Chapter 1: What is an Object-Relational Mapper?

This chapter describes what an Object-Relational Mapper (ORM in short) is, and its differences to its younger sibling, the Micro ORM, the technologies that we will be applying throughout the book – NHibernate, Entity Framework Core and Dapper - and some of the core concepts involved in Domain-Driven Design and Object-Relational Mapping that will be covered in depth throughout the book. They are:

* What is an ORM
* What is NHibernate
* What is Entity Framework Core
* What is Dapper
* Understanding ORM concepts
* Understanding ORM features
* Comparing ORMs to Micro ORMs

The information presented here is of the utmost importance for the understanding of the rest of the book.

What is an ORM

An ORM is a framework or library that lets you query and manipulate data coming from a relational database using an object-oriented (OO) paradigm. Or, put in a different way, an ORM helps us overcome the object-relational impedance mismatch, which happens because, of course, a relational database and an object-oriented language are totally different beasts, allowing us to work with our object-oriented language of choice and having data persisted in the database behind the scenes.

When we think about relational databases, we normally think of:

* Databases
* Schemas
* Tables
* Views
* Columns
* Primary keys
* References to other tables (and foreign keys)
* Constraints and checks
* Records
* Queries

Whereas when we think about object-oriented programming (OOP) we have instead:

* Classes (and the namespaces they live in)
* Inheritance or polymorphism (base classes, inherited classes)
* Fields (and / or properties)
* Instances of classes
* Instance validation
* Class methods

As you can see, there is some similarity between the two concepts:

* A class can be mapped to a table, although most relational databases don’t support inheritance
* A table or view record can be mapped to a class instance
* Table or view columns can be mapped to class fields or properties
* A class can have a field (or fields) that provides its *identity*, meaning, its value is unique for all instances of that class, in pretty much the same way as a table’s primary key
* A class can have fields of another class’ type, just like foreign key columns that connect two tables
* A class can be validated to check if it is in a consistent state, like a record can, even though a record only exists if its columns met all the constraints and checks
* SQL queries can be mapped to class methods or LINQ queries in .NET

ORMs take care of this mapping, allowing us to only care about classes, their lifetime and properties. Retrieving records from the database and persisting them back is just a detail, something that ORMs do for us automagically. Well, almost, this is not exactly true and forgetting that we are working with a database can lead to problems and unexpected situations.

Also, modern ORMs have functionality that help us dealing with typical development scenarios, such building queries for us, knowing how to transform the data and picking the right tables for its persistence, loading data only when it is required, and others.

This first chapter does not have any development recipes, instead it will introduce the concepts so that you are familiar with them.

What is NHibernate

NHibernate is a port for the .NET (and .NET Core) framework of the venerable Hibernate, an open-source project written in Java that is the *de facto* standard for persistence of objects into relational databases in the Java world. This .NET project was started 15 years ago and is maintained by a small team of highly dedicated enthusiasts who work on their free time. Because of its age, it has many features and supports a vast number of relational databases out of the box, but it also has a steeper learning curve. In recent years, its popularity has somewhat decreased, mostly because of the arrival of Entity Framework Core.

NHibernate source code is available at <https://gihub.com/nhibernate/nhibernate-core> and its NuGet package is [https://www.nuget.org/packages/NHibernate](https://www.nuget.org/packages/NHibernate/).

What is Entity Framework Core

Entity Framework Core is Microsoft’s own ORM. It began as Entity Framework back in 2008 with a very limited set of features, and, with the announcement of the .NET Core, it has since split in two, Entity Framework “classic”, still targeting the Microsoft .NET “full” framework and Entity Framework Core, which is built on .NET Core. Microsoft has announced that it will deprecate Entity Framework “classic” and that no more major versions will be released. Entity Framework Core, as .NET Core itself, is now open-source (managed by Microsoft) and has gained significant usage because of its easiness to use and because of the features that have been introduced. Still behind NHibernate in some ways, it is quickly getting close.

Entity Framework Core is made available in binary format from NuGet (many packages, the root one being <https://www.nuget.org/packages/Microsoft.EntityFrameworkCore>) and in source code from <https://github.com/dotnet/efcore>.

What is Dapper

Dapper is a micro-ORM that is also open-source and maintained by StackOverflow owners. It is a very thin layer around ADO.NET, .NET’s own database access API, and unlike its more robust siblings, it does not attempt to add any significant services or features, but instead it embraces SQL and the classic way of accessing databases in .NET.

Dapper’s NuGet package is <https://www.nuget.org/packages/Dapper> and the source code can be found at <https://github.com/StackExchange/Dapper>.

Understanding ORM concepts

Some of the concepts that ORMs embrace are related to Domain-Driven Design (DDD). This is because working with classes and modelling our domains with them is essentially what is preconized by ORMs. So, in this book, the following concepts will be covered:

Entity

An entity is just a **Plain Old CLR Class** (**POCO**) that has properties – a state - and an *identity*. The class’ property values may change over time, but not its identity. By identity I mean a property or set of properties that makes the class instance unique: if two instances of the same class have the same values for their identity properties, then they refer to the same entity. In relational terms, you could think about a table record with its columns and primary key. An entity can also have associations to other entities. Some examples include an **Animal**, a **Product**, a **User**, or a **Post**. A root entity is the one that is loaded from the ORM, and it may contain property references and collections to other entities, which are loaded together. An entity and its associated entities represents a graph.

Value Object

A value object is also a POCO, like an entity, but does not have an identity, meaning, it is just a collection of properties’ values. It is a class that doesn’t have any id property and can be aggregated inside other entities. Value Object’s equality is based on its properties’ values, not some identity property. Some examples include **Address** (street, postcode, city, etc) or **Money** (value, currency).

Properties

Entities and value objects have properties, these are normally of primitive types (strings, characters, integers, floating-point, decimals, bytes, or Boolean values), enumerations or standard value types (**DateTime**, **TimeSpan**, **Guid**).

Identity

The identity property is a (or a collection of) property whose value is unique among all entities of the same type.

References

A reference in an entity or value object is a property that is of another entity type. It represents a one-to-one or many-to-one relation, in the relational database world.

Collections

A collection property of an entity type represents a one-to-many or many-to-many relation in the database.

References

For additional information, please consult Domain-driven design, by Eric Evans, possibly one of the best references on Domain-Driven Design, with examples in .NET.

Understanding ORM features

Modern ORMs feature several services that greatly help our lives as developers: they are the reason why we use an ORM after all. Not all ORMs offer the same features, but some of them are quite common. I will briefly mention them here.

Database Abstraction

By using a class to represent some structured data we hide the database that is underneath it. For example, using the same class and changing the mapping, we can target different table names, database vendors or even totally different concepts, such as NoSQL or SQL database engines. This is a very powerful concept. The ORM can depend on ADO.NET abstraction for database connectivity independence, but it must know something about the database to generate the right SQL for its more complex operations.

Note

ADO.NET is the core database abstraction that all .NET APIs build upon. All database vendors that aim to support .NET must distribute an ADO.NET provider for their product.

Mapping classes to database objects

A mapping is what binds a class to a database, in the case of a relational database, to a specific table. Its properties are mapped to database columns and its foreign keys map to entity reference properties. It can also enable certain features, such as lazy loading or caching, for a specific entity.

A mapping can be code or text-based, meaning, it can make use of external files (XML, for example) or it can be configured in code. Having the mapping configuration in code means it can be more dynamic.

Understanding database relations

A database relation is a foreign key from one table to another:

* A **one-to-many relationship** (or a **many-to-one**, if you look from the other side): in this example, a **Blog** has many **Posts**, and a **Post** has exactly one **Blog**. In the class model, it is represented by a reference property on the **Post** and a collection property on the **Blog** class:

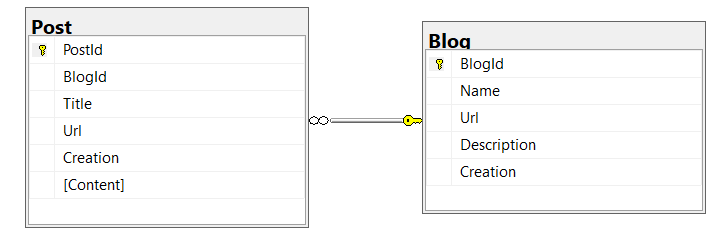


Figure 1 - A one-to-many or many-to-one relationship

* A **one-to-one relationship**: a **Post** can have one **PostImage**. In a class, it is also a reference property on each class, **Post** and **PostImage**:

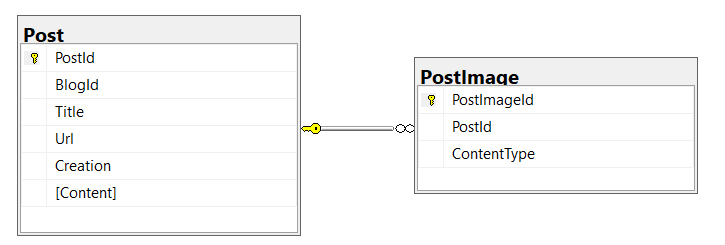


Figure 2 - A one-to-one relationship

* A **many-to-many** relationship: a **Post** can have multiple **Tags** and a **Tag** can be associated with many **Posts**. Is represented by a collection on each class (**Post** and **Tag**):

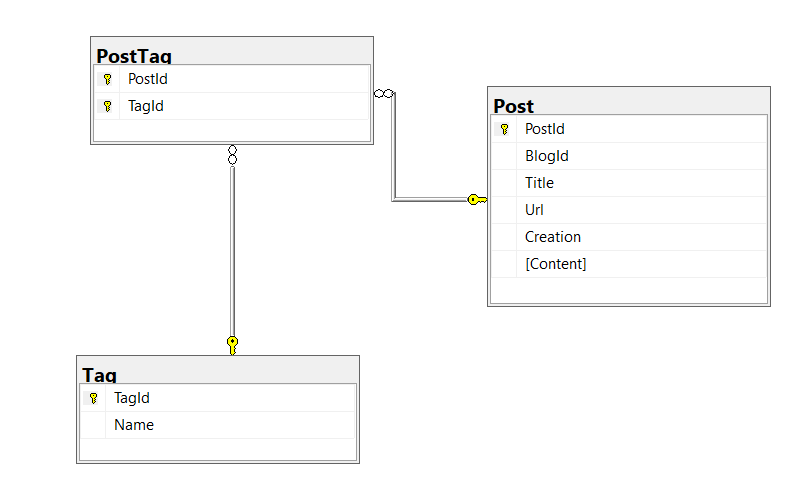


Figure 3 - A many-to-many relationship

In OOP, it can be one either a property that targets another entity (for one-to-one or many-to-one relations) or a property that is an entity collection (for many-to-one and many-to-many). When we map these tables and relations to classes, we may end up with this:

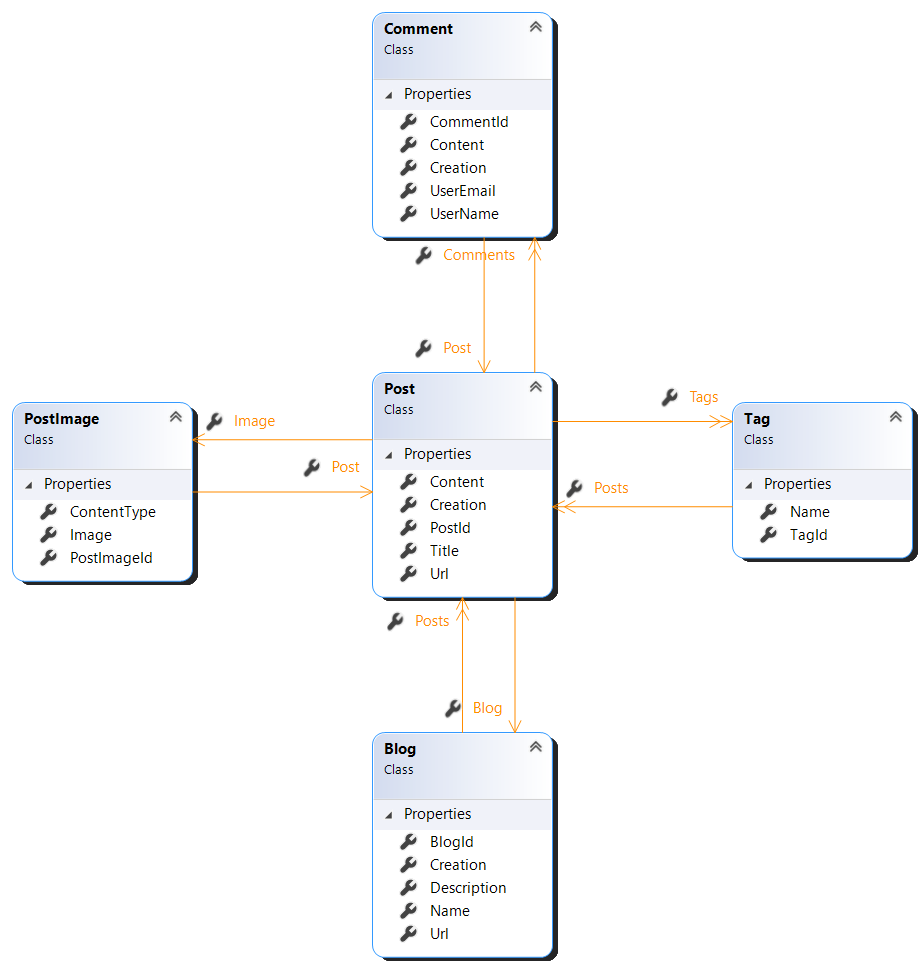


Figure 4 - A domain model

Mind you, of course, that the mapping means that there is no direct connection between a table and the class’ name, or a column and a property. Also notice that for the many-to-many relation between Post and Tag, there is no need for a class for the mapping table, **PostTag**, in this example. The mapping tells what this class is and the ORM takes care of it for us.

Implementing table inheritance

Not all relational databases support table inheritance, actually, most of them don’t. But OOP is based on classes and inheritance, so ORMs had to find ways to represent inheritance. There are three ways to represent an inheritance:

* **Single Table inheritance** (also known as **Table Per Class Hierarchy**) is when a single table is used for all columns of all the derived classes. There must be a discriminator column that says, for each record, to which class it should be mapped. It has both advantages and disadvantages: for once, it is easier to find all records, as only one table needs to be queried, but it means that the number of columns can possibly explode, as one adds more inherited classes to the model, and therefore more properties need to be mapped. Also, columns for all properties of the derived classes need to be nullable, because they are not part of all of them:

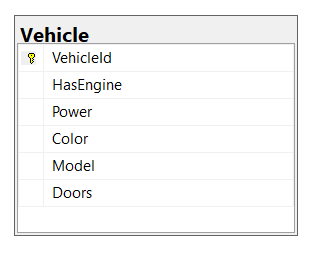


Figure 5 - Single table inheritance

* **Class Table Inheritance** (also **Table Per Subclass**): there is a table for the base class, where all of its properties are stored, and one for each subclass, the primary key is shared by them. It means that there is a need to query more than one table, and its records joined, and, for inserts and updates, there is also the need to update many tables:

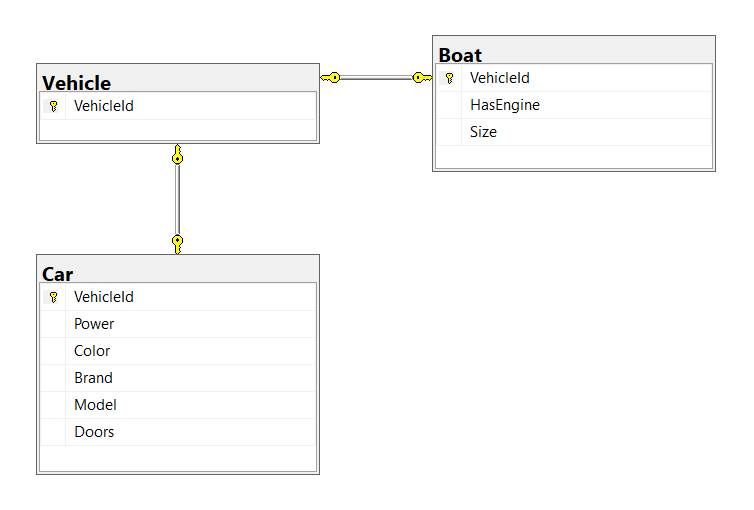


Figure 6 - Class table inheritance

* **Concrete Table Inheritance** (**Table Per Concrete Class**): each subclass maps to its own unique table. For querying from a base class, there is the need to query all of them, to find the right one. Care needs to be taken to prevent duplicate primary keys on each of these tables:

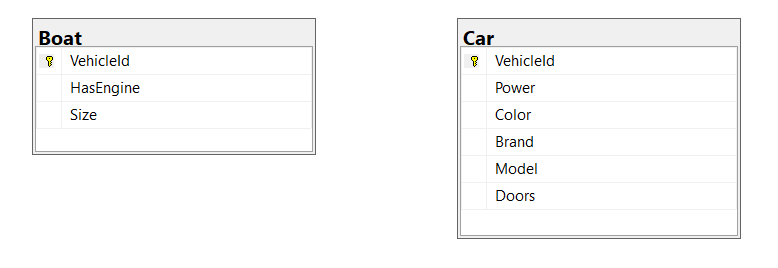


Figure 7 - Concrete table inheritance

Depending on what we want, we may choose one or the other. Right now, EF Core only supports the first two, Concrete Table Inheritance is not supported.

Hydrating and dehydrating objects

By hydration we mean the process by which a record’s values is assigned to a class instance’s properties, and dehydration is exactly the opposite, e.g., taking an instance’s properties and turning them into records to insert or update. The hydration and dehydration process, of course, makes use of the mapping.

Generating table keys

Key generation is the process by which a primary key for a table’s record is generated. There are pre-insert and post-insert generators, for example, SQL Server and MySQL have identity or auto-increment columns, whose value is only known after a record is inserted. Other databases, like Oracle, SQL Server and PostgreSQL, have sequences, which are also only known after an insert. But there are other options for generating primary key values, such as:

* GUIDs
* Timestamps
* The Hi/Lo algorithm
* Manually assigned keys (normally should be avoided)

Not all ORMs support all of these generators, but usually all of them support the Hi/Lo algorithm. This is a very efficient algorithm for generating primary keys which are known in advance and thus allow scenarios such as batching.

Change tracking

When a record is loaded from a database and a POCO class is instantiated and hydrated. The ORM then takes a snapshot of the initial values and keeps them in its first level cache. When we ask the ORM to persist the data, it will compare, for each entity that it is tracking, all its properties against the original value, and will persist only those that have changed.

Query options

.NET ORMs have generally different query options. LINQ is, of course, the common one and the one that is normally used, because it is strongly typed. ORMs usually also support plain SQL, which is useful for scenarios that are not supported by LINQ. Some ORMs also support other proprietary query options: for example, NHibernate has Hibernate Query Language (HQL), similar to an object-oriented SQL, Criteria API (a **Query Object** implementation) and QueryOver (Criteria API with LINQ expressions).

It is always possible to query by an identity value (a primary key in a relational database) or by some criteria. Some query options allow creating queries dynamically.

Batching multiple queries

Batching is the process by which multiple queries or modifications are combined together and issued at once. This greatly improves the performance, but is not supported by all database engines and ORMs.

Lazy loading queries

Reference and collection properties in an entity represent foreign keys to other tables. When loading a root entity, we may, or may not, want all of its related entities to be loaded altogether. Of course, this could mean loading the all database at once, which is seldom what we want. For this ORMs introduced lazy loading: the records associated with a reference or collection property are only loaded if and when they are actually accessed in the code. This is usually useful, but can result in some problems, like, for example, when the connection to the database is closed, or when traversing a collection of entities and having an additional query for each one of them (the infamous SELECT N + 1 problem).

Note

The SELECT N + 1 problem is a common problem when using an ORM and it happens when you use lazy loading to load a collection of entities (1) and then you need to access each of them (N).

Managing transactions

ORMs have the concept of Unit of Work. What it means is, when the ORM is about to persist a graph of entities, it must either succeed or fail all of the operations – a transaction.

ORMs may create a transaction by themselves when there are many entities to update, or there may be a need to create one ourselves; in this case, the ORM automatically makes use of it.

First and Second Level Caches

When an object is loaded from a database and a class is instantiated and hydrated, it is kept in local storage: the Identity Map, or First Level Cache. This is created per type, and it associates a specific key (an identity, or primary key, in database terms) with a class instance. Instances kept in the first level cache are subject to change tracking.

First Level Cache is stored in the process’ memory, and is scoped to a session, which means that it is not useful outside the originating session. While this is usually what we want, we may also want to share this cache between sessions, including potentially across machines. This is where the Second Level Cache makes its entry: usually a place to store less-changing entities, like reference data, so that it does not need to be constantly retrieved from the database. ORMs normally support Second Level Cache through extensions and external services, such as Memcached or Redis.

Note

Redis and Memcached are open-source distributed object memory caches, which you can obtain from <https://redis.io> and https://memcached.org.

Session

A session encapsulates a database connection, but this doesn’t mean that a session is permanently connected to the database: the session manages the connection, opening and closing it as needed. A session needs to be properly disposed, which means all resources used by it are freed when no longer needed. Entities loaded from a session are considered disconnected when the session is disposed of.

Generating and updating the database

From the mappings, the ORM knows very well what the database looks like, which means that it can generate it from scratch or make changes to it as the OOP model changes (new classes or new properties added, for example). Some ORMs, like Entity Framework Core, have migrations, which means that each individual change to the model can be scripted and applied at any given time.

References

For additional information, please refer to Martin Fowler’s Patterns of Enterprise Application Architecture.

Comparing ORMs to Micro ORMs

We talked about the many typical services offered by an ORM; usually, Micro ORMs don’t offer any of these, as they are just small layers on top of a connection (ADO.NET). This just means that Micro ORMs are considerably simpler, and possibly faster, than their siblings, but demand more work from us. Don’t expect them to offer lazy loading, detect changes in your loaded entities or generate or update the database for you, or, understand what table inheritance is: that’s just not what they are meant for.

Usually Micro ORMs support SQL and perhaps LINQ for querying and can instantiate and hydrate/dehydrate classes, but they provide no mapping capabilities, usually they rely on conventions, for example, to infer the naming of columns from the property.

What’s Next

In this chapter we introduced the core concepts around ORMs and Micro ORMs. This was in no means a thorough explanation, but during the next chapters we will look at each of them in detail and explain how they are used in both EF Core and NHibernate. We shall start our journey by looking at Entity Framework Core configuration.