# Pro Hibernate 3

DAVE MINTER AND JEFF LINWOOD

Apress®

**Pro Hibernate 3**



ISBN (pbk): 1-59059-511-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1



Lead Editor: Steve Anglin
Technical Reviewer: Dilip Thomas
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Linda Marousek
Production Manager: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Michael Brinkman
Artist: April Milne
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.



The source code for this book is available to readers at http://www.apress.com in the Downloads section.

*To our families.*

# Contents at a Glance

# Contents

# PART 2 Hibernate 3 Reference

# About the Authors

**DAVE MINTER** is a freelance IT consultant (`http://paperstack.com/cv/`) from rainy London, England. His liking for computers was kicked off when a Wang 2200 minicomputer made a big impression on him at the tender age of six. Since then he has worked for the biggest of blue chips and the smallest of startups. These days he makes his living designing and building multi-tier applications that "just work." Dave is the coauthor with Jeff of *Building Portals with the Java Portlet API* (Apress, 2004). He has a computer studies degree from the University of Glamorgan.

**JEFF LINWOOD** is a software developer and consultant with the Gossamer Group (`http://www.gossamer-group.com/`) in sunny Austin, Texas. Jeff has been in software programming since he had a 286 in high school. He was caught up with the Internet when he got access to a Unix shell account and it has been downhill ever since. Jeff coauthored *Building Portals with the Java Portlet API* (Apress, 2004) with Dave and *Pro Struts Applications* (Apress, 2003), and was a technical reviewer for *Enterprise Java Development on a Budget* (Apress, 2004) and *Extreme Programming with Ant* (SAMS, 2003). He has a chemical engineering degree from Carnegie Mellon University.

# About the Technical Reviewer

**DILIP THOMAS** is an Open Source enthusiast who keeps a close watch on LAMP technologies, Open Standards, and the full range of Apache Jakarta projects. He is coauthor of *PHP MySQL Website Programming: Problem - Design - Solution* (Apress, 2003) and a technical reviewer/editor on several Open Source/Open Standard book projects. Dilip is an Editorial Director at Software & Support Verlag GmbH.

Dilip resides in Bangalore with his beautiful wife, Indu, and several hundred books and journals. You can reach him via email at `dilip.thomas@gmail.com`.

# Acknowledgments

This book would not have been possible without the energy and enthusiasm of the Apress staff: especially Steve Anglin for entrusting us with the project in the first place; Beth Christmas, our Project Manager; Linda Marousek, our Copy Editor; and Katie Stence, our Production Editor. Thank you all.

We are indebted to Dilip Thomas, our Technical Editor, whose timely and pithy commentary helped us to polish the rough edges of this work. The parts of the book that we are particularly proud of are usually the direct result of his notes and enquiries.

Dave would like to thank his parents for entrusting the family ZX81 to his clutches some years past, and his wonderful girlfriend Kalani Seymour for her patience and enthusiasm. He would also like to thank Jeff Verdegan for his assistance with a particularly tricky example that makes an appearance in Chapter 13.

Jeff would like to thank his family, Nancy, Judy, Rob, and Beth, for being supportive during the book-writing process. He'd also like to thank his friends Giuliano, Ronalyn, Roman, Karl, Michael, Jason, Valerie, Micah, and Betsy.

Where this book fails to achieve its aims, we the authors are, of course, entirely responsible.

# Introduction

Virtually every application we have worked on has involved a database. While Java and the JDBC standard have significantly reduced the effort involved in carrying out simple database operations, the point of contact between the object-oriented world and the relational world always causes problems.

Hibernate offers a chance to automate most of the hard parts of this particular problem. By writing a simple XML file, you can make the database look object oriented. Along the way, you get some performance improvements. You also get an enormous improvement in the elegance of your code.

We were originally skeptical of yet another object-relational system, but Hibernate is so much easier to work with that we are entirely converted. Neither of us expect to write a database-based system with JDBC and SQL prepared statements again—Hibernate or its successors will form an essential part of the foundations. Hibernate is not just "another" object-relational system, it is the standard to beat.

Speaking of standards, the EJB3 specification has received a lot of input from the Hibernate team—who have, in turn, made efforts to reflect the emerging standard in their API. Therefore, to learn good practices for EJB3, start here with Hibernate.

## Who This Book Is For

This book is for Java developers who have at least some basic experience with databases and using JDBC. You will need to have a basic familiarity with SQL and database schemas. While it is not required, it will certainly be helpful if you have an understanding of the aims of database normalization. We do not assume that you have prior experience with Hibernate or any other object-relational technology.

Hibernate is an Open Source project, so it is fitting that all of our examples make use of Open Source technologies. Part of Hibernate's promise is that many of the details of a particular relational database are abstracted away, so most of our examples will work with any database Hibernate supports. When we discuss database features that not all supported databases have, such as stored procedures, we specifically mention which database we used—both of our example databases are Open Source and readily available.

This book is not an academic text and does not generally attempt to describe the internal workings of Hibernate itself. As professional developers, we have tried to impart the understanding of how to use Hibernate rather than how to write it!

## How This Book Is Organized

This book is roughly divided into two parts, beginning with a primer on the basics of Hibernate in Chapters 1 through 4 and continuing with more technical or design-oriented content in Chapters 5 through 14.

We encourage readers who are completely new to this technology to read through the primer section as this will give you a grounding in the basic requirements for a Hibernate-based application.

Readers who are familiar with Hibernate 2 should start with Chapter 14, where we discuss the differences between Hibernate 2 and Hibernate 3, and then refer back to appropriate chapters:

- *Chapter 1, An Introduction to Hibernate 3*: A basic introduction to the fundamentals of Hibernate and an overview of some sample code.
- *Chapter 2, Integrating and Configuring Hibernate*: How to integrate Hibernate into your Java application, an overview of the configuration options for Hibernate, and we explain how Hibernate deals with the differences between its supported relational databases.
- *Chapter 3, Building a Simple Application*: Two working-example Hibernate programs in full with extensive commentary on how the source code, mapping files, and database are related.
- *Chapter 4, Using Annotations with Hibernate*: How to use the new EJB3 annotations with Hibernate 3 to create object-relational mappings in your Java source code.
- *Chapter 5, The Persistence Lifecycle*: We explain how Hibernate manages the objects that represent data in tables, and also present an overview of the relevant methods for creating, retrieving, updating, and deleting objects through the Hibernate session.
- *Chapter 6, Creating Mappings*: We discuss mapping files in depth. We cover all of the commonly used elements in detail.
- *Chapter 7, Querying Objects with Criteria*: How to use the Hibernate criteria query API to selectively retrieve objects from the database.
- *Chapter 8, Querying with HQL and SQL*: This chapter contains a discussion of Hibernate's object-oriented query language, HQL, and how to use native SQL with Hibernate.
- *Chapter 9, Using the Session*: The ways in which sessions, transactions, locking, caching, and multi-threading are related. We also present a comprehensive example of a deadlock.
- *Chapter 10, Design Considerations with Hibernate 3:* How the Hibernate persistence layer of an application is designed.
- *Chapter 11, Events and Interceptors*: We discuss these very similar features. Interceptors were available in Hibernate 2 and are relatively unchanged. Events, while similar to Interceptors, were introduced with Hibernate 3.
- *Chapter 12, Hibernate Filters*: Hibernate provides filtering functionality for limiting the data returned by queries to a definable subset. We give an example showing this useful functionality and explain where you might use it in an application.

- *Chapter 13, Fitting Hibernate into the Existing Environment*: We discuss how to go about integrating Hibernate with legacy applications and databases. We also present an example of how to invoke a stored procedure.
- *Chapter 14, Upgrading from Hibernate 2*: For users familiar with Hibernate 2, we discuss the changes and additions in Hibernate 3.

PART 1

# Hibernate 3 Primer

CHAPTER 1

# An Introduction to Hibernate 3

Most significant development projects involve a relational database. The mainstay of most commercial applications is the large-scale storage of ordered information, such as catalogs, customer lists, contract details, published text, and architectural designs.

With the advent of the World Wide Web, the demand for databases has increased. Though they may not know it, the customers of online bookshops and newspapers are using databases. Somewhere in the guts of the application a database is being queried and a response is offered.

While the demand for such applications has grown, their creation has not become noticeably simpler. Some standardization has occurred—the most successful being the Enterprise JavaBeans (EJBs) standard of Java 2 Enterprise Edition (J2EE), which provides for container and bean-managed persistence of Entity Bean classes. Unfortunately, this and other persistence models all suffer to one degree or another from the mismatch between the relational model and the object-oriented model. In short, database persistence is difficult.

There are solutions for which EJBs are more appropriate, for which some sort of Object Relational Mapping (ORM) like Hibernate is appropriate, and some for which the traditional connected[1] approach of direct access via the JDBC API is most appropriate. Making the call on which to use requires an intimate knowledge of the strengths and weaknesses of them all—but all else being equal, we think that Hibernate offers compelling advantages in terms of ease of use.

To illustrate some of Hibernate's other strengths, in this chapter we will show you a "Hello World" Java database example. This application allows previously stored messages to be displayed using a simple key such as the "Message of the day." We will show the essential implementation details using the connected approach and using Hibernate.

Both of our examples are invoked from the main method in Listing 1-1, with the point at which we invoke the specific implementation marked in bold.

1. There is no standard way of referring to "direct use of JDBC" as opposed to ORM or EJBs, so we have adopted the term "connected" as the least awkward way of expressing this.

**Listing 1-1.** *The Main Method That Will Invoke Our Hibernate and Connected Implementations*

```
public static void main(String[] args) {
   if (args.length != 1) {
      System.err.println("Nope, enter one message number");
   } else {
      try {
         int messageId = Integer.parseInt(args[0]);
         Motd motd = getMotd(messageId);
         if (motd != null) {
            System.out.println(motd.getMessage());
         } else {
            System.out.println("No such message");
         }
      } catch (NumberFormatException e) {
         System.err.println("You must enter an integer - " + args[0]
               + " won't do.");
      } catch (MotdException e) {
         System.err.println("Couldn't get the message: " + e);
      }
   }
}
```

# Plain Old Java Objects (POJOs)

In our ideal world, it would be trivial to take any Java object and persist it to the database. No special coding would be required to achieve this, no performance penalty would ensue, and the result would be totally portable. The common term for the direct persistence of traditional Java objects is Object Relational Mapping. There is a direct and simple correspondence between the POJOs[2], something that the forthcoming changes in the EJB3 standard look set to emulate.

Where Entity Beans have to follow a myriad of awkward naming conventions, POJOs can be any Java object at all. Hibernate allows you to persist POJOs, with very few constraints. Listing 1-2 is an example of a simple POJO to represent our "Message of the day" (`Motd`) announcement.

**Listing 1-2.** *The POJO That We Are Using in This Chapter's Examples*

```
public class Motd {
   protected Motd() {
   }
```

2. Java objects requiring no special treatment to be stored are often referred to as Plain Old Java Objects, or POJOs for short. This is really just to provide a handy term to differentiate them from Entity Beans and the like, which must conform to complicated and limiting contracts. The name also conveys something of the benefits of Hibernate—you don't have to do anything special to support persistence of a POJO via Hibernate, so you really can reuse your existing Java objects.

```
   public Motd(int messageId, String message) {
      this.messageId = messageId;
      this.message = message;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getMessage() {
      return message;
   }

   public void setMessage(String message) {
      this.message = message;
   }

   private int messageId;
   private String message;
}
```

What sort of POJO can be persisted? Practically anything. It has to provide a default constructor (but this can be given package or private scope to avoid pollution of your API), but that's it. Not an especially onerous burden.

# Origins of Hibernate and Object Relational Mapping

If Hibernate is the solution, what was the problem? The gargantuan body of code in Listing 1-3 is required to populate our example `Motd` object from the database even when we know the appropriate message identifier:

**Listing 1-3.** *The Connected Approach to Retrieving the POJO*

```
   public static Motd getMotd(int messageId) throws MotdException {
      Connection c = null;
      PreparedStatement p = null;
      Motd message = null;

      try {

         Class.forName("org.postgresql.Driver");
         c = DriverManager.getConnection(
```

```
                "jdbc:postgresql://127.0.0.1/hibernate",
                "hibernate",
                "hibernate");
            p = c.prepareStatement(
                "select message from motd where id = ?");

            p.setInt(1, messageId);
            ResultSet rs = p.executeQuery();

            if (rs.next()) {
               String text = rs.getString(1);
               message = new Motd(messageId, text);

               if (rs.next()) {
                  log.warning(
                     "Multiple messages retrieved for message ID: "
                   + messageId);
               }
            }

         } catch (Exception e) {
            log.log(Level.SEVERE, "Could not acquire message", e);
            throw new MotdException(
                "Failed to retrieve message from the database.", e);
         } finally {
            if (p != null) {
               try {
                  p.close();
               } catch (SQLException e) {
                  log.log(Level.WARNING,
                        "Could not close ostensibly open statement.", e);
               }
            }

            if (c != null) {
               try {
                  c.close();
               } catch (SQLException e) {
                  log.log(Level.WARNING,
                        "Could not close ostensibly open connection.", e);
               }
            }
         }

         return message;
      }
```

While some of this can be trimmed down—for example, the acquisition of a connection is more likely to be done in a single line from a `DataSource`—there is still a lot of boilerplate code here. Entering this manually is tedious and error prone.

## EJBs As a Persistence Solution

So why not just use EJBs to retrieve data? Entity Beans are, after all, designed to represent, store, and retrieve data in the database.

Strictly speaking, an Entity Bean is permitted two types of persistence in an EJB server: Bean-Managed Persistence (BMP) and Container-Managed Persistence (CMP). In BMP, the bean itself is responsible for carrying out all of the Structured Query Language (SQL) associated with storing and retrieving its data—in other words, it requires the author to create the appropriate JDBC logic complete with all the boilerplate in Listing 1-3. CMP, on the other hand, requires the container to carry out the work of storing and retrieving the bean data. So why doesn't that solve the problem? Here are just a few of the reasons:

- CMP Entity Beans require a one-to-one mapping to database tables.
- They are (by reputation at least) slow.
- Someone has to determine which bean field maps to which table column.
- They require special method names. If these are not followed correctly, they will fail silently.
- Entity Beans have to reside within a J2EE application server environment—they are a heavyweight solution.
- They cannot readily be extracted as "general purpose" components for other applications.
- They can't be serializable.
- They rarely exist as portable components to be dropped into a foreign application—you generally have to roll your own EJB solution.

## Hibernate As a Persistence Solution

Hibernate addresses a lot of these points, or alleviates some of the pain where it can't, so we'll address the points in turn.

Hibernate does not require you to map one POJO to one table. A POJO can be constructed out of a selection of table columns, or several POJOs can be persisted into a single table.

Though there is some performance overhead while Hibernate starts up and processes its configuration files, it is generally perceived as being a fast tool. This is very hard to quantify, and, to some extent, the poor reputation of Entity Beans may have been earned more from the mistakes of those designing and deploying such applications than on its own merits. As with all performance questions, you should carry out tests rather than rely on anecdotal evidence.

In Hibernate it is possible, but not necessary, to specify the mappings at deployment time. The EJB solution ensures portability of applications across environments, but the Hibernate approach tends to reduce the pain of deploying an application to a new environment.

Hibernate persistence has no requirement for a J2EE application server or any other special environment. It is, therefore, a much more suitable solution for stand-alone applications, for client-side application storage, and other environments where a J2EE server is not immediately available.

Hibernate uses POJOs that can very easily and naturally be generalized for use in other applications. There is no direct dependency upon the Hibernate libraries so the POJO can be put to any use that does not require persistence—or can be persisted using any other POJO "friendly" mechanism.

Hibernate presents no problems when handling serializable POJOs.

Finally, there is a very large body of preexisting code. Any Java object capable of being persisted to a database is a candidate for Hibernate persistence. Therefore, Hibernate is an excellent solution to choose when adding a standard persistence layer to an application that uses an ad hoc solution, or which has not yet had database persistence incorporated into it. Furthermore, by selecting Hibernate persistence, you are not tying yourself to any particular design decisions for the business objects in your application.

# A Hibernate Hello World Example

Listing 1-4 shows how much less boilerplate is required with Hibernate than with the connected approach in Listing 1-3.

**Listing 1-4.** *The Hibernate Approach to Retrieving Our POJO*

```
public static Motd getMotd(int messageId)
   throws MotdException
{
   SessionFactory sessions =
      new Configuration().configure().buildSessionFactory();
   Session session = sessions.openSession();
   Transaction tx = null;
   try {
      tx = session.beginTransaction();

      Motd motd =
         (Motd)session.get( Motd.class,
                            new Integer(messageId));

      tx.commit();
      tx = null;
      return motd;
   } catch ( HibernateException e ) {
```

```
            if ( tx != null ) tx.rollback();
            log.log(Level.SEVERE, "Could not acquire message", e);
            throw new MotdException(
                  "Failed to retrieve message from the database.",e);
         } finally {
            session.close();
         }
      }
```

Even for this trivial example there would be a further reduction in the amount of code required in a real deployment—particularly in an application-server environment. The `SessionFactory` would normally be created elsewhere and made available to the application as a Java Native Directory Interface (JNDI) resource. It is particularly interesting that because this object is keyed on its identifier, we can carry out all of the SQL of Listing 1-4, along with the population of the object from the result set using the single line

```
Motd motd = (Motd)session.get(Motd.class, new Integer(messageId));
```

When more complex queries that do not directly involve the primary key are required, some SQL (or rather, HQL [the Hibernate Query Language]) is required, but the work of populating objects is permanently eradicated.

Some of the additional code in our Hibernate 3 example actually provides functionality (transactionality and caching) beyond that of the connected example. In addition, we have greatly reduced the amount of error-handling logic required.

# Mappings

Of course there is more to it than this—Hibernate needs something to tell it which tables relate to which objects (the information is actually provided in an XML-mapping file). But while some tools inflict vast, poorly documented XML configuration files on their users, Hibernate offers a breath of fresh air; you create and associate a small, clear mapping file with each of the POJOs that you wish to map into the database. You're permitted to use a single monolithic configuration file if you prefer, but it's neither compulsory nor encouraged.

A Document Type Definition (DTD) is provided for all Hibernate's configuration files, so with a good XML editor you should be able to take advantage of autocompletion and autovalidation of the files as you create them. A number of tools allow the automated creation of the mapping files, and Java 5 annotations can be used to replace them entirely.

Listing 1-5 shows the file mapping our `Motd` POJO into the database.

**Listing 1-5.** *The XML File Mapping Our POJO to the Database*

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
   <class name="Motd" table="Motd">
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="message" column="message" type="string"/>
   </class>
</hibernate-mapping>
```

It would be reasonable to ask if the complexity has simply been moved from the application code into the XML-mapping file. But, in fact, this isn't really the case for several reasons.

First, the XML file is much easier to edit than a complex population of a POJO from a result set.

Second, we have still done away with the complicated error handling that was required with the connected approach. This is probably the weakest reason, however, since there are various techniques to minimize this without resorting to Hibernate.

Third, finally, and most importantly, if the POJO provides bean-like property access methods and a default constructor, then tools provided with Hibernate allow the direct generation of a mapping file. We will discuss the use of these tools in depth in Chapter 10.

# Database Generation

If you are creating a new Hibernate application around an existing database, creating the database schema is not an issue; it is presented to you on a platter. If you are starting a new application, you will need to create the schema, the POJOs, and the mapping directly.

Yet again, there is a tool to make life easy for you. The `SchemaExport` tool allows a database schema to be generated for your database directly from the mapping file. This is even better than it sounds! Hibernate comes with intimate knowledge of various different dialects of SQL, so the schema will be tailored to your particular database software—or can be generated for each different database to which you want to deploy your application.

The generation of a SQL script to create your database schema is similar to the generation of the mapping file from the POJO:

```
java org.hibernate.tool.hbm2ddl.SchemaExport Motd.hbm.xml
```

# Bootstrapping Your Configuration

You've probably now realized one of the things we really like about Hibernate—if you have any one of the three things you need (POJOs, schema, mapping), you can generate the other two without having to write them from scratch.

Figure 1-1 shows the various ways in which files and databases can be created: