped collection-valued property: for example, when an `Item` without bids is loaded from the database. (It doesn't have to use a `HashSet`; the implementation is up to the provider. Hibernate has its own collection implementations with additional capabilities—for example, dirty checking.)

Shouldn't bids on an item be stored in a list?

The first reaction is often to preserve the order of elements as they're entered by users because this may also be the order in which you will show them later. Certainly, in an auction application, there has to be some defined order in which the user sees bids for an item—for example, the highest bid first or the newest bid last. You might even work with a `java.util.List` in your user interface code to sort and display bids of an item. That doesn't mean this display order should be durable; data integrity isn't affected by the order in which bids are displayed. You need to store the amount of each bid, so you can find the highest bid, and you need to store a timestamp for each bid when it's created, so you can find the newest bid. When in doubt, keep your system flexible and sort the data when it's retrieved from the datastore (in a query) and/or shown to the user (in Java code), not when it's stored.

Accessor methods for associations need to be declared `public` only if they're part of the external interface of the persistent class used by the application logic to create a link between two instances. We'll now focus on this issue because managing the link between an `Item` and a `Bid` is much more complicated in Java code than it is in an SQL database, with declarative foreign key constraints. In our experience, engineers are often unaware of this complication arising from a network object model with bidirectional references (pointers). Let's walk through the issue step by step.

The basic procedure for linking a `Bid` with an `Item` looks like this:

```
anItem.getBids().add(aBid);
aBid.setItem(anItem);
```

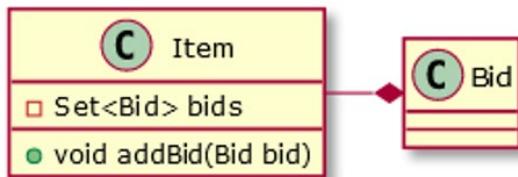


Figure 3.5 Linking a Bid with an Item, step 1: adding a Bid to the set of Bids from the Item

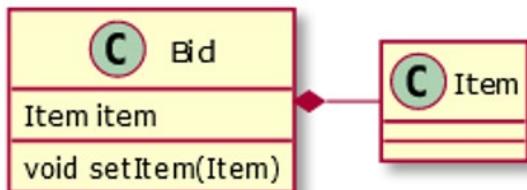


Figure 3.6 Linking a Bid with an Item, step 2: setting the Item on the side of the Bid

Whenever you create this bidirectional link, two actions are required:

- You must add the `Bid` to the `bids` collection of the `Item` (fig. 3.5).
- The `item` property of the `Bid` must be set (fig. 3.6).

JPA doesn't manage persistent associations. If you want to manipulate an association, you must write the same code you would write without Hibernate. If an association is bidirectional, you must consider both sides of the relationship. If you ever have problems understanding the behavior of associations in JPA, just ask yourself, "What would I do without Hibernate?" Hibernate doesn't change the regular Java semantics.

We recommend that you add convenience methods that group these operations, allowing reuse and helping ensure correctness, and in the end, guaranteeing data integrity (a `Bid` is required to have a reference to an `Item`). The next listing shows such a convenience method in the `Item` class (example 3 from the `domainmodel` folder source code).

Listing 3.3 A convenience method simplifies relationship management

Path: Ch03/domainmodel/src/main/java/com/manning/javapersistence/ch03/ex03/Item.java

```
public void addBid(Bid bid) {
    if (bid == null)
        throw new NullPointerException("Can't add null Bid");
    if (bid.getItem() != null)
        throw new IllegalStateException(
            "Bid is already assigned to an Item");
    bids.add(bid);
    bid.setItem(this);
}
```

The `addBid()` method not only reduces the lines of code when dealing with `Item` and `Bid` instances but also enforces the cardinality of the association. You avoid errors that arise from leaving out one of the two required actions. You should always provide this kind of grouping of operations for associations, if possible. If you compare this with the relational model of foreign keys in an SQL database, you can easily see how a network and a pointer model complicate a simple operation: instead of a declarative constraint, you need procedural code to guarantee data integrity.

Because you want `addBid()` to be the only externally visible mutator method for the `bids` of an item (possibly in addition to a `removeBid()` method), consider making the `Bid#setItem()` method package-visible.

The `Item#getBids()` getter method should not return a modifiable collection, so clients can use it to make changes that aren't reflected on the inverse side. Bids added directly to the collection belong to an item, but they wouldn't have a reference to that item—an inconsistent state, according to your database constraints. To prevent this, you can wrap the internal collection before returning it from the getter method, with `Collections.unmodifiableCollection(c)` and `Collections.unmodifiableSet(s)`. The client then gets an exception if it tries to modify the collection; you, therefore, force every modification to go through the relationship management method that guarantees integrity. It is good practice anyway to return an unmodifiable collection from your classes so that the client does not have direct access to it.

An alternative strategy is immutable instances. For example, you could enforce integrity by requiring an `Item` argument in the constructor of `Bid`, as shown in the following listing (example 4 from the `domainmodel` folder source code).

Listing 3.4 Enforcing integrity of relationships with a constructor

Path: Ch03/domainmodel/src/main/java/com/manning/javapersistence/ch03/ex04/Bid.java

```
public class Bid {

    private Item item;

    public Bid(Item item) {
        this.item = item;
        item.bids.add(this); // Bidirectional
    }

    public Item getItem() {
        return item;
    }
}
```

In this constructor, the `item` field is set; no further modification of the field value should occur. The collection on the “other” side is also updated for a bidirectional relationship, while the `bids` field from the `Item` class is now package-private. There is no `Bid#setItem()` method.

There are several problems with this approach. First, Hibernate can’t call this constructor. You need to add a no-argument constructor for Hibernate, and it needs to be at least package-visible. Furthermore, because there is no `setItem()` method, Hibernate would have to be configured to access the `item` field directly. This means the field can’t be `final`, so the class isn’t guaranteed to be immutable.

In the examples in this book, we’ll sometimes write scaffolding methods such as the `Item#addBid()` shown earlier, or we may have additional constructors for required values. It’s up to you how many convenience methods and layers you want to wrap around the persistent association properties and/or fields, but we recommend being consistent and applying the same strategy to all your domain model classes. For the sake of readability, we won’t always show convenience methods, special constructors, and other such scaffolding in future code samples and assume you’ll add them according to your own taste and requirements.

You now have seen domain model classes, how to represent their attributes and the relationships between them. Next, we’ll increase the level of abstraction, adding metadata to the domain model implementation and declaring aspects such as validation and persistence rules.

3.3 Domain model metadata

Metadata is data about data, so domain model metadata is information about your domain model. For example, when you use the Java reflection API to discover the names of classes

of your domain model or the names of their attributes, you're accessing domain model metadata.

ORM tools also require metadata to specify the mapping between classes and tables, properties and columns, associations and foreign keys, Java types and SQL types, and so on. This object/relational mapping metadata governs the transformation between the different type systems and relationship representations in object-oriented and SQL systems. JPA has a metadata API, which you can call to obtain details about the persistence aspects of your domain model, such as the names of persistent entities and attributes. First, it's your job as an engineer to create and maintain this information.

JPA standardizes two metadata options: annotations in Java code and externalized XML descriptor files. Hibernate has some extensions for native functionality, also available as annotations and/or XML descriptors. Usually, we prefer annotations as the primary source of mapping metadata. After reading this section, you'll have the background information to make an informed decision for your own project.

We'll also discuss *Bean Validation* (JSR 303) and how it provides declarative validation for your domain model (or any other) classes. The reference implementation of this specification is the *Hibernate Validator* project. Most engineers today prefer Java annotations as the primary mechanism for declaring metadata.

3.3.1 Annotation-based metadata

The big advantage of annotations is to put metadata next to the information it describes instead of separating it physically into a different file. Here's an example:

```
import javax.persistence.Entity;
@Entity
public class Item { }
```

You can find the standard JPA mapping annotations in the `javax.persistence` package. This example declares the `Item` class as a persistent entity using the `@javax.persistence.Entity` annotation. All of its attributes are now automatically persistent with a default strategy. That means you can load and store instances of `Item`, and all properties of the class are part of the managed state.

Annotations are type-safe, and the JPA metadata is included in the compiled class files. The annotations are still accessible at runtime, and Hibernate reads the classes and metadata with Java reflection when the application starts. The IDE can also easily validate and highlight annotations—they're regular Java types, after all. If you refactor your code, you rename, delete, or move classes and properties all the time. Most development tools and editors can't refactor XML elements and attribute values, but annotations are part of the Java language and are included in all refactoring operations.

Is my class now dependent on JPA?

Yes, but it's a compile-time-only dependency. You need JPA libraries on your classpath when compiling the source of your domain model class. The Jakarta Persistence API isn't required on the classpath when you create an instance of the class: for example, in a client application that doesn't execute any JPA code. Only when you access the annotations through reflection at runtime (as Hibernate does internally when it reads your metadata) will you need the packages on the classpath.

When the standardized Jakarta Persistence annotations are insufficient, a JPA provider may offer additional annotations.

USING VENDOR EXTENSIONS

Even if you map most of your application's model with JPA-compatible annotations from the `javax.persistence` package, you may have to use vendor extensions at some point. For example, some performance-tuning options you'd expect to be available in high-quality persistence software are only available as Hibernate-specific annotations. This is how JPA providers compete, so you can't avoid annotations from other packages—there's a reason why you picked Hibernate.

This is the `Item` entity source code again with a Hibernate-only mapping option:

```
import javax.persistence.Entity;
@Entity
@org.hibernate.annotations.Cache(
    usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
)
public class Item {
```

We prefer to prefix Hibernate annotations with the full `org.hibernate.annotations` package name. Consider this good practice because you can easily see what metadata for this class is from the JPA specification and which is vendor-specific. You can also easily search your source code for “`org.hibernate.annotations`” and get a complete overview of all nonstandard annotations in your application in a single search result.

If you switch your Jakarta Persistence provider, you only have to replace the vendor-specific extensions where you can expect a similar feature set to be available with the most mature JPA implementations. Of course, we hope you'll never have to do this, and it doesn't often happen in practice—just be prepared.

Annotations on classes only cover metadata that applies to that particular class. You often need metadata at a higher level for an entire package or even the whole application.

GLOBAL ANNOTATION METADATA

The `@Entity` annotation maps a particular class. JPA and Hibernate also have annotations for global metadata. For example, a `@NamedQuery` has a global scope; you don't apply it to a particular class. Where should you place this annotation?

Although it's possible to place such global annotations in the source file of a class (any class, really, at the top), we'd rather keep global metadata in a separate file. Package-level annotations are a good choice; they're in a file called `package-info.java` in a particular

package directory. You will be able to look for them in one single place instead of browsing a few files. You can see an example of global named query declarations in the following listing (example 5 from the `domainmodel` folder source code).

Listing 3.5 Global metadata in a package-info.java file

```
Path: Ch03/domainmodel/src/main/java/com/manning/javapersistence/ch03/ex05/package-
info.java

@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByName",
        query = "select i from Item i order by i.name asc"
    )
    ,
    @org.hibernate.annotations.NamedQuery(
        name = "findItemBuyNowPriceGreaterThanOrEqual",
        query = "select i from Item i where i.buyNowPrice > :price",
        timeout = 60, // Seconds!
        comment = "Custom SQL comment"
    )
})
package com.manning.javapersistence.ch03.ex05;
```

Unless you've used package-level annotations before, the syntax of this file with the package and import declarations at the bottom is probably new to you.

Annotations will be our primary tool throughout this book for ORM metadata, and there is much to learn about this subject. Before we look at the alternative mapping style with XML files, let's use some simple annotations to improve the domain model classes with validation rules.

3.3.2 Applying constraints to Java objects

Most applications contain a multitude of data integrity checks. When you violate one of the simplest data-integrity constraints, you may get a `NullPointerException` when you expect a value to be available. Other examples are a string-valued property that shouldn't be empty (an empty string isn't `null`), a string that has to match a particular regular expression pattern, and a number or date value that must be within a certain range.

These business rules affect every layer of an application: The user interface code has to display detailed and localized error messages. The business and persistence layers must check input values received from the client before passing them to the datastore. The SQL database has to be the final validator, ultimately guaranteeing the integrity of durable data.

The idea behind Bean Validation is that declaring rules such as "This property can't be `null`" or "This number has to be in the given range" is much easier and less error-prone than writing if-then-else procedures repeatedly. Furthermore, declaring these rules on the central component of your application, the domain model implementation, enables integrity checks in every layer of the system. The rules are then available to the presentation and persistence layers. And if you consider how data-integrity constraints affect not only your Java

application code but also your SQL database schema—which is a collection of integrity rules—you might think of Bean Validation constraints as additional ORM metadata.

Look at the following extended `Item` domain model class from the `validation` folder source code.

Listing 3.6 Applying validation constraints on `Item` entity fields

```
Path: Ch03/validation/src/main/java/com/manning/javapersistence/ch03/validation/Item.java
import javax.validation.constraints.Future;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.util.Date;

public class Item {
    @NotNull
    @Size(
        min = 2,
        max = 255,
        message = "Name is required, maximum 255 characters."
    )
    private String name;

    @Future
    private Date auctionEnd;
}
```

We add two more attributes—the `name` of an item and the `auctionEnd` date—when an auction concludes. Both are typical candidates for additional constraints: we want to guarantee that the name is always present and human-readable (one-character item names don't make much sense), but it shouldn't be too long—your SQL database will be most efficient with variable-length strings up to 255 characters, and your user interface also has some constraints on visible label space. The ending time of an auction obviously should be in the future. If we don't provide an error message, a default message will be used. Messages can be keys to external properties files for internationalization.

The validation engine will access the fields directly if you annotate the fields. If you prefer calls through accessor methods, annotate the getter method with validation constraints, not the setter. Annotations on setters are not supported. Then constraints are part of the class's API and included in its Javadoc, making the domain model implementation easier to understand. Note that this is independent of access by the JPA provider; that is, Hibernate Validator may call accessor methods, whereas Hibernate ORM may call fields directly.

Bean Validation isn't limited to built-in annotations; you can create your own constraints and annotations. With a custom constraint, you can even use class-level annotations and validate several attribute values at the same time on an instance of the class. The following test code shows how you can manually check the integrity of an `Item` instance.

Listing 3.7 Testing an Item instance for constraint violations

```

Path: Ch03/validation/src/test/java/com/manning/javapersistence/ch03/validation/ModelValidation.java
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Item item = new Item();
item.setName("Some Item");
item.setAuctionEnd(new Date());

Set<ConstraintViolation<Item>> violations = validator.validate(item);

ConstraintViolation<Item> violation = violations.iterator().next();
String failedPropertyName =
    violation.getPropertyPath().iterator().next().getName();

// Validation error, auction end date was not in the future!
assertAll(() -> assertEquals(1, violations.size()),
    () -> assertEquals("auctionEnd", failedPropertyName),
    () -> {
        if (Locale.getDefault().getLanguage().equals("en"))
            assertEquals(violation.getMessage(),
                "must be a future date");
    });
}

```

We're not going to explain this code in detail but offer it for you to explore. You'll rarely write this kind of validation code; most of the time, this aspect is automatically handled by your user interface and persistence framework. It's therefore important to look for Bean Validation integration when selecting a UI framework.

Hibernate, as required from any JPA provider, also automatically integrates with Hibernate Validator if the libraries are available on the classpath and offers the following features:

- You don't have to manually validate instances before passing them to Hibernate for storage.
- Hibernate recognizes constraints on persistent domain model classes and triggers validation before database insert or update operations. When validation fails, Hibernate throws a `ConstraintViolationException`, containing the failure details to the code calling persistence-management operations.
- The Hibernate toolset for automatic SQL schema generation understands many constraints and generates SQL DDL-equivalent constraints for you. For example, an `@NotNull` annotation translates into an SQL `NOT NULL` constraint, and an `@Size(n)` rule defines the number of characters in a `VARCHAR(n)`-typed column.

You can control this behavior of Hibernate with the `<validation-mode>` element in your `persistence.xml` configuration file. The default mode is `AUTO`, so Hibernate will only validate if it finds a Bean Validation provider (such as Hibernate Validator) on the classpath of the running application. With mode `CALLBACK`, validation will always occur, and you'll get a deployment error if you forget to bundle a Bean Validation provider. The `NONE` mode disables automatic validation by the JPA provider.

You'll see Bean Validation annotations again later in this book; you'll also find them in the example code bundles. At this point, we could write much more about Hibernate Validator, but we'd only repeat what is already available in the project's excellent reference guide. You can check it at https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/. Have a look, and find out more about features such as validation groups and the metadata API for the discovery of constraints.

3.3.3 Externalizing metadata with XML files

You can replace or override every annotation in JPA with an XML descriptor element. In other words, you don't have to use annotations if you don't want to, or if keeping mapping metadata separate from source code is for whatever reason advantageous to your system design. Keeping the separate mapping metadata has the benefit of not cluttering the JPA annotations with Java code in exchange for losing the type-safety. This approach will make the Java classes more reusable. It is, however, less in use nowadays, but we'll analyze it as you may still encounter it or choose this approach for your projects.

XML METADATA WITH JPA

The following listing shows a JPA XML descriptor for a particular persistence unit (the `metadat/xmljpa` folder source code).

Listing 3.8 JPA XML descriptor containing the mapping metadata of a persistence unit

```
Path: Ch03/metadat/xmljpa/src/test/resources/META-INF/persistence.xml
<entity-mappings
    version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
        http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd">

    <persistence-unit-metadata> #A
        <xml-mapping-metadata-complete/> #B
        <persistence-unit-defaults> #C
            <delimited-identifiers/> #D
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    <entity class="com.manning.javapersistence.ch03.metadat/xmljpa.Item" #E
        access="FIELD">
        <attributes> #F
            <id name="id"> #F
                <generated-value strategy="AUTO"/> #F
            </id> #F
            <basic name="name"/> #F
            <basic name="auctionEnd"> #F
                <temporal>TIMESTAMP</temporal> #F
            </basic> #F
        </attributes> #F
    </entity>
</entity-mappings>
```

#A First, we declare the global metadata.

#B We ignore all mapping annotations. If we include the `<xml-mapping-metadata-complete>` element, the JPA provider ignores all annotations on the domain model classes in this persistence unit and relies only on the mappings as defined in the XML descriptor(s).

#C As default settings, we escape all SQL columns, tables, and other names.

#D Escaping is useful if the SQL names are actually keywords (a "USER" table, for example).

#E We declare the `Item` class as an entity with field access.

#F Its attributes are the `id`, which is auto-generated, the `name`, and the `auctionEnd`, which is a temporal field.

The JPA provider automatically picks up this descriptor if you place it in a META-INF/orm.xml file on the classpath of the persistence unit. If you prefer to use a different name or several files, you'll have to change the configuration of the persistence unit in your META-INF/persistence.xml file:

```
<persistence-unit name="persistenceUnitName">
    ...
    <mapping-file>file1.xml</mapping-file>
    <mapping-file>file2.xml</mapping-file>
    ...
</persistence-unit>
If you don't want to ignore but override the annotation metadata, don't mark the XML
descriptors as "complete", and name the class and property to override:
<entity class="com.manning.javapersistence.ch03.metadataxmljpa.Item">
    <attributes>
        <basic name="name">
            <column name="ITEM_NAME"/>
        </basic>
    </attributes>
</entity>
```

Here we map the `name` property to the `ITEM_NAME` column; by default, the property would map to the `NAME` column. Hibernate will now ignore any existing annotations from the `javax.persistence.annotation` and `org.hibernate.annotations` packages on the `name` property of the `Item` class. But Hibernate doesn't ignore Bean Validation annotations and still applies them for automatic validation and schema generation! All other annotations on the `Item` class are also recognized. Note that we don't specify an access strategy in this mapping, so field access or accessor methods are used, depending on the position of the `@Id` annotation in `Item`. (We'll get back to this detail in the next chapter.)

We won't talk much about JPA XML descriptors in this book. The syntax of these documents is a 1:1 mirror of the JPA annotation syntax, so you shouldn't have any problems writing them. We'll focus on the important aspect: the mapping strategies. The syntax used to write down metadata is secondary.

3.3.4 Accessing metadata at runtime

The JPA specification provides programming interfaces for accessing the metamodel of persistent classes. There are two flavors of the API. One is more dynamic in nature and similar to basic Java reflection. The second option is a static metamodel. For both options, access is read-only; you can't modify the metadata at runtime.

THE DYNAMIC METAMODEL API IN JAKARTA PERSISTENCE

Sometimes—for example, when you want to write some custom validation or generic UI code—you’d like to get programmatic access to the persistent attributes of an entity. You’d like to know what persistent classes and attributes your domain model has dynamically. The code in the next listing shows how to read metadata with Jakarta Persistence interfaces (the `metamodel` folder source code).

Listing 3.9 Obtaining entity type information with the Metamodel API

```
Path: Ch03/metamodel/src/test/java/com/manning/javapersistence/ch03/metamodel/MetamodelTest.java
Metamodel metamodel = emf.getMetamodel();
Set<managedType> managedTypes = metamodel.getManagedTypes();
managedType itemType = managedTypes.iterator().next();

assertAll(() -> assertEquals(1, managedTypes.size()),
          () -> assertEquals(
                  Type.PersistenceType.ENTITY,
                  itemType.getPersistenceType()));
```

You can get the `Metamodel` from either the `EntityManagerFactory`, of which you typically have only one instance in an application per data source or, if it’s more convenient, from calling `EntityManager#getMetamodel()`. The set of managed types contains information about all persistent entities and embedded classes (which we’ll discuss in the next chapter). In this example, there’s only one: the `Item` entity. This is how you can dig deeper and find out more about each attribute.

Listing 3.10 Obtaining entity attribute information with the Metamodel API

```
Path: Ch03/metamodel/src/test/java/com/manning/javapersistence/ch03/metamodel/MetamodelTest.java
SingularAttribute idAttribute =
    itemType.getSingularAttribute("id"); #A
assertFalse(idAttribute.isOptional()); #B

SingularAttribute nameAttribute =
    itemType.getSingularAttribute("name"); #C

assertAll(() -> assertEquals(String.class, nameAttribute.getJavaType()), #D
          () -> assertEquals(
                  Attribute.PersistentAttributeType.BASIC, #D
                  nameAttribute.getPersistentAttributeType() #D
          ));

SingularAttribute auctionEndAttribute =
    itemType.getSingularAttribute("auctionEnd"); #E
assertAll(() -> assertEquals(Date.class, #F
          auctionEndAttribute.getJavaType()),
          () -> assertFalse(auctionEndAttribute.isCollection()), #F
          () -> assertFalse(auctionEndAttribute.isAssociation()) #F
);
```

```

#A The attributes of the entity are accessed with a string: id
#C name
#E auctionEnd. This obviously isn't type-safe, and if you change the names of the attributes, this code becomes
    broken and obsolete. The strings aren't automatically included in the refactoring operations of your IDE.
#B We check that the id attribute is not optional. This means that it cannot be NULL, being the primary key.
#D We check that the name attribute has the String Java type and the basic persistent attribute type.
#F We check that the auctionEnd attribute has the Date Java type and that it is not a collection or an association.

```

JPA also offers a static type-safe metamodel.

USING A STATIC METAMODEL

In Java (at least up to version 17), you can't access the fields or accessor methods of a bean in a type-safe fashion—only by their names, using strings. This is particularly inconvenient with JPA criteria querying, a type-safe alternative to string-based query languages. Here's an example:

```

Path:
    Ch03/metamodel/src/test/java/com/manning/javapersistence/ch03/metamodel/MetamodelTes
    t.java
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Item> query = cb.createQuery(Item.class);                      #A
Root<Item> fromItem = query.from(Item.class);
query.select(fromItem);
List<Item> items = em.createQuery(query).getResultList();

assertEquals(2, items.size());

```

#A The query is the equivalent of "select i from Item i". This query returns all items in the database; here, there are two. If you now want to restrict this result and only return items with a particular name, you have to use a like expression, comparing the name attribute of each item with the pattern set in a parameter:

```

Path:
    Ch03/metamodel/src/test/java/com/manning/javapersistence/ch03/metamodel/MetamodelTes
    t.java
Path<String> namePath = fromItem.get("name");
query.where(cb.like(namePath, cb.parameter(String.class, "pattern")));
List<Item> items = em.createQuery(query).
    setParameter("pattern", "%Item 1%").                                #A
    getResultList();
assertAll(() -> assertEquals(1, items.size()),
        () -> assertEquals("Item 1", items.iterator().next().getName()));

```

#A The query is the equivalent of "select i from Item i where i.name like :pattern". Notice how the namePath lookup requires the name string. This is where the type-safety of the criteria query breaks down. You can rename the Item entity class with your IDE's refactoring tools, and the query will still work. But as soon as you touch the Item#name property, manual adjustments are necessary. Luckily, you'll catch this when the test fails.

A much better approach, safe for refactoring and detecting mismatches at compile-time and not runtime, is the type-safe static metamodel:

```
Path: Ch03/metamodel/src/test/java/com/manning/javapersistence/ch03/metamodel/MetamodelTest.java
query.where(
    cb.like(
        fromItem.get(Item_.name),
        cb.parameter(String.class, "pattern")
    )
);
```

The special class here is `Item_`; note the underscore. This class is a metadata class and lists all the attributes of the `Item` entity class:

```
Path: Ch03/metamodel/target/classes/com/manning/javapersistence/ch03/metamodel/Item_.class
@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Item.class)
public abstract class Item_ {

    public static volatile SingularAttribute<Item, Date> auctionEnd;
    public static volatile SingularAttribute<Item, String> name;
    public static volatile SingularAttribute<Item, Long> id;

    public static final String AUCTION_END = "auctionEnd";
    public static final String NAME = "name";
    public static final String ID = "id";

}
```

This class may be automatically generated. The *Hibernate JPA2 Metamodel Generator* (a distinct subproject of the Hibernate suite) takes care of this. Its only purpose is to generate static metamodel classes from your managed persistent classes. You need to add this Maven dependency to your `pom.xml` file.

```
Path: Ch03/metamodel/pom.xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
    <version>5.3.14.Final</version>
</dependency>
```

It will run automatically whenever you build your project and generate the appropriate `Item_` metadata class. You will find the generated classes in the `target\generated-sources` folder.

This chapter discussed constructing the domain model and the dynamic and static metamodel. Although you've seen some mapping constructs in the previous sections, we haven't introduced more sophisticated class and property mappings so far. You should now decide which mapping metadata strategy you'd like to use in your project—we recommend the largely used annotations versus the already less used XML—and then read more about class and property mappings in part 2, starting with chapter 4.

3.4 Summary

- We analyze different abstract notions like the information model, data model, and then jump into JPA/Hibernate.
- We implemented persistent classes free of any crosscutting concerns like logging, authorization, and transaction demarcation; the persistent classes only depend on JPA at compile time. Even persistence-related concerns should not leak into the domain model implementation.
- Transparent persistence is important if we want to execute and test the business objects independently and easily.
- We demonstrated the best practices and requirements for the POJO and JPA entity programming model and what concepts they have in common with the old JavaBean specification.
- We analyzed how to access the metadata – using the dynamic metamodel and the static metamodel.
- We are now ready to implement more complex mappings for the persistent classes.

4

Working with Spring Data JPA

This chapter covers

- Overviewing Spring Data and its modules
- Examining the main concepts of Spring Data JPA
- Investigating the Query Builder mechanisms
- Examining projections, modifying and deleting queries
- Examining Query by Example

Spring Data is an umbrella project containing many projects specific to various databases. These projects are developed in partnership with the companies creating the technologies themselves. Spring Data intends to provide an abstraction for data access and maintain at the same time the underlying specificities (including here relational and non-relational databases).

Among the general features provided by Spring Data, we mention:

- Integration with Spring via JavaConfig and XML configuration
- Repository and custom object-mapping abstractions
- Integration with custom repository code
- Dynamic query creation based on repository methods names
- Integration with other Spring projects, as Spring Boot

We have enumerated, in chapter 2, the main Spring Data modules. We'll focus on Spring Data JPA here, as we'll intensively use it during the next chapters to interact with the databases. Spring Data JPA is largely used as an alternative to access databases from Java programs. It provides a new layer of abstraction on top of a JPA provider (as Hibernate), in the spirit of the Spring framework, taking control of the configuration and transactions management. We'll illustrate many of our examples in the next chapters using it, so we'll analyze here its capabilities in depth. We will still define and manage our entities using JPA and Hibernate, but we'll provide Spring Data JPA as an alternative to interact with them.

4.1 Introducing Spring Data JPA

Spring Data JPA is providing support for interacting with JPA repositories. It is built on top of the functionality offered by the Spring Data Commons project and the one offered by the JPA provider (Hibernate in our case), as shown in figure 4.1. To review the main Spring Data modules, refer to chapter 2.

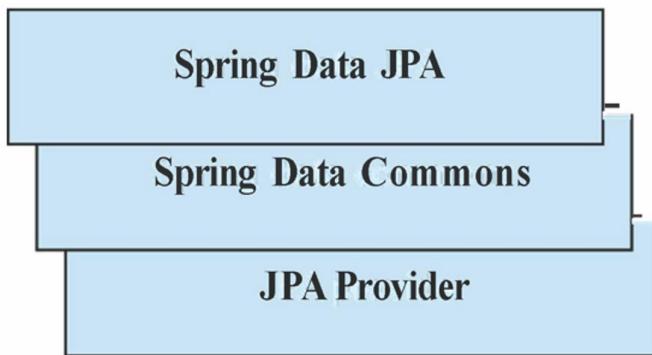


Figure 4.1 Spring Data JPA is built on top of Spring Data Commons and the JPA provider

Across the book, we'll generally interact with databases using both Hibernate JPA and Spring Data as alternatives. That is why we consider appropriate to analyze now the main capabilities of Spring Data JPA, as they will be used along the next chapters. This chapter needs the knowledge provided in chapters 1-3 and will help you start using the most important capabilities of Spring Data JPA. We'll examine more Spring Data JPA features when they will be needed and more Spring Data projects in their dedicated chapters.

As already demonstrated in chapter 2 (see 2.6 "Hello World" with Spring Data JPA), a few things that Spring Data JPA can do to facilitate the interaction with the database are:

- Configuration of the datasource bean
- Configuration of the entity manager factory bean
- Configuration of the transaction manager bean
- Transaction management through annotations

4.2 Starting a new Spring Data JPA project

We'll use the CaveatEmptor example application that we introduced in the previous chapter to demonstrate and analyze the Spring Data JPA capabilities. We'll demonstrate the work on an application that will manage and persist the CaveatEmptor users, using Spring Data JPA as persistence framework, having Hibernate JPA as underlying JPA provider. Spring Data JPA may execute CRUD operations and queries against a database. It may be backed by different JPA implementations (Hibernate in our case) and provides another layer of abstraction to interact with the databases. We'll create a Spring Boot application to use Spring Data JPA. To

do this, we'll access the Spring Initializr website at <https://start.spring.io/> and will create a new Spring Boot project (figure 4.2), having the following characteristics:

- Group: com.manning.javapersistence
- Artifact: springdatajpa
- Description: Spring Data with Spring Boot

We'll also add the following dependencies:

- Spring Data JPA (this will add `spring-boot-starter-data-jpa` in the Maven pom.xml file)
- MySQL Driver (this will add `mysql-connector-java` in the Maven pom.xml file)

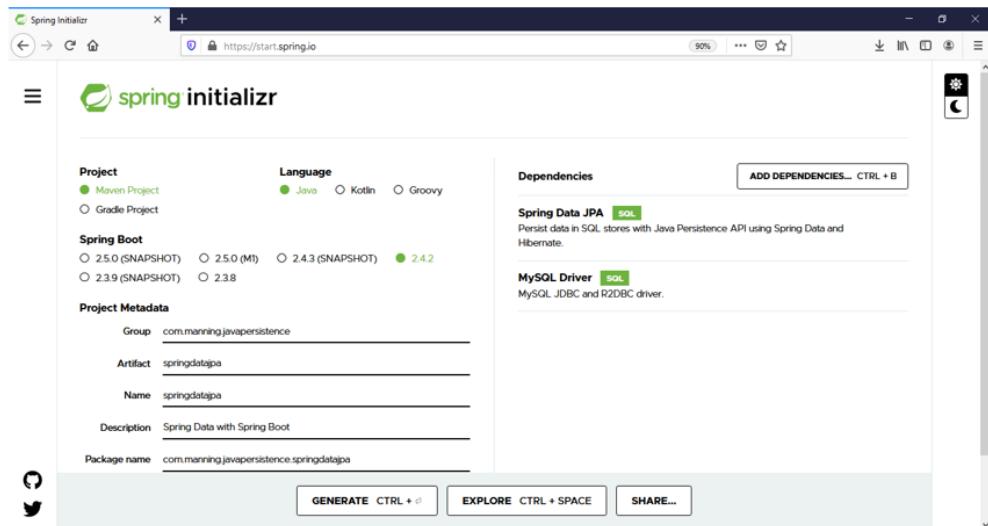


Figure 4.2 Creating a new Spring Boot project using Spring Data JPA and MySQL

After pressing the GENERATE button, the Spring Initializr website will provide an archive to be downloaded. This archive contains a Spring Boot project that uses Spring Data JPA and MySQL. We'll open this project in the IntelliJ IDEA IDE (figure 4.3).

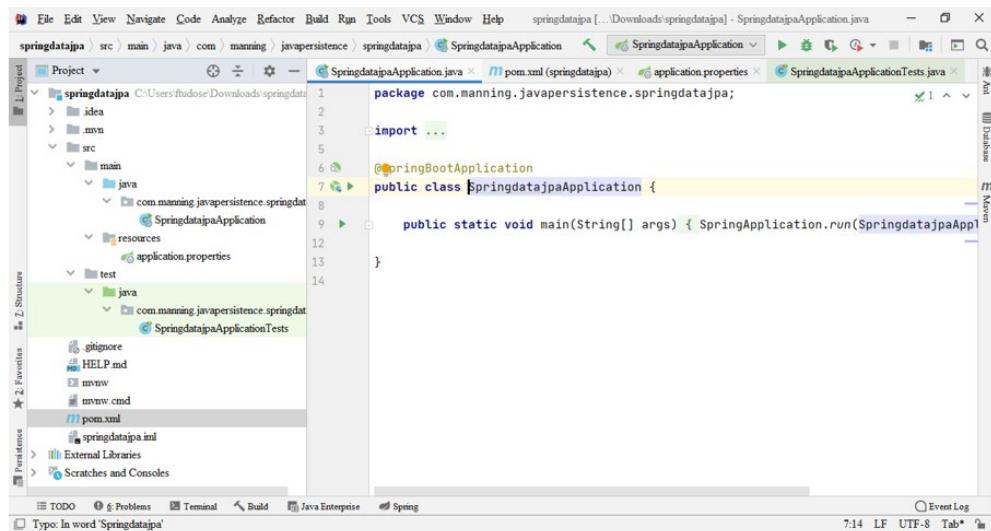


Figure 4.3 Opening the Spring Boot project that uses Spring Data JPA and MySQL

The skeleton of the project contains 4 files:

- `SpringDataJpaApplication`, including a skeleton `main` method
- `SpringDataJpaApplicationTests`, including a skeleton test method
- `application.properties`, empty for the beginning
- `pom.xml`, including the management information needed by Maven

As the first 3 files from the list above are standard ones, we'll take a closer look for now at the `pom.xml` file, as generated by Spring Initializr.

Listing 4.1 The pom.xml Maven file

```

Path: Ch04/springdatajpa/pom.xml
<parent>
    <groupId>org.springframework.boot</groupId>                      #A
    <artifactId>spring-boot-starter-parent</artifactId>                 #A
    <version>2.4.2</version>                                              #A
    <relativePath/> <!-- lookup parent from repository -->          #A
</parent>
<groupId>com.manning.javapersistence</groupId>                         #B
<artifactId>springdatajpa</artifactId>                                    #B
<version>0.0.1-SNAPSHOT</version>                                         #B
<name>springdatajpa</name>                                              #B
<description>Spring Data with Spring Boot</description>                  #B
<properties>
    <java.version>11</java.version>                                         #B
</properties>
<dependencies>
    <dependency>                                                       #C
        <groupId>org.springframework.boot</groupId>                     #C
        <artifactId>spring-boot-starter-data-jpa</artifactId>           #C
    </dependency>                                                       #C

    <dependency>                                                       #D
        <groupId>mysql</groupId>                                         #D
        <artifactId>mysql-connector-java</artifactId>                   #D
        <scope>runtime</scope>                                           #D
    </dependency>                                                       #D

    <dependency>                                                       #E
        <groupId>org.springframework.boot</groupId>                     #E
        <artifactId>spring-boot-starter-test</artifactId>                #E
        <scope>test</scope>                                            #E
    </dependency>                                                       #E
</dependencies>

<build>
    <plugins>
        <plugin>                                                 #F
            <groupId>org.springframework.boot</groupId>                 #F
            <artifactId>spring-boot-maven-plugin</artifactId>           #F
        </plugin>
    </plugins>
</build>

```

#A The parent POM is `spring-boot-starter-parent`. This Spring Boot Starter Parent is providing default configuration, dependency, and plugin management for the Maven applications. It also inherits dependency management from its parent, `spring-boot-dependencies`.

#B We indicate the `groupId`, `artifactId`, `version`, `name`, and `description` of the project, plus the Java version.

#C `spring-boot-starter-data-jpa` is the starter dependency used by Spring Boot to connect to a relational database through Spring Data JPA with Hibernate. It uses Hibernate as a transitive dependency.

#D `mysql-connector-java` is the JDBC driver for MySQL. It is a runtime dependency, indicating that it is not needed in the classpath for compiling, but only at runtime.

#E `spring-boot-starter-test` is the Spring Boot starter dependency for testing. The dependency is needed only for test compilation and execution phases.

#F `spring-boot-maven-plugin` is a utility plugin for building and running a Spring Boot project.

4.3 First steps to configure a Spring Data JPA project

We'll now write the entity class that will describe a `User` entity. The `CaveatEmptor` application has to keep track of the users interacting with it, so it is natural to start with the implementation of this class.

Listing 4.2 The User entity

```
Path: Ch04/springdatajpa/src/main/java/com/manning/javapersistence/springdatajpa/model/User.java
@Entity #A
@Table(name = "USERS") #A
public class User { #A

    @Id #B
    @GeneratedValue #B
    private Long id; #B

    private String username; #C

    private LocalDate registrationDate; #C

    public User() { #D

    }

    public User(String username) { #D
        this.username = username;
    } #D

    public User(String username, LocalDate registrationDate) { #D
        this.username = username;
        this.registrationDate = registrationDate;
    } #D

    public Long getId() { #B
        return id;
    } #B

    public String getUsername() { #C
        return username;
    } #C

    public void setUsername(String username) { #C
        this.username = username;
    } #C

    public LocalDate getRegistrationDate() { #C
        return registrationDate;
    } #C

    public void setRegistrationDate(LocalDate registrationDate) { #C
        this.registrationDate = registrationDate;
    } #C

    @Override #E
    public String toString() { #E
```

```

        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            ", registrationDate=" + registrationDate +
            '}';
    }
}

#A We create the User entity and annotate it with the @Entity and @Table annotations. We indicate USERS as
#B We indicate the id field as the primary key and a getter for it. The @GeneratedValue annotation enables the
#C We declare the username and registrationDate fields, together with getters and setters.
#D We declare 3 constructors, including a no-arguments one. We remind that JPA requires a constructor with no
#E We also create the toString method to nicely display the instances of the User class.

```

We'll also create the UserRepository interface.

Listing 4.3 The UserRepository interface

Path:

```

Ch04/springdatajpa/src/main/java/com/manning/javapersistence/springdatajpa/repositor
ies/UserRepository.java
public interface UserRepository extends CrudRepository<User, Long> {
}

```

The UserRepository interface extends CrudRepository<User, Long>. This means that it is a repository of User entities, having a Long identifier. Remember, the User class has an id field annotated as @Id of type Long. We can directly call methods as save, findAll or findById, inherited from CrudRepository, and we can use them without any other additional information to execute the usual operations against a database. Spring Data JPA will create a proxy class implementing the UserRepository interface and implement its methods.

It is worth mentioning that CrudRepository is a generic persistence technology-agnostic interface that we can use not only for JPA/Relational databases but also for NoSQL databases. For example, we can easily change the database from MySQL to MongoDB without touching the implementation by changing the dependency from the originally added spring-boot-starter-data-jpa to spring-boot-starter-data-mongodb.

The next step will be to fill in the Spring Boot application.properties file. Spring Boot will automatically find and load the application.properties file from the classpath - and the src/main/resources folder is added by Maven to the classpath.

Listing 4.4 The application.properties file

```
Path: Ch04/springdatajpa/src/main/resources/application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/CH04_SPRINGDATAJPA      #A
spring.datasource.username=root                                #B
spring.datasource.password=                                    #B
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect #C
spring.jpa.show-sql=true                                     #D
spring.jpa.hibernate.ddl-auto=create                      #E
```

#A The application.properties file will indicate the URL of the database.

#B The username, and no password for access.

#C The Hibernate dialect is MySQL8, as the database to interact with is MySQL Release 8.0.

#D While executing, the SQL code is shown.

#E Every time the program is executed, the database will be created from scratch.

We'll write the code that saves two users to the database and then try to find them.

Listing 4.5 Persisting and finding User entities

```
Path:
    Ch04/springdatajpa/src/main/java/com/manning/javapersistence/springdatajpa/SpringDataJpaApplication.java
@SpringBootApplication
public class SpringDataJpaApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringDataJpaApplication.class, args);          #B
    }

    @Bean
    public ApplicationRunner configure(UserRepository userRepository) {      #C
        return env ->
        {
            User user1 = new User("beth", LocalDate.of(2020, Month.AUGUST, 3));#D
            User user2 = new User("mike",                                         #D
                                  LocalDate.of(2020, Month.JANUARY, 18));       #D

            userRepository.save(user1);                                         #E
            userRepository.save(user2);                                         #E

            userRepository.findAll().forEach(System.out::println);           #F
        };
    }
}
```

#A The @SpringBootApplication annotation, added by Spring Boot to the class containing the main method, will enable the Spring Boot auto-configuration mechanism, will enable the scan on the package where the application is located, and will allow registering extra beans in the context.

#B SpringApplication.run will load the stand-alone Spring application from the main method. It will create an appropriate ApplicationContext instance and will load beans.

#C Spring Boot will run the @Bean annotated method, returning an ApplicationRunner just before SpringApplication.run() finishes.

#D Create two users.

#E Save them to the database.

#F Retrieve them and display the information about them.

Running this application, we'll get the following output (according to the way the `toString()` method of the `User` class works):

```
User{id=1, username='beth', registrationDate=2020-08-03}
User{id=2, username='mike', registrationDate=2020-01-18}
```

4.4 Defining query methods with Spring Data JPA

We'll extend the `User` class by adding the fields `email`, `level`, and `active`. A user may have different levels, which will allow him or her to execute particular actions (for example, bidding above some amount). A user may be active or may be retired (previously active in the CaveatEmptor auction system, but not anymore). This is important information that the CaveatEmptor application needs to keep about its user. The source code demonstrated from now is to be found in the `springdatajpa2` folder.

Listing 4.6 The modified User class

Path:

```
Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/model/User.java
@Entity
@Table(name = "USERS")
public class User {

    @Id
    @GeneratedValue
    private Long id;

    private String username;

    private LocalDate registrationDate;

    private String email;

    private int level;

    private boolean active;

    public User() {
    }

    public User(String username) {
        this.username = username;
    }

    public User(String username, LocalDate registrationDate) {
        this.username = username;
        this.registrationDate = registrationDate;
    }

    //getters and setters
}
```

We'll start to add new methods to the `UserRepository` interface and use them inside newly created tests.

The `UserRepository` interface will now extend `JpaRepository` instead of `CrudRepository`. `JpaRepository` extends `PagingAndSortingRepository`, which, in turn, extends `CrudRepository`.

`CrudRepository` provides basic CRUD functionality. `PagingAndSortingRepository` offers convenient methods to do sorting and pagination of the records (to be addressed later in the chapter). `JpaRepository` offers JPA-related methods, as flushing the persistence context and delete records in a batch. Additionally, `JpaRepository` overwrites a few methods from `CrudRepository`, as `findAll`, `findAllById`, or `saveAll`, to return `List` instead of `Iterable`.

We'll also add a series of query methods to the `UserRepository` interface that will look like this:

Listing 4.7 The UserRepository interface with new methods

```
Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/repositories/UserRepository.java
public interface UserRepository extends JpaRepository<User, Long> {

    User findByUsername(String username);
    List<User> findAllByOrderByUsernameAsc();
    List<User> findByRegistrationDateBetween(LocalDate start, LocalDate end);
    List<User> findByUsernameAndEmail(String username, String email);
    List<User> findByUsernameOrEmail(String username, String email);
    List<User> findByUsernameIgnoreCase(String username);
    List<User> findByLevelOrderByUsernameDesc(int level);
    List<User> findByLevelGreaterThanOrEqualTo(int level);
    List<User> findByUsernameContaining(String text);
    List<User> findByUsernameLike(String text);
    List<User> findByUsernameStartingWith(String start);
    List<User> findByUsernameEndingWith(String end);
    List<User> findByActive(boolean active);
    List<User> findByRegistrationDateIn(Collection<LocalDate> dates);
    List<User> findByRegistrationDateNotIn(Collection<LocalDate> dates);

}
```

The purpose of the query methods is to retrieve information from the database. Spring Data JPA provides a query builder mechanism that will create the behavior of the repository methods based on their names. We'll analyze later the modifying queries, now we dive into the queries having `as` purpose to find information. This query mechanism removes prefixes and suffixes as `find...By`, `get...By`, `query...By`, `read...By`, `count...By` from the name of the method and parses the remaining of it.

You can declare methods containing expressions as `Distinct` to set a distinct clause, operators as `LessThan`, `GreaterThan`, `Between`, and `Like` or compound conditions with `And` or `Or`. You can apply static ordering with the `OrderBy` clause in the name of the query method referencing a property and providing a sorting direction (`Asc` or `Desc`). You can use the `IgnoreCase` for properties supporting such a clause. For deleting rows, you have to

replace `find` with `delete` in the names of the methods. Also, Spring Data JPA will look at the return type of the method. If you want to find a `User` and return it in an `Optional` container, the method return type will be `Optional<User>`. A full list of possible return types, together with detailed explanations, may be found at <https://docs.spring.io/spring-data/jpa/docs/2.5.2/reference/html/#appendix.query.return.types>.

The names of the methods need to follow the rules to determine the resulting query. If the method naming is wrong (for example, the entity property does not match in the query method), you will get an error while the application context is loaded.

Table 4.1 describes the keywords that Spring Data JPA supports and how each method name is transposed in JPQL.

Table 4.1 Keywords usage in Spring Data JPA and generated JPQL

Keyword	Example	Generated JPQL
Is, Equals	<code>findByUsername</code> <code>findByUsernameIs</code> <code>findByUsernameEquals</code>	<code>... where e.username = ?1</code>
And	<code>findByUsernameAndRegistrationDate</code>	<code>... where e.username = ?1 and e.registrationdate = ?2</code>
Or	<code>findByUsernameOrRegistrationDate</code>	<code>... where e.username = ?1 or e.registrationdate = ?2</code>
LessThan	<code>findByRegistrationDateLessThan</code>	<code>... where e.registrationdate < ?1</code>
LessThanEqual	<code>findByRegistrationDateLessThanEqual</code>	<code>... where e.registrationdate <= ?1</code>
GreaterThan	<code>findByRegistrationDateGreaterThan</code>	<code>... where e.registrationdate > ?1</code>
GreaterThanOrEqual	<code>findByRegistrationDateGreaterThanOrEqual</code>	<code>... where e.registrationdate >= ?1</code>
Between	<code>findByRegistrationDateBetween</code>	<code>... where e.registrationdate between ?1 and</code>

		?2
OrderBy	findByRegistrationDateOrderByUsernameDes c	... where e.registrationdat e = ?1 order by e.username desc
Like	findByUsernameLike	... where e.username like ?1
NotLike	findByUsernameNotLike	... where e.username not like ?1
Before	findByRegistrationDateBefore	... where e.registrationdat e < ?1
After	findByRegistrationDateAfter	... where e.registrationdat e > ?1
Null, IsNull	findByRegistrationDate(Is)Null	... where e.registrationdat e is null
NotNull, IsNotNull	findByRegistrationDate(Is)NotNull	... where e.registrationdat e is not null
Not	findByUsernameNot	... where e.username <> ?1
In	findByRegistrationDateIn (Collection<LocalDate> dates)	... where e.registrationdat e in ?1
NotIn	findByRegistrationDateNotIn (Collection<LocalDate> dates)	... where e.registrationdat e not in ?1
True	findByActiveTrue	... where e.active = true
False	findByActiveFalse	... where e.active = false
StartingWith	findByUsernameStartingWith	... where e.username like ?1%
EndingWith	findByUsernameEndingWith	... where e.username like

		%?1
Containing	findByUsernameContaining	... where e.username like %?1%
IgnoreCase	findByUsernameIgnoreCase	.. where UPPER(e.username) = UPPER(?1)

As a base class for all future tests, we'll write the `SpringDataJpaApplicationTests` abstract class.

Listing 4.8 The SpringDataJpaApplicationTests abstract class

```
Path: Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/SpringDataJpaApplicationTests.java
@SpringBootTest                                     #A
@TestInstance(TestInstance.Lifecycle.PER_CLASS)      #B
abstract class SpringDataJpaApplicationTests {
    @Autowired                                         #C
    UserRepository userRepository;                      #C

    @BeforeAll                                         #D
    void beforeEach() {                                #D
        userRepository.saveAll(generateUsers());        #D
    }                                                   #D

    private static List<User> generateUsers() {        #E
        List<User> users = new ArrayList<>();

        User john = new User("john", LocalDate.of(2020, Month.APRIL, 13));
        john.setEmail("john@somedomain.com");
        john.setLevel(1);
        john.setActive(true);

        //create and set a total of 10 users

        users.add(john);
        //add a total of 10 users to the list

        return users;
    }

    @AfterAll                                         #F
    void afterEach() {                                #F
        userRepository.deleteAll();                     #F
    }                                                   #F
}
```

#A The `@SpringBootTest` annotation, added by Spring Boot to the initially created class, will tell Spring Boot to search the main configuration class (the `@SpringBootApplication` annotated class, for instance) and create the `ApplicationContext` to be used in the tests. We remind that the `@SpringBootApplication` annotation added by Spring Boot to the class containing the `main` method, will enable the Spring Boot auto-

configuration mechanism, will enable the scan on the package where the application is located, and will allow registering extra beans in the context.

#B Using the `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` annotation, we ask JUnit 5 to create one single instance of the test class and reuse it for all test methods. This will allow to make the `@BeforeAll` and `@AfterAll` annotated methods non-static and to directly use inside them the auto-wired `UserRepository` instance field.

#C We auto-wire a `UserRepository` instance. This auto-wiring is possible due to the `@SpringBootApplication` annotation, which enables the scan on the package where the application is located and registers the beans in the context.

#D The `@BeforeAll` annotated method will be executed once before executing all tests from a class that extends `SpringDataJpaApplicationTests`. This method will not be static.

#E The `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` annotation forces the creation of one single instance of the test class. It will save to the database the list of users created by the `generateUsers` method.

#F The `@AfterAll` annotated method will be executed once, after executing all tests from a class that extends `SpringDataJpaApplicationTests`. This method will not be static. The `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` annotation forces the creation of one single instance of the test class.

The next tests will extend this class and use the already populated database. To test the methods that belong now to `UserRepository`, we'll create the `FindUsersUsingQueriesTest` class and follow the same recipe for writing tests: call the repository method and verify its results.

Listing 4.9 The FindUsersUsingQueriesTest class

Path:

```
Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/FindUser
sUsingQueriesTest.java
public class FindUsersUsingQueriesTest extends SpringDataJpaApplicationTests {

    @Test
    void testFindAll() {
        List<User> users = userRepository.findAll();
        assertEquals(10, users.size());
    }

    @Test
    void testFindUser() {
        User beth = userRepository.findByUsername("beth");
        assertEquals("beth", beth.getUsername());
    }

    @Test
    void testFindAllByOrderByUsernameAsc() {
        List<User> users = userRepository.findAllByOrderByUsernameAsc();
        assertEquals(10, users.size());
        () -> assertEquals("beth", users.get(0).getUsername()),
        () -> assertEquals("stephanie",
            users.get(users.size() - 1).getUsername()));
    }

    @Test
    void testFindByRegistrationDateBetween() {
        List<User> users = userRepository.findByRegistrationDateBetween(
            LocalDate.of(2020, Month.JULY, 1),
            LocalDate.of(2020, Month.DECEMBER, 31));
        assertEquals(4, users.size());
    }

    //more tests
}
```

4.5 Limiting query results, sorting, and paging

The `first` and `top` keywords (used equivalently) can limit the results of query methods. The `top` and `first` keywords may be followed by an optional numeric value to indicate the maximum result size to be returned. If this numeric value is missing, the result size will be 1.

`Pageable` is an interface for pagination information. In practice, we use the `PageRequest` class that implements it. This one can specify the page number, the page size, and the sorting criterion.

We'll add the following methods to the `UserRepository` interface:

Listing 4.10 Limiting query results, sorting, and paging in the UserRepository interface

Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/repositories/UserRepository.java

```
User findFirstByOrderByUsernameAsc();
User findTopByOrderByRegistrationDateDesc();
Page<User> findAll(Pageable pageable);
List<User> findFirst2ByLevel(int level, Sort sort);
List<User> findByLevel(int level, Sort sort);
List<User> findByActive(boolean active, Pageable pageable);
```

We'll write the following tests to verify how these newly added methods work:

Listing 4.11 Testing limiting query results, sorting and paging

Path: Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/FindUsersSortingAndPagingTest.java

```
public class FindUsersSortingAndPagingTest extends SpringDataJpaApplicationTests {

    @Test
    void testOrder() {

        User user1 = userRepository.findFirstByOrderByUsernameAsc(); #A
        User user2 = userRepository.findTopByOrderByRegistrationDateDesc(); #A
        Page<User> userPage = userRepository.findAll(PageRequest.of(1, 3)); #B
        List<User> users = userRepository.findFirst2ByLevel(2, #C
                                                       Sort.by("registrationDate")); #C

        assertAll(
            () -> assertEquals("beth", user1.getUsername()),
            () -> assertEquals("julius", user2.getUsername()),
            () -> assertEquals(2, users.size()),
            () -> assertEquals(3, userPage.getSize()),
            () -> assertEquals("beth", users.get(0).getUsername()),
            () -> assertEquals("marion", users.get(1).getUsername())
        );
    }

    @Test
    void testFindByLevel() {
        Sort.TypedSort<User> user = Sort.sort(User.class); #D

        List<User> users = userRepository.findByLevel(3, #E
                                                      user.by(User::getRegistrationDate).descending()); #E
        assertAll(
            () -> assertEquals(2, users.size()),
            () -> assertEquals("james", users.get(0).getUsername())
        );
    }

    @Test
    void testFindByActive() {
        List<User> users = userRepository.findByActive(true, #F
                                                     PageRequest.of(1, 4, Sort.by("registrationDate"))); #F
    }
}
```

```

        assertAll(
            () -> assertEquals(4, users.size()),
            () -> assertEquals("burk", users.get(0).getUsername())
        );
    }
}

```

#A The first test will find the first user by ascending order of the username and the first user by descending order of the registration date.
#B Find all users, split them into pages, and return page number 1 of size 3 (the page numbering starts with 0).
#C Find the first 2 users with level 2, ordered by registration date.
#D The second test will define a sorting criterion on the `User` class. `Sort.TypedSort` extends `Sort` and can use method handles to define properties to sort by.
#E We find the users of level 3 and sort by registration date, descending.
#F The third test will find the active users sorted by registration date, split them into pages, and return page number 1 of size 4 (the page numbering starts with 0).

4.6 Streaming results

Query methods returning more than one result can use standard Java interfaces as `Iterable`, `List`, `Set`. Additionally, Spring Data supports `Streamable`. `Streamable` can be used as an alternative to `Iterable` or any collection type. It offers the possibility to concatenate `Streamables` and to directly filter and map over the elements.

We'll add the following methods to the `UserRepository` interface:

Listing 4.12 Adding methods that return Streamable in the UserRepository interface

```

Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/repositories/UserRepository.java
Streamable<User> findByEmailContaining(String text);
Streamable<User> findByLevel(int level);

```

We'll write the following tests to verify how these newly added methods work:

Listing 4.13 Testing methods that return Streamable

```

Path: Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/QueryResultsTest.java
@Test
void testStreamable() {
    try(Stream<User> result =
        userRepository.findByEmailContaining("someother")
        .and(userRepository.findByLevel(2))
        .stream().distinct()) {
        assertEquals(6, result.count());
    }
}

```

#A The test will call the `findByEmailContaining` method searching for emails containing the "someother" word.
#B The test will concatenate the resulted `Streamable` with the `Streamable` providing the users of level 2.

#C It will transform this into a stream and will keep the distinct users. The stream is given as a resource of the `try` block to be automatically closed. An alternative is to explicitly call the `close()` method. Otherwise, the stream would keep the underlying connection to the database.

#D We check that the resulted stream contains 6 users.

4.7 The @Query annotation

The programmer can create methods for which a custom query can be specified using the `@Query` annotation. Using the `@Query` annotation, the method name does not need to follow any naming convention. The custom query can be parameterized, identifying the parameters by position or by name, and binding these names in the query with the `@Param` annotation. The `@Query` annotation can generate native queries with the `nativeQuery` flag set to `true`. You should be aware however that native queries can impact the portability of the application. To sort the results, you can use a `Sort` object. The properties used to order by must resolve to a query property or a query alias.

Spring Data JPA supports SpEL (Spring Expression Language) expressions in queries defined using the `@Query` annotation. Spring Data JPA supports the `entityName` variable. Writing a query as `select e from #{#entityName} e`, `entityName` is resolved based on the `@Entity` annotation. In our case, `UserRepository` extends `JpaRepository<User, Long>`, `entityName` will resolve to `User`.

We'll add the following methods to the `UserRepository` interface:

Listing 4.14 Limiting query results, sorting, and paging in the UserRepository interface

```
Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/repositories/UserRepository.java
@Query("select count(u) from User u where u.active = ?1") #A
int findNumberOfUsersByActivity(boolean active); #A

@Query("select u from User u where u.level = :level and u.active = :active")#B
List<User> findByLevelAndActive(@Param("level") int level, #B
                                @Param("active") boolean active); #B

@Query(value = "SELECT COUNT(*) FROM USERS WHERE ACTIVE = ?1", #C
       nativeQuery = true) #C
int findNumberOfUsersByActivityNative(boolean active); #C

@Query("select u.username, LENGTH(u.email) as email_length from #D
       #{#entityName} u where u.username like %?1%") #D
List<Object[]> findByAsArrayAndSort(String text, Sort sort); #D
```

#A The `findNumberOfUsersByActivity` method will return the number of active users.

#B The `findByLevelAndActive` method will return the users with the `level` and `active` status given as named parameters. The `@Param` annotation will match the `:level` parameter of the query with the `level` argument of the method and the `:active` parameter of the query with the `active` argument of the method. This is especially useful when changing the order of the parameters from the signature of the method and the query is not updated.

#C The `findNumberOfUsersByActivityNative` method will return the number of users with a given active status. The `nativeQuery` flag set to `true` indicates that, unlike the previous queries, which are written with JPQL, this query is written using a native SQL specific to the database.

#D The `findByAsArrayAndSort` method will return a list of arrays, each array containing the `username` and the length of the `email`, after filtering based on the `username`. The second `Sort` parameter will allow ordering the result of the query based on different criteria.

We'll write the tests for these query methods, which are pretty straightforward. We'll analyze only the test written for the fourth query method, which allows a few variations of the sorting criterion.

Listing 4.15 Testing the query methods

```
Path: Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/QueryResultsTest.java
public class QueryResultsTest extends SpringDataJpaApplicationTests {

    // testing the first 3 query methods

    @Test
    void testFindByAsArrayAndSort() {
        List<Object[]> usersList1 =
            userRepository.findByAsArrayAndSort("ar", Sort.by("username")); #A
        List<Object[]> usersList2 =
            userRepository.findByAsArrayAndSort("ar",
                Sort.by("email_length").descending()); #B
        List<Object[]> usersList3 = userRepository.findByAsArrayAndSort(
            "ar", JpaSort.unsafe("LENGTH(u.email)")); #C
        assertAll(
            () -> assertEquals(2, usersList1.size()),
            () -> assertEquals("darren", usersList1.get(0)[0]),
            () -> assertEquals(21, usersList1.get(0)[1]),
            () -> assertEquals(2, usersList2.size()),
            () -> assertEquals("marion", usersList2.get(0)[0]),
            () -> assertEquals(26, usersList2.get(0)[1]),
            () -> assertEquals(2, usersList3.size()),
            () -> assertEquals("darren", usersList3.get(0)[0]),
            () -> assertEquals(21, usersList3.get(0)[1])
        );
    }
}
```

#A The `findByAsArrayAndSort` method will return the users whose `username` is like `%ar%` and will order them by `username`.

#B The `findByAsArrayAndSort` method will return the users whose `username` is like `%ar%` and will order them by `email_length`, `descending`. Note that the `email_length` alias needed to be specified inside the query to be used for ordering.

#C The `findByAsArrayAndSort` method will return the users whose `username` is like `%ar%` and will order them by `LENGTH(u.email)`. `JpaSort` is a class that extends `Sort` and may use something else than property references and aliases for sorting. An `unsafe` property handling means exactly that the provided `String` is not necessarily a property or an alias but can be an arbitrary expression inside the query.

An important note about the safety of working with the `@Query` annotation. For the previously analyzed methods that follow the Spring Data JPA naming conventions, if the method naming is wrong (for example, the entity property does not match in the query method), you will get an error while the application context is loaded. With the `@Query` annotation, if the query you wrote is wrong, you will get an error at runtime, when executing

that method. So, `@Query` annotated methods are more flexible, but they also provide less safety.

4.8 Projections

Not all attributes of an entity may be needed at a certain time, so we may access only some of them. For example, the front-end may reduce the burden of showing everything and display only the information that is of interest to the end-user. Consequently, instead of returning instances of the root entity managed by the repository, it might be needed to create projections based on certain attributes of those entities. Spring Data JPA can shape the return types to selectively return attributes of the entities.

An interface-based projection requires the creation of an interface that declares getter methods for the properties to be included in the projection. But such an interface can also compute specific values using the `@Value` annotation and SpEL expressions. Executing queries at runtime, the execution engine creates proxy instances of the interface for each returned element and forwards the calls to the exposed methods to the target object.

We'll create the `Projection` class and will add `UserSummary` as a nested interface. We group the projections, as they are logically connected.

Listing 4.16 Interface-based projection

```
Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/model/Projection.java
public class Projection {

    public interface UserSummary {

        String getUsername(); #A

        @Value("#{target.username} #{target.email}")
        String getInfo(); #B #B

    }
}
```

#A The `getUsername` method will return the `username` field.

#B The `getInfo` method is annotated with the `@Value` annotation and will return the concatenation of the `username` field, a space, and the `email` field.

How to approach projections in practice? If we include only methods as #A, we'll create a closed projection - this is an interface whose getters all correspond to properties of the target entity. Working with a closed projection, executing queries can be optimized by Spring Data JPA, as all properties needed by the projection proxy are known from the beginning.

If we include methods as #B, we'll create an open projection, which is more flexible. But Spring Data JPA will not be able to optimize the query execution, as the SpEL expression is evaluated at runtime and may include any property or combination of properties of the entity root.

In general, you should use projections when you need to provide some limited information and not expose the full entity. For performance reasons, you should prefer

closed projections whenever you know from the beginning which information to return. If you need to return the full object from a query, and also have the same query that only returns a projection, you may use the alternate naming conventions, for example, to name one method as `find...By`, and the other method as `get...By`.

A class-based projection requires the creation of a DTO (Data Transfer Object) class that declares the properties to be included in the projection and the getter methods. The usage of a class-based projection is similar to the one of interface-based projections. However, Spring Data JPA will no longer need to create proxy classes for managing projections. Spring Data JPA will instantiate exactly the class that declares the projection, and the properties to be included are determined by the parameter names of the constructor of the class.

We'll add `UsernameOnly` as a nested class of the `Projection` class:

Listing 4.17 Class-based projection

Path:

```
Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/model/Projection.java
```

```
public class Projection {  
  
    //...  
  
    public static class UsernameOnly {  
        private String username;  
  
        public UsernameOnly(String username) {  
            this.username = username;  
        }  
  
        public String getUsername() {  
            return username;  
        }  
  
    }  
}
```

#A The `UsernameOnly` class

#B The `username` field

#C The declared constructor

#D The `username` field exposed through a getter #D.

The methods that we'll add to the `UserRepository` interface will look like:

Path:

```
Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/repositories/UserRepository.java  
List<Projection.UserSummary> findByRegistrationDateAfter(LocalDate date);  
List<Projection.UsernameOnly> findByEmail(String username);
```

These repository methods use the same naming conventions we applied in our previous demonstrations and know their return type from compile-time as collections of projection types. But we may generify the return types of repository methods, which will make them dynamic. We'll add a new method to the `UserRepository` interface:

```
Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/repositories/UserRepository.java

<T> List<T> findByEmail(String username, Class<T> type);
```

We'll write the tests for these query methods using projections.

Listing 4.18 Testing query methods using projections

```
Path: Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/Projecti
onTest.java
public class ProjectionTest extends SpringDataJpaApplicationTests {

    @Test
    void testProjectionUsername() {

        List<Projection.UsernameOnly> users = #A
            userRepository.findByEmail("john@somedomain.com"); #A

        assertEquals(1, users.size()); #B
        assertEquals("john", users.get(0).getUsername()); #B
    }

    @Test
    void testProjectionUserSummary() {
        List<Projection.UserSummary> users = #C
            userRepository.findByName("john");
        assertEquals(1, users.size()); #C
        assertEquals("john", users.get(0).getUsername()); #C
        assertEquals("john@someotherdomain.com", users.get(0).getEmail()); #C
        assertEquals("john@someotherdomain.com", users.get(0).getInfo()); #C
    }

    @Test
    void testDynamicProjection() {
        List<Projection.UsernameOnly> usernames = #E
            userRepository.findByEmail("mike@somedomain.com",
                Projection.UsernameOnly.class);
        List<User> users = userRepository.findByEmail("mike@somedomain.com", #F
            User.class); #F

        assertEquals(1, usernames.size()); #G
        assertEquals("mike", usernames.get(0).getUsername()); #G
        assertEquals(1, users.size()); #G
        assertEquals("mike", users.get(0).getUsername()); #G
    }
}
```

```

#A The findByEmail method will return a list of Projection.UsernameOnly instances.
#B We verify assertions.
#C The findByRegistrationDateAfter method will return a list of Projection.UserSummary instances.
#D We verify assertions.
#E This findByEmail method provides a dynamic projection. It may return a list of Projection.UsernameOnly
instances.
#F This findByEmail method may also return a list of User instances, depending on the class that it is generified
by.
#G We then verify assertions.

```

4.9 Modifying queries

The programmer can define modifying methods with the `@Modifying` annotation. For example, INSERT, UPDATE or DELETE queries or DDL statements modify the content of the database. The `@Query` annotation will have as an argument the modifying query and may need binding parameters. Such a method must also be annotated with `@Transactional` or run from a programmatically managed transaction. Modifying queries have the advantage of clearly emphasizing which column they address and they may include conditions, so they can make the code clearer from this point of view compared with persisting/deleting the whole object. Also, changing a limited number of columns in the database will execute quicker.

Spring Data JPA can also generate delete queries based on method names. The mechanism works by following the examples from table 4.1 and replacing the `find` keyword with `delete`.

We'll add to the `UserRepository` interface the following methods:

Listing 4.19 Adding modifying methods to the UserRepository interface

```

Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence/springdatajpa/repositories/UserRepository.java
@Modifying #A
@Transactional #A
@Query("update User u set u.level = ?2 where u.level = ?1") #A
int updateLevel(int oldLevel, int newLevel); #A

@Transactional #B
int deleteByLevel(int level); #B

@Transactional #C
@Modifying #C
@Query("delete from User u where u.level = ?1") #C
int deleteBulkByLevel(int level); #C

```

```

#A The updateLevel method will change the level of the users with the oldLevel parameter and set it to the
newLevel, as the argument of the @Query annotation indicates. The method is also annotated with
@Modifying and @Transactional.
#B The deleteByLevel method will generate a query based on the method name that will remove all users with
the level given as a parameter. The method is annotated with @Transactional.
#C The deleteBulkByLevel method will remove all users with the level given as a parameter, as the argument
of the @Query annotation indicates. The method is also annotated with @Modifying and @Transactional.
Which is the difference between the deleteByLevel method and the deleteBulkByLevel method? The
first one runs a query, then will remove the returned instances one by one. If there are callback methods to

```

control the lifecycle of each instance (for example, a method to be run when a user is removed), they will be executed. The second method will remove the users in bulk, executing a single JPQL query. No `User` instance (not even the ones that are already loaded in the memory) will execute lifecycle callback methods.

We'll write the tests for these modifying methods.

Listing 4.20 Testing modifying methods

Path:

```
Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/ModifyQueryTest.java

@Test
void testModifyLevel() {
    int updated = userRepository.updateLevel(5, 4);
    List<User> users = userRepository.findByLevel(4, Sort.by("username"));

    assertAll(
        () -> assertEquals(1, updated),
        () -> assertEquals(3, users.size()),
        () -> assertEquals("katie", users.get(1).getUsername())
    );
}

@Test
void testDeleteByLevel() {
    int deleted = userRepository.deleteByLevel(2);
    List<User> users = userRepository.findByLevel(2, Sort.by("username"));
    assertEquals(0, users.size());
}

@Test
void testDeleteBulkByLevel() {
    int deleted = userRepository.deleteBulkByLevel(2);
    List<User> users = userRepository.findByLevel(2, Sort.by("username"));
    assertEquals(0, users.size());
}
```

4.10 Query by Example

Query by Example (QBE) is a querying technique that does not require writing classical queries to include entities and properties. It allows dynamic query creation and consists of three pieces: a probe, an `ExampleMatcher`, and an `Example`.

A probe is a domain object with already set properties. The `ExampleMatcher` provides the rules about matching particular properties. An `Example` puts head to head the probe and the `ExampleMatcher` and generates the query. Multiple `Examples` may reuse one single `ExampleMatcher`.

The most appropriate use cases for QBE are:

- Decoupling the work on the code from the underlying data store API
- Frequent changes of the internal structure of the domain objects without propagating them to the existing queries
- Building a set of static or dynamic constraints to query the repository

Limitations of QBE include:

- Supports only starting/ending/containing/regex matching for String properties and only exact matching for other types
- Does not support nested or grouped property constraints, such as `username = ?0` or `(username = ?1 and email = ?2)`

We'll no longer add any method to the `UserRepository` interface and will write only tests to build the probe, the `ExampleMatchers`, and the `Examples`.

Listing 4.21 Query By Example tests

```
Path: Ch04/springdatajpa2/src/test/java/com/manning/javapersistence/springdatajpa/QueryByExampleTest.java
public class QueryByExampleTest extends SpringDataJpaApplicationTests {

    @Test
    void testEmailWithQueryByExample() {
        User user = new User();                                #A
        user.setEmail("@someotherdomain.com");                #A

        ExampleMatcher matcher = ExampleMatcher.matching()      #B
            .withIgnorePaths("level", "active")                 #B
            .withMatcher("email", match -> match.endsWith());   #B

        Example<User> example = Example.of(user, matcher);     #C
        List<User> users = userRepository.findAll(example);      #D
        assertEquals(4, users.size());                          #E

    }

    @Test
    void testUsernameWithQueryByExample() {
        User user = new User();                                #F
        user.setUsername("J");                                #F

        ExampleMatcher matcher = ExampleMatcher.matching()      #G
            .withIgnorePaths("level", "active")                 #G
            .withStringMatcher(ExampleMatcher.StringMatcher.STARTING) #G
            .withIgnoreCase();                                #G

        Example<User> example = Example.of(user, matcher);     #H
        List<User> users = userRepository.findAll(example);      #I
        assertEquals(3, users.size());                          #J
    }
}
```

#A We initialize a `User` instance and setup an `email` for it. This will represent the probe.

#B We create the `ExampleMatcher` with the help of the builder pattern. Any `null` reference property will be ignored by the matcher. However, we need to explicitly ignore the `level` and `active` properties that are primitives. If they were not ignored, they would be included in the matcher with their default values (0 for `level` and `false` for `active`) and would change the generated query. We configure the matcher condition so that the `email` property will end with a given string.

#C We create an Example that puts head to head the probe and the ExampleMatcher and generates the query.
The query will search for the users having the email property ending with the string defining the email of the probe.

#D We execute the query to find all users matching the probe.

#E We verify that there are 4 users of this kind #E.

#F We initialize a User instance and setup a name for it. This will represent the second probe.

#G We create the ExampleMatcher with the help of the builder pattern. Any null reference property will be ignored by the matcher. However, we need to explicitly ignore the level and active properties that are primitives. If they were not ignored, they would be included in the matcher with their default values (0 for level and false for active) and would change the generated query. We configure the matcher condition so that the match will be made on starting strings for the configured properties (the username property from the probe in our case).

#H We create an Example that puts head to head the probe and the ExampleMatcher and generates the query.
The query will search for the users having the username property, starting with the string defining the username of the probe.

#I We execute the query to find all users matching the probe.

#J We verify that there are 6 users of this kind.

To clearly emphasize the importance of ignoring the default primitive properties, we compare the generated queries with and without the call to the withIgnorePaths("level", "active") methods.

For the first test, the query generated with the call to the withIgnorePaths("level", "active") method:

```
select user0_.id as id1_0_, user0_.active as active2_0_, user0_.email as email3_0_,  

       user0_.level as level4_0_, user0_.registration_date as regista5_0_, user0_.username  

       as username6_0_ from users user0_ where user0_.email like ? escape ?
```

For the first test, the query generated without the call to the withIgnorePaths("level", "active") method:

```
select user0_.id as id1_0_, user0_.active as active2_0_, user0_.email as email3_0_,  

       user0_.level as level4_0_, user0_.registration_date as regista5_0_, user0_.username  

       as username6_0_ from users user0_ where user0_.active=? and (user0_.email like ?  

       escape ?) and user0_.level=0
```

For the second test, the query generated with the call to the withIgnorePaths("level", "active") method:

```
select user0_.id as id1_0_, user0_.active as active2_0_, user0_.email as email3_0_,  

       user0_.level as level4_0_, user0_.registration_date as regista5_0_, user0_.username  

       as username6_0_ from users user0_ where lower(user0_.username) like ? escape ?
```

For the second test, the query generated without the call to the withIgnorePaths("level", "active") method:

```
select user0_.id as id1_0_, user0_.active as active2_0_, user0_.email as email3_0_,  

       user0_.level as level4_0_, user0_.registration_date as regista5_0_, user0_.username  

       as username6_0_ from users user0_ where user0_.active=? and user0_.level=0 and  

       (lower(user0_.username) like ? escape ?)
```

We notice the additionally introduced conditions on primitive properties when the withIgnorePaths("level", "active") method was removed:

```
user0_.active=? and user0_.level=0
```

and this will change the query result.

4.11 Summary

- This chapter examined the Spring Data JPA project and its main features.
- We created and configured a Spring Data JPA project using Spring Boot.
- We defined and used a series of query methods to access repositories by following the Spring Data JPA query builder mechanisms.
- We demonstrated the Spring Data JPA capabilities of limiting query results, sorting, paging, and streaming the results.
- We operated with the `@Query` annotation to define both non-native and native custom queries.
- We implemented projections to shape the return types and selectively return attributes of the entities.
- We created and used modifying queries to update and delete entities.
- We demonstrated the work with the Query by Example (QBE) querying technique and analyzed its use cases and its limitations.

5

Mapping persistent classes

This chapter covers

- Understanding entities and value type concepts
- Mapping entity classes with identity
- Controlling entity-level mapping options

This chapter presents some fundamental mapping options and explains how to map entity classes to SQL tables. This is essential knowledge to start with the structure of the classes in the application, no matter if you work with Hibernate, with Spring Data JPA, or some other persistence framework implementing the JPA specification. We demonstrate and analyze how you can handle database identity and primary keys and how you can use various other metadata settings to customize how Hibernate and Spring Data JPA using Hibernate as a persistence provider load and store instances of your domain model classes. Hibernate is a JPA implementation, while Spring Data JPA, as data access abstraction, comes on top of a JPA provider (Hibernate, for example) and will significantly reduce the boilerplate code to interact with the database. That is why, once the mapping of persistent classes is made, it may be used both from Hibernate and from Spring Data JPA. Our demonstrations emphasize these things.

All mapping examples use JPA annotations. First, though, we define the essential distinction between entities and value types and explain how you should approach the object/relational mapping of your domain model.

The role of the engineer will be to make the connection between the application domain, meaning the environment of the real problem that the system needs to address, and the solution domain, meaning the software and the technologies that will build this system. In figure 5.1, the application model is represented through the application model domain (the real entities), while the solution domain is represented by the system model (the objects from the software application).

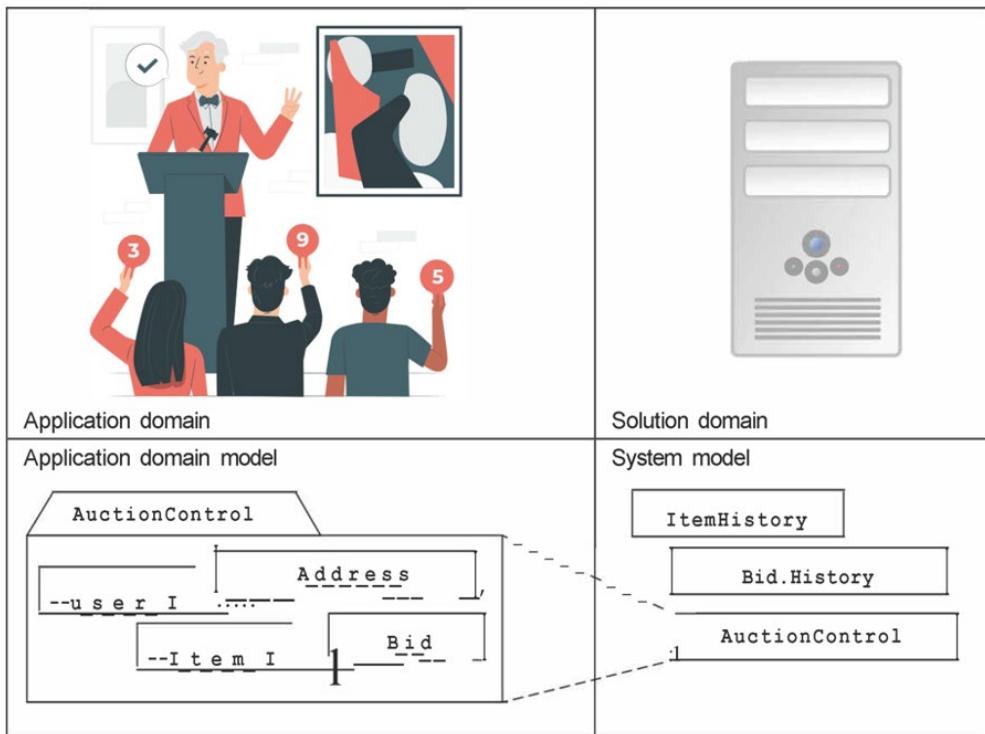


Figure 5.1 The different domains and models to be connected

5.1 Understanding entities and value types

When you look at your domain model, you'll notice a difference between classes: some of the types seem more important, representing first-class business objects (the term *object* is used here in its natural sense). Examples are the `Item`, `Category`, and `User` classes: these are entities in the real world you're trying to represent (refer back to figure 3.3 for a view of the example domain model). Other types present in your domain model, such as `Address`, seem less important. In this section, we look at what it means to use fine-grained domain models and making the distinction between entity and value types.

5.1.1 Fine-grained domain models

A major objective of Hibernate and Spring Data JPA using Hibernate as a persistence provider is support for fine-grained and rich domain models. It's one reason we work with POJOs (the acronym for Plain Old Java Object) – regular Java objects, not bound to any framework. In crude terms, *fine-grained* means more classes than tables.

For example, a user may have a home address in your domain model. In the database, you may have a single `USERS` table with the columns `HOME_STREET`, `HOME_CITY`, and `HOME_ZIPCODE`. (Remember the problem of SQL types we discussed in section 1.2.1?)

In the domain model, you could use the same approach, representing the address as three string-valued properties of the `User` class. But it's much better to model this using an `Address` class, where `User` has a `homeAddress` property. This domain model achieves improved cohesion and greater code reuse, and it's more understandable than SQL with inflexible type systems.

JPA emphasizes the usefulness of fine-grained classes for implementing type safety and behavior. For example, many people model an email address as a string-valued property of `User`. A more sophisticated approach is to define an `EmailAddress` class, which adds higher-level semantics and behavior—it may provide a `prepareMail()` method (it shouldn't have a `sendMail()` method because you don't want your domain model classes to depend on the mail subsystem).

This granularity problem leads us to a distinction of central importance in ORM. In Java, all classes are of equal standing—all instances have their own identity and life cycle. When you introduce persistence, some instances may not have their own identity and life cycle but depend on others. Let's walk through an example.

5.1.2 Defining application concepts

Two people live in the same house, and they both register user accounts in `Caveat-Emptor`. Let's call them John and Jane.

An instance of `User` represents each account. Because you want to load, save, and delete these `User` instances independently, `User` is an entity class and not a value type. Finding entity classes is easy.

The `User` class has a `homeAddress` property; it's an association with the `Address` class. Do both `User` instances have a runtime reference to the same `Address` instance, or does each `User` instance have a reference to its own `Address`? Does it matter that John and Jane live in the same house?

In figure 5.2, you can see how two `User` instances share a single `Address` instance representing their home address (this is a UML object diagram, not a class diagram). If `Address` is supposed to support shared runtime references, it's an entity type. The `Address` instance has its own life. You can't delete it when John removes his `User` account—Jane might still have a reference to the `Address`.

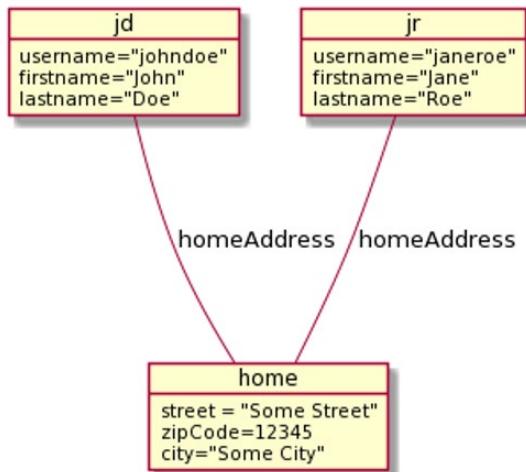


Figure 5.2 Two `User` instances have a reference to a single `Address`.

Now let's look at the alternative model where each `User` has a reference to its own `homeAddress` instance, as shown in figure 5.3. In this case, you can make an instance of `Address` dependent on an instance of `User`: you make it a value type. When John removes his `User` account, you can safely delete his `Address` instance. Nobody else will hold a reference.

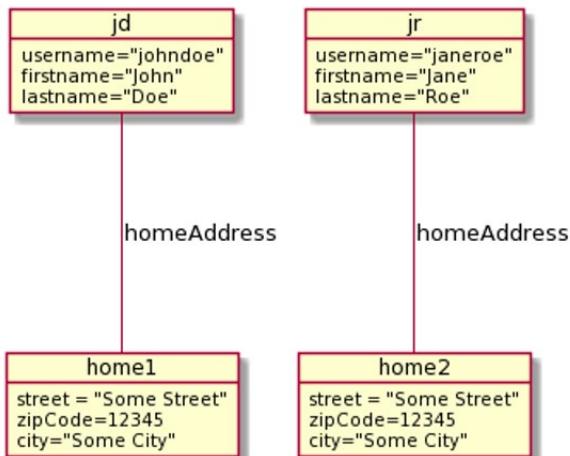


Figure 5.3 Two `User` instances each have their own dependent `Address`.

Hence, we make the following essential distinction:

- You can retrieve an instance of *entity type* using its persistent identity: for example, a `User`, `Item`, or `Category` instance. A reference to an entity instance (a pointer in the JVM) is persisted as a reference in the database (a foreign key-constrained value). An entity instance has its own life cycle; it may exist independently of any other entity. You map selected classes of your domain model as entity types.
- An instance of *value type* has no persistent identifier property; it belongs to an entity instance. Its lifespan is bound to the owning entity instance. A value-type instance doesn't support shared references. You can map your own domain model classes as value types: for example, `Address` and `MonetaryAmount`.

If you read the JPA specification, you'll find the same concept. But value types in JPA are called *basic property types* or *embeddable classes*. We come back to this in the next chapter; first, our focus is on entities.

Identifying entities and value types in your domain model isn't an ad hoc task but follows a certain procedure.

5.1.3 Distinguishing entities and value types

You may find it helpful to add stereotype (a UML extensibility mechanism) information to your UML class diagrams so you can immediately recognize entities and value types. This practice also forces you to think about this distinction for all your classes, which is the first step to an optimal mapping and well-performing persistence layer. Figure 5.4 shows an example.

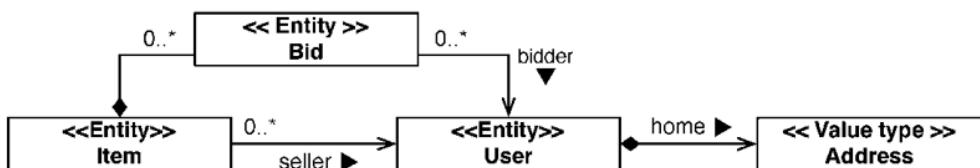


Figure 5.4 Diagramming stereotypes for entities and value types

The `Item` and `User` classes are obvious entities. They each have their own identity, their instances have references from many other instances (shared references), and they have independent lifespans.

Marking the `Address` as a value type is also easy: a single `User` instance references a particular `Address` instance. You know this because the association has been created as a composition, where the `User` instance has been made fully responsible for the life cycle of the referenced `Address` instance. Therefore, `Address` instances can't be referenced by anyone else and don't need their own identity.

The `Bid` class could be a problem. In object-oriented modeling, this is marked as a composition (the association between `Item` and `Bid` with the full diamond). Composition is a

type of association where an object can only exist as part of a container. If the container is destroyed, then the included object is also destroyed. Thus, an `Item` is the owner of its `Bid` instances and holds a collection of references. The `Bid` instances cannot exist in the absence of the `Item`. At first, this seems reasonable because bids in an auction system are useless when the item they were made for disappears.

But what if a future extension of the domain model requires a `User#bids` collection containing all bids made by a particular `User`? Right now, the association between `Bid` and `User` is unidirectional; a `Bid` has a `bidder` reference. What if this was bidirectional?

In that case, you have to deal with possible shared references to `Bid` instances, so the `Bid` class needs to be an entity. It has a dependent life cycle, but it must have its own identity to support (future) shared references.

You'll often find this kind of mixed behavior, but your first reaction should be to make everything a value-typed class and promote it to an entity only when absolutely necessary. `Bid` is a value type as its identity is defined by `Item` and `User`. This does not necessarily mean that it may not live in its own table. Try to simplify your associations: persistent collections, for example, frequently add complexity without offering any advantages. Instead of mapping `Item#bids` and `User#bids` collections, you can write queries to obtain all the bids for an `Item` and those made by a particular `User`. The associations in the UML diagram would point from the `Bid` to the `Item` and `User`, unidirectionally, and not the other way. The stereotype on the `Bid` class would then be `<<Value type>>`. We come back to this point again in chapter 8.

Next, take your domain model diagram and implement POJOs for all entities and value types. You'll have to take care of three things:

- *Shared references*—Avoid shared references to value type instances when you write your POJO classes. For example, make sure only one `User` can reference an `Address`. You can make `Address` immutable with no public `setUser()` method and enforce the relationship with a public constructor that has a `User` argument. Of course, you still need a no-argument, probably protected constructor, as we discussed in chapter 3, so Hibernate or Spring Data JPA can also create an instance.
- *Life cycle dependencies*—If a `User` is deleted, its `Address` dependency has to be deleted as well. Persistence metadata will include the cascading rules for all such dependencies, so Hibernate or Spring Data JPA or the database can take care of removing the obsolete `Address`. You must design your application procedures and user interface to respect and expect such dependencies—write your domain model POJOs accordingly.
- *Identity*—Entity classes need an identifier property in almost all cases. Value type classes (and, of course, JDK classes such as `String` and `Integer`) don't have an identifier property because instances are identified through the owning entity.

We come back to references, associations, and life cycle rules when we discuss more advanced mappings throughout later chapters in this book. Object identity and identifier properties are our next topic.

5.2 Mapping entities with identity

Mapping entities with identity requires you to understand Java identity and equality before we can walk through an entity class example and its mapping. After that, we'll be able to dig in deeper and select a primary key, configure key generators, and finally go through identifier generator strategies. First, it's vital to understand the difference between Java object identity and object equality before we discuss terms like *database identity* and the way JPA manages identity.

5.2.1 Understanding Java identity and equality

Java developers understand the difference between Java object identity and equality. Object identity (`==`) is a notion defined by the Java virtual machine. Two references are identical if they point to the same memory location.

On the other hand, object equality is a notion defined by a class's `equals()` method, sometimes also referred to as *equivalence*. Equivalence means two different (non-identical) instances have the same value—the same state. If you have a stack of freshly printed books of the same kind and you will have to choose one of them, it means you will have to choose one from non-identical but equivalent objects.

Two different instances of `String` are equal if they represent the same sequence of characters, even though each has its own location in the memory space of the virtual machine. (If you're a Java guru, we acknowledge that `String` is a special case. Assume we used a different class to make the same point.)

Persistence complicates this picture. With object/relational persistence, a persistent instance is an in-memory representation of a particular row (or rows) of a database table (or tables). Along with Java identity and equality, we define database identity. You now have three methods for distinguishing references:

- Objects are identical if they occupy the same memory location in the JVM. This can be checked with the `a == b` operator. This concept is known as *object identity*.
- Objects are equal if they have the same state, as defined by the `a.equals(Object b)` method. Classes that don't explicitly override this method inherit the implementation defined by `java.lang.Object`, which compares object identity with `==`. This concept is known as *object equality*.
- Objects stored in a relational database are identical if they share the same table and primary key value. This concept, mapped into the Java space, is known as *database identity*.

We now need to look at how database identity relates to object identity and how to express database identity in the mapping metadata. As an example, you'll map an entity of a domain model.

5.2.2 A first entity class and mapping

The `@Entity` annotation isn't enough to map a persistent class. You also need an `@Id` annotation, as shown in the following listing (see the `generator` folder for the source code).

To be able to execute the examples from the source code, you need first to run the Ch05.sql script.

Listing 5.1 Mapped Item entity class with an identifier property

```
Path: Ch05/generator/src/main/java/com/manning/javapersistence/ch05/model/Item.java
@Entity
public class Item {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;
    public Long getId() {
        return id;
    }
}
```

This is the most basic entity class, marked as “persistence capable” with the `@Entity` annotation and with an `@Id` mapping for the database identifier property—the class maps by default to a table named `ITEM` in the database schema.

Every entity class has to have an `@Id` property; it’s how JPA exposes database identity to the application. We don’t show the identifier property in our diagrams; we assume that each entity class has one. In our examples, we always name the identifier property `id`. This is a good practice for your own project; use the same identifier property name for all your domain model entity classes. If you specify nothing else, this property maps to a primary key column named `ID` of the `ITEM` table in your database schema.

Hibernate and Spring Data JPA will use the field to access the identifier property value when loading and storing items, not getter or setter methods. Because `@Id` is on a field, Hibernate or Spring Data JPA will now enable every field of the class as a persistent property by default. The rule in JPA is this: if `@Id` is on a field, the JPA provider will access fields of the class directly and consider all fields as part of the persistent state by default. In our experience, field access is often the best choice because it gives you more freedom for accessor method design.

Should you have a (public) getter method for the identifier property? Well, the application often uses database identifiers as a convenient handle to a particular instance, even outside the persistence layer. For example, it’s common for web applications to display the results of a search screen to the user as a list of summaries. When the user selects a particular element, the application may need to retrieve the selected item, and it’s common to use a lookup by identifier for this purpose—you’ve probably already used identifiers this way, even in applications that rely on JDBC.

Should you have a setter method? Primary key values never change, so you shouldn’t allow modification of the identifier property value. Hibernate and Spring Data JPA having Hibernate as provider won’t update a primary key column, and you shouldn’t expose a public identifier setter method on an entity.

The Java type of the identifier property, `java.lang.Long` in the previous example, depends on the primary key column type of the `ITEM` table and how key values are produced. This brings us to the `@GeneratedValue` annotation and primary keys in general.

5.2.3 Selecting a primary key

The database identifier of an entity is mapped to some table primary key, so let's first get some background on primary keys without worrying about mappings. Take a step back and think about how you identify entities.

A *candidate key* is a column or set of columns that you could use to identify a particular row in a table. To become the primary key, a candidate key must satisfy the following requirements:

- The value of any candidate key column is never null. You can't identify something with data that is unknown, and there are no nulls in the relational model. Some SQL products allow defining (composite) primary keys with nullable columns, so you must be careful.
- The value of the candidate key column(s) is a unique value for any row.
- The value of the candidate key column(s) never changes; it's immutable.

Must primary keys be immutable?

The relational model defines that a candidate key must be unique and irreducible (no subset of the key attributes has the uniqueness property). Beyond that, picking a candidate key as *the* primary key is a matter of taste. But Hibernate and Spring Data JPA expect a candidate key to be immutable when used as the primary key. Hibernate and Spring Data JPA having Hibernate as provider don't support updating primary key values with an API; if you try to work around this requirement, you'll run into problems with Hibernate's caching and dirty-checking engine. If your database schema relies on updatable primary keys (and maybe uses `ON UPDATE CASCADE` foreign key constraints), you must change the schema before it works with Hibernate or Spring Data JPA having Hibernate as provider.

If a table has only one identifying attribute, it becomes, by definition, the primary key. But several columns or combinations of columns may satisfy these properties for a particular table; you choose between candidate keys to decide the best primary key for the table. You should declare candidate keys not chosen as the primary key as unique keys in the database if their value is indeed unique (but maybe not immutable).

Many legacy SQL data models use natural primary keys. A *natural key* is a key with business meaning: an attribute or combination of attributes that is unique by virtue of its business semantics. Examples of natural keys are the US Social Security Number and Australian Tax File Number. Distinguishing natural keys is simple: if a candidate key attribute has meaning outside the database context, it's a natural key, regardless of whether it's automatically generated. Think about the application users: if they refer to a key attribute when talking about and working with the application, it's a natural key: "Can you send me the pictures of item #A23-abc?"

Experience has shown that natural primary keys usually cause problems in the end. A good primary key must be unique, immutable, and never null. Few entity attributes satisfy these requirements, and some that do can't be efficiently indexed by SQL databases (although this is an implementation detail and shouldn't be the deciding factor for or against a particular key). Besides, you should make certain that a candidate key definition never changes throughout the lifetime of the database. Changing the value (or even definition) of a

primary key and all foreign keys that refer to it is a frustrating task. Expect your database schema to survive decades, even if your application won't.

Furthermore, you can often only find natural candidate keys by combining several columns in a *composite* natural key. These composite keys, although certainly appropriate for some schema artifacts (like a link table in a many-to-many relationship), potentially make maintenance, ad hoc queries, and schema evolution much more difficult.

For these reasons, we strongly recommend that you add synthetic identifiers, also called *surrogate keys*. Surrogate keys have no business meaning—they have unique values generated by the database or application. Application users ideally don't see or refer to these key values; they're part of the system internals. Introducing a surrogate key column is also appropriate in the common situation when there are no candidate keys. In other words, (almost) every table in your schema should have a dedicated surrogate primary key column with only this purpose.

There are several well-known approaches to generating surrogate key values. The aforementioned `@GeneratedValue` annotation is how you configure this.

5.2.4 Configuring key generators

The `@Id` annotation is required to mark the identifier property of an entity class. Without the `@GeneratedValue` next to it, the JPA provider assumes that you'll take care of creating and assigning an identifier value before you save an instance. We call this an *application-assigned* identifier. Assigning an entity identifier manually is necessary when you're dealing with a legacy database and/or natural primary keys.

Usually, you want the system to generate a primary key value when you save an entity instance, so you write the `@GeneratedValue` annotation next to `@Id`. JPA standardizes several value-generation strategies with the `javax.persistence.GenerationType` enum, which you select with `@GeneratedValue(strategy = ...)`:

- `GenerationType.AUTO`—Hibernate (or Spring Data JPA using Hibernate as a persistence provider) picks an appropriate strategy, asking the SQL dialect of your configured database what is best. This is equivalent to `@GeneratedValue()` without any settings.
- `GenerationType.SEQUENCE`—Hibernate (or Spring Data JPA using Hibernate as a persistence provider) expects (and creates, if you use the tools) a sequence named `HIBERNATE_SEQUENCE` in your database. The sequence will be called separately before every `INSERT`, producing sequential numeric values.
- `GenerationType.IDENTITY`—Hibernate (or Spring Data JPA using Hibernate as a persistence provider) expects (and creates in table DDL) a special auto-incremented primary key column that automatically generates a numeric value on `INSERT` in the database.
- `GenerationType.TABLE`—Hibernate (or Spring Data JPA using Hibernate as a persistence provider) will use an extra table in your database schema that holds the next numeric primary key value, one row for each entity class. This table will be read and updated accordingly before `INSERTS`. The default table name is `HIBERNATE_SEQUENCES` with columns `SEQUENCE_NAME` and `NEXT_VALUE`.

Although `AUTO` seems convenient, you need more control, so you usually shouldn't rely on it and explicitly configure a primary key generation strategy. Besides, most applications work with database sequences, but you may want to customize the name and other settings of the database sequence. Therefore, instead of picking one of the JPA strategies, you may map the identifier with `@GeneratedValue(generator = "ID_GENERATOR")`, as shown in example 5.1.

This is a *named* identifier generator; you are now free to set up the `ID_GENERATOR` configuration independently from your entity classes.

JPA has two built-in annotations you can use to configure named generators: `@javax.persistence.SequenceGenerator` and `@javax.persistence.TableGenerator`. With these annotations, you can create a named generator with your own sequence and table names. As usual with JPA annotations, you can unfortunately only use them at the top of a (maybe otherwise empty) class and not in a package-info.java file.

For this reason, and because the JPA annotations don't give us access to the full Hibernate feature set, we prefer the native `@org.hibernate.annotations.GenericGenerator` annotation as an alternative. It supports all Hibernate identifier generator strategies and their configuration details. Unlike the rather limited JPA annotations, you can use the Hibernate annotation in a package-info.java file, typically in the same package as your domain model classes. The next listing shows a recommended configuration, also to be found in the `generator` folder.

Listing 5.2 Hibernate identifier generator configured as package-level metadata

Path: Ch05/generator/src/main/java/com/manning/javapersistence/ch05/package-info.java

```
@org.hibernate.annotations.GenericGenerator(
    name = "ID_GENERATOR",
    strategy = "enhanced-sequence",
    parameters = {
        @org.hibernate.annotations.Parameter(
            name = "sequence_name",
            value = "JPWHSD_SEQUENCE"
        ),
        @org.hibernate.annotations.Parameter(
            name = "initial_value",
            value = "1000"
        )
})
```

#A The `enhanced-sequence` strategy produces sequential numeric values. If your SQL dialect supports sequences, Hibernate (or Spring Data JPA using Hibernate as a persistence provider) will use an actual database sequence. If your DBMS doesn't support native sequences, Hibernate (or Spring Data JPA using Hibernate as a persistence provider) will manage and use an extra "sequence table", simulating the behavior of a sequence. This gives you real portability: the generator can always be called before performing an SQL `INSERT`, unlike, for example, auto-increment identity columns, which produce a value on `INSERT` that has to be returned to the application afterward.

#B You can configure the `sequence_name`. Hibernate (or Spring Data JPA using Hibernate as a persistence provider) will either use an existing sequence or create it when you generate the SQL schema automatically. If your DBMS doesn't support sequences, this will be the special "sequence table" name.

#C You can start with an `initial_value` that gives you room for test data. For example, when your integration test runs, Hibernate (or Spring Data JPA using Hibernate as a persistence provider) will make any new data insertions from test code with identifier values greater than 1000. Any test data you want to import before the test can use numbers 1 to 999, and you can refer to the stable identifier values in your tests: "Load item with id 123 and run some tests on it." This is applied when Hibernate (or Spring Data JPA using Hibernate as a persistence provider) generates the SQL schema and sequence; it's a DDL option.

You can share the same database sequence among all your domain model classes. There is no harm in specifying `@GeneratedValue(generator = "ID_GENERATOR")` in all your entity classes. It doesn't matter if primary key values aren't contiguous for a particular entity, as long as they're unique within one table.

Finally, you use `java.lang.Long` as the type of the identifier property in the entity class, which maps perfectly to a numeric database sequence generator. You could also use a `long` primitive. The main difference is what `someItem.getId()` returns on a new item that hasn't been stored in the database: either `null` or `0`. If you want to test whether an item is new, a `null` check is probably easier to understand for someone else reading your code. You shouldn't use another integral type, such as `int` or `short`, for identifiers. Although they will work for a while (perhaps even years), as your database size grows, you may be limited by their range. An `Integer` would work for almost two months if you generated a new identifier each millisecond with no gaps, and a `Long` would last for about 300 million years.

Although recommended for most applications, the `enhanced-sequence` strategy, as shown in listing 5.2, is just one of the strategies built into Hibernate.

The key generators configuration is unaware of the framework that uses it. The programmer will never manage the value of the primary key. This is done at the level of the frameworks. The code will look like in listings 5.3 and 5.4.

Listing 5.3 Persisting an Item with a generated primary key from Hibernate JPA

```
Path: Ch05/generator/src/test/java/com/manning/javapersistence/ch05/HelloWorldJPATest.java
em.getTransaction().begin();

Item item = new Item();
item.setName("Some Item");
em.persist(item);

em.getTransaction().commit();
```

Listing 5.4 Persisting an Item with a generated primary key from Spring Data JPA

```
Path:
    Ch05/generator/src/test/java/com/manning/javapersistence/ch05/HelloWorldSpringDataJPATest.java

Item item = new Item();
item.setName("Some Item");
itemRepository.save(item);
```

After running any of the Hibernate JPA or Spring Data JPA programs, a new ITEM will be inserted in the database, having the id 1000, the first one as specified by the generator (figure 5.5).

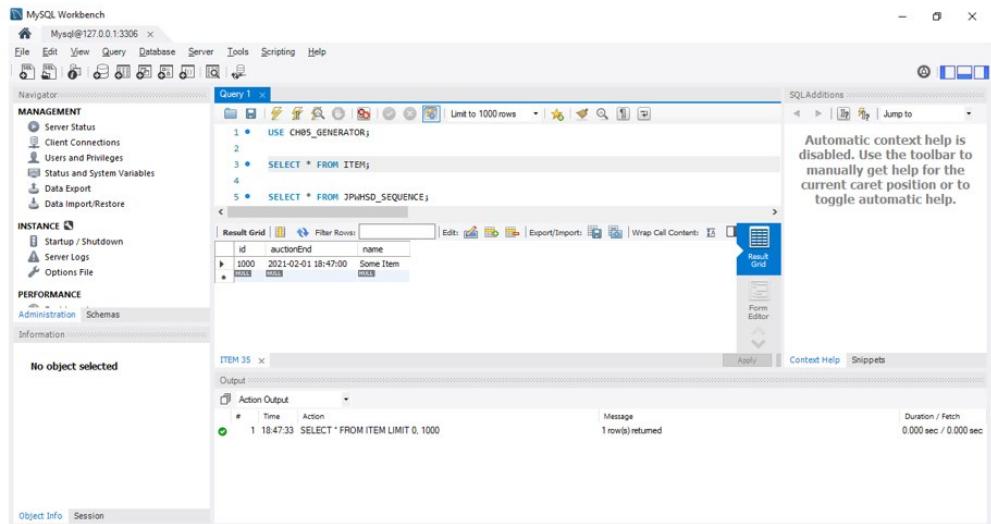


Figure 5.5 The content of the `ITEM` table after inserting a row with a generated primary key

The value to be generated for the next insertion is kept inside `JPWHSD_SEQUENCE` (figure 5.6).

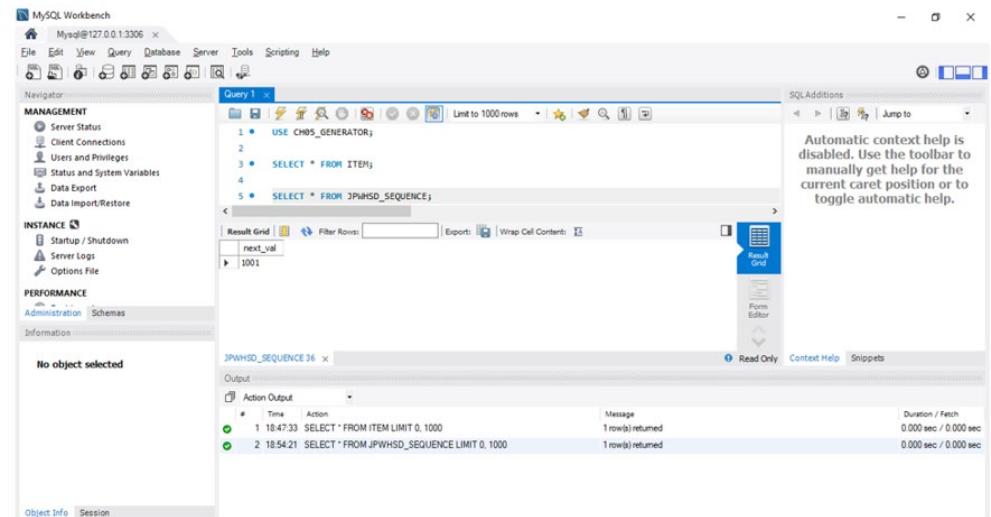


Figure 5.6 The next generated value kept by `JPWHSD_SEQUENCE`

5.2.5 Identifier generator strategies

Following is a list of all identifier generator strategies available to Hibernate and Spring Data JPA using Hibernate as provider, their options, and our usage recommendations. We do not address the deprecated generator strategies. If you don't want to read the whole list now, enable `GenerationType.AUTO` and check what Hibernate defaults to for your database dialect. It's most likely `sequence` or `identity`—a good but maybe not the most efficient or portable choice. If you require consistent, portable behavior and identifier values to be available before `INSERTS`, use `enhanced-sequence`, as shown in the previous section. This is a portable, flexible, and modern strategy, also offering various optimizers for large datasets.

Generating identifiers before or after INSERT: what's the difference

An ORM service tries to optimize SQL `INSERTS`: for example, by batching several at the JDBC level. Hence, SQL execution occurs as late as possible during a unit of work, not when you call

`entityManager.persist(someItem)`. This merely queues the insertion for later execution and, if possible, assigns the identifier value. But if you now call `someItem.getId()`, you might get `null` back if the engine wasn't able to generate an identifier before the `INSERT`. In general, we prefer pre-insert generation strategies that produce identifier values independently before `INSERT`. A common choice is a shared and concurrently accessible database sequence. Auto-incremented columns, column default values, or trigger-generated keys are only available after the `INSERT`.

Before analyzing their full list, the recommendations on identifier generator strategies are as follows:

- In general, prefer pre-insert generation strategies that produce identifier values independently before `INSERT`.
- Use `enhanced-sequence`, which uses a native database sequence when supported and otherwise falls back to an extra database table with a single column and row, emulating a sequence.

We also show the relationship between each standard JPA strategy and its native Hibernate equivalent. Hibernate has been growing organically, so there are now two sets of mappings between standard and native strategies; we call them *Old* and *New* in the list. You can switch this mapping with the `hibernate.id.new_generator_mappings` setting in your `persistence.xml` file. The default is `true`, hence the New mapping. Software doesn't age quite as well as wine:

- `native`—Automatically selects other strategies, such as `sequence` or `identity`, depending on the configured SQL dialect. You have to look at the Javadoc (or even the source) of the SQL dialect you configured in `persistence.xml`. Equivalent to JPA `GenerationType.AUTO` with the Old mapping.

- `sequence`—Uses a native database sequence named `HIBERNATE_SEQUENCE`. The sequence is called before each `INSERT` of a new row. You can customize the sequence name and provide additional DDL settings; see the Javadoc for the class `org.hibernate.id.SequenceGenerator`.
- `enhanced-sequence`—Uses a native database sequence when supported; otherwise falls back to an extra database table with a single column and row, emulating a sequence—defaults to name `HIBERNATE_SEQUENCE`. Always calls the database “sequence” before an `INSERT`, providing the same behavior independently of whether the DBMS supports real sequences. Supports an `org.hibernate.id.enhanced.Optimizer` to avoid hitting the database before each `INSERT`; defaults to no optimization and fetching a new value for each `INSERT`. Equivalent to JPA `GenerationType.SEQUENCE` and `GenerationType.AUTO` with the New mapping enabled, most likely your best option of the built-in strategies. For all parameters, see the Javadoc for the class `org.hibernate.id.enhanced.SequenceStyleGenerator`.
- `enhanced-table`—Uses an extra table named `HIBERNATE_SEQUENCES`, with one row by default representing the sequence, storing the next value. This value is selected and updated when an identifier value has to be generated. You can configure this generator to use multiple rows instead: one for each generator; see the Javadoc for `org.hibernate.id.enhanced.TableGenerator`. Equivalent to JPA `GenerationType.TABLE` with the New mapping enabled. Replaces the outdated but similar `org.hibernate.id.MultipleHiLoPerTableGenerator`, which is the Old mapping for JPA `GenerationType.TABLE`.
- `identity`—Supports `IDENTITY` and auto-increment columns in DB2, MySQL, MS SQL Server, and Sybase. The identifier value for the primary key column will be generated on `INSERT` of a row. It has no options. Unfortunately, due to a quirk in Hibernate’s code, you can *not* configure this strategy in `@GenericGenerator`. The DDL generation will not include the identity or auto-increment option for the primary key column. The only way to use it is with JPA `GenerationType.IDENTITY` and the Old or New mapping, making it the default for `GenerationType.IDENTITY`.
- `increment`—At Hibernate startup, reads the maximum (numeric) primary key column value of each entity’s table and increments the value by one each time a new row is inserted. Especially efficient if a non-clustered Hibernate application has exclusive access to the database - but don’t use it in any other scenario.
- `select`—Hibernate won’t generate a key value or include the primary key column in an `INSERT` statement. Hibernate expects the DBMS to assign a (default in schema or by trigger) value to the column on insertion. Hibernate then retrieves the primary key column with a `SELECT` query after insertion. Required parameter is `key`, naming the database identifier property (such as `id`) for the `SELECT`. This strategy isn’t very efficient and should only be used with old JDBC drivers that can’t return generated keys directly.
- `uuid2`—Produces a unique 128-bit UUID in the application layer. Useful when you need globally unique identifiers across databases (say, you merge data from several distinct production databases in batch runs every night into an archive). The UUID

can be encoded either as a `java.lang.String`, a `byte[16]`, or a `java.util.UUID` property in your entity class. Replaces the legacy `uuid` and `uuid.hex` strategies. You configure it with an `org.hibernate.id.UUIDGenerationStrategy`; see the Javadoc for the class `org.hibernate.id.UUIDGenerator` for more details.

- `guid`—Uses a globally unique identifier produced by the database, with an SQL function available on Oracle, Ingres, MS SQL Server, and MySQL. Hibernate calls the database function before an `INSERT`. Maps to a `java.lang.String` identifier property. If you need full control over identifier generation, configure the strategy of `@GenericGenerator` with the fully qualified name of a class that implements the `org.hibernate.id.IdentityGenerator` interface.

We assume from now on that you've added identifier properties to the entity classes of your domain model. Make sure you do not expose the identifier outside the business logic, for example through an API – this identifier has no business logic meaning and it is related only to the persistence part. After you complete the basic mapping of each entity and its identifier property, you continue to map the value-typed properties of the entities. We talk about value-type mappings in the next chapter. Read on for some special options that can simplify and enhance your class mappings.

5.3 Entity-mapping options

You've now mapped a persistent class with `@Entity`, using defaults for all other settings, such as the mapped SQL table name. The following section explores some class-level options and how you control them:

- Naming defaults and strategies
- Dynamic SQL generation
- Entity mutability

These are options; you can skip this section and come back later when you have to deal with a specific problem.

5.3.1 Controlling names

Let's first talk about the naming of entity classes and tables. If you only specify `@Entity` on the persistence-capable class, the default mapped table name is the same as the class name. Note that we write SQL artifact names in `UPPERCASE` to make them easier to distinguish—SQL is actually case insensitive. So the Java entity class `Item` maps to the `ITEM` table. A Java entity class `BidItem` will map to the `BID_ITEM` table (here, the camel case will be converted into a snake case). You can override the table name with the JPA `@Table` annotation, as shown next (see the `mapping` folder for the source code).

Listing 5.5 @Table annotation overrides the mapped table name

```
Path: Ch05/mapping/src/main/java/com/manning/javapersistence/ch05/model/User.java
@Entity
@Table(name = "USERS")
public class User {
    // ...
}
```

The `User` entity would map to the `USER` table; this is a reserved keyword in most SQL DBMSs. You can't have a table with that name, so you instead map it to `USERS`. The `@javax.persistence.Table` annotation also has `catalog` and `schema` options if your database layout requires these as naming prefixes.

If you really have to, quoting allows you to use reserved SQL names and even work with case-sensitive names.

QUOTING SQL IDENTIFIERS

From time to time, especially in legacy databases, you'll encounter identifiers with strange characters or whitespace or wish to force case sensitivity. Or, as in the previous example, the automatic mapping of a class or property would require a table or column name that is a reserved keyword.

Hibernate and Spring Data JPA having Hibernate as provider know the reserved keywords of your DBMS through the configured database dialect. Hibernate and Spring Data JPA can automatically put quotes around such strings when generating SQL. You can enable this automatic quoting with `hibernate.auto_quote_keyword=true` in your persistence unit configuration. If you're using an older version of Hibernate, or you find that the dialect's information is incomplete, you must still apply quotes on names manually in your mappings if there is a conflict with a keyword.

If you quote a table or column name in your mapping with backticks, Hibernate always quotes this identifier in the generated SQL. This still works in latest versions of Hibernate, but JPA 2.0 standardized this functionality as *delimited identifiers* with double quotes.

This is the Hibernate-only quoting with backticks, modifying the previous example:

```
@Table(name = "`USER`")
```

To be JPA-compliant, you also have to escape the quotes in the string:

```
@Table(name = "\"USER\"")
```

Either way works fine with Hibernate and Spring Data JPA using Hibernate as provider. It knows the native quote character of your dialect and now generates SQL accordingly: `[USER]` for MS SQL Server, `'USER'` for MySQL, `"USER"` for H2, and so on.

If you have to quote *all* SQL identifiers, create an `orm.xml` file and add the setting `<delimited-identifiers/>` to its `<persistence-unit-defaults>` section, as shown in listing 3.8. Hibernate then enforces quoted identifiers everywhere.

You should consider renaming tables or columns with reserved keyword names whenever possible. Ad hoc SQL queries are difficult to write in an SQL console if you have to quote and

escape everything properly by hand. Also, you should avoid using quoted identifiers for databases that are also accessed by other means than Hibernate/JPA/Spring Data (e.g. reporting). Having to use delimiters for all identifiers in a (complex) report query is really painful.

Next, you'll see how Hibernate and Spring Data JPA having Hibernate as provider can help when you encounter organizations with strict conventions for database table and column names.

IMPLEMENTING NAMING CONVENTIONS

Hibernate provides a feature that allows you to enforce naming standards automatically. Suppose that all table names in `CaveatEmptor` should follow the pattern `CE_<table name>`. One solution is to manually specify the `@Table` annotation on all entity classes. This approach is time-consuming and easily forgotten. Instead, you can implement Hibernate's `PhysicalNamingStrategy` interface or override an existing implementation, as in the following listing.

Listing 5.6 PhysicalNamingStrategy, overriding default naming conventions

Path: Ch05/mapping/src/main/java/com/manning/javapersistence/ch05/CENamingStrategy.java

```
public class CENamingStrategy extends PhysicalNamingStrategyStandardImpl {

    @Override
    public Identifier toPhysicalTableName(Identifier name,
                                         JdbcEnvironment context) {
        return new Identifier("CE_" + name.getText(), name.isQuoted());
    }
}
```

The overridden method `toPhysicalTableName()` prepends `CE_` to all generated table names in your schema. Look at the Javadoc of the `PhysicalNamingStrategy` interface; it offers methods for custom naming of columns, sequences, and other artifacts.

You have to enable the naming strategy implementation. With Hibernate JPA, this is done in `persistence.xml`:

```
Path: Ch05/mapping/src/main/resources/META-INF/persistence.xml

<persistence-unit name="ch05.mapping">
    ...
    <properties>
        ...
        <property name="hibernate.physical_naming_strategy"
            value="com.manning.javapersistence.ch05.CENamingStrategy"/>
    </properties>
</persistence-unit>
```

With Spring Data JPA using Hibernate as a persistence provider, this is done from the `LocalContainerEntityManagerFactoryBean` configuration:

```
Path: Ch05/mapping/src/test/java/com/manning/javapersistence/ch05/configuration/SpringData
Configuration.java

properties.put("hibernate.physical_naming_strategy",
    "com.manning.javapersistence.ch05.CENamingStrategy");
```

Let's have a quick look at another related issue, the naming of entities for queries.

NAMING ENTITIES FOR QUERYING

By default, all entity names are automatically imported into the namespace of the query engine. In other words, you can use short class names without a package prefix in JPA query strings, which is convenient:

```
Path: Ch05/generator/src/test/java/com/manning/javapersistence/ch05/HelloWorldJPATest.java

List result = em.createQuery("select i from Item i")
    .getResultList();
```

This only works when you have one `Item` class in your persistence unit. If you add another `Item` class in a different package, you should rename one of them for JPA if you want to continue using the short form in queries:

```
package my.other.model;
@javax.persistence.Entity(name = "AuctionItem")
public class Item {
    // ...
}
```

The short query form is now `select i from AuctionItem i` for the `Item` class in the `my.other.model` package. Thus you resolve the naming conflict with another `Item` class in another package. Of course, you can always use fully qualified long names with the package prefix.

This completes our tour of the naming options. Next, we discuss how Hibernate and Spring Data JPA using Hibernate generate the SQL that contains these names.

5.3.2 Dynamic SQL generation

By default, Hibernate and Spring Data JPA using Hibernate as provider create SQL statements for each persistent class when the persistence unit is created on startup. These statements are simple create, read, update, and delete (CRUD) operations for reading a single row, deleting a row, and so on. It's cheaper to create and cache them instead of generating SQL strings every time such a simple query has to be executed at runtime. Besides, prepared statement caching at the JDBC level is much more efficient if there are fewer statements.

How can Hibernate create an `UPDATE` statement on startup? After all, the columns to be updated aren't known at this time. The answer is that the generated SQL statement updates all columns, and if the value of a particular column isn't modified, the statement sets it to its old value.

In some situations, such as a legacy table with hundreds of columns where the SQL statements will be large for even the simplest operations (say, only one column needs updating), you should disable this startup SQL generation and switch to dynamic statements generated at runtime. An extremely large number of entities can also impact startup time because Hibernate has to generate all SQL statements for CRUD up front. Memory consumption for this query statement cache will also be high if a dozen statements must be cached for thousands of entities. This can be an issue in virtual environments with memory limitations or on low-power devices.

To disable generation of `INSERT` and `UPDATE` SQL statements on startup, you need native Hibernate annotations:

```
@Entity
@org.hibernate.annotations.DynamicInsert
@org.hibernate.annotations.DynamicUpdate
public class Item {
    // ...
}
```

By enabling dynamic insertion and updates, you tell Hibernate to produce the SQL strings when needed, not up front. The `UPDATE` will only contain columns with updated values, and the `INSERT` will only contain non-null columns.

5.3.3 Making an entity immutable

Instances of a particular class may be immutable. For example, in `CaveatEmptor`, a `Bid` made for an item is immutable. Hence, Hibernate or Spring Data JPA having Hibernate as provider never needs to execute `UPDATE` statements on the `BID` table. Hibernate can also make a few other optimizations, such as avoiding dirty checking, if you map an immutable class, as shown in the next example. Here, the `Bid` class is immutable, and instances are never modified:

```
@Entity
@org.hibernate.annotations.Immutable
public class Bid {
    // ...
}
```

A POJO is immutable if no public setter methods for any properties of the class are exposed—all values are set in the constructor. Hibernate or Spring Data JPA having Hibernate as provider should access the fields directly when loading and storing instances. We talked about this earlier in this chapter: if the `@Id` annotation is on a field, Hibernate will access the fields directly, and you are free to design your getter and setter methods. Also, remember that not all frameworks work with POJOs without setter methods.

When you can't create a view in your database schema, you can map an immutable entity class to an SQL `SELECT` query.

5.3.4 Mapping an entity to a subselect

Sometimes your DBA won't allow you to change the database schema; even adding a new view might not be possible. Let's say you want to create a view that contains the identifier of an auction `Item` and the number of bids made for that item (see the `subselect` folder for the source code).

Using a Hibernate annotation, you can create an application-level view, a read-only entity class mapped to an SQL `SELECT`:

```
Path: Ch05/subselect/src/main/java/com/manning/javapersistence/ch05/model/ItemBidSummary.java

@Entity
@org.hibernate.annotations.Immutable
@org.hibernate.annotations.Subselect(
    value = "select i.ID as ITEMID, i.NAME as NAME, " +
        "count(b.ID) as NUMBEROFBIDS " +
        "from ITEM i left outer join BID b on i.ID = b.ITEM_ID " +
        "group by i.ID, i.NAME"
)
@org.hibernate.annotations.Synchronize({"ITEM", "BID"})
public class ItemBidSummary {
    @Id
    private Long itemId;
    private String name;
    private long numberOfBids;
    public ItemBidSummary() {
    }
    // Getter methods...
    // ...
}
```

You should list all table names referenced in your `SELECT` in the `@org.hibernate.annotations.Synchronize` annotation. The framework will then know it has to flush modifications of `Item` and `Bid` instances before it executes a query against `ItemBidSummary`. If there are in-memory modifications that haven't been yet persisted to the database but that may affect the query, Hibernate (or Spring Data JPA having Hibernate as provider) detects this and flushes the changes before executing the query. Otherwise, the result may be a stale state. As there is no `@Table` annotation on the `ItemBidSummary` class, the framework doesn't know when it must auto-flush before executing a query. The

`@org.hibernate.annotations.Synchronize` annotation indicates that the framework needs to flush the `ITEM` and `BID` tables before executing the query.

Using the read-only `ItemBidSummary` entity class from Hibernate JPA will look like this:

```
Path: Ch05/subselect/src/test/java/com/manning/javapersistence/ch05/ItemBidSummaryTest.java

Query query = em.createQuery(
        "select ibs from ItemBidSummary ibs where ibs.itemId = :id");

ItemBidSummary itemBidSummary =
        (ItemBidSummary) query.setParameter("id", 1000L).getSingleResult();
```

Using the read-only `ItemBidSummary` entity class from Spring Data JPA will first need the introduction of a new Spring Data repository:

```
Path:
    Ch05/mapping/src/main/java/com/manning/javapersistence/ch05/repositories/ItemBidSummaryRepository.java

public interface ItemBidSummaryRepository extends
        CrudRepository<ItemBidSummary, Long> {
```

The repository will be effectively used like this:

```
Path:
    Ch05/subselect/src/test/java/com/manning/javapersistence/ch05/ItemBidSummarySpringDataTest.java

Optional<ItemBidSummary> itemBidSummary =
        itemBidSummaryRepository.findById(1000L);
```

5.4 Summary

- We examined entities as coarser-grained classes of a system. Their instances have an independent life cycle and their own identity, and many other instances can reference them.
- We also examined value types, which are dependent on a particular entity class. A value type instance is bound to its owning entity instance, and only one entity instance can reference it—it has no individual identity.
- We contrasted Java identity, object equality, and database identity, and we analyzed what makes good primary keys.
- We compared what generators for primary key provide out of the box and how to use and extend this identifier system.
- We investigated some useful class mapping options, such as naming strategies and dynamic SQL generation.
- We demonstrated how to use entities, mapping options, and naming strategies both from Hibernate JPA and from Spring Data JPA.

6

Mapping value types

This chapter covers

- **Mapping basic properties**
- **Mapping embeddable components**
- **Controlling mapping between Java and SQL types**

After spending the previous chapter almost exclusively on entities and the respective class- and identity-mapping options, we now focus on value types in their various forms. Value types are frequently encountered in the classes under development. We split value types into two categories: basic value-typed classes that come with the JDK, such as `String`, `Date`, primitives, and their wrappers; and developer-defined value-typed classes, such as `Address` and `MonetaryAmount` in `CaveatEmptor`.

In this chapter, we first map persistent properties with JDK types and learn the basic mapping annotations. We'll see how to work with various aspects of properties: overriding defaults, customizing access, and generated values. We'll also see how SQL is used with derived properties and transformed column values. We'll work with basic properties, temporal properties, and mapping enumerations. We then examine custom value-typed classes and map them as embeddable components. We'll learn how classes relate to the database schema and make the classes embeddable while allowing for overriding embedded attributes. We complete embeddable components by mapping nested components. Finally, we analyze how to customize loading and storing of property values at a lower level with flexible JPA converters, a standardized extension point of every JPA provider.

Major new features in JPA 2

JPA 2.2 supports the Java 8 Date and Time API. There is no more need to use additional mapping annotations, as `@Temporal`, that needed to annotate fields of type `java.util.Date`.

6.1 Mapping basic properties

Mapping is at the heart of the ORM technique. It makes the connection between the object-oriented world and the relational world. When we map a persistent class, whether it's an entity or an embeddable type (more about these later, in section 6.2), all of its properties are considered persistent by default. The default JPA rules for properties of persistent classes are these ones:

- If the property is a primitive or a primitive wrapper, or of type `String`, `BigInteger`, `BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `byte[]`, `Byte[]`, `char[]`, or `Character[]`, it's automatically persistent. Hibernate or Spring Data JPA using Hibernate loads and stores the value of the property in a column with an appropriate SQL type and the same name as the property.
- Otherwise, if we annotate the class of the property as `@Embeddable`, or we map the property itself as `@Embedded`, the property is mapped as an embedded component of the owning class. We analyze the embedding of components later in this chapter, with the `Address` and `MonetaryAmount` embeddable classes of `CaveatEmptor`.
- Otherwise, if the type of the property is `java.io.Serializable`, its value is stored in its serialized form. This may raise compatibility problems (we could have stored the information using one class format and would like to retrieve it later using another class format) and performance problems (the serialization/deserialization operations are costly). We should always map Java classes instead of storing a series of bytes in the database. Maintaining a database with this binary information when the application is gone in a few years will mean that the classes to which the serialized version maps to are no longer available.
- Otherwise, an exception will be thrown on startup, complaining that it doesn't understand the type of the property.

This *configuration by exception* approach means we don't have to annotate a property to make it persistent; we only have to configure the mapping in an exceptional case. Several annotations are available in JPA to customize and control basic property mappings.

6.1.1 Overriding basic property defaults

We might not want all properties of an entity class to be persistent. So, which information will make sense to be persisted and which not? For example, although it makes sense to have a persistent `Item#initialPrice` property, an `Item#totalPriceIncludingTax` property shouldn't be persistent if we only compute and use its value at runtime, and hence shouldn't be stored in the database. To exclude a property, mark the field or the getter method of the

property with the annotation `@javax.persistence.Transient` or use the Java `transient` keyword. The `transient` keyword excludes fields both for Java serialization and for persistence, as it is also recognized by JPA providers. The `@javax.persistence.Transient` annotation will only exclude the field from being persisted. To decide if a property must be persistent or not, ask yourself: Is this a basic attribute that shapes the instance? Do we need it from the very beginning, or will we calculate it based on some other properties? Does it make sense to rebuild the information after some time, or will the information no longer be significant? Is it sensitive information that we would rather avoid to persist for later reveal (e.g., a password in clear)? Is it information that does not have significance in some other environment (e.g., a local IP address that is meaningless in another network)?

We'll come back to the placement of the annotation on fields or getter methods in a moment. Let's assume, as we have before, that Hibernate or Spring Data JPA using Hibernate will access fields directly because `@Id` has been placed on a field. Therefore, all other JPA and Hibernate mapping annotations are also on fields.

We are currently working on a piece of the `CaveatEmptor` application. Our goal is to take care of the persistence logic from the program but also to build flexible and easy-to-change code. To be able to execute the examples from the source code, you need first to run the `Ch06.sql` script. The source code is to be found in the `mapping-value-types` folder.

If we do not want to rely on property mapping defaults, we'll apply the `@Basic` annotation to a particular property—for example, the `initialPrice` of an `Item`:

```
@Basic(optional = false)
BigDecimal initialPrice;
```

This annotation isn't providing many alternatives. It only has two parameters: the one shown here, `optional`, marks the property as not optional at the Java object level. By default, all persistent properties are nullable and optional; an `Item` may have an unknown `initialPrice`. Mapping the `initialPrice` property as non-optional makes sense when having a `NOT NULL` constraint on the `INITIALPRICE` column in the SQL schema. The generated SQL schema will include a `NOT NULL` constraint automatically for non-optional properties.

Now, the application will store an `Item` without setting a value on the `initialPrice` field. An exception will be thrown before hitting the database with an SQL statement. Such a value is required to perform an `INSERT` or `UPDATE`. If we do not mark the property as optional and try to save a `NULL`, the database will reject the SQL statement, and a constraint-violation exception will be thrown. We'll talk about the other parameter of `@Basic`, the `fetch` option, when we explore optimization strategies later, in section 12.1.

Instead of `@Basic`, we may use the `@Column` annotation to declare nullability:

```
@Column(nullable = false)
BigDecimal initialPrice;
```

We've demonstrated three ways to declare whether a property value is required: with the `@Basic` annotation, the `@Column` annotation, and earlier with the Bean Validation `@NotNull` annotation in section 3.3.2. All have the same effect on the JPA provider: a `null` check when saving and generating a `NOT NULL` constraint in the database schema. We recommend the

Bean Validation `@NotNull` annotation so we can manually validate an `Item` instance and/or have our user interface code in the presentation layer execute validation checks automatically. There isn't much difference in the end result, but it's cleaner to avoid hitting the database with a statement that fails.

The `@Column` annotation can also override the mapping of the property name to the database column:

```
@Column(name = "START_PRICE", nullable = false)
BigDecimal initialPrice;
```

The `@Column` annotation has a few other parameters, most of which control SQL-level details such as `catalog` and `schema` names. They're rarely needed, and we only demonstrate them throughout this book when necessary.

Property annotations aren't always on fields, and we may not want the JPA provider to access fields directly.

6.1.2 Customizing property access

The persistence engine accesses the properties of a class either directly through fields or indirectly through getter and setter methods. We will try now to answer the question: how should we access each persistent property? An annotated entity inherits the default from the position of the mandatory `@Id` annotation. For example, if we declare `@Id` on a field, not a getter method, all other mapping annotations for that entity are expected on fields. Annotations are never on the setter methods, they are not supported here.

The default access strategy isn't only applicable to a single entity class. Any `@Embedded` class inherits the default or explicitly declared access strategy of its owning root entity class. We cover embedded components later in this chapter. Furthermore, any `@MappedSuperclass` properties are accessed with the default or explicitly declared access strategy of the mapped entity class. Inheritance is the topic of chapter 7.

The JPA specification offers the `@Access` annotation for overriding the default behavior, using the parameters `AccessType.FIELD` (access through fields) and `AccessType.PROPERTY` (access through getters). Setting `@Access` on the class/entity level, all properties of the class will be accessed according to the selected strategy. Any other mapping annotations, including the `@Id`, may be set on either fields or getter methods, respectively.

We can also use the `@Access` annotation to override the access strategy of individual properties, as in the following example.

Listing 6.1 Overriding access strategy for the name property

```
Path: Ch06/mapping-value-
      types/src/main/java/com/manning/javapersistence/ch06/model/Item.java
@Entity
public class Item {

    @Id
    @GeneratedValue(generator = "ID_GENERATOR")                                #A
    private Long id;                                                          #A
                                                                           #A

    @Access(AccessType.PROPERTY)                                              #B
    @Column(name = "ITEM_NAME")                                               #B
    private String name;                                                       #B

    public String getName() {                                                 #C
        return name;                                                          #C
    }                                                                           #C

    public void setName(String name) {                                         #C
        this.name =                                                        #C
            !name.startsWith("AUCTION: ") ? "AUCTION: " + name : name; #C
    }                                                                           #C

}
```

#A The Item entity defaults to field access. The @Id is on the field.

#B The @Access (AccessType.PROPERTY) setting on the name field switches this particular property to runtime access through getter/setter methods by the JPA provider.

#C Hibernate or Spring Data JPA using Hibernate calls getName() and setName() when loading and storing items.

In the listing above:

Note that the position of other mapping annotations like @Column doesn't change—only how instances are accessed at runtime.

Now turn it around: if the default (or explicit) access type of the entity were through property getter and setter methods, @Access(AccessType.FIELD) on a getter method would tell Hibernate or Spring Data JPA using Hibernate to access the field directly. All other mapping information would still have to be on the getter method, not the field.

Some properties don't map to a column. In particular, a derived property (like a calculated field) takes its value from an SQL expression.

6.1.3 Using derived properties

We arrived now to work with derived properties – the ones resulting from some other properties. We will define the value of a derived property as calculated at runtime by evaluating an SQL expression declared with the `@org.hibernate.annotations.Formula` annotation, as in the next listing.

Listing 6.2 Two read-only derived properties

```
Path: Ch06/mapping-value-
      types/src/main/java/com/manning/javapersistence/ch06/model/Item.java

@Formula(
    "CONCAT(SUBSTR(DESCRIPTION, 1, 12), '...')")
)
private String shortDescription;

@Formula(
    "(SELECT AVG(B.AMOUNT) FROM BID B WHERE B.ITEM_ID = ID)"
)
private BigDecimal averageBidAmount;
```

The given SQL formulas are evaluated every time the `Item` entity is retrieved from the database and not at any other time, so the result may be outdated if other properties are modified. The properties never appear in an SQL `INSERT` or `UPDATE`, only in `SELECTS`. Evaluation occurs in the database; the SQL formula is embedded in the `SELECT` clause when loading the instance.

Formulas may refer to columns of the database table, they can call specific database SQL functions, and they may even include SQL subselects. In the previous example, the `SUBSTR()` and `CONCAT()` functions are called. The SQL expression is passed to the underlying database as is. Relying on vendor-specific operators or keywords may bind the mapping metadata to a particular database product. For example, the `CONCAT()` function in the example above is specific to MySQL. So you should be aware that portability may be impacted. Notice that unqualified column names refer to columns of the table of the class to which the derived property belongs.

The database evaluates SQL expressions in formulas only when an entity instance is retrieved from the database. Hibernate also supports a variation of formulas called *column transformers*, allowing to write a custom SQL expression for reading and writing a property value. Let's investigate this capability.

6.1.4 Transforming column values

We'll deal now with the information having different representations in the object-oriented system and the relational system. The database has a column called `IMPERIALWEIGHT`, storing the weight of an `Item` in pounds. The application, however, has the property `Item#metricWeight` in kilograms, so we have to convert the value of the database column when reading and writing a row from and to the `ITEM` table. We can implement this with this Hibernate extension: the `@org.hibernate.annotations.ColumnTransformer` annotation.

Listing 6.3 Transforming column values with SQL expressions

```
Path: Ch06/mapping-value-
      types/src/main/java/com/manning/javapersistence/ch06/model/Item.java

@Column(name = "IMPERIALWEIGHT")
@ColumnTransformer(
    read = "IMPERIALWEIGHT / 2.20462",
    write = "? * 2.20462"
)
private double metricWeight;
```

When reading a row from the `ITEM` table, Hibernate or Spring Data JPA using Hibernate embed the expression `-IMPERIALWEIGHT / 2.20462`, so the calculation occurs in the database, and the metric value is returned in the result to the application layer. For writing to the column, Hibernate or Spring Data JPA using Hibernate set the metric value on the mandatory, single placeholder (the question mark), and the SQL expression calculates the actual value to be inserted or updated.

Hibernate also applies column converters in query restrictions. For example, the following query retrieves all items with a weight of two kilograms.

Listing 6.4 Applying column converters in query restrictions

```
Path: Ch06/mapping-value-
      types/src/test/java/com/manning/javapersistence/ch06/MappingValuesJPATest.java

List<Item> result =
    em.createQuery("SELECT i FROM Item i WHERE i.metricWeight = :w")
    .setParameter("w", 2.0)
    .getResultList();
```

The actual SQL executed for this query contains the following restriction in the `WHERE` clause:

```
// ...
where
    i.IMPERIALWEIGHT / 2.20462=?
```

Note that, most likely, the database will not be able to rely on an index for this restriction; a full table scan will be performed because the weight for *all* `ITEM` rows has to be calculated to evaluate the restriction.

Another special kind of property relies on database-generated values.

6.1.5 Generated and default property values

The database sometimes generates a property value, usually when we insert a row for the first time. Examples of database-generated values are a creation timestamp, a default price for an item, and a trigger that runs for every modification.

Typically, Hibernate or Spring Data JPA using Hibernate applications need to refresh instances that contain any properties for which the database generates values after saving. This means one would have to make another round trip to the database to read the value after inserting or updating a row. Marking properties as generated, however, lets the application delegate this responsibility to Hibernate or Spring Data JPA using Hibernate.

Essentially, whenever an SQL `INSERT` or `UPDATE` for an entity that has declared generated properties is issued, it does a `SELECT` immediately afterward to retrieve the generated values.

We use the `@org.hibernate.annotations.Generated` annotation to mark generated properties. For the temporal properties, we'll use the `@CreationTimestamp` and `@UpdateTimestamp` annotations. The `@CreationTimestamp` annotation was used to mark the `createdOn` property. This tells to Hibernate or Spring Data using Hibernate to generate the property value automatically. In this case, the value is set to the current date before the entity instance is inserted into the database. The other similar built-in annotation is `@UpdateTimestamp`, to generate the property value automatically when an entity instance is updated.

Listing 6.5 Database-generated property values

```
Path: Ch06/mapping-value-
      types/src/main/java/com/manning/javapersistence/ch06/model/Item.java

@CreationTimestamp
private LocalDate createdOn;

@UpdateTimestamp
private LocalDateTime lastModified;

@Column(insertable = false)
@ColumnDefault("1.00")
@Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
private BigDecimal initialPrice;
```

Available settings for `GenerationTime` are `ALWAYS` and `INSERT`.

With `GenerationTime.ALWAYS`, Hibernate or Spring Data JPA using Hibernate refresh the entity instance after every SQL `UPDATE` or `INSERT`. With `GenerationTime.INSERT`, refreshing only occurs after an SQL `INSERT` to retrieve the default value provided by the database. We also map the `initialPrice` property as not insertable. The `@ColumnDefault` annotation sets the default value of the column when Hibernate or Spring Data JPA using Hibernate export and generate the SQL schema DDL.

Timestamps are frequently automatically generated values, either by the database, as in the previous example, or by the application. As long as we are using JPA 2.2 and the Java 8 classes `LocalDate`, `LocalDateTime`, and `LocalTime`, we no longer need to use the `@Temporal` annotation. The enumerated Java 8 classes from the `java.time` package include the temporal precision by themselves: only the date, the date and the time, or only the time respectively. Anyway, let's have a look at the usage of this `@Temporal` annotation that you may still encounter.

6.1.6 The `@Temporal` annotation

The JPA specification allows annotating temporal properties with `@Temporal` to declare the accuracy of the SQL data type of the mapped column. The Java temporal types before Java 8

are `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`.

The next listing provides an example using the `@Temporal` annotation.

Listing 6.6 Property of a temporal type that must be annotated with `@Temporal`

```
@CreationTimestamp
@Temporal(TemporalType.DATE)
private Date createdOn;

@UpdateTimestamp
@Temporal(TemporalType.TIMESTAMP)
private Date lastModified;
```

Available `TemporalType` options are `DATE`, `TIME`, and `TIMESTAMP`, establishing what part of the temporal value should be stored in the database. The default is `TemporalType.TIMESTAMP`, when no `@Temporal` annotation is present.

Another special property type is represented by the enumerations.

6.1.7 Mapping enumerations

An *enumeration type* is a common Java idiom where a class has a constant (small) number of immutable instances. In `CaveatEmptor`, for example, we can apply this to auctions having a limited number of types:

```
Path: Ch06/mapping-value-types/
src/main/java/com/manning/javapersistence/ch06/model/AuctionType.java
public enum AuctionType {
    HIGHEST_BID,
    LOWEST_BID,
    FIXED_PRICE
}
```

We can now set the appropriate `auctionType` on each `Item`:

```
Path:
Ch06/mapping-value-types/src/main/java/com/manning/javapersistence/ch06/model/Item.java

@NotNull
@Enumerated(EnumType.STRING)
private AuctionType auctionType = AuctionType.HIGHEST_BID;
```

Without the `@Enumerated` annotation, Hibernate or Spring Data JPA using Hibernate would store the `ORDINAL` position of the value. That is, it would store 1 for `HIGHEST_BID`, 2 for `LOWEST_BID`, and 3 for `FIXED_PRICE`. This is a brittle default; making changes to the `AuctionType` enum, adding a new instance, existing values may no longer map to the same position and break the application. The `EnumType.STRING` option is, therefore, a better choice; Hibernate or Spring Data JPA using Hibernate store the label of the enum value as is.

This completes our tour of basic properties and their mapping options. So far, we have been showing properties of JDK-supplied types such as `String`, `Date`, and `BigDecimal`. The domain model also has custom value-typed classes, those with a composition association in the UML diagram.

6.2 Mapping embeddable components

So far, the mapped classes of the domain model have all been entity classes, each with its own life cycle and identity. The `User` class, however, has a special kind of association with the `Address` class, as shown in figure 6.1.

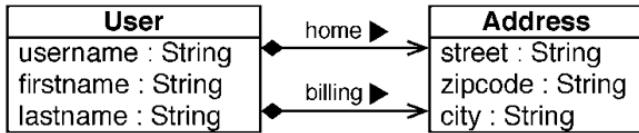


Figure 6.1 Composition of `User` and `Address`

In object-modeling terms, this association is a kind of *aggregation*—a *part-of* relationship. An aggregation is a form of association; it has some additional semantics concerning the life cycle of objects. In this case, we have an even stronger form, *composition*, where the life cycle of the part is fully dependent on the life cycle of the whole. An `Address` object cannot exist in the absence of the `User` object. So, a composed class in UML such as `Address` is often a candidate value type for the object/relational mapping.

6.2.1 The database schema

Let's map such a composition relationship with `Address` as a value type, with the same semantics as `String` or `BigDecimal`, and `User` as an entity. First, let's have a look at the targeted SQL schema in figure 6.2.

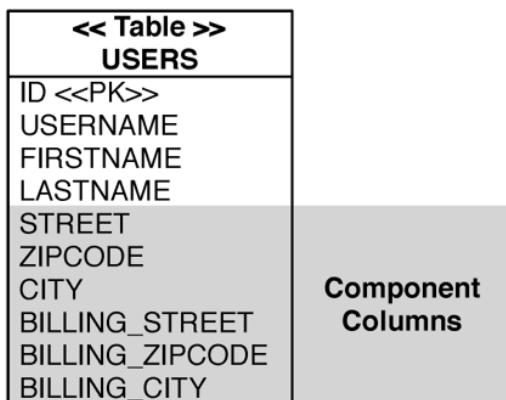


Figure 6.2 The columns of the components are embedded in the entity table.

There is only one mapped table, `USERS`, for the `User` entity. This table embeds all details of the components, where a single row holds a particular `User` and his/her `homeAddress` and `billingAddress`. If another entity has a reference to an `Address`—for example, `Shipment#deliveryAddress`—then the `SHIPMENT` table will also have all columns needed to store an `Address`.

This schema reflects value type semantics: a particular `Address` can't be shared; it doesn't have its own identity. Its primary key is the mapped database identifier of the owning entity. An embedded component has a dependent life cycle: when the owning entity instance is saved, the component instance is saved. When the owning entity instance is deleted, the component instance is deleted. No special SQL needs to be executed for this; all the data is in a single row.

Having “more classes than tables” is how fine-grained domain models are supported. Let's write the classes and mappings for this structure.

6.2.2 Making classes embeddable

Java has no concept of composition—a class or property can't be marked as a component. The only difference from an entity is the database identifier: a component class has no individual identity; hence, the component class requires no identifier property or identifier mapping. It's a simple POJO, as in the following listing.

Listing 6.7 Address class: an embeddable component

```

Path: Ch06/mapping-value-
      types/src/main/java/com/manning/javapersistence/ch06/model/Address.java

@Embeddable                                     #A
public class Address {
    @NotNull                                       #B
    @Column(nullable = false)                      #C
    private String street;

    @NotNull                                       #D
    @Column(nullable = false, length = 5)
    private String zipcode;

    @NotNull                                       #E
    @Column(nullable = false)
    private String city;

    public Address() {                            #F
    }

    public Address(String street, String zipcode, String city) { #G
        this.street = street;
        this.zipcode = zipcode;
        this.city = city;
    }
    //getters and setters
}

```

#A Instead of `@Entity`, this component POJO is marked with `@Embeddable`. It has no identifier property.

#B The `@NotNull` annotation is ignored by the DDL generation

#C `@Column(nullable=false)` is used for DDL generation.

#D The length argument of the `@Column` annotation will override the default generation of a column as `VARCHAR(255)`.

#E The type of the `city` column will be by default `VARCHAR(255)`.

Hibernate or Spring Data JPA using Hibernate call this no-argument constructor to #F create an instance and then populate the fields directly.

#G We can have additional (public) constructors for convenience.

In the listing above:

The properties of the embeddable class are all by default persistent, just like the properties of a persistent entity class. The property mappings can be configured with the same annotations, such as `@Column` or `@Basic`. The properties of the `Address` class map to the columns `STREET`, `ZIPCODE`, and `CITY` and are constrained with `NOT NULL`.

Issue: Hibernate Validator doesn't generate NOT NULL constraints

At the time of writing, an open issue remains with Hibernate Validator: Hibernate won't map `@NotNull` constraints on embeddable component properties to `NOT NULL` constraints when generating the database schema. Hibernate will only use `@NotNull` on the components' properties at runtime for Bean Validation. We have to map the property with `@Column(nullable = false)` to generate the constraint in the schema. The Hibernate bug database is tracking this issue as HVAL-3.

That's the entire mapping. There's nothing special about the `User` entity:

Listing 6.8 User class containing a reference to an Address

```
@Entity
@Table(name = "USERS")
public class User {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    private Long id;

    private Address homeAddress; #A

    // ...
}
```

#A The `Address` is `@Embeddable`; no annotation is needed here.

In the listing above:

Hibernate or Spring Data using Hibernate detect that the `Address` class is annotated with `@Embeddable`; the `STREET`, `ZIPCODE`, and `CITY` columns are mapped on the `USERS` table, the owning entity's table.

When we talked about property access earlier in this chapter, we mentioned that embeddable components inherit their access strategy from their owning entity. This means Hibernate or Spring Data using Hibernate will access the properties of the `Address` class with the same strategy as for `User` properties. This inheritance also affects the placement of mapping annotations in embeddable component classes. The rules are as follows:

- If the owning `@Entity` of an embedded component is mapped with field access, either implicitly with `@Id` on a field or explicitly with `@Access(AccessType.FIELD)` on the class, all mapping annotations of the embedded component class are expected on fields of the component class. Annotations are expected on the fields of the `Address` class, and the fields are directly read/written at runtime. Getter and setter methods on `Address` are optional.
- If the owning `@Entity` of an embedded component is mapped with property access, either implicitly with `@Id` on a getter method or explicitly with `@Access(AccessType.PROPERTY)` on the class, all mapping annotations of the embedded component class are expected on getter methods of the component class. Values are read and written by calling getter and setter methods on the embeddable

component class.

- If the embedded property of the owning entity class—`User#homeAddress` in the last example—is marked with `@Access(AccessType.FIELD)`, annotations are expected on the fields of the `Address` class, and fields are accessed at runtime.
- If the embedded property of the owning entity class—`User#homeAddress` in the last example—is marked with `@Access(AccessType.PROPERTY)`, annotations are expected on getter methods of the `Address` class, and access is made using getter and setter methods at runtime.
- If `@Access` annotates the embeddable class itself, the selected strategy will be used for reading mapping annotations on the embeddable class and runtime access.

Let's now compare the field-based and the property-based access. Why should you use one or another one?

- Using field-based access you may omit the getter methods for the fields that should not be exposed.
- The fields are declared on one single line, while the accessor methods will spread on multiple lines. Field-based access will provide easier readability of the code.
- Accessor methods may execute additional logic. If this is what you would like to happen when persisting an object, you may use property-based access. If the persistence would like to avoid these additional actions, you may use field-based access.

There's one more thing to remember: there's no elegant way to represent a `null` reference to an `Address`. Consider what would happen if the columns `STREET`, `ZIPCODE`, and `CITY` were nullable. Loading a `User` without any address information, what should be returned by `someUser.getHomeAddress()`? A `null` is returned in this case. Hibernate or Spring Data using Hibernate also store a `null` embedded property as `NULL` values in all mapped columns of the component. Consequently, storing a `User` with an “empty” `Address` (an `Address` instance exists, but all its properties are `null`), no `Address` instance will be returned when loading the `User`. This can be counterintuitive; on the other hand, you probably shouldn't have nullable columns anyway and avoid ternary logic, as you will most probably want your user to have a real address.

We should override the `equals()` and `hashCode()` methods of `Address` and compare instances by value. This isn't critically important as long as we don't have to compare instances: for example, by putting them in a `HashSet`. We'll discuss this issue later, in the context of collections; see section 8.2.1.

In a more realistic scenario, a user would probably have separate addresses for different purposes. Figure 6.1 showed an additional composition relationship between `User` and `Address`: the `billingAddress`.

6.2.3 Overriding embedded attributes

The `billingAddress` is another embedded component property of the `User` class that we need to use, so another `Address` has to be stored in the `USERS` table. This creates a mapping

conflict: so far, we only have columns in the schema to store one Address in STREET, ZIPCODE, and CITY.

We need additional columns to store another Address for each USERS row. When we map the billingAddress, override the column names:

Listing 6.9 Overriding the column names

```
@Entity
@Table(name = "USERS")
public class User {
    @Embedded
    @AttributeOverride(name = "street",
        column = @Column(name = "BILLING_STREET")) #A
    @AttributeOverride(name = "zipcode",
        column = @Column(name = "BILLING_ZIPCODE", length = 5)) #B
    @AttributeOverride(name = "city",
        column = @Column(name = "BILLING_CITY")) #B
    private Address billingAddress; #B

    public Address getBillingAddress() {
        return billingAddress;
    }

    public void setBillingAddress(Address billingAddress) {
        this.billingAddress = billingAddress;
    }
    // ...
}
```

#A The billingAddress field is marked as embedded. The @Embedded annotation actually isn't necessary. Mark either the component class or the property in the owning entity class (using both doesn't hurt but has no advantage).

The @Embedded annotation is useful if we want to map a third-party component class without source and no annotations but using the right getter/setter methods (like regular JavaBeans).

#B The repeatable @AttributeOverride annotation selectively overrides property mappings of the embedded class; in this example, we override all three properties and provides different column names. Now we can store two Address instances in the USERS table, each instance in a different set of columns (check the schema again in figure 6.2).

Each @AttributeOverride for a component property is “complete”: any JPA or Hibernate annotation on the overridden property is ignored. This means the @Column annotations on the Address class are ignored—all BILLING_* columns are nullable! (Bean Validation still recognizes the @NotNull annotation on the component property, though; only persistence annotations are overridden.)

We'll create two Spring Data JPA repository interfaces to interact with the database.

Listing 6.10 The UserRepository interface

```
Path: Ch06/mapping-value-
      types/src/main/java/com/manning/javapersistence/ch06/repositories/UserRepository.jav
      a
public interface UserRepository extends CrudRepository<User, Long> {
```

The `UserRepository` interface only extends `CrudRepository`, and it will inherit all methods from this interface. It is generified by `User` and `Long`, as it manages `User` entities having `Long` ids.

Listing 6.11 The ItemRepository interface

```
Path: Ch06/mapping-value-
      types/src/main/java/com/manning/javapersistence/ch06/repositories/ItemRepository.jav
      a

public interface ItemRepository extends CrudRepository<Item, Long> {
    Iterable<Item> findByMetricWeight(double weight);
}
```

The `ItemRepository` interface extends `CrudRepository`, and it will inherit all methods from this interface. Additionally, it declares the `findByMetricWeight` method, following the Spring Data JPA naming conventions. It is generified by `Item` and `Long`, as it manages `Item` entities having `Long` ids.

We'll test the functionality of the code we wrote using the Spring Data JPA framework, as demonstrated in the listing below. The source code of the book also contains the testing code alternatives that use JPA and Hibernate.

Listing 6.12 Testing the functionality of the persistence code

```

Path: Ch06/mapping-value-
      types/src/test/java/com/manning/javapersistence/ch06/MappingValuesSpringDataJPATest.
            java
@ExtendWith(SpringExtension.class)                                     #A
@ContextConfiguration(classes = {SpringDataConfiguration.class})      #B
public class MappingValuesSpringDataJPATest {

    @Autowired
    private UserRepository userRepository;                                #C
    @Autowired
    private ItemRepository itemRepository;                               #D

    @Test
    void storeLoadEntities() {

        User user = new User();                                         #E
        user.setUsername("username");                                    #E
        user.setHomeAddress(new Address("Flowers Street",
                                         "12345", "Boston"));          #E
        userRepository.save(user);                                       #F

        Item item = new Item();                                         #G
        item.setName("Some Item");                                     #G
        item.setMetricWeight(2);                                       #G
        item.setDescription("descriptiondescription");                 #G
        itemRepository.save(item);                                      #H

        List<User> users = (List<User>) userRepository.findAll();       #I
        List<Item> items = (List<Item>)
                           itemRepository.findByMetricWeight(2.0);        #J

        assertEquals(1, users.size());                                    #K
        assertEquals("username", users.get(0).getUsername());           #L
        assertEquals("Flowers Street", users.get(0).getHomeAddress().getStreet()); #M
        assertEquals("12345", users.get(0).getHomeAddress().getZipcode()); #N
        assertEquals("Boston", users.get(0).getHomeAddress().getCity());   #O
        assertEquals(1, items.size());                                    #P
        assertEquals("AUCTION: Some Item", items.get(0).getName());     #Q
        assertEquals("descriptiondescription", items.get(0).getDescription()); #R
        assertEquals(AuctionType.HIGHEST_BID, items.get(0).getAuctionType()); #S
        assertEquals("descriptiond...", items.get(0).getShortDescription()); #T
        assertEquals(2.0, items.get(0).getMetricWeight());                #U
        assertEquals(LocalDate.now(), items.get(0).getCreatedOn());       #V
        assertEquals(ChronoUnit.SECONDS.between(
                     LocalDateTime.now(),
                     items.get(0).getLastModified()) < 1),                  #W
    }
}

```

```

        () -> assertEquals(new BigDecimal("1.00"),
                           items.get(0).getInitialPrice())
                     #X
    );
}

#A We extend the test using SpringExtension. This extension is used to integrate the Spring test context with the
#B The Spring test context is configured using the beans defined in the SpringDataConfiguration class.
#C A UserRepository bean is injected by Spring through auto-wiring.
#D An ItemRepository bean is injected by Spring through auto-wiring. This is possible as the
    com.manning.javapersistence.ch06.repositories package where UserRepository and
    ItemRepository are located was used as the argument of the @EnableJpaRepositories annotation on
    the SpringDataConfiguration class. To revisit how the SpringDataConfiguration class looks like,
    you may go back to chapter 2.
#E We create and set a user.
#F We save it to the repository.
#G We create and set an item.
#H We save it to the repository.
#I We get the list of all users.
#J We get the list of items having the metric 2.0.
#K We check the size of the list of users.
#L We check the name.
#M Check the street address.
#N Check the zip code.
#O Check the city of the first user in the list.
#P Check the size of the list of items.
#Q Check the name of the first item.
#R Check its description.
#S Check the auction type.
#T Check the short description.
#U Check its metric weight.
#V Check the creation date.
#W Check the last modification date and time.
#X Check the initial price of the first item in the list. The last modification date and time is checked against the
    current date and time, to be within 1 second (to include the retrieval delay).

```

In the listing above:

The domain model can further improve reusability and be made more fine-grained by nesting embedded components.

6.2.4 Mapping nested embedded components

Let's consider the `Address` class and how it encapsulates address details: instead of a simple `city` string, this detail can be moved into a new `City` embeddable class. The changed domain model diagram is in figure 6.3. The SQL schema targeted for the mapping still has only one `USERS` table, as shown in figure 6.4. The source code to follow is to be found in the `mapping-value-types2` folder.

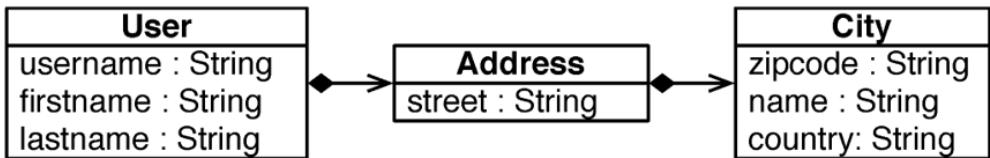


Figure 6.3 Nested composition of Address and City

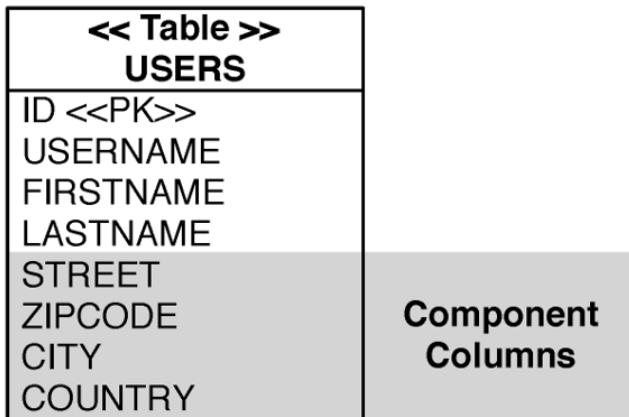


Figure 6.4 Embedded columns hold Address and City details.

An embeddable class can have an embedded property. Address has a city property.

Listing 6.13 The Address class with a city property

```

Path: Ch06/mapping-value-
      types2/src/main/java/com/manning/javapersistence/ch06/model/Address.java

@Embeddable
public class Address {
    @NotNull
    @Column(nullable = false)
    private String street;

    @NotNull
    @AttributeOverride(
        name = "name",
        column = @Column(name = "CITY", nullable = false)
    )
    private City city;
    // ...
}
  
```

We'll create the embeddable City class having only basic properties:

Listing 6.14 The embeddable City class

```
Path: Ch06/mapping-value-
      types2/src/main/java/com/manning/javapersistence/ch06/model/City.java

@Embeddable
public class City {
    @NotNull
    @Column(nullable = false, length = 5)
    private String zipcode;

    @NotNull
    @Column(nullable = false)
    private String name;

    @NotNull
    @Column(nullable = false)
    private String country;
    // ...
}
```

We could continue this kind of nesting by creating a `Country` class, for example. All embedded properties, no matter how deep they are in the composition, are mapped to columns of the owning entity's table—here, the `USERS` table.

The `name` property of the `City` class is mapped to the `CITY` column. This can be achieved with either (as demonstrated) an `@AttributeOverride` in `Address` or an override in the root entity class, `User`. Nested properties can be referenced with dot notation: for example, on `User#address`, `@AttributeOverride(name = "city.name")` references the `Address#city#name` attribute.

We'll come back to embedded components later, in section 8.2. We can even map collections of components or have references from a component to an entity.

At the beginning of this chapter, we analyzed basic properties and how Hibernate or Spring Data JPA using Hibernate map a JDK type such as `java.lang.String`, for example, to an appropriate SQL type. Let's find out more about this type system and how values are converted at a lower level.

6.3 Mapping Java and SQL types with converters

Until now, we've assumed that Hibernate or Spring Data JPA using Hibernate select the right SQL type when we map a `java.lang.String` property. Nevertheless, what is the correct mapping between the Java and SQL types, and how can we control it? We'll try to shape a correspondence between these types, delving into the specifics.

6.3.1 Built-in types

Any JPA provider has to support a minimum set of Java-to-SQL type conversions. Hibernate and Spring Data JPA using Hibernate support all of these mappings, as well as some additional adapters that aren't standard but are useful in practice. First, the Java primitives and their SQL equivalents.

PRIMITIVE AND NUMERIC TYPES

The built-in types shown in table 6.1 map Java primitives, and their wrappers, to appropriate SQL standard types. We've also included some other numeric types.

Table 6.1 Java primitive types that map to SQL standard types

Name	Java type	ANSI SQL type
integer	int, java.lang.Integer	INTEGER
long	long, java.lang.Long	BIGINT
short	short, java.lang.Short	SMALLINT
float	float, java.lang.Float	FLOAT
double	double, java.lang.Double	DOUBLE
byte	byte, java.lang.Byte	TINYINT
boolean	boolean, java.lang.Boolean	BOOLEAN
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC

The names are Hibernate-specific; we'll need them later when customizing type mappings.

You probably noticed that your DBMS product doesn't support some of the mentioned SQL types. These SQL type names are ANSI-standard type names. Most DBMS vendors ignore this part of the SQL standard, usually because their legacy-type systems preceded the standard. But JDBC provides a partial abstraction of vendor-specific data types, allowing Hibernate to work with ANSI-standard types when executing DML statements such as `INSERT` and `UPDATE`. For product-specific schema generation, Hibernate translates from the ANSI-standard type to an appropriate vendor-specific type using the configured SQL dialect. This means we usually don't have to worry about SQL data types if we let Hibernate create the schema for us.

If we have an existing schema and/or we need to know the native data type for our DBMS, we will look at the source of our configured SQL dialect. For example, the `H2Dialect` shipping with Hibernate contains this mapping from the ANSI `NUMERIC` type to the vendor-specific `DECIMAL` type: `registerColumnType(Types.NUMERIC, "decimal($p,$s)")`.

The `NUMERIC` SQL type supports decimal precision and scale settings. The default precision and scale setting, for a `BigDecimal` property, for example, is `NUMERIC(19, 2)`. To override this for schema generation, apply the `@Column` annotation on the property and set its `precision` and `scale` parameters.

Next are types that map to strings in the database.

CHARACTER TYPES

Table 6.2 shows types that map character and string value representations.

Table 6.2 Adapters for character and string values

Name	Java type	ANSI SQL type
string	<code>java.lang.String</code>	<code>VARCHAR</code>
character	<code>char[]</code> , <code>Character[]</code> , <code>java.lang.String</code>	<code>CHAR</code>
yes_no	<code>boolean</code> , <code>java.lang.Boolean</code>	<code>CHAR(1)</code> , ' <code>Y</code> ' or ' <code>N</code> '
true_false	<code>boolean</code> , <code>java.lang.Boolean</code>	<code>CHAR(1)</code> , ' <code>T</code> ' or ' <code>F</code> '
class	<code>java.lang.Class</code>	<code>VARCHAR</code>
locale	<code>java.util.Locale</code>	<code>VARCHAR</code>
timezone	<code>java.util.TimeZone</code>	<code>VARCHAR</code>
currency	<code>java.util.Currency</code>	<code>VARCHAR</code>

The Hibernate type system picks an SQL data type depending on the declared length of a string value: if the `String` property is annotated with `@Column(length = ...)` or `@Length` of Bean Validation, Hibernate selects the right SQL data type for the given string size. This selection also depends on the configured SQL dialect. For example, for MySQL, a length of up to 65,535 produces a regular `VARCHAR(length)` column when the schema is generated by Hibernate. For a length of up to 16,777,215, a MySQL-specific `MEDIUMTEXT` data type is produced, and even greater lengths use a `LONGTEXT`. The default length of Hibernate for all `java.lang.String` properties is 255, so without any further mapping, a `String` property maps to a `VARCHAR(255)`-column. We can customize this type selection by extending the class of our SQL dialect; read the dialect documentation and source code to find out more details for your DBMS product.

A database usually enables the internationalization of text with a sensible (UTF-8) default character set for the entire database or at least whole tables. This is a DBMS-specific setting.

If you need more fine-grained control and want to switch to the nationalized variants of character data type (e.g., NVARCHAR, NCHAR, or NCLOB), annotate the property mapping with `@org.hibernate.annotations.Nationalized`.

Also built in are some special converters for legacy databases or DBMSs with limited type systems, such as Oracle. The Oracle DBMS doesn't even have a truth-valued data type, the only data type required by the relational model. Many existing Oracle schemas, therefore, represent Boolean values with Y/N or T/F characters. Or—and this is the default in Hibernate's Oracle dialect—a column of type NUMBER(1,0) is expected and generated. Again, refer to the SQL dialect of the DBMS if you want to know all mappings from ANSI data type to vendor-specific type.

Next are types that map to dates and times in the database.

DATE AND TIME TYPES

Table 6.3 lists types associated with dates, times, and timestamps.

Table 6.3 Date and time types

Name	Java type	ANSI SQL type
date	java.util.Date, java.sql.Date	DATE
time	java.util.Date, java.sql.Time	TIME
timestamp	java.util.Date, java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
duration	java.time.Duration	BIGINT
instant	java.time.Instant	TIMESTAMP
localdatetime	java.time.LocalDateTime	TIMESTAMP
localdate	java.time.LocalDate	DATE
localtime	java.time.LocalTime	TIME
offsetdatetime	java.time.OffsetDateTime	TIMESTAMP
offsettime	java.time.OffsetTime	TIME
zoneddatetime	java.time.ZonedDateTime	TIMESTAMP

In the domain model, we may represent date and time data as either `java.util.Date`, `java.util.Calendar`, or the subclasses of `java.util.Date` defined in the `java.sql` package, or the Java 8 classes from the `java.time` package. The best decision at this time is to use the Java 8 API in the `java.time` package. These classes may represent a date, a time, a date with a time, or even to include the offset to the UTC zone (`OffsetDateTime` and `OffsetTime`). JPA 2.2 officially supports the Java 8 date and time classes.

Hibernate's behavior for `java.util.Date` properties might be a surprise at first: when storing a `java.util.Date`, Hibernate won't return a `java.util.Date` after loading. It will return a `java.sql.Date`, a `java.sql.Time`, or a `java.sql.Timestamp`, depending on whether the property was mapped with `TemporalType.DATE`, `TemporalType.TIME`, or `TemporalType.TIMESTAMP`.

Hibernate has to use the JDBC subclass when loading data from the database because the database types have higher accuracy than `java.util.Date`. A `java.util.Date` has millisecond accuracy, but a `java.sql.Timestamp` includes nanosecond information that may be present in the database. Hibernate won't cut off this information to fit the value into `java.util.Date`. This Hibernate behavior may lead to problems when trying to compare `java.util.Date` values with the `equals()` method; it's not symmetric with the `java.sql.Timestamp` subclass's `equals()` method.

The solution in such a case is simple and not even specific to Hibernate: don't call `aDate.equals(bDate)`. We should always compare dates and times by comparing Unix time milliseconds (assuming we don't care about the nanoseconds): `aDate.getTime() > bDate.getTime()`, for example, is true if `aDate` is a later time than `bDate`. Be careful: collections such as `HashSet` call the `equals()` method as well. Don't mix `java.util.Date` and `java.sql.Date|Time|Timestamp` values in such a collection. We won't have this kind of problem with a `Calendar` property. When storing a `Calendar` value, Hibernate will always return a `Calendar` value, created with `Calendar.getInstance()` (the actual type depends on locale and time zone).

Alternatively, we can write our own *converter*, as shown later in this chapter, and transform any instance of a `java.sql` temporal type, given by Hibernate, into a plain `java.util.Date` instance. A custom converter is also a good starting point if, for example, a `Calendar` instance should have a non-default time zone after loading the value from the database.

All these concerns will go away by choosing to represent date and time data using the Java 8 classes `LocalDate`, `LocalTime`, `LocalDateTime`, as previously demonstrated in this chapter. As you may still encounter a lot of code using the old classes, you should be aware of the problems that these ones are raising.

Next are types that map to binary data and large values in the database.

BINARY AND LARGE VALUE TYPES

Table 6.4 lists types for handling binary data and large values. Note that only `binary` is supported as the type of an identifier property.

First, consider how Hibernate represents the potentially large value, as `binary` or `text`.

Table 6.4 Binary and large value types

Name	Java type	ANSI SQL type
binary	byte[], java.lang.Byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	java.io.Serializable	VARBINARY

If a property in the persistent Java class is of type `byte[]`, Hibernate maps it to a `VARBINARY` column. The real SQL data type depends on the dialect; for example, in PostgreSQL, the data type is `BYTEA`, and in Oracle DBMS, it's `RAW`. In some dialects, the `length` set with `@Column` also affects the selected native type: for example, `LONG RAW` for a length of 2000 and greater in Oracle. In MySQL, the default SQL data type will be `TINYBLOB`. Depending on the `length` set with `@Column`, it may be `BLOB`, `MEDIUMBLOB`, or `LONGBLOB`.

A `java.lang.String` property is mapped to an SQL `VARCHAR` column and the same for `char[]` and `Character[]`. As we've discussed, some dialects register different native types depending on declared length.

In both cases, Hibernate initializes the property value right away when the entity instance that holds the property variable is loaded. This is inconvenient when we have to deal with potentially large values, so we usually want to override this default mapping. The JPA specification has a convenient shortcut annotation for this purpose, `@Lob`:

```
@Entity
public class Item {
    @Lob
    private byte[] image;
    @Lob
    private String description;
}
```

This maps the `byte[]` to an SQL `BLOB` data type and the `String` to a `CLOB`. Unfortunately, we still don't get lazy loading with this design. Hibernate or Spring Data JPA using Hibernate would have to intercept field access and, for example, load the bytes of the `image` when we call `someItem.getImage()`. This approach requires bytecode instrumentation of the classes after compilation, for the injection of extra code. We'll discuss lazy loading through bytecode instrumentation and interception in section 12.1.3.

Alternatively, we can switch the type of property in the Java class. JDBC supports locator objects (LOBs) directly. If the Java property is `java.sql.Clob` or `java.sql.Blob`, we get lazy loading without bytecode instrumentation:

```
@Entity
public class Item {
    @Lob
    private java.sql.Blob imageBlob;
    @Lob
    private java.sql.Clob description;
}
```

What does BLOB/CLOB mean?

Jim Starkey, who came up with the idea of LOBs, says that the marketing department created the terms BLOB and CLOB. BLOB is interpreted as Binary Large OBject: binary data (usually a multimedia object - image, video, or audio) stored as a single entity. CLOB means Character Large OBject - character data stored in a separate location that the table only references.

These JDBC classes include behavior to load values on demand. When the owning entity instance is loaded, the property value is a placeholder, and the real value isn't immediately materialized. Once we access the property, within the same transaction, the value is materialized or even streamed directly (to the client) without consuming temporary memory:

```
Item item = em.find(Item.class, ITEM_ID);
InputStream imageDataStream = item.getImageBlob().getBinaryStream();          #A
ByteArrayOutputStream outStream = new ByteArrayOutputStream();                  #B
StreamUtils.copy(imageDataStream, outStream);
byte[] imageBytes = outStream.toByteArray();
```

#A We can stream the bytes directly.

#B Or we can materialize them into memory.

The downside is that the domain model is then bound to JDBC; in unit tests, we can't access LOB properties without a database connection.

To create and set a `Blob` or `Clob` value, Hibernate offers some convenience methods. This example reads `byteLength` bytes from an `InputStream` directly into the database, without consuming temporary memory:

```
Session session = em.unwrap(Session.class);          #A
Blob blob = session.getLobHelper()                  #B
    .createBlob(inputStream, byteLength);
someItem.setImageBlob(blob);
em.persist(someItem);
```

#A We need the native Hibernate API, so we have to unwrap the Session from the EntityManager.

#B Then, we need to know the number of bytes we want to read from the stream.

Finally, Hibernate provides fallback serialization for any property type that is `java.io.Serializable`. This mapping converts the value of the property to a byte stream

stored in a `VARBINARY` column. Serialization and deserialization occur when the owning entity instance is stored and loaded. Naturally, we should use this strategy with extreme caution because data lives longer than applications. One day, nobody will know what those bytes in the database mean. Serialization is sometimes useful for temporary data, such as user preferences, login session data, and so on.

Hibernate will pick the right type of adapter depending on the Java type of the property. If you don't like the default mapping, read on to override it.

SELECTING A TYPE ADAPTER

We have seen many adapters and their Hibernate names in the previous sections. Use the name when overriding Hibernate's default type selection and explicitly select a particular adapter:

```
@Entity
public class Item {
    @org.hibernate.annotations.Type(type = "yes_no")
    private boolean verified = false;
}
```

Instead of `BIT`, this `boolean` now maps to a `CHAR` column with values `Y` or `N`.

We can also override an adapter globally in the Hibernate boot configuration with a custom user type, which we'll demonstrate how to write later in **this** chapter:

```
metaBuilder.applyBasicType(new MyUserType(), new String[]{"date"});
```

This setting will override the built-in `Date` type adapter and delegate value conversion for `java.util.Date` properties to the custom implementation.

We consider this extensible type system one of Hibernate's core features and an important aspect that makes it so flexible. Next, we explore the type system and JPA custom converters in more detail.

6.3.2 Creating custom JPA converters

A new requirement for the online auction system is using multiple currencies. Rolling out this kind of change can be complex. We have to modify the database schema, we may have to migrate existing data from the old to the new schema, and we have to update all applications that access the database. In this section, we demonstrate how JPA converters and the extensible Hibernate type system can assist in this process, providing an additional, flexible buffer between the application and the database.

To support multiple currencies, we introduce a new class in the `CaveatEmptor` domain model: `MonetaryAmount`, shown in the following listing.

Listing 6.15 Immutable MonetaryAmount value-type class

```

Path: Ch06/mapping-value-
      types2/src/main/java/com/manning/javapersistence/ch06/model/MonetaryAmount.java

public class MonetaryAmount implements Serializable {                      #A
    private final BigDecimal value;                                     #B
    private final Currency currency;                                    #B

    public MonetaryAmount(BigDecimal value, Currency currency) {        #B
        this.value = value;
        this.currency = currency;
    }

    public BigDecimal getValue() {                                         #B
        return value;
    }

    public Currency getCurrency() {                                       #B
        return currency;
    }

    @Override
    public boolean equals(Object o) {                                      #C
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MonetaryAmount that = (MonetaryAmount) o;
        return Objects.equals(value, that.value) &&
               Objects.equals(currency, that.currency);
    }

    public int hashCode() {                                              #C
        return Objects.hash(value, currency);
    }

    public String toString() {                                            #D
        return value + " " + currency;
    }

    public static MonetaryAmount fromString(String s) {                  #E
        String[] split = s.split(" ");
        return new MonetaryAmount(
            new BigDecimal(split[0]),
            Currency.getInstance(split[1])
        );
    }
}

```

#A This value-typed class should be `java.io.Serializable`: when Hibernate stores entity instance data in the shared second-level cache, it *disassembles* the entity's state. If an entity has a `MonetaryAmount` property, the serialized representation of the property value is stored in the second-level cache region. When entity data is retrieved from the cache region, the property value is deserialized and reassembled.

#B The class defines the `value` and `currency` fields, a constructor using both of them and getters on these fields.

#C The class implements the `equals()` and `hashCode()` methods and compares monetary amounts “by value”.

#D The class implements the `toString()` method.

#E The class implements a static method to create an instance from a `String`.

CONVERTING BASIC PROPERTY VALUES

As is often the case, the database folks can't implement multiple currencies right away and need more time. All they can provide quickly is a column data type change in the database schema.

We'll add the `buyNowPrice` field to the `Item` class.

```
Path: Ch06/mapping-value-types/
src/main/java/com/manning/javapersistence/ch06/model/Item.java

@NotNull
@Convert(converter = MonetaryAmountConverter.class)
@Column(name = "PRICE", length = 63)
private MonetaryAmount buyNowPrice;
```

We'll store the `BUYNOWPRICE` in the `ITEM` table in a `VARCHAR` column and will append the currency code of the monetary amount to its string value. We will store, for example, the value `11.23 USD` or `99 EUR`.

We convert an instance of `MonetaryAmount` to such a `String` representation when storing data. When loading data, we convert the `String` back into a `MonetaryAmount`. The simplest solution is an implementation of a standardized extension point in JPA, `javax.persistence.AttributeConverter`, the `MonetaryAmoutConverter` class used in the `@Convert` annotation above and shown in the next listing.

Listing 6.16 Converting between strings and MonetaryValue

```
Path: Ch06/mapping-value-
types2/src/main/java/com/manning/javapersistence/ch06/converter/MonetaryAmountConver-
ter.java

@Converter
public class MonetaryAmountConverter
    implements AttributeConverter<MonetaryAmount, String> { #A

    @Override
    public String convertToDatabaseColumn(MonetaryAmount monetaryAmount) { #B
        return monetaryAmount.toString();
    } #B

    @Override
    public MonetaryAmount convertToEntityAttribute(String s) { #C
        return MonetaryAmount.fromString(s);
    } #C
}
```

#A A converter has to implement the `AttributeConverter` interface; the two types arguments are the type of the Java property and the type in the database schema. The Java type is `MonetaryAmount`, and the database type is `String`, which maps, as usual, to an SQL `VARCHAR`. We must annotate the class with `@Converter`.

#B The `convertToDatabaseColumn` method will convert from the `MonetaryAmount` entity type to the string database column.

#C The `convertToEntityAttribute` method will convert from the string database column to the `MonetaryAmount` entity type.

In the example above:

We'll update the code we wrote using the Spring Data JPA framework, as demonstrated in the listing below. The source code of the book also contains the testing code alternatives that use JPA and Hibernate.

Listing 6.17 Testing the functionality of the persistence code

```

@ExtendWith(SpringExtension.class)                                     #A
@ContextConfiguration(classes = {SpringDataConfiguration.class})      #B
public class MappingValuesSpringDataJPATest {

    @Autowired                                                       #C
    private UserRepository userRepository;                            #C

    @Autowired                                                       #D
    private ItemRepository itemRepository;                          #D

    @Test
    void storeLoadEntities() {

        City city = new City();                                      #E
        city.setName("Boston");                                    #E
        city.setZipcode("12345");                                 #E
        city.setCountry("USA");                                   #E
        User user = new User();                                  #F
        user.setUsername("username");                           #F
        user.setHomeAddress(new Address("Flowers Street", city)); #F
        userRepository.save(user);                             #G

        Item item = new Item();                                    #H
        item.setName("Some Item");                            #H
        item.setMetricWeight(2);                                #H
        item.setBuyNowPrice(new MonetaryAmount(
            BigDecimal.valueOf(1.1), Currency.getInstance("USD"))); #H
        item.setDescription("descriptiondescription");          #H
        itemRepository.save(item);                            #I

        List<User> users = (List<User>) userRepository.findAll(); #J
        List<Item> items = (List<Item>)
            itemRepository.findByMetricWeight(2.0);             #K

        assertAll(
            () -> assertEquals(1, users.size()),                  #L
            () -> assertEquals("username", users.get(0).getUsername()), #M
            () -> assertEquals("Flowers Street",
                users.get(0).getHomeAddress().getStreet()),           #N
            () -> assertEquals("Boston",
                users.get(0).getHomeAddress().getCity().getName()),   #O
            () -> assertEquals("12345",
                users.get(0).getHomeAddress().getCity().getZipcode()), #P
            () -> assertEquals("USA",
                users.get(0).getHomeAddress().getCity().getCountry()), #Q
            () -> assertEquals(1, items.size()),                  #R
            () -> assertEquals("AUCTION: Some Item",
                items.get(0).getName()),                           #S
            () -> assertEquals("1.1 USD",
                items.get(0).getBuyNowPrice().toString()),          #T
            () -> assertEquals("descriptiondescription",
                items.get(0).getDescription()));                 #U
    }
}

```

```

        () -> assertEquals(AuctionType.HIGHEST_BID,           #V
                            items.get(0).getAuctionType()),
        () -> assertEquals("description...",               #W
                            items.get(0).getShortDescription()),
        () -> assertEquals(2.0, items.get(0).getMetricWeight()), #X
        () -> assertEquals(LocalDate.now(),                 #Y
                            items.get(0).getCreatedOn()),
        () -> assertTrue(ChronoUnit.SECONDS.between(          #Z
                            LocalDateTime.now(),
                            items.get(0).getLastModified()) < 1),
        () -> assertEquals(new BigDecimal("1.00"),           #Z
                            items.get(0).getInitialPrice())
    );
}

}

```

#A We extend the test using `SpringExtension`. This extension is used to integrate the Spring test context with the JUnit 5 Jupiter test.

#B The Spring test context is configured using the beans defined in the `SpringDataConfiguration` class #B.

#C A `UserRepository` bean is injected by Spring through auto-wiring.

#D An `ItemRepository` bean is injected by Spring through auto-wiring. This is possible as the `com.manning.javapersistence.ch06.repositories` package where `UserRepository` and `ItemRepository` are located was used as the argument of the `@EnableJpaRepositories` annotation on the `SpringDataConfiguration` class. To revisit how the `SpringDataConfiguration` class looks like, you may go back to chapter 2.

#E We create and set a city.

#F We create and set a user.

#G We save it to the repository.

#H We create and set an item.

#I We save it to the repository.

#J We get the list of all users.

#K We get the list of items having the metric 2.0.

#L We check the size of the list of users.

#M We check the name of the first user in the list.

#N Check the street address of the first user in the list.

#O Check the city of the first user in the list.

#P Check the zip code of the first user in the list.

#Q Check the country of the first user in the list.

#R Check the size of the list of items.

#S Check the name of the first item.

#T Check its current buying price.

#U Check its description.

#V Check the auction type.

#W Check its short description.

#X Check its metric weight.

#Y Check the creation date.

#Z Check the last modification date and time, and the initial price of the first item in the list. The last modification date and time is checked against the current date and time, to be within 1 second (to include the retrieval delay).

Later, when the DBA upgrades the database schema and offers separate columns for the monetary amount value and currency, we'll only have to change the application in a few places. We'll drop the `MonetaryAmountConverter` from the project and make

`MonetaryAmount` an `@Embeddable`; it then maps automatically to two database columns. It's easy to selectively enable and disable converters, too, if some tables in the schema haven't been upgraded.

The converter we just wrote is for `MonetaryAmount`, a new class in the domain model. Converters aren't limited to custom classes: we can even override Hibernate's built-in type adapters. For example, we could create a custom converter for some or even all `java.util.Date` properties in the domain model.

We can apply converters to properties of entity classes, like `Item#buyNowPrice` in the last example. We can also apply them to properties of embeddable classes.

CONVERTING PROPERTIES OF COMPONENTS

We've been making a case for fine-grained domain models in this chapter. Earlier, we isolated the address information of the `User` and mapped the embeddable `Address` class. We continue the process and introduces inheritance with an abstract `Zipcode` class, as shown in figure 6.5. The source code to follow is to be found in the `mapping-value-types3` folder.

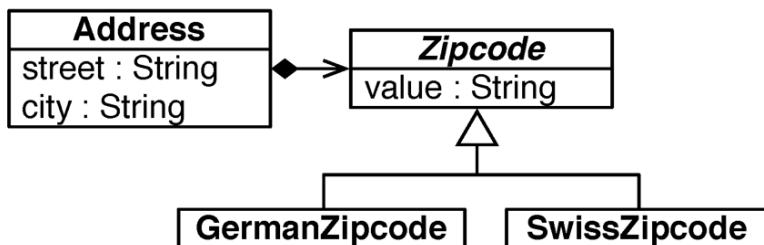


Figure 6.5. The abstract `Zipcode` class has two concrete subclasses.

The `Zipcode` class is trivial, but we have to implement equality by `value`:

```

Path: Ch06/mapping-value-
      types3/src/main/java/com/manning/javapersistence/ch06/model/Zipcode.java

public abstract class Zipcode {
    private String value;

    public Zipcode(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Zipcode zipcode = (Zipcode) o;
        return Objects.equals(value, zipcode.value);
    }

    @Override
    public int hashCode() {
        return Objects.hash(value);
    }
}

```

We can now encapsulate domain subclasses, the difference between German and Swiss postal codes, and any processing:

```

Path: Ch06/mapping-value-
      types3/src/main/java/com/manning/javapersistence/ch06/model/GermanZipcode.java

public class GermanZipcode extends Zipcode {
    public GermanZipcode(String value) {
        super(value);
    }
}

```

We haven't implemented any special processing in the subclass. We'll start with the most obvious difference: German zip codes are five numbers long, Swiss are four. A custom converter will take care of this.

Listing 6.18 The ZipcodeConverter class

```
Path: Ch06/mapping-value-
      types3/src/main/java/com/manning/javapersistence/ch06/converter/ZipcodeConverter.java

@Converter
public class ZipcodeConverter
    implements AttributeConverter<Zipcode, String> {

    @Override
    public String convertToDatabaseColumn(Zipcode attribute) { #A
        return attribute.getValue(); #A
    } #A

    @Override
    public Zipcode convertToEntityAttribute(String s) { #B
        if (s.length() == 5) #B
            return new GermanZipcode(s); #C
        else if (s.length() == 4) #C
            return new SwissZipcode(s); #D
        throw new IllegalArgumentException(
            "Unsupported zipcode in database: " + s); #E
    } #E
}
```

#A Hibernate calls the `convertToDatabaseColumn()` method of this converter when storing a property value; we return a `String` representation. The column in the schema is `VARCHAR`. When loading a value, we examine its length and creates either a `GermanZipcode` or `SwissZipcode` instance. This is a custom type discrimination routine; we can pick the Java type of the given value.

#B Hibernate calls the `convertToEntityAttribute` method of this converter when loading a property from the database.

#C If the length of the string is 5, a new `GermanZipcode` is created.

#D If the length of the string is 4, a new `SwissZipcode` is created.

#E Otherwise, an exception is thrown – the zip code in the database is not supported.

Now we apply this converter on some `Zipcode` properties—for example, the embedded `homeAddress` of a `User`:

```
Path: Ch06/mapping-value-types3/
src/main/java/com/manning/javapersistence/ch06/model/User.java

@Entity
@Table(name = "USERS")
public class User {
    @Convert(
        converter = ZipcodeConverter.class,
        attributeName = "city.zipcode"
    )
    private Address homeAddress;
    // ...
}
```

The `attributeName` declares the `zipcode` attribute of the embeddable `Address` class. This setting supports a dot syntax for the attribute path; if `zipcode` isn't a property of the

`Address` class but is a property of a nested embeddable `City` class, it is referenced with `city.zipcode`, its nested path.

In JPA 2.2, we can apply several `@Convert` annotations on a single embedded property to convert several attributes of the `Address`. Up to JPA 2.1, we had to group them within a `@Converts` annotation. We can also apply converters to values of collections and maps if their values and/or keys are of basic or embeddable type. For example, we can add the `@Convert` annotation on a persistent `Set<Zipcode>`. We'll demonstrate how to map persistent collections later, with `@ElementCollection`, in chapter 8.

For persistent maps, the `attributeName` option of the `@Convert` annotation has some special syntax:

- On a persistent `Map<Address, String>`, we can apply a converter for the `zipcode` property of each map key with the attribute name `key.zipcode`.
- On a persistent `Map<String, Address>`, we can apply a converter for the `zipcode` property of each map value with the attribute name `value.zipcode`.
- On a persistent `Map<Zipcode, String>`, we can apply a converter for the key of each map entry with the attribute name `key`.
- On a persistent `Map<String, Zipcode>`, we can apply a converter for the value of each map entry by not setting any `attributeName`.

As before, the attribute name can be a dot-separated path if the embeddable classes are nested; we can write `key.city.zipcode` to reference the `zipcode` property of the `City` class in a composition with the `Address` class.

Some limitations of the JPA converters are as follows:

- We can't apply them to identifier or version properties of an entity.
- We shouldn't apply a converter on a property mapped with `@Enumerated` or `@Temporal`, because these annotations already declare what kind of conversion has to occur. If we want to apply a custom converter for enums or date/time properties, we shouldn't annotate them with `@Enumerated` or `@Temporal`.

We'll have to change a little the testing code we wrote. We'll replace this line:

```
city.setZipcode("12345");
```

with this one:

```
city.setZipcode(new GermanZipcode("12345"));
```

We'll also replace this line:

```
() -> assertEquals("12345",
    users.get(0).getHomeAddress().getCity().getZipcode())
```

with this one:

```
() -> assertEquals("12345",
    users.get(0).getHomeAddress().getCity().getZipcode().getValue())
```

The source code of the book contains the tests using Spring Data JPA, Hibernate, and JPA.

Let's get back to multiple currency support in `CaveatEmptor`. The database administrators changed the schema again and we have to update the application.

6.3.3 Extending Hibernate with UserTypes

Finally, new columns were added to the database schema to support multiple currencies. The `ITEM` table now has a `BUYNOWPRICE_AMOUNT` and a separate column for the currency of the amount, `BUYNOWPRICE_CURRENCY`. There are also `INITIALPRICE_AMOUNT` and `INITIALPRICE_CURRENCY` columns. We have to map these columns to the `MonetaryAmount` properties of the `Item` class, `buyNowPrice`, and `initialPrice`.

Ideally, we don't want to change the domain model; the properties already use the `MonetaryAmount` class. Unfortunately, the standardized JPA converters don't support the transformation of values from/to multiple columns. Another limitation of JPA converters is integration with the query engine. We can't write the following query: `select i from Item i where i.buyNowPrice.amount > 100`. Thanks to the converter from the previous section, Hibernate knows how to convert a `MonetaryAmount` to and from a string. It doesn't know that `MonetaryAmount` has an `amount` attribute, so it can't parse such a query.

A simple solution would be to map `MonetaryAmount` as `@Embeddable`, as analyzed earlier in this chapter for the `Address` class. Each property of `MonetaryAmount—amount` and `currency`—maps to its respective database column.

The database admins, however, add a twist to their requirements: because other old applications also access the database, we have to convert each amount to a target currency before storing it in the database. For example, `Item#buyNowPrice` should be stored in US dollars, and `Item#initialPrice` should be stored in Euros. (If this example seems far-fetched, we can assure you that you'll see worse in the real world. The evolution of a shared database schema can be costly but is, of course, necessary because data always lives longer than applications.) Hibernate offers a native converter API: an extension point that allows much more detailed and low-level customization access.

THE EXTENSION POINTS

Hibernate's extension interfaces for its type system can be found in the `org.hibernate.usertype` package. The following interfaces are available:

- `UserType`—We can transform values by interacting with the plain JDBC - `PreparedStatement` (when storing data) and `ResultSet` (when loading data). By implementing this interface, we can also control how Hibernate caches and dirty-checks values.
- `CompositeUserType`— We can tell Hibernate that the `MonetaryAmount` component has two properties: `amount` and `currency`. We can then reference these properties in queries with dot notation: for example, `select avg(i.buyNowPrice.amount) from Item i`.
- `ParameterizedType`—This provides settings to the adapter in mappings. We have to implement this interface for the `MonetaryAmount` conversion because, in some mappings, we want to convert the amount to US dollars and in other mappings to

Euros. We only have to write a single adapter and can customize its behavior when mapping a property.

- `DynamicParameterizedType`—This more powerful settings API gives access to dynamic information in the adapter, such as the mapped column and table names. We might as well use this instead of `ParameterizedType`; there is no additional cost or complexity.
- `EnhancedUserType`—This is an optional interface for adapters of identifier properties and discriminators. Unlike JPA converters, a `UserType` in Hibernate can be an adapter for any kind of entity property. Because `MonetaryAmount` won't be the type of an identifier property or discriminator, we won't need it.
- `UserVersionType`—This is an optional interface for adapters of version properties.
- `UserCollectionType`—This rarely needed interface is used to implement custom collections. We have to implement it to persist a non-JDK collection (e.g., the Google Guava collections: `Multiset`, `Multimap`, `BiMap`, `Table`, etc.) and preserve additional semantics.

The custom type adapter for `MonetaryAmount` will implement several of these interfaces. The source code to follow is to be found in the `mapping-value-types4` folder.

IMPLEMENTING THE `USER TYPE`

`MonetaryAmountUserType` is a large class and we examine it here.

Listing 6.19 The `MonetaryAmountUserType` class

```
Path: Ch06/mapping-value-
      types4/src/main/java/com/manning/javapersistence/ch06/converter/MonetaryAmountUserTy
      pe.java
public class MonetaryAmountUserType                                #A
    implements CompositeUserType, DynamicParameterizedType {      #A

    private Currency convertTo;                                     #B

    public void setParameterValues(Properties parameters) {          #C
        String convertToParameter = parameters.getProperty("convertTo"); #D
        this.convertTo = Currency.getInstance(
            convertToParameter != null ? convertToParameter : "USD"   #E
        );
    }

    public Class returnedClass() {                                     #F
        return MonetaryAmount.class;                                  #F
    }

    public boolean isMutable() {                                      #G
        return false;                                                 #G
    }

    public Object deepCopy(Object value) {                            #H
        return value;                                                 #H
    }

    public Serializable disassemble(Object value,                      #I

```

```

        SharedSessionContractImplementor session){ #I
    return value.toString(); #I
}

public Object assemble(Serializable cached, #J
                      SharedSessionContractImplementor session, Object owner) { #J
    return MonetaryAmount.fromString((String) cached); #J
} #J

public Object replace(Object original, Object target, #K
                      SharedSessionContractImplementor session, Object owner) { #K
    return original; #K
} #K

public boolean equals(Object x, Object y) { #L
    return x == y || !(x == null || y == null) && x.equals(y); #L
} #L

public int hashCode(Object x) { #L
    return x.hashCode(); #L
} #L

public Object nullSafeGet(ResultSet resultSet, #M
                         String[] names, #M
                         SharedSessionContractImplementor session, #M
                         Object owner) throws SQLException { #M
    BigDecimal amount = resultSet.getBigDecimal(names[0]); #N
    if (resultSet.wasNull()) #N
        return null; #N
    Currency currency = #N
        Currency.getInstance(resultSet.getString(names[1])); #N
    return new MonetaryAmount(amount, currency); #O
} #O

public void nullSafeSet(PreparedStatement statement, #P
                       Object value, int index, #P
                       SharedSessionContractImplementor session) throws SQLException { #P
    if (value == null) { #Q
        statement.setNull( #Q
            index, #Q
            StandardBasicTypes.BIG_DECIMAL.sqlType()); #Q
        statement.setNull( #Q
            index + 1, #Q
            StandardBasicTypes.CURRENCY.sqlType()); #Q
    } else { #R
        MonetaryAmount amount = (MonetaryAmount) value; #R
        MonetaryAmount dbAmount = convert(amount, convertTo); #R
        statement.setBigDecimal(index, dbAmount.getValue()); #S
        statement.setString(index + 1, convertTo.getCurrencyCode()); #S
    } #R
} #R

public MonetaryAmount convert(MonetaryAmount amount, #T
                             Currency toCurrency) { #T
    return new MonetaryAmount( #U
        amount.getValue().multiply(new BigDecimal(2)), #U
        toCurrency #U
    ); #U
} #U
}

```

```

public String[] getPropertyNames() {                                     #V
    return new String[]{"value", "currency"};                         #V
}

public Type[] getPropertyTypes() {                                     #W
    return new Type[]{                                         #W
        StandardBasicTypes.BIG_DECIMAL,                         #W
        StandardBasicTypes.CURRENCY};                           #W
}

public Object getPropertyValue(Object component,                      #X
                               int property) {                                #X
    MonetaryAmount monetaryAmount = (MonetaryAmount) component; #X
    if (property == 0)                                           #X
        return monetaryAmount.getValue();                         #X
    else                                                       #X
        return monetaryAmount.getCurrency();                    #X
}

public void setPropertyValue(Object component,                      #Y
                            int property,                         #Y
                            Object value) {                     #Y
    throw new UnsupportedOperationException(                  #Y
        "MonetaryAmount is immutable");                   #Y
}
}

```

#A The interfaces we implement are `CompositeUserType` and `DynamicParameterizedType`.

#B The target currency

#C The `setParameterValues` method.

#D We use the `convertTo` parameter to determine the target currency when saving a value into the database.

#E If the parameter hasn't been set, default to US dollars. This method is inherited from the `DynamicParameterizedType` interface.

#F The method `returnedClass` adapts the given class, in this case, `MonetaryAmount`. This method and the ones to follow are inherited from the `CompositeUserType` interface.

#G Hibernate can enable some optimizations if it knows that `MonetaryAmount` is immutable.

#H If Hibernate has to make a copy of the value, it calls this `deepCopy` method. For simple immutable classes like `MonetaryAmount`, we can return the given instance.

#I Hibernate calls the `disassemble` method when it stores a value in the global shared second-level cache. We need to return a `Serializable` representation. For `MonetaryAmount`, a `String` representation is an easy solution. Or, because `MonetaryAmount` is `Serializable`, we could return it directly.

#J Hibernate calls the `assemble` method when it reads the serialized representation from the global shared second-level cache. We create a `MonetaryAmount` instance from the `String` representation. Or, if we stored a serialized `MonetaryAmount`, we could return it directly.

#K The `replace` method is called during `EntityManager#merge()` operations. We need to return a copy of the original. Or, if the value type is immutable, like `MonetaryAmount`, we can return the original.

#L Hibernate uses value equality to determine whether the value was changed and the database needs to be updated. We rely on the equality and hash code routines we have already written on the `MonetaryAmount` class.

#M The `nullSafeGet` method is called to read the `ResultSet` when a `MonetaryAmount` value has to be retrieved from the database.

#N We take the `amount` and `currency` values as given in the query result.

#O We create a new instance of `MonetaryAmount`.

#P The `nullSafeSet` method is called when a `MonetaryAmount` value has to be stored in the database.

#Q If `MonetaryAmount` was `null`, we call `setNull()` to prepare the statement. #R Otherwise, we convert the value to the target currency.

#S We then set the amount and currency on the provided `PreparedStatement()`.

#T We can implement whatever currency conversion routine we need.

#U For the sake of the example, we double the value so we can easily test whether the conversion was successful. We'll have to replace this code with a real currency converter in a real application. This `convert` method is not a method of the Hibernate `UserType` API.

#V The remaining methods inherited from `CompositeUserType` are providing the details of the `MonetaryAmount` properties, so Hibernate can integrate the class with the query engine. The `getPropertyNames` method will return a `String` array with two elements, "value" and "currency" – the names of the properties of the `MonetaryAmount` class.

#W The `GetPropertyTypes` method will return a `Type` array with two elements, `BIG_DECIMAL` and `CURRENCY` – the types of the properties of the `MonetaryAmount` class.

#X The `GetPropertyValue` method will return either the `value` field or the `currency` field of the `MonetaryAmount` object, depending on the property index.

#Y The `setPropertyValue` method will not allow setting any field of the `MonetaryAmount` object, as this one is immutable.

The `MonetaryAmountUserType` is now complete, and we can already use it in mappings with its fully qualified class name in `@org.hibernate.annotations.Type`, as demonstrated in the section "Selecting a type adapter". This annotation also supports parameters, so we can set the `convertTo` argument to the target currency.

But we recommend creating *type definitions*, bundling the adapter with some parameters.

USING TYPE DEFINITIONS

We need an adapter that converts to US dollars and an adapter that converts to Euros. If we declare these parameters once as a *type definition*, we don't have to repeat them in property mappings. A good location for type definitions is package metadata, in a `package-info.java` file:

```
Path: Ch06/mapping-value-types4/
src/main/java/com/manning/javapersistence/ch06/converter/package-info.java

@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name = "monetary_amount_usd",
        typeClass = MonetaryAmountUserType.class,
        parameters = {@Parameter(name = "convertTo", value = "USD")})
),
    @org.hibernate.annotations.TypeDef(
        name = "monetary_amount_eur",
        typeClass = MonetaryAmountUserType.class,
        parameters = {@Parameter(name = "convertTo", value = "EUR")})
)
package com.manning.javapersistence.ch06.converter;
import org.hibernate.annotations.Parameter;
```

We are now ready to use the adapters in mappings, using the names `monetary_amount_usd` and `monetary_amount_eur`.

We map the `buyNowPrice` and `initialPrice` of `Item`:

```

Path: Ch06/mapping-value-types4/
src/main/java/com/manning/javapersistence/ch06/model/Item.java

@Entity
public class Item {
    @NotNull
    @org.hibernate.annotations.Type(
        type = "monetary_amount_usd"
    )
    @org.hibernate.annotations.Columns(columns = {
        @Column(name = "BUYNOWPRICE_AMOUNT"),
        @Column(name = "BUYNOWPRICE_CURRENCY", length = 3)
    })
    private MonetaryAmount buyNowPrice;

    @NotNull
    @org.hibernate.annotations.Type(
        type = "monetary_amount_eur"
    )
    @org.hibernate.annotations.Columns(columns = {
        @Column(name = "INITIALPRICE_AMOUNT"),
        @Column(name = "INITIALPRICE_CURRENCY", length = 3)
    })
    private MonetaryAmount initialPrice;
    // ...
}

```

If `UserType` transforms values for only a single column, we don't need an `@Column` annotation. `MonetaryAmountUserType`, however, accesses two columns, so we need to explicitly declare two columns in the property mapping. Because JPA doesn't support multiple `@Column` annotations on a single property, we have to group them with the proprietary `@org.hibernate.annotations.Columns` annotation. Note that the order of the annotations is now important! Re-check the code for `MonetaryAmountUserType`; many operations rely on indexed access of arrays. The order when accessing `PreparedStatement` or `ResultSet` is the same as that of the declared columns in the mapping. Also, note that the number of columns isn't relevant for the choice of `UserType` versus `CompositeUserType`—only the desire to expose value type properties for queries.

We'll have to change the testing code we wrote. We'll add this line to set the `Item`:

```
item.setBuyNowPrice(new MonetaryAmount(BigDecimal.valueOf(1.1),
                                         Currency.getInstance("USD")));
```

We'll replace this line:

```
() -> assertEquals("1.1 USD",
                     items.get(0).getBuyNowPrice().toString())
```

with this one:

```
() -> assertEquals("2.20 USD",
                     items.get(0).getBuyNowPrice().toString())
```

We'll also replace this line:

```
(() -> assertEquals(new BigDecimal("1.00"),
                     items.get(0).getInitialPrice()))
```

with this one:

```
(() -> assertEquals("2.00 EUR",
                     items.get(0).getInitialPrice().toString()))
```

This is because the `convert` method from the `MonetaryAmountUserType` class doubles the value of the amount (listing 6.19, #U).

The source code of the book contains the tests using Spring Data JPA, Hibernate, and JPA.

With `MonetaryAmountUserType`, we've extended the buffer between the Java domain model and the SQL database schema. Both representations are now more robust to changes, and we can handle even rather eccentric requirements without modifying the essence of the domain model classes.

6.4 Summary

- We analyzed the mapping of basic and embedded properties of an entity class.
- We demonstrated how to override basic mappings, how to change the name of a mapped column, how to use derived, default, temporal, and enumeration properties, and how to test them.
- We implemented embeddable component classes and created fine-grained domain models.
- We mapped the properties of several Java classes in a composition, such as `Address` and `City`, to one entity table.
- We examined how Hibernate selects Java to SQL type converters and what types are built into Hibernate.
- We wrote a custom type converter for the `MonetaryAmount` class with the standard JPA extension interfaces and then a low-level adapter with the native Hibernate `UserType` API.

7

Mapping inheritance

This chapter covers

- Examining inheritance-mapping strategies
- Investigating polymorphic associations

We deliberately haven't talked much about inheritance mapping so far. Mapping makes the connection between the object-oriented world and the relational world. Inheritance is specific only to object-oriented systems. Consequently, mapping a hierarchy of classes to tables can be a complex issue, and we demonstrate various strategies in this chapter.

A basic strategy for mapping classes to database tables might be "one table for every persistent entity class". This approach sounds simple enough and indeed works well until we encounter inheritance.

Inheritance is such a visible structural mismatch between the object-oriented and relational worlds because the object-oriented systems model provides both *is a* and *has a* relationships. SQL-based models provide only *has a* relationships; SQL database management systems don't support type inheritance—and even when it's available, it's usually proprietary or incomplete.

There are four different strategies for representing an inheritance hierarchy:

- Use one table per concrete class and default runtime polymorphic behavior.
- Use one table per concrete class but discard polymorphism and inheritance relationships completely from the SQL schema. Use SQL UNION queries for runtime polymorphic behavior.
- Use one table per class hierarchy: enable polymorphism by denormalizing the SQL schema and relying on row-based discrimination to determine super/subtypes.
- Use one table per subclass: represent *is a* (inheritance) relationships as *has a* (foreign key) relationships, and use SQL JOIN operations.

This chapter takes a top-down approach, assuming that we're starting with a domain model and trying to derive a new SQL schema. The mapping strategies described are just as relevant if we're working bottom-up, starting with existing database tables. We examine some tricks along the way to help you deal with imperfect table layouts.

7.1 Table per concrete class with implicit polymorphism

We are working on the CaveatEmptor application, to implement persistence for a hierarchy of classes. We may stick with the simplest approach suggested: exactly one table for each concrete class. We can map all properties of a class, including inherited properties, to columns of this table, as shown in figure 7.1. To be able to execute the examples from the source code, you need first to run the Ch07.sql script.

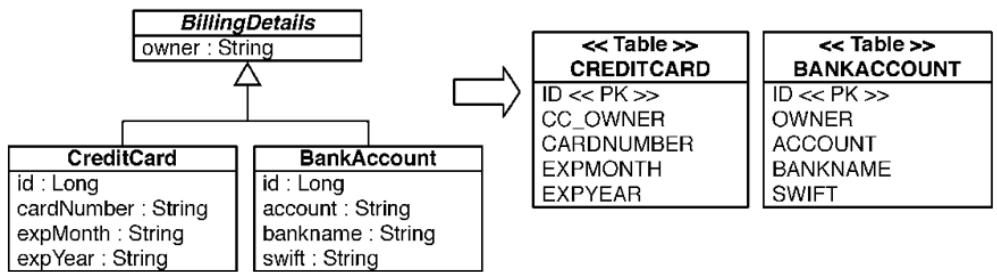


Figure 7.1 Mapping all concrete classes to an independent table

Relying on this implicit polymorphism, we map concrete classes with `@Entity`, as usual. By default, properties of the superclass are ignored and not persistent! We have to annotate the superclass with `@MappedSuperclass` to enable embedding of its properties in the concrete subclass tables; see the following listing, to be found in the `mapping-inheritance-mappedsuperclass` folder.

Listing 7.1 Mapping BillingDetails (abstract superclass) with implicit polymorphism

```
Path: Ch07/mapping-inheritance-
      mappedsuperclass/src/main/java/com/manning/javapersistence/ch07/model/BillingDetails
      .java

@MappedSuperclass
public abstract class BillingDetails {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;

    @NotNull
    private String owner;
    // ...
}
```

Now map the concrete subclasses.

Listing 7.2 Mapping CreditCard (concrete subclass)

```
Path: Ch07/mapping-inheritance-
      mappedsuperclass/src/main/java/com/manning/javapersistence/ch07/model/CreditCard.jav
      a

@Entity
@Override(
    name = "owner",
    column = @Column(name = "CC_OWNER", nullable = false))
public class CreditCard extends BillingDetails {

    @NotNull
    private String cardNumber;

    @NotNull
    private String expMonth;

    @NotNull
    private String expYear;

    // ...
}
```

We can override column mappings from the superclass in a subclass with the `@AttributeOverride` annotation. Starting with JPA 2.2, we may use several `@AttributeOverride` annotations on the same class. Up to JPA 2.1, we had to group the `@AttributeOverride` annotations within an `@AttributeOverrides` annotation. The previous example renamed the `OWNER` column to `CC_OWNER` in the `CREDITCARD` table.

The following listing also shows the mapping of the `BankAccount` subclass.

Listing 7.3 Mapping BankAccount (concrete subclass)

```
Path: Ch07/mapping-inheritance-
      mappedsuperclass/src/main/java/com/manning/javapersistence/ch07/model/BankAccount.ja
      va

@Entity
public class BankAccount extends BillingDetails {

    @NotNull
    private String account;

    @NotNull
    private String bankname;

    @NotNull
    private String swift;
// ...
}
```

We can declare the identifier property in the superclass, with a shared column name and generator strategy for all subclasses (as in the example above), or we can repeat it inside each concrete class.

To work with these classes, we'll create three Spring Data JPA repository interfaces.

Listing 7.4 The BillingDetailsRepository interface

```
Path: Ch07/mapping-inheritance-
      mappedsuperclass/src/main/java/com/manning/javapersistence/ch07/repositories/Billing
      DetailsRepository.java

@NoArgsConstructor
public interface BillingDetailsRepository<T extends BillingDetails, ID>
    extends JpaRepository<T, ID> {
    List<T> findByOwner(String owner);
}
```

In the listing above, the `BillingDetailsRepository` interface is annotated as `@NoArgsConstructor`. This prevents its instantiation as a Spring Data JPA repository instance. This is necessary as, following the schema from figure 7.1, there will be no `BILLINGDETAILS` table. However, the `BillingDetailsRepository` interface intends to be extended by the repository interfaces to deal with the `CreditCard` and `BankAccount` subclasses. That is why `BillingDetailsRepository` is generified by a `T` that extends `BillingDetails`. Additionally, it contains the `findByOwner` method – the `owner` field from `BillingDetails` will be included both in the `CREDITCARD` and in the `BANKACCOUNT` tables.

We'll create two more Spring Data repository interfaces.

Listing 7.5 The BankAccountRepository interface

```
Path: Ch07/mapping-inheritance-
      mappedsuperclass/src/main/java/com/manning/javapersistence/ch07/repositories/BankAcc
      ountRepository.java

public interface BankAccountRepository
    extends BillingDetailsRepository<BankAccount, Long> {
    List<BankAccount> findBySwift(String swift);
}
```

The `BankAccountRepository` interface extends `BillingDetailsRepository`, generified by `BankAccount` (as it deals with `BankAccount` instances) and by `Long` (as the id of the class if of this type). It adds the `findBySwift` method, whose name is following the Spring Data JPA conventions (see chapter 4).

Listing 7.6 The CreditCardRepository interface

```
Path: Ch07/mapping-inheritance-
      mappedsuperclass/src/main/java/com/manning/javapersistence/ch07/repositories/CreditC
      ardRepository.java

public interface CreditCardRepository
    extends BillingDetailsRepository<CreditCard, Long> {
    List<CreditCard> findByExpYear(String expYear);
}
```

The `CreditCardRepository` interface extends `BillingDetailsRepository`, generified by `CreditCard` (as it deals with `CreditCard` instances) and by `Long` (as the id of the class if of this type). It adds the `findByExpYear` method, whose name is following the Spring Data JPA conventions (see chapter 4).

We'll create the following test to check the functionality of the persistence code.

Listing 7.7 Testing the functionality of the persistence code

```

Path: Ch07/mapping-inheritance-
      mappedsuperclass/src/test/java/com/manning/javapersistence/ch07/
      MappingInheritanceSpringDataJPATest.java

@ExtendWith(SpringExtension.class)                                     #A
@ContextConfiguration(classes = {SpringDataConfiguration.class})       #B
public class MappingInheritanceSpringDataJPATest {

    @Autowired                                                       #C
    private CreditCardRepository creditCardRepository;                 #C

    @Autowired                                                       #D
    private BankAccountRepository bankAccountRepository;               #D

    @Test
    void storeLoadEntities() {

        CreditCard creditCard = new CreditCard(                                #E
            "John Smith", "123456789", "10", "2030");                      #E
        creditCardRepository.save(creditCard);                                #E

        BankAccount bankAccount = new BankAccount(                            #F
            "Mike Johnson", "12345", "Delta Bank", "BANKXY12");           #F
        bankAccountRepository.save(bankAccount);                            #F

        List<CreditCard> creditCards =                               #G
            creditCardRepository.findByOwner("John Smith");                #G
        List<BankAccount> bankAccounts =                               #H
            bankAccountRepository.findByOwner("Mike Johnson");             #H
        List<CreditCard> creditCards2 =                               #I
            creditCardRepository.findByExpYear("2030");                  #I
        List<BankAccount> bankAccounts2 =                               #J
            bankAccountRepository.findBySwift("BANKXY12");                #J

        assertEquals(1, creditCards.size());                                #K
        assertEquals("123456789", creditCards.get(0).getCardNumber());      #L
        assertEquals(1, bankAccounts.size());                                #M
        assertEquals("12345", bankAccounts.get(0).getAccount());           #N
        assertEquals(1, creditCards2.size());                                #O
        assertEquals("John Smith", creditCards2.get(0).getOwner());         #P
        assertEquals(1, bankAccounts2.size());                                #Q
        assertEquals("Mike Johnson", bankAccounts2.get(0).getOwner());      #R
    }

}

```

#A We extend the test using `SpringExtension`. This extension is used to integrate the Spring test context with the JUnit 5 Jupiter test.

#B The Spring test context is configured using the beans defined in the `SpringDataConfiguration` class.

#C A `CreditCardRepository` bean is injected by Spring through auto-wiring.

#D A BankAccountRepository bean is injected by Spring through auto-wiring. This is possible as the com.manning.javapersistence.ch07.repositories package where CreditCardRepository and BankAccountRepository are located was used as the argument of the @EnableJpaRepositories annotation on the SpringDataConfiguration class. To revisit how the SpringDataConfiguration class looks like, you may go back to chapter 2.

#E We create a credit card and save it to the repository.

#F We create a bank account and save it to the repository.

#G We get the list of all credit cards having John Smith as owner.

#H We get the list of all bank accounts having Mike Johnson as owner.

#I We get the credit cards expiring in 2030.

#J We get the bank accounts with SWIFT BANKXY12.

#K We check the size of the list of credit cards.

#L We get the number of the first credit card in the list.

#M We check the size of the list of bank accounts.

#N We check the number of the first bank account in the list.

#O We check the size of the list of credit cards expiring in 2030.

#P We check the owner of the first credit card in this list.

#Q We check the size of the list of bank accounts with SWIFT BANKXY12.

#R We check the owner of the first bank account in this list.

The source code of this chapter also demonstrates how to test these classes using JPA and Hibernate.

The main problem with implicit inheritance mapping is that it doesn't support polymorphic associations very well. In the database, we usually represent associations as foreign key relationships. In the schema from figure 7.1, if the subclasses are all mapped to different tables, a polymorphic association to their superclass (`abstract BillingDetails`) can't be represented as a simple foreign key relationship. We can't have another entity mapped with a foreign key "referencing BILLINGDETAILS"—there is no such table. This would be problematic in the domain model because `BillingDetails` is associated with `User`; both the `CREDITCARD` and `BANKACCOUNT` tables would need a foreign key reference to the `USERS` table. None of these issues can be easily resolved, so we should consider an alternative mapping strategy.

Polymorphic queries that return instances of all classes that match the interface of the queried class are also problematic. Hibernate must execute a query against the superclass as several SQL SELECTs, one for each concrete subclass. The JPA query `select bd from BillingDetails bd` requires two SQL statements:

```
select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT
from
    BANKACCOUNT
select
    ID, CC_OWNER, CARDNUMBER, EXPMONTH, EXPYEAR
from
    CREDITCARD
```

Hibernate or Spring Data JPA using Hibernate use a separate SQL query for each concrete subclass. On the other hand, queries against the concrete classes are trivial and perform well—Hibernate uses only one of the statements.

A further conceptual problem with this mapping strategy is that several different columns of different tables share exactly the same semantics. This makes schema evolution more

complex. For example, renaming or changing the type of a superclass property results in changes to multiple columns in multiple tables. Many of the standard refactoring operations offered by your IDE would require manual adjustments because the automatic procedures usually don't count for things like `@AttributeOverride` or `@AttributeOverrides`. It also makes it much more difficult to implement database integrity constraints that apply to all subclasses.

We recommend this approach (only) for the top level of your class hierarchy, where polymorphism isn't usually required and when modification of the superclass in the future is unlikely. This may happen for particular domain models that you may face in your real-life applications. However, it isn't a good fit for the `CaveatEmptor` domain model, where queries and other entities refer to `BillingDetails` – so we'll look for other alternatives.

With the help of the SQL `UNION` operation, we can eliminate most of the issues with polymorphic queries and associations.

7.2 Table per concrete class with unions

First, we consider a union subclass mapping with `BillingDetails` as an abstract class (or interface), as in the previous section. In this situation, there are again two tables and superclass columns that are duplicated in both: `CREDITCARD` and `BANKACCOUNT`. What's new is an inheritance strategy known as `TABLE_PER_CLASS`, declared on the superclass, as shown next. The source code is to be found in the `mapping-inheritance-tableperclass` folder.

Listing 7.8 Mapping `BillingDetails` with `TABLE_PER_CLASS`

```
Path: Ch07/mapping-inheritance-
      tableperclass/src/main/java/com/manning/javapersistence/ch07/model/BillingDetails.java

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;
    @NotNull
    private String owner;
    // ...
}
```

The database identifier and its mapping have to be present in the superclass to share it in all subclasses and their tables. This is no longer optional, as it was for the previous mapping strategy. The `CREDITCARD` and `BANKACCOUNT` tables both have an `ID` primary key column. All concrete class mappings inherit persistent properties from the superclass (or interface). An `@Entity` annotation on each subclass is all that is required.

Listing 7.9 Mapping CreditCard

```
Path: Ch07/mapping-inheritance-
      tableperclass/src/main/java/com/manning/javapersistence/ch07/model/CreditCard.java

@Entity
public class CreditCard extends BillingDetails {
    @NotNull
    private String cardNumber;
    @NotNull
    private String expMonth;
    @NotNull
    private String expYear;
    // ...
}
```

Listing 7.10 Mapping BankAccount

```
Path: Ch07/mapping-inheritance-
      tableperclass/src/main/java/com/manning/javapersistence/ch07/model/BankAccount.java

@Entity
public class BankAccount extends BillingDetails {
    @NotNull
    private String account;
    @NotNull
    private String bankName;
    @NotNull
    private String swift;
    // ...
}
```

We'll have to change the `BillingDetailsRepository` interface and remove the `@NoRepositoryBean` annotation. This change, together with the fact that the `BillingDetails` class is now annotated as `@Entity`, will allow the usage of this repository to interact with the database. This is how the `BillingDetailsRepository` interface looks like now.

Listing 7.11 The BillingDetailsRepository interface

```
Path: Ch07/mapping-inheritance-
      tableperclass/src/main/java/com/manning/javapersistence/ch07/model/BankAccount.java

public interface BillingDetailsRepository<T extends BillingDetails, ID>
    extends JpaRepository<T, ID> {
    List<T> findByOwner(String owner);
}
```

Keep in mind that the SQL schema still isn't aware of the inheritance; the tables look exactly alike, as shown in figure 7.1.

Note that the JPA standard specifies that `TABLE_PER_CLASS` is optional, so not all JPA implementations may support it.

If `BillingDetails` were concrete, we'd need an additional table to hold instances. We have to emphasize again that there is still no relationship between the database tables, except for the fact that they have some (many) similar columns.

The advantages of this mapping strategy are clearer if we examine polymorphic queries.

We may use the Spring Data JPA `BillingDetailsRepository` interface to query the database, like this:

```
billingDetailsRepository.findAll();
```

Or, from JPA or Hibernate, we may execute the following query:

```
select bd from BillingDetails bd
```

Both approaches will generate the following SQL statement:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, CLAZZ_
from
    ( select
        ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
        null as ACCOUNT,
        null as BANKNAME,
        null as SWIFT,
        1 as CLAZZ_
    from
        CREDITCARD
    union all
    select
        id, OWNER,
        null as EXPMONTH,
        null as EXPYEAR,
        null as CARDNUMBER,
        ACCOUNT, BANKNAME, SWIFT,
        2 as CLAZZ_
    from
        BANKACCOUNT
    ) as BILLINGDETAILS
```

This SELECT uses a FROM-clause subquery to retrieve all instances of `BillingDetails` from all concrete class tables. The tables are combined with a UNION operator, and a literal (in this case, 1 and 2) is inserted into the intermediate result; Hibernate reads this to instantiate the correct class given the data from a particular row. A union requires that the queries that are combined project over the same columns; hence, you have to pad and fill nonexistent columns with NULL. You may ask whether this query will really perform better than two separate statements. Here you can let the database optimizer find the best execution plan to combine rows from several tables instead of merging two result sets in memory as Hibernate's polymorphic loader engine would do.

Another much more important advantage is the ability to handle polymorphic associations; for example, an association mapping from `User` to `BillingDetails` would now be possible. Hibernate can use a UNION query to simulate a single table as the target of the association mapping. We cover this topic in detail later in this chapter.

The examples from this paragraph, together with tests using Spring Data JPA, JPA, and Hibernate, are to be found in the `mapping-inheritance-tableperclass` folder of the source code of this book.

So far, the inheritance-mapping strategies we've examined don't require extra consideration concerning the SQL schema. This situation changes with the next strategy.

7.3 Table per class hierarchy

We can map an entire class hierarchy to a single table. This table includes columns for all properties of all classes in the hierarchy. The value of an extra type discriminator column or formula identifies the concrete subclass represented by a particular row. Figure 7.2 shows this approach. The source code to follow is to be found in the `mapping-inheritance-singletable` folder.

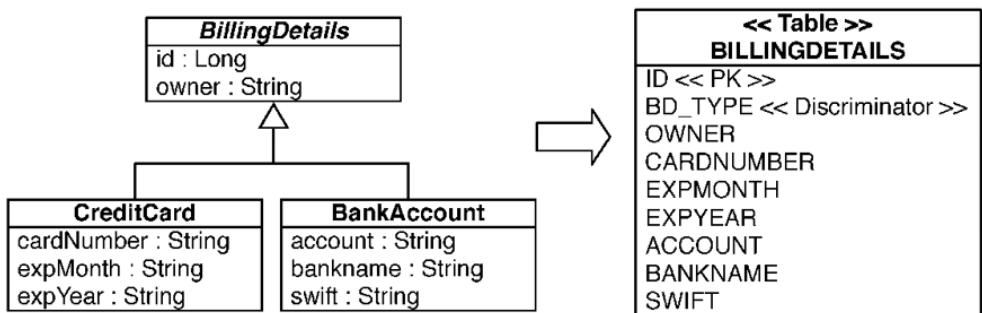


Figure 7.2 Mapping an entire class hierarchy to a single table

This mapping strategy is a winner in terms of both performance and simplicity. It's the best-performing way to represent polymorphism—both polymorphic and non-polymorphic queries perform well—and it's even easy to write queries by hand. Ad hoc reporting is possible without complex joins or unions. Schema evolution is straightforward.

There is one major problem: data integrity. We must declare columns for properties declared by subclasses to be nullable. If the subclasses each define several non-nullable properties, the loss of `NOT NULL` constraints may be a serious problem from the point of view of data correctness. Imagine that an expiration date for credit cards is required, but the database schema can't enforce this rule because all columns of the table can be `NULL`. A simple application programming error can lead to invalid data.

Another important issue is normalization. We've created functional dependencies between non-key columns, violating the third normal form. As always, denormalization for performance reasons can be misleading because it sacrifices long-term stability, maintainability, and the integrity of data for immediate gains that may also be achieved by proper optimization of the SQL execution plans (in other words, ask the DBA).

We use the `SINGLE_TABLE` inheritance strategy to create a table-per-class hierarchy mapping, as shown in the following listing.

Listing 7.12 Mapping `BillingDetails` with `SINGLE_TABLE`

```
Path: Ch07/mapping-inheritance-
singletable/src/main/java/com/manning/javapersistence/ch07/model/BillingDetails.java

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "BD_TYPE")
public abstract class BillingDetails {

    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;

    @NotNull
    @Column(nullable = false)
    private String owner;

    // ...
}
```

The root class `BillingDetails` of the inheritance hierarchy is mapped to the table `BILLINGDETAILS` automatically. Shared properties of the superclass can be NOT NULL in the schema; every subclass instance must have a value. An implementation quirk of Hibernate requires that we declare nullability with `@Column` because Hibernate ignores Bean Validation's `@NotNull` when it generates the database schema.

We have to add a special discriminator column to distinguish what each row represents. This isn't a property of the entity; it's used internally by Hibernate. The column name is `BD_TYPE`, and the values are strings—in this case, "`CC`" or "`BA`". Hibernate or Spring Data JPA using Hibernate automatically sets and retrieves the discriminator values.

If we don't specify a discriminator column in the superclass, its name defaults to `DTYPE`, and the values are strings. All concrete classes in the inheritance hierarchy can have a discriminator value, such as `CreditCard`.

Listing 7.13 Mapping CreditCard using the SINGLE_TABLE inheritance strategy

```
Path: Ch07/mapping-inheritance-
singletable/src/main/java/com/manning/javapersistence/ch07/model/CreditCard.java

@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
    @NotNull
    private String cardNumber;
    @NotNull
    private String expMonth;
    @NotNull
    private String expYear;
    // ...
}
```

Without an explicit discriminator value, Hibernate defaults to the fully qualified class name if we use Hibernate XML files and the simple entity name if we use annotations or JPA XML files. Note that JPA doesn't specify a default for non-string discriminator types; each persistence provider can have different defaults. Therefore, we should always specify discriminator values for the concrete classes.

Annotate every subclass with `@Entity`, and then map properties of a subclass to columns in the `BILLINGDETAILS` table. Remember that `NOT NULL` constraints aren't allowed in the schema because a `BankAccount` instance won't have an `expMonth` property, and the `EXPMONTH` column must be `NULL` for that row. Hibernate and Spring Data JPA using Hibernate ignore the `@NotNull` for schema DDL generation, but they observe it at runtime before inserting a row. This helps us avoid programming errors; we don't want to accidentally save credit card data without its expiration date. (Other, less well-behaved applications can, of course, still store incorrect data in this database.)

We may use the Spring Data JPA `BillingDetailsRepository` interface to query the database, like this:

```
billingDetailsRepository.findAll();
```

Or, from JPA or Hibernate, we may execute the following query:

```
select bd from BillingDetails bd
```

Both approaches will generate the following SQL statement:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, BD_TYPE
from
    BILLINGDETAILS
```

To query the `CreditCard` subclass, we also have alternatives.

We may use the Spring Data JPA `CreditCardRepository` interface to query the database, like this:

```
creditCardRepository.findAll();
```

Or, from JPA or Hibernate, we may execute the following query:

```
select cc from CreditCard cc
```

Hibernate adds a restriction on the discriminator column:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
from
    BILLINGDETAILS
where
    BD_TYPE='CC'
```

Sometimes, especially in legacy schemas, we don't have the freedom to include an extra discriminator column in the entity tables. In this case, we can apply an expression to calculate a discriminator value for each row. Formulas for discrimination aren't part of the JPA specification, but Hibernate has an extension annotation, `@DiscriminatorFormula`. The source code to follow is to be found in the `mapping-inheritance-singletableformula` folder.

Listing 7.14 Mapping BillingDetails with a @DiscriminatorFormula

```
Path: Ch07/mapping-inheritance-
      singletableformula/src/main/java/com/manning/javapersistence/ch07/model/BillingDetai
ls.java

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@org.hibernate.annotations.DiscriminatorFormula(
    "case when CARDNUMBER is not null then 'CC' else 'BA' end"
)
public abstract class BillingDetails {
    // ...
}
```

There is no discriminator column in the schema, so this mapping relies on an SQL `CASE/WHEN` expression to determine whether a particular row represents a credit card or a bank account (many developers have never used this kind of SQL expression; check the ANSI standard if you aren't familiar with it). The result of the expression is a literal, `CC` or `BA`, declared on the subclass mappings.

The disadvantages of the table-per-class hierarchy strategy may be too serious for the design—considering denormalized schemas can become a major burden in the long term. Your DBA may not like it at all. The next inheritance-mapping strategy doesn't expose you to this problem.

7.4 Table per subclass with joins

The fourth option is to represent inheritance relationships as SQL foreign key associations. Every class/subclass that declares persistent properties—including abstract classes and even interfaces—has its own table. The source code to follow is to be found in the `mapping-inheritance-joined` folder.

Unlike the table-per-concrete-class strategy we mapped first, the table of a concrete `@Entity` here contains columns only for each non-inherited property, declared by the subclass itself, along with a primary key that is also a foreign key of the superclass table. This is easier than it sounds; have a look at figure 7.3.

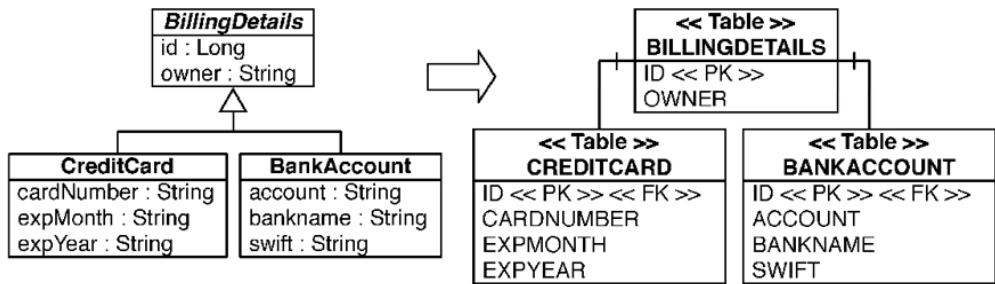


Figure 7.3 Mapping all classes of the hierarchy to their own table

If we make an instance of the `CreditCard` subclass persistent, Hibernate inserts two rows. The values of properties declared by the `BillingDetails` superclass are stored in a new row of the `BILLINGDETAILS` table. Only the values of properties declared by the subclass are stored in a new row of the `CREDITCARD` table. The primary key shared by the two rows links them together. Later, the subclass instance may be retrieved from the database by joining the subclass table with the superclass table.

The primary advantage of this strategy is that it normalizes the SQL schema. Schema evolution and integrity-constraint definition are straightforward. A foreign key referencing the table of a particular subclass may represent a polymorphic association to that particular subclass. We'll use the `JOINED` inheritance strategy to create a table-per-subclass hierarchy mapping.

Listing 7.15 Mapping BillingDetails with JOINED

```
Path: Ch07/mapping-inheritance-
joined/src/main/java/com/manning/javapersistence/ch07/model/BillingDetails.java

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;
    @NotNull
    private String owner;

    // ...
}
```

The root class `BillingDetails` is mapped to the table `BILLINGDETAILS`. Note that no discriminator is required with this strategy.

In subclasses, we don't need to specify the join column if the primary key column of the subclass table has (or is supposed to have) the same name as the primary key column of the superclass table.

Listing 7.16 Mapping BankAccount (concrete class)

```
Path: Ch07/mapping-inheritance-
joined/src/main/java/com/manning/javapersistence/ch07/model/BankAccount.java

@Entity
public class BankAccount extends BillingDetails {

    @NotNull
    private String account;

    @NotNull
    private String bankname;

    @NotNull
    private String swift;

    // ...
}
```

This entity has no identifier property; it automatically inherits the `ID` property and column from the superclass, and Hibernate knows how to join the tables if we want to retrieve instances of `BankAccount`. Of course, we can specify the column name explicitly.

Listing 7.17 Mapping CreditCard

```
Path: Ch07/mapping-inheritance-
joined/src/main/java/com/manning/javapersistence/ch07/model/CreditCard.java

@Entity
@PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
public class CreditCard extends BillingDetails {
    @NotNull
    private String cardNumber;
    @NotNull
    private String expMonth;
    @NotNull
    private String expYear;
    // ...
}
```

The primary key columns of the `BANKACCOUNT` and `CREDITCARD` tables each also have a foreign key constraint referencing the primary key of the `BILLINGDETAILS` table.

We may use the Spring Data JPA `BillingDetailsRepository` interface to query the database, like this:

```
billingDetailsRepository.findAll();
```

Or, from JPA or Hibernate, we may execute the following query:

```
select bd from BillingDetails bd
```

Hibernate relies on an SQL outer join and will generate the following:

```
select
    BD.ID, BD.OWNER,
    CC.EXPMONTH, CC.EXPYEAR, CC.CARDNUMBER,
    BA.ACCOUNT, BA.BANKNAME, BA.SWIFT,
    case
        when CC.CREDITCARD_ID is not null then 1
        when BA.ID is not null then 2
        when BD.ID is not null then 0
    end
from
    BILLINGDETAILS BD
    left outer join CREDITCARD CC on BD.ID=CC.CREDITCARD_ID
    left outer join BANKACCOUNT BA on BD.ID=BA.ID
```

The SQL `CASE ... WHEN` clause detects the existence (or absence) of rows in the subclass tables `CREDITCARD` and `BANKACCOUNT`, so Hibernate or Spring Data using Hibernate can determine the concrete subclass for a particular row of the `BILLINGDETAILS` table.

For a narrow subclass query like

```
creditCardRepository.findAll();
```

or

```
select cc from CreditCard cc,
```

Hibernate uses an inner join:

```

select
    CREDITCARD_ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
from
    CREDITCARD
    inner join BILLINGDETAILS on CREDITCARD_ID=ID

```

As you can see, this mapping strategy is more difficult to implement by hand—even ad hoc reporting is more complex. This is an important consideration if you plan to mix Spring Data JPA or Hibernate code with handwritten SQL. A usual approach and a portable solution may be working with JPQL (Jakarta Persistence Query Language) and annotate methods with JPQL queries.

Furthermore, even though this mapping strategy is deceptively simple, our experience is that performance can be unacceptable for complex class hierarchies. Queries always require a join across many tables or many sequential reads.

Inheritance with joins and discriminator

Hibernate doesn't need a special discriminator database column to implement the `InheritanceType.JOINED` strategy and the JPA specification doesn't contain any requirements either. The `CASE ... WHEN` clause in the SQL `SELECT` statement is a smart way to distinguish the entity type of each retrieved row. Some JPA examples you might find elsewhere, however, use `InheritanceType.JOINED` and a `@DiscriminatorColumn` mapping. Apparently, some other JPA providers don't use `CASE ... WHEN` clauses and rely only on a discriminator value, even for the `InheritanceType.JOINED` strategy. Hibernate doesn't need the discriminator but uses a declared `@DiscriminatorColumn`, even with a `JOINED` mapping strategy. If you prefer to ignore the discriminator mapping with `JOINED` (it was ignored in older Hibernate versions), enable the configuration property `hibernate.discriminator.ignore_explicit_for_joined`.

Before we analyze when to choose which strategy, let's consider mixing inheritance-mapping strategies in a single class hierarchy.

7.5 Mixing inheritance strategies

We can map an entire inheritance hierarchy with the `TABLE_PER_CLASS`, `SINGLE_TABLE`, or `JOINED` strategy. We can't mix them—for example, to switch from a table-per-class hierarchy with a discriminator to a normalized table-per-subclass strategy. Once we've decided on an inheritance strategy, we have to stick with it.

This isn't completely true, however. By using some tricks, we can switch the mapping strategy for a particular subclass. For example, we can map a class hierarchy to a single table, but, for a particular subclass, switch to a separate table with a foreign key-mapping strategy, just as with table-per-subclass. Look at the schema in figure 7.4. The source code to follow is to be found in the `mapping-inheritance-mixed` folder.

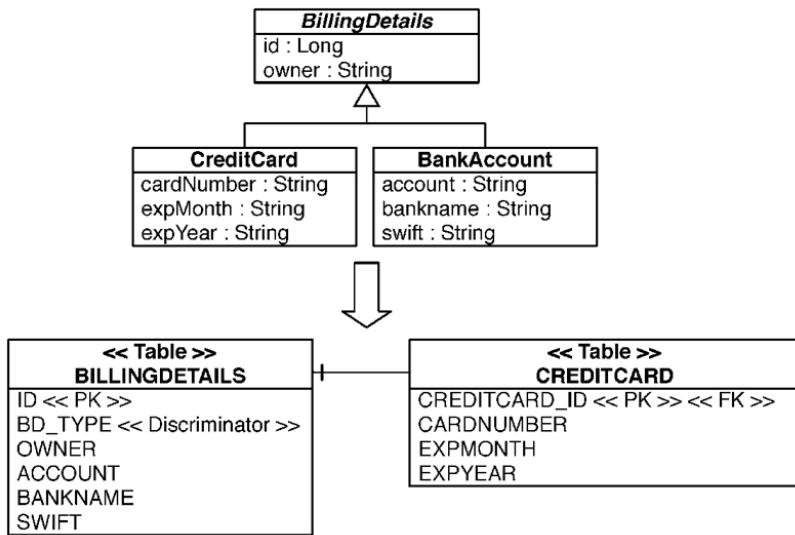


Figure 7.4 Breaking out a subclass to its own secondary table

We'll map the superclass `BillingDetails` with `InheritanceType.SINGLE_TABLE`, as we did before. Then we'll map the subclass we want to break out of the single table to a secondary table.

Listing 7.18 Mapping CreditCard

```

Path: Ch07/mapping-inheritance-
      mixed/src/main/java/com/manning/javapersistence/ch07/model/CreditCard.java

@Entity
@DiscriminatorValue("CC")
@SecondaryTable(
    name = "CREDITCARD",
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
)
public class CreditCard extends BillingDetails {
    @NotNull
    @Column(table = "CREDITCARD", nullable = false)
    private String cardNumber;

    @Column(table = "CREDITCARD", nullable = false)
    private String expMonth;

    @Column(table = "CREDITCARD", nullable = false)
    private String expYear;
    // ...
}
  
```

The `@SecondaryTable` and `@Column` annotations group some properties and tell Hibernate to get them from a secondary table. We map all properties that we moved into the secondary table with the name of that secondary table. This is done with the `table` parameter of `@Column`, which we haven't shown before. This mapping has many uses, and you'll see it again later in this book. In this example, it separates the `CreditCard` properties from the single table strategy into the `CREDITCARD` table. It would be a viable solution if we'll add a new class to extend `BillingDetails`, `Paypal` for example.

The `CREDITCARD_ID` column of this table is at the same time the primary key, and it has a foreign key constraint referencing the `ID` of the single hierarchy table. If we don't specify a primary key join column for the secondary table, the name of the primary key of the single inheritance table is used—in this case, `ID`.

Remember that `InheritanceType.SINGLE_TABLE` enforces all columns of subclasses to be nullable. One of the benefits of this mapping is that we can now declare columns of the `CREDITCARD` table as `NOT NULL`, guaranteeing data integrity.

At runtime, Hibernate executes an outer join to fetch `BillingDetails` and all subclass instances polymorphically:

```
select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT,
    EXPMONTH, EXPYEAR, CARDNUMBER,
    BD_TYPE
from
    BILLINGDETAILS
    left outer join CREDITCARD on ID=CREDITCARD_ID
```

We can also use this trick for other subclasses in the class hierarchy. For an exceptionally wide class hierarchy, the outer join can become a problem. Some database systems (Oracle, for example) limit the number of tables in an outer join operation. For a wide hierarchy, you may want to switch to a different fetching strategy that executes an immediate second SQL select instead of an outer join.

7.6 Inheritance of embeddable classes

An embeddable class is a component of its owning entity; hence, the normal entity inheritance rules presented in this chapter don't apply. As a Hibernate extension, we can map an embeddable class that inherits some persistent properties from a superclass (or interface). Consider these two new attributes of an auction item: dimensions and weight.

An item's dimensions are its width, height, and depth, expressed in a given unit and its symbol: for example, inches ("") or centimeters (cm). An item's weight also carries a unit of measurement: for example, pounds (lbs) or kilograms (kg). To capture the common attributes (name and symbol) of measurement, we define a superclass for `Dimension` and `Weight` called `Measurement`. The source code to follow is to be found in the `mapping-inheritance-embeddable` folder.

Listing 7.19 Mapping the Measurement abstract embeddable superclass

```
Path: Ch07/mapping-inheritance-
      embeddable/src/main/java/com/manning/javapersistence/ch07/model/Measurement.java

@MappedSuperclass
public abstract class Measurement {
    @NotNull
    private String name;
    @NotNull
    private String symbol;
    // ...
}
```

We use the `@MappedSuperclass` annotation on the superclass of the embeddable class we're mapping just like we would for an entity. Subclasses will inherit the properties of this class as persistent properties.

We define the `Dimensions` and `Weight` subclasses as `@Embeddable`. For `Dimensions`, we override all the superclass attributes and add a column-name prefix.

Listing 7.20 Mapping the Dimensions class

```
Path: Ch07/mapping-inheritance-
      embeddable/src/main/java/com/manning/javapersistence/ch07/model/Dimensions.java

@Embeddable
@AttributeOverride(name = "name",
    column = @Column(name = "DIMENSIONS_NAME"))
@AttributeOverride(name = "symbol",
    column = @Column(name = "DIMENSIONS_SYMBOL"))
public class Dimensions extends Measurement {
    @NotNull
    private BigDecimal depth;
    @NotNull
    private BigDecimal height;
    @NotNull
    private BigDecimal width;
    // ...
}
```

Without this override, an `Item` embedding both `Dimension` and `Weight` would map to a table with conflicting column names. Following is the `Weight` class; its mapping also overrides the column names with a prefix (for uniformity, we avoid the conflict with the previous override).

Listing 7.21 Mapping the Weight class

```
Path: Ch07/mapping-inheritance-
      embeddable/src/main/java/com/manning/javapersistence/ch07/model/Weight.java

@Embeddable
@AttributeOverride(name = "name",
                   column = @Column(name = "WEIGHT_NAME"))
@AttributeOverride(name = "symbol",
                   column = @Column(name = "WEIGHT_SYMBOL"))
public class Weight extends Measurement {
    @NotNull
    @Column(name = "WEIGHT")
    private BigDecimal value;
    // ...
}
The owning entity Item defines two regular persistent embedded properties.
```

Listing 7.22 Mapping the Item class

```
Path: Ch07/mapping-inheritance-
      embeddable/src/main/java/com/manning/javapersistence/ch07/model/Item.java

@Entity
public class Item {
    private Dimensions dimensions;
    private Weight weight;
    // ...
}
```

Figure 7.5 illustrates this mapping.

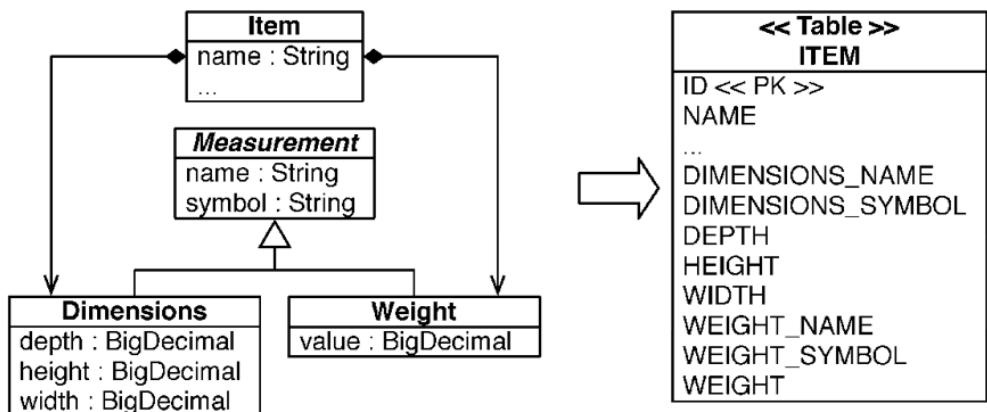


Figure 7.5 Mapping concrete embeddable classes with their inherited properties

Alternatively, we could override the conflicting `Measurement` column names of the embedded properties in the `Item` class, as demonstrated in section 6.2. Instead, we prefer to override them once in the `@Embeddable` classes, so *any* consumers of these classes don't have to resolve the conflict.

A pitfall to watch out for is embedding a property of abstract superclass type (like `Measurement`) in an entity (like `Item`). This can never work; the JPA provider doesn't know how to store and load `Measurement` instances polymorphically. It doesn't have the information necessary to decide whether the values in the database are `Dimension` or `Weight` instances because there is no discriminator. This means although we *can* have an `@Embeddable` class inherit some persistent properties from a `@MappedSuperclass`, the reference to an instance isn't polymorphic—it always names a concrete class.

Compare this with the alternative inheritance strategy for embeddable classes examined in the section "Converting properties of components" in chapter 6, which supported polymorphism but required some custom type-discrimination code.

Next, we provide more tips about how to choose an appropriate combination of mapping strategies for the application's class hierarchies.

7.7 Choosing a strategy

Picking the right inheritance-mapping strategy depends on the usage of the superclasses of the entity hierarchy. We have to consider how frequently we query for instances of the superclasses and whether we have associations targeting the superclasses. Another important aspect is the attributes of super- and subtypes: whether subtypes have many additional attributes or only different behavior than their supertypes. Here are some rules of thumb:

- If we don't require polymorphic associations or queries, lean toward table-per-concrete class—in other words, if we never or rarely select `bd` from `BillingDetails` `bd` and we have no class that has an association to `BillingDetails`. An explicit UNION-based mapping with `InheritanceType.TABLE_PER_CLASS` should be preferred because (optimized) polymorphic queries and associations will then be possible later.
- If we do require polymorphic associations (an association to a superclass, hence to all classes in the hierarchy with a dynamic resolution of the concrete class at runtime) or queries, and subclasses declare relatively few properties (particularly if the main difference between subclasses is in their behavior), lean toward `InheritanceType.SINGLE_TABLE`. The goal is to minimize the number of nullable columns and to convince ourselves (and the DBA) that a denormalized schema won't create problems in the long run.
- If we do require polymorphic associations or queries, and subclasses declare many (non-optional) properties (subclasses differ mainly by the data they hold), lean toward `InheritanceType.JOINED`. Alternatively, depending on the width and depth of the inheritance hierarchy and the possible cost of joins versus unions, use `InheritanceType.TABLE_PER_CLASS`. This decision might require the evaluation of SQL execution plans with real data.

By default, choose `InheritanceType.SINGLE_TABLE` only for simple problems. Otherwise, for complex cases, or when a data modeler insisting on the importance of `NOT NULL` constraints and normalization overrules us, we should consider the `Inheritance-Type.JOINED` strategy. At that point, we should ask ourselves whether it may not be better to remodel inheritance as delegation in the class model. Complex inheritance is often best avoided for all sorts of reasons unrelated to persistence or ORM. Hibernate acts as a buffer between the domain and relational models, but that doesn't mean we can ignore persistence concerns completely when designing the classes.

When we start thinking about mixing inheritance strategies, we must remember that implicit polymorphism in Hibernate is smart enough to handle exotic cases. Also, we must consider that we can't put inheritance annotations on interfaces; this isn't standardized in JPA.

For example, consider an additional interface in the example application: `ElectronicPaymentOption`. This is a business interface that doesn't have a persistence aspect—except that in the application, a persistent class such as `CreditCard` will likely implement this interface. No matter how we map the `BillingDetails` hierarchy, Hibernate can answer the query `select o from ElectronicPaymentOption o` correctly. This even works if other classes, which aren't part of the `BillingDetails` hierarchy, are mapped as persistent and implement this interface. Hibernate always knows what tables to query, which instances to construct, and how to return a polymorphic result.

We can apply all mapping strategies to abstract classes. Hibernate won't try to instantiate an abstract class, even if we query or load it.

We mentioned the relationship between `User` and `BillingDetails` several times and how it influences the selection of an inheritance-mapping strategy. In the following and last section of this chapter, we explore this more advanced topic in detail: polymorphic associations. If you don't have such a relationship in your model right now, you may want to read this material later, when you encounter the issue in your application.

7.8 Polymorphic associations

Polymorphism is a defining feature of object-oriented languages like Java. Support for polymorphic associations and polymorphic queries is a fundamental feature of an ORM solution like Hibernate. Surprisingly, we've managed to get this far without needing to talk much about polymorphism. Refreshingly, there isn't much to say on the topic—polymorphism is so easy to use in Hibernate that we don't need to expend a lot of effort explaining it.

To provide an overview, we first consider a *many-to-one* association to a class that may have subclasses and then a *one-to-many* relationship. For both examples, the classes of the domain model are the same; see figure 7.6.

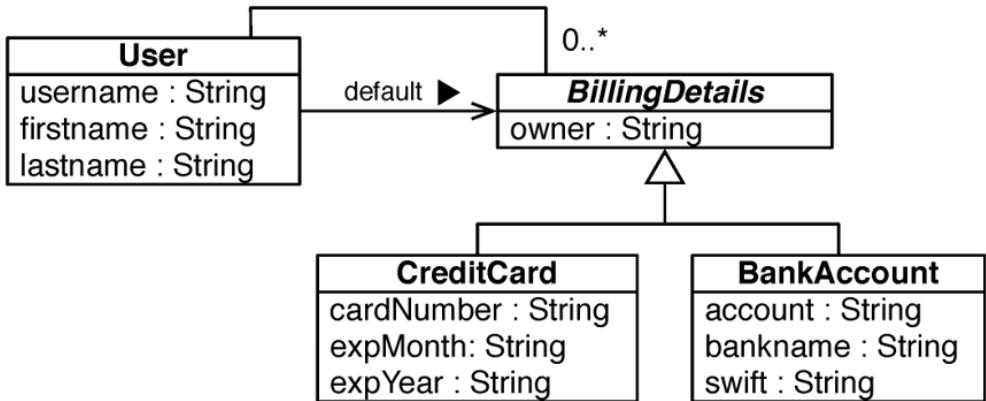


Figure 7.6 A user has either a credit card or a bank account as the default billing details.

7.8.1 Polymorphic many-to-one associations

First, consider the `defaultBilling` property of `User`. It references one particular `BillingDetails` instance, which at runtime can be any concrete instance of that class. The source code to follow is to be found in the `mapping-inheritance-manytoone` folder.

We map this unidirectional association to the abstract class `BillingDetails` as follows:

```

Path: Ch07/mapping-inheritance-
      manytoone/src/main/java/com/manning/javapersistence/ch07/model/User.java

@Entity
@Table(name = "USERS")
public class User {
    @ManyToOne
    private BillingDetails defaultBilling;
    // ...
}
  
```

The `USERS` table now has the join/foreign key column `DEFAULTBILLING_ID` representing this relationship. It's a nullable column because a `User` might not have a default billing method assigned. Because `BillingDetails` is abstract, the association must refer to an instance of one of its subclasses—`CreditCard` or `BankAccount`—at runtime.

We don't have to do anything special to enable polymorphic associations in Hibernate; if the target class of an association is mapped with `@Entity` and `@Inheritance`, the association is naturally polymorphic.

The following Spring Data JPA code demonstrates the creation of an association to an instance of the `CreditCard` subclass:

```
Path: Ch07/mapping-inheritance-manytoone/src/test/java/com/manning/javapersistence/ch07/
      MappingInheritanceSpringDataJPATest.java

CreditCard creditCard = new CreditCard(
    "John Smith", "123456789", "10", "2030"
);
User john = new User("John Smith");
john.setDefaultBilling(creditCard);
creditCardRepository.save(creditCard);
userRepository.save(john);
```

Now, when we navigate the association in a second unit of work, Hibernate automatically retrieves the `CreditCard` instance:

```
List<User> users = userRepository.findAll();
users.get(0).getDefaultBilling().pay(123);
```

The second line above will invoke the `pay` method from the concrete subclass of `BillingDetails`.

We can handle *one-to-one* associations the same way. What about plural associations, like the collection of `billingDetails` for each `User`? Let's look at that next.

7.8.2 Polymorphic collections

A `User` may have references to many `BillingDetails`, not only a single default (one of the many is the default; let's ignore that for now). We can map this with a bidirectional *one-to-many* association. The source code to follow is to be found in the `mapping-inheritance-onetomany` folder.

```
Path: Ch07/mapping-inheritance-
      onetomany/src/main/java/com/manning/javapersistence/ch07/model/User.java

@Entity
@Table(name = "USERS")
public class User {
    @OneToMany(mappedBy = "user")
    private Set<BillingDetails> billingDetails = new HashSet<>();
    // ...
}
```

Next, here's the owning side of the relationship (declared with `mappedBy` in the previous mapping). By owning side, we understand that side of the relationship that owns the foreign key in the database - `BillingDetails` in this example.

```
Path: Ch07/mapping-inheritance-
      onetomany/src/main/java/com/manning/javapersistence/ch07/model/BillingDetails.java

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
    @ManyToOne
    private User user;
    // ...
}
```

So far, there is nothing special about this association mapping. The `BillingDetails` class hierarchy can be mapped with `TABLE_PER_CLASS`, `SINGLE_TABLE`, or a `JOINED` inheritance type. Hibernate is smart enough to use the right SQL queries, with either `JOIN` or `UNION` operators, when loading the collection elements.

There is one limitation, however: the `BillingDetails` class can't be a `@MappedSuperclass`, as shown in section 7.1. It has to be mapped with `@Entity` and `@Inheritance`.

7.9 Summary

- We examined table per concrete class with implicit polymorphism as the simplest strategy to map inheritance hierarchies of entities and emphasized it doesn't support polymorphic associations very well. In addition, different columns from different tables share exactly the same semantics, making schema evolution more complex. We recommended this approach for the top level of the class hierarchy only, where polymorphism isn't usually required and when modification of the superclass in the future is unlikely.
- The table-per-concrete-class-with-unions strategy that we demonstrated next is optional, and JPA implementations may not support it, but it does handle polymorphic associations.
- We examined the table-per-class-hierarchy strategy and demonstrated it is a winner in terms of both performance and simplicity; ad hoc reporting is possible without complex joins or unions, and schema evolution is straightforward. The one major problem is data integrity because we must declare some columns as nullable. Another issue is normalization: this strategy creates functional dependencies between non-key columns, violating the third normal form.
- We analyzed the table-per-subclass-with-joins strategy and concluded that its primary advantage is that it normalizes the SQL schema, making schema evolution and integrity constraint definition straightforward. The disadvantages are that it's more difficult to implement by hand, and performance can be unacceptable for complex class hierarchies.

8

Mapping collections and entity associations

This chapter covers

- **Mapping persistent collections**
- **Examining collections of basic and embeddable type**
- **Investigating simple many-to-one and one-to-many entity associations**

The first thing many developers try to do when they begin using Hibernate or Spring Data JPA is to map a *parent/children relationship*. This is usually the first time they encounter collections. It's also the first time they have to think about the differences between entities and value types or get lost in the complexity of ORM.

Managing the associations between classes and the relationships between tables is at the heart of ORM. Most of the difficult problems involved in implementing an ORM solution relate to collections and entity association management. We start this chapter with basic collection-mapping concepts and simple examples. After that, you'll be prepared for the first collection in an entity association—although we'll come back to more complicated entity association mappings in the next chapter. To get the full picture, we recommend you read both this chapter and the next.

Major new features in JPA 2

Support for collections and maps of basic and embeddable types.

Support for persistent lists where the index of each element is stored in an additional database column.

One-to-many associations now have an orphan removal option.

8.1 Sets, bags, lists, and maps of value types

Java has a rich collection API, from which we can choose the interface and implementation that best fits the domain model design. We'll use the Java Collections framework for the implementation. Let's walk through the most common collection mappings, repeating the same `Image` and `Item` example with minor variations. We'll start by looking at the database schema and creating and mapping a collection property in general. The database goes first, as it is generally first designed and our programs must work with it. Then we'll proceed to how to select a specific collection interface and map various collection types: a set, identifier bag, list, map, and finally sorted and ordered collections.

8.1.1 The database schema

We extend `CaveatEmptor` to support attaching images to auction items. An item with an associated image is more interesting for the potential buyer to inspect it. We'll ignore the Java code for now, and we'll take a step back to consider only the database schema. To be able to execute the examples from the source code, you need first to run the `Ch08.sql` script. The source code to follow is to be found in the `mapping-collections` folder.

For the auction item and images example, assume that the image is stored somewhere on the file system and that we keep just the filename in the database. When an image is deleted from the database, a separate process must delete the file from the disk.

We need an `IMAGE` table in the database to hold the images, or maybe just the filenames of images. This table also has a foreign key column, say `ITEM_ID`, referencing the `ITEM` table. Look at the schema shown in figure 8.1.

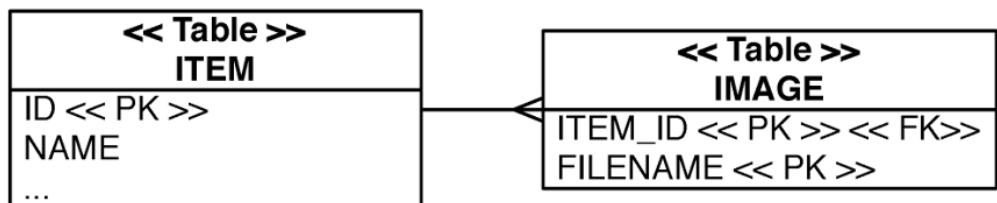


Figure 8.1 The `IMAGE` table holds image filenames, each referencing an `ITEM_ID`.

That's all there is to the schema—no collections or composition life cycle.

8.1.2 Creating and mapping a collection property

How would we map this `IMAGE` table with what we know so far? We'd probably map it as an `@Entity` class named `Image`. Later in this chapter, we'll map a foreign key column with a `@ManyToOne` property, to make the association between the entities. We'd also need a composite primary key mapping for the entity class, as demonstrated in section 10.2.2. What we need to know for now is that a composite primary key is a combination of more than one column to uniquely identify a row in a table. Individual columns may not be unique, but their combination must be unique.

There are no mapped collections; they aren't necessary. When we need an item's images, we'll write and execute a query in the JPA query language: `select img from Image img where img.item = :itemParameter`. Persistent collections are *always* optional—a feature. Why would we map a collection?

The collection we could create is `Item#images`, referencing all images for a particular item. We may create and map this collection property to do the following:

- Execute the SQL query `SELECT * from IMAGE where ITEM_ID = ?` automatically when we call `someItem.getImages()`. As long as the domain model instances are in a *managed* state (more later), we can read from the database on-demand while navigating the associations between the classes. We don't have to manually write and execute a query to load data. On the other hand, the collection query when we start iterating the collection is always "all images for this item," never "only images that match criteria XYZ."
- Avoid saving each `Image` with `entityManager.persist()` or `imageRepository.save()`. If we have a mapped collection, adding the `Image` to the collection with `someItem.getImages().add()` will make it persistent automatically when the `Item` is saved. This cascading persistence is convenient because we can save instances without calling the repository or the `EntityManager`.
- Have a dependent life cycle of `Images`. When an `Item` is deleted, Hibernate deletes all attached `Images` with an extra SQL `DELETE`. We don't have to worry about the life cycle of images and cleaning up orphans (assuming the database foreign key constraint doesn't `ON DELETE CASCADE`). The JPA provider handles the composition life cycle.

It's important to realize that although these benefits sound great, the price to pay is additional mapping complexity. Many JPA beginners struggle with collection mappings, and frequently the answer to "Why are you doing this?" has been "I thought this collection was required."

Analyzing how we can treat the scenario with images for auction items, we'd benefit from a collection mapping. The images have a dependent life cycle; when an item is deleted, all the attached images should be deleted. When an item is stored, all attached images should be stored. And when an item is displayed, we often also display all images, so `someItem.getImages()` is convenient in UI code – this is rather an eager loading of the information. We don't have to call the persistence service again to get the images; they're just *there*.

Now, we are on to choosing the collection interface and implementation that best fits the domain model design. Let's walk through the most common collection mappings, repeating the same `Image` and `Item` example with minor variations.

8.1.3 Selecting a collection interface

The idiom for a collection property in the Java domain model is

```
<<Interface>> images = new <<Implementation>>();
// Getter and setter methods
// ...
```

Use an interface to declare the type of the property, not an implementation. Pick a matching implementation, and initialize the collection right away; doing so avoids uninitialized collections. We don't recommend initializing collections late in constructors or setter methods.

Using generics, here's a typical `Set`:

```
Set<Image> images = new HashSet<Image>();
```

Raw collections without generics

If we don't specify the type of collection elements with generics, or the key/value types of a map, we need to tell Hibernate the type(s). For example, instead of a `Set<String>`, we map a raw `Set` with `@ElementCollection(targetClass= String.class)`. This also applies to type parameters of a Map. Specify the key type of a Map with `@MapKeyClass`. All the examples in this book use generic collections and maps, and so should you.

Out of the box, Hibernate supports the most important JDK collection interfaces and preserves the semantics of JDK collections, maps, and arrays in a persistent fashion. Each JDK interface has a matching implementation supported by Hibernate, and it's important that we use the right combination. Hibernate wraps the already initialized collection on the declaration of the field or sometimes replaces it if it's not the right one. It does that to enable, among other things, lazy loading and dirty checking of collection elements.

Without extending Hibernate, we can choose from the following collections:

- A `java.util.Set` property, initialized with a `java.util.HashSet`. The order of elements isn't preserved, and duplicate elements aren't allowed. All JPA providers support this type.
- A `java.util.SortedSet` property, initialized with a `java.util.TreeSet`. This collection supports stable order of elements: sorting occurs in memory after Hibernate loads the data. This is a Hibernate-only extension; other JPA providers may ignore the "sorted" aspect of the set.
- A `java.util.List` property, initialized with a `java.util.ArrayList`. Hibernate preserves the position of each element with an additional index column in the database table. All JPA providers support this type.

- A `java.util.Collection` property, initialized with a `java.util.ArrayList`. This collection has *bag* semantics; duplicates are possible, but the order of elements isn't preserved. All JPA providers support this type.
- A `java.util.Map` property, initialized with a `java.util.HashMap`. The key and value pairs of a map can be preserved in the database. All JPA providers support this type.
- A `java.util.SortedMap` property, initialized with a `java.util.TreeMap`. It supports stable order of elements: sorting occurs in memory after Hibernate loads the data. This is a Hibernate-only extension; other JPA providers may ignore the "sorted" aspect of the map.
- Hibernate supports persistent arrays, but JPA doesn't. They're rarely used, and we won't show them in this book: Hibernate can't wrap array properties, so many benefits of collections, such as on-demand lazy loading, won't work. Only use persistent arrays in your domain model if you're sure you won't need lazy loading. (You can load arrays on-demand, but this requires interception with bytecode enhancement, as explained in section 12.1.3.)

If we want to map collection interfaces and implementations not directly supported by Hibernate, we need to tell Hibernate about the semantics of your custom collections. The extension point in Hibernate is the `PersistentCollection` interface in the `org.hibernate.collection.spi` package, where we usually extend one of the existing `PersistentSet`, `PersistentBag`, and `PersistentList` classes. Custom persistent collections aren't easy to write, and we don't recommend doing this if you aren't an experienced Hibernate user.

Transactional file systems

If we only keep the filenames of images in the SQL database, we have to store the binary data of each picture—the files—somewhere. We could store the image data in the SQL database in `BLOB` columns (see the section “Binary and large value types” in chapter 6). If we decide not to store the images in the database, but as regular files, we should be aware that the standard Java file system APIs, `java.io.File`, and `java.nio.file.Files`, aren’t transactional. File system operations aren’t enlisted in a (JTA – Java Transaction API) system transaction; a transaction might successfully complete, with Hibernate writing the filename into the SQL database, but then storing or deleting the file in the file system might fail. We won’t be able to roll back these operations as one atomic unit, and we won’t get proper isolation of operations.

You may use a separate system transaction manager such as Bitronix. File operations are then enlisted, committed, and rolled back together with Hibernate’s SQL operations in the same transaction.

Let’s map a collection of image filenames of an `Item`.

8.1.4 Mapping a set

The simplest implementation is a `Set` of `String` image filenames. Add a collection property to the `Item` class, as demonstrated in the following listing from the `setofstrings` example.

Listing 8.1 Images mapped as a simple set of strings

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofstrings/Item.java

@Entity
public class Item {
    ...

    @ElementCollection
    @CollectionTable(
        name = "IMAGE",
        joinColumns = @JoinColumn(name = "ITEM_ID"))
    @Column(name = "FILENAME")
    private Set<String> images;
```

#A We declare the `images` field as an `@ElementCollection`. We refer here to image paths on the system, but for brevity, we’ll use the names of the fields and columns as `image` or `images`.

#B The collection table will be named `IMAGE`. Otherwise, it will default to `ITEM_IMAGES`.

#C The join column between the `ITEM` and the `IMAGE` tables will be `ITEM_ID` (the default name, in fact).

#D The name of the column to contain the string information from the `images` collection will be `FILENAME`. Otherwise, it will default to `IMAGES`.

#E We initialize the `images` collection as a `HashSet`.

In the listing above:

The `@ElementCollection` JPA annotation is required for a collection of value-typed elements. Without the `@CollectionTable` and `@Column` annotations, Hibernate would use default schema names. Look at the schema in figure 8.2: the primary key columns are underlined.

ITEM		IMAGE	
id	name	ITEM_ID	FILENAME
1	Foo	1	landscape.jpg
		1	foreground.jpg
		1	background.jpg
		1	portrait.jpg

Figure 8.2 Table structure and example data for a set of strings

The `IMAGE` table has a composite primary key of both the `ITEM_ID` and `FILENAME` columns. That means we can't have duplicate rows: each image file can only be attached once to one item. The order of images isn't stored. This fits the domain model and `Set` collection. The image is stored somewhere on the file system and we keep just the filename in the database.

To interact with the `Item` entities, we'll create the following Spring Data JPA repository:

Listing 8.2 The ItemRepository interface

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/repositories/setofstrings
      /Item.java

public interface ItemRepository extends JpaRepository<Item, Long> {

    @Query("select i from Item i inner join fetch i.images where i.id = :id") #A
    Item findItemWithImages(@Param("id") Long id); #A

    @Query(value = "SELECT FILENAME FROM IMAGE WHERE ITEM_ID = ?1", #B
           nativeQuery = true) #B
    Set<String> findImagesNative(Long id); #B
}
```

#A We declare a `findItemWithImages` method that will get the `Item` by `id`, including the `images` collection. To fetch this collection, we will use the `inner join fetch` capability of JPQL (Jakarta Persistence Query Language).

#B We also declare the `findImagesNative` method, which is annotated with a native query and will get the set of strings representing the images of a given `id`.

In the listing above:

We'll also create the following test:

Listing 8.3 The MappingCollectionsSpringDataJPATest class

```
Path: Ch08/mapping-collections/src/test/java/com/manning/javapersistence/ch08/
      /setofstrings/MappingCollectionsSpringDataJPATest.java

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {SpringDataConfiguration.class})
public class MappingCollectionsSpringDataJPATest {

    @Autowired
    private ItemRepository itemRepository;

    @Test
    void storeLoadEntities() {

        Item item = new Item("Foo");                                #A

        item.addImage("background.jpg");
        item.addImage("foreground.jpg");
        item.addImage("landscape.jpg");
        item.addImage("portrait.jpg");                             #B
                                                               #B
                                                               #B
                                                               #B

        itemRepository.save(item);                               #C

        Item item2 = itemRepository.findItemWithImages(item.getId()); #D

        List<Item> items2 = itemRepository.findAll();           #E
        Set<String> images = itemRepository.findImagesNative(item.getId()); #F

        assertEquals(1, items2.size());
        assertEquals(4, images.size());                         #G
        assertEquals(4, item2.getImages().size());             #G
        assertEquals(1, item2.getImages().size());             #G
        assertEquals(4, item.getImages().size());              #G
    }
}
```

#A We create an `Item`.

#B We add 4 images paths to it.

#C We save it to the database.

#D We access the repository to get the item together with the `images` collection. As we specified in the JPQL query with which the `findItemWithImages` method is annotated, the collection will also be fetched from the database.

#E We get all `Items` from the database.

#F We get the set of strings representing the images, using the `findImagesNative` method.

#G We make the checks regarding the sizes of the different collections we have previously obtained.

It doesn't seem likely that we'd allow the user to attach the same image more than once to the same item, but let's suppose we did. What kind of mapping would be appropriate in that case for us to choose?

8.1.5 Mapping an identifier bag

A *bag* is an unordered collection that allows duplicate elements, like the `java.util.Collection` interface. Curiously, the Java Collections framework doesn't include

a bag implementation. We may initialize the property with an `ArrayList`, and Hibernate ignores the index of elements when storing and loading elements. The code is to be found in the `bagofstrings` example.

Listing 8.4 Bag of strings, allowing duplicate elements

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/bagofstrings/Item.java

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @GenericGenerator(name = "sequence_gen", strategy = "sequence")          #A
    @org.hibernate.annotations.CollectionId(                                #B
        columns = @Column(name = "IMAGE_ID"),                                #C
        type = @org.hibernate.annotations.Type(type = "long"),                #D
        generator = "sequence_gen")                                         #E
    private Collection<String> images = new ArrayList<>();                  #F
```

#A We declare a `@GenericGenerator` with the name “`sequence_gen`” and strategy “`sequence`” to take care of the surrogate keys in the `IMAGE` table.

#B The `IMAGE` collection table needs a different primary key to allow duplicate `FILENAME` values for each `ITEM_ID`.

#C We introduce a surrogate primary key column named `IMAGE_ID`. Usually, you may retrieve all images at the same time or store all of them at the same time, but a database table still needs a primary key.

#D We use a Hibernate-only annotation.

#E We configure how the primary key is generated.

#F There is no bag implementation in JDK. We initialize the collection as `ArrayList`.

In the listing above, we made the following changes from the example mapping a set:

ITEM		IMAGE		
id	name	IMAGE_ID	ITEM_ID	FILENAME
1	Foo	1	1	landscape.jpg
		2	1	foreground.jpg
		3	1	background.jpg
		4	1	portrait.jpg

Figure 8.3 Surrogate primary key column for a bag of strings

The Spring Data JPA repository and the test will be just like in the previous example.

Here's an interesting question: if all you see is this schema, can you tell how the tables are mapped in Java? The `ITEM` and `IMAGE` tables look the same: each has a surrogate primary key column and some other normalized columns. Each table could be mapped with an `@Entity` class. We may decide to use a JPA feature and map a collection to `IMAGE`, however, even with a composition life cycle. This is, effectively, a decision that some predefined query and manipulation rules are all we need for this table instead of the more generic `@Entity` mapping. When you make such a decision, be sure you know the reasons and consequences.

The next mapping technique preserves the order of images with a list.

8.1.6 Mapping a list

When you haven't used ORM software before, a persistent list seems to be a very powerful concept; imagine how much work storing and loading a `java.util.List <String>` is with plain JDBC and SQL. If we add an element to the middle of the list, depending on the list implementation, the list shifts all subsequent elements to the right or rearranges pointers. If we remove an element from the middle of the list, something else happens, and so on. If the ORM software can do all of this automatically for database records, this makes a persistent list look more appealing than it actually is.

As we noted in section 3.2.4, the first reaction is often to preserve the order of data elements as users enter them. We often have to show them later in the same order. But if another criterion can be used for sorting the data, like an entry timestamp, we should sort the data when querying and not store the display order. What if the display order changes? The order in which data is displayed is most likely not an integral part of the data but an orthogonal concern. Think twice before mapping a persistent `List`; Hibernate isn't as smart as you might think, as we'll see in the next example.

First, let's change the `Item` entity and its collection property, as demonstrated in the `listofstrings` example.

Listing 8.5 Persistent list, preserving the order of elements in the database

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/listofstrings/Item.java

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @OrderColumn // Enables persistent order, Defaults to IMAGES_ORDER
    @Column(name = "FILENAME")
    private List<String> images = new ArrayList<>();
```

There is a new annotation in this example: `@OrderColumn`. This column stores an index in the persistent list, starting at zero. The column name will default to `IMAGES_ORDER`. Note that Hibernate stores and expects the index to be contiguous in the database. If there are gaps,

Hibernate will add `null` elements when loading and constructing the `List`. Look at the schema in figure 8.4.

ITEM		IMAGE		
id	name	ITEM_ID	IMAGES_ORDER	FILENAME
1	Foo	1	0	landscape.jpg
2	Bar	1	1	foreground.jpg
		1	2	background.jpg
		1	3	background.jpg
		2	0	portrait.jpg
		2	1	foreground.jpg

Figure 8.4 The collection table preserves the position of each list element.

The primary key of the `IMAGE` table is a composite of `ITEM_ID` and `IMAGES_ORDER`. This allows duplicate `FILENAME` values, which is consistent with the semantics of a `List`. Remember, the image is stored somewhere on the file system and we keep just the filename in the database.

The Spring Data JPA repository and the test will be just like in the previous example.

We said earlier that Hibernate isn't as smart as you might think. Consider modifications to the list: say the list has three images A, B, and C, in that order. What happens if you remove A from the list? Hibernate executes one SQL `DELETE` for that row. Then it executes two `UPDATES`, for B and C, shifting their position to the left to close the gap in the index. For each element to the right of the deleted element, Hibernate executes an `UPDATE`. If we write SQL for this by hand, we can do it with one `UPDATE`. The same is true for insertions in the middle of the list. Hibernate shifts all existing elements to the right one by one. At least Hibernate is smart enough to execute a single `DELETE` when we `clear()` the list.

Now, suppose the images for an item have user-supplied names in addition to the filename. One way to model this in Java is with a map, using key/value pairs.

8.1.7 Mapping a map

Again, make a small change to the Java class to use a `Map` property, demonstrated in the `mapofstrings` example.

Listing 8.6 Persistent map storing its key and value pairs

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/mapofstrings/Item.java

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")                                     #A
    @Column(name = "IMAGENAME")                                         #B
    private Map<String, String> images = new HashMap<>();
```

#A Each map entry is a key/value pair. Here the key is mapped with `@MapKeyColumn` to `FILENAME`.

As you can see from the schema in figure 8.5, the primary key of the collection table is a composite of `ITEM_ID` and `FILENAME`. The example uses a `String` as the key for the map, but Hibernate supports any basic type, such as `BigDecimal` and `Integer`. If the key is a Java enum, we must use `@MapKeyEnumerated`. With any temporal types such as `java.util.Date`, use `@MapKeyTemporal`. We discussed these options, albeit not for collections, in sections 6.1.6 and 6.1.7.

ITEM		IMAGE		
Id	name	ITEM_ID	FILENAME	IMAGENAME
1	Foo	1	landscape.jpg	Landscape
2	Bar	1	foreground.jpg	Foreground
		1	background.jpg	Background
		1	portrait.jpg	Portrait
		2	landscape.jpg	Landscape
		2	foreground.jpg	Foreground

Figure 8.5 Tables for a map, using strings as indexes and elements

The map in the previous example was unordered. If the list of files is pretty long and we would like to quickly look for something at a glance, what should we do to always sort map entries by filename?

8.1.8 Sorted and ordered collections

We *sort* a collection in memory using a Java comparator. We *order* a collection when it's loaded from the database, using an `ORDER BY` clause.

Let's make the map of images a sorted map. We need to change the Java property and the mapping, as demonstrated in the `sortedmapofstrings` example.

Listing 8.7 Sorting map entries in memory, using a comparator

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/sortedmapofstrings/Item.j
ava

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.SortComparator(ReverseStringComparator.class)
    private SortedMap<String, String> images = new TreeMap<>();
```

Sorted collections are a Hibernate feature; hence the annotation `org.hibernate.annotations.SortComparator` with an implementation of `java.util.Comparator` – it sorts strings in reverse order.

The database schema doesn't change, which is also the case for all the following examples. Look at the illustrations in the previous sections if you need a reminder.

We add the following two lines to the test, which will check keys are now in reverse order:

```
() -> assertEquals("Portrait", item2.getImages().firstKey()),
() -> assertEquals("Background", item2.getImages().lastKey())
```

We'll map a `java.util.SortedSet` as demonstrated next, in the `sortedsetofstrings` example.

Listing 8.8 Sorting set elements in memory with String#compareTo()

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/sortedsetofstrings/Item.j
ava

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @org.hibernate.annotations.SortNatural
    private SortedSet<String> images = new TreeSet<>();
```

Here natural sorting is used, falling back on the `String#compareTo()` method.

Unfortunately, we can't sort a bag; there is no `TreeBag`. The indexes of list elements predefined their order.

Alternatively, instead of switching to `Sorted*` interfaces, we may want to retrieve the elements of a collection in the right order from the database and not sort in memory. Instead of a `java.util.SortedSet`, a `java.util.LinkedHashSet` is used in the following `setofstringsorderby` example.

Listing 8.9 `LinkedHashSet` offers insertion order for iteration

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofstringsorderby/Item.
      java

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    // @javax.persistence.OrderBy // One possible order: "FILENAME asc"
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    private Set<String> images = new LinkedHashSet<>();
```

The `LinkedHashSet` class has a stable iteration order over its elements, and Hibernate will fill it in the right order when loading a collection. To do this, Hibernate applies an `ORDER BY` clause to the SQL statement that loads the collection. We must declare this SQL clause with the proprietary `@org.hibernate.annotations.OrderBy` annotation. We could even call an SQL function, like `@OrderBy("substring(FILENAME, 0, 3) desc")`, which would sort by the first three letters of the filename. Be careful to check that the DBMS supports the SQL function you're calling. Furthermore, you can use the SQL:2003 syntax `ORDER BY ... NULLS FIRST|LAST`, and Hibernate will automatically transform it into the dialect supported by your DBMS.

Hibernate @OrderBy vs. JPA @OrderBy

We can apply the annotation `@org.hibernate.annotations.OrderBy` to any collection; its parameter is a plain SQL fragment that Hibernate attaches to the SQL statement loading the collection. Java Persistence has a similar annotation, `@javax.persistence.OrderBy`. Its (only) parameter is not SQL but `someProperty DESC | ASC`. A String or Integer element value has no properties. Hence, when we apply JPA's `@OrderBy` annotation on a collection of basic type, as in the previous example with a `Set<String>`, the specification says, “the ordering will be by value of the basic objects.” This means we can't change the order: in the previous example, the order will always be by `FILENAME asc` in the generated SQL query. We use the JPA annotation later when the element value class has persistent properties and isn't of basic/scalar type, in section 8.2.2.

The next example from `bagofstringsorderby` demonstrates the same ordering at load time with a bag mapping.

Listing 8.10 ArrayList provides stable iteration order

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/bagofstringsorderby/Item.
            java

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @GenericGenerator(name = "sequence_gen", strategy = "sequence")
    @org.hibernate.annotations.CollectionId(
        columns = @Column(name = "IMAGE_ID"),
        type = @org.hibernate.annotations.Type(type = "long"),
        generator = "sequence_gen")
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    private Collection<String> images = new ArrayList<String>();
```

Finally, we can load ordered key/value pairs with a `LinkedHashMap`, as in the `mapofstringsorderby` example.

Listing 8.11 LinkedHashMap keeps key/value pairs in order

```
Path: Ch08/mapping-
collections/src/main/java/com/manning/javapersistence/ch08/mapofstringsorderby/Item.java

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    private Map<String, String> images = new LinkedHashMap<>();
```

Keep in mind that the elements of ordered collections are only in the desired order when they're loaded. As soon as we add and remove elements, the iteration order of the collections might be different than "by filename"; they behave like regular linked sets, maps, or lists. We demonstrated the technical approach, but we need to be aware of its shortcomings and to conclude that these ones make it a not very reliable solution.

In a real system, it's likely that we'll need to keep more than just the image name and filename. We'll probably need to create an `Image` class for this extra information (as title, width, height). This is the perfect use case for a collection of components that we'll deal with further.

8.2 Collections of components

We mapped an embeddable component earlier: the `address` of a `User` in section 6.2. The current situation is different because an `Item` has many references to an `Image`, as shown in figure 8.6. The association in the UML diagram is a composition (the black diamond); hence, the referenced `Images` are bound to the life cycle of the owning `Item`.

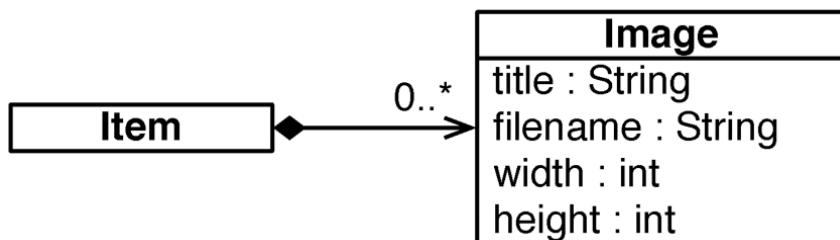


Figure 8.6 Collection of `Image` components in `Item`

The code in the next listing from the `setofembeddables` example demonstrates the new `Image` embeddable class, capturing all the properties of an image that interest us.

Listing 8.12 Encapsulating all properties of an image

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofembeddables/Image.java

@Embeddable
public class Image {
    @Column(nullable = false)
    private String filename;
    private int width;
    private int height;
    //...
}
```

First, note that all properties are non-optional, NOT NULL. The size properties are non-nullable because their values are primitives. Second, we have to consider equality and how the database and Java tier compare two images.

8.2.1 Equality of component instances

Let's say we keep several `Image` instances in a `HashSet`. We know that sets don't allow duplicate elements. How do sets detect duplicates? The `HashSet` calls the `equals()` method on each `Image` we put in the set. (It also calls the `hashCode()` method to get a hash, obviously.) How many images are in the following collection?

```
someItem.addImage(new Image("background.jpg", 640, 480));
someItem.addImage(new Image("foreground.jpg", 800, 600));
someItem.addImage(new Image("landscape.jpg", 1024, 768));
someItem.addImage(new Image("landscape.jpg", 1024, 768));
assertEquals(3, someItem.getImages().size());
```

Did you expect four images instead of three? You're right: the regular Java equality check relies on identity. The `java.lang.Object#equals()` method compares instances with `a==b`. Using this procedure, we'd have four instances of `Image` in the collection. Clearly, three is the "correct" answer for this use case.

For the `Image` class, we don't rely on Java identity—we override the `equals()` and `hashCode()` methods.

Listing 8.13 Implementing custom equality with equals() and hashCode()

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofembeddables/Image.java

@Override
public class Image {
    //...
    @Override
    public boolean equals(Object o) {                                     #A
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Image image = (Image) o;
        return width == image.width &&
               height == image.height &&
               filename.equals(image.filename);
    }

    @Override
    public int hashCode() {                                              #B
        return Objects.hash(filename, width, height);
    }
    //...
}
```

#A This custom equality check in `equals()` compares all values of one `Image` to the values of another `Image`. If all values are the same, then the images must be the same.

#B The `hashCode()` method has to fulfill the contract requiring that if two instances are equal, they must have the same hash code.

Why didn't we override equality before, when we mapped the `Address` of a `User` in section 6.2? Well, the truth is, we probably should have done that. Our only excuse is that we won't have any problems with the regular identity equality unless we put embeddable components into a `Set` or use them as keys in a `Map`. Then we should redefine equality based on values, not identity. It's best if we override these methods on every `@Embeddable` class; all value types should be compared "by value."

Now consider the database primary key: Hibernate will generate a schema that includes all non-nullable columns of the `IMAGE` collection table in a composite primary key. The columns have to be non-nullable because we can't identify what we don't know. This reflects the equality implementation in the Java class. We'll see the schema in the next section, with more details about the primary key.

NOTE We have to mention a minor issue with Hibernate's schema generator: if we annotate an embeddable's property with `@NotNull` instead of `-@Column(nullable=false)`, Hibernate won't generate a NOT NULL constraint for the collection table's column. A Bean Validation check of an instance works as expected, only the database schema is missing the integrity rule. Use `@Column(nullable=false)` if the embeddable class is mapped in a collection, and the property should be part of the primary key.

The component class is now ready, and we can use it in collection mappings.

8.2.2 Set of components

We map a `Set` of components as shown next in the same `setofembeddables` example. We remind that a `Set` is a type of collection allowing only unique items.

Listing 8.14 Set of embeddable components with an override

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofembeddables/Item.jav
      a

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")                                #A
    @AttributeOverride(
        name = "filename",
        column = @Column(name = "FNAME", nullable = false)          #B
    )
    private Set<Image> images = new HashSet<>();
```

#A As before, the `@ElementCollection` annotation is required. Hibernate automatically knows that the target of the collection is an `@Embeddable` type, from the declaration of a generic collection.

#B The `@CollectionTable` annotation overrides the default name for the collection table, which would have been `ITEM_IMAGES`.

The `Image` mapping defines the columns of the collection table. Just as for a single embedded value, we can use `@AttributeOverride` to customize the mapping without modifying the target embeddable class. Look at the database schema in figure 8.7.

ITEM		IMAGE			
ID	NAME	ITEM_ID	FNAME	WIDTH	HEIGHT
1	Foo	1	landscape.jpg	640	480
2	Bar	1	foreground.jpg	800	600
		1	background.jpg	1024	768
		1	portrait.jpg	480	640
		2	landscape.jpg	640	480
		2	foreground.jpg	800	600

Figure 8.7 Example data tables for a collection of components

We're mapping a set, so the primary key of the collection table is a composite of the foreign key column `ITEM_ID` and all "embedded" non-nullable columns: `FNAME`, `WIDTH`, and `HEIGHT`.

The `ITEM_ID` value wasn't included in the overridden `equals()` and `hashCode()` methods of `Image`, as discussed in the previous section. Therefore, if we mix images of different items in one set, we'll run into equality problems in the Java tier. In the database table, we obviously can distinguish images of different items because the item's identifier is included in primary key equality checks.

If we want to include the `Item` in the equality routine of the `Image`, to be symmetric with the database primary key, we need an `Image#item` property. This is a simple back-pointer provided by Hibernate when `Image` instances are loaded:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofembeddables/Image.java

@Embeddable
public class Image {
    ...
    @org.hibernate.annotations.Parent
    private Item item;
    ...
}
```

We can now include the parent `Item` value in the `equals()` and `hashCode()` implementations.

As we matched the `FILENAME` field to the `FNAME` column using the `@AttributeOverride` annotation:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofembeddables/Item.java

@AttributeOverride(
    name = "filename",
    column = @Column(name = "FNAME", nullable = false)
)
```

we'll also have to change the native query from the `ItemRepository` interface:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/repositories/setofembedda-
      bles/ItemRepository.java

@Query(value = "SELECT FNAME FROM IMAGE WHERE ITEM_ID = ?1",
       nativeQuery = true)
Set<String> findImagesNative(Long id);
```

If we need load-time ordering of elements and a stable iteration order with a `LinkedHashSet`, we use the JPA `@OrderBy` annotation, as in the `setofembeddablesorderby` example:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/setofembeddablesorderby/I-
      tem.java

@Entity
public class Item {
    ...
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @OrderBy("filename DESC, width DESC")
    private Set<Image> images = new LinkedHashSet<>();
```

The arguments of the `@OrderBy` annotation are properties of the `Image` class, followed by either `ASC` for ascending or `DESC` for descending order. The default is ascending. This example sorts descending by image filename and then descending by the width of each image. Note that this is different from the proprietary `@org.hibernate.annotations.OrderBy` annotation, which takes a plain SQL clause, as discussed in section 8.1.8.

Declaring all properties of `Image` as `@NotNull` may not be something we want. If any of the properties are optional, we need a different primary key for the collection table.

8.2.3 Bag of components

We used the `@org.hibernate.annotations.CollectionId` annotation before adding a surrogate key column to the collection table. The collection type, however, was not a `Set` but a general `Collection`, a bag. This is consistent with the updated schema: if we have a surrogate primary key column, duplicate “element values” are allowed. Let's walk through this with the `bagofembeddables` example.

First, the `Image` class may now have nullable properties, as we'll have a surrogate key:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/bagofembeddables/Image.java
      va
@Embeddable
public class Image {
    @Column(nullable = true)
    private String title;
    @Column(nullable = false)
    private String filename;
    private int width;
    private int height;
    // ...
}
```

Remember to account for the optional `title` of the `Image` in the overridden `equals()` and `hashCode()` methods when comparing instances "by value". For example, the comparison of the `title` fields will be done in the `equals` method like this:

```
Objects.equals(title, image.title)
```

Next, the mapping of the bag collection in `Item`:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/bagofembeddables/Item.java
      a

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
        @GenericGenerator(name = "sequence_gen", strategy = "sequence")
    @org.hibernate.annotations.CollectionId(
        columns = @Column(name = "IMAGE_ID"),
        type = @org.hibernate.annotations.Type(type = "long"),
        generator = "sequence_gen")
    private Collection<Image> images = new ArrayList<>();
    // ...
}
```

As before, in section 8.1.5, we declare an additional surrogate primary key column `IMAGE_ID` with the proprietary `@org.hibernate.annotations.CollectionId` annotation. Figure 8.8 shows the database schema.

ITEM		IMAGE					
ID	NAME	IMAGE_ID	ITEM_ID	TITLE	FILENAME	WIDTH	HEIGHT
1	Foo	1	1	Landscape	landscape.jpg	640	480
2	Bar	2	1		foreground.jpg	800	600
		3	1	Background	background.jpg	1024	768
		4	1	Portrait	portrait.jpg	480	640
		5	2	Landscape	landscape.jpg	640	480
		6	2	Foreground	foreground.jpg	800	600

Figure 8.8 Collection of components table with a surrogate primary key column

The title of the Image with identifier 2 is null.

Next, we analyze another way to change the primary key of the collection table with a Map.

8.2.4 Map of component values

A map keeps the information as pairs of keys and values. If the Images are stored in a Map, the filename can be the map key, as demonstrated in the `mapofstringsembeddables/Item.java` example:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/mapofstringsembeddables/Item.java

@Entity
public class Item {
    //...

    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "TITLE") #A
    private Map<String, Image> images = new HashMap<>();
    // ...
}
```

#A The key column of the map is set to TITLE. Otherwise, it will default to IMAGES_KEY.

The test will set the TITLE column by executing instructions of this kind:

```
item.putImage("Background", new Image("background.jpg", 640, 480));
```

The primary key of the collection table, as shown in figure 8.9, is now the foreign key column ITEM_ID and the key column of the map, TITLE.

ITEM		IMAGE				
ID	NAME	ITEM_ID	TITLE	FILENAME	WIDTH	HEIGHT
1	Foo	1	Landscape	landscape.jpg	640	480
2	Bar	1	Foreground	foreground.jpg	800	600
		1	Background	background.jpg	1024	768
		1	Portrait	portrait.jpg	480	640
		2	Landscape	landscape.jpg	640	480
		2	Foreground	foreground.jpg	800	600

Figure 8.9 Database tables for a map of components

The embeddable `Image` class maps all other columns, which may be nullable:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/mapofstringsembeddables/I
      mage.java

@Embeddable
public class Image {
    @Column(nullable = true)                                         #A
    private String filename;
    private int width;
    private int height;
    // ...
}
```

#A The `filename` field can now be null, it is not part of the primary key.

In the previous example, the values in the map were instances of an embeddable component class and the keys of the map a basic string. Next, we use embeddable types for both key and value.

8.2.5 Components as map keys

Our final example is mapping a `Map`, with both keys and values of embeddable type, as you can see in figure 8.10.

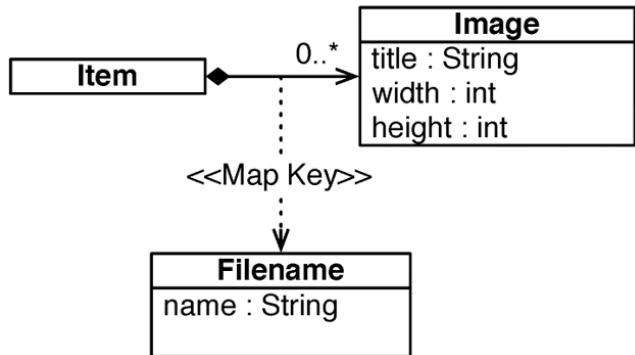


Figure 8.10 The Item has a Map keyed by Filename.

Instead of a string representation, we can represent a filename with a custom type, as demonstrated in the example `mapofembeddables`.

Listing 8.15 Representing a filename with a custom type

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/mapofsembeddables/Filenam
          e.java

@Embeddable
public class Filename {
    @Column(nullable = false)                                     #A
    private String name;
    // ...
}
```

#A The name field must not be null, as it is part of the primary key. If we want to use this class for the keys of a map, the mapped database columns can't be nullable because they're all part of a composite primary key. We also have to override the equals() and hashCode() methods because the keys of a map are a set, and each Filename must be unique within a given key set.

We don't need any special annotations to map the collection:

```
Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/mapofsembeddables/Item.java

@Entity
public class Item {
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    private Map<Filename, Image> images = new HashMap<>();
    // ...
}
```

In fact, we can't apply `@MapKeyColumn` and `@AttributeOverrides`; they have no effect when the map's key is an `@Embeddable` class. The composite primary key of the `IMAGE` table includes the `ITEM_ID` and `NAME` columns, as you can see in figure 8.11.

<code>ITEM</code>		<code>IMAGE</code>				
<code>ID</code>	<code>NAME</code>	<code>ITEM_ID</code>	<code>NAME</code>	<code>TITLE</code>	<code>WIDTH</code>	<code>HEIGHT</code>
1	Foo	1	landscape.jpg	Landscape	640	480
2	Bar	1	foreground.jpg	Foreground	800	600
		1	background.jpg		1024	768
		1	portrait.jpg	Portrait	480	640
		2	landscape.jpg	Landscape	640	480
		2	foreground.jpg	Foreground	800	600

Figure 8.11 Database tables for a Map of Images keyed on Filenames

A composite embeddable class like `Image` isn't limited to simple properties of basic type. We've already seen how to nest other components, such as `City` in `Address`. We could extract and encapsulate the `width` and `height` properties of `Image` in a new `Dimensions` class.

An embeddable class can also have its own collections.

8.2.6 Collection in an embeddable component

Suppose that for each `Address`, we want to store a list of contacts. This is a simple `Set<String>` in the embeddable class from the `embeddablesetofstrings` example:

```

Path: Ch08/mapping-
      collections/src/main/java/com/manning/javapersistence/ch08/embeddablesetofstrings/Address.java

@Embeddable
public class Address {
    @NotNull
    @Column(nullable = false)
    private String street;
    @NotNull
    @Column(nullable = false, length = 5)
    private String zipcode;
    @NotNull
    @Column(nullable = false)
    private String city;

    @ElementCollection
    @CollectionTable(
        name = "CONTACT",
        joinColumns = @JoinColumn(name = "USER_ID")) #A
        @Column(name = "NAME", nullable = false) #B
    private Set<String> contacts = new HashSet<>(); #C
    // ...
}

```

#A The `@ElementCollection` is the only required annotation; the table and column names have default values. The table name would default to `USER_CONTACTS`.

#B The join column would be `USER_ID` by default.

Look at the schema in figure 8.12: the `USER_ID` column has a foreign key constraint referencing the owning entity's table, `USERS`. The primary key of the collection table is a composite of the `USER_ID` and `NAME` columns, preventing duplicate elements appropriate for a `Set`.

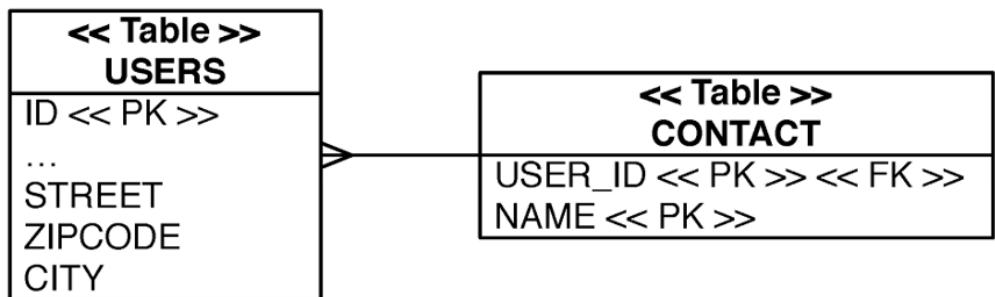


Figure 8.12 `USER_ID` has a foreign key constraint referencing `USERS`.

Instead of a `Set`, we could map a list, bag, or map of basic types. Hibernate also supports collections of embeddable types, so instead of a simple contact string, we could write an embeddable `Contact` class and have `Address` hold a collection of `Contacts`.

Although Hibernate gives a lot of flexibility with component mappings and fine-grained models, be aware that code is read more often than written. Think about the next developer who has to maintain this in a few years.

Switching focus, we turn our attention to entity associations: in particular, simple *many-to-one* and *one-to-many* associations.

8.3 Mapping entity associations

At the beginning of this chapter, we promised to talk about parent/children relationships. So far, we followed the mapping of an entity, `Item`. Let's say this is the parent. It has a collection of children: the collection of `Image` instances. The term *parent/child* implies some kind of life cycle dependency, so a collection of strings or embeddable components is appropriate. The children are fully dependent on the parent; they will always be saved, updated, and removed with the parent, never alone. We already mapped a parent/child relationship! The parent was an entity, and the many children were of value type. When an `Item` is removed, its collection of `Image` instances will also be removed. The actual images may be removed in a transactional way, meaning we either delete the rows from the database together with the files from the disk, or nothing at all. This is however a separate problem and we'll not deal with it.

Now we want to map a relationship of a different kind: an association between two entity classes. Their instances don't have a dependent life cycle. An instance can be saved, updated, and removed without affecting any other. Naturally, *sometimes* we have dependencies, even between entity instances. We need more fine-grained control of how the relationship between two classes affects instance state, not completely dependent (embedded) types. Are we still talking about a parent/child relationship? It turns out that *parent/child* is vague, and everyone has their own definition. We'll try not to use that term from now on and will instead rely on more precise or at least well-defined vocabulary.

The relationship we'll explore in the following sections is always the same, between the `Item` and `Bid` entity classes, as shown in figure 8.13.

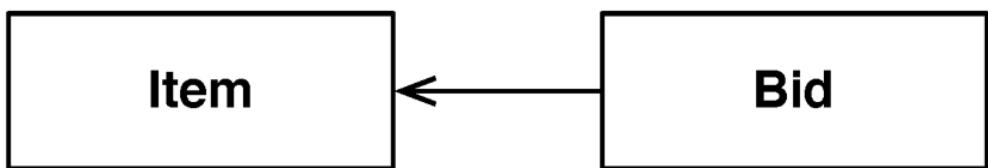


Figure 8.13 Relationship between `Item` and `Bid`

The association from `Bid` to `Item` is a *many-to-one* association. Later we'll make this association bidirectional, so the inverse association from `Item` to `Bid` will be *one-to-many*.

The *many-to-one* association is the simplest, so we'll talk about it first. The other associations, *many-to-many* and *one-to-one*, are more complex, and we'll discuss them in the next chapter.

Let's start with the *many-to-one* association that we need to implement in the CaveatEmptor application, and let's see which alternatives we have. The source code to follow is to be found in the mapping-associations folder.

We call the mapping of the `Bid#item` property a *unidirectional many-to-one association*. Before we analyze this mapping, look at the database schema in figure 8.14 and follow the bidirectional example.

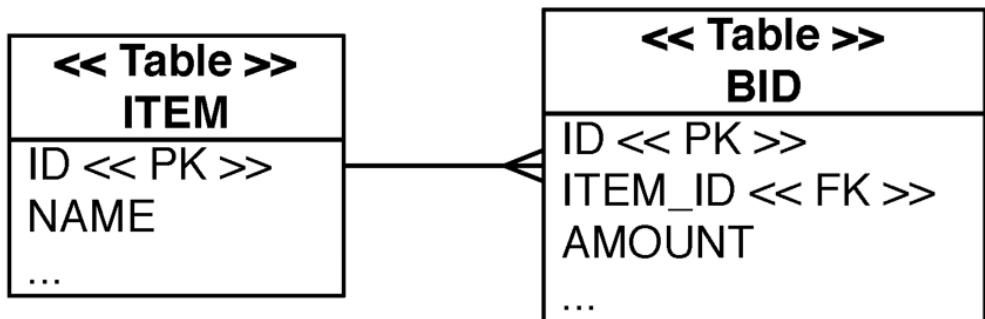


Figure 8.14 A *many-to-one* relationship in the SQL schema

Listing 8.16 Bid has a single reference to an Item

```

Path: Ch08/mapping-
      associations/src/main/java/com/manning/javapersistence/ch08/onetomany/bidirectional/
      Bid.java

@Entity
public class Bid {
    @ManyToOne(fetch = FetchType.LAZY)                                #A
    @JoinColumn(name = "ITEM_ID", nullable = false)
    private Item item;
    // ...
}

```

#A The `@ManyToOne` annotation marks a property as an entity association, and it's required. Its `fetch` parameter defaults to `EAGER`: this means the associated `Item` is loaded whenever the `Bid` is loaded.

We usually prefer lazy loading as a default strategy, and we'll talk more about it later in section 12.1.1.

A *many-to-one* entity association maps naturally to a foreign key column: `ITEM_ID` in the `BID` table. In JPA, this is called the *join column*. We don't need anything but the `@ManyToOne` annotation on the property. The default name for the join column is `ITEM_ID`: Hibernate automatically uses a combination of the target entity name and its identifier property, separated with an underscore.

We can override the foreign key column with the `@JoinColumn` annotation. We used it here for a different reason: to make the foreign key column `NOT NULL` when Hibernate generates the SQL schema. A bid always has to have a reference to an item; it can't survive on its own. (Note that this already indicates some kind of life cycle dependency we have to keep in mind.) Alternatively, we could mark this association as non-optional with either `@ManyToOne(optional = false)` or, as usual, Bean Validation's `@NotNull`.

This was easy. It's critically important to realize that we can write a complete and complex application without using anything else.

We don't need to map the other side of this relationship; we can ignore the *one-to-many* association from `Item` to `Bid`. There is only a foreign key column in the database schema, and we've already mapped it. We are serious about this: when we see a foreign key column and two entity classes involved, we should probably map it with `@ManyToOne` and nothing else. We can now get the `Item` of each `Bid` by calling `someBid.getItem()`. The JPA provider will dereference the foreign key and load the `Item` for us; it also takes care of managing the foreign key values. How do we get all of an item's bids? Well, we write a query and execute it with `EntityManager` or `JpaRepository`, in whatever query language Hibernate supports. For example, in JPQL, we'd use `select b from Bid b where b.item = :itemParameter`. One of the reasons we use Hibernate or Spring Data JPA is, of course, that in most cases, we don't want to write and execute that query ourselves.

8.3.2 Making it bidirectional

At the beginning of this chapter, we had a list of reasons a mapping of the collection `Item#images` was a good idea. Let's do the same for the collection `Item#bids`. This collection would implement the *one-to-many* association between `Item` and `Bid` entity classes. If we create and map this collection property, we get the following:

- Hibernate executes the SQL query `SELECT * from BID where ITEM_ID = ?` automatically when we call `someItem.getBids()` and start iterating through the collection elements.
- We can cascade state changes from an `Item` to all referenced `Bids` in the collection. We can select what life cycle events should be transitive: for example, we could declare that all referenced `Bid` instances should be saved when the `Item` is saved, so we don't have to call `EntityManager#persist()`, or `ItemRepository#save()` repeatedly for all bids.

Well, that isn't a very long list. The primary benefit of *one-to-many* mappings is navigational access to data. It's one of the core promises of ORM, enabling us to access data by calling only methods of our Java domain model. The ORM engine is supposed to take care of loading the required data in a smart way while we work with a high-level interface of our own design: `someItem.getBids().iterator().next().getAmount()`, and so on.

The fact that you can optionally cascade some state changes to related instances is a nice bonus. Consider, though, that some dependencies indicate value types at the Java level, not entities. Ask yourself if any table in the schema will have a `BID_ID` foreign key column. If not, map the `Bid` class as `@Embeddable`, not `@Entity`, using the same tables as before but

So, should we map the `Item#bids` collection at all? We get navigational data access, but the price to pay is additional Java code and significantly more complexity. This is frequently a difficult decision and it should rarely be taken. How often will we call `someItem.getBids()` in the application and then access/display *all* bids in a predefined order? If we only want to display a subset of bids, or if we need to retrieve them in a different order every time, then we need to write and execute queries manually anyway. The *one-to-many* mapping and its collection would only be maintenance baggage. In our experience, this is a frequent source of problems and bugs, especially for ORM beginners.

In *CaveatEmptor*'s case, the answer is yes, we frequently call `someItem.getBids()` and then show a list to the user who wants to participate in an auction. Figure 8.15 shows the updated UML diagram with this bidirectional association that we need to implement.

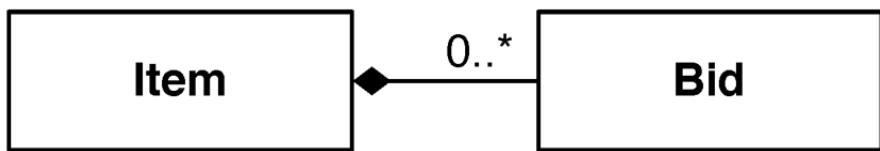


Figure 8.15 Bidirectional association between `Item` and `Bid`

The mapping of the collection and the *one-to-many* side is as follows.

Listing 8.17 `Item` has a collection of `Bid` references

```

Path: Ch08/mapping-
      associations/src/main/java/com/manning/javapersistence/ch08/onetomany/bidirectional/
      Item.java
@Entity
public class Item {
    ...
    @OneToMany(mappedBy = "item",
               fetch = FetchType.LAZY)                      #A
    private Set<Bid> bids = new HashSet<>();          #B
    ...
}

#A The @OneToMany annotation is required to make the association bidirectional. In this case, we also have to set
the mappedBy parameter.
#B The argument is the name of the property on the "other side." The fetching will be LAZY by default.
  
```

#A The `@OneToMany` annotation is required to make the association bidirectional. In this case, we also have to set the `mappedBy` parameter.

#B The argument is the name of the property on the “other side.” The fetching will be `LAZY` by default.

Look again at the other side: the *many-to-one* mapping in listing 8.17. The property name in the `Bid` class is `item`. The bid side is responsible for the foreign key column, `ITEM_ID`, which we mapped with `@ManyToOne`. `mappedBy` tells Hibernate to “load this collection using the foreign key column already mapped by the given property”—in this case, `Bid#item`. The

`mappedBy` parameter is always required when the *one-to-many* is bidirectional when we already mapped the foreign key column. We'll talk about that again in the next chapter.

The default for the `fetch` parameter of a collection mapping is always `FetchType.LAZY`. We won't need this option in the future. It's a good default setting; the opposite would be the rarely needed `EAGER`. We don't want all the `bids` eagerly loaded every time we load an `Item`. They should be loaded when accessed, on-demand.

We create the following Spring Data JPA repositories:

Listing 8.18 The ItemRepository interface

```
Path: Ch08/mapping-  
      associations/src/test/java/com/manning/javapersistence/ch08/repositories/onetomany/b  
      idirectional/ItemRepository.java  
public interface ItemRepository extends JpaRepository<Item, Long> {  
}
```

Listing 8.19 The BidRepository interface

```
Path: Ch08/mapping-  
      associations/src/test/java/com/manning/javapersistence/ch08/repositories/onetomany/b  
      idirectional/BidRepository.java  
public interface BidRepository extends JpaRepository<Item, Long> {  
    Set<Bid> findByItem(Item item);  
}
```

These are usual Spring Data JPA repositories, the `BidRepository` adding a method to get the `bids` by `Item`.

The second reason for mapping the `Item#bids` collection is the ability to cascade state changes.

8.3.3 Cascading state

If an entity state change can be cascaded across an association to another entity, we need fewer lines of code to manage relationships. But this may have a serious performance impact.

The following code creates a new `Item` and a new `Bid` and then links them:

```
Item someItem = new Item("Some Item");  
Bid someBid = new Bid(new BigDecimal("123.00"), someItem);  
someItem.addBid(someBid);
```

We have to consider both sides of this relationship: the `Bid` constructor accepts an `item`, used to populate `Bid#item`. To maintain the integrity of the instances in memory, we need to add the bid to `Item#bids`. Now the link is complete from the perspective of the Java code; all references are set. If you aren't sure why you need this code, please see section 3.2.4.

Let's save the item and its bids in the database, first without and then with transitive persistence.

ENABLING TRANSITIVE PERSISTENCE

With the current mapping of `@ManyToOne` and `@OneToMany`, we need to write the following code to save a new `Item` and several `Bid` instances.

Listing 8.20 Managing independent entity instances separately

```
Path: Ch08/mapping-
      associations/src/test/java/com/manning/javapersistence/ch08/onetomany/bidirectional/
      MappingAssociationsSpringDataJPATest.java

Item item = new Item("Foo");
Bid bid = new Bid(BigDecimal.valueOf(100), item);
Bid bid2 = new Bid(BigDecimal.valueOf(200), item);

itemRepository.save(item);
item.addBid(bid);
item.addBid(bid2);
bidRepository.save(bid);
bidRepository.save(bid2);
```

When we create several bids, calling `EntityManager#persist()` or `BidRepository#save()` on each seems redundant. New instances are transient and have to be made persistent. The relationship between `Bid` and `Item` doesn't influence their life cycle. If `Bid` were to be a value type, the state of a `Bid` would be automatically the same as the owning `Item`. In this case, however, `Bid` has its own completely independent state.

We said earlier that fine-grained control is sometimes necessary to express the dependencies between associated entity classes; this is such a case. The mechanism for this in JPA is the `cascade` option. For example, to save all bids when the item is saved, map the collection as demonstrated next.

Listing 8.21 Cascading persistent state from Item to all bids

```
Path: Ch08/mapping-
      associations/src/main/java/com/manning/javapersistence/ch08/onetomany/cascadepersist
      /Item.java

@Entity
public class Item {
    ...

    @OneToMany(mappedBy = "item", cascade = CascadeType.PERSIST)
    private Set<Bid> bids = new HashSet<>();
    ...
}
```

Cascading options are per operation we'd like to be transitive, so we use `CascadeType.PERSIST` for the `ItemRepository#save()` or `EntityManager#persist()` operation. We can now simplify the code that links items and bids and then saves them.

Listing 8.22 All referenced bids are automatically made persistent

```
Path: Ch08/mapping-
      associations/src/test/java/com/manning/javapersistence/ch08/onetomany/cascadepersist
      /MappingAssociationsSpringDataJPATest.java
Item item = new Item("Foo");

Bid bid = new Bid(BigDecimal.valueOf(100), item);
Bid bid2 = new Bid(BigDecimal.valueOf(200), item);
item.addBid(bid);
item.addBid(bid2);

itemRepository.save(item);                                     #A
```

#A We save the bids automatically but later. At commit time, Spring Data JPA using Hibernate examines the managed/persistent Item instance and looks into the bids collection. It then calls save() internally on each of the referenced Bid instances, saving them as well. The value stored in the column BID#ITEM_ID is taken from each Bid by inspecting the Bid#item property. The foreign key column is “mapped by” with @ManyToOne on that property.

The @ManyToOne annotation also has the cascade option. We won’t use this often. For example, we can’t really say, “when the bid is saved, also save the item”. The item has to exist beforehand; otherwise, the bid won’t be valid in the database. Think about another possible @ManyToOne: the Item#seller property. The User has to exist before they can sell an Item.

Transitive persistence is a simple concept, frequently useful with @OneToMany or @ManyToMany mappings. On the other hand, we have to apply transitive deletion carefully.

CASCADING DELETION

It seems reasonable that deletion of an item implies deletion of all the bids for the item because they’re no longer relevant alone. This is what the composition (the filled-out diamond) in the UML diagram means. With the current cascading options, we have to write the following code to delete an item:

```
Path: Ch08/mapping-
      associations/src/test/java/com/manning/javapersistence/ch08/onetomany/cascadepersist
      /MappingAssociationsSpringDataJPATest.java
Item retrievedItem = itemRepository.findById(item.getId()).get();

for (Bid someBid : bidRepository.findByItem(retrievedItem)) {
    bidRepository.delete(someBid);                                #A
}

itemRepository.delete(retrievedItem);                           #B
```

#A First, we remove the bids.

#B Then, we remove the Item owner.

JPA offers a cascading option to help with this. The persistence engine can remove an associated entity instance automatically.

Listing 8.23 Cascading removal from Item to all bids

```
Path: Ch08/mapping-
      associations/src/main/java/com/manning/javapersistence/ch08/onetomany/cascaderemove/
      Item.java
@Entity
public class Item {
    ...
    @OneToMany(mappedBy = "item",
               cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private Set<Bid> bids = new HashSet<>();
    ...
}
```

Just as before with `PERSIST`, the `delete()` operations on this association will be cascaded. If we call `ItemRepository#delete()` or `EntityManager#remove()` on an `Item`, Hibernate loads the `bids` collection elements and internally calls `remove()` on each instance:

```
Path: Ch08/mapping-
      associations/src/test/java/com/manning/javapersistence/ch08/onetomany/cascaderemove/
      MappingAssociationsSpringDataJPATest.java
itemRepository.delete(item);
```

One line of code will be enough to also delete bids one by one.

This deletion process is inefficient: Hibernate or Spring Data JPA must always load the collection and delete each `Bid` individually. A single SQL statement would have the same effect on the database: `delete from BID where ITEM_ID = ?`.

We know this because nobody in the database has a foreign key reference on the `BID` table. Hibernate doesn't know this and can't search the whole database for any row that might have a `BID_ID`.

If `Item#bids` was instead a collection of embeddable components, `someItem.getBids().clear()` would execute a single SQL `DELETE`. With a collection of value types, Hibernate assumes that nobody can possibly hold a reference to the bids, and removing only the reference from the collection makes it orphan removable data.

ENABLING ORPHAN REMOVAL

JPA offers a (questionable) flag that enables the same behavior for `@OneToMany` (and only `@OneToMany`) entity associations.

Listing 8.24 Enabling orphan removal on a @OneToOne collection

```
Path: Ch08/mapping-
      associations/src/main/java/com/manning/javapersistence/ch08/onetomany/orphanremoval/
      Item.java
@Entity
public class Item {
    ...
    @OneToMany(mappedBy = "item",
               cascade = CascadeType.PERSIST, orphanRemoval = true)
    private Set<Bid> bids = new HashSet<>();
    ...
}
```

The `orphanRemoval=true` argument tells Hibernate that we want to permanently remove a `Bid` when it's removed from the collection.

We'll change the `ItemRepository` interface as in the following listing:

Listing 8.25 The modified ItemRepository interface

```
Path: Ch08/mapping-
      associations/src/test/java/com/manning/javapersistence/ch08/repositories/onetomany/o
      rphanremoval/ItemRepository.java
public interface ItemRepository extends JpaRepository<Item, Long> {

    @Query("select i from Item i inner join fetch i.bids where i.id = :id") #A
    Item findItemWithBids(@Param("id") Long id); #A
}
```

#A The `findItemWithBids` method that was added will get the `Item` by id, including the `bids` collection. To fetch this collection, we will use the inner join fetch capability of JPQL (Jakarta Persistence Query Language).

Here is an example of deleting a single `Bid`:

```
Path: Ch08/mapping-
      associations/src/test/java/com/manning/javapersistence/ch08/onetomany/orphanremoval/
      MappingAssociationsSpringDataJPATest.java
Item item1 = itemRepository.findItemWithBids(item.getId());
Bid firstBid = item1.getBids().iterator().next();
item1.removeBid(firstBid);

itemRepository.save(item1);
```

Hibernate or Spring Data JPA using Hibernate monitor the collection, and on transaction commit will notice that we removed an element from the collection. Hibernate now considers the `Bid` to be orphaned. We guarantee that nobody else had a reference to it; the only reference was the one we just removed from the collection. Hibernate or Spring Data JPA using Hibernate automatically execute an SQL `DELETE` to remove the `Bid` instance in the database.

We still won't get the `clear()` one-shot `DELETE` as with a collection of components. Hibernate respects the regular entity-state transitions, and the bids are all loaded and removed individually.

Why is orphan removal questionable? Well, it's fine in this example case. There is so far no other table in the database with a foreign key reference on `BID`. There are no consequences to deleting a row from the `BID` table; the only in-memory references to bids are in `Item#bids`. As long as all of this is true, there is no problem with enabling orphan removal. It's a convenient option, for example, when the presentation layer can remove an element from a collection to delete something; we only work with domain model instances, and we don't need to call a service to perform this operation.

Consider what happens when we create a `User#bids` collection mapping—another `@OneToMany`, as shown in figure 8.16. This is a good time to test your knowledge of Hibernate: what will the tables and schema look like after this change? (Answer: The `BID` table has a `BIDDER_ID` foreign key column, referencing `USERS`.)



Figure 8.16 Bidirectional associations between Item, Bid, and User

The test shown in the following listing won't pass.

Listing 8.26 Hibernate and Spring Data JPA don't clean up in-memory references after database removal

```

Path: Ch08/mapping-
      associations/src/test/java/com/manning/javapersistence/ch08/onetomany/orphanremoval/
      MappingAssociationsSpringDataJPATest.java
User user = userRepository.findUserWithBids(john.getId());
assertEquals(2, user.getBids().size());
Item item1 = itemRepository.findItemWithBids(item.getId());
Bid firstBid = item1.getBids().iterator().next();
item1.removeBid(firstBid);
itemRepository.save(item1);
//FAILURE
//assertEquals(1, user.getBids().size());
assertEquals(2, user.getBids().size());
  
```

Hibernate or Spring Data JPA think the removed `Bid` is orphaned and deletable; it will be deleted automatically in the database, but we still hold a reference to it in the other collection, `User#bids`. The database state is fine when this transaction commits; the deleted row of the `BID` table contained both foreign keys, `ITEM_ID` and `BIDDER_ID`. We have an inconsistency in memory because saying, “Remove the entity instance when the reference is removed from the collection” naturally conflicts with shared references.

Instead of orphan removal, or even `CascadeType.REMOVE`, always consider a simpler mapping. Here, `Item#bids` would be fine as a collection of components, mapped with `@ElementCollection`. The `Bid` would be `@Embeddable` and have an `@ManyToOne` bidder

property, referencing a `User`. (Embeddable components can own unidirectional associations to entities.)

This would provide the life cycle we're looking for: a full dependency on the owning entity. We have to avoid shared references; the UML diagram (figure 8.16) makes the association from `Bid` to `User` unidirectional. Drop the `User#bids` collection; we don't need this `@OneToMany`. If we need all the bids made by a user, write a query: `select b from Bid b where b.bidder = :userParameter`. (In the next chapter, we'll complete this mapping with an `@ManyToOne` in an embeddable component.)

ENABLING ON DELETE CASCADE ON THE FOREIGN KEY

All the removal operations we've shown are inefficient; bids have to be loaded into memory, and many SQL `DELETEs` are necessary. SQL databases support a more efficient foreign key feature: the `ON DELETE` option. In DDL, it looks like this: `foreign key (ITEM_ID) references ITEM on delete cascade` for the `BID` table.

This option tells the database to maintain the referential integrity of composites transparently for all applications accessing the database. Whenever we delete a row in the `ITEM` table, the database will automatically delete any row in the `BID` table with the same `ITEM_ID` key value. We only need one `DELETE` statement to remove all dependent data recursively, and nothing has to be loaded into application (server) memory.

You should check whether your schema already has this option enabled on foreign keys. If you want this option added to the Hibernate-generated schema, use the Hibernate `@OnDelete` annotation.

You should also check if this option works with your DBMS and if Hibernate or Spring Data JPA using Hibernate generates indeed a foreign key with the `ON DELETE CASCADE` option. As it does not work with MySQL, we chose this particular example to be demonstrated on the H2 database. You will find it this way in the source code (Maven dependency in `pom.xml` and Spring Data JPA configuration).

Listing 8.27 Generating foreign key ON DELETE CASCADE in the schema

```
Path: Ch08/mapping-
      associations/src/main/java/com/manning/javapersistence/ch08/onetomany/onDeletecascade
      e/Item.java
@Entity
public class Item {
    ...

    @OneToMany(mappedBy = "item", cascade = CascadeType.PERSIST)
    @org.hibernate.annotations.OnDelete(
        action = org.hibernate.annotations.OnDeleteAction.CASCADE
    )
    private Set<Bid> bids = new HashSet<>();
    ...
}
```

#A One of the Hibernate quirks is visible here: the `@OnDelete` annotation affects only the schema generation by Hibernate. Settings that affect schema generation are usually on the “other” mappedBy side, where the foreign key/join column is mapped. The `@OnDelete` annotation is usually next to the `@ManyToOne` in `Bid`. When the association is mapped bidirectional, however, Hibernate will only recognize it on the `@OneToMany` side.

Enabling foreign key cascade deletion in the database doesn't influence Hibernate's runtime behavior. We can still run into the same problem as shown in listing 8.26. Data in memory may no longer accurately reflect the state in the database. If all related rows in the `BID` table are automatically removed when a row in the `ITEM` table is deleted, the application code is responsible for cleaning up references and catching up with the database state. If we aren't careful, we may even end up saving something that we or someone else previously deleted.

The `Bid` instances don't go through the regular life cycle, and callbacks such as `@PreRemove` have no effect. Additionally, Hibernate doesn't automatically clear the optional second-level global cache, which potentially contains stale data. Fundamentally, the kinds of problems encountered with database-level foreign key cascading are the same as when another application besides ours is accessing the same database, or any other database trigger makes changes. Hibernate can be a very effective utility in such a scenario, but there are other moving parts to consider.

If you work on a new schema, the easiest approach is to not enable database-level cascading and map a composition relationship in your domain model as embedded/embeddable, not as an entity association. Hibernate or Spring Data JPA using Hibernate can then execute efficient SQL `DELETE` operations to remove the entire composite. We made this recommendation in the previous section: if you can avoid shared references, map the `Bid` as an `@ElementCollection` in `Item`, not as a standalone entity with `@ManyToOne` and `@OneToMany` associations. Alternatively, of course, you might not map any collections at all and use only the simplest mapping: a foreign key column with `@ManyToOne`, unidirectional between `@Entity` classes.

8.4 Summary

- Using simple collection mappings, such as a `Set<String>`, we worked through a rich set of interfaces and implementations.
- We demonstrated how sorted collections work as well as Hibernate's options for letting the database return the collection elements in the desired order.
- We analyzed complex collections of user-defined embeddable types and sets, bags, and maps of components.
- We used components as both keys and values in maps and a collection in an embeddable component.
- Mapping the first foreign key column to an entity many-to-one association makes it bidirectional as a one-to-many. We also implemented several cascading options.
- We investigated key concepts of object/relational mapping. Once you've mapped your first `@ManyToOne` and maybe a simple collection of strings, the worst will be behind you.
- Be sure you try the code (and watch the SQL log)!

9

Advanced entity association mappings

This chapter covers

- Applying mapping through one-to-one entity associations
- Using one-to-many mapping options
- Creating many-to-many and ternary entity relationships

In the previous chapter, we demonstrated a unidirectional *many-to-one* association, made it bidirectional, and finally enabled transitive state changes with cascading options. One reason we discuss more advanced entity mappings in a separate chapter is that we consider quite a few of them rare or at least optional. It's possible to only use component mappings and *many-to-one* (occasionally *one-to-one*) entity associations. You can write a sophisticated application without ever mapping a collection! We've demonstrated the particular benefits to gain from collection mappings in the previous chapter; the rules for when a collection mapping is appropriate also apply to all examples in this chapter. Always make sure you actually need a collection before attempting a complex collection mapping.

Let's start with mappings that don't involve collections: *one-to-one* entity associations.

Major new features in JPA 2

- *Many-to-one* and *one-to-one* associations may now be mapped with an intermediate join/link table.
- Embeddable component classes may have unidirectional associations to entities, even many-valued with collections.

9.1 One-to-one associations

We argued in section 6.2 that the relationships between `User` and `Address` (the user has a `billingAddress`, `homeAddress`, and `shippingAddress`) are best represented with an `@Embeddable` component mapping. This is usually the simplest way to represent *one-to-one* relationships because the life cycle is typically dependent in such a case. It's either aggregation or composition in UML.

What about using a dedicated `ADDRESS` table and mapping both `User` and `Address` as entities? One benefit of this model is the possibility for shared references—another entity class (let's say `Shipment`) can also have a reference to a particular `Address` instance. If a `User` also has a reference to this instance, as their `shippingAddress`, the `Address` instance has to support shared references and needs its own identity.

In this case, `User` and `Address` classes have a true *one-to-one* association. Look at the revised class diagram in figure 9.1.

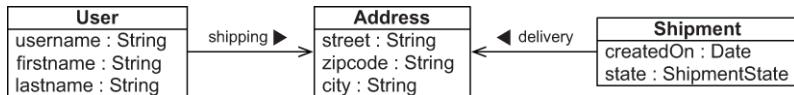


Figure 9.1 Address as an entity with two associations, supporting shared references

We are working on the `CaveatEmptor` application and need to map the entities from figure 9.1. There are several possible mappings for *one-to-one* associations. The first strategy we'll consider is a shared primary key value. To be able to execute the examples from the source code, you need first to run the `Ch09.sql` script. The source code to follow is to be found in the `onetoone-sharedprimarykey` folder.

9.1.1 Sharing a primary key

Rows in two tables related by a primary key association share the same primary key values. If each user has exactly one shipping address, then the approach is that the `User` has the same primary key value as the (shipping-) `Address`. The main difficulty with this approach is ensuring that associated instances are assigned the same primary key value when the instances are saved. Before looking at this issue, let's create the basic mapping. The `Address` class is now a standalone entity; it's no longer a component.

Listing 9.1 Address class as a standalone entity

```
Path: onetoone-
      sharedprimarykey/src/main/java/com/manning/javapersistence/ch09/onetoone/sharedprimarykey/Address.java
@Entity
public class Address {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    private Long id;
    @NotNull
    private String street;
    @NotNull
    private String zipcode;
    @NotNull
    private String city;
    // ...
}
```

The User class is also an entity with the shippingAddress association property. We'll introduce here two new annotations: `@OneToOne` and `@PrimaryKeyJoinColumn`.

`@OneToOne` does what you'd expect: it's required to mark an entity-valued property as a **one-to-one** association. We'll force the necessity that a User has an Address with the optional=false clause. We'll force cascading any change from User to Address with the cascade = CascadeType.ALL clause.

`@PrimaryKeyJoinColumn` selects the shared primary key strategy we'd like to map.

Listing 9.2 User entity and shippingAddress association

```
Path: onetoone-
      sharedprimarykey/src/main/java/com/manning/javapersistence/ch09/onetoone/sharedprimarykey/User.java
@Entity
@Table(name = "USERS")
public class User {
    @Id
    private Long id; #A

    private String username;

    @OneToOne(
        fetch = FetchType.LAZY,
        optional = false,
        cascade = CascadeType.ALL #B #C #D #E
    )
    @PrimaryKeyJoinColumn #F
    private Address;

    public User() { }

    public User(Long id, String username) { #G
        this.id = id;
        this.username = username;
    }
    // ...
}
```

```
}
```

- #A For the `User`, we don't declare an identifier generator. As mentioned in section 5.2.4, this is one of the rare cases to use an *application-assigned* identifier value.
- #B The relationship between `User` and `Address` is one-to-one.
- #C As usual, we should prefer the lazy-loading strategy, so we override the default `FetchType.EAGER` with `LAZY`.
- #D The `optional=false` switch defines that a `User` must have a `shippingAddress`.
- #E The Hibernate-generated database schema reflects this with a foreign key constraint. Any change here must be cascaded to `Address`. The primary key of the `USERS` table also has a foreign key constraint referencing the primary key of the `ADDRESS` table. See the tables in figure 9.2.
- #F Using `@PrimaryKeyJoinColumn`, this is now a unidirectional shared primary key one-to-one association mapping, from `User` to `Address`.
- #G We can see that the constructor design (weakly) enforces this: the public API of the class requires an identifier value to create an instance.

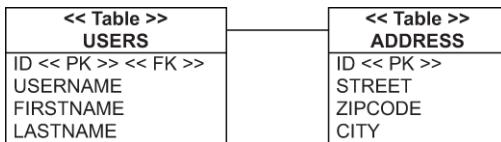


Figure 9.2 The `USERS` table has a foreign key constraint on its primary key.

For some of the examples in this chapter (as is the case now), we need to make a few changes to our usual configuration for the tests, as the execution will need to be transactional. The `SpringDataConfiguration` class will have more annotations:

```
Path: onetoone-
      sharedprimarykey/src/test/java/com/manning/javapersistence/ch09/configuration/onetoone/
      sharedprimarykey/SpringDataConfiguration.java
@Configuration #A
@EnableTransactionManagement #B
@ComponentScan(basePackages = "com.manning.javapersistence.ch09.*") #C
@EnableJpaRepositories("com.manning.javapersistence.ch09.repositories.
      onetoone.sharedprimarykey") #D
public class SpringDataConfiguration {
//...
}
```

`@Configuration` #A tells that this class declares one or more bean definitions to be used by the Spring container. `@EnableTransactionManagement` will enable the transaction management capabilities of Spring through annotations #B. We'll need to execute a few operations in a transactional way to test the code from this chapter. `@ComponentScan` will require Spring to scan for components the package provided as an argument and its sub-packages #C. `@EnableJpaRepositories` will scan the indicated package to find Spring Data repositories #D.

We'll isolate the operations against the database in a dedicated class.

```

Path: onetoone-
      sharedprimarykey/src/test/java/com/manning/javapersistence/ch09/onetoone/sharedprimarykey/TestService.java
@Service #A
public class TestService {
    @Autowired #B
    private UserRepository userRepository; #B

    @Autowired #B
    private AddressRepository addressRepository; #B

    @Transactional #C
    public void storeLoadEntities() { #C
    //...
}

```

#A The `TestService` class is annotated as `@Service` to allow Spring to automatically create a bean, later to be injected in the effective test. Remember that in the `SpringDataConfiguration` class, we scan for components the `com.manning.javapersistence.ch09` package and its sub-packages.

#B We inject two repository beans.

#C We define the `storeLoadEntities` method, annotating it with `@Transactional`. The operations we'll need to execute against the database need to be transactional, and we let Spring control this.

The effective testing class will differ from the previously presented ones, as it will delegate to the `TestService` class. This will allow us to keep the transactional operations isolated in their own method and to call that method from the test.

```

Path: onetoone-
      sharedprimarykey/src/test/java/com/manning/javapersistence/ch09/onetoone/sharedprimarykey/AdvancedMappingSpringDataJPATest.java

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {SpringDataConfiguration.class})
public class AdvancedMappingSpringDataJPATest {

    @Autowired
    private TestService testService;

    @Test
    void testStoreLoadEntities() {
        testService.storeLoadEntities();
    }
}

```

The JPA specification doesn't include a standardized method to deal with the problem of shared primary key generation. This means we're responsible for setting the identifier value of a `User` instance correctly before we save it to the identifier value of the linked `Address` instance:

```

Path: onetoone-
      sharedprimarykey/src/test/java/com/manning/javapersistence/ch09/onetoone/sharedprimarykey/TestService.java

Address address =
    new Address("Flowers Street", "01246", "Boston");
addressRepository.save(address);                                #A

User john = new User(address.getId(),"John Smith");          #B
john.setShippingAddress(address);

userRepository.save(john);                                     #C

```

#A We persist the `Address`.

#B We take its generated identifier value and set it on `User`.

#C We save it.

There are three problems with the mapping and code:

- We have to remember that the `Address` must be saved first and then get its identifier value. This is only possible if the `Address` entity has an identifier generator that produces values on `save()` before the `INSERT`, as we analyzed in section 5.2.5. Otherwise, `someAddress.getId()` returns `null`, and we can't manually set the identifier value of the `User`.
- Lazy loading with proxies only works if the association is non-optional. This is often a surprise for developers new to JPA. The default for `@OneToOne` is `FetchType.EAGER`: when Hibernate or Spring Data JPA using Hibernate load a `User`, it loads the `shippingAddress` right away. Conceptually, lazy loading with proxies only makes sense if Hibernate knows that there is a linked `shippingAddress`. If the property were nullable, Hibernate would have to check in the database whether the property value is `NULL` by querying the `ADDRESS` table. If we have to check the database, we might as well load the value right away because there would be no benefit in using a proxy.
- The `one-to-one` association is unidirectional; sometimes we need bidirectional navigation.

The first issue has no other solution, in the example above we are doing exactly this thing: save the `Address`, get its primary key and manually set it as the identifier value of the `User`. It is one of the reasons we should always prefer identifier generators capable of producing values before any SQL `INSERT`.

An `@OneToOne(optional=true)` association doesn't support lazy loading with proxies. This is consistent with the JPA specification. `FetchType.LAZY` is a hint for the persistence provider, not a requirement. We could get lazy loading of nullable `@OneToOne` with bytecode instrumentation, as we'll show in section 12.1.3.

As for the last problem, if we make the association bidirectional (the `Address` references the `User` and the `User` references the `Address`), we can also use a special Hibernate-only identifier generator to help with assigning key values.

9.1.2 The foreign primary key generator

A bidirectional mapping always requires a `mappedBy` side. The source code to follow is to be found in the `onetoone-foreigngenerator` folder. We'll pick the `User` side (this is a matter of taste and perhaps other, secondary requirements):

```
Path: onetoone-foreigngenerator/src/main/java/com/manning/javapersistence/ch09/onetoone/
      foreigngenerator/User.java
@Entity
@Table(name = "USERS")
public class User {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    private Long id;

    private String username;

    @OneToOne(
        mappedBy = "user",
        cascade = CascadeType.PERSIST
    )
    private Address shippingAddress;
    // ...
}
```

Compare this with the previous mapping: we add the `mappedBy` option, telling Hibernate or Spring Data JPA using Hibernate that the lower-level details are now mapped by the “property on the other side”, named `user`. As a convenience, we enable `CascadeType.PERSIST`; transitive persistence will make it easier to save the instances in the right order. When we make the `User` persistent, Hibernate makes the `shippingAddress` persistent and generates the identifier for the primary key automatically.

Next, let's look at the “other side”: the `Address`.

We'll use the `@GenericGenerator` on the identifier property to define a special-purpose primary key value generator with the Hibernate-only `foreign` strategy. We didn't mention this generator in the overview in section 5.2.5; the shared primary key *one-to-one* association is its only use case. When persisting an instance of `Address`, this special generator grabs the value of the `user` property and takes the identifier value of the referenced entity instance, the `User`.

Listing 9.3 Address has the special foreign key generator

```

Path: onetoone-foreigngenerator/src/main/java/com/manning/javapersistence/ch09/onetooone/
      foreigngenerator/Address.java

@Entity
public class Address {
    @Id
    @GeneratedValue(generator = "addressKeyGenerator")
    @org.hibernate.annotations.GenericGenerator(
        name = "addressKeyGenerator",
        strategy = "foreign",
        parameters =
            @org.hibernate.annotations.Parameter(
                name = "property", value = "user"
            )
    )
    private Long id;

    //...

    @OneToOne(optional = false)                                #B
    @PrimaryKeyJoinColumn                                       #C
    private User user;

    public Address() {                                         #D
    }

    public Address(User user) {                               #D
        this.user = user;
    }

    public Address(User user, String street,                 #D
                   String zipcode, String city) {           #D
        this.user = user;
        this.street = street;
        this.zipcode = zipcode;
        this.city = city;
    }
    // ...
}

```

#A As mentioned, with the `@GenericGenerator` annotation, when we persist an instance of `Address`, this special generator grabs the value of the `user` property and takes the identifier value of the referenced entity instance, the `User`.

#B The `@OneToOne` mapping is set to `optional=false`, so an `Address` must have a reference to a `User`.

#C The `user` property is marked as a shared primary key entity association with the `@PrimaryKeyJoinColumn` annotation.

#D The public constructors of `Address` now require a `User` instance. The foreign key constraint reflecting `optional=false` is now on the primary key column of the `ADDRESS` table, as we can see in the schema in figure 9.3.

That's quite a bit of new code. Let's start with the identifier property and then the *one-to-one* association.

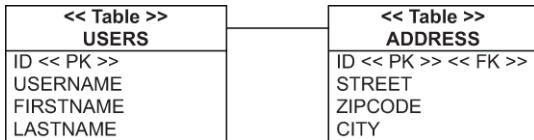


Figure 9.3 The ADDRESS table has a foreign key constraint on its primary key.

We no longer have to call `address.getId()` or `user.getId()` in our unit of work. Storing data is simplified:

```

Path: onetoone-foreigngenerator/src/test/java/com/manning/javapersistence/ch09/onetoone/
      foreigngenerator/AdvancedMappingJPATest.java
User john = new User("John Smith");
Address address =
  new Address(
    john,
    "Flowers Street", "01246", "Boston"
  );
john.setShippingAddress(address);
userRepository.save(john);
  
```

#A We must link both sides of a bidirectional entity association. Note that with this mapping, we won't get lazy loading of `User#shippingAddress` (it's optional/nullable), but we can load `Address#user` on-demand with proxies (it's non-optional).

#B When we persist the user, we'll get the transitive persistence of `shippingAddress`.

Shared primary key *one-to-one* associations are relatively rare. Instead, we'll often map a “to-one” association with a foreign key column and a unique constraint.

9.1.3 Using a foreign key join column

Instead of sharing a primary key, two rows can have a relationship based on a simple additional foreign key column. One table has a foreign key column that references the primary key of the associated table. (The source and target of this foreign key constraint can even be the same table: we call this a *self-referencing relationship*.) The source code to follow is to be found in the `onetoone-foreignkey` folder.

Let's change the mapping for `User#shippingAddress`. Instead of the shared primary key, we now add a `SHIPPINGADDRESS_ID` column in the `USERS` table. Additionally, the column has a `UNIQUE` constraint, so no two users can reference the same shipping address. Look at the schema in figure 9.4.

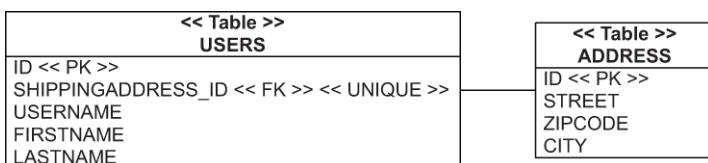


Figure 9.4 A one-to-one join column association between the USERS and ADDRESS tables

The `Address` is a regular entity class, like the first one we demonstrated in this chapter in listing 9.1. The `User` entity class has the `shippingAddress` property, implementing this unidirectional association.

We should enable lazy loading for this `User -> Address` association. Unlike for shared primary keys, we don't have a problem with lazy loading here: when a row of the `USERS` table has been loaded, it contains the value of the `SHIPPINGADDRESS_ID` column. Hibernate or Spring Data using Hibernate, therefore, know whether an `ADDRESS` row is present, and a proxy can be used to load the `Address` instance on demand.

```
Path: onetoone-foreignkey/src/main/java/com/manning/javapersistence/ch09/onetooone/
      foreignkey/User.java
@Entity
@Table(name = "USERS")
public class User {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    private Long id;

    @OneToOne(
        fetch = FetchType.LAZY,
        optional = false,                                     #A
        cascade = CascadeType.PERSIST
    )
    @JoinColumn(unique = true)                                #B
    private Address shippingAddress;
    // ...
}
```

#A We don't need any special identifier generators or primary key assignment; we'll make sure the `shippingAddress` is not null.

#B Instead of `@PrimaryKeyJoinColumn`, we apply the regular `@JoinColumn`, which will default to `SHIPPINGADDRESS_ID`. If you're more familiar with SQL than JPA, it helps to think “foreign key column” every time you see `@JoinColumn` in a mapping.

In the mapping, though, we set `optional=false`, so the user must have a shipping address. This won't affect the loading behavior but is a logical consequence of the `unique=true` setting on the `@JoinColumn`. This setting adds a unique constraint to the generated SQL schema. If the values of the `SHIPPINGADDRESS_ID` column must be unique for all users, only one user could possibly have “no shipping address.” Hence, nullable unique columns typically aren't meaningful.

Creating, linking, and storing instances is straightforward:

```
Path: onetoone-foreignkey/src/test/java/com/manning/javapersistence/ch09/onetooone/
      foreignkey/AdvancedMappingJPATest.java
User john = new User("John Smith");
Address address = new Address("Flowers Street", "01246", "Boston");
john.setShippingAddress(address);                               #A
userRepository.save(john);                                  #B
```

#A We create the link between the user and the address.

#B When we save `john`, we'll transitively save the `address`.

We've now completed two basic *one-to-one* association mappings: the first with a shared primary key, the second with a foreign key reference, and a unique column constraint. The last option we want to discuss is a bit more exotic: mapping a *one-to-one* association with the help of an additional table.

9.1.4 Using a join table

You've probably noticed that nullable columns can be problematic. Sometimes a better solution for optional values is an intermediate table, which contains a row if a link is present or doesn't if not.

Let's consider the `Shipment` entity in `CaveatEmptor` and discuss its purpose. Sellers and buyers interact in `CaveatEmptor` by starting and bidding on auctions. Shipping goods seems outside the scope of the application; the seller and the buyer agree on a method of shipment and payment after the auction ends. They can do this offline, outside of `CaveatEmptor`.

On the other hand, we could offer an escrow service in `CaveatEmptor`. Sellers would use this service to create a trackable shipment once the auction ends. The buyer would pay the price of the auction item to a trustee (us), and we'd inform the seller that the money was available. Once the shipment arrived, and the buyer accepted it, we'd transfer the money to the seller.

If you've ever participated in an online auction of significant value, you've probably used such an escrow service we explained above. But we want more in `CaveatEmptor`: not only will we provide trust services for completed auctions, but we also allow users to create a trackable and trusted shipment for any deal they make outside an auction, outside `CaveatEmptor`.

This scenario calls for a `Shipment` entity with an optional *one-to-one* association to `Item`. Look at the class diagram for this domain model in figure 9.5.

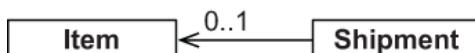


Figure 9.5 A `Shipment` has an optional link with an auction `Item`.

NOTE We briefly considered abandoning the `CaveatEmptor` example for this section because we couldn't find a natural scenario that requires optional *one-to-one* associations. If this escrow example seems contrived, consider the equivalent problem of assigning employees to workstations. This is also an optional *one-to-one* relationship.

In the database schema, we add an intermediate link table called `ITEM_SHIPMENT`. A row in this table represents a `Shipment` made in the context of an auction. Figure 9.6 shows the tables.



Figure 9.6 The intermediate table links items and shipments.

Note how the schema enforces uniqueness and the *one-to-one* relationship: the primary key of `ITEM_SHIPMENT` is the `SHIPMENT_ID` column, and the `ITEM_ID` column is unique. An item can therefore be in only one shipment. Of course, that also means a shipment can contain only one item.

We map this model with a `@OneToOne` annotation in the `Shipment` entity class. The source code to follow is to be found in the `onetoone-jointable` folder.

```

Path: onetoone-jointable/src/main/java/com/manning/javapersistence/ch09/onetoone/
      jointable /Shipment.java
@Entity
public class Shipment {
    /**
     * @OneToOne(fetch = FetchType.LAZY) #A
     * @JoinTable(
     *     name = "ITEM_SHIPMENT",
     *     joinColumns =
     *         @JoinColumn(name = "SHIPMENT_ID"), #B
     *     inverseJoinColumns =
     *         @JoinColumn(name = "ITEM_ID", #C
     *                     nullable = false,
     *                     unique = true) #D
     * )
     * private Item auction; #E
    // ...
}
  
```

#A Lazy loading has been enabled, with a twist: when Hibernate or Spring Data JPA using Hibernate load a `Shipment`, it queries both the `SHIPMENT` and the `ITEM_SHIPMENT` join table. Hibernate has to know if there is a link to an `Item` present before it can use a proxy. It does that in one outer join SQL query, so we won't see any extra SQL statements. If there is a row in `ITEM_SHIPMENT`, Hibernate uses an `Item` placeholder.

#B The `@JoinTable` annotation is new; we always have to specify the name of the intermediate table. This mapping effectively hides the join table; there is no corresponding Java class. The annotation defines the column names of the `ITEM_SHIPMENT` table.

#C The join column is `SHIPMENT_ID` (it would default to `ID`).

#D The inverse join column is `ITEM_ID` # it would default to `AUCTION_ID`.

#E Hibernate generates in the schema the `UNIQUE` constraint on the `ITEM_ID` column. Hibernate also generates the appropriate foreign key constraints on the columns of the join table.

Here we store a `Shipment` without `Items` and another linked to a single `Item`:

```

Path: onetoone-jointable/src/test/java/com/manning/javapersistence/ch09/onetooone/jointable
      /AdvancedMappingSpringDataJPATest.java
Shipment shipment = new Shipment();
shipmentRepository.save(shipment);
Item item = new Item("Foo");
itemRepository.save(item);
Shipment auctionShipment = new Shipment(item);
shipmentRepository.save(auctionShipment);

```

This completes our discussion of *one-to-one* association mappings from which we had to choose. To summarize, use a shared primary key association if one of the two entities is always stored before the other and can act as the primary key source. Use a foreign key association in all other cases and a hidden intermediate join table when the *one-to-one* association is optional.

We now focus on plural, or *many-valued* entity associations, beginning by exploring some advanced options for *one-to-many*.

9.2 One-to-many associations

A *plural entity association* is, by definition, a collection of entity references. We mapped one of these, a *one-to-many* association, in the previous chapter, section 8.3.2. *One-to-many* associations are the most important kind of entity association that involves a collection. We go so far as to discourage the use of more complex association styles when a simple bidirectional *many-to-one/one-to-many* will do the job.

Also, remember that we don't have to map any collection of entities if we don't want to; we can always write an explicit query instead of direct access through iteration. If we decide to map collections of entity references, we have a few options, and we analyze some more complex situations now.

9.2.1 Considering one-to-many bags

So far, we have only seen a `@OneToMany` on a `Set`, but it's possible to use a bag mapping instead for a bidirectional *one-to-many* association. Why would we do this?

Bags have the most efficient performance characteristics of all the collections we can use for a bidirectional *one-to-many* entity association. By default, collections in Hibernate are loaded when they're accessed for the first time in the application. Because a bag doesn't have to maintain the index of its elements (like a list) or check for duplicate elements (like a set), we can add new elements to the bag without triggering the loading. This is an important feature if we're going to map a possibly large collection of entity references.

On the other hand, we can't eager-fetch two collections of bag type simultaneously, as the generated **SELECT queries are unrelated and need to be kept separately**. This may happen, for example, if `bids` and `images` of an `Item` were *one-to-many* bags. This is no big loss because fetching two collections simultaneously always results in a Cartesian product; we want to avoid this kind of operation whether the collections are bags, sets, or lists. We'll come back to fetching strategies in chapter 12. In general, we'd say that a bag is the best inverse collection for a *one-to-many* association if it is mapped as a `@OneToMany(mappedBy = "...")`.

To map a bidirectional *one-to-many* association as a bag, we have to replace the type of the `bids` collection in the `Item` entity with a `Collection` and an `ArrayList` implementation. The source code to follow is to be found in the `onetomany-bag` folder. The mapping for the association between `Item` and `Bid` remains essentially unchanged:

```
Path: onetomany-bag/src/main/java/com/manning/javapersistence/ch09/onetomany/bag/Item.java
@Entity
public class Item {
    ...
    @OneToMany(mappedBy = "item")
    private Collection<Bid> bids = new ArrayList<>();
    ...
}
```

The `Bid` side with its `@ManyToOne` (which is the “mapped by” side), and even the tables, are the same as in section 8.3.1.

A bag also allows duplicate elements, which the set does not:

```
Path: onetomany-bag/src/test/java/com/manning/javapersistence/ch09/onetomany/bag/
AdvancedMappingSpringDataJPATest.java
Item item = new Item("Foo");
itemRepository.save(item);
Bid someBid = new Bid(new BigDecimal("123.00"), item);
item.addBid(someBid);
item.addBid(someBid);
bidRepository.save(someBid);
assertEquals(2, someItem.getBids().size());
```

It turns out this isn’t relevant in this case because *duplicate* means we’ve added a particular reference to the same `Bid` instance several times #A. We wouldn’t do this in our application code. Even if we add the same reference several times to this collection, though, Hibernate or Spring Data JPA using Hibernate ignore it – there is no persistent effect. The side relevant for updates of the database is the `@ManyToOne`, and the relationship is already “mapped by” that side. When we load the `Item`, the collection doesn’t contain the duplicate:

```
Path: onetomany-bag/src/test/java/com/manning/javapersistence/ch09/onetomany/bag/
AdvancedMappingSpringDataJPATest.java
Item item2 = itemRepository.findItemWithBids(item.getId());
assertEquals(1, item2.getBids().size());
```

As mentioned, the advantage of bags is that the collection doesn’t have to be initialized when we add a new element:

```
Path: onetomany-bag/src/test/java/com/manning/javapersistence/ch09/onetomany/bag/
AdvancedMappingSpringDataJPATest.java
Bid bid = new Bid(new BigDecimal("456.00"), item);                      #A
item.addBid(bid);
bidRepository.save(bid);
```

#A This code example triggers one SQL SELECT to load the `Item`. Hibernate still initializes and returns an `Item` proxy with a `SELECT` as soon as we call `item.addBid()`. But as long as we don’t iterate the `Collection`, no more queries are necessary, and an `INSERT` for the new `Bid` will be made without loading all the bids. If the collection is a `Set` or a `List`, Hibernate loads all the elements when we add another element.

We'll now change the collection to a persistent List.

9.2.2 Unidirectional and bidirectional list mappings

If we need a real list to hold the position of the elements in a collection, we have to store that position in an additional column. For the *one-to-many* mapping, this also means we should change the `Item#bids` property to `List` and initialize the variable with an `ArrayList`. There will be a unidirectional mapping: there will be no other "mapped by" side. The `Bid` will not have a `@ManyToOne` property. For persistent list indexes, we'll use the annotation `@OrderColumn`.

The source code to follow is to be found in the `onetomany-list` folder.

```
Path: onetomany-list/src/main/java/com/manning/javapersistence/ch09/onetomany/list
      /Item.java
@Entity
public class Item {
    @OneToOne
    @JoinColumn(
        name = "ITEM_ID",
        nullable = false
    )
    @OrderColumn(
        name = "BID_POSITION",           #A
        nullable = false                #B
    )
    private List<Bid> bids = new ArrayList<>();
    // ...
}
```

#A As mentioned, this is a unidirectional mapping: there is no other "mapped by" side. The `Bid` doesn't have a `@ManyToOne` property. The annotation `@OrderColumn` will make the name of the index column to be set to `BID_POSITION`. Otherwise, it will default to `BIDS_ORDER`.

#B As usual, we should make the column NOT NULL.

The database view of the `BID` table, with the join and order columns, is shown in figure 9.7.

BID			
ID	ITEM_ID	BID_POSITION	AMOUNT
1	1	0	99.00
2	1	1	100.00
3	1	2	101.00
4	2	0	4.99

Figure 9.7 The `BID` table contains the `ITEM_ID` (join column) and `BID_POSITION` (order column)

The stored index of each collection starts at zero and is contiguous (there are no gaps). Hibernate or Spring Data JPA using Hibernate will execute potentially many SQL statements when we add, remove, and shift elements of the `List`. We talked about this performance issue in section 8.1.6.

Let's make this mapping bidirectional, with a `@ManyToOne` property on the `Bid` entity:

```
Path: onetomany-
      list/src/main/java/com/manning/javapersistence/ch09/onetomany/list/Bid.java
@Entity
public class Bid {
    ...

    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        updatable = false, insertable = false
    ) #A
    @NotNull
    private Item item;
    // ...
}
```

#A The `Item#bids` collection is no longer read-only because Hibernate now has to store the index of each element. If the `Bid#item` side was the owner of the relationship, Hibernate would ignore the collection when storing data and not write the element indexes. We have to map the `@JoinColumn` twice and then disable writing on the `@ManyToOne` side with `updatable=false` and `insertable=false`. Hibernate now considers the collection side when storing data, including the index of each element. The `@ManyToOne` is effectively read-only, as it would be if it had a `mappedBy` attribute.

You probably expected different code—maybe `@ManyToOne(mappedBy="bids")` and no additional `@JoinColumn` annotation. But `@ManyToOne` doesn't have a `mappedBy` attribute: it's always the “owning” side of the relationship. We'd have to make the other side, `@OneToMany`, the `mappedBy` side. Here we run into a conceptual problem and some Hibernate quirks.

Finally, the Hibernate schema generator always relies on the `@JoinColumn` of the `@ManyToOne` side. Hence, if we want the correct schema produced, we should add the `@NotNull` on this side or declare `@JoinColumn(nullable=false)`. The generator ignores the `@OneToMany` side and its join column if there is a `@ManyToOne`.

In a real application, we wouldn't map the association with a `List`. Preserving the order of elements in the database seems like a common use case, but on second thought, it isn't very useful: sometimes we want to show a list with the highest or newest bid first, or only bids made by a certain user, or bids made within a certain time range. None of these operations requires a persistent list index. As mentioned in section 3.2.4, avoid storing a display order in the database, as in general the display order may change frequently; keep it flexible with queries instead of hardcoded mappings. Furthermore, maintaining the index when the application removes, adds, or shifts elements in the list can be expensive and may trigger many SQL statements. Map the foreign key join column with `@ManyToOne`, and drop the collection.

Next, we'll work on one more scenario with a *one-to-many* relationship: an association mapped to an intermediate join table.

9.2.3 Optional one-to-many with a join table

A useful addition to the `Item` class is a `buyer` property. We can then call `someItem.getBuyer()` to access the `User` who made the winning bid. If made bidirectional,

this association will also help to render a screen that shows all auctions a particular user has won: we call `someUser.getBoughtItems()` instead of writing a query.

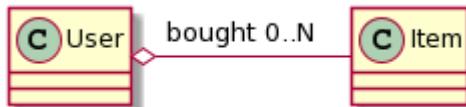


Figure 9.8 The User-Item “bought” relationship

Why is this association different from the one between `Item` and `Bid`? The multiplicity `0..*` in UML indicates that the reference is optional. This doesn’t influence the Java domain model much, but it has consequences for the underlying tables. We expect a `BUYER_ID` foreign key column in the `ITEM` table. The column has to be nullable because a user may not have bought a particular `Item` (as long as the auction is still running).

We could accept that the foreign key column can be `NULL` and apply additional constraints: “Allowed to be `NULL` only if the auction end time hasn’t been reached or if no bid has been made.” We always try to avoid nullable columns in a relational database schema. Unknown information degrades the quality of the data we store. Tuples represent propositions that are true; we can’t assert something we don’t know. Moreover, in practice, many developers and DBAs don’t create the right constraint and rely on often buggy application code to provide data integrity.



Figure 9.9 An intermediate table links users and items.

We added a join table earlier in this chapter for a *one-to-one* association. To guarantee the multiplicity of *one-to-one*, we applied a unique constraint on a foreign key column of the join table. In the current case, we have a *one-to-many* multiplicity, so only the `ITEM_ID` primary key column has to be unique: only one `User` can buy any given `Item` once. The `BUYER_ID` column isn’t unique because a `User` can buy many `Items`. The source code to follow is to be found in the `onetomany-jointable` folder.

The mapping of the `User#boughtItems` collection is simple:

```
Path: onetomany-
      jointable/src/main/java/com/manning/javapersistence/ch09/onetomany/jointable/User.java
      va
@Entity
@Table(name = "USERS")
public class User {
    ...
    @OneToOne(mappedBy = "buyer")
    private Set<Item> boughtItems = new HashSet<>();
    ...
}
```

This is the usual read-only side of a bidirectional association, with the actual mapping to the schema on the “mapped by” side, the `Item#buyer`. It will be a clean, optional *one-to-many/many-to-one* relationship.

```
Path: onetomany-
      jointable/src/main/java/com/manning/javapersistence/ch09/onetomany/jointable/Item.java
      va
@Entity
public class Item {
    ...
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinTable(
        name = "ITEM_BUYER",
        joinColumns =
            @JoinColumn(name = "ITEM_ID"), #A
        inverseJoinColumns =
            @JoinColumn(nullable = false) #B
    )
    private User buyer;
    ...
}
```

#A If an `Item` hasn't been bought, there is no corresponding row in the join table `ITEM_BUYER`. The relationship will thus be optional. The join column is named `ITEM_ID` (it would default to `ID`).

#B The inverse join column will default to `BUYER_ID`, and it is not nullable.

We don't have any problematic nullable columns in our schema. Still, we should write a procedural constraint and a trigger that runs on `INSERT` for the `ITEM_BUYER` table: “Only allow insertion of a buyer if the auction end time for the given item has been reached and the user made the winning bid.”

The next example is our last with *one-to-many* associations. So far, you've seen *one-to-many* associations from an entity to another entity. An embeddable component class may also have a *one-to-many* association to an entity – and this is what we'll deal with now.

9.2.4 One-to-many association in an embeddable class

Consider again the embeddable component mapping we've been repeating for a few chapters: the `Address` of a `User`. We now extend this example by adding a *one-to-many* association from `Address` to `Shipment`: a collection called `deliveries`. Figure 9.10 shows the UML class diagram for this model.

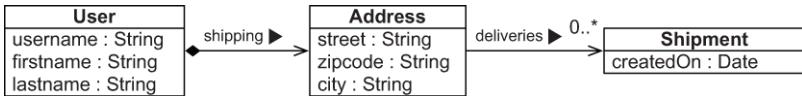


Figure 9.10 The one-to-many relationship from Address to Shipment

The `Address` is an `@Embeddable` class, not an entity. It can own a unidirectional association to an entity; here, it's a *one-to-many* multiplicity to `Shipment`. (We see an embeddable class having a *many-to-one* association with an entity in the next section.) The source code to follow is to be found in the `onetomany-embeddable` folder.

The `Address` class has a `Set<Shipment>` representing this association:

```

Path: onetomany-embeddable
      /src/main/java/com/manning/javapersistence/ch09/onetomany/embeddable/Address.java
@Embeddable
public class Address {
    @NotNull
    @Column(nullable = false)
    private String street;
    @NotNull
    @Column(nullable = false, length = 5)
    private String zipcode;
    @NotNull
    @Column(nullable = false)
    private String city;
    @OneToMany
    @JoinColumn(
        name = "DELIVERY_ADDRESS_USER_ID",
        nullable = false
    )
    private Set<Shipment> deliveries = new HashSet<>();
    // ...
}

#A The first mapping strategy for this association is with an @JoinColumn named DELIVERY_ADDRESS_USER_ID (it would default to DELIVERIES_ID).
  
```

This foreign key-constrained column is in the `SHIPMENT` table, as we can see in figure 9.11.

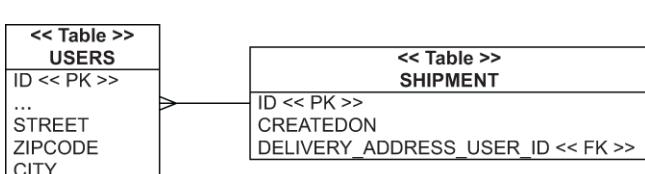


Figure 9.11 A primary key in the `USERS` table links the `USERS` and `SHIPMENT` tables

Embeddable components don't have their own identifier, so the value in the foreign key column is the value of a `User`'s identifier, which embeds the `Address`. Here we also declare the join column `nullable = false`, so a `Shipment` must have an associated delivery address.

Of course, bidirectional navigation isn't possible: the `Shipment` can't have a reference to the `Address` because embedded components can't have shared references.

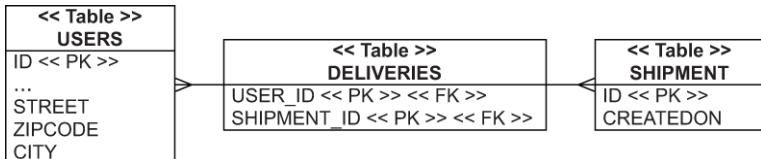


Figure 9.12 Using an intermediate table between `USERS` and `SHIPMENT` to represent an optional association

If the association is optional and we don't want a nullable column, we can map the association to an intermediate join/link table, as shown in figure 9.12. The mapping of the collection in `Address` now uses an `@JoinTable` instead of an `@JoinColumn`. The source code to follow is to be found in the `onetomany-embeddable-jointable` folder.

```

Path: onetomany-embeddable-jointable
      /src/main/java/com/manning/javapersistence/ch09/onetomany/ embeddablejointable
      /Address.java
@Embeddable
public class Address {
    @NotNull
    @Column(nullable = false)
    private String street;
    @NotNull
    @Column(nullable = false, length = 5)
    private String zipcode;
    @NotNull
    @Column(nullable = false)
    private String city;

    @OneToMany
    @JoinTable(
        name = "DELIVERIES",                                     #A
        joinColumns =
            @JoinColumn(name = "USER_ID"),                      #B
        inverseJoinColumns =
            @JoinColumn(name = "SHIPMENT_ID")                   #C
    )
    private Set<Shipment> deliveries = new HashSet<>();
    // ...
}
  
```

#A The name of the join table will be `DELIVERIES` (it would otherwise default to `USERS_SHIPMENT`).

#B The name of the join column will be `USER_ID` (it would otherwise default to `USERS_ID`).

#C The name of the inverse join column will be `SHIPMENT_ID` (it would otherwise default to `SHIPMENTS_ID`).

Note that if we declare neither `@JoinTable` nor `@JoinColumn`, the `@OneToMany` in an embeddable class defaults to a join table strategy.

From within the owning entity class, we can override property mappings of an embedded class with `@AttributeOverride`, as demonstrated in section 6.2.3. If we want to override the join table or column mapping of an entity association in an embeddable class, we use

`@AssociationOverride` in the owning entity class instead. We can't, however, switch the mapping strategy; the mapping in the embeddable component class decides whether a join table or join column is used.

A join table mapping is, of course, also applicable in true *many-to-many* mappings.

9.3 Many-to-many and ternary associations

The association between `Category` and `Item` is a *many-to-many* association, as we can see in figure 9.13. In a real system, we may not have a *many-to-many* association. Our experience is that there is almost always other information that must be attached to each link between associated instances. Some examples are the timestamp when an `Item` was added to a `Category` and the `User` responsible for creating the link. We expand the example later in this section to cover such a case. We'll start with a regular and simpler *many-to-many* association.



Figure 9.13 A *many-to-many* association between `Category` and `Item`

9.3.1 Unidirectional and bidirectional many-to-many associations

A join table in the database represents a regular *many-to-many* association, which some developers also call the *link table* or *association table*. Figure 9.14 shows a *many-to-many* relationship with a link table.

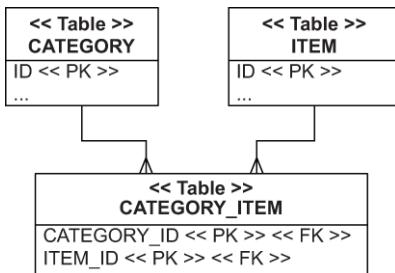


Figure 9.14 `CategorizedItem` is the link between `Category` and `Item`.

The link table `CATEGORY_ITEM` has two columns, both with a foreign key constraint referencing the `CATEGORY` and `ITEM` tables, respectively. Its primary key is a composite key of both columns. We can only link a particular `Category` and `Item` once, but we can link the same item to several categories. The source code to follow is to be found in the `manytomany-bidirectional` folder.

In JPA, we map *many-to-many* associations with `@ManyToMany` on a collection:

```
Path: manytomany-bidirectional
      /src/main/java/com/manning/javapersistence/ch09/manytomany/bidirectional/Category.java
      va
@Entity
public class Category {
    ...
    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID")
    )
    private Set<Item> items = new HashSet<>();
    ...
}
```

As usual, we can enable `CascadeType.PERSIST` to make it easier to save data. When we reference a new `Item` from the collection, Hibernate or Spring Data JPA using Hibernate make it persistent. Let's make this association bidirectional (we don't have to if we don't need it):

```
Path: manytomany-bidirectional
      /src/main/java/com/manning/javapersistence/ch09/manytomany/bidirectional/Item.java
@Entity
public class Item {
    ...
    @ManyToMany(mappedBy = "items")
    private Set<Category> categories = new HashSet<>();
    ...
}
```

As for any bidirectional mapping, one side is "mapped by" the other side. The `Item#categories` collection is effectively read-only; Hibernate will analyze the content of the `Category#items` side when storing data. Next we create two categories and two items and link them with *many-to-many* multiplicity:

```
Path: manytomany-bidirectional
      /src/test/java/com/manning/javapersistence/ch09/manytomany/bidirectional/TestService.java
Category someCategory = new Category("Some Category");
Category otherCategory = new Category("Other Category");
Item someItem = new Item("Some Item");
Item otherItem = new Item("Other Item");
someCategory.addItem(someItem);
someItem.addCategory(someCategory);
someCategory.addItem(otherItem);
otherItem.addCategory(someCategory); Hint:
otherCategory.addItem(someItem);
someItem.addCategory(otherCategory);
categoryRepository.save(someCategory);
categoryRepository.save(otherCategory);
```

Because we enabled transitive persistence, saving the categories makes the entire network of instances persistent. On the other hand, the cascading options `ALL`, `REMOVE`, and orphan deletion (see section 8.3.3) aren't meaningful for *many-to-many* associations. This is a good

point to test whether we understand entities and value types. Try to come up with reasonable answers as to why these cascading types don't make sense for a *many-to-many* association. Hint: think about what may happen if deleting a record will automatically delete a related record.

Can we use a `List` instead of a `Set`, or even a `bag`? The `Set` matches the database schema perfectly because there can be no duplicate links between `Category` and `Item`.

A `bag` implies duplicate elements, so we need a different primary key for the join table. The proprietary `@CollectionId` annotation of Hibernate can provide this, as demonstrated in section 8.1.5. One of the alternative *many-to-many* strategies we discuss in a moment is a better choice if we need to support duplicate links.

We can map indexed collections such as a `List` with the regular `@ManyToMany`, but only on one side. Remember that in a bidirectional relationship, one side has to be "mapped by" the other side, meaning its value is ignored when Hibernate synchronizes with the database. If both sides are lists, we can only make persistent the index of one side.

A regular `@ManyToMany` mapping hides the link table; there is no corresponding Java class, only some collection properties. So whenever someone says, "My link table has more columns with information about the link"—and, in our experience, someone always says this sooner rather than later—we need to map this information to a Java class.

9.3.2 Many-to-many with an intermediate entity

We may always represent a *many-to-many* association as two *many-to-one* associations to an intervening class, and this is what we'll do next. We don't hide the link table but represent it with a Java class. This model is usually more easily extensible, so we tend not to use regular *many-to-many* associations in applications. It's a lot of work to change code later when inevitably more columns are added to a link table; so before mapping an `@ManyToMany` as shown in the previous section, consider the alternative shown in figure 9.15.



Figure 9.15 `CategorizedItem` is the link between `Category` and `Item`.

Imagine that we need to record some information each time we add an `Item` to a `Category`. The `CategorizedItem` captures the timestamp and the user who created the link. This domain model requires additional columns on the join table, as we can see in figure 9.16.

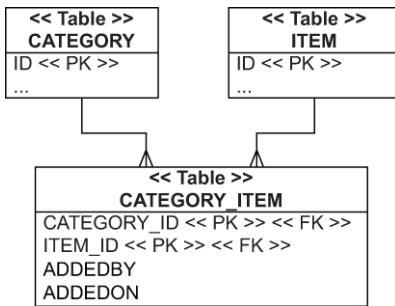


Figure 9.16 Additional columns on the join table in a many-to-many relationship

The new `CategorizedItem` entity maps to the link table, as demonstrated next. The source code to follow is to be found in the `manytomany-linkentity` folder.

This will be a large chunk of code with some new annotations. First, it will be an immutable entity class (it will be annotated with `@org.hibernate.annotations.Immutable`), so we'll never update properties after creation. Hibernate can do some optimizations, such as avoiding dirty checking during flushing of the persistence context if we declare the class immutable.

The entity class will have a composite key, which we'll encapsulate in a static nested embeddable component class for convenience. The identifier property and its composite key columns will be mapped to the entity's table through the `@EmbeddedId` annotation.

Listing 9.4 Mapping a *many-to-many* relationship with CategorizedItem

```

Path: manytomany-linkentity
      /src/main/java/com/manning/javapersistence/ch09/manytomany/linkentity/CategorizedIte
          m.java

@Entity
@Table(name = "CATEGORY_ITEM")
@org.hibernate.annotations.Immutable
public class CategorizedItem {
    @Embeddable
    public static class Id implements Serializable { #A
        @Column(name = "CATEGORY_ID")
        private Long categoryId;
        @Column(name = "ITEM_ID")
        private Long itemId;

        public Id() {
        }

        public Id(Long categoryId, Long itemId) { #B
            this.categoryId = categoryId;
            this.itemId = itemId;
        }
        //implementing equals and hashCode
    }
    @EmbeddedId #C
    private Id id = new Id();
    @Column(updatable = false)
    @NotNull
    private String addedBy; #D
    @Column(updatable = false)
    @NotNull
    @CreationTimestamp
    private LocalDateTime addedOn;
    @ManyToOne #E
    @JoinColumn(
        name = "CATEGORY_ID",
        insertable = false, updatable = false)
    private Category category; #F
    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        insertable = false, updatable = false)
    private Item item; #G
    public CategorizedItem( #H
        String addedByUsername,
        Category category,
        Item item) { #I
        this.addedBy = addedByUsername;
        this.category = category;
        this.item = item;
        this.id.categoryId = category.getId(); #J
        this.id.itemId = item.getId();
        category.addCategorizedItem(this); #I
        item.addCategorizedItem(this);
    }
    // ...
}

```

#A The class is immutable, as annotated with `@org.hibernate.annotations.Immutable`.
#B An entity class needs an identifier property. The primary key of the link table is the composite of `CATEGORY_ID` and `ITEM_ID`. We can externalize this `Id` class into its own file, of course.
#C The new `@EmbeddedId` annotation maps the identifier property and its composite key columns to the entity's table.
#D The basic property mapping the `addedBy` username to a column of the join table.
#E The basic property mapping the `addedOn` timestamp to a column of the join table. This is the “additional information about the link” that interests us.
#F The `@ManyToOne` property `category` is already mapped in the identifier.
#G The `@ManyToOne` property `item` is already mapped in the identifier. The trick here is to make them read-only, with the `updatable=false, insertable=false` setting. This means Hibernate or Spring Data JPA using Hibernate write the values of these columns by taking the identifier value of `CategorizedItem`. At the same time, we can read and browse the associated instances through `categorizedItem.getItem()` and `getCategory()`, respectively. (If we map the same column twice without making one mapping read-only, Hibernate or Spring Data JPA using Hibernate will complain on startup about a duplicate column mapping.)
#H We can also see that constructing a `CategorizedItem` involves setting the values of the identifier—the application always assigns composite key values; Hibernate doesn't generate them.
#I The constructor sets the `addedBy` field value and guarantees referential integrity by managing collections on both sides of the association.
#J The constructor sets the `categoryId` field value. We map these collections next to enable bidirectional navigation. This is a unidirectional mapping and enough to support the *many-to-many* relationship between `Category` and `Item`. To create a link, we instantiate and persist a `CategorizedItem`. If we want to break a link, we remove the `CategorizedItem`. The constructor of `CategorizedItem` requires that we provide already persistent `Category` and `Item` instances.

If bidirectional navigation is required, map an `@OneToMany` collection in `Category` and/or `Item`:

```
Path: manytomany-linkentity
      /src/main/java/com/manning/javapersistence/ch09/manytomany/linkentity/Category.java
@Entity
public class Category {
    ...
    @OneToMany(mappedBy = "category")
    private Set<CategorizedItem> categorizedItems = new HashSet<>();
    ...
}
Path: manytomany-linkentity
      /src/main/java/com/manning/javapersistence/ch09/manytomany/linkentity/Item.java
@Entity
public class Item {
    ...
    @OneToMany(mappedBy = "item")
    private Set<CategorizedItem> categorizedItems = new HashSet<>();
    ...
}
```

Both sides are “mapped by” the annotations in `CategorizedItem`, so Hibernate already knows what to do when we iterate through the collection returned by either `getCategorizedItems()` method.

This is how we create and store links:

```

Path: manytomany-linkentity
/src/test/java/com/manning/javapersistence/ch09/manytomany/linkentity/TestService.java
va
Category someCategory = new Category("Some Category");
Category otherCategory = new Category("Other Category");
categoryRepository.save(someCategory);
categoryRepository.save(otherCategory);
Item someItem = new Item("Some Item");
Item otherItem = new Item("Other Item");
itemRepository.save(someItem);
itemRepository.save(otherItem);
CategorizedItem linkOne = new CategorizedItem(
    "John Smith", someCategory, someItem
);
CategorizedItem linkTwo = new CategorizedItem(
    "John Smith", someCategory, otherItem
);
CategorizedItem linkThree = new CategorizedItem(
    "John Smith", otherCategory, someItem
);
categorizedItemRepository.save(linkOne);
categorizedItemRepository.save(linkTwo);
categorizedItemRepository.save(linkThree);

```

The primary advantage of this strategy is the possibility for bidirectional navigation: we can get all items in a category by calling `someCategory.getCategorizedItems()`, and then also navigate from the opposite direction with `someItem.getCategorizedItems()`. A disadvantage is a more complex code needed to manage the `CategorizedItem` entity instances to create and remove links, which we have to save and delete independently. We also need some infrastructure in the `CategorizedItem` class, such as the composite identifier. One small improvement would be to enable `CascadeType.PERSIST` on some of the associations, reducing the number of calls to `save()`.

In the previous example, we stored the user who created the link between `Category` and `Item` as a simple name string. If the join table instead had a foreign key column called `USER_ID`, we'd have a ternary relationship. The `CategorizedItem` would have a `@ManyToOne` for `Category`, `Item`, and `User`.

In the following section, we demonstrate another *many-to-many* strategy. To make it a bit more interesting, we make it a ternary association.

9.3.3 Ternary associations with components

In the previous section, we represented a *many-to-many* relationship with an entity class mapped to the link table. A potentially simpler alternative is mapping to an embeddable component class. The source code to follow is to be found in the `manytomany-ternary` folder.

```

Path: manytomany-ternary
/src/main/java/com/manning/javapersistence/ch09/manytomany/ternary/CategorizedItem.j
ava
@Embeddable
public class CategorizedItem {
    @ManyToOne
    @JoinColumn(
        name = "ITEM_ID",
        nullable = false, updatable = false
    )
    private Item item;
    @ManyToOne
    @JoinColumn(
        name = "USER_ID",
        updatable = false
    )
    @NotNull
    private User addedBy; #A
    @Column(updatable = false)
    @NotNull
    private LocalDateTime addedOn = LocalDateTime.now(); #A
    public CategorizedItem() {
    }
    public CategorizedItem(User addedBy,
                          Item item) {
        this.addedBy = addedBy;
        this.item = item;
    }
    // ...
}

```

#A The `@NotNull` annotations do not generate an SQL constraint, so the annotated fields will not be part of the primary key.

The new mappings here are `@ManyToOne` associations in an `@Embeddable`, and the additional foreign key join column `USER_ID`, making this a ternary relationship. Look at the database schema in figure 9.17.

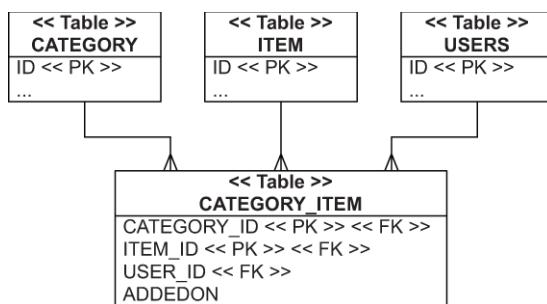


Figure 9.17 A link table with three foreign key columns

The owner of the embeddable component collection is the `Category` entity:

```
Path: manytomany-ternary
      /src/main/java/com/manning/javapersistence/ch09/manytomany/ternary/Category.java
@Entity
public class Category {
    ...
    @ElementCollection
    @CollectionTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID")
    )
    private Set<CategorizedItem> categorizedItems = new HashSet<>();
    ...
}
```

Unfortunately, this mapping isn't perfect: when we map an `@ElementCollection` of embeddable type, all properties of the target type that are `nullable=false` become part of the (composite) primary key. We want all columns in `CATEGORY_ITEM` to be `NOT NULL`. Only `CATEGORY_ID` and `ITEM_ID` columns should be part of the primary key, though. The trick is to use the Bean Validation `@NotNull` annotation on properties that shouldn't be part of the primary key. In that case (because it's an embeddable class), Hibernate ignores the Bean Validation annotation for primary key realization and SQL schema generation. The downside is that the generated schema won't have the appropriate `NOT NULL` constraints on the `USER_ID` and `ADDEDON` columns, which we should fix manually.

The advantage of this strategy is the implicit life cycle of the link components. To create an association between a `Category` and an `Item`, add a new `CategorizedItem` instance to the collection. To break the link, remove the element from the collection. No extra cascading settings are required, and the Java code is simplified (albeit spread over more lines):

```
Path: manytomany-ternary
      /src/test/java/com/manning/javapersistence/ch09/manytomany/ternary /TestService.java
Category someCategory = new Category("Some Category");
Category otherCategory = new Category("Other Category");
categoryRepository.save(someCategory);
categoryRepository.save(otherCategory);
Item someItem = new Item("Some Item");
Item otherItem = new Item("Other Item");
itemRepository.save(someItem);
itemRepository.save(otherItem);
User someUser = new User("John Smith");
userRepository.save(someUser);
CategorizedItem linkOne = new CategorizedItem(
    someUser, someItem
);
someCategory.addCategorizedItem(linkOne);
CategorizedItem linkTwo = new CategorizedItem(
    someUser, otherItem
);
someCategory.addCategorizedItem(linkTwo);
CategorizedItem linkThree = new CategorizedItem(
    someUser, someItem
);
otherCategory.addCategorizedItem(linkThree);
```

There is no way to enable bidirectional navigation: an embeddable component, such as `CategorizedItem` by definition, can't have shared references. We can't navigate from `Item` to `CategorizedItem`, and there is no mapping of this link in `Item`. Instead, we can write a query to retrieve the categories given an `Item`:

```
Path: manytomany-ternary
      /src/test/java/com/manning/javapersistence/ch09/manytomany/ternary /TestService.java
      List<Category> categoriesOfItem =
          categoryRepository.findCategoryWithCategorizedItems(item1);
      assertEquals(2, categoriesOfItem.size());
```

The `findCategoryWithCategorizedItems` method is annotated with the `@Query` annotation:

```
Path: manytomany-ternary
      /src/main/java/com/manning/javapersistence/ch09/repositories/manytomany/ternary
      /CategoryRepository.java
      @Query("select c from Category c join c.categorizedItems ci where
              ci.item = :itemParameter")
      List<Category> findCategoryWithCategorizedItems(
          @Param("itemParameter") Item itemParameter);
```

We've now completed our first ternary association mapping. In the previous chapters, we saw ORM examples with maps; the keys and values of the shown maps were always of basic or embeddable type. In the following section, we'll use more complex key/value pair types and their mappings.

9.4 Entity associations with Maps

Map keys and values can be references to other entities, providing another strategy for mapping *many-to-many* and ternary relationships. First, let's assume that only the value of each map entry is a reference to another entity.

9.4.1 One-to-many with a property key

If the value of each map entry is a reference to another entity, we have a *one-to-many* entity relationship. The key of the map is of a basic type: for example, a `Long` value. The source code to follow is to be found in the `maps-mapkey` folder.

An example of this structure would be the `Item` entity with a map of `Bid` instances, where each map entry is a pair of `Bid` identifier and reference to a `Bid` instance. When we iterate through `someItem.getBids()`, we iterate through map entries that look like `(1, <reference to Bid with PK 1>), (2, <reference to Bid with PK 2>)`, and so on:

```
Path: maps-mapkey /src/test/java/com/manning/javapersistence/ch09/maps/mapkey
      /TestService.java
      Item item = itemRepository.findById(someItem.getId()).get();
      assertEquals(2, item.getBids().size());
      for (Map.Entry<Long, Bid> entry : item.getBids().entrySet()) {
          assertEquals(entry.getKey(), entry.getValue().getId());
      }
```

The underlying tables for this mapping are nothing special; we have an `ITEM` and a `BID` table, with an `ITEM_ID` foreign key column in the `BID` table. This is the same schema as

demonstrated in figure 8.14 for a *one-to-many/many-to-one* mapping with a regular collection instead of a Map. Our motivation here is a slightly different representation of the data in the application.

In the `Item` class, we include a `Map` property named `bids`:

```
Path: maps-mapkey /src/main/java/com/manning/javapersistence/ch09/maps/mapkey/Item.java
@Entity
public class Item {
    /**
     * ...
     * @MapKey(name = "id")
     * @OneToMany(mappedBy = "item")
     * private Map<Long, Bid> bids = new HashMap<>();
     * ...
}
```

New here is the `@MapKey` annotation. It maps a property of the target entity, in this case, the `Bid` entity, as the key of the map. The default if we omit the `name` attribute is the identifier property of the target entity, so the `name` option here is redundant. Because the keys of a map form a set, we should expect values to be unique for a particular map. This is the case for `Bid` primary keys but likely not for any other property of `Bid`. It's up to us to ensure that the selected property has unique values—Hibernate or Spring Data JPA using Hibernate won't check.

The primary and rare use case for this mapping technique is the desire to iterate map entries with some property of the entry entity value as the entry key, maybe because it's convenient for how we'd like to render the data. A more common situation is a map in the middle of a ternary association.

9.4.2 Key/Value ternary relationship

We may be a little bored by now with all the mapping experiments that we executed, but we promise this is the last time we show another way to map the association between `Category` and `Item`. Previously, in section 9.3.3, we used an embeddable `CategorizedItem` component to represent the link. Here we show a representation of the relationship with a `Map` instead of an additional Java class. The key of each map entry is an `Item`, and the related value is the `User` who added the `Item` to the `Category`, as shown in figure 9.18.

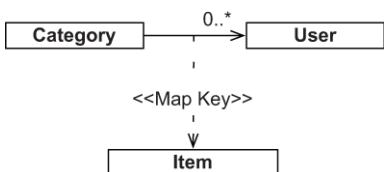


Figure 9.18 A Map with entity associations as key/value pairs

The link/join table in the schema, as we can see in figure 9.19, has three columns: CATEGORY_ID, ITEM_ID, and USER_ID. The Map is owned by the Category entity. The source code to follow is to be found in the maps-ternary folder.

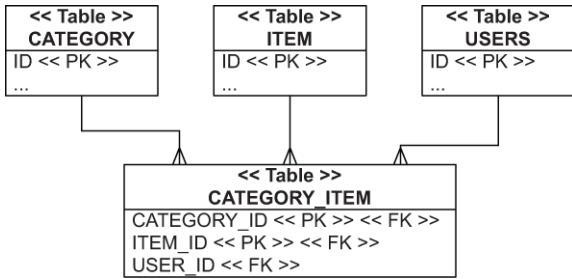


Figure 9.19 The link table represents the Map key/value pairs.

```

Path: maps-ternary
      /src/main/java/com/manning/javapersistence/ch09/maps/ternary/Category.java
@Entity
public class Category {
    /**
     * ...
     * @ManyToMany(cascade = CascadeType.PERSIST)
     * @MapKeyJoinColumn(name = "ITEM_ID") #A
     * @JoinTable(
     *     name = "CATEGORY_ITEM",
     *     joinColumns = @JoinColumn(name = "CATEGORY_ID"),
     *     inverseJoinColumns = @JoinColumn(name = "USER_ID")
     * )
     * private Map<Item, User> itemAddedBy = new HashMap<>();
     * ...
}

```

#A The `@MapKeyJoinColumn` is optional; Hibernate or Spring Data JPA using Hibernate would default to the column name `ITEMADDEDBY_KEY` for the join/foreign key column referencing the `ITEM` table.

To create a link between all three entities, all instances must already be in persistent state and then put into the map:

```

Path: maps-ternary/src/test/java/com/manning/javapersistence/ch09/maps/ternary
      /TestService.java
someCategory.putItemAddedBy(someItem, someUser);
someCategory.putItemAddedBy(otherItem, someUser);
otherCategory.putItemAddedBy(someItem, someUser);

```

To remove the link, remove the entry from the map. This is a convenient Java API for managing a complex relationship, hiding a database link table with three columns. But remember that in practice, link tables often grow additional columns, and changing all the Java application code later is expensive if we depend on a `Map` API. Earlier, we had an `ADDEDON` column with a timestamp when the link was created, but we had to drop it for this mapping.

9.5 Summary

- You learned how to map complex entity associations using *one-to-one* associations, *one-to-many* associations, *many-to-many* associations, ternary associations, and entity associations with maps.
- We demonstrated how to create *one-to-one* associations by sharing a primary key; using a foreign primary key generator; using a foreign key join column; using a join table.
- We analyzed the creation of *one-to-many* associations by considering one-to-many bags; using unidirectional and bidirectional list mappings; applying optional one-to-many with a join table; creating one-to-many associations in an embeddable class.
- We investigated the creation of unidirectional and bidirectional *many-to-many* associations and many-to-many associations with an intermediate entity.
- We built ternary associations with components and entity associations with maps.
- If you simplify the relationships between the classes you'll rarely need many of the techniques we've demonstrated. In particular, you can often best represent *many-to-many* entity associations as two *many-to-one* associations from an intermediate entity class or with a collection of components.
- Before attempting a complex collection mapping, always make sure you actually need a collection. Ask yourself whether you frequently iterate through its elements.
- The Java structures used in this chapter may make data access easier sometimes, but typically they complicate data storage, updates, and deletion.

10

Managing data

This chapter covers

- Examining the lifecycle and states of objects
- Working with the EntityManager interface
- Working with the Jakarta Persistence API
- Working with detached state

You now understand how ORM solves the static aspects of the object/relational mismatch. With what you know so far, you can create a mapping between Java classes and an SQL schema, solving the structural mismatch problem. We remind that the paradigm mismatch covers the problems of granularity, inheritance, identity, association, and data navigation. For a deeper review, see section 1.2.

An efficient application solution requires something more: you must investigate strategies for runtime data management. These strategies are crucial to the performance and correct behavior of the applications.

In this chapter, we analyze the life cycle of entity instances—how an instance becomes persistent, and how it stops being considered persistent—and the method calls and management operations that trigger these transitions. The JPA EntityManager is the primary interface for accessing data.

Before we look at the API, let's start with entity instances, their life cycle, and the events that trigger a change of state. Although some of the material may be formal, a solid understanding of the persistence life cycle is essential.

Major new features in JPA 2

- We can get a vendor-specific variation of the persistence manager API with `EntityManager#unwrap()`: for example, the `org.hibernate.Session` API. Use the already demonstrated `EntityManagerFactory#unwrap()` to obtain an instance of `org.hibernate.SessionFactory` (see section 2.5).
- The new `detach()` operation provides fine-grained management of the persistence context, evicting individual entity instances.
- From an existing `EntityManager`, we can obtain the `EntityManagerFactory` used to create the persistence context with `getEntityManagerFactory()`.
- The new static `PersistenceUtil` and `PersistenceUnitUtil` helper methods determine whether an entity instance (or one of its properties) was fully loaded or is an uninitialized reference (Hibernate proxy or unloaded collection wrapper.)

10.1 The persistence life cycle

Because JPA is a transparent persistence mechanism—classes are unaware of their own persistence capability—it is possible to write application logic that is unaware whether the data it operates on represents a persistent state or a temporary state that exists only in memory. The application shouldn't necessarily need to care that an instance is persistent when invoking its methods. We can, for example, invoke the `Item#calculateTotalPrice()` business method without having to consider persistence at all (for example, in a unit test). The method may be executing being unaware of any persistence concept.

Any application with a persistent state must interact with the persistence service whenever it needs to propagate the state held in memory to the database (or vice versa). In other words, we have to call the Jakarta Persistence interfaces to store and load data.

When interacting with the persistence mechanism that way, the application must concern itself with the state and life cycle of an entity instance with respect to persistence. We refer to this as the *persistence life cycle*: the states an entity instance goes through during its life, and we'll analyze them in a moment (figure 10.1). We also use the term *unit of work*: a set of (possibly) state-changing operations considered one (usually atomic) group. Another piece of the puzzle is the *persistence context* provided by the persistence service. Think of the persistence context as a service that remembers all the modifications and state changes we made to data in a particular unit of work (this is somewhat simplified, but it's a good starting point).

We now dissect all these terms: entity states, persistence contexts, and managed scope. You're probably more accustomed to thinking about what SQL statements you have to manage to get stuff in and out of the database, but one of the key factors of the success with Java Persistence is the analysis of *state management*, so stick with us through this section.

10.1.1 Entity instance states

Different ORM solutions use different terminology and define different states and state transitions for the persistence life cycle. Moreover, the states used internally may be different from those exposed to the client application. JPA defines four states, hiding the

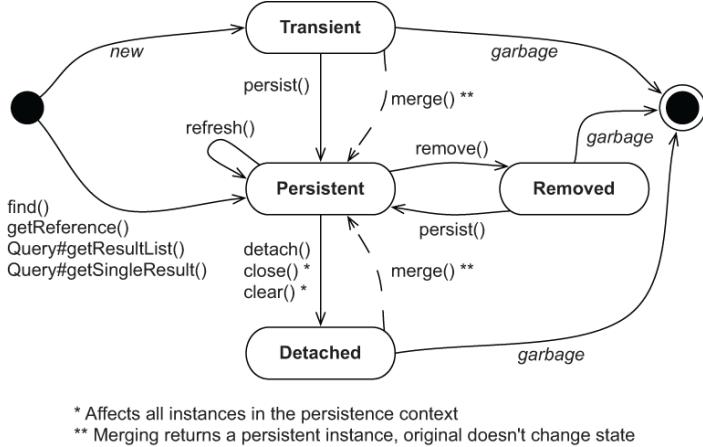


Figure 10.1 Entity instance states and their transitions

The statechart also includes the method calls to the `EntityManager` (and `Query`) API that trigger transitions. We analyze this chart in this chapter; refer to it whenever you need an overview. Let's explore the states and transitions in more detail.

TRANSIENT STATE

Instances created with the `new` Java operator are *transient*, which means their state is lost and garbage-collected as soon as they're no longer referenced. For example, `new Item()` creates a transient instance of the `Item` class, just like `new Long()` and `new BigDecimal()`. Hibernate doesn't provide any rollback functionality for transient instances; if we modify the price of a transient `Item`, we can't automatically undo the change.

For an entity instance to transition from transient to persistent state, to become managed, requires either a call to the `EntityManager#persist()` method or the creation of a reference from an already-persistent instance and enabled cascading of state for that mapped association.

PERSISTENT STATE

A *persistent* entity instance has a representation in the database. It's stored in the database—or it will be stored when the unit of work completes. It's an instance with a database identity, as defined in section 5.2; its database identifier is set to the primary key value of the database representation.

The application may have created instances and then made them persistent by calling `EntityManager#persist()`. There may be instances that became persistent when the application created a reference to the object from another persistent instance that the JPA provider already manages. A persistent entity instance may be an instance retrieved from

the database by execution of a query, by an identifier lookup, or by navigating the object graph starting from another persistent instance.

Persistent instances are always associated with a persistence context. We'll see more about this in a moment.

REMOVED STATE

We can delete a persistent entity instance from the database in several ways. For example, we can remove it with `EntityManager#remove()`. It may also become available for deletion if we remove a reference to it from a mapped collection with *orphan removal* enabled.

An entity instance is then in the *removed state*: the provider will delete it at the end of a unit of work. We should discard any references we may hold to it in the application after we finish working with it—for example, after we've rendered the removal-confirmation screen the users see.

DETACHED STATE

To understand *detached* entity instances, consider loading an instance. We call `EntityManager#find()` to retrieve an entity instance by its (known) identifier. Then we end our unit of work and close the persistence context. The application still has a *handle*—a reference to the instance we loaded. It's now in a detached state, and the data is becoming stale. We could discard the reference and let the garbage collector reclaim the memory. Or, we could continue working with the data in the detached state and later call the `merge()` method to save our modifications in a new unit of work. We'll discuss detachment and merging again later in this chapter, in a dedicated section.

You should now have a basic understanding of entity instance states and their transitions. Our next topic is the persistence context: an essential service of any Jakarta Persistence provider.

10.1.2 The persistence context

In a Java Persistence application, an `EntityManager` has a persistence context. We create a persistence context when we call `EntityManagerFactory#createEntityManager()`. The context is closed when we call `EntityManager#close()`. In JPA terminology, this is an *application-managed* persistence context; our application defines the scope of the persistence context, demarcating the unit of work.

The persistence context monitors and manages all entities in the persistent state. The persistence context is the centerpiece of much of the functionality of a JPA provider.

The persistence context allows the persistence engine to perform *automatic dirty checking*, detecting which entity instances the application modified. The provider then synchronizes with the database the state of instances monitored by a persistence context, either automatically or on-demand. Typically, when a unit of work completes, the provider propagates state held in memory to the database through the execution of SQL `INSERT`, `UPDATE`, and `DELETE` statements (all part of the Data Modification Language [DML]). This *flushing* procedure may also occur at other times. For example, Hibernate may synchronize with the database before the execution of a query. This ensures that queries are aware of changes made earlier during the unit of work.

The persistence context acts as a *first-level cache*; it remembers all entity instances we've handled in a particular unit of work. For example, if we ask Hibernate to load an entity instance using a primary key value (a lookup by identifier), Hibernate can first check the current unit of work in the persistence context. If Hibernate finds the entity instance in the persistence context, no database hit occurs—this is a repeatable read for an application. Consecutive `em.find(Item.class, ITEM_ID)` calls with the same persistence context will yield the same result.

This cache also affects results of arbitrary queries, executed for example with the `javax.persistence.Query` API. Hibernate reads the SQL result set of a query and transforms it into entity instances. This process first tries to resolve every entity instance in the persistence context by identifier lookup. Only if an instance with the same identifier value can't be found in the current persistence context does Hibernate read the rest of the data from the result-set row. Hibernate ignores any potentially newer data in the result set, due to read-committed transaction isolation at the database level, if the entity instance is already present in the persistence context.

The persistence context cache is always on—it can't be turned off. It ensures the following:

- The persistence layer isn't vulnerable to stack overflows in the case of circular references in an object graph.
- There can never be conflicting representations of the same database row at the end of a unit of work. The provider can safely write all changes made to an entity instance to the database.
- Likewise, changes made in a particular persistence context are always immediately visible to all other code executed inside that unit of work and its persistence context. JPA guarantees repeatable entity-instance reads.

The persistence context provides a *guaranteed scope of object identity*; in the scope of a single persistence context, only one instance represents a particular database row. Consider the comparison of references `entityA == entityB`. This is true only if both are references to the same Java instance on the heap. Now, consider the comparison `entityA.getId().equals(entityB.getId())`. This is true if both have the same database identifier value. Within one persistence context, Hibernate guarantees that both comparisons will yield the same result. This solves one of the fundamental O/R mismatch problems we introduced in section 1.2.3.

Would process-scoped identity be better?

For a typical web or enterprise application, persistence context-scoped identity is preferred. Process-scoped identity, where only one in-memory instance represents the row in the entire process (JVM), would offer some potential advantages in terms of cache utilization. In a pervasively multithreaded application, though, the cost of always synchronizing shared access to persistent instances in a global identity map is too high a price to pay. It's simpler and more scalable to have each thread work with a distinct copy of the data in each persistence context.

The life cycle of entity instances and the services provided by the persistence context can be difficult to understand at first. Let's look at some code examples of dirty checking, caching, and how the guaranteed identity scope works in practice. To do this, we work with the persistence manager API.

10.2 The EntityManager interface

To be able to execute the examples from the source code, you need first to run the Ch10.sql script. The source code to follow is to be found in the `managing-data` and `managing-data2` folders. We do not use Spring Data JPA in this chapter. The examples to follow will use JPA and, sometimes, the Hibernate API – they are finer-grained for our demonstrations and analysis.

Any transparent persistence tool includes a persistence manager API. This persistence manager usually provides services for basic CRUD (create, read, update, delete) operations, query execution, and controlling the persistence context. In Jakarta Persistence applications, the main interface we interact with is the `EntityManager`, to create units of work.

10.2.1 The canonical unit of work

In Java SE and some EE architectures (if we only have plain servlets, for example), we get an `EntityManager` by calling `EntityManagerFactory#createEntityManager()`. The application code shares the `EntityManagerFactory`, representing one persistence unit, or one logical database. Most applications have only one shared `EntityManagerFactory`.

We use the `EntityManager` for a single unit of work in a single thread, and it's inexpensive to create. The following listing shows the canonical, typical form of a unit of work.

Listing 10.1 A typical unit of work

```

Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      basicUOW()
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("ch10");

//...
EntityManager em = emf.createEntityManager();

try {
    em.getTransaction().begin();

    //...
    em.getTransaction().commit();
} catch (Exception ex) {
    // Transaction rollback, exception handling
    //...
} finally {
    if (em != null && em.isOpen())
        em.close();
}

```

Everything between `em.getTransaction().begin()` and `em.getTransaction().commit()` occurs in one transaction. For now, keep in mind that all database operations in a transaction scope, such as the SQL statements executed by Hibernate, completely either succeed or fail. Don't worry too much about the transaction code for now; you'll read more about concurrency control in the next chapter. We'll look at the same example again with a focus on the transaction and exception-handling code. Don't write empty `catch` clauses in the code, though—you'll have to roll back the transaction and handle exceptions.

Creating an `EntityManager` starts its persistence context. Hibernate won't access the database until necessary; the `EntityManager` doesn't obtain a JDBC Connection from the pool until SQL statements have to be executed. We can create and close an `EntityManager` without hitting the database. Hibernate executes SQL statements when we look up or query data and when it flushes changes detected by the persistence context to the database. Hibernate joins the in-progress system transaction when an `EntityManager` is created and waits for the transaction to commit. When Hibernate is notified of commit, it performs dirty checking of the persistence context and synchronizes with the database. We can also force dirty checking synchronization manually by calling `EntityManager#flush()` at any time during a transaction.

We decide the scope of the persistence context by choosing when to `close()` the `EntityManager`. We have to close the persistence context at some point, so always place the `close()` call in a `finally` block.

How long should the persistence context be open? Let's assume for the following examples that we're writing a server, and each client request will be processed with one persistence context and system transaction in a multithreaded environment. If you're familiar with servlets, imagine the code in listing 10.1 embedded in a servlet's `service()` method. Within this unit of work, you access the `EntityManager` to load and store data.

10.2.2 Making data persistent

Let's create a new instance of an entity and bring it from transient into persistent state. You will do such a thing whenever you would like to save the information from a newly created object to the database. We can see the same unit of work and how the `Item` instances change state in figure 10.2.

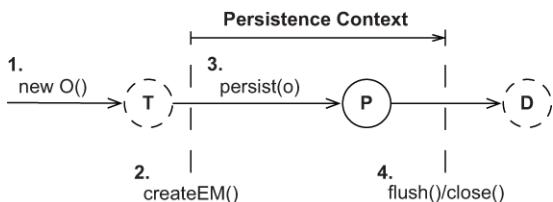


Figure 10.2 Making an instance persistent in a unit of work

```

Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      makePersistent()
Item item = new Item();
item.setName("Some Item");
em.persist(item);
Long ITEM_ID = item.getId();

```

A new transient `Item` is instantiated as usual. Of course, we may also instantiate it before creating the `EntityManager`. A call to `persist()` makes the transient instance of `Item` persistent. It's now managed by and associated with the current persistence context.

To store the `Item` instance in the database, Hibernate has to execute an SQL `INSERT` statement. When the transaction of this unit of work commits, Hibernate flushes the persistence context, and the `INSERT` occurs at that time. Hibernate may even batch the `INSERT` at the JDBC level with other statements. When we call `persist()`, only the identifier value of the `Item` is assigned. Alternatively, if the identifier generator isn't *pre-insert*, the `INSERT` statement will be executed immediately when `persist()` is called. You may want to review section 5.2.5.

Detecting entity state using the identifier

Sometimes we need to know whether an entity instance is transient, persistent, or detached. An entity instance is in persistent state if `EntityManager#contains(e)` returns `true`. It's in transient state if `PersistenceUnitUtil#getIdentifier(e)` returns `null`. It's in the detached state if it's not persistent, and `PersistenceUnitUtil#getIdentifier(e)` returns the value of the entity's identifier property. We can get to the `PersistenceUnitUtil` from the `EntityManagerFactory`.

There are two issues to look out for. First, be aware that the identifier value may not be assigned and available until the persistence context is flushed. Second, Hibernate (unlike some other JPA providers) never returns `null` from `PersistenceUnitUtil#getIdentifier()` if the identifier property is a primitive (a `long` and not a `Long`).

It's better (but not required) to fully initialize the `Item` instance before managing it with a persistence context. The SQL `INSERT` statement contains the values that were held by the instance at the point when `persist()` was called. If we don't set the `name` of the `Item` before making it persistent, a `NOT NULL` constraint may be violated. We can modify the `Item` after calling `persist()`, and the changes will be propagated to the database with an additional SQL `UPDATE` statement.

If one of the `INSERT` or `UPDATE` statements made when flushing fails, Hibernate causes a rollback of changes made to persistent instances in this transaction at the database level. But Hibernate doesn't roll back in-memory changes to persistent instances. If we change the `Item#name` after `persist()`, a commit failure won't roll back to the old name. This is reasonable because a failure of a transaction is normally non-recoverable, and we have to discard the failed persistence context and `EntityManager` immediately. We'll discuss exception handling in the next chapter.

Next, we load and modify the stored data.

10.2.3 Retrieving and modifying persistent data

We can retrieve persistent instances from the database with the `EntityManager`. In a real-life use case, we are in the situation when we've kept the identifier value of the `Item` stored in the previous section somewhere and are now looking up the same instance in a new unit of work by identifier.

Figure 10.3 shows this transition graphically.

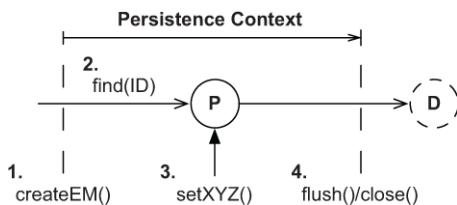


Figure 10.3 Making an instance persistent in a unit of work

```

Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      retrievePersistent()
Item item = em.find(Item.class, ITEM_ID);                                #A
if (item != null)
    item.setName("New Name");                                              #B

```

#A The instruction will hit the database if `item` is not already in the persistence context.
#B Then, we modify the name.

We don't need to cast the returned value of the `find()` operation; it's a generified method, and its return type is set as a side effect of the first parameter. The retrieved entity instance is in a persistent state, and we can now modify it inside the unit of work.

If no persistent instance with the given identifier value can be found, `find()` returns `null`. The `find()` operation always hits the database if there was no hit for the given entity type and identifier in the persistence context cache. The entity instance is always initialized during loading. We can expect to have all of its values available later in a detached state: for example, when rendering a screen after we close the persistence context. (Hibernate may not hit the database if its optional second-level cache is enabled).

We can modify the `Item` instance, and the persistence context will detect these changes and record them in the database automatically. When Hibernate flushes the persistence context during commit, it executes the necessary SQL DML statements to synchronize the changes with the database. Hibernate propagates state changes to the database as late as possible, toward the end of the transaction. DML statements usually create locks in the database that are held until the transaction completes, so Hibernate keeps the lock duration in the database as short as possible.

Hibernate writes the new `Item.name` to the database with an SQL `UPDATE`. By default, Hibernate includes all columns of the mapped `ITEM` table in the SQL `UPDATE` statement, updating unchanged columns to their old values. Hence, Hibernate can generate these basic SQL statements at startup, not at runtime. If we want to include only modified (or non-nullable for `INSERT`) columns in SQL statements, we can enable dynamic SQL generation as demonstrated in section 5.3.2.

Hibernate detects the changed `name` by comparing the `Item` with a snapshot copy it took before when the `Item` was loaded from the database. If the `Item` is different from the snapshot, an `UPDATE` is necessary. This snapshot in the persistence context consumes memory. Dirty checking with snapshots can also be time-consuming because Hibernate has to compare all instances in the persistence context with their snapshot during flushing.

We mentioned earlier that the persistence context enables repeatable reads of entity instances and provides an object-identity guarantee:

```

Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      retrievePersistent()
Item itemA = em.find(Item.class, ITEM_ID);                                #A
Item itemB = em.find(Item.class, ITEM_ID);                                              #B
assertTrue(itemA == itemB);
assertTrue(itemA.equals(itemB));
assertTrue(itemA.getId().equals(itemB.getId()));

```

```
#A The first find() operation hits the database and retrieves the Item instance with a SELECT statement.
#B The second find() is a repeatable read and is resolved in the persistence context, and the same
     cached Item instance is returned.
```

Sometimes we need an entity instance but we don't want to hit the database.

10.2.4 Getting a reference

If we don't want to hit the database when loading an entity instance, because we aren't sure we need a fully initialized instance, we can tell the EntityManager to attempt the retrieval of a hollow placeholder—a proxy.

If the persistence context already contains an Item with the given identifier, that Item instance is returned by `getReference()` without hitting the database. Furthermore, if no persistent instance with that identifier is currently managed, Hibernate produces the hollow placeholder: the proxy. This means `getReference()` won't access the database, and it doesn't return `null`, unlike `find()`.

JPA offers the `PersistenceUnitUtil` helper methods. A helper method is `isLoaded()` to detect whether we're working with an uninitialized proxy.

As soon as we call any method such as `Item#getName()` on the proxy, a SELECT is executed to fully initialize the placeholder. The exception to this rule is a mapped -database identifier getter method, such as `getId()`. A proxy may look like the real thing, but it's only a placeholder carrying the identifier value of the entity instance it represents. If the database record no longer exists when the proxy is initialized, an `EntityNotFoundException` is thrown. Note that the exception can be thrown when `Item#getName()` is called.

Hibernate has a convenient static `initialize()` method that loads the proxy's data.

After the persistence context is closed, item is in a detached state.

If we don't initialize the proxy while the persistence context is still open, we get a `LazyInitializationException` if we access the proxy. We can't load data on demand once the persistence context is closed. The solution is simple: load the data before closing the persistence context.

```
Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      retrievePersistentReference()
Item item = em.getReference(Item.class, ITEM_ID);                      #A
PersistenceUnitUtil persistenceUtil =                                     #B
    emf.getPersistenceUnitUtil();
assertFalse(persistenceUtil.isLoaded(item));                           #C
// assertEquals("Some Item", item.getName());                         #D
// Hibernate.initialize(item);                                         #E
em.getTransaction().commit();
em.close();                                                               #F
assertThrows(LazyInitializationException.class, () -> item.getName()); #G
```

#A The persistence context.

#B The helper methods.

#C Detecting an uninitialized proxy.

#D Mapping the exception to the rule.

#E Load the proxy data.

#F item is in a detached state.

#G Load data before closing the persistence context.

We'll have much more to say about proxies, lazy loading, and on-demand fetching in chapter 12. If we want to remove the state of an entity instance from the database, we have to make it transient.

10.2.5 Making data transient

Figure 10.4 shows the same process.

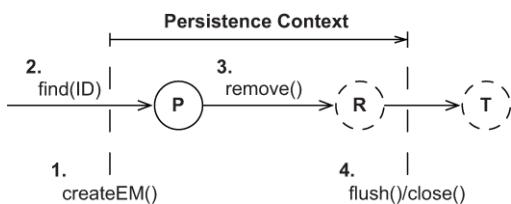


Figure 10.4 Removing an instance in a unit of work

If we call `find()`, Hibernate executes a `SELECT` to load the `Item`. If we call `getReference()`, Hibernate attempts to avoid the `SELECT` and returns a proxy.

Calling `remove()` queues the entity instance for deletion when the unit of work completes; it's now in `removed` state. If `remove()` is called on a proxy, Hibernate executes a `SELECT` to load the data. An entity instance must be fully initialized during life cycle transitions. We may have life cycle callback methods or an entity listener enabled (see section 13.2), and the instance must pass through these interceptors to complete its full life cycle.

An entity in a removed state is no longer in a persistent state. We can check this with the `contains()` operation. We can make the removed instance persistent again, canceling deletion. Now, `item` will look like a transient instance, if the `hibernate.use_identifier_rollback` property is enabled in `persistence.xml`.

When the transaction commits, Hibernate synchronizes the state transitions with the database and executes the SQL `DELETE`. The JVM garbage collector detects that the `item` is no longer referenced by anyone and finally deletes the last trace of the data. We can finally close `EntityManager`.

```

Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      makeTransient()
Item item = em.find(Item.class, ITEM_ID);                      #A
//Item item = em.getReference(Item.class, ITEM_ID);
em.remove(item);                                              #B
assertFalse(em.contains(item));                                #C
// em.persist(item);
assertNull(item.getId());                                     #D
em.getTransaction().commit();                                  #E
em.close();                                                    #F
#G

#A Call find(), Hibernate executes a SELECT to load the Item.
#B Call remove(), Hibernate queues the entity instance for deletion when the unit of work completes.
#C An entity in a removed state is no longer contained in the persistence context.
#D Canceling deletion makes the removed instance persistent again..
#E item will now look like a transient instance.
#F The transaction commits, Hibernate synchronizes the state transitions with the database and executes the SQL
     DELETE.
#G Close EntityManager.

```

By default, Hibernate won't alter the identifier value of a removed entity instance. This means the `item.getId()` method still returns the now outdated identifier value. Sometimes it's useful to work with the "deleted" data further: for example, we might want to save the removed `Item` again if our user decides to undo. As shown in the example, we can call `persist()` on a removed instance to cancel the deletion before the persistence context is flushed. Alternatively, if we set the property `hibernate.use_identifier_rollback` to `true` in `persistence.xml`, Hibernate will reset the identifier value after the removal of an entity instance. In the previous code example, the identifier value is reset to the default value of `null` (it's a `Long`). The `Item` is now the same as in a transient state, and we can save it again in a new persistence context.

Let's say we load an entity instance from the database and work with the data. For some reason, we know that another application or maybe another thread of the application has updated the underlying row in the database. Next, we'll see how to refresh the data held in memory.

10.2.6 Refreshing data

It is possible that, after you loaded the entity instance, some other process changes the information corresponding to the instance in the database. The following example demonstrates refreshing a persistent entity instance:

```

Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      refresh()
Item item = em.find(Item.class, ITEM_ID);
item.setName("Some Name");
// Someone updates this row in the database with "Concurrent UpdateName"
em.refresh(item);
em.close();
assertEquals("Concurrent UpdateName", item.getName());

```

After we load the entity instance, we realize (it isn't important how) that someone else changed the data in the database. Calling `refresh()` causes Hibernate to execute a `SELECT` to read and marshal a whole result set, overwriting changes we already made to the persistent instance in application memory. As a result, the `item's name` was updated with the value set from the other side. If the database row no longer exists (someone deleted it), Hibernate throws an `EntityNotFoundException` on `refresh()`.

Most applications don't have to manually refresh the in-memory state; concurrent modifications are typically resolved at transaction commit time. The best use case for refreshing is with an extended persistence context, which might span several request/response cycles and/or system transactions. While we wait for user input with an open persistence context, data gets stale, and selective refreshing may be required depending on the duration of the conversation and the dialogue between the user and the system. Refreshing can be useful to undo changes made in memory during a conversation if the user cancels the dialogue.

Another infrequently used operation is the replication of an entity instance.

10.2.7 Replicating data

Replication is useful, for example, when we need to retrieve data from one database and store it in another. Replication takes detached instances loaded in one persistence context and makes them persistent in another persistence context. We usually open these contexts from two different `EntityManagerFactory` configurations, enabling two logical databases. We have to map the entity in both configurations.

The `replicate()` operation is only available on the Hibernate Session API. Here is an example that loads an `Item` instance from one database and copies it into another:

```
Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      replicate()
EntityManager emA = getDatabaseA().createEntityManager();
emA.getTransaction().begin();
Item item = emA.find(Item.class, ITEM_ID);
emA.getTransaction().commit();

EntityManager emB = getDatabaseB().createEntityManager();
emB.getTransaction().begin();
emB.unwrap(Session.class)
    .replicate(item, org.hibernate.ReplicationMode.LATEST_VERSION);
Item item1 = emB.find(Item.class, ITEM_ID);
assertEquals("Some Item", item1.getName());
emB.getTransaction().commit();
emA.close();
emB.close();
```

`ReplicationMode` controls the details of the replication procedure:

- `IGNORE`—Ignores the instance when there is an existing database row with the same identifier in the database.
- `OVERWRITE`—Overwrites any existing database row with the same identifier in the database.

- `EXCEPTION`—Throws an exception if there is an existing database row with the same identifier in the target database.
- `LATEST_VERSION`—Overwrites the row in the database if its version is older than the version of the given entity instance, or ignores the instance otherwise. Requires enabled optimistic concurrency control with entity versioning (see section 11.2.2).

We may need replication when we reconcile data entered into different databases. One use case is a product upgrade: if the new version of the application requires a new database (schema), we may want to migrate and replicate the existing data once.

The persistence context does many things for you: automatic dirty checking, guaranteed scope of object identity, and so on. It's equally important that you know some of the details of its management, and that you sometimes influence what goes on behind the scenes.

10.2.8 Caching in the persistence context

The persistence context is a cache of persistent instances. Every entity instance in a persistent state is associated with the persistence context.

Many Hibernate users who ignore this simple fact run into an `OutOfMemoryException`. This is typically the case when we load thousands of entity instances in a unit of work but never intend to modify them. Hibernate still has to create a snapshot of each instance in the persistence context cache, which can lead to memory exhaustion. (Obviously, we should execute a bulk data operation if we modify thousands of rows).

The persistence context cache never shrinks automatically. Keep the size of the persistence context to the necessary minimum. Often, many persistent instances in the context are there by accident—for example, because we needed only a few items but queried for many. Extremely large graphs can have a serious performance impact and require significant memory for state snapshots. Check that the queries return only data you need, and consider the following ways to control Hibernate's caching behavior.

We can call `EntityManager#detach(i)` to evict a persistent instance manually from the persistence context. We can call `EntityManager#clear()` to detach all persistent entity instances, leaving us with an empty persistence context.

The native `Session` API has some extra operations we might find useful. We can set the entire persistence context to read-only mode. This disables state snapshots and dirty checking, and Hibernate won't write modifications to the database:

```
Path: managing-data2/src/test/java/com/manning/javapersistence/ch10/ReadOnly.java -
      selectiveReadOnly()
em.unwrap(Session.class).setDefaultReadonly(true);                                #A
Item item = em.find(Item.class, ITEM_ID);
item.setName("New Name");
em.flush();                                                               #B
```

#A We set the persistence context to read-only.

#B Consequently, `flush()` will not update the database.

We can disable dirty checking for a single entity instance:

```
Path: managing-data2/src/test/java/com/manning/javapersistence/ch10/ReadOnly.java -
      selectiveReadOnly()
Item item = em.find(Item.class, ITEM_ID);
em.unwrap(Session.class).setReadOnly(item, true);                                #A
item.setName("New Name");
em.flush();                                                               #B
```

#A We set `item` in the persistence context to read-only.
#B Consequently, `flush()` will not update the database.

A query with the `org.hibernate.Query` interface can return read-only results, which Hibernate doesn't check for modifications:

```
Path: managing-data2/src/test/java/com/manning/javapersistence/ch10/ReadOnly.java -
      selectiveReadOnly()
org.hibernate.query.Query query = em.unwrap(Session.class)
    .createQuery("select i from Item i");
query.setReadOnly(true).list();                                                 #A
List<Item> result = query.list();
for (Item item : result)
    item.setName("New Name");
em.flush();                                                               #B
```

#A We set the query to read-only.
#B Consequently, `flush()` will not update the database.

Thanks to query hints, we can also disable dirty checking for instances obtained with the JPA standard `javax.persistence.Query` interface:

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

Be careful with read-only entity instances: you can still delete them, and modifications to collections are tricky! The Hibernate manual has a long list of special cases you need to read if you use these settings with mapped collections.

So far, flushing and synchronization of the persistence context have occurred automatically, when the transaction commits. In some cases, we need more control over the synchronization process.

10.2.9 Flushing the persistence context

By default, Hibernate flushes the persistence context of an `EntityManager` and synchronizes changes with the database whenever the joined transaction is committed. All the previous code examples, except some in the last section, have used that strategy. JPA allows implementations to synchronize the persistence context at other times if they wish.

Hibernate, as a JPA implementation, synchronizes at the following times:

- When a joined JTA (Java Transaction API) system transaction is committed
- Before a query is executed—we don't mean lookup with `find()` but a query with `javax.persistence.Query` or the similar Hibernate API
- When the application calls `flush()` explicitly

We can control this behavior with the `FlushModeType` setting of an `EntityManager`:

```
Path: managing-
       data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
       flushModeType()

em.getTransaction().begin();
Item item = em.find(Item.class, ITEM_ID);
item.setName("New Name");

em.setFlushMode(FlushModeType.COMMIT);

assertEquals(
    "Original Name",
    em.createQuery("select i.name from Item i where i.id = :id")
        .setParameter("id", ITEM_ID).getSingleResult()
);

em.getTransaction().commit(); // Flush!
em.close();
```

Here, we load an `Item` instance and change its name. Then we query the database, retrieving the item's name.

Usually, Hibernate recognizes that data has changed in memory and synchronizes these modifications with the database before the query. This is the behavior of `FlushModeType.AUTO`, the default if we join the `EntityManager` with a transaction. With `FlushModeType.COMMIT` #C, we're disabling flushing before queries, so we may see different data returned by the query than what we have in memory. The synchronization then occurs only when the transaction commits.

We can at any time, while a transaction is in progress, force dirty checking and synchronization with the database by calling `EntityManager#flush()`.

This concludes our discussion of the *transient*, *persistent*, and *removed* entity states, and the basic usage of the `EntityManager` API. Mastering these state transitions and API methods is essential; every JPA application is built with these operations.

Next, we look at the *detached* entity state. We already mentioned some issues we'll see when entity instances aren't associated with a persistence context anymore, such as disabled lazy initialization. Let's explore the detached state with some examples, so we know what to expect when we work with data outside of a persistence context.

10.3 Working with detached state

If a reference leaves the scope of guaranteed identity, we call it a *reference* to a *detached entity instance*. When the persistence context is closed, it no longer provides an identity-mapping service. You'll run into aliasing problems when you work with detached entity instances, so make sure you understand how to handle the identity of detached instances.

10.3.1 The identity of detached instances

If we look up data using the same database identifier value in the same persistence context, the result is two references to the same in-memory instance on the JVM heap. Consider the two units of work shown next.

When different references are obtained from the same persistence context, they have the same Java identity. The references may be equal because by default `equals()` relies on Java identity comparison. They obviously have the same database identity. They reference the same instance, in persistent state, managed by the persistence context for that unit of work.

References are in a detached state when the first persistence context is closed. We may be dealing with instances that live outside of a guaranteed scope of object identity.

Listing 10.2 Guaranteed scope of object identity in Java Persistence

```
Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      scopeOfIdentity()
em = emf.createEntityManager();                                     #A
em.getTransaction().begin();                                       #B
Item a = em.find(Item.class, ITEM_ID);                            #C
Item b = em.find(Item.class, ITEM_ID);                            #D
assertTrue(a == b);                                              #E
assertEquals(a.equals(b));                                         #F
em.getTransaction().commit();                                      #G
em.close();                                                       #H
em = emf.createEntityManager();
em.getTransaction().begin();
Item c = em.find(Item.class, ITEM_ID);
assertTrue(a != c);                                              #I
assertFalse(a.equals(c));                                         #J
assertEquals(a.getId(), c.getId());                                #K
em.getTransaction().commit();
em.close();
```

#A Create a persistence context.

#B Begin the transaction.

#C Load some entity instances.

#D References `a` and `b` are obtained from the same persistence context, they have the same Java identity.

#E `equals()` relies on Java identity comparison.

#F They reference the same `Item` instance, in persistent state, managed by the persistence context for that unit of work.

#G Commit the transaction.

#H Close the persistence context. References `a` and `b` are in a detached state when the first persistence context is closed.

#I `a` and `c`, loaded in a different persistence context, aren't identical.

#J `a.equals(c)` is also false.

#K A test for database identity still returns `true`.

This behavior can lead to problems if we treat entity instances as equal in detached state. For example, consider the following extension of the code, after the second unit of work has ended:

```

em.close();
Set<Item> allItems = new HashSet<>();
allItems.add(a);
allItems.add(b);
allItems.add(c);
assertEquals(2, allItems.size());

```

This example adds all three references to a `Set`. All are references to detached instances. Now, if we check the size of the collection—the number of elements—what result do we expect?

A `Set` doesn't allow duplicate elements. Duplicates are detected by the `Set`; whenever we add a reference, the `Item#equals()` method is called automatically against all other elements already in the collection. If `equals()` returns `true` for any element already in the collection, the addition doesn't occur.

By default, all Java classes inherit the `equals()` method of `java.lang.Object`. This implementation uses a double-equals (`==`) comparison to check whether two references refer to the same in-memory instance on the Java heap.

You may guess that the number of elements in the collection is 2. After all, `a` and `b` are references to the same in-memory instance; they have been loaded in the same persistence context. We obtained reference `c` from another persistence context; it refers to a different instance on the heap. We have three references to two instances, but we know this only because we've seen the code that loaded the data. In a real application, we may not know that `a` and `b` are loaded in a different context than `c`. Furthermore, we obviously expect that the collection has exactly one element because `a`, `b`, and `c` represent the same database row, the same `Item`.

Whenever we work with instances in a detached state and we test them for equality (usually in hash-based collections), we need to supply our own implementation of the `equals()` and `hashCode()` methods for our mapped entity class. This is an important issue: if we don't work with entity instances in a detached state, no action is needed, and the default `equals()` implementation of `java.lang.Object` is fine. We rely on Hibernate's guaranteed scope of object identity within a persistence context. Even if we work with detached instances: if we never check if they're equal, we never put them in a `Set` or use them as keys in a `Map`, we don't have to worry. If all we do is render a detached `Item` on the screen, we aren't comparing it to anything.

Let's assume that we want to use detached instances and that we have to test them for equality with our own method.

10.3.2 Implementing equality methods

We can implement `equals()` and `hashCode()` methods several ways. Keep in mind that when we override `equals()`, we always need to also override `hashCode()` so the two methods are consistent. If two instances are equal, they must have the same hash value.

A seemingly clever approach is to implement `equals()` to compare just the database identifier property, which is often a surrogate primary key value. Basically, if two `Item` instances have the same identifier returned by `getId()`, they must be the same. If `getId()` returns `null`, it must be a transient `Item` that hasn't been saved.

Unfortunately, this solution has one huge problem: identifier values aren't assigned by Hibernate until an instance becomes persistent. If a transient instance were added to a `Set` before being saved, then when we save it, its hash value would change while it's contained by the `Set`. This is contrary to the contract of `java.util.Set`, breaking the collection. In particular, this problem makes cascading persistent states useless for mapped associations based on sets. We strongly discourage database identifier equality.

To get to the solution that we recommend, you need to understand the notion of a *business key*. A business key is a property or some combination of properties, that is unique for each instance with the same database identity. Essentially, it's the natural key that we would use if we weren't using a surrogate primary key instead. Unlike a natural primary key, it isn't an absolute requirement that the business key never changes—as long as it changes rarely, that's enough.

We argue that essentially every entity class should have a business key, even if it includes all properties of the class (which would be appropriate for some immutable classes). If our users are looking at a list of items on the screen, how do they differentiate between items A, B, and C? The same property, or combination of properties, is our business key. The business key is what the user thinks of as uniquely identifying a particular record, whereas the surrogate key is what the application and database systems rely on. The business key property or properties are most likely constrained `UNIQUE` in our database schema.

Let's write custom equality methods for the `User` entity class; this is easier than comparing `Item` instances. For the `User` class, `username` is a great candidate business key. It's always required, it's unique with a database constraint, and it changes rarely, if ever.

Listing 10.3 Custom implementation of User equality

```
@Entity
@Table(name = "USERS",
        uniqueConstraints =
            @UniqueConstraint(columnNames = "USERNAME"))
public class User {
    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (!(other instanceof User)) return false;
        User that = (User) other;
        return this.getUsername().equals(that.getUsername());
    }
    @Override
    public int hashCode() {
        return getUsername().hashCode();
    }
    // ...
}
```

You may have noticed that the `equals()` method code always accesses the properties of the “other” reference via getter methods. This is extremely important because the reference passed as `other` may be a Hibernate proxy, not the actual instance that holds the persistent state. We can't access the `username` field of a `User` proxy directly. To initialize the proxy to get the property value, we need to access it with a getter method. This is one point where

Hibernate isn't *completely* transparent, but it's good practice anyway to use getter methods instead of direct instance variable access.

Check the type of the `other` reference with `instanceof`, not by comparing the values of `getClass()`. Again, the `other` reference may be a proxy, which is a runtime-generated subclass of `User`, so `this` and `other` may not be exactly the same type but a valid super/subtype. You can find more about proxies in section 12.1.1.

We can now safely compare `User` references in persistent state:

```
em = emf.createEntityManager();
em.getTransaction().begin();
User a = em.find(User.class, USER_ID);
User b = em.find(User.class, USER_ID);
assertTrue(a == b);
assertTrue(a.equals(b));
assertEquals(a.getId(), b.getId());
em.getTransaction().commit();
em.close();
```

In addition, of course, we get correct behavior if we compare references to instances in the persistent and detached state:

```
em = emf.createEntityManager();
em.getTransaction().begin();
User c = em.find(User.class, USER_ID);
assertFalse(a == c);                                     #A
assertTrue(a.equals(c));                                #B
assertEquals(a.getId(), c.getId());
em.getTransaction().commit();
em.close();
Set<User> allUsers = new HashSet();
allUsers.add(a);
allUsers.add(b);
allUsers.add(c);
assertEquals(allUsers.size(), 1);                         #C
```

#A Comparing the two references will of course still be false.

#B Now they are equal.

#C The size of the set is finally correct.

For some other entities, the business key may be more complex, consisting of a combination of properties. Here are some hints that should help you identify a business key in the domain model classes:

- Consider what attributes users of the application will refer to when they have to identify an object (in the real world). How do users tell the difference between one element and another if they're displayed on the screen? This is probably the business key to look for.
- Every immutable attribute is probably a good candidate for the business key. Mutable attributes may be good candidates, too, if they're updated rarely or if you can control the case when they're updated—for example, by ensuring the instances aren't in a `Set` at the time.
- Every attribute that has a `UNIQUE` database constraint is a good candidate for the business key. Remember that the precision of the business key has to be good

enough to avoid overlaps.

- Any date or time-based attribute, such as the creation timestamp of the record, is usually a good component of a business key, but the accuracy of `System.currentTimeMillis()` depends on the virtual machine and operating system. Our recommended safety buffer is 50 milliseconds, which may not be accurate enough if the time-based property is the single attribute of a business key.
- You can use database identifiers as part of the business key. This seems to contradict our previous statements, but we aren't talking about the database identifier value of the given entity. You may be able to use the database identifier of an associated entity instance. For example, a candidate business key for the `Bid` class is the identifier of the `Item` it matches together with the bid amount. You may even have a unique constraint that represents this composite business key in the database schema. You can use the identifier value of the associated `Item` because it never changes during the life cycle of a `Bid`—the `Bid` constructor can require an already-persistent `Item`.

If you follow our advice, you shouldn't have much difficulty finding a good business key for all your business classes. If you encounter a difficult case, try to solve it without considering Hibernate. After all, it's purely an object-oriented problem. Notice that it's extremely rarely correct to override `equals()` on a subclass and include another property in the comparison. It's a little tricky to satisfy the `Object` identity and equality requirements that equality is both symmetric and transitive in this case; and, more important, the business key may not correspond to any well-defined candidate natural key in the database (subclass properties may be mapped to a different table). For more information on customizing equality comparisons, see *Effective Java*, 3rd edition, by Joshua Bloch (Bloch, 2017), a mandatory book for all Java programmers.

The `User` class is now prepared for the detached state; we can safely put instances loaded in different persistence contexts into a `Set`. Next, we'll look at some examples that involve the detached state, and you see some of the benefits of this concept.

Sometimes you might want to detach an entity instance manually from the persistence context.

10.3.3 Detaching entity instances

We don't have to wait for the persistence context to close. We can evict entity instances manually:

```
Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      detach()
User user = em.find(User.class, USER_ID);
em.detach(user);
assertFalse(em.contains(user));
```

This example also demonstrates the `EntityManager#contains()` operation, which returns `true` if the given instance is in the managed persistent state in this persistence context.

We can now work with the `user` reference in a detached state. Many applications only read and render the data after the persistence context is closed.

Modifying the loaded `user` after the persistence context is closed has no effect on its persistent representation in the database. JPA allows us to merge any changes back into the database in a new persistence context, though.

10.3.4 Merging entity instances

Let's assume we've retrieved a `User` instance in a previous persistence context, and now we want to modify it and save these modifications:

```
Path: managing-
      data/src/test/java/com/manning/javapersistence/ch10/SimpleTransitionsTest.java -
      mergeDetached()
detachedUser.setUsername("johndoe");
em = emf.createEntityManager();
em.getTransaction().begin();
User mergedUser = em.merge(detachedUser);
mergedUser.setUsername("doejohn");
em.getTransaction().commit();
em.close();
```

Consider the graphical representation of this procedure in figure 10.5. It's not as difficult as it seems.

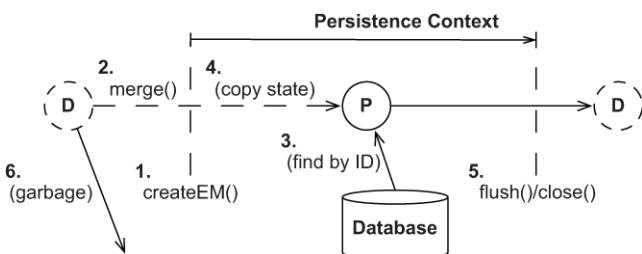


Figure 10.5 Making an instance persistent in a unit of work

The goal is to record the new `username` of the detached `User`. First, when we call `merge()`, Hibernate checks whether a persistent instance in the persistence context has the same database identifier as the detached instance we merging.

In this example, the persistence context is empty; nothing has been loaded from the database. Hibernate, therefore, loads an instance with this identifier from the database. Then, `merge()` copies the detached entity instance *onto* this loaded persistent instance. In other words, the new `username` we have set on the detached `User` is also set on the persistent `mergedUser`, which `merge()` returns to us.

Now discard the old reference to the stale and outdated detached state; the `detachedUser` no longer represents the current state. We can continue modifying the returned `mergedUser`; Hibernate will execute a single `UPDATE` when it flushes the persistence context during commit.

If there is no persistent instance with the same identifier in the persistence context, and a lookup by identifier in the database is negative, Hibernate instantiates a fresh `User`. Hibernate then copies our detached instance onto this fresh instance, which it inserts into the database when we synchronize the persistence context with the database.

If the instance we're giving to `merge()` is not detached but rather is transient (it doesn't have an identifier value), Hibernate instantiates a fresh `User`, copies the values of the transient `User` onto it, and then makes it persistent and returns it to us. In simpler terms, the `merge()` operation can handle detached *and* transient entity instances. Hibernate always returns the result to us as a persistent instance.

An application architecture based on detachment and merging may not call the `persist()` operation. We can merge new and detached entity instances to store data. The important difference is the returned current state and how we handle this switch of references in our application code. We have to discard the `detachedUser` and from now on reference the current `mergedUser`. Every other component in our application still holding on to `detachedUser` has to switch to `mergedUser`.

Can I reattach a detached instance?

The Hibernate Session API has a method for reattachment called `saveOrUpdate()`. It accepts either a transient or a detached instance and doesn't return anything. The given instance will be in a persistent state after the operation, so we don't have to switch references. Hibernate will execute an `INSERT` if the given instance was transient or an `UPDATE` if it was detached. We recommend that you rely on merging instead because it's standardized and therefore easier to integrate with other frameworks. In addition, instead of an `UPDATE`, merging may only trigger a `SELECT` if the detached data wasn't modified. If you're wondering what the `saveOrUpdateCopy()` method of the Session API does, it's the same as `merge()` on the EntityManager.

If we want to delete a detached instance, we have to merge it first. Then call `remove()` on the persistent instance returned by `merge()`.

10.4 Summary

- We analyzed the most important strategies and some optional ones for interacting with entity instances in a JPA application.
- We investigated the life cycle of entity instances and how they become persistent, detached, and removed.
- We extensively used the most important interface in JPA - EntityManager.
- We used the EntityManager to make data persistent, retrieve and modify persistent data, get a reference, make data transient, refresh and replicate data, cache in the persistence context and flush the persistence context.
- We worked with the detached state, using the identity of detached instances and implementing equality methods.

11

Transactions and concurrency

This chapter covers:

- Defining database and system transaction essentials
- Controlling concurrent access with Hibernate and JPA
- Using non-transactional data access
- Managing transactions with Spring and Spring Data

In this chapter, we finally talk about transactions: how we create and control concurrent units of work in an application. A *unit of work* is an atomic group of operations. Transactions allow us to set unit of work boundaries and help us isolate one unit of work from another. In a multiuser application, we may also be processing these units of work concurrently.

To handle concurrency, we first focus on units of work at the lowest level: database and system transactions. You'll learn the APIs for transaction demarcation and how to define units of work in Java code. We'll demonstrate how to preserve isolation and control concurrent access with pessimistic and optimistic strategies. The overall architecture of the system impacts the scope of a transaction. A bad architecture may lead to fragile transactions.

Then, we analyze some special cases and JPA features, based on accessing the database without explicit transactions. Finally, we'll demonstrate how to work with transactions with Spring and Spring Data.

Let's start with some background information.

Major new features in JPA 2

There are new lock modes and exceptions for pessimistic locking.

You can set a lock mode, pessimistic or optimistic, on a `Query`.

You can set a lock mode when calling `EntityManager#find()`, `refresh()`, or `lock()`. A lock timeout hint for pessimistic lock modes is also standardized.

When the new `QueryTimeoutException` or `LockTimeoutException` is thrown, the transaction doesn't have to be rolled back.

The persistence context can now be in an *unsynchronized* mode with disabled automatic flushing. This allows us to queue modifications until we join a transaction and to decouple the `EntityManager` usage from transactions.

11.1 Transaction essentials

Application functionality requires that several things be done in one go. For example, when an auction finishes, the `CaveatEmptor` application must perform three different tasks:

1. Find the winning bid (highest amount) for the auction item.
2. Charge the seller of the item the cost of the auction.
3. Notify the seller and successful bidder.

What happens if we can't bill the auction costs because of a failure in the external credit card system? The business requirements may state that either all listed actions must succeed or none must succeed. If so, we call these steps collectively a *transaction* or unit of work. If only a single step fails, the entire unit of work must fail.

11.1.1 ACID attributes

ACID stands for *atomicity*, *consistency*, *isolation*, *durability*. *Atomicity* is the notion that all operations in a transaction execute as an atomic unit. Furthermore, transactions allow multiple users to work concurrently with the same data without compromising the *consistency* of the data (consistent with database integrity rules). A particular transaction should not be visible to other concurrently running transactions; they should run in *isolation*. Changes made in a transaction should be *durable*, even if the system fails after the transaction has been completed successfully.

In addition, we want the *correctness* of a transaction. For example, the business rules dictate that the application charges the seller once, not twice. This is a reasonable assumption, but we may not be able to express it with database constraints. Hence, the correctness of a transaction is the responsibility of the application, whereas consistency is the responsibility of the database. Together, these transaction attributes define the *ACID* criteria.

11.1.2 Database and system transactions

We've also mentioned *system* and *database* transactions. Consider the last example again: during the unit of work ending an auction, we might mark the winning bid in a database system. Then, in the same unit of work, we talk to an external system to bill the seller's credit card. This is a transaction spanning several systems, with coordinated subordinate transactions on possibly several resources such as a database connection and an external billing processor. This chapter focuses on transactions spanning one system and one database.

Database transactions have to be short because open transactions consume database resources and potentially prevent concurrent access due to exclusive locks on data. A single database transaction usually involves only a single batch of database operations.

To execute all of the database operations inside a system transaction, we have to set the boundaries of that unit of work. We must start the transaction and, at some point, commit the changes. If an error occurs (either while executing database operations or when committing the transaction), we have to roll back the changes to leave the data in a consistent state. This process defines a *transaction demarcation* and, depending on the technique we use, involves a certain level of manual intervention. In general, transaction boundaries that begin and end a transaction can be set either programmatically in the application code or declaratively. We'll demonstrate both ways, focusing on declarative transactions while working with Spring and Spring Data.

All examples in this chapter work in any Java SE environment, without a special runtime container. Hence, from now on, you'll see programmatic transaction demarcation code until we move to specific Spring application examples.

Next, we focus on the most complex aspect of ACID properties: how you *isolate* concurrently running units of work from each other.

11.2 Controlling concurrent access

Databases (and other transactional systems) attempt to ensure transaction *isolation*, meaning that, from the point of view of each concurrent transaction, it appears that no other transactions are in progress. Traditionally, database systems have implemented isolation with locking. A transaction may place a lock on a particular item of data in the database, temporarily preventing read and/or write access to that item by other transactions. Some modern database engines implement transaction isolation with multi-version concurrency control (MVCC), which vendors generally consider more scalable. We'll analyze isolation assuming a locking model, but most of the observations are also applicable to MVCC.

How databases implement concurrency control is of the utmost importance in the Java Persistence application. Applications may inherit the isolation guarantees provided by the database management system, but frameworks may come on top of them and allow to start, commit and rollback transactions in a resource-agnostic way. If you consider the many years of experience that database vendors have with implementing concurrency control, you'll see the advantage of this approach. Additionally, some features in Java Persistence, either because you explicitly use them or by design, can improve the isolation guarantee beyond what the database provides.

We discuss concurrency control in several steps. First, we explore the lowest layer: the transaction isolation guarantees provided by the database. After that, you'll see the Java Persistence features for pessimistic and optimistic concurrency control at the application level, and what other isolation guarantees Hibernate can provide.

11.2.1 Understanding database-level concurrency

If we're talking about isolation, you may assume that two things are either isolated or not; there is no grey area in the real world. When we talk about database transactions, complete isolation comes at a high price. You can't stop the world to access data exclusively in a multiuser OLTP (Online transaction processing) system. Therefore, several isolation levels are available, which, naturally, weaken full isolation but increase the performance and scalability of the system.

TRANSACTION ISOLATION ISSUES

First, let's examine several phenomena that may occur when you weaken full transaction isolation. The ANSI SQL standard defines the standard transaction isolation levels in terms of which of these phenomena are permissible.

A *lost update* occurs when two concurrent transactions simultaneously update the same information from a database. The first transaction will read a value. The second transaction will start shortly and read the same value. The first transaction will change and write the updated value, while the second transaction will overwrite the value with its own update. Thus, the update of the first transaction is lost, being overwritten by the second transaction. The *last commit wins*. This occurs in systems that don't implement concurrency control, where concurrent transactions aren't isolated. This is shown in figure 11.1. The `buyNowPrice` field was updated from two transactions, but in reality, only one update occurred, the other update was lost.

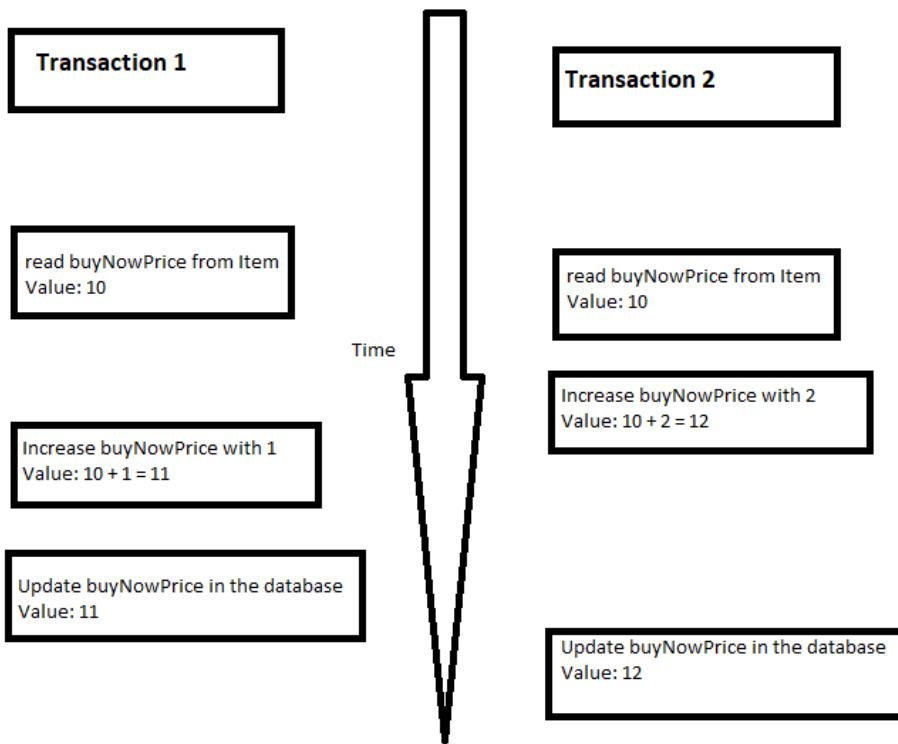


Figure 11.1 Lost update: two transactions update the same data without isolation.

A *dirty read* occurs if transaction 2 reads changes made by transaction 1, and this one hasn't yet been committed. This is dangerous because the changes made by transaction 1 may be later rolled back, and invalid data have been read by transaction 2; see figure 11.2.

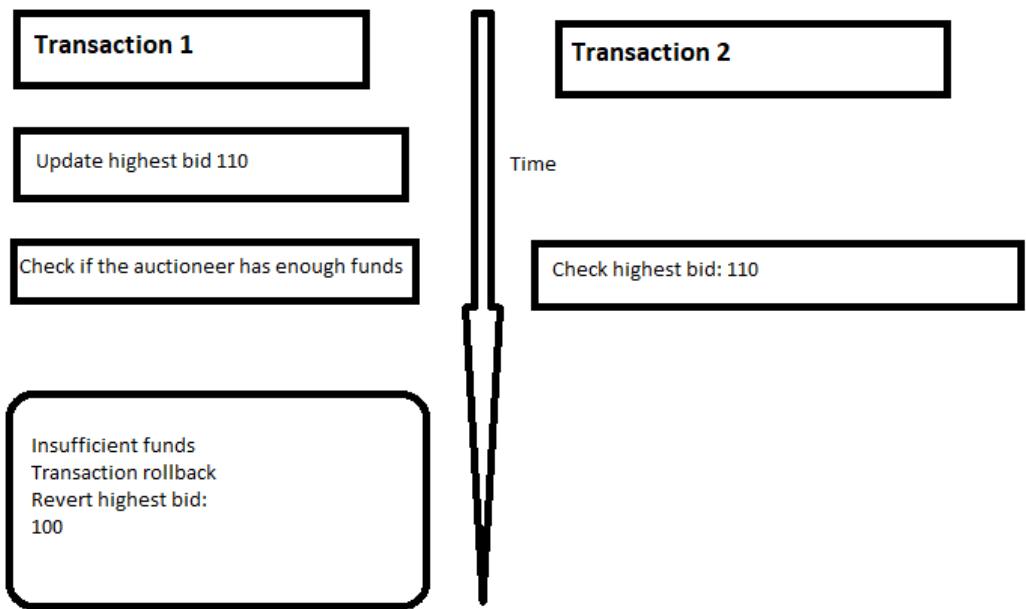


Figure 11.2 Dirty read: transaction 2 reads uncommitted data from transaction 1.

An *unrepeatable read* occurs if a transaction reads a data item twice and reads different states each time. For example, another transaction may have written to the data item and committed between the two reads, as shown in figure 11.3.

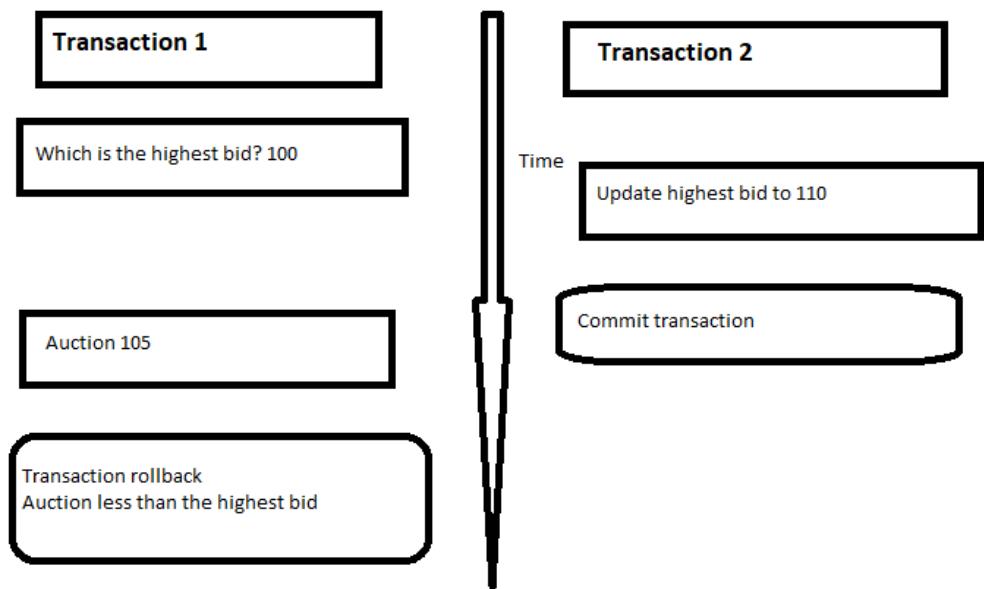


Figure 11.3 Unrepeatable read: the highest bid changed while transaction 1 was in execution.

A *phantom read* is said to occur when a transaction executes a query twice, and the second result includes data that wasn't visible in the first result because something was added, or it includes less data because something was deleted. It needs not necessarily be exactly the same query. Another transaction inserting or deleting data between the executions of the two queries causes this situation, as shown in figure 11.4.

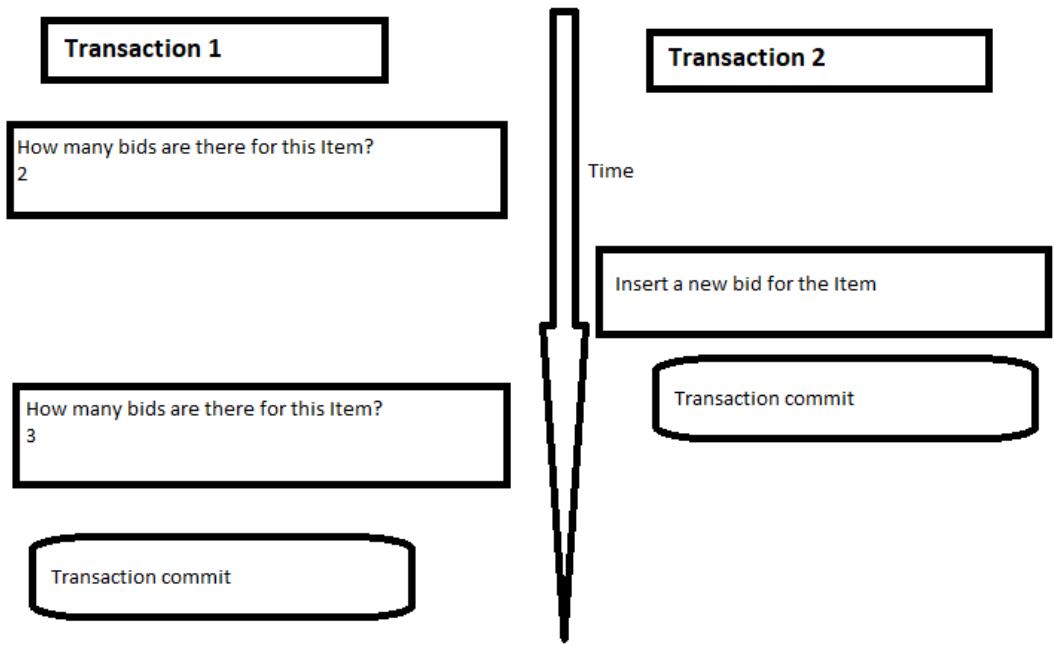


Figure 11.4 Phantom read: transaction 1 reads new data in the second query

Now that you understand all the bad things that can occur, we can define the transaction isolation levels and see what problems they prevent.

ANSI ISOLATION LEVELS

The standard isolation levels are defined by the ANSI SQL standard, but they aren't specific to SQL databases. Spring defines exactly the same isolation levels, and we'll use these levels to declare the desired transaction isolation. With increased levels of isolation come higher cost and serious degradation of performance and scalability:

- *Read uncommitted isolation*—A system that does not permit lost updates operates in read uncommitted isolation. One transaction may not write to a row if another uncommitted transaction has already written to it. Any transaction may read any row, however. A DBMS may implement this isolation level with exclusive write locks.
- *Read committed isolation*—A system that permits unrepeatable reads and phantom reads but neither lost updates, nor dirty reads implements read committed isolation. A DBMS may achieve this by using shared read locks and exclusive write locks. Reading transactions don't block other transactions from accessing a row, but an uncommitted writing transaction blocks all other transactions from accessing the row.
- *Repeatable read isolation*—A system operating in repeatable read isolation mode does not permit lost updates, dirty reads, or unrepeatable reads. Phantom reads may

occur. Reading transactions block writing transactions but do not block other reading transactions, and writing transactions block all other transactions.

- *Serializable isolation*—The strictest isolation, serializable, emulates serial execution as if transactions were executed one after another, rather than concurrently. A DBMS may not implement serializable using only row-level locks. A DBMS must instead provide some other mechanism that prevents a newly inserted row from becoming visible to a transaction that has already executed a query that would return the row. A crude mechanism is exclusively locking the entire database table after a write, so no phantom reads can occur.

Table 11.1 summarizes the ANSI isolation levels and the issues that they address.

Table 11.1 Isolation levels and the issues that they address

Isolation level	Phantom reading	Unrepeatable read	Dirty read	Lost update
READ UNCOMMITTED	-	-	-	+
READ COMMITTED	-	-	+	+
REPEATABLE READ	-	+	+	+
SERIALIZABLE	+	+	+	+

How exactly a DBMS implements its locking system varies significantly; each vendor has a different strategy. You should study the documentation of your DBMS to find out more about its locking system, how locks are escalated (from row-level to pages and to entire tables, for example), and what impact each isolation level has on the performance and scalability of the system.

It's nice to know how all these technical terms are defined, but how does that help us choose an isolation level for the application?

CHOOSING AN ISOLATION LEVEL

Developers (ourselves included) are often unsure what transaction isolation level to use in a production application. Too high an isolation level harms the scalability of a highly concurrent application. Insufficient isolation may cause subtle, difficult-to-reproduce bugs in an application that we won't discover until the system is working under heavy load.

Note that we refer to *optimistic locking* (with versioning) in the following explanation, a concept analyzed later in this chapter. You may revisit this section when it's time to pick an isolation level for your application. Choosing the correct isolation level is, after all, highly dependent on the particular scenario. Read the following discussion as recommendations, not dictums carved in stone.

Hibernate tries hard to be as transparent as possible regarding the transactional semantics of the database. Nevertheless, persistence context caching and versioning affect these semantics. What is a sensible database isolation level to choose in a JPA application?

First, for almost all scenarios, eliminate the *read uncommitted* isolation level. It's extremely dangerous to use one transaction's uncommitted changes in a different transaction. The rollback or failure of one transaction will affect other concurrent

transactions. Rollback of the first transaction could bring other transactions down with it, or perhaps even cause them to leave the database in an incorrect state (the seller of an auction item might be charged twice—consistent with database integrity rules but incorrect). It's possible that changes made by a transaction that ends up being rolled back could be committed anyway because they could be read and then propagated by another successful transaction! Eventually, you may use the *read uncommitted* isolation level for debugging purposes, to follow the execution of long insert queries, make some rough estimates of aggregate functions (as `SUM(*)` or `COUNT(*)`).

Second, most applications don't need *serializable* isolation. Phantom reads aren't usually problematic, and this isolation level tends to scale poorly. Few existing applications use serializable isolation in production, but rather rely on selectively applied pessimistic locks that effectively force a serialized execution of operations in certain situations.

Next, let's consider *repeatable read*. This level provides reproducibility for query result sets for the duration of a database transaction. This means we won't read committed updates from the database if we query it several times. But phantom reads are still possible: new rows might appear—rows we thought existed might disappear if another transaction committed such changes concurrently. Although we may sometimes want repeatable reads, we typically don't need them in every transaction.

The JPA specification assumes that read committed is the default isolation level.

This means we have to deal with unrepeatable reads and phantom reads.

Let's assume we're enabling versioning of our domain model entities, something that Hibernate can do for us automatically. The combination of the (mandatory) persistence context cache and versioning already gives most of the nice features of *repeatable read* isolation. The persistence context cache ensures that the state of the entity instances loaded by one transaction is isolated from changes made by other transactions. If we retrieve the same entity instance twice in a unit of work, the second lookup will be resolved within the persistence context cache and not hit the database. Hence, our read is repeatable, and we won't see conflicting committed data. (We still get phantom reads, though, which are typically much easier to deal with.) Additionally, versioning switches to *first commit wins*. Hence, for almost all multiuser JPA applications, *read committed* isolation for all database transactions is acceptable with enabled entity versioning.

Hibernate retains the isolation level of the database connection; it doesn't change the level. Most products default to read committed isolation. MySQL defaults to repeatable read. There are several ways we can change either the default transaction isolation level or the settings of the current transaction.

First, we can check whether the DBMS has a global transaction isolation level setting in its proprietary configuration. If the DBMS supports the standard SQL statement `SET SESSION CHARACTERISTICS`, we can execute it to set the transaction settings of all transactions started in this particular database *session* (which means a particular connection to the database, not a Hibernate *Session*). SQL also standardizes the `SET TRANSACTION` syntax, which sets the isolation level of the current transaction. Finally, the JDBC Connection API offers the `setTransactionIsolation()` method, which (according to its documentation) "attempts to change the transaction isolation level for this connection." In a Hibernate/JPA application, we can obtain a JDBC Connection from the native Session API.

Frequently, the database connections are by default in *read committed* isolation level. From time to time, a particular unit of work in the application may require a different, usually stricter isolation level. Instead of changing the isolation level of the entire transaction, we should use the Jakarta Persistence API to obtain additional locks on the relevant data. This fine-grained locking is more scalable in a highly concurrent application. JPA offers optimistic version checking and database-level pessimistic locking.

11.2.2 Optimistic concurrency control

Handling concurrency in an optimistic way is appropriate when concurrent modifications are rare and it's feasible to detect conflicts late in a unit of work. JPA offers automatic version checking as an optimistic conflict-detection procedure.

First, we'll enable versioning, because it's turned off by default. Most multiuser applications, especially web applications, should rely on versioning for any concurrently modified `@Entity` instances, enabling the more user-friendly *first commit wins*.

The previous sections have been somewhat dry; it's time for code. After enabling automatic version checking, we'll see how manual version checking works and when we have to use it. To be able to execute the examples from the source code, you need first to run the Ch11.sql script.

ENABLING VERSIONING

We enable versioning with an `@Version` annotation on a special additional property of the entity class, as demonstrated next.

Listing 11.1 Enabling versioning on a mapped entity

```
Path: Ch11/transactions/src/main/java/com/manning/javapersistence/ch11/concurrency/Item.java
@javax.persistence.Entity
public class Item {
    @javax.persistence.Version
    private long version;
    // ...
}
```

In this example, each entity instance carries a numeric version. It's mapped to an additional column of the `ITEM` database table; as usual, the column name defaults to the property name, here `VERSION`. The actual name of the property and column doesn't matter—we could rename it if `VERSION` is a reserved keyword in the DBMS.

We could add a `getVersion()` method to the class, but we shouldn't have a setter method and the application shouldn't modify the value. Hibernate automatically changes the version value: it increments the version number whenever an `Item` instance has been found dirty during flushing of the persistence context. The version is a simple counter without any useful semantic value beyond concurrency control. We can use an `int`, an `Integer`, a `short`, a `Short`, or a `Long` instead of a `long`; Hibernate wraps and starts from zero again if the version number reaches the limit of the data type.

After incrementing the version number of a detected dirty `Item` during flushing, Hibernate compares versions when executing the `UPDATE` and `DELETE` SQL statements. For example, assume that in a unit of work, we load an `Item` and change its name, as follows.

Listing 11.2 Hibernate incrementing and checking the version automatically

```
Path: /Ch11/transactions/src/test/java/com/manning/javapersistence/ch11/concurrency/Versioning.java - firstCommitWins()
EntityManager em1 = emf.createEntityManager();
em1.getTransaction().begin();
Item item = em1.find(Item.class, ITEM_ID); #A
// select * from ITEM where ID = ?
assertEquals(0, item.getVersion()); #B
item.setName("New Name");
//... Another transaction changes the record
assertThrows(OptimisticLockException.class, () -> em1.flush()); #C
// update ITEM set NAME = ?, VERSION = 1 where ID = ? and VERSION = 0
```

#A Retrieving an entity instance by identifier loads the current version from the database with a `SELECT`.

#B The current version of the `Item` instance is 0.

#C When the persistence context is flushed, Hibernate detects the dirty `Item` instance and increments its version to 1. `SQL UPDATE` now performs the version check, storing the new version in the database, but only if the database version is still 0.

Pay attention to the SQL statements, in particular, the `UPDATE` and its `WHERE` clause. This update will be successful only if there *is* a row with `VERSION = 0` in the database. JDBC returns the number of updated rows to Hibernate; if that result is zero, it means the `ITEM` row is either gone or doesn't have version 0 anymore. Hibernate detects this conflict during flushing, and a `javax.persistence.OptimisticLockException` is thrown.

Now imagine two users executing this unit of work at the same time, as shown previously in figure 11.1. The first user to commit updates the name of the `Item` and flushes the incremented version 1 to the database. The second user's flush (and commit) will fail because their `UPDATE` statement can't find the row in the database with version 0. The database version is 1. Hence, the *first commit wins*, and we can catch the `OptimisticLockException` and handle it specifically. For example, we could show the following message to the second user: "The data you have been working with has been modified by someone else. Please start your unit of work again with fresh data. Click the Restart button to proceed."

What modifications trigger the increment of an entity's version? Hibernate increments the version whenever an entity instance is dirty. This includes all dirty value-typed properties of the entity, no matter if they're single-valued (like a `String` or `int` property), embedded (like an `Address`), or collections. The exceptions are `@OneToOne` and `@ManyToOne` association collections that have been made read-only with `mappedBy`. Adding or removing elements to these collections doesn't increment the version number of the owning entity instance. You should know that none of this is standardized in JPA—don't rely on two JPA providers implementing the same rules when accessing a shared database.

Versioning with a shared database

If several applications access the database, and they don't all use Hibernate's versioning algorithm, we'll have concurrency problems. An easy solution is to use database-level triggers and stored procedures: An INSTEAD OF trigger can execute a stored procedure when any UPDATE is made; it runs instead of the update. In the procedure, we can check whether the application incremented the version of the row; if the version isn't updated or the version column isn't included in the update, we know the statement wasn't sent by a Hibernate application. We can then increment the version in the procedure before applying the UPDATE.

If we don't want to increment the version of the entity instance when a particular property's value has changed, annotate the property with `@org.hibernate.annotations.OptimisticLock(excluded = true)`.

You may not like the additional VERSION column in the database schema. Alternatively, you may already have a "last updated" timestamp property on the entity class and a matching database column. Hibernate can check versions with timestamps instead of the extra counter field.

VERSIONING WITH.timestamps

If the database schema already contains a timestamp column such as LASTUPDATED or MODIFIED_ON, we can map it for automatic version checking instead of using a numeric counter.

Listing 11.3 Enabling versioning with timestamps

```
Path: Ch11/transactions2/src/main/java/com/manning/javapersistence/ch11/timestamp/Item.java

@Entity
public class Item {
    @Version
    // Optional: @org.hibernate.annotations.Type(type = "dbtimestamp")
    private LocalDateTime lastUpdated;
    // ...
}
```

This example maps the column LASTUPDATED to a `java.time.LocalDateTime` property; a `Date` or `Calendar` type would also work with Hibernate. The JPA standard doesn't define these types for version properties; JPA only considers `java.sql.Timestamp` portable. This is less attractive because we'd have to import that JDBC class in the domain model. We should try to keep implementation details such as JDBC out of the domain model classes so they can be tested, instantiated, serialized, and deserialized in as many environments as possible.

In theory, versioning with a timestamp is slightly less safe, because two concurrent transactions may both load and update the same `Item` in the same millisecond; this is exacerbated by the fact that a JVM usually doesn't have millisecond accuracy (you should check your JVM and operating system documentation for the guaranteed precision). Furthermore, retrieving the current time from the JVM isn't necessarily safe in a clustered environment, where the system time of nodes may not be synchronized, or time synchronization isn't as accurate as you'd need for your transactional load.

You can switch to retrieval of the current time from the database machine by placing an `@org.hibernate.annotations.Type(type="dbtimestamp")` annotation on the version property. Hibernate now asks the database for the current time before updating. This gives a single source of time for synchronization. Not all Hibernate SQL dialects support this, so check the source of the configured dialect. In addition, there is always the overhead of hitting the database for every increment.

We recommend that new projects rely on versioning with a numeric counter, not timestamps. If you're working with a legacy database schema or existing Java classes, it may be impossible to introduce a version or timestamp property and column. If that's the case, Hibernate provides an alternative strategy.

VERSIONING WITHOUT VERSION NUMBERS OR TIMESTAMPS

If you don't have version or timestamp columns, Hibernate can still perform automatic versioning. This alternative implementation of versioning checks the current database state against the unmodified values of persistent properties at the time Hibernate retrieved the entity instance (or the last time the persistence context was flushed).

You enable this functionality with the proprietary Hibernate annotation `@org.hibernate.annotations.OptimisticLocking`:

```
Path: Ch11/transactions3/src/main/java/com/manning/javapersistence/ch11/versionall/Item.java

@Entity
@org.hibernate.annotations.OptimisticLocking(
    type = org.hibernate.annotations.OptimisticLockType.ALL)
@org.hibernate.annotations.DynamicUpdate
public class Item {
    // ...
}
```

For this strategy, you also have to enable dynamic SQL generation of `UPDATE` statements, using `@org.hibernate.annotations.DynamicUpdate` as explained in section 5.3.2.

Hibernate now executes the following SQL to flush a modification of an `Item` instance:

```
update ITEM set NAME = 'New Name'
where ID = 123
    and NAME = 'Old Name'
    and PRICE = '9.99'
    and DESCRIPTION = 'Some item for auction'
    and ...
    and SELLER_ID = 45
```

Hibernate lists all columns and their last known values in the `WHERE` clause. If any concurrent transaction has modified any of these values or even deleted the row, this statement returns with zero updated rows. Hibernate then throws an exception at flush time.

Alternatively, Hibernate includes only the modified properties in the restriction (only `NAME`, in this example) if you switch to `OptimisticLockType.DIRTY`. This means two units of work may modify the same `Item` concurrently, and Hibernate detects a conflict only if they both modify the same value-typed property (or a foreign key value). The `WHERE` clause of the

last SQL excerpt would be reduced to `where ID = 123 and NAME = 'Old Name'`. Someone else could concurrently modify the price, and Hibernate wouldn't detect any conflict. Only if the application modified the name concurrently, we would get a `javax.persistence.OptimisticLockException`.

In most cases, checking only dirty properties isn't a good strategy for business entities. It's probably not OK to change the price of an item if the description changes!

This strategy also doesn't work with detached entities and merging: if we merge a detached entity into a new persistence context, the "old" values aren't known. The detached entity instance will have to carry a version number or timestamp for optimistic concurrency control.

Automatic versioning in Java Persistence prevents lost updates when two concurrent transactions try to commit modifications on the same piece of data. Versioning can also help to obtain additional isolation guarantees manually when we need them.

MANUAL VERSION CHECKING

Here's a scenario that requires repeatable database reads: imagine there are some categories in the auction system and that each `Item` is in a `Category`. This is a regular `@ManyToOne` mapping of an `Item#category` entity association.

Let's say you want to sum up all item prices in several categories. This requires a query for all items in each category, to add up the prices. The problem is, what happens if someone moves an `Item` from one `Category` to another `Category` while you're still querying and iterating through all the categories and items? With read-committed isolation, the same `Item` might show up twice while your procedure runs!

To make the "get items in each category" reads repeatable, JPA's `Query` interface has a `setLockMode()` method. Look at the procedure in the following listing.

Listing 11.4 Requesting a version check at flush time to ensure repeatable reads

```
Path:
/Ch11/transactions/src/test/java/com/manning/javapersistence
    /ch11/concurrency/Versioning.java - manualVersionChecking()
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

BigDecimal totalPrice = new BigDecimal(0);
for (Long categoryId : CATEGORIES) {
    List<Item> items =                                     #A
        em.createQuery("select i from Item i where i.category.id = :catId")
            .setLockMode(LockModeType.OPTIMISTIC)
            .setParameter("catId", categoryId)
            .getResultList();
    for (Item item : items)
        totalPrice = totalPrice.add(item.getBuyNowPrice());
}
em.getTransaction().commit();                                #B
em.close();
assertEquals("108.00", totalPrice.toString());
```

- #A For each Category, query all Item instances with an OPTIMISTIC lock mode. Hibernate now knows it has to check each Item at flush time.
- #B For each Item loaded earlier with the locking query, Hibernate executes a SELECT during flushing. It checks whether the database version of each ITEM row is still the same as when it was loaded. If any ITEM row has a different version or the row no longer exists, an OptimisticLockException is thrown.

Don't be confused by the *locking* terminology: The JPA specification leaves open how exactly each LockModeType is implemented; for OPTIMISTIC, Hibernate performs version checking. There are no actual locks involved. We'll have to enable versioning on the Item entity class as explained earlier; otherwise, we can't use the optimistic LockModeTypes with Hibernate.

Hibernate doesn't batch or otherwise optimize the SELECT statements for manual version checking: if we sum up 100 items, we get 100 additional queries at flush time. A pessimistic approach, as we demonstrate later in this chapter, may be a better solution for this particular case.

FAQ: Why can't the persistence context cache prevent this problem?

The “get all items in a particular category” query returns item data in a `ResultSet`. Hibernate then looks at the primary key values in this data and first tries to resolve the rest of the details of each `Item` in the persistence context cache—it checks whether an `Item` instance has already been loaded with that identifier. This cache, however, doesn’t help in the example procedure: if a concurrent transaction moved an item to another category, that item might be returned several times in different `ResultSet`s. Hibernate will perform its persistence context lookup and say, “Oh, I’ve already loaded that `Item` instance; let’s use what we already have in memory.” Hibernate isn’t even aware that the category assigned to the item changed or that the item appeared again in a different result. Hence this is a case where the repeatable-read feature of the persistence context hides concurrently committed data. We need to manually check the versions to find out if the data changed while we were expecting it not to change.

As shown in the previous example, the `Query` interface accepts a `LockModeType`. Explicit lock modes are also supported by the `TypedQuery` and the `NamedQuery` interfaces, with the same `setLockMode()` method.

An additional optimistic lock mode is available in JPA, forcing an increment of an entity’s version.

FORCING A VERSION INCREMENT

What happens if two users place a bid for the same auction item at the same time? When a user makes a new bid, the application must do several things:

1. Retrieve the currently highest `Bid` for the `Item` from the database.
2. Compare the new `Bid` with the highest `Bid`; if the new `Bid` is higher, it must be stored in the database.

There is the potential for a race condition in between these two steps. If, in between reading the highest `Bid` and placing the new `Bid`, another `Bid` is made, you won’t see it. This conflict isn’t visible; even enabling versioning of the `Item` doesn’t help. The `Item` is never modified during the procedure. Forcing a version increment of the `Item` makes the conflict detectable.

Listing 11.5 Forcing a version increment of an entity instance

```

Path: /Ch11/transactions/src/test/java/com/manning/javapersistence/ch11/concurrency/Versioning.java - forceIncrement()
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Item item = em.find(                                     #A
    Item.class,
    ITEM_ID,
    LockModeType.OPTIMISTIC_FORCE_INCREMENT
);
Bid highestBid = queryHighestBid(em, item);
Bid newBid = new Bid(
    new BigDecimal("45.45"),
    item,
    highestBid);
em.persist(newBid);                                    #B
assertThrows(RollbackException.class,
    () -> em.getTransaction().commit());                #C
em.close();

```

- #A `find()` accepts a `LockModeType`. The `OPTIMISTIC_FORCE_INCREMENT` mode tells Hibernate that the version of the retrieved `Item` should be incremented after loading, even if it's never modified in the unit of work.
- #B The code persists a new `Bid` instance; this doesn't affect any values of the `Item` instance. A new row is inserted into the `BID` table. Hibernate wouldn't detect concurrently made bids without a forced version increment of the `Item`.
- #C When flushing the persistence context, Hibernate executes an `INSERT` for the new `Bid` and forces an `UPDATE` of the `Item` with a version check. If someone modified the `Item` concurrently or placed a `Bid` concurrently with this procedure, Hibernate throws an exception.

For the auction system, placing bids concurrently is certainly a frequent operation. Incrementing a version manually is useful in many situations where we insert or modify data and want the version of some root instance of an aggregate to be incremented.

Note that if instead of a `Bid#item` entity association with `@ManyToOne`, we have an `@ElementCollection` of `Item#bids`, adding a `Bid` to the collection *will* increment the `Item` version. The forced increment then isn't necessary. You may want to review the discussion of parent/child ambiguity and how aggregates and composition work with ORM in section 8.3.

So far, we've focused on optimistic concurrency control: we expect that concurrent modifications are rare, so we don't prevent concurrent access and detect conflicts late. Sometimes we know that conflicts will happen frequently, and we want to place an exclusive lock on some data. This calls for a pessimistic approach.

11.2.3 Explicit pessimistic locking

Let's repeat the procedure demonstrated in the section "Manual version checking" with a pessimistic lock instead of optimistic version checking. We again summarize the total price of all items in several categories. This is the same code as shown earlier in listing 11.5, with a different `LockModeType`.

Listing 11.6 Locking data pessimistically

```

Path:
/Ch11/transactions/src/test/java/com/manning/javapersistence /ch11/concurrency/Locking.java
    - pessimisticReadWrite()
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

BigDecimal totalPrice = new BigDecimal(0);
for (Long categoryId : CATEGORIES) {
    List<Item> items =
        em.createQuery("select i from Item i where i.category.id = :catId") #A
            .setLockMode(LockModeType.PESSIMISTIC_READ)
            .setHint("javax.persistence.lock.timeout", 5000)
            .setParameter("catId", categoryId)
            .getResultList();
    for (Item item : items) #B
        totalPrice = totalPrice.add(item.getBuyNowPrice());
}
em.getTransaction().commit(); #C
em.close();
assertEquals(0, totalPrice.compareTo(new BigDecimal("108")));

```

#A For each Category, query all Item instances in PESSIMISTIC_READ lock mode. Hibernate locks the rows in the database with the SQL query. If possible, wait 5 seconds if another transaction holds a conflicting lock. If the lock can't be obtained, the query throws an exception.

#B If the query returns successfully, we know that we hold an exclusive lock on the data, and no other transaction can access it with an exclusive lock or modify it until this transaction commits.

#C The locks are released after commit when the transaction completes.

The JPA specification defines that the lock mode PESSIMISTIC_READ guarantees repeatable reads. JPA also standardizes the PESSIMISTIC_WRITE mode, with additional guarantees: in addition to repeatable reads, the JPA provider must serialize data access, and no phantom reads can occur.

It's up to the JPA provider to implement these requirements. For both modes, Hibernate appends a "for update" clause to the SQL query when loading data. This places a lock on the rows at the database level. What kind of lock Hibernate uses depends on the LockModeType and the Hibernate database dialect.

On H2, for example, the query is `SELECT * FROM ITEM ... FOR UPDATE`. Because H2 supports only one type of exclusive lock, Hibernate generates the same SQL for all pessimistic modes.

PostgreSQL, on the other hand, supports shared read locks: the PESSIMISTIC_READ mode appends `FOR SHARE` to the SQL query. PESSIMISTIC_WRITE uses an exclusive write lock with `FOR UPDATE`.

On MySQL, PESSIMISTIC_READ translates to LOCK IN SHARE MODE, and PESSIMISTIC_WRITE to FOR UPDATE.

Check your database dialect. This is configured with the `getReadLockString()` and `getWriteLockString()` methods.

The duration of a pessimistic lock in JPA is a single database transaction. This means we can't use an exclusive lock to block concurrent access for longer than a single database transaction. When the database lock can't be obtained, an exception is thrown. Compare this with an optimistic approach, where Hibernate throws an exception at commit time, not when

you query. With a pessimistic strategy, we know that we can read and write the data safely as soon as the locking query succeeds. With an optimistic approach, we hope for the best and may be surprised later, when we commit.

Offline locks

Pessimistic database locks are held only for a single transaction. Other lock implementations are possible: for example, a lock held in memory, or a so-called *lock table* in the database. A common name for these kinds of locks is offline locks.

Locking pessimistically for longer than a single database transaction is usually a performance bottleneck; every data access involves additional lock checks to a globally synchronized lock manager. Optimistic locking, however, is the perfect concurrency control strategy for long-running conversations (as you'll see in the next chapter) and performs well. Depending on the conflict-resolution strategy—what happens after a conflict is detected—the application users may be just as happy as with blocked concurrent access. They may also appreciate the application not locking them out of particular screens while others look at the same data.

We can configure how long the database will wait to obtain the lock and block the query in milliseconds with the `javax.persistence.lock.timeout` hint. As usual with hints, Hibernate might ignore it, depending on the database product. H2, for example, doesn't support lock timeouts per query, only a global lock timeout per connection (defaulting to 1 second). With some dialects, such as PostgreSQL and Oracle, a lock timeout of 0 appends the `NOWAIT` clause to the SQL string.

We've demonstrated the lock timeout hint applied to a `Query`. We can also set the timeout hint for `find()` operations:

```

Path: /Ch11/transactions/src/test/java/com/manning/javapersistence/ch11/concurrency/Locking.java
- findLock()

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Map<String, Object> hints = new HashMap<>();
hints.put("javax.persistence.lock.timeout", 5000);

Category category =
    em.find(
        Category.class,
        CATEGORY_ID,
        LockModeType.PESSIMISTIC_WRITE,
        Hints
    );

category.setName("New Name");

em.getTransaction().commit();
em.close();

```

#A Executes SELECT ... FOR UPDATE WAIT 5000, if supported by the dialect

When a lock can't be obtained, Hibernate throws either a `javax.persistence.LockTimeoutException` or a `javax.persistence.PessimisticLockException`. If Hibernate throws a `PessimisticLockException`, the transaction must be rolled back, and the unit of work ends. A timeout exception, on the other hand, isn't fatal for the transaction, as explained in section 11.1.4. Which exception Hibernate throws again depends on the SQL dialect. For example, because H2 doesn't support per-statement lock timeouts, we always get a `PessimisticLockException`.

We can use both the `PESSIMISTIC_READ` and `PESSIMISTIC_WRITE` lock modes even if we haven't enabled entity versioning. They translate to SQL statements with database-level locks.

The special mode `PESSIMISTIC_FORCE_INCREMENT` requires versioned entities, however. In Hibernate, this mode executes a `FOR UPDATE NOWAIT` lock (or whatever the dialect supports; check its `getForUpdateNowaitString()` implementation). Then, immediately after the query returns, Hibernate increments the version and `UPDATE (!)` each returned entity instance. This indicates to any concurrent transaction that we have updated these rows, even if we haven't so far modified any data. This mode is rarely useful, mostly for aggregate locking as analyzed in the section "Forcing a version increment."

What about lock modes READ and WRITE?

These are older lock modes from JPA 1.0, and you should no longer use them. `LockModeType.READ` is equivalent to `OPTIMISTIC`, and `LockModeType.WRITE` is equivalent to `OPTIMISTIC_FORCE_INCREMENT`.

If we enable pessimistic locking, Hibernate locks only rows that correspond to entity instance state. In other words, if we lock an `Item` instance, Hibernate will lock its row in the `ITEM` table. If we have a joined inheritance mapping strategy, Hibernate will recognize this and lock the appropriate rows in super- and sub-tables. This also applies to any secondary table mappings of an entity. Because Hibernate locks entire rows, any relationship where the foreign key is in that row will also effectively be locked: the `Item#seller` association is locked if the `SELLER_ID` foreign key column is in the `ITEM` table. The actual `Seller` instance isn't locked! Neither are collections or other associations of the `Item` where the foreign key(s) are in other tables.

Extending lock scope

JPA 2.0 defines the `PessimisticLockScope.EXTENDED` option. It can be set as a query hint with `javax.persistence.lock.scope`. If enabled, the persistence engine expands the scope of locked data to include any data in collection and association join tables of locked entities.

With exclusive locking in the DBMS, you may experience transaction failures because you run into deadlock situations.

11.2.4 Avoiding deadlocks

Deadlocks can occur if the DBMS relies on exclusive locks to implement transaction isolation. Consider the following unit of work, updating two `Item` entity instances in a particular order:

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Item itemOne = em.find(Item.class, ITEM_ONE_ID);
itemOne.setName("First new name");
Item itemTwo = em.find(Item.class, ITEM_TWO_ID);
itemTwo.setName("Second new name");
em.getTransaction().commit();
em.close();
```

Hibernate executes two SQL `UPDATE` statements when the persistence context is flushed. The first `UPDATE` locks the row representing `Item` one, and the second `UPDATE` locks `Item` two:

<pre>update ITEM set ... where ID = 1; update ITEM set ... where ID = 2;</pre>	#A #B
--	----------

#A Locks row 1
#B Attempts to lock row 2

A deadlock may (or it may not!) occur if a similar procedure, with the opposite order of `Item` updates, executes in a concurrent transaction:

<pre>update ITEM set ... where ID = 2;</pre>	#C
--	----

```
update ITEM set ... where ID = 1; #D
#C Locks row 2
#D Attempts to lock row 1
```

With a deadlock, both transactions are blocked and can't move forward, each waiting for a lock to be released. The chance of a deadlock is usually small, but in highly concurrent applications, two Hibernate applications may execute this kind of interleaved update. Note that we may not see deadlocks during testing (unless we write the right kinds of tests). Deadlocks can suddenly appear when the application has to handle a high transaction load in production. Usually, the DBMS terminates one of the deadlocked transactions after a timeout period and fails; the other transaction can then proceed. Alternatively, depending on the DBMS, the DBMS may detect a deadlock situation automatically and immediately abort one of the transactions.

You should try to avoid transaction failures because they're difficult to recover from in application code. One solution is to run the database connection in *Serializable* mode when updating a single row locks the entire table. The concurrent transaction has to wait until the first transaction completes its work. Alternatively, the first transaction can obtain an exclusive lock on all data when you `SELECT` the data, as demonstrated in the previous section. Then any concurrent transaction also has to wait until these locks are released.

An alternative pragmatic optimization that significantly reduces the probability of deadlocks is to order the `UPDATE` statements by primary key value: Hibernate should always update the row with primary key 1 before updating row 2, no matter in what order the data was loaded and modified by the application. You can enable this optimization for the entire persistence unit with the configuration property `hibernate.order_updates`. Hibernate then orders all `UPDATE` statements it executes in ascending order by the primary key value of the modified entity instances and collection elements detected during flushing. (As mentioned earlier, make sure you fully understand the transactional and locking behavior of your DBMS product. Hibernate inherits most of its transaction guarantees from the DBMS; for example, your MVCC database product may avoid read locks but probably depends on exclusive locks for writer isolation, and you may see deadlocks.)

We didn't have an opportunity to mention the `EntityManager#lock()` method. It accepts an already-loaded persistent entity instance and a lock mode. It performs the same locking you've seen with `find()` and a `Query`, except that it doesn't load the instance. Additionally, if a versioned entity is being locked pessimistically, the `lock()` method performs an immediate version check on the database and potentially throws an `OptimisticLockException`. If the database representation is no longer present, Hibernate throws an `EntityNotFoundException`. Finally, the `EntityManager#refresh()` method also accepts a lock mode, with the same semantics.

We've now covered concurrency control at the lowest level—the database—and the optimistic and pessimistic locking features of JPA. We still have one more aspect of concurrency to examine: accessing data outside of a transaction.

11.3 Non-transactional data access

A JDBC `Connection` is by default in *auto-commit* mode. This mode is useful for executing ad hoc SQL.

Imagine that you connect to the database with an SQL console and that you run a few queries, and maybe even update and delete rows. This interactive data access is ad hoc; most of the time you don't have a plan or a sequence of statements that you consider a unit of work. The default auto-commit mode on the database connection is perfect for this kind of data access—after all, you don't want to type `begin transaction` and `end transaction` for every SQL statement you write and execute. In auto-commit mode, a (short) database transaction begins and ends for each SQL statement you send to the database. You're working effectively in the non-transactional mode because there are no atomicity or isolation guarantees for your session with the SQL console. (The only guarantee is that a single SQL statement is atomic.)

An application, by definition, always executes a planned sequence of statements. It seems reasonable that you therefore always create transaction boundaries to group the statements into units that are atomic and isolated from each other. In JPA, however, special behavior is associated with auto-commit mode, and you may need it to implement long-running conversations. You can access the database in auto-commit mode and read data.

11.3.1 Reading data in auto-commit mode

Consider the following example, which loads an `Item` instance, changes its `name`, and then rolls back that change by refreshing.

No transaction is active when we create the `EntityManager`. The persistence context will be in a special unsynchronized mode; Hibernate won't flush automatically.

You can access the database to read data, and such an operation executes a `SELECT`, sent to the database in auto-commit mode.

Usually Hibernate flushes the persistence context when you execute a `Query`. If the context is unsynchronized, flushing doesn't occur, and the query returns the old, original database value. Queries with scalar results aren't repeatable: you see whatever values are in the database and given to Hibernate in the `ResultSet`. This also isn't a repeatable read if you're in synchronized mode.

Retrieving a managed entity instance involves a lookup during JDBC result-set marshaling, in the current persistence context. The already-loaded instance with the changed name is returned from the persistence context; values from the database are ignored. This is a repeatable read of an entity instance, even without a system transaction.

If you try to flush the persistence context manually, to store a new `Item#name`, Hibernate throws a `javax.persistence.TransactionRequiredException`. You can't execute an `UPDATE` in unsynchronized mode, because you wouldn't be able to roll back the change.

You can rollback the change you made with the `refresh()` method. It loads the current `Item` state from the database and overwrites the change you made in memory.

Listing 11.7 Reading data in auto-commit mode

```

Path: Ch11/transactions4/src/test/java/com/manning/javapersistence/ch11/concurrency/NonTransactional.java

    EntityManager em = emf.createEntityManager();                                     #A
Item item = em.find(Item.class, ITEM_ID);                                         #B
item.setName("New Name");
assertEquals(                                            #C
    "Original Name",
    em.createQuery("select i.name from Item i where i.id = :id")
        .setParameter("id", ITEM_ID).getSingleResult()
);
assertEquals(                                            #D
    "New Name",
    ((Item) em.createQuery("select i from Item i where i.id = :id")
        .setParameter("id", ITEM_ID).getSingleResult()).getName()
);
// em.flush();                                         #E
em.refresh(item);                                         #F
assertEquals("Original Name", item.getName());
em.close();

```

#A No transaction active when creating the EntityManager.

#B Access the database to read data.

#C Because the context is unsynchronized, flushing doesn't occur, and the query returns the old, original database value.

#D The already-loaded Item instance with the changed name is returned from the persistence context; values from the database are ignored.

#E Cannot execute an UPDATE in unsynchronized mode, because you wouldn't be able to roll back the change.

#F Rollback the change you made with the refresh() method.

With an unsynchronized persistence context, you read data in auto-commit mode with find(), getReference(), refresh(), or queries. You can load data on demand as well: proxies are initialized if you access them, and collections are loaded if you start iterating through their elements. But if you try to flush the persistence context or lock data with anything but LockModeType.NONE, a TransactionRequiredException will occur.

So far, the auto-commit mode doesn't seem very useful. Indeed, many developers often rely on auto-commit for the wrong reasons:

- Many small per-statement database transactions (that's what auto-commit means) won't improve the performance of the application.
- You won't improve the scalability of the application: a longer-running database transaction, instead of many small transactions for every SQL statement, may hold database locks for a longer time. This is a minor issue, because Hibernate writes to the database as late as possible within a transaction (flush at commit), so the database already holds write locks for a short time.
- You also have weaker isolation guarantees if the application modifies data concurrently. Repeatable reads based on read locks are impossible with auto-commit mode. (The persistence context cache helps here, naturally.)
- If your DBMS has MVCC (for example, Oracle, PostgreSQL), you likely want to use its capability for *snapshot isolation* to avoid unrepeatable and phantom reads. Each transaction gets its own snapshot of the data; you only see a (database-internal)

version of the data as it was before your transaction started. With auto-commit mode, snapshot isolation makes no sense, because there is no transaction scope.

- Your code will be more difficult to understand. Any reader of your code now has to pay special attention to whether a persistence context is joined with a transaction, or if it's in unsynchronized mode. If you always group operations within a system transaction, even if you only read data, everyone can follow this simple rule, and the likelihood of difficult-to-find concurrency issues is reduced.

So, what are the benefits of an unsynchronized persistence context? If flushing doesn't happen automatically, you can prepare and *queue* modifications outside of a transaction.

11.3.2 Queueing modifications

The following example stores a new `Item` instance with an unsynchronized `EntityManager`.

You can call `persist()` to save a transient entity instance with an unsynchronized persistence context. Hibernate only fetches a new identifier value, typically by calling a database sequence, and assigns it to the instance. The instance will be in a persistent state in the context, but the SQL `INSERT` hasn't happened. Note that this is only possible with `pre-insert` identifier generators; see section 5.2.5.

When you're ready to store the changes, you must join the persistence context with a transaction. Synchronization and flushing occur as usual when the transaction commits. Hibernate writes all queued operations to the database.

```
Path: Ch11/transactions4/src/test/java/com/manning/javapersistence/ch11/concurrency/NonTransactional.java
EntityManager em = emf.createEntityManager();
Item newItem = new Item("New Item");
em.persist(newItem); #A
assertNotNull(newItem.getId());
em.getTransaction().begin(); #B
if (!em.isJoinedToTransaction())
    em.joinTransaction();
em.getTransaction().commit(); #C
em.close();
```

#A Call `persist()` to save a transient entity instance with an unsynchronized persistence context.

#B Join the persistence context with a transaction.

#C Synchronization and flushing occur when the transaction commits.

Merged changes of a detached entity instance can also be queued:

```
Path: Ch11/transactions4/src/test/java/com/manning/javapersistence/ch11/concurrency/NonTransactional.java
detachedItem.setName("New Name");
EntityManager em = emf.createEntityManager();
Item mergedItem = em.merge(detachedItem); #A
em.getTransaction().begin();
em.joinTransaction();
em.getTransaction().commit(); #B
em.close();
```

#A Hibernate executes a `SELECT` in auto-commit mode when you `merge()`.

#B Hibernate defers the `UPDATE` until a joined transaction commits.

Queuing also works for removal of entity instances and `DELETE` operations:

```
Path: Ch11/transactions4/src/test/java/com/manning/javapersistence/ch11/concurrency/NonTransactional.java
EntityManager em = emf.createEntityManager();
Item item = em.find(Item.class, ITEM_ID);
em.remove(item);
em.getTransaction().begin();
em.joinTransaction();
em.getTransaction().commit();
em.close();
```

An unsynchronized persistence context, therefore, allows you to decouple persistence operations from transactions. The ability to queue data modifications, independent from transactions (and client/server requests), is a major feature of the persistence context.

Hibernate's MANUAL flush mode

Hibernate offers a `Session#setFlushMode()` method, with the additional `FlushMode.MANUAL`. It's a much more convenient switch that disables any automatic flushing of the persistence context, even when a joined

transaction commits. With this mode, you have to call `flush()` explicitly to synchronize with the database. In JPA, the idea was that a “transaction commit should always write any outstanding changes”, so reading was separated from writing with the *unsynchronized* mode. If you don’t agree with this and/or don’t want auto-committed statements, enable manual flushing with the `Session API`. You can then have regular transaction boundaries for all units of work, with repeatable reads and even snapshot isolation from your MVCC database, but still, queue changes in the persistence context for later execution and manual `flush()` before your transaction commits.

11.4 Managing transactions with Spring and Spring Data

We move now to demonstrate how to implement transactions with Spring and Spring Data. The transactional model used by Spring is applicable for different APIs as Hibernate, JPA, Spring Data JPA. The management of the transactions can be either programmatic (as we have previously already demonstrated) and declarative, with the help of annotations (as we’ll mostly use for this part of the chapter).

The key Spring transaction abstraction is defined by the `org.springframework.transaction.PlatformTransactionManager` interface.

```
public interface PlatformTransactionManager extends TransactionManager {
    TransactionStatus getTransaction(@Nullable TransactionDefinition var1)
        throws TransactionException;
    void commit(TransactionStatus var1) throws TransactionException;
    void rollback(TransactionStatus var1) throws TransactionException;
}
```

Typically, this interface is not used directly. You will either mark transactions declaratively, through annotations, or you may eventually use `TransactionTemplate` for the programmatic transaction definition.

Spring uses the previously analyzed ANSI isolation levels. You may revisit section 11.2.1 and in particular table 11.1, which summarizes the isolation levels and the issues that they address.

11.4.1 Transaction propagation

Spring deals with the transaction propagation problem. In brief, if `method-A` is transactional and this one calls `method-B`, how will this last one behave from the transactional point of view (see figure 11.5)?

Bean1 contains `method-A`, that is transactional, executed in transaction TX1.

`method-A` calls `bean-2.method-B()`, which is also transactional.

In which transaction will be `method-B` executed?

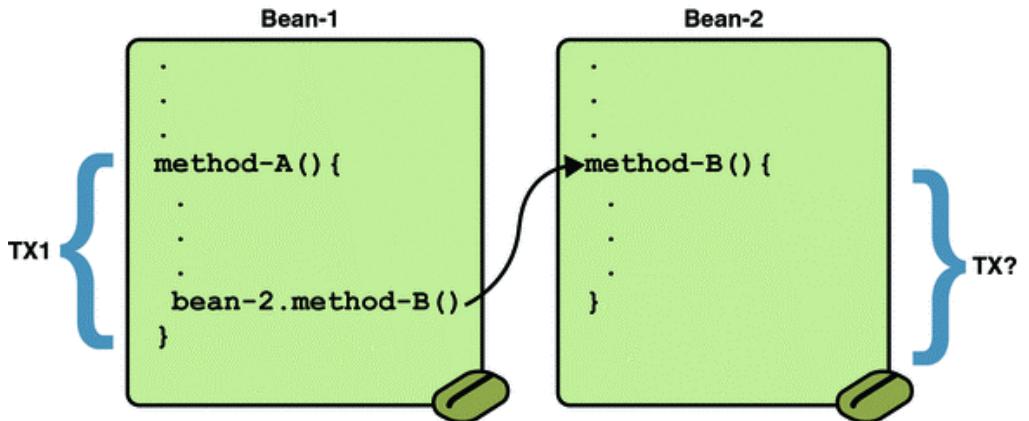


Figure 11.5 The transaction propagation concept

Spring defines the list of possible propagations through the org.springframework.transaction.annotation.Propagation enum.

- REQUIRED means that, if a transaction is in progress, the execution will continue within that transaction. Otherwise, a new transaction will be created. REQUIRED is the default propagation for transactions in Spring.
- SUPPORTS means that, if a transaction is in progress, the execution will continue within that transaction. Otherwise, no transaction will be created.
- MANDATORY means that, if a transaction is in progress, the execution will continue within that transaction. Otherwise, a `TransactionRequiredException` exception will be thrown.
- REQUIRES_NEW means that, if a transaction is in progress, it will be suspended and a new transaction will be started. Otherwise, a new transaction will be created anyway.
- NOT_SUPPORTED means that, if a transaction is in progress, it will be suspended and a non-transactional execution will continue. Otherwise, the execution will simply continue.
- NEVER means that, if a transaction is in progress, an `IllegalTransactionStateException` will be thrown. Otherwise, the execution will simply continue.
- NESTED means that, if a transaction is in progress, a subtransaction of this one will be created, and at the same time a savepoint will be created. If the subtransaction fails, the execution will rollback to this savepoint. If no transaction was originally in progress, a new transaction will be created.

Table 11.2 summarizes the possible transaction propagations in Spring.

Table 11.2 Transactions propagation in Spring

Transaction propagation	Transaction in the caller method	Transaction in the called method
REQUIRED	No	T1
	T1	T1
SUPPORTS	No	No
	T1	T1
MANDATORY	No	Exception
	T1	T1
REQUIRES_NEW	No	T1
	T1	T2
NOT_SUPPORTED	No	No
	T1	No
NEVER	No	No
	T1	Exception
NESTED	No	T1
	T1	T2 with savepoint

11.4.2 Transaction rollback

Spring transactions define the default rollback rules: a transaction will be rolled back for `RuntimeException`. This behavior may be overwritten and we may specify which exceptions automatically rollback the transaction and which will not do it. This is done with the help of the `@Transactional` annotation properties `rollbackFor`, `rollbackForClassname`, `noRollbackFor`, `noRollbackForClassname`. The behavior determined by these properties is summarized in table 11.3.

Table 11.3 Transactions rollback rules

Property	Type	Behavior
<code>rollbackFor</code>	Array of Class objects extending <code>Throwable</code>	Defines exception classes that must cause rollback
<code>rollbackForClassname</code>	Array of Class names extending <code>Throwable</code>	Defines exception class names that must cause rollback
<code>noRollbackFor</code>	Array of Class objects extending <code>Throwable</code>	Defines exception classes that must not cause rollback
<code>noRollbackForClassname</code>	Array of Class names extending <code>Throwable</code>	Defines exception class names that must not cause rollback

11.4.3 Transaction properties

The `@Transactional` annotation defines the properties from table 11.4. We deal here with the already examined isolation and propagation, but also with other properties. All the metainformation will be transposed at the level of how the transactional operation is executed.

Table 11.4 `@Transactional` annotation properties

Property	Type	Behavior
isolation	Isolation enum	Declares the isolation levels, following the ANSI standard.
propagation	Propagation enum	Propagation settings following the values from table 11.2.
timeout	int (seconds)	Timeout after which the transaction will automatically rollback.
readOnly	boolean	Declares if the transaction is read-only or read-write. Read-only transactions allow optimizations that can make them faster.

The `@Transactional` annotation may be applied to interfaces, to methods in interfaces, to classes, or to methods in classes. Once applied to an interface or to a class, the annotation is taken over by all methods from that class or from that interface. You can change the behavior by annotating particular methods in a different way. Also, once applied to an interface or to a method in an interface, the annotation is taken over by the classes implementing that interface or by the corresponding methods from the classes implementing that interface. The behavior can be overwritten. Consequently, for a fine-grained behavior, it is advisable to apply the `@Transactional` annotation to methods from classes.

11.4.4 Programmatic transaction definition

Declarative transactions management is generally the way to go when using Spring in an application. It requires less code to write, and the behavior is determined through the metainformation provided by the annotations. However, programmatic transactions management is still possible, using the `TransactionTemplate` class.

Once a `TransactionTemplate` object is created, the behavior of the transaction can be defined programmatically, as in the code below.

```

TransactionTemplate transactionTemplate;
//...
transactionTemplate.setIsolationLevel(
    TransactionDefinition.ISOLATION_REPEATABLE_READ);
transactionTemplate.setPropagationBehavior(
    TransactionDefinition.PROPAGATION_REQUIRE_NEW);
transactionTemplate.setTimeout(5);
transactionTemplate.setReadOnly(false);

```

Once defined, a `TransactionTemplate` object supports callback approach through the `execute` method. This one receives as argument a `TransactionCallback`, as in the code below. The operations to be executed in transaction are defined in the `doInTransaction` method.

```

transactionTemplate.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        //operations to be executed in transaction
    }
});

```

11.4.5 Transactional development with Spring and Spring Data

We are currently working on the `CaveatEmptor` application. We have to implement a feature concerning working with items and logging the result of these actions. We'll start the implementation using Spring Data JPA and will first create the `ItemRepositoryCustom` interface and its methods, as it looks like in listing 11.8. Such an interface is known as a fragment interface and its purpose is to extend a repository with custom functionality, to be provided by a later implementation.

Listing 11.8 The `ItemRepositoryCustom` interface

Path:
Ch11/transactions5-
`springdata/src/main/java/com/manning/javapersistence/ch11/repositories/ItemRepositor`
`yCustom.java`

```

public interface ItemRepositoryCustom {
    void addItem(String name, LocalDate creationDate);
    void checkNameDuplicate(String name);
    void addLogs();
    void showLogs();
    void addItemNoRollback(String name, LocalDate creationDate);
}

```

Then, we'll create the `ItemRepository` interface, extending both `JpaRepository` and the previously declared `ItemRepositoryCustom` interface. Additionally, we'll declare the `findByName` method, following the Spring Data JPA naming conventions, as in listing 11.9.

Listing 11.9 The ItemRepository interface

Path:
Ch11/transactions5-
springdata/src/main/java/com/manning/javapersistence/ch11/repositories/ItemRepositor
y.java

```
public interface ItemRepository extends JpaRepository<Item, Long>,
    ItemRepositoryCustom {
    Optional<Item> findByName(String name);
}
```

We'll then create the `LogRepositoryCustom` interface and its methods, as it looks like in listing 11.10. Again, it is a fragment interface and its purpose is to extend a repository with custom functionality, to be provided by a later implementation.

Listing 11.10 The LogRepositoryCustom interface

Path:
Ch11/transactions5-
springdata/src/main/java/com/manning/javapersistence/ch11/repositories/LogRepository
Custom.java

```
public interface LogRepositoryCustom {
    void log(String message);
    void showLogs();
    void addSeparateLogsNotSupported();
    void addSeparateLogsSupports();
}
```

We'll now create the `LogRepository` interface, extending both `JpaRepository` and the previously declared `LogRepositoryCustom` interface, as in listing 11.11.

Listing 11.11 The LogRepository interface

Path:
Ch11/transactions5-
springdata/src/main/java/com/manning/javapersistence/ch11/repositories/LogRepository
.java

```
public interface LogRepository extends JpaRepository<Log, Integer>,
    LogRepositoryCustom {
```

We'll provide an implementation class for `ItemRepository`. The key part of this class name is the fact that it ends with `Impl`. But it is not connected to Spring Data and it only implements `ItemRepositoryCustom`. When injecting an `ItemRepository` bean, Spring Data will have to create a proxy class. It will detect that `ItemRepository` implements `ItemRepositoryCustom` and will look up a class called `ItemRepositoryCustomImpl` to act as a custom repository implementation. Consequently, the methods of the injected `ItemRepository` bean will have the same behavior as the methods of the `ItemRepositoryCustomImpl` class.

Listing 11.12 The ItemRepositoryImpl class

```

Path: Ch11/transactions5-
      springdata/src/main/java/com/manning/javapersistence/ch11/repositories/ItemRepository
      yImpl.java

public class ItemRepositoryImpl implements ItemRepositoryCustom {

    @Autowired                                     #A
    private ItemRepository itemRepository;          #A

    @Autowired                                     #A
    private LogRepository logRepository;            #A

    @Override
    @Transactional(propagation = Propagation.MANDATORY)           #B
    public void checkNameDuplicate(String name) {
        if (itemRepository.findAll().stream().map(item ->
            item.getName()).filter(n -> n.equals(name)).count() > 0) {   #C
            throw new DuplicateItemNameException("Item with name " + name + #C
                " already exists");                                         #C
        }
    }

    @Override
    @Transactional                                       #D
    public void addItem(String name, LocalDate creationDate) {
        logRepository.log("adding item with name " + name);
        checkNameDuplicate(name);
        itemRepository.save(new Item(name, creationDate));
    }

    @Override
    @Transactional(noRollbackFor = DuplicateItemNameException.class)  #E
    public void addItemNoRollback(String name, LocalDate creationDate) {
        logRepository.save(new Log(
            "adding log in method with no rollback for item " + name));
        checkNameDuplicate(name);
        itemRepository.save(new Item(name, creationDate));
    }

    @Override
    @Transactional                                       #D
    public void addLogs() {
        logRepository.addSeparateLogsNotSupported();
    }

    @Override
    @Transactional                                       #D
    public void showLogs() {
        logRepository.showLogs();
    }
}

```

#A Autowiring an ItemRepository and a LogRepository bean.

#B MANDATORY propagation - Spring Data will check if a transaction is already in progress and will continue with it.
Otherwise, an exception will be thrown.

#C Throw a DuplicateItemNameException if an Item with the given name already exists.

#D Default propagation is REQUIRED.

#E No rollback the transaction in case of a DuplicateItemNameException.

We'll provide an implementation class for `LogRepository`. The key part of this class name is the fact that it ends with `Impl`. But it is not connected to Spring Data and it only implements `LogRepositoryCustom`. When injecting a `LogRepository` bean, Spring Data will have to create a proxy class. It will detect that `LogRepository` implements `LogRepositoryCustom` and will look up a class called `LogRepositoryCustomImpl` to act as a custom repository implementation. Consequently, the methods of the injected `LogRepository` bean will have the same behavior as the methods of the `LogRepositoryCustomImpl` class.

Listing 11.13 The LogRepositoryImpl class

```
Path:  
Ch11/transactions5-  
    springdata/src/main/java/com/manning/javapersistence/ch11/repositories/LogRepository  
    Impl.java

public class LogRepositoryImpl implements LogRepositoryCustom {  
    @Autowired  
    private LogRepository logRepository; #A  
  
    @Override  
    @Transactional(propagation = Propagation.REQUIRES_NEW) #B  
    public void log(String message) {  
        logRepository.save(new Log(message)); #C  
    }  
  
    @Transactional(propagation = Propagation.NOT_SUPPORTED) #D  
    public void addSeparateLogsNotSupported() {  
        logRepository.save(new Log("check from not supported 1"));  
        if (true) throw new RuntimeException();  
        logRepository.save(new Log("check from not supported 2"));  
    }  
  
    @Transactional(propagation = Propagation.SUPPORTS) #E  
    public void addSeparateLogsSupports() {  
        logRepository.save(new Log("check from supports 1"));  
        if (true) throw new RuntimeException();  
        logRepository.save(new Log("check from supports 2"));  
    }  
  
    @Transactional(propagation = Propagation.NEVER) #F  
    public void showLogs() {  
        System.out.println("Current log:");  
        logRepository.findAll().forEach(System.out::println);  
    }  
}
```

#A Autowiring a `LogRepository` bean.

```

#B REQUIRES_NEW propagation. Spring Data will execute the logging in a separate transaction, independent of the
eventual transaction of the method that called log.
#C The log method will save a message to the repository.
#D NOT_SUPPORTED propagation. If a transaction is in progress, it will be suspended and a non-transactional
execution will continue. Otherwise, the execution will simply continue.
#E SUPPORTED propagation. If a transaction is in progress, the execution will continue within that transaction.
Otherwise, no transaction will be created.
#F NEVER propagation. If a transaction is in progress, an IllegalStateException will be thrown.
Otherwise, the execution will simply continue.

```

We'll write now a series of tests to verify the behavior of the transactional methods we have just written.

Listing 11.14 The TransactionPropagationTest class

```

Path:
Ch11/transactions5-springdata/src/test/java/com/manning/javapersistence/ch11/concurrency/
TransactionPropagationTest.java

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {SpringDataConfiguration.class})
public class TransactionPropagationTest {

    @Autowired
    private ItemRepository itemRepository;                                #A
    @Autowired
    private LogRepository logRepository;                                  #A

    @BeforeEach
    public void clean() {                                                 #B
        itemRepository.deleteAll();
        logRepository.deleteAll();
    }

    @Test
    public void notSupported() {
        assertAll(
            () -> assertThrows(RuntimeException.class, () ->           #C
                    itemRepository.addLogs()),
            () -> assertEquals(1, logRepository.findAll().size()),          #D
            () -> assertEquals("check from not supported 1",             #D
                    logRepository.findAll().get(0).getMessage())
        );
        logRepository.showLogs();                                         #E
    }

    @Test
    public void supports() {
        assertAll(
            () -> assertThrows(RuntimeException.class, () ->           #F
                    logRepository.addSeparateLogsSupports()),
            () -> assertEquals(1, logRepository.findAll().size()),          #G
            () -> assertEquals("check from supports 1",                  #G
                    logRepository.findAll().get(0).getMessage())
        );
        logRepository.showLogs();                                         #H
    }
}

```

```

@Test
public void mandatory() {
    IllegalTransactionStateException ex =
        assertThrows(IllegalTransactionStateException.class,
            () -> itemRepository.checkNameDuplicate("Item1"));
    assertEquals("No existing transaction found for transaction marked
        with propagation 'mandatory'", ex.getMessage()); #I
}

@Test
public void never() {
    itemRepository.addItem("Item1", LocalDate.of(2022, 5, 1)); #J
    logRepository.showLogs(); #J

    IllegalTransactionStateException ex =
        assertThrows(IllegalTransactionStateException.class,
            () -> itemRepository.showLogs()); #K
    assertEquals(
        "Existing transaction found for transaction marked with propagation
        'never'", ex.getMessage()); #K
}

@Test
public void requiresNew() {
    itemRepository.addItem("Item1", LocalDate.of(2022, 5, 1));
    itemRepository.addItem("Item2", LocalDate.of(2022, 3, 1));
    itemRepository.addItem("Item3", LocalDate.of(2022, 1, 1));

    DuplicateItemNameException ex =
        assertThrows(DuplicateItemNameException.class, () ->
            itemRepository.addItem("Item2", LocalDate.of(2016, 3, 1))); #L
    assertAll(
        () -> assertEquals("Item with name Item2 already exists",
            ex.getMessage()), #M
        () -> assertEquals(4, logRepository.findAll().size()), #N
        () -> assertEquals(3, itemRepository.findAll().size()) #N
    );

    System.out.println("Logs: ");
    logRepository.findAll().forEach(System.out::println);

    System.out.println("List of added items: ");
    itemRepository.findAll().forEach(System.out::println);
}

@Test
public void noRollback() {
    itemRepository.addItemNoRollback("Item1", LocalDate.of(2022, 5, 1));
    itemRepository.addItemNoRollback("Item2", LocalDate.of(2022, 3, 1));
    itemRepository.addItemNoRollback("Item3", LocalDate.of(2022, 1, 1));

    DuplicateItemNameException ex =
        assertThrows(DuplicateItemNameException.class,
            () -> itemRepository.addItem("Item2",
                LocalDate.of(2016, 3, 1))); #O
    assertAll(
        () -> assertEquals("Item with name Item2 already exists",
            ex.getMessage()), #P
        () -> assertEquals(3, itemRepository.findAll().size()) #P
)
}

```

```

        () -> assertEquals(4, logRepository.findAll().size()),      #Q
        () -> assertEquals(3, itemRepository.findAll().size())      #Q
    );

    System.out.println("Logs: ");
    logRepository.findAll().forEach(System.out::println);

    System.out.println("List of added items: ");
    itemRepository.findAll().forEach(System.out::println);
}
}

```

#A Autowiring an `ItemRepository` and a `LogRepository` bean.
#B Before the execution of each test, all `Item` entities and all `Log` entities are removed from the repositories.
#C The `addLogs` method is starting a transaction, but it calls the `addSeparateLogsNotSupported` method which will suspend it before explicitly throwing an exception.
#D Before an exception was thrown, the `logRepository` was however able to save one message.
#E The `showLog` method will display one message in a non-transactional way.
#F The `addSeparateLogsSupports` method will explicitly throw an exception.
#G Before an exception was thrown, the `logRepository` was however able to save one message.
#H The `showLog` method will display one message in a non-transactional way.
#I The method `checkNameDuplicate` can be executed only in a transaction, so an `IllegalTransactionStateException` will be thrown when calling it without a transaction. We also check the message from the exception.
#J After adding an `Item` to the repository, it is safe to call the `showLogs` method from `LogRepository` without a transaction.
#K However, it is prohibited to call the `showLogs` method from `LogRepository` within a transaction, as the calling method `showLogs` from `ItemRepository` is transactional.
#L Trying to insert a duplicate `Item` in the repository will throw a `DuplicateItemNameException`.
#M However, a log message is persisted in the logs even after exception, because it was added in a separate transaction.
#N The repository will contain 4 `Log` messages (one for each attempt to insert an `Item`, successful or unsuccessful), but only 3 `Items` (the duplicate `Item` was rejected).
#O Trying to insert a duplicate `Item` in the repository will throw a `DuplicateItemNameException`.
#P However, a log message is persisted in the logs even after exception, because the transaction was not rolled back. The `addItemNoRollback` method from `ItemRepository` does not rollback for `DuplicateItemNameException`.
#Q The repository will contain 4 `Log` messages (one for each attempt to insert an `Item`, successful or unsuccessful), but only 3 `Items` (the duplicate `Item` was rejected).

11.5 Summary

- You learned to use transactions, concurrency, isolation, and locking.
- Hibernate relies on a database's concurrency-control mechanism but provides better isolation guarantees in a transaction, thanks to automatic versioning and the persistence context cache.
- We analyzed how to set transaction boundaries programmatically and handle exceptions.
- We examined optimistic concurrency control and explicit pessimistic locking.
- We demonstrated how to work with auto-commit mode and an unsynchronized persistence context outside of a transaction, and how to queue modification.

- We examined how to work with transactions with Spring and Spring Data, how to define and configure such a transaction using various properties.
- We used the Spring and Spring Data transactional capabilities to develop an application that manages items and log messages.

12

Fetch plans, strategies, and profiles

This chapter covers:

- Working with lazy and eager loading
- Applying fetch plans, strategies, and profiles
- Optimizing SQL execution

In this chapter, we explore Hibernate's solution for the fundamental ORM problem of navigation, as introduced in section 1.2.5. We demonstrate how to retrieve data from the database and how to optimize this loading.

Hibernate provides the following ways to get data out of the database and into memory:

- Retrieving an entity instance by identifier is the most convenient method when the unique identifier value of an entity instance is known: for example, `entityManager.find(Item.class, 123)`.
- We can navigate the entity graph, starting from an already-loaded entity instance, by accessing the associated instances through property accessor methods such as `someItem.getSeller().getAddress().getCity()`, and so on. Elements of mapped collections are also loaded on demand when we start iterating through a collection. Hibernate automatically loads nodes of the graph if the persistence context is still open. What and how data is loaded when we call accessors and iterate through collections is the focus of this chapter.
- We can use the Jakarta Persistence Query Language (JPQL), a full object-oriented query language based on strings such as `select i from Item i where i.id = ?`.
- The `CriteriaQuery` interface provides a type-safe and object-oriented way to perform queries without string manipulation.
- We can write native SQL queries, call stored procedures, and let Hibernate take care of mapping the JDBC result sets to instances of the domain model classes.

In the JPA applications, we'll use a combination of these techniques. By now you should be familiar with the basic Jakarta Persistence API for retrieval by identifier. We keep our JPQL and CriteriaQuery examples as simple as possible, and you won't need the SQL query-mapping features.

Major new features in JPA 2

We can manually check the initialization state of an entity or an entity property with the new `PersistenceUtil` static helper class.

We can create standardized declarative fetch plans with the new `EntityGraph` API.

This chapter analyzes what happens behind the scenes when we navigate the graph of the domain model and Hibernate retrieves data on demand. In all the examples, we interpret the SQL executed by Hibernate in a comment right immediately after the operation that triggered the SQL execution.

What Hibernate loads depends on the *fetch plan*: we define the (sub)graph of the network of objects that should be loaded. Then we pick the right *fetch strategy*, defining *how* the data should be loaded. We can store the selection of plan and strategy as a *fetch profile* and reuse it.

Defining fetch plans and *what* data should be loaded by Hibernate relies on two fundamental techniques: *lazy* and *eager* loading of nodes in the network of objects.

12.1 Lazy and eager loading

At some point, we must decide what data should be loaded into memory from the database. When we execute `entityManager.find(Item.class, 123)`, what is available in memory and loaded into the persistence context? What happens if we use `EntityManager#getReference()` instead?

In the domain-model mapping, we define the global *default fetch plan*, with the `FetchType.LAZY` and `FetchType.EAGER` options on associations and collections. This plan is the default setting for all operations involving the persistent domain model classes. It's always active when we load an entity instance by identifier and when we navigate the entity graph by following associations and iterating through persistent collections.

Our recommended strategy is a *lazy* default fetch plan for all entities and collections. If we map all of the associations and collections with `FetchType.LAZY`, Hibernate will only load the data we're accessing at this time. While we navigate the graph of the domain model instances, Hibernate will load data on demand, bit by bit. We then override this behavior on a per-case basis when necessary.

To implement lazy loading, Hibernate relies on runtime-generated entity placeholders called *proxies* and on *smart wrappers* for collections.

12.1.1 Understanding entity proxies

Consider the `getReference()` method of the `EntityManager` API. In section 10.2.3, we had a first look at this operation and how it may return a proxy. Let's further explore this important feature and find out how proxies work. To be able to execute the examples from the source code, you need first to run the Ch12.sql script.

Path:

```
Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections
.java
```

```
Item item = em.getReference(Item.class, ITEM_ID); #A
assertEquals(ITEM_ID, item.getId()); #B
```

#A There is no database hitting, meaning there is no `SELECT`.

#B Calling the identifier getter (no field access!) doesn't trigger initialization.

This code doesn't execute any SQL against the database. All Hibernate does is create an `Item` proxy: it looks (and smells) like the real thing, but it's only a placeholder. In the persistence context, in memory, we now have this proxy available in persistent state, as shown in figure 12.1.

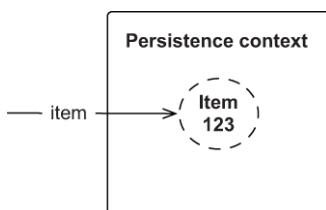


Figure 12.1 The persistence context, under the control of Hibernate, contains an `Item` proxy.

The proxy is an instance of a runtime-generated subclass of `Item`, carrying the identifier value of the entity instance it represents. This is why Hibernate (in line with JPA) requires that entity classes have at least a public or protected no-argument constructor (the class may have other constructors, too). The entity class and its methods must not be final; otherwise, Hibernate can't produce a proxy. Note that the JPA specification doesn't mention proxies; it's up to the JPA provider how lazy loading is implemented.

If we call any method on the proxy that isn't the "identifier getter", we trigger initialization of the proxy and hit the database. If we call `item.getName()`, the SQL `SELECT` to load the `Item` will be executed. The previous example called `item.getId()` without triggering initialization because `getId()` is the identifier getter method in the given mapping; the `getId()` method was annotated with `@Id`. If `@Id` was on a field, then calling `getId()`, just like calling any other method, would initialize the proxy! (Remember that we usually prefer mappings and access on fields, because this allows more freedom when designing accessor methods; see section 3.2.3. It's up to you whether calling `getId()` without initializing a proxy is more important.)

With proxies, be careful how you compare classes. Because Hibernate generates the proxy class, it has a funny-looking name, and it is *not* equal to `Item.class`:

```
Path: Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections.java

assertEquals(Item.class, item.getClass());                                     #A
assertEquals(
    Item.class,
    HibernateProxyHelper.getClassWithoutInitializingProxy(item)
);

#A The class is runtime generated and named something like Item$HibernateProxy$BLsrPly8
```

If we really must get the actual type represented by a proxy, we use the `HibernateProxyHelper`.

JPA provides `PersistenceUtil`, which we can use to check the initialization state of an entity or any of its attributes:

```
Path: Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections.java

PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertFalse(persistenceUtil.isLoaded(item));
assertFalse(persistenceUtil.isLoaded(item, "seller"));
assertFalse(Hibernate.isInitialized(item));
// assertFalse(Hibernate.isInitialized(item.getSeller()));                      #A
```

#A Executing this line of code would effectively trigger the initialization of the item.

The static `isLoaded()` method also accepts the name of a property of the given entity (proxy) instance, checking its initialization state. Hibernate offers an alternative API with `Hibernate.isInitialized()`. If we call `item.getSeller()`, though, the `item` proxy is initialized first!

Hibernate also offers a utility method for quick-and-dirty initialization of proxies:

```
Path: Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections.java

Hibernate.initialize(item);                                                 #A
// select * from ITEM where ID = ?
assertFalse(Hibernate.isInitialized(item.getSeller()));                      #B
Hibernate.initialize(item.getSeller());                                       #C
// select * from USERS where ID = ?
```

#A The first call hits the database and loads the `Item` data, populating the proxy with the item's name, price, and so on.

#B Make sure the default EAGER of `@ManyToOne` has been overridden with LAZY. That is why the seller of the item is not yet initialized.

#C Initializing the seller of the item, we hit the database and load the User data.

The seller of the `Item` is a `@ManyToOne` association mapped with `FetchType.LAZY`, so Hibernate creates a `User` proxy when the `Item` is loaded. We can check the seller proxy

state and load it manually, just like the `Item`. Remember that the JPA default for `@ManyToOne` is `FetchType.EAGER`! We usually want to override this to get a lazy default fetch plan, as first demonstrated in section 8.3.1 and again here:

```
Path: Ch12/proxy/src/main/java/com/manning/javapersistence/ch12/proxy/Item.java

@Entity
public class Item {
    @ManyToOne(fetch = FetchType.LAZY)
    public User getSeller() {
        return seller;
    }
    // ...
}
```

With such a lazy fetch plan, we might run into a `LazyInitializationException`. Consider the following code:

```
Path: Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections.java

Item item = em.find(Item.class, ITEM_ID);                                #A
// select * from ITEM where ID = ?
em.detach(item);                                                        #B
em.detach(item.getSeller());
// em.close();
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();      #C
assertTrue(persistenceUtil.isLoaded(item));
assertFalse(persistenceUtil.isLoaded(item, "seller"));
assertEquals(USER_ID, item.getSeller().getId());                           #D
//assertNotNull(item.getSeller().getUsername());                         #E
```

#A An `Item` entity instance is loaded in the persistence context. Its `seller` isn't initialized: it's a `User` proxy.

#B We can manually detach the data from the persistence context or close the persistence context and detach everything.

#C The static `PersistenceUtil` helper works without a persistence context. We can check at any time whether the data we want to access has been loaded.

#D In detached state, we can call the identifier getter method of the `User` proxy.

#E Calling any other method on the proxy, such as `getUsername()`, will throw a

`LazyInitializationException`. Data can only be loaded on demand while the persistence context manages the proxy, not in detached state.

How does lazy loading of one-to-one associations work?

Lazy loading for one-to-one entity associations is sometimes confusing for new Hibernate users. If we consider one-to-one associations based on shared primary keys (see section 9.1.1), an association can be proxied only if it's optional=false. For example, an `Address` always has a reference to a `User`. If this association is nullable and optional, Hibernate must first hit the database to find out whether it should apply a proxy or a null—and the purpose of lazy loading is to not hit the database at all.

Hibernate proxies are useful beyond simple lazy loading. For example, we can store a new `Bid` without loading any data into memory.

Path:

`Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections.java`

```
Item item = em.getReference(Item.class, ITEM_ID);
User user = em.getReference(User.class, USER_ID);
Bid newBid = new Bid(new BigDecimal("99.00"));
newBid.setItem(item);
newBid.setBidder(user);
em.persist(newBid);                                     #A
// insert into BID values (?, ?, ?, ...)
```

#A There is no SQL SELECT in this procedure, only one INSERT.

The first two calls produce proxies of `Item` and `User`, respectively. Then the `item` and `bidder` association properties of the transient `Bid` are set with the proxies. The `persist()` call queues one SQL `INSERT` when the persistence context is flushed, and no `SELECT` is necessary to create the new row in the `BID` table. All (foreign) key values are available as identifier values of the `Item` and `User` proxy.

Runtime proxy generation as provided by Hibernate is an excellent choice for transparent lazy loading. The domain model classes don't have to implement any special (super)type, as some older ORM solutions would require. No code generation or post-processing of bytecode is needed either, simplifying the build procedure. But we should be aware of some potentially negative aspects:

- Cases where runtime proxies aren't completely transparent are polymorphic associations that are tested with `instanceof`, a problem demonstrated in section 7.8.1.
- With entity proxies, we have to be careful not to access fields directly when writing custom `equals()` and `hashCode()` methods, as analyzed in section 10.3.2.
- Proxies can only be used to lazy-load entity associations. They can't be used to lazy load individual basic properties or embedded components, such as `Item#description` or `User#homeAddress`. If we set the `@Basic(fetch = FetchType.LAZY)` hint on such a property, Hibernate ignores it; the value is eagerly loaded when the owning entity instance is loaded. Optimizing at the level of individual columns selected in SQL is unnecessary if we aren't working with (a) a significant number of optional/nullable columns or (b) columns containing large values that have to be retrieved on demand because of the physical limitations of the system. Large values are best represented with locator objects (LOBs) instead; they provide lazy loading by definition (see the section "Binary and large value types" in chapter 6).

Proxies enable lazy loading of entity instances. For persistent collections, Hibernate has a slightly different approach.

12.1.2 Lazy persistent collections

We map persistent collections with either `@ElementCollection` for a collection of elements of basic or embeddable type or with `@OneToMany` and `@ManyToMany` for many-valued entity associations. These collections are, unlike `@ManyToOne`, lazy-loaded by default. We don't have to specify the `FetchType.LAZY` option on the mapping.

The `lazy` bids *one-to-many* collection is only loaded on demand when accessed and needed:

```
Path: Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections.java

Item item = em.find(Item.class, ITEM_ID);                                #A
// select * from ITEM where ID = ?
Set<Bid> bids = item.getBids();                                         #B
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
assertFalse(persistenceUtil.isLoaded(item, "bids"));
assertTrue(Set.class.isAssignableFrom(bids.getClass()));                  #C
assertNotEquals(HashSet.class, bids.getClass());                          #D
assertEquals(org.hibernate.collection.internal.PersistentSet.class, bids.getClass()); #E

#A The find() operation loads the Item entity instance into the persistence context, as we can see in figure 12.2.
#B The Item instance has a reference to an uninitialized Set of bids. It also has a reference to an
     uninitialized User proxy: the seller.
#C The bids field is a Set.
#D However, the bids field is not a HashSet.
#E The bids field is a Hibernate proxy class.
```

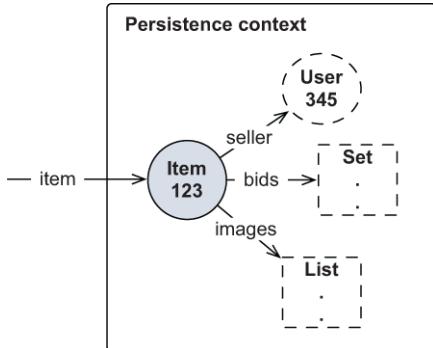


Figure 12.2 Proxies and collection wrappers are the boundary of the loaded graph under Hibernate control

Hibernate implements lazy loading (and dirty checking) of collections with its own special implementations called *collection wrappers*. Although the `bids` certainly look like a `Set`, while we weren't looking, Hibernate replaced the implementation with an `org.hibernate.collection.internal.PersistentSet`. It's not a `HashSet`, but it has the same behavior. That's why it's so important to program with interfaces in the domain model and only rely on `Set` and not `HashSet`. Lists and maps work the same way.

These special collections can detect when we access them and load their data at that time. As soon as we start iterating through the `bids`, the collection and all bids made for the item are loaded:

Path:

```
Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections
.java
```

```
Bid firstBid = bids.iterator().next();
// select * from BID where ITEM_ID = ?
// Alternative: Hibernate.initialize(bids);
```

Alternatively, just as for entity proxies, we can call the static utility method `Hibernate.initialize()` to load a collection. It will be completely loaded; we can't say "only load the first two bids," for example. For this, we'd have to write a query.

For convenience, so we don't have to write many trivial queries, Hibernate offers a proprietary setting on collection mappings:

Path: Ch12/proxy/src/main/java/com/manning/javapersistence/ch12/proxy/Item.java

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.LazyCollection(
        org.hibernate.annotations.LazyCollectionOption.EXTRA
    )
    public Set<Bid> getBids() {
        return bids;
    }
    // ...
}
```

With `LazyCollectionOption.EXTRA`, the collection supports operations that don't trigger initialization. For example, we could ask for the collection's size:

Path:

Ch12/proxy/src/test/java/com/manning/javapersistence/ch12/proxy/LazyProxyCollections.java

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?
assertEquals(3, item.getBids().size());
// select count(b) from BID b where b.ITEM_ID = ?
```

The `size()` operation triggers a `SELECT COUNT()` SQL query but doesn't load the `bids` into memory. On all extra lazy collections, similar queries are executed for the `isEmpty()` and `contains()` operations. An extra lazy `Set` checks for duplicates with a simple query when we call `add()`. An extra lazy `List` only loads one element if we call `get(index)`. For `Map`, extra lazy operations are `containsKey()` and `containsValue()`.

12.1.3 Eager loading of associations and collections

We've recommended a lazy default fetch plan, with `FetchType.LAZY` on all the association and collection mappings. Sometimes, although not often, we want the opposite: to specify that a particular entity association or collection should always be loaded. We want the guarantee that this data is available in memory without an additional database hit.

More importantly, we want a guarantee that, for example, we can access the `seller` of an `Item` once the `Item` instance is in detached state. When the persistence context is closed, lazy loading is no longer available. If `seller` were an uninitialized proxy, we'd get a `LazyInitializationException` when we accessed it in detached state. For data to be available in detached state, we need to either load it manually while the persistence context is still open or, if we always want it loaded, change the fetch plan to be eager instead of lazy.

Let's assume that we always require loading of the `seller` and the `bids` of an `Item`:

Path: Ch12/eagerjoin/src/main/java/com/manning/javapersistence/ch12/eagerjoin/Item.java

```
@Entity
public class Item {
    @ManyToOne(fetch = FetchType.EAGER) #A
    private User seller;
```

```

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)           #B
    private Set<Bid> bids = new HashSet<>();
    // ...
}

```

#A `FetchType.EAGER` is the default for entity instances.

#B Generally, `FetchType.EAGER` on a collection is not recommended. Unlike `FetchType.LAZY`, which is a hint the JPA provider can ignore, a `FetchType.EAGER` is a hard requirement. The provider has to guarantee that the data is loaded and available in detached state; it can't ignore the setting.

Consider the collection mapping: is it really a good idea to say, “Whenever an item is loaded into memory, load the bids of the item right away, too”? Even if we only want to display the item’s name or find out when the auction ends, all bids will be loaded into memory. Always eager-loading collections, with `FetchType.EAGER` as the default fetch plan in the mapping, usually isn’t a great strategy. We’ll also analyze the *Cartesian product problem* that appears if we eagerly load several collections, which we demonstrate later in this chapter. It’s best if we leave collections as the default `FetchType.LAZY`.

If we now `find()` an `Item` (or force the initialization of an `Item` proxy), both the `seller` and all the `bids` are loaded as persistent instances into the persistence context:

```

Path: Ch12/eagerjoin/src/test/java/com/manning/javapersistence/ch12/eagerjoin/EagerJoin.java
va

Item item = em.find(Item.class, ITEM_ID);
// select i.*, u.*, b.*
// from ITEM i
//   left outer join USERS u on u.ID = i.SELLER_ID
//   left outer join BID b on b.ITEM_ID = i.ID
// where i.ID = ?
em.detach(item);                                #A
assertEquals(3, item.getBids().size());          #B
assertNotNull(item.getBids().iterator().next().getAmount());
assertEquals("johndoe", item.getSeller().getUsername()); #C

```

#A When calling `detach()`, the fetching is done. There will be no more lazy loading.

#B In detached state, the `bids` collection is available. So, we can check its size.

#C In detached state, the `seller` is available. So, we can check his name.

For the `find()`, Hibernate executes a single SQL `SELECT` and `JOINS` three tables to retrieve the data. We can see the contents of the persistence context in figure 12.3. Note how the boundaries of the loaded graph are represented: each `Bid` has a reference to an uninitialized `User` proxy, the `bidder`. If we now detach the `Item`, we access the loaded `seller` and `bids` without causing a `LazyInitializationException`. If we try to access one of the `bidder` proxies, we’ll get an exception!

In the following examples, we assume that the domain model has a lazy default fetch plan. Hibernate will only load the data we explicitly request and the associations and collections we access.

Next, we investigate *how* data should be loaded when we find an entity instance by identity and when we navigate the network, using the pointers of the mapped associations

and collections. We're interested in what SQL is executed and finding the ideal *fetch strategy*.

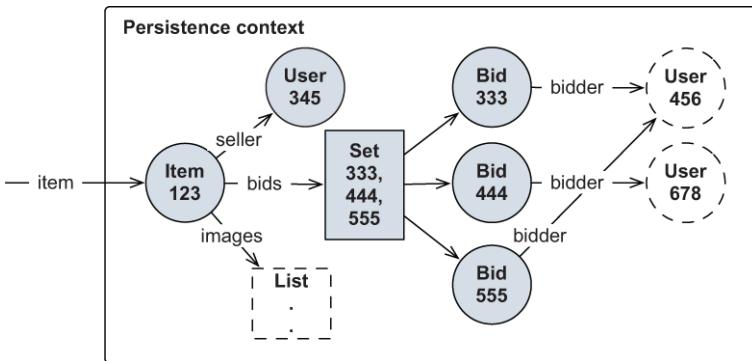


Figure 12.3 The seller and the bids of an `Item` are loaded in the Hibernate persistence context

12.2 Selecting a fetch strategy

Hibernate executes SQL `SELECT` statements to load data into memory. If we load an entity instance, one or more `SELECT`(s) are executed, depending on the number of tables involved and the *fetching strategy* we've applied. Our goal is to minimize the number of SQL statements and to simplify the SQL statements so that querying can be as efficient as possible.

Consider our recommended fetch plan from earlier in this chapter: every association and collection should be loaded on demand, lazily. This default fetch plan will most likely result in too many SQL statements, each loading only one small piece of data. This will lead to *n+1 selects problems*, and we analyze this issue first. The alternative fetch plan, using eager loading, will result in fewer SQL statements, because larger chunks of data are loaded into memory with each SQL query. We might then see the *Cartesian product problem*, as SQL result sets become too large.

We need to find the middle ground between these two extremes: the ideal fetching strategy for each procedure and use case in our application. Like fetch plans, we can set a global fetching strategy in the mappings: the default setting that is always active. Then, for a particular procedure, we might override the default fetching strategy with a custom JPQL, `CriteriaQuery`, or even SQL query.

First, let's investigate the fundamental problems, starting with the *n+1 selects* issue.

12.2.1 The n+1 selects problem

This problem is easy to understand with some example code. Let's assume that we mapped a lazy fetch plan, so everything is loaded on demand. The following example code checks whether the `seller` of each `Item` has a `username`:

```
Path: Ch12/nplusoneselects/src/test/java/com/manning/javapersistence/ch12/nplusoneselects/
NPlusOneSelects.java

List<Item> items =
    em.createQuery("select i from Item i").getResultList();
// select * from ITEM
for (Item item : items) {
    assertNotNull(item.getSeller().getUsername()); #A
    // select * from USERS where ID = ?
}
```

#A Whenever we access a seller, each of these ones must be loaded with an additional SELECT.

We see one SQL SELECT to load the `Item` entity instances. Then, while we iterate through all the `items`, retrieving each `User` requires an additional SELECT. This amounts to one query for the `Item` plus n queries depending on how many `items` we have and whether a particular `User` is selling more than one `Item`. Obviously, this is a very inefficient strategy if we know we'll access the `seller` of each `Item`.

We can see the same issue with lazily loaded collections. The following example checks whether each `Item` has some `bids`:

```
Path: Ch12/nplusoneselects/src/test/java/com/manning/javapersistence/ch12/nplusoneselects/
NPlusOneSelects.java

List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM
for (Item item : items) {
    assertTrue(item.getBids().size() > 0); #A
    // select * from BID where ITEM_ID = ?
}
```

#A Each `bids` collection has to be loaded with an additional SELECT.

Again, if we know we'll access each `bids` collection, loading only one at a time is inefficient. If we have 100 items, we'll execute 101 SQL queries!

With what we know so far, we might be tempted to change the default fetch plan in the mappings and put a `FetchType.EAGER` on the `seller` or `bids` associations. But doing so can lead to our next topic: the *Cartesian product* problem.

12.2.2 The Cartesian product problem

If we look at the domain and data model and say, "Every time I need an `Item`, I also need the `seller` of that `Item`", we can map the association with `FetchType.EAGER` instead of a lazy fetch plan. We want a guarantee that whenever an `Item` is loaded, the `seller` will be loaded right away—we want that data to be available when the `Item` is detached and the persistence context is closed:

```
Path: Ch12/cartesianproduct/src/main/java/com/manning/javapersistence/ch12/cartesianproduct/
Item.java
```

```
@Entity
public class Item {
    @ManyToOne(fetch = FetchType.EAGER)
    private User seller;
    // ...
}
```

To implement the eager fetch plan, Hibernate uses an SQL `JOIN` operation to load an `Item` and a `User` instance in one `SELECT`:

```
item = em.find(Item.class, ITEM_ID);
// select i.*, u.*
// from ITEM i
// left outer join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

The result set contains one row with data from the `ITEM` table combined with data from the `USERS` table, as shown in figure 12.4.

i.ID	i.NAME	iSELLER_ID	...	u.ID	u.USERNAME	...
1	One	2	...	2	johndoe	...

Figure 12.4 Hibernate joins two tables to eagerly fetch associated rows.

Eager fetching with the default `JOIN` strategy isn't problematic for `@ManyToOne` and `@OneToOne` associations. We can eagerly load, with one SQL query and `JOINS`, an `Item`, its `seller`, the `User's Address`, the `City` they live in, and so on. Even if we map all these associations with `FetchType.EAGER`, the result set will have only one row. Now, Hibernate has to stop following the `FetchType.EAGER` plan at *some* point. The number of tables joined depends on the global `hibernate.max_fetch_depth` configuration property. By default, no limit is set. Reasonable values are small, usually between 1 and 5. We may even disable `JOIN` fetching of `@ManyToOne` and `@OneToOne` associations by setting the property to 0. If Hibernate reaches the limit, it will still eagerly load the data according to the fetch plan, but with additional `SELECT` statements. (Note that some database dialects may preset this property: for example, `MySQLDialect` sets it to 2.)

Eagerly loading collections with `JOINS`, on the other hand, can lead to serious performance issues. If we also switched to `FetchType.EAGER` for the `bids` and `images` collections, we'd run into the *Cartesian product problem*.

This issue appears when we eagerly load two collections with one SQL query and a `JOIN` operation. First, let's create such a fetch plan and then look at the SQL problem:

```
Path: Ch12/cartesianproduct/src/main/java/com/manning/javapersistence/ch12/cartesianproduct/Item.java

@Entity
public class Item {
    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
```

```

private Set<Bid> bids = new HashSet<>();
@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable(name = "IMAGE")
@Column(name = "FILENAME")
private Set<String> images = new HashSet<String>();
// ...
}

```

It doesn't matter whether both collections are @OneToMany, @ManyToMany, or @ElementCollection. Eager fetching more than one collection at once with the SQL JOIN operator is the fundamental issue, no matter what the collection content is. If we load an Item, Hibernate executes the problematic SQL statement:

```

Path: Ch12/cartesianproduct/src/test/java/com/manning/javapersistence/ch12/cartesianproduct/CartesianProduct.java

Item item = em.find(Item.class, ITEM_ID);
// select i.*, b.*, img.*
// from ITEM i
// left outer join BID b on b.ITEM_ID = i.ID
// left outer join IMAGE img on img.ITEM_ID = i.ID
// where i.ID = ?
em.detach(item);
assertEquals(3, item.getImages().size());
assertEquals(3, item.getBids().size());

```

As we can see, Hibernate obeyed the eager fetch plan, and we can access the bids and images collections in detached state. The problem is *how* they were loaded, with an SQL JOIN that results in a product. Let's look at the result set in figure 12.5.

i.ID	i.NAME	...	b.ID	b.AMOUNT	img.FILENAME
1	One	...	1	99.00	foo.jpg
1	One	...	1	99.00	bar.jpg
1	One	...	1	99.00	baz.jpg
1	One	...	2	100.00	foo.jpg
1	One	...	2	100.00	bar.jpg
1	One	...	2	100.00	baz.jpg
1	One	...	3	101.00	foo.jpg
1	One	...	3	101.00	bar.jpg
1	One	...	3	101.00	baz.jpg

Figure 12.5 A product is the result of two joins with many rows.

This result set contains many redundant data items, and only the shaded cells are relevant for Hibernate. The Item has three bids and three images. The size of the product depends on the size of the collections we're retrieving: three times three is nine rows total.

Now imagine that we have an `Item` with 50 `bids` and 5 `images`—we'll see a result set with possibly 250 rows! We can create even larger SQL products when we write our own queries with JPQL or `CriteriaQuery`: imagine what happens if we load 500 items and eager fetch dozens of bids and images with `JOINS`.

Considerable processing time and memory are required on the database server to create such results, which then must be transferred across the network. If you're hoping that the JDBC driver will compress the data on the wire somehow, you're probably expecting too much from database vendors. Hibernate immediately removes all duplicates when it marshals the result set into persistent instances and collections; information in cells that aren't shaded in figure 12.5 will be ignored. Obviously, we can't remove these duplicates at the SQL level; the SQL `DISTINCT` operator doesn't help here.

Instead of one SQL query with an extremely large result, three separate queries would be faster to retrieve an entity instance and two collections at the same time. Next, we focus on this kind of optimization and how we find and implement the best fetch strategy. We start again with a default lazy fetch plan and try to solve the *n+1 selects* problem first.

12.2.3 Prefetching data in batches

If Hibernate fetches every entity association and collection only on demand, many additional SQL `SELECT` statements may be necessary to complete a particular procedure. As before, consider a routine that checks whether the `seller` of each `Item` has a `username`. With lazy loading, this would require one `SELECT` to get all `Item` instances and n more `SELECTS` to initialize the `seller` proxy of each `Item`.

Hibernate offers algorithms that can prefetch data. The first algorithm we analyze is *batch fetching*, and it works as follows: if Hibernate must initialize one `User` proxy, go ahead and initialize several with the same `SELECT`. In other words, if we already know that there are several `Item` instances in the persistence context and that they all have a proxy applied to their `seller` association, we may as well initialize several proxies instead of just one if we make the round trip to the database.

Let's see how this works. First, enable batch fetching of `User` instances with a proprietary Hibernate annotation:

```
Path: Ch12/batch/src/main/java/com/manning/javapersistence/ch12/batch/User.java

@Entity
@org.hibernate.annotations.BatchSize(size = 10)
@Table(name = "USERS")
public class User {
    // ...
}
```

This setting tells Hibernate that it may load up to 10 `User` proxies if one has to be loaded, all with the same `SELECT`. Batch fetching is often called a *blind-guess optimization* because we don't know how many uninitialized `User` proxies may be in a particular persistence context. We can't say for sure that 10 is an ideal value—it's a guess. We know that instead of $n+1$ SQL queries, we'll now see $n+1/10$ queries, a significant reduction. Reasonable values are

usually small because we don't want to load too much data into memory either, especially if we aren't sure we'll need it.

This is the optimized procedure, which checks the `username` of each `seller`:

```
Path: Ch12/batch/src/test/java/com/manning/javapersistence/ch12/batch/Batch.java

List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM
for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
    // select * from USERS where ID in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
}
```

Note the SQL query that Hibernate executes while we iterate through the `items`. When we call `item.getSeller().getUserName()` for the first time, Hibernate must initialize the first User proxy. Instead of only loading a single row from the `USERS` table, Hibernate retrieves several rows, and up to 10 User instances are loaded. Once we access the eleventh `seller`, another 10 are loaded in one batch, and so on, until the persistence context contains no uninitialized `User` proxies.

FAQ: What is the real batch-fetching algorithm?

Our explanation of batch loading was somewhat simplified, and you may see a slightly different algorithm in practice. As an example, imagine a batch size of 32. At startup time, Hibernate creates several batch loaders internally. Each loader knows how many proxies it can initialize: 32, 16, 10, 9, 8, 7, ..., 1. The goal is to minimize the memory consumption for loader creation and to create enough loaders that every possible batch fetch can be produced. Another goal is to minimize the number of SQL queries, obviously.

To initialize 31 proxies, Hibernate executes 3 batches (you probably expected 1, because $32 > 31$). The batch loaders that are applied are 16, 10, and 5, as automatically selected by Hibernate. You can customize this batch-fetching algorithm with the property `hibernate.batch_fetch_style` in the persistence unit configuration. The default is `LEGACY`, which builds and selects several batch loaders on startup. Other options are `PADDED` and `DYNAMIC`. With `PADDED`, Hibernate builds only one batch loader SQL query on startup with placeholders for 32 arguments in the `IN` clause and then repeats bound identifiers if fewer than 32 proxies have to be loaded. With `DYNAMIC`, Hibernate dynamically builds the batch SQL statement at runtime, when it knows the number of proxies to initialize.

Batch fetching is also available for collections:

Path: Ch12/batch/src/main/java/com/manning/javapersistence/ch12/batch/Item.java

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.BatchSize(size = 5)
    private Set<Bid> bids = new HashSet<>();
    // ...
}
```

If we now force the initialization of one `bids` collection, up to five more `Item#bids` collections, if they're uninitialized in the current persistence context, are loaded right away:

Path: Ch12/batch/src/test/java/com/manning/javapersistence/ch12/batch/Batch.java

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM
for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID in (?, ?, ?, ?, ?)
}
```

When we call `item.getBids().size()` for the first time while iterating, a whole batch of `Bid` collections are preloaded for the other `Item` instances.

Batch fetching is a simple and often smart optimization that can significantly reduce the number of SQL statements that would otherwise be necessary to initialize all the proxies and collections. Although we may prefetch data we won't need in the end and consume more memory, the reduction in database round trips can make a huge difference. Memory is cheap, but scaling database servers isn't.

Another prefetching algorithm that isn't a blind guess uses subselects to initialize many collections with a single statement.

12.2.4 Prefetching collections with subselects

A potentially better strategy for loading all `bids` of several `Item` instances is prefetching with a subselect. To enable this optimization, add a Hibernate annotation to the collection mapping:

Path: Ch12/subselect/src/main/java/com/manning/javapersistence/ch12/subselect/Item.java

```
@Entity
public class Item {
    @OneToMany(mappedBy = "item")
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SUBSELECT
    )
    private Set<Bid> bids = new HashSet<>();
    // ...
}
```

Hibernate now initializes all `bids` collections for all loaded `Item` instances as soon as we force the initialization of one `bids` collection:

Path:

```

Ch12/subselect/src/test/java/com/manning/javapersistence/ch12/subselect/Subselect.java

List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM
for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID in (
    // select ID from ITEM
    // )
}

```

Hibernate remembers the original query used to load the `items`. It then embeds this initial query (slightly modified) in a subselect, retrieving the collection of `bids` for each `Item`.

Note that the original query that is rerun as a subselect is only remembered by Hibernate for a particular persistence context. If we detach an `Item` instance without initializing the collection of `bids`, and then merge it with a new persistence context and start iterating through the collection, no prefetching of other collections occurs.

Batch and subselect prefetching reduce the number of queries necessary for a particular procedure if you stick with a global lazy fetch plan in the mappings, helping mitigate the *n+1 selects problem*. If instead, your global fetch plan has eager loaded associations and collections, you have to avoid the *Cartesian product problem*—for example, by breaking down a `JOIN` query into several `SELECT`s.

12.2.5 Eager fetching with multiple SELECTs

When trying to fetch several collections with one SQL query and `JOINS`, we run into the *Cartesian product problem*, as analyzed earlier. Instead of a `JOIN` operation, we can tell Hibernate to eagerly load data with additional `SELECT` queries and hence avoid large results and SQL products with duplicates:

Path: Ch12/eagerselect/src/main/java/com/manning/javapersistence/ch12/eagerselect/Item.java

```
@Entity
public class Item {
    @ManyToOne(fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT           #A
    )
    private User seller;
    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT           #A
    )
    private Set<Bid> bids = new HashSet<>();
    // ...
}
```

#A FetchMode.SELECT means that the property should be loaded lazily. The default value is FetchMode.JOIN, meaning the property would be retrieved eagerly, via a JOIN.

Now, when an Item is loaded, the seller and bids have to be loaded as well:

Path:

```
Ch12/eagerselect/src/test/java/com/manning/javapersistence/ch12/eagerselect/EagerSelect.java

Item item = em.find(Item.class, ITEM_ID);                      #A
// select * from ITEM where ID = ?
// select * from USERS where ID = ?
// select * from BID where ITEM_ID = ?
em.detach(item);
assertEquals(3, item.getBids().size());                         #B
assertNotNull(item.getBids().iterator().next().getAmount());   #B
assertEquals("johndoe", item.getSeller().getUsername());       #B
```

#A Hibernate uses one SELECT to load a row from the ITEM table. It then immediately executes two more SELECTS: one loading a row from the USERS table (the seller) and the other loading several rows from the BID table (the bids). The additional SELECT queries aren't executed lazily; the find() method produces several SQL queries.

#B Hibernate followed the eager fetch plan: all data is available in detached state.

Still, all of these settings are global; they're always active. The danger is that adjusting one setting for one problematic case in the application might have negative side effects on some other procedure. Maintaining this balance can be difficult, so our recommendation is to map every entity association and collection as FetchType.LAZY, as mentioned before.

A better approach is to *dynamically* use eager fetching and JOIN operations only when needed, for a particular procedure.

12.2.6 Dynamic eager fetching

As in the previous sections, let's say we have to check the username of each Item#seller. With a lazy global fetch plan, load the needed data for this procedure and apply a *dynamic* eager fetch strategy in a query:

Path:

```
Ch12/eagerselect/src/test/java/com/manning/javapersistence/ch12/eagerselect/EagerQue
```

```
ryUsers.java

List<Item> items =
    em.createQuery("select i from Item i join fetch i.seller")           #A
        .getResultList();
// select i.*, u.*
//  from ITEM i
//  inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
em.close();                                                               #B
for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());                         #C
}
```

#A Apply a *dynamic eager* strategy in a query.

#B Detach all.

#C Hibernate followed the eager fetch plan: all data is available in detached state.

The important keywords in this JPQL query are `join fetch`, telling Hibernate to use a SQL `JOIN` (an `INNER JOIN`, actually) to retrieve the `seller` of each `Item` in the same query. The same query can be expressed with the `CriteriaQuery` API instead of a JPQL string:

Path:

```
Ch12/eagerselect/src/test/java/com/manning/javapersistence/ch12/eagerselect/EagerQue
ryUsers.java
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);
i.fetch("seller");
criteria.select(i);
List<Item> items = em.createQuery(criteria).getResultList();           #A
em.close();                                                               #B
for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());                         #C
}
```

#A Apply an *eager* strategy in a query dynamically constructed with the `CriteriaQuery` API.

#B Detach all.

#C Hibernate followed the eager fetch plan: all data is available in detached state.

Dynamic eager join fetching also works for collections. Here we load all `bids` of each `Item`:

```
Path: Ch12/eagerselect/src/test/java/com/manning/javapersistence/ch12/eagerselect/EagerQueryBids.java

List<Item> items =
    em.createQuery("select i from Item i left join fetch i.bids")           #A
    .getResultList();
// select i.*, b.*
// from ITEM i
// left outer join BID b on b.ITEM_ID = i.ID
// where i.ID = ?
em.close();                                                               #B
for (Item item : items) {
    assertTrue(item.getBids().size() > 0);                                #C
}
```

#A Apply a *dynamic* eager strategy in a query.

#B Detach all.

#C Hibernate followed the eager fetch plan: all data is available in detached state.

Now the same with the `CriteriaQuery` API:

```
Path: Ch12/eagerselect/src/test/java/com/manning/javapersistence/ch12/eagerselect/EagerQueryBids.java

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();
Root<Item> i = criteria.from(Item.class);
i.fetch("bids", JoinType.LEFT);
criteria.select(i);
List<Item> items = em.createQuery(criteria).getResultList();          #A
em.close();                                                               #B
for (Item item : items) {
    assertTrue(item.getBids().size() > 0);                                #C
}
```

#A Apply an eager strategy in a query dynamically constructed with the `CriteriaQuery` API.

#B Detach all.

#C Hibernate followed the eager fetch plan: all data is available in detached state.

Note that for collection fetching, a `LEFT OUTER JOIN` is necessary, because we also want rows from the `ITEM` table if there are no `bids`.

Writing queries by hand isn't the only available option if we want to override the global fetch plan of the domain model dynamically. We can write *fetch profiles* declaratively.

12.3 Using fetch profiles

Fetch profiles complement the fetching options in the query languages and APIs. They allow maintaining the profile definitions in either XML or annotation metadata. Early Hibernate versions didn't have support for special fetch profiles, but today Hibernate supports the following:

- *Fetch profiles*—A proprietary API based on the declaration of the profile with `@org.hibernate.annotations.FetchProfile` and execution with `Session`

`#enableFetchProfile()`. This simple mechanism currently supports overriding lazy-mapped entity associations and collections selectively, enabling a `JOIN` eager fetching strategy for a particular unit of work.

- *Entity graphs*—Specified in JPA 2.1, we can declare a graph of entity attributes and associations with the `@EntityGraph` annotation. This fetch plan, or a combination of plans, can be enabled as a hint when executing `EntityManager #find()` or queries (JPQL, criteria). The provided graph controls *what* should be loaded; unfortunately, it doesn't control *how* it should be loaded.

It's fair to say that there is room for improvement here, and we expect future versions of Hibernate and JPA to offer a unified and more powerful API.

We can externalize JPQL and SQL statements and move them to metadata. A JPQL query *is* a declarative (named) fetch profile; what we're missing is the ability to overlay different plans easily on the same base query. We've seen some creative solutions with string manipulation that are best avoided. With criteria queries, on the other hand, we already have the full power of Java available to organize the query-building code. Then the value of entity graphs is being able to reuse fetch plans across any kind of query.

Let's talk about Hibernate fetch profiles first and how we can override a global lazy fetch plan for a particular unit of work.

12.3.1 Declaring Hibernate fetch profiles

Hibernate fetch profiles are global metadata: they're declared for the entire persistence unit. Although we could place the `@FetchProfile` annotation on a class, we prefer it as package-level metadata in a `package-info.java`:

Path: Ch12/profile/src/main/java/com/manning/javapersistence/ch12/profile/package-info.java

```
@org.hibernate.annotations.FetchProfiles({
    @FetchProfile(name = Item.PROFILE_JOIN_SELLER,                      #A
        fetchOverrides = @FetchProfile.FetchOverride(
            entity = Item.class,
            association = "seller",
            mode = FetchMode.JOIN                                #C
        )),
    @FetchProfile(name = Item.PROFILE_JOIN_BIDS,
        fetchOverrides = @FetchProfile.FetchOverride(
            entity = Item.class,
            association = "bids",
            mode = FetchMode.JOIN
        ))
})
```

#A Each profile has a name. This is a simple string isolated in a constant.

#B Each override in a profile names one entity association or collection.

#C `FetchMode.JOIN` means the property would be retrieved eagerly, via a JOIN.

The profiles can now be enabled for a unit of work. We need the Hibernate API to enable a profile. It's then active for any operation in that unit of work. The `Item#seller` may be fetched with a join in the same SQL statement whenever an `Item` is loaded with this `EntityManager`.

We can overlay another profile on the same unit of work. The `Item#seller` and the `Item#bids` collections will be fetched with a join in the same SQL statement whenever an `Item` is loaded.

Path: Ch12/profile/src/test/java/com/manning/javapersistence/ch12/profile/Profile.java

```
Item item = em.find(Item.class, ITEM_ID);                      #A
em.clear();
em.unwrap(Session.class).enableFetchProfile(Item.PROFILE_JOIN_SELLER);  #B
item = em.find(Item.class, ITEM_ID);
em.clear();
em.unwrap(Session.class).enableFetchProfile(Item.PROFILE_JOIN_BIDS);  #C
item = em.find(Item.class, ITEM_ID);
```

#A The `Item#seller` is mapped lazily, so the default fetch plan only retrieves the `Item` instance.

#B Fetch the `Item#seller` with a join in the same SQL statement whenever an `Item` is loaded with this `EntityManager`.

#C Fetch `Item#seller` and `Item#bids` with a join in the same SQL statement whenever an `Item` is loaded.

Although basic, Hibernate fetch profiles can be an easy solution for fetching optimization in smaller or simpler applications. Starting from JPA 2.1, the introduction of *entity graphs* enables similar functionality in a standard fashion.

12.3.2 Working with entity graphs

An entity graph is a declaration of entity nodes and attributes, overriding or augmenting the default fetch plan when we execute an `EntityManager#find()` or with a hint on query operations. This is an example of a retrieval operation using an entity graph:

```
Path: Ch12/fetchloadgraph/src/test/java/com/manning/javapersistence/ch12/fetchloadgraph/FetchLoadGraph.java

Map<String, Object> properties = new HashMap<>();
properties.put(
    "javax.persistence.loadgraph",
    em.getEntityGraph(Item.class.getSimpleName())) #A
);
Item item = em.find(Item.class, ITEM_ID, properties);
// select * from ITEM where ID = ?
```

#A The name of the entity graph we're using is `Item`, and the hint for the `find()` operation indicates it should be the *load graph*. This means attributes that are specified by attribute nodes of the entity graph are treated as `FetchType.EAGER` and attributes that aren't specified are treated according to their specified or default `FetchType` in the mapping.

This is the declaration of this graph and the default fetch plan of the entity class:

```
Path: Ch12/fetchloadgraph/src/main/java/com/manning/javapersistence/ch12/fetchloadgraph/Item.java

@NamedEntityGraphs({
    @NamedEntityGraph
})
@Entity
public class Item {
    @NotNull
    @ManyToOne(fetch = FetchType.LAZY)
    private User seller;
    @OneToMany(mappedBy = "item")
    private Set<Bid> bids = new HashSet<>();
    @ElementCollection
    private Set<String> images = new HashSet<>();
    // ...
}
```

#A Entity graphs in metadata have names and are associated with an entity class; they're usually declared in annotations on top of an entity class. We can put them in XML if we like. If we don't give an entity graph a name, it gets the simple name of its owning entity class, which here is `Item`.

If we don't specify any attribute nodes in the graph, like the empty entity graph in the last example, the defaults of the entity class are used. In `Item`, all associations and collections are mapped lazy; this is the default fetch plan. Hence, what we've done so far makes little difference, and the `find()` operation without any hints will produce the same result: the `Item` instance is loaded, and the `seller`, `bids`, and `images` aren't.

Alternatively, we can build an entity graph with an API:

```
Path: Ch12/fetchloadgraph/src/test/java/com/manning/javapersistence/ch12/fetchloadgraph/FetchLoadGraph.java

EntityGraph<Item> itemGraph = em.createEntityGraph(Item.class);
Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", itemGraph);
Item item = em.find(Item.class, ITEM_ID, properties);
```

This is again an empty entity graph with no attribute nodes, given directly to a retrieval operation.

Let's say we want to write an entity graph that changes the lazy default of `Item#seller` to eager fetching, when enabled:

```
Path: Ch12/fetchloadgraph/src/main/java/com/manning/javapersistence/ch12/fetchloadgraph/Item.java

@NamedEntityGraphs({
    @NamedEntityGraph(
        name = "ItemSeller",
        attributeNodes = {
            @NamedAttributeNode("seller")
        }
    )
})
@Entity
public class Item {
    // ...
}
```

Now enable this graph by name when you want the `Item` and the `seller` eagerly loaded:

```
Path: Ch12/fetchloadgraph/src/main/java/com/manning/javapersistence/ch12/fetchloadgraph/Item.java

Map<String, Object> properties = new HashMap<>();
properties.put(
    "javax.persistence.loadgraph",
    em.getEntityGraph("ItemSeller")
);
Item item = em.find(Item.class, ITEM_ID, properties);
// select i.*, u.*
// from ITEM i
// inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

If we don't want to hardcode the graph in annotations, we build it with the API instead:

```
Path: Ch12/fetchloadgraph/src/test/java/com/manning/javapersistence/ch12/fetchloadgraph/FetchLoadGraph.java

EntityGraph<Item> itemGraph = em.createEntityGraph(Item.class);
itemGraph.addAttributeNodes(Item_.seller); #A
Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", itemGraph);
Item item = em.find(Item.class, ITEM_ID, properties);
// select i.*, u.*
// from ITEM i
// inner join USERS u on u.ID = i.SELLER_ID
// where i.ID = ?
```

#A The `Item_.class` belongs to the static metamodel. It is automatically generated by including the *Hibernate JPA2 Metamodel Generator dependency* in the project. You may revisit section 3.3.4 for more details.

So far we've seen only properties for the `find()` operation. Entity graphs can also be enabled for queries, as hints:

```
Path: Ch12/fetchloadgraph/src/test/java/com/manning/javapersistence/ch12/fetchloadgraph/FetchLoadGraph.java

List<Item> items =
    em.createQuery("select i from Item i")
        .setHint("javax.persistence.loadgraph", itemGraph)
        .getResultList();
// select i.*, u.*
//   from ITEM i
//   left outer join USERS u on u.ID = i.SELLER_ID
```

Entity graphs can be complex. The following declaration shows how to work with reusable subgraph declarations:

```
Path: Ch12/fetchloadgraph/src/main/java/com/manning/javapersistence/ch12/fetchloadgraph/Bid.java

@NamedEntityGraphs({
    @NamedEntityGraph(
        name = "BidBidderItemSellerBids",
        attributeNodes = {
            @NamedAttributeNode(value = "bidder"),
            @NamedAttributeNode(
                value = "item",
                subgraph = "ItemSellerBids"
            )
        },
        subgraphs = {
            @NamedSubgraph(
                name = "ItemSellerBids",
                attributeNodes = {
                    @NamedAttributeNode("seller"),
                    @NamedAttributeNode("bids")
                }
            )
        }
    )
    @Entity
    public class Bid {
        // ...
    }
}
```

This entity graph, when enabled as a load graph when retrieving `Bid` instances, also triggers eager fetching of `Bid#bidder`, the `Bid#item`, and furthermore the `Item#seller` and all `Item#bids`. Although you're free to name your entity graphs any way you like, we recommend that you develop a convention that everyone in your team can follow, and move the strings to shared constants.

With the entity graph API, the previous plan looks as follows:

Path:

```
Ch12/fetchloadgraph/src/test/java/com/manning/javapersistence/ch12/fetchloadgraph/FetchLoadGraph.java
```

```
EntityGraph<Bid> bidGraph = em.createEntityGraph(Bid.class);
bidGraph.addAttributeNodes(Bid_.bidder, Bid_.item);
Subgraph<Item> itemGraph = bidGraph.addSubgraph(Bid_.item);
itemGraph.addAttributeNodes(Item_.seller, Item_.bids);
Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.loadgraph", bidGraph);
Bid bid = em.find(Bid.class, BID_ID, properties);
```

We've only seen entity graphs as *load* graphs so far. There is another option: we can enable an entity graph as a *fetch graph* with the `javax.persistence.fetchgraph` hint. If we execute a `find()` or query operation with a fetch graph, any attributes and collections not in the plan will be made `FetchType.LAZY` and any nodes in the plan will be `FetchType.EAGER`. This effectively ignores all `FetchType` settings in the entity attribute and collection mappings, whereas the load graph feature was only augmenting.

Two weak points of the JPA entity graph operations are worth mentioning because you'll run into them quickly. First, you can only modify fetch plans, not the Hibernate fetch strategy (batch/subselect/join/select). Second, declaring an entity graph in annotations or XML isn't fully type-safe: the attribute names are strings. The `EntityGraph` API at least is type-safe.

12.4 Summary

- A fetch profile combines a fetch plan (what data should be loaded) with a fetch strategy (how the data should be loaded), encapsulated in reusable metadata or code.
- We created a global fetch plan and defined which associations and collections should be loaded into memory at all times. We defined the fetch plan based on use cases, how to access associated entities and iterate through collections in the application, and which data should be available in detached state.
- We analyzed how to select the right fetching strategy for the fetch plan. The goal is to minimize the number of SQL statements and the complexity of each SQL statement that must be executed. We especially want to avoid the *n+1 selects* and *Cartesian product* issues we examined in detail, using various optimization strategies.
- We explored Hibernate fetch profiles and entity graphs, the fetch profiles in JPA.