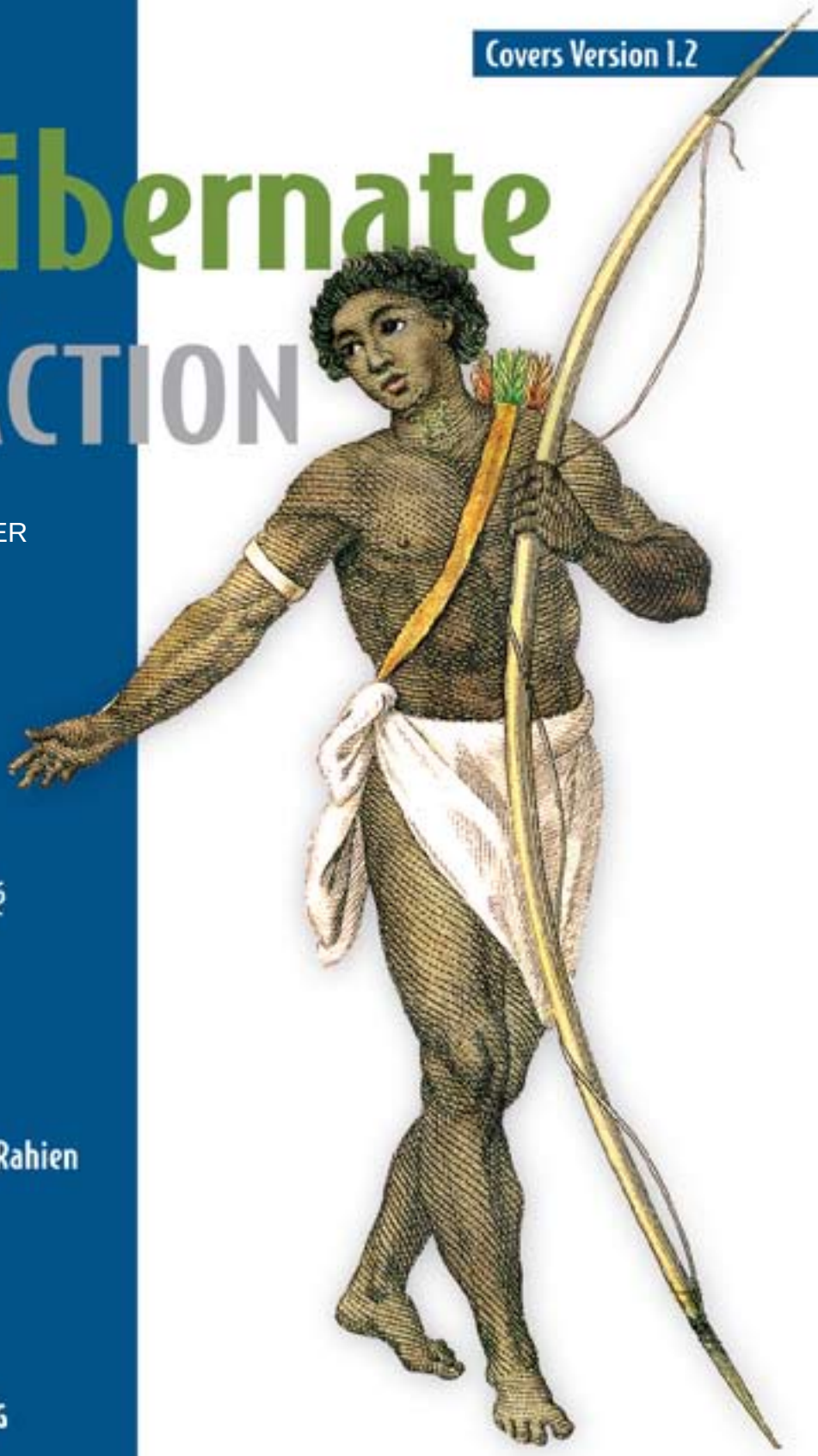
Covers Version 1.2

NHibernate IN ACTION

SAMPLE CHAPTER



Pierre Henri Kuaté
Tobin Harris
Christian Bauer
Gavin King

FOREWORD BY Ayende Rahien

MANNING

NHibernate in Action

Pierre Henri Kuaté
Tobin Harris
Christian Bauer
Gavin King

Chapter 1

brief contents

Part 1

Discovering ORM with NHibernate

The first part of the book provides insights into what ORM is, why it exists, and how it fits in a typical .NET application. We then introduce NHibernate, using a clear and simple example to help you understand how the various pieces of an NHibernate application fit together.

1 Object/relational persistence in .NET

This chapter covers

- .NET persistence and relational databases
- Layering .NET applications
- Approaches to implementing persistence in .NET
- How NHibernate solves persistence of objects in relational databases
- Advanced persistence features

Software development is an ever-changing discipline in which new techniques and technologies are constantly emerging. As software developers, we have an enormous array of tools and practices available, and picking the right ones can often make or break a project. One choice that is thought to be particularly critical is how to manage persistent data—or, more simply, how to load and save data.

Almost endless options are available. You can store data in binary or text files on a disk. You can choose a format such as CSV, XML, JSON, YAML, or SOAP, or invent your own format. Alternatively, you can send data over the network to another

application or service, such as a relational database, an Active Directory server, or a message queue. You may even need to store data in several places, or combine all these options within a single application.

As you may begin to realize, managing persistent data is a thorny topic. Relational databases are extremely popular, but many choices, questions, and options still confront you in your daily work. For example, should you use DataSets, or are DataReaders more suitable? Should you use stored procedures? Should you hand-code your SQL or let your tools dynamically generate it? Should you strongly type your DataSets? Should you build a hand-coded domain model containing classes? If so, how do you load data to and save it from the database? Do you use code generation? The list of questions continues.

This topic isn't restricted to .NET. The entire development community has been debating this topic, often fiercely, for many years.

But one approach has gained widespread popularity: *object/relational mapping* (ORM). Over the years, many libraries and tools have emerged to help developers implement ORM in their applications. One of these is NHibernate—a sophisticated and mature object/relational mapping tool for .NET.

NHibernate is a .NET port of the popular Java Hibernate library. NHibernate aims to be a complete solution to the problem of managing persistent data when working with relational databases and domain model classes. It strives to undertake the hard work of mediating between the application and the database, leaving you free to concentrate on the business problem at hand. This book covers both basic and advanced NHibernate usage. It also recommends best practices for developing new applications using NHibernate.

Before we can get started with NHibernate, it will be useful for you to understand what persistence is and the various ways it can be implemented using the .NET framework. This chapter will explain why tools like NHibernate are needed.

Do I need to read all this background information?

No. If you want to try NHibernate right away, skip to chapter 2, where you'll jump in and start coding a (small) NHibernate application. You'll be able to understand chapter 2 without reading chapter 1, but we recommend that you read this chapter if you're new to persistence in .NET. That way, you'll understand the advantages of NHibernate and know when to use it. You'll also learn about important concepts like *unit of work*. If you're interested by this discussion, you may as well continue with chapter 1, get a broad idea of persistence in .NET, and then move on.

First, we define the notion of persistence in the context of .NET applications. We then demonstrate how a classic .NET application is implemented, using the standard persistence tools available in the .NET framework. You'll discover some common difficulties encountered when using relational databases with object-oriented frameworks such as .NET, and how popular persistence approaches try to solve these problems. Collectively,

these issues are referred to as the *paradigm mismatch* between object-oriented and database design. We then go on to introduce the approach taken by NHibernate and discuss many of its advantages. Following that, we dig into some complex persistence challenges that make a tool like NHibernate essential. Finally, we define ORM and discuss why you should use it. By the end of this chapter, you should have a clear idea of the great benefits you can reap by using NHibernate.

1.1 What is persistence?

Persistence is a fundamental concern in application development. If you have some experience in software development, you've already dealt with it. Almost all applications require persistent data. You use persistence to allow data to be stored even when the programs that use it aren't running.

To illustrate, let's say you want to create an application that lets users store their company telephone numbers and contact details, and retrieve them whenever needed. Unless you want the user to leave the program running all the time, you'll soon realize that your application needs to somehow save the contacts somewhere. You're faced with a persistence decision: you need to work out which *persistence mechanism* you want to use. You have the option of persisting your data in many places, the simplest being a text file. More often than not, you may choose a *relational database,* because such databases are widely understood and offer great features for reliably storing and retrieving data.

1.1.1 Relational databases

You've probably already worked with a relational database such as Microsoft SQL Server, MySQL or Oracle. If you haven't, see appendix A. Most developers use relational databases every day; they have widespread acceptance and are considered a robust and mature solution to modern data-management challenges.

A relational database management system (RDBMS) isn't specific to .NET, and a relational database isn't necessarily specific to any one application. You can have several applications accessing a single database, some written in .NET, some written in Java or Ruby, and so on. Relational technology provides a way of sharing data between many different applications. Even different components within a single application can independently access a relational database (a reporting engine and a logging component, for example).

Relational technology is a common denominator of many unrelated systems and technology platforms. The relational data model is often the common enterprise-wide representation of *business objects*: a business needs to store information about various things such as customers, accounts, and products (the business objects), and the relational database is usually the chosen central place where they're defined and stored. This makes the relational database an important piece in the IT landscape.

RDBMSs have SQL-based application programming interfaces (APIs). So today's relational database products are called SQL *database management systems* or, when we're talking about particular systems, SQL *databases*.

1.1.2 Understanding SQL

As with any .NET database development, a solid understanding of relational databases and SQL is a prerequisite when you're using NHibernate. You'll use SQL to tune the performance of your NHibernate application. NHibernate automates many repetitive coding tasks, but your knowledge of persistence technology must extend beyond NHibernate if you want take advantage of the full power of modern SQL databases. Remember that the underlying goal is robust, efficient management of persistent data.

If you feel you may need to improve your SQL skills, then pick up a copy of the excellent books *SQL Tuning* by Dan Tow [Tow 2003] and *SQL Cookbook* by Anthony Molinaro [Mol 2005]. Joe Celko has also written some excellent books on advanced SQL techniques. For a more theoretical background, consider reading *An Introduction to Database Systems* [Date 2004].

1.1.3 Using SQL in .NET applications

.NET offers many tools and choices when it comes to making applications work with SQL databases. You might lean on the Visual Studio IDE, taking advantage of its drag-and-drop capabilities: in a series of mouse clicks, you can create database connections, execute queries, and display editable data onscreen. We think this approach is great for simple applications, but the approach doesn't scale well for larger, more complex applications.

Alternatively, you can use SqlCommand objects and manually write and execute SQL to build DataSets. Doing so can quickly become tedious; you want to work at a slightly higher level of abstraction so you can focus on solving business problems rather than worrying about data access concerns. If you're interested in learning more about the wide range of tried and tested approaches to data access, then consider reading Martin Fowler's *Patterns of Enterprise Application Architecture* [Fowler 2003], which explains many techniques in depth.

Of all the options, the approach we take is to write classes—or *business entities*—that can be loaded to and saved from the database. Unlike DataSets, these classes aren't designed to mirror the structure of a relational database (such as rows and columns). Instead, they're concerned with solving the business problem at hand. Together, such classes typically represent the object-oriented *domain model*.

1.1.4 Persistence in object-oriented applications

In an object-oriented application, persistence allows an object to outlive the process or application that created it. The state of the object may be stored to disk and an object with the same state re-created at some point in the future.

This application isn't limited to single objects—entire graphs of interconnected objects may be made persistent and later re-created. Most objects aren't persistent; a *transient* object is one that has a limited lifetime that is bounded by the life of the process that instantiated the object. A simple example is a web control object, which exists in memory for only a fraction of a second before it's rendered to screen and flushed

from memory. Almost all .NET applications contain a mix of persistent and transient objects, and it makes good sense to have a subsystem that manages the persistent ones.

Modern relational databases provide a structured representation of persistent data, enabling sorting, searching, and grouping of data. Database management systems are responsible for managing things like concurrency and data integrity; they're responsible for sharing data between multiple users and multiple applications. A database management system also provides data-level security. When we discuss persistence in this book, we're thinking of all these things:

- Storage, organization, and retrieval of structured data
- Concurrency and data integrity
- Data sharing

In particular, we're thinking of these issues in the context of an object-oriented application that uses a domain model. An application with a domain model doesn't work directly with the tabular representation of the business entities (using DataSets); the application has its own, object-oriented model of the business entities. If a database has ITEM and BID tables, the .NET application defines `Item` and `Bid` classes rather than uses DataTables for them.

Then, instead of directly working with the rows and columns of a DataTable, the business logic interacts with this object-oriented domain model and its runtime realization as a graph of interconnected objects. The business logic is never executed in the database (as a SQL stored procedure); it's implemented in .NET. This allows business logic to use sophisticated object-oriented concepts such as inheritance and polymorphism. For example, you could use well-known design patterns such as Strategy, Mediator, and Composite [GOF 1995], all of which depend on polymorphic method calls.

Now, a caveat: Not all .NET applications are designed this way, nor should they be. Simple applications may be much better off without a domain model. SQL and ADO.NET are serviceable for dealing with pure tabular data, and the DataSet makes CRUD operations even easier. Working with a tabular representation of persistent data is straightforward and well understood.

But in the case of applications with nontrivial business logic, the domain model helps to improve code reuse and maintainability significantly. We focus on applications with a domain model in this book, because NHibernate and ORM in general are most relevant to this kind of application.

It will be useful to understand how this domain model fits into the bigger picture of a software system. To explain this, we take a step back and look at the *layered architecture*.

1.1.5 Persistence and the layered architecture

Many, if not most, systems today are designed with a layered architecture, and NHibernate works well with that design. What is a layered architecture?

A layered architecture splits a system into several groups, where each group contains code addressing a particular problem area. These groups are called *layers*. For example, a user interface layer (also called the presentation layer) might contain all

the application code for building web pages and processing user input. One major benefit of the layering approach is that you can often make changes to one layer without significant disruption to the other layers, thus making systems less fragile and more maintainable.

The practice of layering follows some basic rules:

- Layers communicate top to bottom. A layer is dependent only on the layer directly below it.
- Each layer is unaware of any other layers except the layer just below it.

Business applications use a popular, proven, high-level application architecture that comprises three layers: the presentation layer, the business layer, and the persistence layer. See figure 1.1.

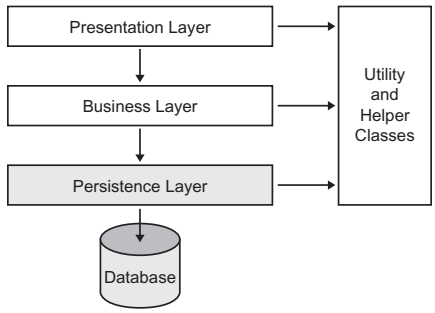


Figure 1.1 Layered architecture highlighting the persistence layer

Let's take a closer look at the layers and elements in the diagram:

- *Presentation layer*—The user interface logic is topmost. In a web application, this layer contains the code responsible for drawing pages or screens, collecting user input, and controlling navigation.
- *Business layer*—The exact form of this layer varies widely between applications. But it's generally agreed that the business layer is responsible for implementing any business rules or system requirements that users would understand as part of the problem domain. In some systems, this layer has its own internal representation of the business domain entities. In others, it reuses the model defined by the persistence layer. We revisit this issue in chapter 3.
- *Persistence layer*—The persistence layer is a group of classes and components responsible for saving application data to and retrieving it from one or more data stores. This layer defines a mapping between the business domain entities and the database. It may not surprise you to hear that NHibernate would be used primarily in this layer.
- *Database*—The database exists outside the .NET application. It's the actual, persistent representation of the system state. If a SQL database is used, the database includes the relational schema and possibly stored procedures.
- *Helper/utility classes*—Every application has a set of infrastructural helper or utility classes that support the other layers: for example, UI widgets, messaging classes, `Exception` classes, and logging utilities. These infrastructural elements aren't considered to be layers, because they don't obey the rules for interlayer dependency in a layered architecture.

Should all applications have three layers?

Although a *three-layers architecture* is common and advantageous in many cases, not all .NET applications are designed like that, nor should they be. Simple applications may be better off without complex objects. SQL and the ADO.NET API are serviceable for dealing with pure tabular data, and the ADO.NET DataSet makes basic operations even easier. Working with a tabular representation of persistent data is straightforward and well understood.

Remember that layers are particularly useful for breaking down large and complex applications, and are often overkill for the extremely simple .NET applications. For such simple programs, you may choose to put all your code in one place. Instead of neatly separating business rules and database-access functions into separate layers, you can put them all in your web/Windows code-behind files. Tools like Visual Studio .NET make it easy and painless to build this kind of application. But be aware that this approach can quickly lead to a problematic code base; as the application grows, you have to add more and more code to each form or page, and things become increasingly difficult to work with. Moreover, changes made to the database may easily break your application, and finding and fixing the affected parts can be time consuming and painful!

1.2 Approaches to persistence in .NET

We've discussed how, in any sizeable application, you need a persistence layer to handle loading and saving data. Many approaches are available when you're building this persistence layer, and each has advantages and disadvantages. Some popular choices are as follow:

- Hand coding
- DataSets
- LINQ-to-SQL
- NHibernate (or similar)
- ADO.NET Entity Framework

Despite the fact that we highly recommend NHibernate, it's always wise to consider the alternatives. As you'll soon learn, building applications with NHibernate is straightforward, but that doesn't mean it's perfect for every project. In the following sections, we examine and compare these strategies in detail, discussing the implications for database access and the user interface.

1.2.1 Choice of persistence layer

In your applications, you'll often want to load, manipulate, and save database items. Regardless of which persistence approach you use, at some point ADO.NET objects must be created and SQL commands must be executed. It would be tedious and

unproductive to write all this SQL code each time you have to manipulate data, so you can use a persistence layer to take care of these low-level steps.

The persistence layer is the set of classes and utilities used to make life easier when it comes to saving and loading data. ADO.NET lets you execute SQL commands that perform the persistence, but the complexity of this process requires that you wrap these commands behind components that understand how your entities should be persisted. These components can also hide the specifics of the database, making your application less coupled to the database and easier to maintain. For example, when you use a SQL identifier containing spaces or reserved keywords, you must delimit this identifier. Databases like SQL Server use brackets for that, whereas MySQL uses back-ticks. It's possible to hide this detail and let the persistence layer select the right delimiter.

Based on the approach you use, the internals of the persistence layer differ widely.

HAND-CODED PERSISTENCE LAYER

Hand-coding a persistence layer can involve a lot of work; it's common to first build a generic set of functions to handle database connections, execution of SQL commands, and so on. Then, on top of this sublayer, you have to build another set of functions that save, load, and find your business entities. Things get much more involved if you need to introduce caching, business-rule enforcement, or handling of entity relationships.

Hand-coding your persistence layer gives you the greatest degree of flexibility and control; you have ultimate design freedom and can easily exploit specialized database features. But it can be a huge undertaking and is often tedious and repetitive work, even when you use code generation.

DATASET-BASED PERSISTENCE LAYER

Visual Studio lets you effortlessly generate your own persistence layer, which you can then extend with new functionality with few clicks. The classes generated by Visual Studio know how to access the database and can be used to load and save the entities contained in the DataSet.

Again, a small amount of work is required to get started. You have to resort to hand-coding when you need more control, which is usually inevitable (as described in section 1.3).

NHIBERNATE PERSISTENCE LAYER

NHibernate provides all the features required to quickly build an advanced persistence layer in code. It's capable of loading and saving entire graphs of interconnected objects while maintaining the relationships between them.

In the context of an auction application (such as eBay), NHibernate lets you easily save an `Item` and its `Bid`s by implementing a method like this:

```
public void Save(Item item) {
   OpenNHibernateSession();
   session.Save(item);
   CloseNHibernateSession();
}
```

Here, `session` is an object provided by NHibernate. Don't worry about understanding the code yet. For now, we want you to see how simple the persistence layer is with

NHibernate. You'll start using NHibernate in chapter 2, where you'll discover that it's straightforward to execute persistence operations. All you need to do is write your entities and explain to NHibernate how to persist them. Before moving on to a deeper discussion of NHibernate, let's take a quick look at the newest generation of persistence technologies introduced by Microsoft.

LINQ TO SQL–BASED PERSISTENCE LAYER

Language INtegrated Query (LINQ) was introduced in 2007 by Microsoft. It allows for query and set operations, similar to what SQL statements offer for databases directly within .NET languages like C# and Visual Basic through a set of extensions to these languages. LINQ's ambition is to make queries a natural part of the programming language. LINQ to SQL provides language-integrated data access by using LINQ's extension mechanism. It builds on ADO.NET to map tables and rows to classes and objects.

LINQ to SQL uses mapping information encoded in .NET custom attributes or contained in an XML document. This information is used to automatically handle the persistence of objects in relational databases. A table can be mapped to a class, the table's columns can be mapped to properties of the class, and relationships between tables can be represented by properties. LINQ to SQL automatically keeps track of changes to objects and updates the database accordingly through dynamic SQL queries or stored procedures. Consequently, when you use LINQ to SQL, you don't have to provide the SQL queries yourself most of the time.

LINQ to SQL has some significant limitations when compared to NHibernate. For example, its mapping of classes to tables is strictly one-to-one, and it can't map base class properties to table columns. Although you can create a custom provider in LINQ, LINQ to SQL is a SQL Server–specific solution.

ADO.NET ENTITY FRAMEWORK

The Microsoft ADO.NET Entity Framework is a new approach to persistence, available since .NET 3.5 SP1. At a high level, it proposes to provide a persistence solution similar to NHibernate, but with the full commercial support and backing of Microsoft. This promises to be an attractive option for developers who require a vendor-supported solution. But at the time of this writing, the Entity Framework is early beta software, and its feature set is incomplete.

The ADO.NET Entity Framework 1.0 version supports multiple databases and more complex mapping. But it won't support true "object-first" development, where you design and build, and then generate the database tables from that mapping, until version 2—planned for late 2009 at the earliest. For situations requiring a robust ORM, NHibernate still offers significant advantages.

1.2.2 Implementing the entities

Once you've chosen a persistence-layer approach, you can focus on building the business objects, or entities, that the application will manipulate. These are classes representing the real-world elements that the application must manipulate. For an auction application, `User`, `Item`, and `Bid` are common examples. We now discuss how to implement business entities using each of the three approaches.

HAND-CODED ENTITIES

Returning to the example of an auction application, consider the entities: `User`, `Item`, and `Bid`. In addition to the data they contain, you expect relationships to exist between them. For example, an `Item` has a collection of bids, and a `Bid` refers to an `Item`; in C# classes, this might be expressed using a collection like `item.Bids` and a property like `bid.Item`. The object-oriented view is different that the relational view: instead of having primary and foreign keys, you have associations. Object-oriented design also provides other powerful modeling concepts, such as inheritance and polymorphism.

Hand-coded entities are free from any constraints; they're even free from the way they're persisted in the database. They can evolve (almost) independently and be shared by different applications; this is an important advantage when you're working in a complex environment.

But they're difficult and tedious to code. Think about the manual work required to support the persistence of entities inheriting from other entities. It's common to use code generation or base classes (like DataSet) to add features with a minimal effort. These features may be related to the persistence, transfer, or presentation of information. But without a helpful framework, these features can be time consuming to implement.

ENTITIES IN A DATASET

A DataSet represents a collection of database tables, and in turn these tables contain the data of the entities. A DataSet stores data about business objects in a fashion similar to a database. You can use a generated-typed DataSet to ease the manipulation of data, and it's also possible to insert business logic and rules.

As long as you want to manipulate data, .NET and IDEs provide most features required to work with a DataSet. But as soon as you think about business objects as *objects* in the sense of object-oriented design, you can hardly be satisfied by a DataSet (typed or not). After all, business objects represent real-world elements, and these elements have data and behavior. They may be linked by advanced relationships like inheritance, but this isn't possible with DataSets. This level of freedom in the design of entities can be achieved only by hand coding them.

ENTITIES AND NHIBERNATE

NHibernate is *non-intrusive*. It's common to use it with hand-coded (or generated) entities. You must provide mapping information indicating how these entities should be loaded and saved. Once this is done, NHibernate can take care of moving your object-oriented entities to and from a relational database.

There are many fundamental differences between objects and relational data. Trying to use them together reveals the *paradigm mismatch* (also called the *object/relational impedance mismatch*). We explore this mismatch in section 1.3. By the end of this chapter, you'll have a clear idea of the problems caused by the paradigm mismatch and how NHibernate solves these problems.

ENTITIES AND LINQ TO SQL

The LINQ to SQL approach looks a lot like the NHibernate way of doing ORM. LINQ to SQL uses POCO objects to represent your application data (the entities). The mapping of those objects to database tables is described either in declarative attributes in code or in an XML document. After the mapping and the classes are complete, the LINQ to SQL framework takes care of generating SQL for database operations.

Once the entities are implemented, you must think about how they will be presented to the end user.

1.2.3 Displaying entities in the user interface

Using NHibernate implies using entities, and using entities has consequences for the way the user interface (UI) is written. For the end user, the UI is one of the most important elements. Whether it's a web application (using ASP.NET) or a Windows application, it must satisfy the needs of the user. A deep discussion of implementing a UI isn't in the scope of this book; but the way the persistence layer is implemented has a direct effect on the way the UI will be implemented.

In this book, we refer to the UI as the *presentation layer*. .NET provides controls to display information. The simplicity of this operation depends on how the information is stored.

It's worth noting that we expect .NET entity data binding to change soon. Microsoft is beginning to actively push the use of entities in .NET applications as the company promotes the ADO.NET Entity Framework and LINQ to SQL. For this reason, we won't discuss those technologies in this section.

DATASET-BASED PRESENTATION LAYER

Microsoft has added support for data binding with DataSet in most .NET controls. It's easy to bind a DataSet to a control so that its information is displayed and any changes (made by the user) reverberate in the DataSet.

Using DataSets is probably the most productive way to implement a presentation layer. You may lose some control over how information is presented, but it's good enough in most cases. The situation is more complicated with hand-coded entities.

PRESENTATION LAYER AND ENTITIES

Data-binding the hand-coded entities typically used in NHibernate applications can be tricky. This is because entities can be implemented in so many different ways. A DataSet is always made of tables, columns, and rows; but a hand-coded entity—a class of your own design—contains fields and methods with no standardized way to access and display them. .NET allows us to data-bind controls to the public properties of any object. This is good enough in simple cases. If you want more flexibility, you must do some hand coding to get entity data into the UI and back again.

Hand coding your own entity/UI bindings is still fairly simple. However, if you find this tedious, then take a look at some of the open source projects designed to tackle this problem for you. "ObjectViews" is one of many projects out there.

Also, don't forget that you're free to fall back to DataSets when you're dealing with edge cases like complex reporting, where DataSets are much easier to manipulate. In fact, at the time of writing, few reporting tools provide good support for entities, so DataSets may be your best option. We discuss this issue in chapter 9.

Using persistence-able information affects the way the UI is designed. Data should be loaded when the UI opens and saved when the UI closes. NHibernate proposes some patterns to deal with this process, as you'll learn in chapter 8.

Now all the layers are in place, and you can work on performing actions.

1.2.4 Implementing CRUD operations

When you're working with persistent information, you're concerned with persisting and retrieving this information. Create, read, update, delete (CRUD) operations are primitive operations executed even in the simplest application. Most of the time, these operations are triggered by events raised in the presentation layer. For example, the user may click a button to view an item. The persistence layer is used to load this item, which is then bound to a form displaying its data.

No matter which approach you use, these primitive operations are well understood and easy to implement. Operations that are more complex are covered in the next section.

HAND-CODED CRUD OPERATIONS

A hand-coded CRUD operation does exactly what you want because you write the SQL command to execute—but it's repetitive and annoying work. It's possible to implement a framework that generates these SQL commands. Once you understand that loading an entity implies executing a `SELECT` on its database row, you can automate primitive CRUD operations. But much more work is required for complex queries and manipulating interconnected entities.

CRUD OPERATIONS WITH DATASETS

You know that much of the persistence layer can be generated when using DataSets. This persistence layer contains classes to execute CRUD operations. And Visual Studio 2005 and .NET 2.0 come with more powerful classes called *table adapters*.

Not only do these classes support primitive CRUD operations, but they're also extensible. You can either add methods calling stored procedures or generate SQL commands with few clicks. But if you want to implement anything more complex, you must hand code it; you'll see in the next section that some useful features aren't easy to implement, and the structure of a DataSet may make doing so even harder.

CRUD OPERATIONS USING NHIBERNATE

As soon as you give NHibernate your entities' mapping information, you can execute a CRUD operation with a single method call. This is a fundamental feature of an ORM tool. Once it has all the information it needs, it can solve the object/relational impedance mismatch at each operation.

NHibernate is designed to efficiently execute CRUD operations. Experience and tests have helped uncover many optimizations and best practices. For example, when you're

manipulating entities, you can achieve the best performance by delaying persistence to the end of the *transaction*. At this point, you use a single connection to save all entities.

LINQ TO SQL CRUD OPERATIONS

On the surface, executing CRUD operations with LINQ to SQL is similar to using NHibernate—you can load, save, update, and delete objects with simple method calls. LINQ to SQL offers less fine tuning of your CRUD operations for each entity, which can be a good thing or a bad thing depending on the complexity of your project.

Now that we've covered all the basic persistence steps and operations, we explore some advanced features that illustrate the advantages of NHibernate.

1.3 Why do we need NHibernate?

So far, we've talked about a simple application. In the real world, you rarely deal with simple applications. An enterprise application has many entities with complex business logic and design goals: productivity, maintainability, and performance are all essential.

In this section, we walk through some features indispensable to implementing a successful application. First, we give some examples illustrating the fundamental differences between objects and relational database. You'll also learn how NHibernate helps create a bridge between these representations. Then, we turn to the persistence layer to discover how NHibernate deals with complex and numerous entities. You'll learn the patterns and features it provides to achieve the best performance. Finally, we cover complex queries; you'll see that you can use NHibernate to write a powerful search engine.

Let's start with the entities and their mapping to a relational database.

1.3.1 The paradigm mismatch

A database is relational, but we're using object-oriented languages. There is no direct way to persist an object as a database row. Persistence shouldn't hinder your ability to design entities that correctly represent what you're manipulating.

The *paradigm mismatch* (or *object/relational impedance mismatch*) refers to the fundamental incompatibilities that exist between the design of objects and relational databases. Let's look at some of the problems created by the paradigm mismatch.

THE PROBLEM OF GRANULARITY

Granularity refers to the relative size of the objects you're working with. When we talk about .NET objects and database tables, the granularity problem means persisting objects that can have various kinds of granularity to tables and columns that are inherently limited in granularity.

Let's take an example from the auction use case we mentioned in section 1.1.4. Let's say you want to add an address to a `User` object, not as a string but as another object. How can you persist this user in a table? You can add an ADDRESS table, but it's generally not a good idea (for performance reasons). You can create a *user-defined column type*, but this option isn't broadly supported and portable. Another option is to merge the address information into the user, but this isn't a good object-oriented design and it isn't reusable.

It turns out that the granularity problem isn't difficult to solve. We wouldn't even discuss it if it weren't for the fact that it's visible in so many approaches, including the DataSet. We describe the solution to this problem in section 3.6.

A much more difficult and interesting problem arises when we consider inheritance, a common feature of object-oriented design.

THE PROBLEM OF INHERITANCE AND POLYMORPHISM

Object-oriented languages support the notion of *inheritance*, but relational databases typically don't. Let's say that the auction application can have many kinds of items. You could create subclasses like `Furniture` and `Book`, each with specific information. How can you persist this hierarchy of entities in a relational database? A `Bid` can refer to any subclass of `Item`. It should be possible to run *polymorphic queries*, such as retrieving all bids on books. In section 3.8, we discuss how ORM solutions like NHibernate solve the problem of persisting a class hierarchy to a database table or tables.

THE PROBLEM OF IDENTITY

The identity of a database row is commonly expressed as the *primary key* value. As you'll see in section 3.5, .NET *object identity* isn't naturally equivalent to the primary key value. With relational databases, it's recommended that you use a *surrogate key*—a primary key column with no meaning to the user. But .NET objects have an intrinsic identity, which is based either on their memory location or on a user-defined convention (by using the implementation of the `Equals()` method).

Given this problem, how can you represent associations? Let's look at that next.

PROBLEMS RELATING TO ASSOCIATIONS

In an object model, *associations* represent the relationships between objects. For instance, a bid has a relationship with an item. This association is created using object references. In the relational world, an association is represented by a *foreign key column*, with copies of key values in several tables. There are subtle differences between the two representations.

Object references are inherently directional: the association is from one object to the other. If an association between objects should be navigable in both directions, you must define the association *twice*, once in each of the associated classes.

On the other hand, foreign-key associations aren't by nature directional. Navigation has no meaning for a relational data model, because you can create arbitrary data associations with table joins and projection. We discuss association mappings in detail in chapters 3 and 6.

If you think about the DataSet in all these problems, you'll realize how rigid its structure is. The information in a DataSet is presented exactly as in the database. To navigate from one row to another, you must manually resolve their relationship by using a *foreign key* to find the referred row in the related table. Let's move from the representation of the entities to how you can manipulate them efficiently.

1.3.2 Units of work and conversations

When users work on applications, they perform distinct unitary operations. These operations can be referred to as *conversations* (or *business transactions* or *application*

transactions). For example, placing a bid on an item is a conversation. Seasoned programmers know how hard it can be to make sure that many related operations performed by the user are treated as if they were a single bigger business transaction (a *unit*). You'll learn in this section that NHibernate makes this easier to achieve. Let's take another example to illustrate this concept.

Popular media players allow you to rate the songs you hear and later sort them based on your rating. This means your ratings are persisted. When you open a list of songs, you listen and rate them one by one. When should persistence take place?

The first solution that may come to mind is to persist the rating when the user enters it. This technique is inefficient: the user may change the rating many times, and the persistence will be done separately for each song. (But this approach is safest if you expect the application to crash at any moment.)

Instead, you can let the user rate all the songs and then persist the ratings when the user closes the list. The process of rating these songs is a conversation.

Let's see how it works and what its benefits are.

THE UNIT OF WORK PATTERN

When you're working with a relational database, you may tend to think of commands: saving or loading. But an application can perform operations involving many entities. When these entities are loaded or saved depends on the context.

For example, if you want to load the last item created by a user, you must first save the user (and the user's collection of items); then you can run a query retrieving the item. If you forget to save the user, you'll start getting hard-to-detect bugs.

> **The Identity Map pattern**
>
> NHibernate uses the *Identity Map* pattern to make sure an item's user is the same object as the user you had before loading the item (as long as you're working in the same transaction). You'll learn more about the concept of identity in section 3.5.

Now imagine that you're involved in a complex conversation involving many updates and deletes. If you have to manually track which entities to save or delete, while making sure you load each entity only once, things can quickly become very difficult.

NHibernate follows the Unit of Work pattern to solve this problem and ease the implementation of conversations. (We cover conversations in chapter 5 and implement them in chapter 10.)

You can create entities and associate them with NHibernate; then, NHibernate keeps track of all loading and saving of changes only when required. At the end of the transaction, NHibernate figures out and applies all changes in their correct order.

TRANSPARENT PERSISTENCE AND LAZY LOADING

Because NHibernate keeps track of all entities, it can greatly simplify your application and increase the application's performance. Here are two simple examples.

When working on an item in the auction application, users can add, modify, or delete their bids. It would be painful to manually track these changes one by one.

Instead, you can use NHibernate's *transparent persistence* feature: you ask NHibernate to save all changes in the collection of bids when the item is persisted. It automatically figures out which CRUD operations must be executed.

Now, if you want to modify a `User`, you load, change, and persist it. But what about the collection of items this user has? Should you load these items or leave the collection un-initialized? Loading the items would be inefficient, but leaving the collection un-initialized will limit your ability to manipulate the user.

NHibernate support a feature called *lazy loading* to solve this problem. When loading the user, you can decide between loading the items or not. If you choose not to do so, the collection is transparently initialized when you need it.

Using these features has many implications; we progressively cover them in this book.

CACHING

Tracking entities implies keeping their references somewhere. NHibernate uses a *cache*. This cache is indispensable for implementing the Unit of Work pattern, and it can also make applications more efficient. We cover caching in depth in section 5.3.

NHibernate's identity map uses a cache to avoid loading an entity many times. This cache can be shared by transactions and applications.

Suppose you build a website for the auction application. Visitors may be interested in some items. Without a cache, these items will be loaded from the database each time a visitor wants to see them. With a few lines of code, you can ask NHibernate to cache these items, and then enjoy the performance gain.

1.3.3 Complex queries and the ADO.NET Entity Framework

This is the last (but not least) feature related to persistence. In section 1.2.5, we talked about CRUD operations. You've learned about features related to CRUD (all having to do with the Unit of Work pattern). Now we talk about retrieve operations: searching for and loading information.

You can easily generate code to load an entity using its identifier (its primary key, in the context of a relational database). But in real-world applications, users rarely deal with identifiers; instead, they use criteria to run a search and then pick the information they want.

IMPLEMENTING A QUERY ENGINE

If you're familiar with SQL, you know that you can write complex queries using the `SELECT ... FROM ... WHERE ...` construct. But if you work with business objects, you have to transform the results of your SQL queries into entities. We already advertised the benefits of working with entities, so it makes more sense to take advantage of those benefits even when querying the database.

Based on the fact that NHibernate can load and save entities, we can deduce that it knows how each entity is mapped to the database. When you ask for an entity by its identifier, NHibernate knows how to find it. You should be able to express a query using entity names and properties, and then NHibernate should be able to convert that into a corresponding SQL query understood by the relational database.

NHibernate provides two query APIs:

- *Hibernate Query Language* (HQL) is similar to SQL in many ways, but it also has useful object-oriented features. You can query NHibernate using plain old SQL; but as you'll learn, using HQL offers several advantages.
- *Query by Criteria API* (QBC) provides a set of type-safe classes to build queries in your chosen .NET language. This means that if you're using Visual Studio, you'll benefit from the inline error reporting and IntelliSense.

To give you a taste of the power of these APIs, let's build three simple queries. First, here is some HQL. It finds all bids for items where the seller's name starts with the letter *K*:

```
from Bid bid
where bid.Item.Seller.Name like 'K%'
```

As you can see, this code is easy to understand. If you want to write SQL to do the same thing, you need something more verbose, along these lines:

```
select B.*
from BID B
inner join ITEM I on B.ITEM_ID = I.ITEM_ID
inner join USER U on I.AUTHOR_ID = U.USER_ID
where U.NAME like 'K%'
```

To illustrate the power of the Query by Criteria API, we use an example derived from one later in the book, in section 8.5.1. This shows a method that lets you find and load all users who are similar to an example user, and who also have a bid item similar to a given example item:

```
public IList<User> FindUsersWithSimilarBidItem(User u, Item i) {
   Example exampleUser =
     Example.Create(u).EnableLike(MatchMode.Anywhere);
   Example exampleItem =
     Example.Create(i).EnableLike(MatchMode.Anywhere);
   return GetSession().CreateCriteria(typeof(User))
         .Add( exampleUser )
            .CreateCriteria("Items")
               .Add( exampleItem )
         .List<User>();
}
```

This method lets you pass objects that represent the kind of users you want NHibernate to find and load. It creates two NHibernate `Example` objects and uses the Query by Criteria API to run the query and retrieve a list of users. The notion of an example entity (here, example `User` and example `Item`) is both powerful and elegant, as demonstrated here:

```
User u = new User();
Item i = new Item();
u.Name = "K";
i.State = ItemState.Active;
i.ApprovedBy = administratorUser;
List<User> result = FindUsersWithSimilarBidItem(u, i);
```

You use the `FindUsersWithSimilarBidItem` method to retrieve users whose names contain *K* and who are selling an active bid `Item`, which has also been approved by the administrator. Quite a feat for so little code! If you're new to this approach, you may find it unbelievable. Don't even try to implement this query using hand-coded SQL.

You'll learn more about queries in chapters 5 and 7. If you aren't fully satisfied by these APIs, you may also want to watch for upcoming developments that allow LINQ to be used with NHibernate.

ADO.NET ENTITY FRAMEWORK

At the time of this writing, Microsoft is working on its next-generation data-access technology, which introduces a number of interesting innovations. You may think this technology will soon replace NHibernate, but this is unlikely. Let's see why.

Perhaps the most exciting new feature is a powerful query framework code-named LINQ. LINQ extends your favorite .NET language so that you can run queries against various types of data source without having to embed query strings in your code. When querying a relational database, you can do something like this:

```
IEnumerable users = from u in Users
       where u.Lastname.StartsWith("K")
             order by user.Lastname descending
       select u;
```

As you can see, the queries are type-safe and allow you to take advantage of many .NET language features. One key aspect of LINQ is that it gives you a declarative way of working with data, so you can express what you want in simple terms rather than typing lots of for-each loops. You can also benefit from helpful IDE capabilities such as autocompletion and parameter assistance. This is a big win for everybody.

Because LINQ is designed to be extensible, other tools such as NHibernate can integrate with this technology and benefit from it. At the time of writing, good progress is being made toward a LINQ to NHibernate project. And Manning Publications has published *LINQ in Action*, a fantastic book by our good friend Fabrice Marguerie.

As mentioned earlier, Microsoft is also working on a framework currently called the ADO.NET Entity Framework, which aims to provide developers with an ORM framework not completely unlike NHibernate. This is a good step forward because Microsoft will promote the DataSet less often and begin promoting the benefits of ORM tools. Another project called LINQ over DataSet greatly improves DataSet's query capabilities, but it doesn't yet solve many other issues discussed in this chapter.

All these technologies will take time to mature. Many questions remain unanswered, such as, how extensible will this framework be? Will it support most popular RDBMSs or just SQL Server? Will it be easy to work with legacy database schemas? No framework can provide all features, so it must be extendable to let you integrate your own features. (If your particular projects require you to work with legacy databases, you can read section 10.2 to learn about the features NHibernate gives you to work with more exotic data structures.)

Now, let's dig into the theory behind NHibernate.

1.4 Object/relational mapping

You already have an idea of how NHibernate provides object/relational persistence. But you may still be unable to tell what ORM is. We try to answer this question now. After that, we discuss some nontechnical reasons to use ORM.

1.4.1 What is ORM?

Time has proven that relational databases provide a good means of storing data, and that object-oriented programming is a good approach to building complex applications. With object/relational mapping, it's possible to create a translation layer that can easily transform objects into relational data and back again. As this bridge will manipulate objects, it can provide many of the features we need (like caching, transaction, and concurrency control). All we have to do is provide information on how to map objects to tables.

Briefly, object/relational mapping is the automated (and possibly transparent) persistence of objects in an application to the tables in a relational database, using metadata that describes the mapping between the objects and the database. ORM, in essence, works by transforming data from one representation to another.

> **Isn't ORM a Visio plug-in?**
>
> The acronym ORM can also mean *object role modeling*, and this term was invented before object/relational mapping became relevant. It describes a method for information analysis, used in database modeling, and is primarily supported by Microsoft Visio, a graphical modeling tool. Database specialists use it as a replacement or as an addition to the more popular entity-relationship modeling. But if you talk to .NET developers about ORM, it's usually in the context of object/relational mapping.

You learned in section 1.3.1 that there are many problems to solve when using ORM. We refer to these problems as the paradigm mismatch. Let's discuss, from a non-technical point of view, why we should face this mismatch and use an ORM tool like NHibernate.

1.4.2 Why ORM?

The overall solution for mismatch problems can require a significant outlay of time and effort. In our experience, the main purpose of up to 30 percent of the .NET application code written is to handle tedious SQL/ADO.NET and manual bridging of the object/relational paradigm mismatch. Despite all this effort, the end result doesn't feel right. We've seen projects nearly sink due to the complexity and inflexibility of their database abstraction layers.

MODELING MISMATCH

One of the major costs is in the area of modeling. The relational and object models must both encompass the same business entities. But an object-oriented purist will model these entities differently than an experienced relational data modeler. You

learned some details of this problem in section 1.3.1. The usual solution is to bend and twist the object model until it matches the underlying relational technology.

This can be done successfully, but only at the cost of losing some of the advantages of object orientation. Keep in mind that relational modeling is underpinned by relational theory. Object orientation has no such rigorous mathematical definition or body of theoretical work. No elegant transformation is waiting to be discovered. (Doing away with .NET and SQL and starting from scratch isn't considered elegant.)

PRODUCTIVITY AND MAINTAINABILITY

The domain-modeling mismatch isn't the only problem solved by ORM. A tool like NHibernate makes you more productive. It eliminates much of the grunt work (more than you'd expect) and lets you concentrate on business problems. No matter which application-development strategy you prefer—top-down, starting with a domain model; or bottom-up, starting with an existing database schema—NHibernate used together with the appropriate tools will significantly reduce development time.

Using fewer lines of code makes the system more understandable because it emphasizes business logic rather than plumbing. Most important, a system with less code is easier to refactor. NHibernate substantially improves maintainability, not only because it reduces the number of lines of code, but also because it provides a buffer between the object model and the relational representation. It allows a more elegant use of object orientation on the .NET side, and it insulates each model from minor changes to the other.

PERFORMANCE

A common claim is that hand-coded persistence can always be at least as fast, and often faster, than automated persistence. This is true in the same sense that it's true that assembly code can always be at least as fast as .NET code—in other words, it's beside the point.

The unspoken implication of the claim is that hand-coded persistence will perform at least as well in an application. But this implication will be true only if the effort required to implement at-least-as-fast hand-coded persistence is similar to the amount of effort involved in utilizing an automated solution. The interesting question is, what happens when we consider time and budget constraints?

The best way to address this question is to define a means to measure performance and thresholds of acceptability. Then you can find out whether the performance cost of an ORM is unacceptable. Experience has proven that a good ORM has a minimal impact on performance. It can even perform better than classic ADO.NET when correctly used, due to features like caching and batching. NHibernate is based on a mature architecture that lets you take advantage of many performance optimizations with minimal effort.

DATABASE INDEPENDENCE

NHibernate abstracts your application away from the underlying SQL database and SQL dialect. The fact that it supports a number of different databases confers a level of portability on your application.

You shouldn't necessarily aim to write totally database-independent applications, because database capabilities differ and achieving full portability would require sacrificing some of the strength of the more powerful platforms. But an ORM can help mitigate some of the risks associated with vendor lock-in. In addition, database independence helps in development scenarios where you use a lightweight local database but deploy for production on a different database platform.

1.5 Summary

In this chapter, we've discussed the concept of object persistence and the importance of NHibernate as an implementation technique. Object persistence means that individual objects can outlive the application process; they can be saved to a data store and be re-created later. We've walked through the layered architecture of a .NET application and the implementation of persistence, exploring four possible approaches.

You now understand the productivity of DataSet, but you also realize how limited and rigid it is. You've learned about many useful features that would be painful to hand code. In addition, you know how NHibernate solves the object/relational mismatch.

This mismatch comes into play when the data store is a SQL-based RDBMS. For instance, a graph of richly typed objects can't be saved to a database table; it must be disassembled and persisted to columns of portable SQL data types.

We glanced at NHibernate's powerful query APIs. After you've started using them, you may never want to go back to SQL.

Finally, you learned what ORM is. We discussed, from a non-technical point of view, the advantages of using this approach.

ORM isn't a silver bullet for all persistence tasks; its job is to relieve the developer of 95 percent of object persistence work, such as writing complex SQL statements with many table joins and copying values from ADO.NET result sets to objects or graphs of objects. A full-featured ORM middleware like NHibernate provides database portability, certain optimization techniques like caching, and other functions that aren't easy to hand code in a limited time with SQL and ADO.NET.

It's likely that a better solution than ORM will exist some day. We (and many others) may have to rethink everything we know about SQL, persistence API standards, and application integration. The evolution of today's systems into true relational database systems with seamless object-oriented integration remains pure speculation. But we can't wait, and there is no sign that any of these issues will improve soon (a multibillion-dollar industry isn't agile). ORM is the best solution currently available, and it's a timesaver for developers facing the object/relational mismatch every day.

We've given you background on the reasons behind ORM, the critical issues that must be addressed, and the tools and approaches available with .NET for addressing them. We've explained that NHibernate is a fantastic ORM tool that lets you combine the benefits of both object orientation and relational databases simultaneously. The next step is to give you a hands-on look at NHibernate so you can see how to use it in your projects. That's where chapter 2 comes in.

.NET PROGRAMMING

NHibernate IN ACTION

Pierre Henri Kuaté • Tobin Harris • Christian Bauer • Gavin King

FOREWORD BY Ayende Rahien NHibernate Committer

Efficient and secure data access is key for software applications. Microsoft is promoting object/relational mapping to help achieve this, but is yet to offer a complete solution. In the meantime, with NHibernate you get an open source object/relational mapper that is based on the hugely popular Java Hibernate project. With a wealth of features and a straightforward setup, you can build best-practice enterprise .NET applications with less effort and less time.

NHibernate in Action is a carefully crafted guide that introduces NHibernate and ORM to .NET developers. You'll learn to map information between business objects and database tables and explore NHibernate's internal architecture. A complete example demonstrates entity and relationship mappings and CRUD operations, along with advanced techniques like caching, concurrency access, and isolation levels. The book shows you how to refactor to a layered architecture. It also discusses patterns for integrating services and for crossing distribution boundaries.

What's Inside

- Object/relational mapping for .NET
- NHibernate configuration, mapping, and query APIs
- Data-binding objects to .NET GUI controls
- Advanced session management and distributed transactions

About the Authors

Pierre Henri Kuaté is a developer on the NHibernate project team and author of the NHibernate.Mapping.Attributes library. Tobin Harris is an independent consultant and founder of the Open Source SqlBuddy project. Gavin King and Christian Bauer are the authors of Hibernate in Action and original founders of the Hibernate project.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/NHibernateinAction

MANNING US/CAN $49.99 [INCLUDING EBOOK]

"A much needed book for novices and experts"

—Mark Monster, Rubicon

"Finally, a great book covering NHibernate"

—Paul Wilson, McKesson

"A must-have for NHibernate developers"

—Ayende Rahien
NHibernate Committer

"Accelerates your learning curve for ORM in .NET."

—Doug Warren
Java Web Services

ISBN-13: 978-1932394924
ISBN-10: 1932394923