Microsoft

ORM & Entity Framework

Object Relational Mapping (ORM)

Object-Relational Mapping (ORM)

- O/RMs work by mapping between two worlds: the relational database, with its own API, and the object-oriented software world of classes and software code.
- **Popular ORM Tools:**
  - .Net: Entity Framework, NHibernate, Dapper.
  - Java: Hibernate, EclipseLink.
  - Python: SQLObject, SQLAlchemy

Why to use ORMs?

- It **speeds up** development time for teams.
- **Decreases the cost** of development.
- Handles the logic required to interact with databases.
- Improves security.
  - ORM tools are built to eliminate the possibility of **SQL injection** attacks.
- You write **less code** when using ORM tools than with SQL.

ORM in action

- The following is a simple example, written in C# code, to execute a query written in SQL using a database engine:

```
var sql = "SELECT id, first_name, last_name, phone, birth_date,
sex, age FROM persons WHERE id = 10";
var result = context.Persons.FromSqlRaw(sql).ToList();
var name = result[0]["first_name"];
```

- In contrast, the following makes use of an ORM API which makes it possible to write code that naturally makes use of the features of the language:

```
var person = repository.GetPerson(10);
var firstName = person.GetFirstName();
```

Entity Framework

Classical Programming Steps to Access a Database

Establish a Database Connection

```
var connection = new SqlConnection("Server=.;
Database=MyDb;User Id=sa;Password=your_password;");
connection.Open();
```

Create and Execute SQL Commands

```
var command = new SqlCommand("SELECT * FROM Products",
  connection);
var reader = command.ExecuteReader();
```

Classical Programming Steps to Access a Database

Read Data from the Result Set

```
while (reader.Read())
{
    var product = new Product
    {
        Id = (int)reader["Id"],
        Name = reader["Name"].ToString(),
        Price = (decimal)reader["Price"]
    };
    products.Add(product);
}
```

EF Core: Code-First vs Model-First Comparison

EF Code First

- Define your database schema (model) using C# classes
  - These are called **entity classes**
- Use EF Core to generate the database from those classes
  - This is done through EF Migrations
- Migrations
  - Define or modify your model (add/modify any class)
  - Migration is created, which captures the changes.
  - Migration is applied, updating the database schema.

DbContext

- **DbContext** is a class provided by EF Core that:
  - Manages the connection to the database.
  - Tracks changes to your entities.
  - Handles querying and saving data.
  - Applies configurations and mappings between your classes and the database schema.
- You typically create your own **class** that inherits from **DbContext**
- Your **class** contains DbSet<TEntity> properties for each entity in the model
  - `Entity is your class that maps to a table in DB`

DbContext (cont…)

Override the OnConfiguring(DbContextOptionsBuilder) method to configure the database (and other options) to be used for the context.

```
protected override void
    OnConfiguring(DbContextOptionsBuilder options)
            ⇒ options.UseSQLServer (“<conn_str>”)

protected override void
    OnConfiguring(DbContextOptionsBuilder options)
    {
        options.UseSqlServer("<conn_str>");
    }
```

Conn_str → connection string

EF Migrations

https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli

EF Core – Migrations

In real world projects:

- data models change as features get implemented
- new entities or properties are added and removed
- database schemas need to be changed accordingly to be kept in sync with the application.

The migrations feature in EF Core provides a way to:

- incrementally update the database schema to keep it in sync with the application's data model
- preserves existing data in the database during the updates

Migrations – How do they work?

When a data model change is introduced:

- the developer uses **EF Core tools** to add a corresponding migration describing the updates necessary to keep the database schema in sync.
- EF Core compares the **current model** against a **snapshot of the old model** to determine the differences
- generates migration source files that can be applied to the database

Once a new migration has been generated:

- it can be applied to a database in various ways (using EF Tools).
- EF Core records all applied migrations in a special history table in the database, allowing it to know which migrations have been applied and which haven't.

https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli

EF Core - Tools

`dotnet ef migrations add InitialCreate`

EF Core will create a directory called **Migrations** in your project, and generate some files. It's a good idea to inspect what exactly EF Core generated

`dotnet ef migrations add InitialCreate`

your application is ready to run on your new database, and you didn't need to write a single line of SQL. Note that this way of applying migrations is ideal for local development, but is less suitable for production environments

```
dotnet ef migrations add AddBlogCreatedTimestamp
```

https://learn.microsoft.com/en-us/ef/core/cli/dotnet

EF Core: Code-First vs Model-First Comparison

Add a Migration

```
dotnet ef migrations add AddProductTable
```

Apply Migrations to the Database

```
dotnet ef database update
```

Remove the Last Migration (if not applied)

```
dotnet ef migrations remove
```

Generate SQL Script from Migrations

```
dotnet ef migrations script
```

EF Approach (Code First Approach)

Installing EF Core Packages

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```