UNIVERSITÉ
LIBRE
DE BRUXELLES

Object Relational Mapping
and
Entity Framework

Advanced Databases Project

Professor: Esteban Zimányi

Students: Bubacarr Jallow Shafagh Kashef

12/18/2018

Contents

1.Abstract 3
2.Introduction 3
2.1.Object-Relational Mapping 3
2.2.Comparison ORM with traditional data access techniques 4
2.3.Pros and Cons 4
2.4.But ORM can be a pain: 5
2.5.Hibernate ORM 5
2.5.1.Persistence 5
2.5.2.Relational Databases 5
2.5.3.The Object-Relational Impedance Mismatch 5
2.5.4.Granularity 5
2.5.5.Subtypes (inheritance) 6
2.5.6.Identity 6
2.5.7.Associations 6
2.5.8.Data navigation 6
2.6.Why are ORMs useful? 6
3.Entity Framework 7
3.1.ENTITY DATA MODEL 7
3.2.The Conceptual Model 7
3.3.Mapping 8
3.4.The Storage Model 8
3.5.ENTITY FRAMEWORK CORE 8
3.6.DbContext 9
3.7.DbSet 9
3.8.DbContextOptionsBuilder 9
3.9.ModelBuilder 9
3.10.Migrations 10
3.11.Data Loading 10
3.12.NuGet Package Manager Console 10
3.13.EFCore: Code First 10
3.14.EFCore: Database First 10
4.IMPLEMENTATION 11
4.1.Create Entity Classes and Underlying Data Sources 11
4.2.Perform CRUD operation 12
4.3.Making changes to the Entity Model 14
5.Conclusion 15
6.References: 16

1.Abstract

Object relational mapping is a framework that abstracts the communication between two different programming paradigms, viz, object oriented and relational databases, which has been an issue for developers since it makes their work more tedious. To tackle this problem, different technologies have been offered and the Entity Framework is Microsoft's implementation of an ORM. Recently Microsoft is migrating towards an open source, cross-platform technology and the latest versions of the Entity Framework called the Entity Framework Core (EF Core) have been completely rewritten to render them cross-platform. Development in the EF Core follows basically two approaches, viz, the database first approach and the code first approach. The EF Core has functionalities that includes APIs, execution engines, migration toolings, etc., in order to effectively implement and ORM. In the implementation of the code first, the application is written from scratch. Since the EF Core is now open source, it is project to gain more popularity in the future.

2.Introduction

2.1.Object-Relational Mapping

Object-relational mapping (ORM) is a mechanism that makes it possible to address, access and manipulate objects without having to consider how those objects relate to their data sources. ORM lets programmers maintain a consistent view of objects over time, even as the sources that deliver them, the sinks that receive them and the applications that access them change.

ORM manages the mapping details between a set of objects and underlying relational databases, XML repositories or other data sources and sinks, while simultaneously hiding the often changing details of related interfaces from developers and the code they create.

ORM hides and encapsulates change in the data source itself, so that when data sources or their APIs change, only ORM needs to change to keep up—not the applications that use ORM to insulate themselves from this kind of effort. This capacity lets developers take advantage of new classes as they become available and also makes it easy to extend ORM-based applications. In many cases, ORM changes can incorporate new technology and capability without requiring changes to the code for related applications.

An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code.

Object-relational mapping (ORM, O/RM, and O/R mapping tool) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to construct their own ORM tools.

In object-oriented programming, data-management tasks act on object-oriented (OO) objects that are almost always non-scalar values.
Many popular database products such as SQL database management systems (DBMS) can only store and manipulate scalar values such as integers and strings organized within tables. The programmer must either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping implements the first approach.

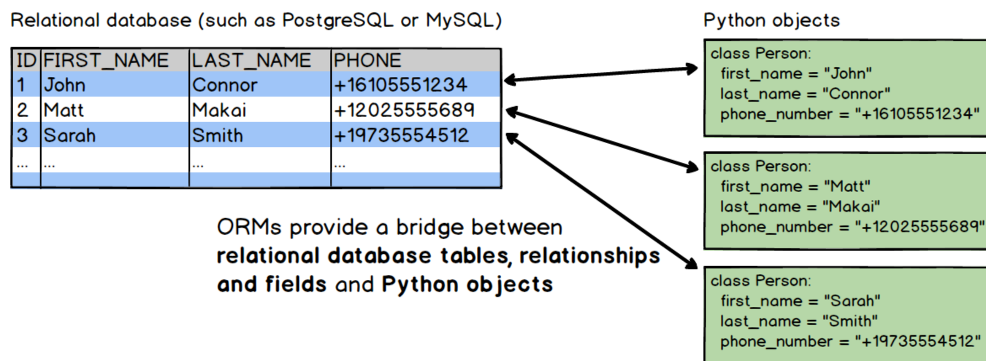

Figure 0: Object Relational Mapping

2.2.Comparison ORM with traditional data access techniques

Compared to traditional techniques of exchange between an object-oriented language and a relational database, ORM often reduces the amount of code that needs to be written.
Disadvantages of ORM tools generally stem from the high level of abstraction obscuring what is actually happening in the implementation code. Also, heavy reliance on ORM software has been cited as a major factor in producing poorly designed databases.
Object-Relational Mapping (ORM) is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. When talking about ORM, most people are referring to a library that implements the Object-Relational Mapping technique, hence the phrase "an ORM".
An ORM library is a completely ordinary library written in your language of choice that encapsulates the code needed to manipulate the data, so you don't use SQL anymore; you interact directly with an object in the same language you're using.

2.3.Pros and Cons

Using ORM saves a lot of time because:
DRY: You write your data model in only one place, and it's easier to update, maintain, and reuse the code.
It forces you to write MVC code, which, in the end, makes your code a little cleaner.
You don't have to write poorly-formed SQL (most Web programmers really use it, because SQL is treated like a "sub" language, when in reality it's a very powerful and complex one).
Sanitizing; using prepared statements or transactions are as easy as calling a method.

Using an ORM library is more flexible because:
It fits in your natural way of coding (it's your language!).
It abstracts the DB system, so you can change it whenever you want.
The model is weakly bound to the rest of the application, so you can change it or use it anywhere else.

2.4.But ORM can be a pain:

You have to learn it, and ORM libraries are not lightweight tools;
You have to set it up.
Performance is OK for usual queries, but a SQL master will always do better with his own SQL for big projects.
It abstracts the DB. While it's OK if you know what's happening behind the scene, it's a trap for new programmers that can write very greedy statements, like a heavy hit in a for loop.
ORM libraries:Java: Hibernate. PHP: Propel or Doctrine Python: the Django ORM or SQLAlchemyORM library in Web programming using an entire framework stack like: Symfony (PHP, using Propel or Doctrine). Django (Python, using an internal ORM).

2.5.Hibernate ORM

2.5.1.Persistence

Hibernate ORM is concerned with helping your application to achieve persistence. Persistence means that we would like our application’s data to outlive the applications process. In Java terms, we would like the state of (some of) our objects to live beyond the scope of the JVM so that the same state is available later.

2.5.2.Relational Databases

Specifically, Hibernate ORM is concerned with data persistence as it applies to relational databases (RDBMS). In the world of Object-Oriented applications, there is often a discussion about using an object database (ODBMS) as opposed to a RDBMS.

2.5.3.The Object-Relational Impedance Mismatch

Object-Relational Impedance Mismatch' (sometimes called the 'paradigm mismatch') is just a fancy way of saying that object models and relational models do not work very well together. RDBMSs represent data in a tabular format (a spreadsheet is a good visualization for those not familiar with RDBMSs), whereas object-oriented languages, such as Java, represent it as an interconnected graph of objects. Loading and storing graphs of objects using a tabular relational database exposes us to 5 mismatch problems…

2.5.4.Granularity

Sometimes you will have an object model which has more classes than the number of corresponding tables in the database (we says the object model is more granular than the relational model). Take for example the notion of an Address…

2.5.5.Subtypes (inheritance)

Inheritance is a natural paradigm in object-oriented programming languages. However, RDBMSs do not define anything similar on the whole (yes some databases do have subtype support but it is completely non-standardized)…

2.5.6.Identity

A RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity a==b and object equality a.equals(b).

2.5.7.Associations

Associations are represented as unidirectional references in Object Oriented languages whereas RDBMSs use the notion of foreign keys. If you need bidirectional relationships in Java, you must define the association twice.
Likewise, you cannot determine the multiplicity of a relationship by looking at the object domain model.

2.5.8.Data navigation

The way you access data in Java is fundamentally different than the way you do it in a relational database. In Java, you navigate from one association to an other walking the object network.
This is not an efficient way of retrieving data from a relational database. You typically want to minimize the number of SQL queries and thus load several entities via JOINs and select the targeted entities before you start walking the object network.

2.6.Why are ORMs useful?

ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.
The ability to write Python code instead of SQL can speed up web application development, especially at the beginning of a project. The potential development speed boost comes from not having to switch from Python code into writing declarative paradigm SQL statements. While some software developers may not mind switching back and forth between languages, it's typically easier to knock out a prototype or start a web application using a single programming language.
ORMs also make it theoretically possible to switch an application between various relational databases. For example, a developer could use SQLite for local development and MySQL in production. A production application could be switched from MySQL to PostgreSQL with minimal code modifications.
In practice however, it's best to use the same database for local development as is used in production. Otherwise unexpected errors could hit in production that were not seen in a local

development environment. Also, it's rare that a project would switch from one database in production to another one unless there was a pressing reason.

3.Entity Framework

The entity framework is Microsoft's implementation of an ORM. It enables .NET developers to work with relational data using domain-specific objects, i.e., developers are able to with data within a conceptual model. It is aimed at increasing the developer's productivity by reducing the redundant task of persisting the data used in the applications. The figure below shows the overall architecture of the entity frame

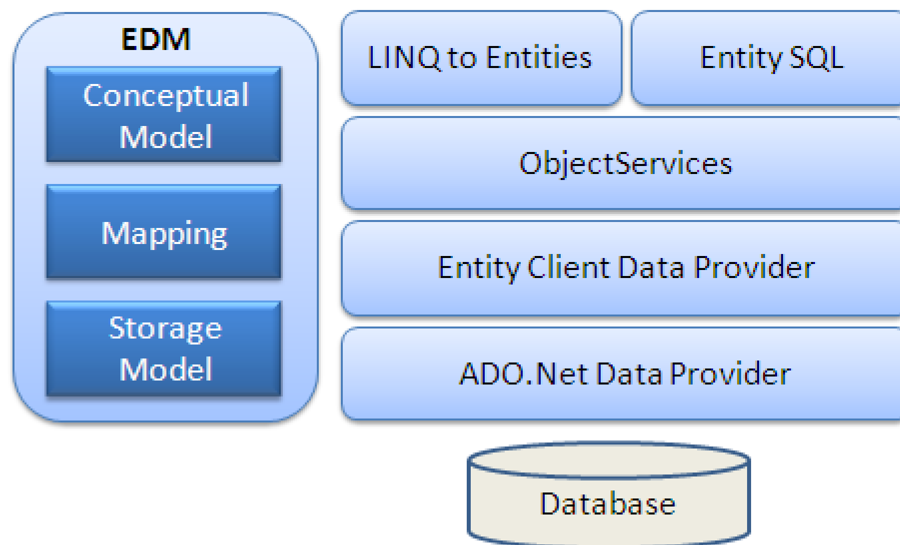

Figure 1: Entity Framework Architecture (EntityFrameworkTutorial.net, 2016)

EDM (Entity Data Model): Model classes, relationships, mappings, database models etc.
LINQ to Entities: Query language for writing queries against the object model.
Entity SQL: Also a query language
Object Service: For materialization, i.e., conversion of RDBMS data into object instances.
Entity Client Data Provider: Translating LINQ and Entity SQL to DBMS SQL.
ADO.Net Data Provider: Communication with databases using standard ADO.Net. (EntityFrameworkTutorial.net, 2016)

3.1.ENTITY DATA MODEL

The EDM is one of the main components of the entity framework and . "It is a set of concepts that describe the structure of data, regardless of its stored form." (Microsoft, 2017). The EDM is made up of three major parts:

3.2.The Conceptual Model

This conceptual model constitutes the real entity model against which we write our queries. This is where we define our entity classes including relationships between entities, properties. It gives us a high-level view of the type of database we will generate.

3.3.Mapping

This provides a mapping between the conceptual model and storage model.

3.4.The Storage Model

The Storage Model gives a schematic representation of the backend data store.
The entity model mainly uses three concepts to describe data structure:

- **Entity Type**

Entity types are used to describe the structure of the data in the Entity Data Model.

- **Association Type**

Association types are used for describing relationships such as the referential constraints that will exist in our database.

- **Property Type**
  The property type gives meaning and structure to the entities by defining the characteristics or composition of that entity, e.g., a Customer entity, has CustomerId, CustomerName, etc as properties. These property type can have both primitive types and complex types.

3.5.ENTITY FRAMEWORK CORE

In this project, we consider a version of the entity framework called the Entity Framework (Core EF Core). EF Core is lightweight, extensible and cross-platform and a complete re-write of the original entity framework.
The following are basically, the core functionalities of the entity framework core:

- An API for defining the Entity Model
- An API for querying the Entity Model
- An Execution Engine that figures out how to convert a LINQ query into SQL command and executing it.
- Reverse Engineering a database in order to create a baseline entity model based on an existing database schema using.
- Migrations tooling that takes a snapshot of the existing model state and uses it to generate the code that needs to be applied to the database in order to update it, bringing it into sync with the current state of the entity model.
- Database Generation Tooling that takes the model and Migration classes and generates or updates the database schema.

3.6.DbContext

The DbContext is the most fundamental core of the EFCore. It is the class that “knows” how to execute a query on a data source. Each DbContext is associated with a data source. The following are some of the main functions of the DbContext:

- Provide connection to the database and keeping that data
- Querying the model and converting the queries into T-Sql. Queries run against the database directly and do not contain any pending changes.
- Help in the process of retrieving the results and putting them into instances of the model and holding on to them.
- Keep track of changes made in the object instances so as to enable it to know what to update when the method SaveChanges() is invoked.

The afore mentioned are accomplished through the use of special classes and methods within the DbContext:

3.7.DbSet

This is used to represent a collection for a given entity within the model and it is actually the gateway to database operation against the entity. They are mapped by default to database table. (learnentityframeworkcore.com, 2017)

3.8.DbContextOptionsBuilder

Contains a series of extension methods such UseSqlServer() that helps in the configuration of a DbContextOptions such that we are able to connect to and use SqlServer in this case. Since EFCore is platform independent, you can connect to different databases aside SqlServer. The DbContextOptions contains a host of settings for individual database engines with each engine having its own list of settings.

3.9.ModelBuilder

Based on the entity classes definitions while adhering to conventions, the EFCore is able to construct a model for the database and apply certain configurations. These configurations can however, be overwritten or supplemented through data annotations or Fluent API. The Fluent API is accessed by overriding the OnModelCreating method in the derived context and uses the ModelBuilder API to configure the model. This is the most powerful method for configuring the model and allows the configuration to be specified without modifying the entity classes. (Microsoft, 2017).

3.10.Migrations

Migration allows us to continue to change the entity model throughout the application development lifecycle. It contains two methods, name the Up and Down methods. The Up method updates the database to the latest version of the entity model whilst the Down method rolls back changes from the current migration and restores the database to a previous migration.

3.11.Data Loading

There are three ways data can be loaded into your application from the database. Lazy loading, which is often the default, involves loading the main entity referenced in the query. It does not load the related entities included in your query. Lazy loading can be turned on or off. If the lazy loading is turned off, we can explicitly load the entities we require by using the Explicit loading. We use the Load method on the related entity's entry.
The third type of loading is used to load the related entities that are part of a query after the main entity of the query has been loaded. This is called Eager Loading and involves of the use of Include (and ThenInclude) methods. This fetches a huge chunk of data in one query operation as it has to return all related data.

3.12.NuGet Package Manager Console

This command line interface is extensively used to run commands that help us setup or install our EFCore, add migrations, update databases and so forth.

3.13.EFCore: Code First

In this approach, all the coding is done in the application platform and the entity classes are built from scratch. The entity classes are then used to generate the database. This approach is used in the implementation for our project.

3.14.EFCore: Database First

Reverse engineering techniques are applied in this approach to generate entity classes from already existing databases. The process involves running the following command in the NuGet Package Manager and specifying the database connection string and optionally the folder and context.
Scaffold-DbContext "<connection-string>" Microsoft.EntityFrameworkcore.SqlServer -OutputDir <Folder> -Context <context name>

4.IMPLEMENTATION

We will be using the code first approach to illustrate the basic features and processes involved in creating and adding an EFCore into your project or application. The following workflow is assumed:

1. Install EFCore and Tooling
2. Create Entity Model and underlying data source
3. Perform CRUD operation

Installing EFCore and Tooling

The following three commands are used to add the EF core to our application:

- Install-Package Microsoft.EntityFrameWorkcore.SqlServer
- Install-Package Microsoft.EntityFrameWorkcore.Tools -Pre
- Install-Package Microsoft.EntityFrameWorkcore.SqlServer.Domain

4.1.Create Entity Classes and Underlying Data Sources

In the code first based approach, we create our entity classes the same we way write our normal plain old CLR object (POCO) classes as follows:

```
public class Order
{
    0 references
    public int OrderId { get; set; }
    1 reference
    public DateTime OrderDate { get; set; }
    1 reference
    public Customer Customer { get; set; }

    3 references
    public List<OrderItem> OrderItems { get; set; }
}
```

Figure 2

The class and class properties will represent the table and columns of our database respectively, that will be generated. By following certain conventions, the EF core automatically identifies constraints like primary keys and generate them. Foreign key relationships are also established by creating instance of the class e.g., Customer inside the Order class. In the same vein, a one to many relationship is also established in a similar manner but declaring it as a List type.

In order for the EF Core to generate our database appropriate, we must first establish a connection to our database using the DbContextOptionsBuilder:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=HelloEFCore1;Trusted_Connection=True;");
}
```

Figure 3

Then we use the DbSets to indicate to the EF Core how to set up our tables and naming them:

```
public class OrderContext : DbContext
{
    0 references
    public DbSet<Product> Products { get; set; }
    0 references
    public DbSet<Order> Orders { get; set; }

    0 references
    public DbSet<Customer> Customers { get; set; }
```

Figure 4

In this example, these three tables are generated when we migrate changes. In addition, a fourth table, the OrderItems table, is also generated in our case without creating a DbSet for it. This is because an Order contains a list of OrderItems and indeed we declared a list of OrderItems in the Order class. Therefore we optionally omit it and the EF Core automatically detects the parent child relationship (one to many).

4.2.Perform CRUD operation

To perform a create operation, i.e., inserting into our database, we will need to open a connection to our database by creating an instance of our Context using the following syntax, which is one way of doing it:

```
using (var db = new OrderContext())
{
    /*
```

Figure 5

We create a using method (part of the EF Core) and pass an instance of our Context. We then create our objects and add them to our context using the Add() method, then SaveChanges();

```
//Create
var p = new Product();
p.Title = "Widget";
p.Price = 20d;
db.Products.Add(p);

var o = new Order();
o.OrderItems = new List<OrderItem>();
o.Customer.CustomerName = "BOB";
o.OrderDate = DateTime.Now;

var oi1 = new OrderItem();
oi1.Order = o;
oi1.Product = p;
oi1.SalesPrice = 19d;
oi1.Quantity = 2;

o.OrderItems.Add(oi1);
db.Orders.Add(o);

db.SaveChanges();
```

Figure 6

Once the using method finishes execution and the program quits out of it, the database connection is automatically closed. The Read, Update and Delete operations follow similar procedure in opening and closing connections to the database.

To read from the database, we create an object to hold our data that will be retrieved from the database and using a for each loop to iterate through and access the individual rows.

```
//Read
var orders = db.Orders
    .Include(order => order.Customer)
    .Include(order => order.OrderItems)
    .ThenInclude(orderItem => orderItem.Product);


foreach (var order in orders)
{
    Console.WriteLine($"{order.OrderDate} - {order.Customer.CustomerName}");
}

Console.WriteLine();

foreach (var order in orders.Where(c => c.Customer.CustomerName == "Brian"))
{
    Console.WriteLine($"{order.OrderDate} - {order.Customer.CustomerName}");

    foreach (var orderItem in order.OrderItems)
    {
        Console.WriteLine($"{orderItem.Product.Title}   {orderItem.Quantity} * {orderItem.SalesPrice}");
    }
}
```

Figure 7

If a join is involved, we do an eager loading to load the other tables using the Include().

Updating and Deleting involves retrieving the particular row, and updating and deleting them respectively

```
//Update
var customer = (from c in db.Customers
                where c.CustomerName == "Brian"
                select c).Single();

customer.Address = "Pleinlann 2/105";
db.SaveChanges();


//Delete
var pro = (from p in db.Products
           where p.Title == "Logy"
           select p).Single();

db.Products.Remove(pro);
db.SaveChanges();
```

Figure 8

4.3.Making changes to the Entity Model

We use Data Annotations and the Fluent API to effect changes or make further configurations such enforcing constraints in our entity classes. Data Annotations are applied directly to the classes or class properties we intend to configure. The Data annotations, however, is just a subset of the fluent API and a lot more configuration can be done with the fluent without having to modifying the entity classes directly. The Fluent API is accessed by overriding the OnModelCreating method derived from the DbContext.

```
[Table(name:"Customer")]
2 references
public class Customer
```

Figure 9

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>().Property(t => t.Title).HasColumnName("ProductName");
}
```

Figure 10

5.Conclusion

Microsoft is making a number of changes to most of their frameworks to render them open source and cross-platform. The Entity framework is one of those frameworks in the form of the entity framework core which has been completely rewritten from scratch but maintains most of the “relevant” functionalities from the most recent release before the entity framework core (i.e., EF 6). In addition, the EF Core also has other functionalities making it simpler and easier to use. Being open source and cross-platform, the EF core and other such frameworks from Microsoft are projected to gain more popularity and prominence in the future. The easy syntax and the numerous abstractions being added to this new breed of the entity framework, i.e., the EF Core, developers are now sure to concentrate more on the important task of the business logic and make production faster. The performance of the EF Core has been proven to be relatively solid in most case but still needs to cope with larger applications, one of the main drawbacks of EF Core.

6.References:

- EntityFrameworkTutorial.net. (2016). Entity Framework Architecture. Retrieved December 16, 2017, from http://www.entityframeworktutorial.net/EntityFramework-Architecture.aspx
- learnentityframeworkcore.com. (2017). The Entity Framework Core DbSet - Learn Entity Framework Core. Retrieved December 17, 2017, from http://www.learnentityframeworkcore.com/dbset
- Microsoft. (2017). Entity Data Model | Microsoft Docs. Retrieved December 16, 2017, from https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/entity-data-model
- https://en.wikipedia.org/wiki/Object-relational_mapping
- "What is Object/Relational Mapping?". Hibernate Overview. JBOSS Hibernate. Retrieved 19 April 2011.
- Douglas Barry, Torsten Stanienda, "Solving the Java Object Storage Problem," Computer, vol. 31, no. 11, pp. 33-40, Nov. 1998