Chapter XXV
Hibernate:
A Full Object Relational Mapping Service

Allan M. Hart
Minnesota State University, Mankato, USA

ABSTRACT

This chapter presents a brief overview of the object/relational mapping service known as Hibernate. Based on work provided in the book Java Persistence with Hibernate, it is argued that the paradigm mismatch problem consists of five problems: the problem of granularity, the problem of subtypes, the problem of identity, the problem of associations, and the problem of data navigation. It is argued that Hibernate, if it is to be considered a successful object/relational mapping service, must solve the paradigm mismatch problem and, hence, each of the five problems noted above. A simplified version of an order entry system is presented together with the mapping files required to store persistent objects to a database. Examples are given for one-to-one, one-to-many, and many-to-many mappings. The distinction between value and entity types is explained and the mapping technique required for value types is introduced into the order entry system application. The n+1 selects problem is explained and a strategy for solving that problem using Hibernate's support for lazy, batch, and eager fetching strategies is discussed.

INTRODUCTION

The purpose of this chapter is to provide the reader with an introduction to Hibernate. Hibernate, as described at its Web site, is "a powerful, high performance object/relational persistencee and query service" (Hibernate, 2008).

Hibernate is one among a number of so-called persistence frameworks. Other notables include TopLink, iBATIS, and Java data objects (JDO) (Oracle TopLink, 2008; IBATis, 2008; JDO, 2008). The basic responsibility of any persistence framework is to manage persistent data, that is, data that needs to be saved to persistent storage (usually a relational database) from one invocation of the application to the next. Not all objects created by a given application constitute persistent data but many of them do. For example, in an ecommerce application, information regarding the customer's name, address, and credit card information as well as the particular products the customer has ordered (and the quantity of each), constitutes data that needs to be saved to persistent storage.

Persistence frameworks can generally be divided into those that are based on an approach known as object/relational mapping (ORM) and those that are not. Both Hibernate and TopLink, for example, are correctly classified as ORM services. iBATIS, on the other hand, though often listed as an ORM service, is, strictly speaking, not an ORM service at all, but rather a data mapping service. With an ORM service like Hibernate, what are mapped, *very* roughly speaking, are classes to tables. Instances of classes become rows in a database table and associations between classes become foreign key constraints between database tables. What are mapped with iBATIS's data mapping services, on the other hand, are not tables to classes, but rather the parameters and results of SQL statements to classes.[1]

The need for ORM services grew out of the realization over the last several decades that a paradigm mismatch problem exists between the world of object-oriented programming languages and relational databases. In this chapter we will explore some of the details of this problem as well as the ways in which an ORM service like Hibernate attempts to solve the problem.

BACKGROUND

The evolution of programming languages over the last 50 years has seen a large growth not only in the *number* of different programming languages but also in the number of different *types* of programming languages. During the 60s procedural languages such as FORTRAN, BASIC, and Pascal were the rage. While object-oriented programming (OOP) was in its infancy in the 60s, its first implementation language, Smalltalk, appeared. During the 70s and 80s SmallTalk continued to evolve and newer OOP languages such as Object Pascal and C++ appeared. The appearance of Java during the mid 90s solidified OOP's position as the predominant programming language paradigm. During this same period, changes in the database world were also afoot. Edgar F. Codd's seminal paper "A Relational Model of Data for Large Shared Data Banks" was published in 1970. This paper triggered much work in the 70s and, during this time, implementations of the relational model began to appear. Soon, thereafter, relational databases had largely replaced their hierarchical and network predecessors.

As the development of OOP languages and relational database management systems proceeded, it soon became clear that there was a paradigm mismatch problem.[2] The details of the problem will be presented in the next section. For now, suffice it to say that there were primarily two responses to this problem. Some have argued that the relational model should be abandoned and that object-oriented databases should be embraced. Others have argued that relational databases should instead be expanded to include at least some of the features found in object-oriented programming languages, for example, user-defined types and inheritance. This bifurcation in the historical road down which database development has gone, constitutes, one might say, a fork in that road. During the 90s, one saw much development being done in the area of pure object-oriented databases. However, while this work deserves much praise, the effort did not bear much fruit. While pure object-oriented database management systems were developed, none of them caught on in the marketplace and much of that effort has now been abandoned. During roughly the same time period, much effort was also expended toward the other fork. The notion of a user defined type (UDT) was brought into the database world with implementations being provided for both Oracle and SQL server. While this has served to alleviate the paradigm mismatch problem to some degree (at least for those platforms), it has not successfully overcome the problem in all of its detail. Moreover, the solutions provided are not portable among those platforms.