Learning NHibernate 4

Connecting the object-oriented world of .NET to the relational world of databases has always been fiddly but with the onset of ORMs such as NHibernate, developers have finally got some relief in this area.

You will begin with a bird’s eye view of NHibernate, touching upon its core concepts. You will define domain model and map it with database schema using different techniques. You will then look into multiple ways of storing domain entities in a database and learn important concepts such as transitive persistence, transactions and unit of work. This will be followed by retrieving data from database. Next, you will optimize your code, explore concepts such as the onion architecture, and learn where NHibernate fits best in an application’s architecture. After introducing a well-known repository pattern into our application, you will learn to deal with legacy databases. We will conclude with infrequently used features such as stateless sessions, the second level cache, concurrency, and so on, which might come handy.

Who this book is written for
This book targets .NET developers who have never used an ORM before, developers who have used an ORM before but are new to NHibernate, or have used NHibernate sparingly and want to learn more about NHibernate.

What you will learn from this book
- Map domain entities to a database schema using the different mapping mechanisms available
- Configure NHibernate through XML configuration
- Save, update, and delete entities in the database and query data from a database using different querying methods
- Optimize database operations for speed and memory consumption
- Use NHibernate in real-life software projects
- Get to know about data access patterns such as repository, specification, and query object
- Use NHibernate with legacy databases

Who this book is written for
This book targets .NET developers who have never used an ORM before, developers who have used an ORM before but are new to NHibernate, or have used NHibernate sparingly and want to learn more about NHibernate.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction to NHibernate'
- A synopsis of the book’s content
- More information on Learning NHibernate 4
Suhas Chatekar has been in the business of building software for 12 years. He has mainly worked on .NET but possesses the skills of a full-stack developer. He started his career as a software developer, working on a trading platform. Since then, he has worked in several different domains, building both desktop and web-based applications. He is very passionate about coding and learning. He believes that practices such as test-driven development, automated testing, continuous deployment, and Agile are at the heart of quality software development. He currently works as a technical architect and mentor for a London-based company, working in the fields of loyalty, assistance, and insurance.

He has used NHibernate in various commercial projects, including both greenfield and brownfield. He also has experience in using NHibernate along with other ORMs and data retrieval techniques in large code bases. He is especially fond of clean code and following the right architectural guidelines. Through years of experience of working with NHibernate, he believes he has acquired some knowledge about best architectural principles that help in building resilient data access code using NHibernate, which he can share with readers.

In his spare time, he explores new technologies and writes a blog. He lives in London with his wife, Snehal, and daughter, Ovee. You can reach out to Suhas on Twitter using his handle at suhas_chatekar.
Preface

Connecting the object-oriented world of .NET to the relational world of databases has always been a fiddly issue. With the onset of ORMs, such as NHibernate, developers have got some relief in this area. However, using NHibernate without understanding its core features, and how they work, can always lead to more fiddly code and frustration. Learning how to use NHibernate in practical scenarios with the help of proven patterns and techniques goes a long way to building a robust data access layer. Knowing the best practices of designing a data access layer with NHibernate, before you start your next project, will help you to avoid any problems and deliver a successful project.

What this book covers

Chapter 1, Introduction to NHibernate, provides a view of NHibernate from 10,000 feet above and touches upon topics such as what an ORM is and why developers should use it. New features of NHibernate 4.0 are examined in this chapter along with the basic building blocks of NHibernate.

Chapter 2, Let’s Build a Simple Application, describes a problem statement and builds a very simple domain model, so that we have a set of known business requirements to refer to in the rest of the book. This chapter also covers different software and tools that we will use throughout the book.

Chapter 3, Let’s Tell NHibernate About Our Database, takes a deep dive into one of the building blocks of NHibernate — mapping. The chapter starts with a brief introduction to the different mechanisms of mapping supported by NHibernate. The rest of the chapter goes into the details of mapping classes, properties, identifiers, associations, and inheritance hierarchies. In this chapter, the domain model defined in Chapter 2, Let’s Build a Simple Application, is mapped to an appropriate database schema.
Preface

Chapter 4, *NHibernate Warm-up*, walks through the configuration of NHibernate. The chapter starts with a discussion on the different ways of configuring NHibernate and the minimal configurations required to get it up and running. Many important, but optional, configurations that include caching, session context, batch size, and so on, are covered next. The chapter ends with a discussion on generating database creation and updating scripts using NHibernate mappings.

Chapter 5, *Let’s Store Some Data into the Database*, in a nutshell covers the NHibernate API that saves data in a database. Beyond this, this chapter covers important concepts such as transactions, flush modes, cascades, and inverse collections. This chapter explains how transitive persistence works in different scenarios. Entity equality and its implementation in the context of NHibernate is provided in this chapter.

Chapter 6, *Let’s Retrieve Some Data from the Database*, as the name suggests, talks about retrieving data from the database using the different querying methods supported by NHibernate. The book stresses using LINQ but other methods, namely native SQL, HQL, Criteria API and QueryOver, are also introduced and covered in detail wherever required. The chapter goes into the details of important concepts such as joins, lazy loading, eager fetching, and polymorphic queries. Basic querying requirements such as sorting and ordering the results are explained in this chapter.

Chapter 7, *Optimizing the Data Access Layer*, talks about making the NHibernate code more efficient by making use of techniques such as batching, different fetching strategies, and avoiding “select N+1” and eager fetching.

Chapter 8, *Using NHibernate in a Real-world Application*, introduces some architectural principles such as onion architecture and dependency injection that will help in building a robust data access layer using NHibernate in production code. This chapter covers important concepts, such as unit of work and session per request along with their implementation in NHibernate. Repository pattern, the most commonly used pattern for abstracting the low level details of data access code, is also covered in this chapter.

Chapter 9, *Advanced Data Access Patterns*, begins with a discussion on the shortcomings of repository pattern. The chapter then introduces two data access patterns, namely specification pattern and query object pattern, which address the shortcomings of repository pattern.

Chapter 10, *Working with Legacy Database*, goes into the NHibernate features that come handy in legacy database situations — where database schema is not ideal for the domain model at hand. The chapter talks about composite IDs, components, working with database views, stored procedures, and custom DTOs. Custom lazy loading retention is introduced, so that it can be used with DTOs when there is a chance that NHibernate's lazy loading is rendered ineffective.
Chapter 11, *A Whirlwind Tour of Other NHibernate Features*, talks about the features of NHibernate that are slightly advanced and are not touched upon in any of the previous chapters. Important features such as concurrency control, event system, caching, and user-defined types are covered along with practical examples of their usage.
Introduction to NHibernate

In this chapter, we will go over a very brief introduction to NHibernate. This chapter is mainly aimed at absolute beginners to either NHibernate or ORMs. Therefore, we will begin with a discussion around what is an ORM and what kind of problems it solves. We will then dive into NHibernate and talk about what NHibernate offers in specific. Next, we would talk about the latest version of NHibernate and what new features are added in the latest version. While I am introducing NHibernate to you, I will also touch upon the important building blocks of NHibernate. In the end, I would try to answer the question of whether it is a bad idea to use ORM, as portrayed by some experts.

For those of you who have used or read about Entity Framework (EF), I have added a section talking about features in NHibernate that are missing in the current version of EF. This is no way tries to present a picture that NHibernate is superior that EF. Rather, this information will help you to choose the right tool for the job.

What is ORM?

ORM stands for Object Relational Mapping. It is a technique that lets you map objects to relational databases and vice versa. Here, the term "object" mostly means an instance of a class. Tools that make use of ORM techniques are also referred to as ORM, which stands for Object Relational Mapper. NHibernate is one such tool. Let's take a look at a very simple example in order to understand what exactly this technique does. Suppose that you have the following Customer class:

```csharp
public class Customer
{
    public string FirstName {get; set;}
    public string LastName {get; set;}
    public string EmailAddress {get; set;}
    public int Age {get; set;}
}
```
Every customer will be represented by an instance of the preceding class in the memory. Every such instance will also be persisted in the database. Suppose that the following database table is used to store customer instances:

```sql
CREATE TABLE tblCustomer
(
    FirstName NVARCHAR(100),
    LastName NVARCHAR(100),
    EmailAddress NVARCHAR(100),
    Age INT
);
```

The preceding syntax is for the MS SQL Server database. If you are using a different RDBMS, then the syntax may differ.

Depending on which ORM you are using, you will have a way to tell your ORM that the `Customer` class corresponds to the `tblCustomer` table in your database. Similarly, you can tell that the `FirstName`, `LastName`, `EmailAddress`, and `Age` properties map to columns with the same names in the `tblCustomer` table. Once all of this is in place, you can just tell your ORM to save an instance of the `Customer` class; it will ensure that a record is inserted into the `tblCustomer` table with the appropriate values for all the columns.

In a nutshell, that is ORM for you. ORM will handle all your database CRUD operations for you without you having to write a single line of SQL script. However, the most important principle to understand around an ORM is that it tries to bridge the gap between the OO world and the relational world. Programs written using the object-oriented principles adhere to a set of rules and support a particular set of data types. On the other hand, most RDBMSs follow rules derived from the set theory and support a set of data types that may not all be compatible with the corresponding data type on the OO side of the world. Besides this, there are differences in how new objects are constructed, how they associate with each other, what kind of operations are permitted on them, and so on.
All these differences make working with databases difficult when viewed from the perspective of the OO program. These differences are also called **impedance mismatch**, a term taken from electrical engineering. Impedance gives a measure of how easily the current can flow through an electrical circuit. For two electrical circuits to work in coherence with each other when connected, their impedances should match. In the same way, for a program written in OOP to work with an RDBMS in coherence, the impedance between them has to be matched. An ORM does this job.

**What is NHibernate?**

Obviously, it is an ORM. If I may quote from the NHibernate website:

"NHibernate is a mature, open source object-relational mapper for the .NET framework. It's actively developed, fully featured and used in thousands of successful projects."

NHibernate is a .NET port of a very famous ORM from the Java land called Hibernate. Now, Hibernate is quite old and is used by thousands of production applications. NHibernate has not only inherited most of the good features from its Java ancestor but has also been enriched with features from .NET, such as LINQ queries. To understand how powerful it can be, let's take a very simple example. If you have the `Customer` class from the previous section, mapped to a database table called `tblCustomer`, and you want to fetch all customers between 28 and 35 years of age, then with NHibernate, you can write something as follows:

```csharp
var customers = from c in session.Query<Customer>()
    where c.Age > 28 && c.Age < 35
    select c;
```

This is an LINQ query written in C#. **LINQ** stands for **Language-Integrated Query**. This is a feature introduced in C# and Visual Basic that offers data query and update syntax, which is similar to SQL. LINQ works against a variety of data sources but within the constraints of the C# language; further, it works nicely with the `IQueryable` types. If you are not familiar with LINQ, then do not worry too much as we will cover them in the coming chapters. You must be wondering what is `session.Query<Customer>()`? This is an NHibernate API that we will use extensively throughout this book. For now, `session.Query<Customer>()` returns `IQueryable<Customer>`. It is this queryable type that makes writing the previous LINQ query possible.
NHibernate then processes the previous LINQ query and generates the following SQL:

```sql
SELECT a_.FirstName,
a_.LastName,
a_.EmailAddress,
a_.Age
FROM tblCustomer a_
WHERE a_.Age > 28 AND a_.Age < 35
```

As you can see, all you wrote was a simple C# LINQ expression against a C# class. You did not have to think about what the database table name is and how to form the correct SQL query; NHibernate has handled all of that for you. NHibernate has a lot of such features that will make writing database interaction code a breeze. We are going to cover these features of NHibernate throughout this book.

NHibernate is an open source project which prospers entirely on community contributions. As with any open source project, this gives a lot of transparency to every aspect of NHibernate. You can download the source code from their GitHub repository at https://github.com/nhibernate/nhibernate-core/ and check out how things are implemented. You can raise bugs on their official JIRA board, contribute to discussions around the right fixes for the defects, development of new features, and the future roadmap. All these activities are mostly led by a few top contributors, but transparency and community inclusion are what keep NHibernate going. You can access the NHibernate JIRA board by navigating to http://nhibernate.jira.com. I will not spend too much space talking about NHibernate here. Besides the fact that the whole book covers all important aspects of NHibernate anyway, I am now going to peek into some interesting NHibernate features in the remaining sections of this chapter.

Before I move onto the next section, I would like to mention that the latest stable version of NHibernate is 4.0.3.400 as of the time of writing of this book. The latest version of NHibernate can be confirmed by visiting the NuGet page of NHibernate at https://www.nuget.org/packages/NHibernate/. The latest version is an important milestone since this a major release that has come about 3 years after the previous major release 3.0. There were several minor releases in these 3 years. All these releases have collectively added a lot of great features to version 4.0 of NHibernate. Let's take a brief tour of these features.
What is new in NHibernate 4.0?

The General Availability release of NHibernate 4.0.3.400 was released on August 17 2014. This is the latest release of NHibernate as of the time of writing of this book. This release is quite important from a lot of perspectives. This is a major release of NHibernate that has come almost after three and a half years, so it contains a lot of interesting features and bug fixes.

If you are new to the ORM land or have never used NHibernate before, then you may not find this section very interesting. But hey, why not keep reading and you might find something that strikes a chord with you:

- NH 4.0 is built for .NET 4.0. This may not mean much for the users of NHibernate 4.0 as you can still use older versions of NHibernate on a project that is built on .NET 4.0. However, this is a major move for NHibernate since it now gives NHibernate developers the ability to utilize new features of .NET 4.0. An immediate consequence of this was that NHibernate 4.0 now does not depend on the Iesi.Collection library to support sets (Iesi.Collections.Generic.ISet<T>) because .NET 4.0 has native support for sets, in the form of ISet<T>, and implementations such as HashSet<T>. Back in the days when the old versions of .NET did not have support for sets, the Iesi.Collections library offered a mature set functionality. NHibernate had a dependency on this external library as NHibernate uses sets to represent collections.

- Support for SQL Server 2012 has improved with the implementation of the native IIF function, query paging, and SQL Server 2012 sequences.

- "Mapping by code" rids you of having to write and maintain cumbersome XML mapping files. However, the general audience was never happy with this feature as it was buggy till recent times. NHibernate 4.0 has several defects in this area fixed, thus giving Mapping by code a facelift.

- We all love LINQ and we all hated the fact that NH's LINQ support never matched that of Criteria or QueryOver. There has been great community contribution in this area with over 140 issues fixed. LINQ support in NHibernate 4.0 has matched that of other query providers in a lot of respects. We will be using this feature a lot in the coming chapters of this book.
NHibernate for users of Entity Framework

If you are using or have used Entity Framework (EF), then chances are you know about NHibernate. While EF has been in business for a few years now, it is only in the last couple of years that it has got some serious attention, thanks to the performance improvements brought about in EF 6.x and the implementation of some features that are at the core of any ORM. In spite of the recent popularity of EF, a natural question to ask is why you should be taking a look at NHibernate? Well, for starters, NHibernate is probably the oldest ORM in existence. The NHibernate code base has been used in thousands of production applications over the years and is quite sturdy. However, this sounds very subjective to you, so let me give you a list of features that NHibernate has but EF does not.

- NHibernate has a stable and sophisticated support for second-level cache. The caching implementation of NHibernate is supported by more than one caching provider ranging from in-memory hash tables to distributed caching solutions such as NCafe.
- NHibernate has native support for mapping dictionaries. Dictionary mapping comes in handy in various different situations but one interesting situation that comes to my mind is building a multi-tenant system. In a multi-tenant system, multiple tenants are served from the same instance of the application and their data is stored in a single database instance. If a particular tenant has a requirement that needs to have additional properties on some entity to handle additional data about that entity, then this additional data can be stored in a dictionary of key-value pairs. This dictionary is declared as a property on the entity. NHibernate can then map these dictionaries directly to a suitable table schema.
- In situations where you need to update a large number of database records as part of batch jobs, the usual session object of NHibernate may be too slow as it tracks every change made to the entities and enforces all the database relations/constraints before the changes are saved to the database. However, the stateless sessions of NHibernate speed things up manifold at the cost of losing features such as change tracking and in-memory enforcement of database constraints. EF has some support for such a behavior through an extension method called AsNoTracking or through specific options in the DbContextConfiguration class. However, in my opinion, both of these options are not as intuitive as the stateless sessions of NHibernate.
- NH, in general, supports more number of databases than EF does. EF is built, tested, and most commonly used for MS SQL Server. A lot of people also use it for Oracle, but the level of support for databases such as MySQL, Sybase ASE, PostgreSQL, and DB2 is not at par with that for the other databases. If you use one of these databases, then chances are that you will face fewer problems if you are using NHibernate.
Identity generation is one of the important features offered by an ORM. Every database table must have an identity column that uniquely identifies a particular record from the other records. We skipped this part in the introduction section for the sake of simplicity. Corresponding to the identity column in the database table, you will have a class property that holds the identity value. You can then generate a unique identity value for every new instance of this class or you can ask your ORM to generate one for you. If used wisely, identity generation can save a few database trips and improve performance. A particular identity generation strategy may not work well for all situations, so having more than one strategy always helps. NHibernate comes bundled with several identity generation strategies, while EF is still catching up.

NHibernate’s support for managing concurrency is far better. Concurrency is when two sessions (usually representing two different end users connecting to the database) try to update the same database record. The last session to update the record is usually working on a stale copy of data. Data is called stale when there is a latest updated version of the data present in the database than the one that you have loaded in the memory. In such a situation, in the absence of proper checks, it is possible that the session working with stale data ends up overwriting a legitimate record with some old value. NHibernate has at least three different models of concurrency management wherein NHibernate detects updates on stale data, aborts the update, and lets the application know of the same. We are going to look at these concurrency models in one of the upcoming chapters in detail, but it is worth mentioning that EF has a limited support for concurrency and it works most effectively only when used with the MS SQL Server database.

NHibernate supports global filters. Global filters let you define dynamic SQL filters once and apply them to every database query or selectively apply them to some of the database queries. One situation where this feature comes in very handy is when you are working with a multi-tenant database. Another situation that I can think of is when you are storing the localized versions of strings in the database and you filter out the correct version by using a culture filter based on the current thread culture.

NHibernate supports four different mechanisms for querying data. While LINQ is the easiest and is supported by both NHibernate and EF, the other three, namely, HQL, Criteria, and QueryOver, are only available in NHibernate. One of the useful things that you can do with these is the tuning of the final SQL generated for your code. With these different query mechanisms, you get a better control over the final SQL generated. Most DBAs would want this ability.
Is using ORM a bad idea?

ORMs offer a frictionless development experience when working with relational databases. Still, some minds in the industry feel that using an ORM is not a good idea. Of all the reasons presented against using ORMs, the two most commonly given are as follows:

- ORMs abstract away databases, and hence, the important database concepts that are essential for any developer to master usually get ignored
- The use of ORMs results in a performance degradation of the application

It will be interesting to drill deeper into these reasons. It is true that ORMs abstract away databases and handle almost all database interactions for you in a transparent way. You do not need to bother yourself with a deeper understanding of how an ORM is fetching data from the database. If you master APIs offered by the ORM, you will do just good to interact with a moderately complex database through the most commonly used queries. Further, since this almost always works, you never need to figure out what the ORM is doing under the hood for you. So, you indeed miss out on learning how to work with databases. If you are using your ORM to generate a database schema, then chances are that you are missing out on important database building blocks such as keys, associations, relations, and indexes. So, there is an element of truth in the first reasoning. If you do fall into this category of developers, then this lack of database knowledge will hit you the hardest when least expected.

Talking about the performance degradation caused by ORMs, I can only say that this has an element of truth but it is very subjective. I say subjective because the word "performance" cannot stand on its own. When we talk about performance, we usually refer to a benchmark. This benchmark can be the expected performance of a system or the performance of a similar system used for the purpose of comparison. The performance argument against ORMs is usually in the context of a comparison with other ways of interacting with databases, which include, but are not limited to, the use of plain SQL with ADO.NET or the use of micro-ORMs. While micro-ORMs do offer the best performance, the situation with ADO.NET is slightly different. ADO.NET offers barebones, and you need to write most of the code around reading data from datasets/data readers and hydrating objects. Depending on your experience with building performant systems, you may be able to squeeze more performance out of ADO.NET than is offered by ORMs. However, the key phrase is "experience with building a performant system." If you do not have such experience, then it may be better to stick to an ORM as you will tend to not make the common mistakes made while working with databases.
Why ORM is a better bet?

So, why should we use an ORM then? Let's take a step back and think about our options when we do not use any ORM. In the absence of an ORM, you can either use a micro-ORM or put together your own data access layer implemented using something such as ADO.NET. Micro-ORMs take a minimalist approach towards object relational mapping. Some of them use advanced language features and conventions to automatically map classes to a database schema, while others explicitly make use of custom SQL.

Micro-ORM can be called the little brother of ORMs and is an entirely different topic, so I am not going to consider this option for the sake of this discussion. So, we are left with building our own data access layer. Now what does this mean? First and foremost, this means that you are going to spend a significant amount of time in building something that lets you interact with relational data stores. Further, you are going to do this along with numerous other fellow developers from the "No-ORM" camp. Can you imagine how many times this wheel is going to be reinvented? If you are working in an environment where everyone values "code reuse," then chances are that you would try to reuse this data access layer in more than one project, thus limiting the duplication. However, in reality, this rarely happens for two reasons. One, most of the time, we are so busy delivering our own stuff that we tend to ignore work by other teams, work that you can potentially use as is in your project. The second reason is the realization of the fact that the first team that built the original data access layer did it to tailor to their needs and it does not quite fit your own needs. So, you decide to roll your own. So, all of this leads to countless developers building the similar thing all over again for every new project that they start working on.

Next is the challenge of maintaining the quality of such a data access layer. For simple applications, you can write a complete data access layer in a single class, but such an approach usually does not work for complex applications. When requirements from a data store become complex, it becomes challenging to implement a data access layer that can cope with all the requirements. Without proper experience and knowledge of different experience patterns, you are bound to build a data access layer whose quality cannot be guaranteed.
Every data access layer follows a particular pattern to interact with the database. While some data access patterns have stood the test of time, there are others that should be avoided at any cost. Rolling out a home-grown data access layer means having knowledge of what has worked for others and what has not. It also means having a gut feeling about what will work for your project depending on how the project turns out in the future. These things are not easy to come by and people who are usually good at these things have years of experience in dealing with such problems. It is not possible to have such people onboard every time you are building a data access layer. So, chances are that things may go wrong and you would invest time in fixing things. ORMs, most notably NHibernate, have been around for years. The data access patterns implemented in NHibernate are proven to be the best. At the same time, a care has been taken to avoid designs that are considered harmful for database interactions. Being an open source project, NHibernate welcomes contributions and defect reports from the general public. So, NHibernate has become richer and more stable over time. Contributions from the general public mean that NHibernate is built by learning from the mistakes of hundreds of developers and by listening to the needs of a large community of developers around the world. Not using NHibernate or any mature, open source ORM for that matter would only mean shunting out such a huge source of common knowledge from your system and reinventing the wheel.

Non-functional features required by the data access layer
Besides the above, there are some non-functional requirements that every data access layer would need to offer sooner or later. The following is a representative list of such features:

- **Logging**: Logging probably is one of the most important non-functional requirements. You would need to log not only the errors and exceptions but also important information such as details of the SQL query sent to the database and the data returned from the database.

- **Caching**: Caching may sound a simple thing to implement but as your domain grows complex so do the data querying patterns. Such complex data query patterns have very different caching requirements, which are not trivial to implement.

- **Connection management**: You might think that ADO.NET offers connection management and connection pooling. Indeed, it does. But there are more features around connection management that we need. In real-life applications, you need the ability to reuse connections across multiple queries, be able to make new connections, disconnect the existing connections at the right time, and so on.
• **Transaction management**: As with connection management, ADO.NET does offer basic transaction management but the needs of most of the real-life software around this area can soon grow complex.

• **Concurrency management**: Concurrent updates to the same data from multiple users may result in defects that are difficult to find out. Being able to know whether you are working on stale data or not is critical to building robust applications.

• **Lazy loading**: Lazy loading is when you load only the part of an object that is needed at the moment. We will cover lazy loading in greater detail in the coming chapters. For now, it is worthwhile to note that lazy loading is not a simple feature to implement.

• **Security**: Your data access layer is the primary gateway to the critical data that you store in your database. The security of this data is paramount, and hence, any data access layer needs to be secured and should be hardened against attacks such as SQL injection attacks.

• **Batching of queries**: Some queries can be batched with other queries and executed at a later time. This improves the throughput of the SQL connection and hence improves the performance. Not all data access layers are intelligent to batch and execute queries at a later time.

It is unlikely that your home-grown data access layer will have all of these features. You might manage to build some of these features, but getting all of the features in would practically not be possible. This can mainly be driven by the fact that your main objective on the project is to deliver business functionality and not build a fully featured data access layer. Even if you do manage to put together a decent data access layer that has most of the above features, what are the odds of you having tested every edge case and fixed most defects?

On the other hand, any mature ORM would offer most of these features. Since this book is about NHibernate, I will concentrate on NHibernate here. NHibernate has all of these features built into it from the beginning. These features have not only been timely tested but have also taken shape according to the real needs of the developers. Because it is an open source project, a lot of community feedback and contribution has gone into shaping these features. So, you are guaranteed to see superior implementations of most of these features.

Given all of the preceding advantages, would I consider using an ORM a bad idea? I think not. On the contrary, I would consider using an ORM to be a good idea.
Introduction to NHibernate

Building blocks of NHibernate

Now that we know that using an ORM is not essentially a bad idea but can actually be a very productive way of writing code that interacts with the database, let's turn our attention back to NHibernate.

NHibernate stands on three important building blocks. To understand these building blocks is essential to understand and use NHibernate effectively. We are going to explore these building blocks in detail in the coming chapters, but I would like to give you a brief introduction to these now.

Mappings

NHibernate uses mappings to find out which class maps to which database table and which property on the class maps to which column on the table. Every time your application tries to interact with the database, NHibernate refers to the mappings in order to generate the appropriate SQL to be sent to the database for execution.

There are three different ways that you can tell NHibernate about the mappings. One of the ways is based on traditional XML, whereas the other two are based on the code and provide a fluent API to express the mappings. In Chapter 3, Let's Tell NHibernate About Our Database, I will talk about these ways in enough detail. However, mappings that we will use in the other chapters will only be explained using a code-based API, also called mapping by code.

I have simplified mappings for the sake of introduction, but mappings offer a lot of advanced features that go beyond the basics such as mapping tables and columns. These advanced features impact the generation of SQL, performance of the application, and resilience of the data access layer. Hence, it is important to understand and master the mappings in order to use NHibernate most effectively.

Configuration

NHibernate lets you specify via configuration a lot of details that it uses at runtime. These details affect what NHibernate does and how NHibernate does a particular thing. Although configuration is a small thing, it is important to master it. On most projects, you will deal with configuration in the initial phase, and once everything is settled, you will rarely come back to it. The only times that you will come back to it are when you need to turn on or off something and knowing how to do this will save you some time. The NHibernate configuration lets you control the following:

• which database you want to connect to (MS SQL Server, Oracle, and so on)
• where is that database (read connection strings)
Chapter 1

- which SQL driver to use (or which SQL client library should NHibernate use)
- where are the class mappings
- should NHibernate log important information
- should NHibernate cache entities by default
- how should NHibernate manage sessions
- should NHibernate use second-level cache
- where is the second-level cache

As you can see, there are some useful configuration options there. We will see the details of these options and how to configure them in Chapter 4, NHibernate Warmup.

Configuration is not just about providing the defaults for a few NHibernate settings. Configuration can be loaded in the memory, and you can do some amazing things with it. One of such commonly done things is the use of the configuration to generate database scripts. Once you define your mappings and build your configuration, you can generate a SQL script to build all database schemas, tables, relations, constraints, indexes and so on. We are going to cover this in detail in Chapter 4, NHibernate Warmup. For now, it is worth noting how important and powerful the NHibernate configuration is.

**Session**

Session is the most commonly used object from NHibernate. Any database interaction that your application needs to do using NHibernate, it must do so by using a Session object. Think of a Session object as a wrapper around the pipe to the database, which is an SQL connection. The Session object not only manages the connection to the database but also exposes methods (some directly and the others indirectly through extension methods) that let you write code to interact with the database in the form of queries and create, update, and delete operations. The Session object also exposes the API for transaction management. Further, the Session object implements a form of caching that boosts the performance of queries. This list can go on as there are so many features that the Session object offers. However, let's leave those features to later chapters where we will look at them in detail.
Introduction to NHibernate

Because of the rich set of features that the Session object offers, the Session object is the most commonly used NHibernate type. Since Session is so commonly used, NHibernate has implemented it in such a way that creating a new Session object is a very cheap operation. However, this does not mean that you should freely create new instances of the Session object. There is some penalty that you will have to pay directly or indirectly when you create a large number of session objects randomly. There are some rules that if you play with, you can gain all the benefits of cheap Session objects. We will cover all of these rules at appropriate times in different chapters of this book.

Summary

Most people choose to skip the introduction chapter of any book. If you are not one of them and have actually gone through my attempt at introducing you to the world of ORMs and NHibernate in particular, then congratulations, you have had a very good start. You now know briefly what NHibernate is and what kind of problems it solves for you. You may also have an idea of some of the advanced features of NHibernate. Last, but not least, I hope that now, you have an answer to the question "Whether using an ORM is a bad idea?," and I hope that the answer is a resounding "No."

In next chapter, I am going to present a business problem. The aim is to use NHibernate to build the data access layer as one of the parts of the solution to this problem. We will build the data access layer incrementally as we go over each chapter. In the next chapter, I will also talk about how to set up your development environment and how to get ready for some NHibernate action. Just flip over when you are ready.
Where to buy this book

You can buy Learning NHibernate 4 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.