Entity Framework

We would like to use a relational database in our systems.

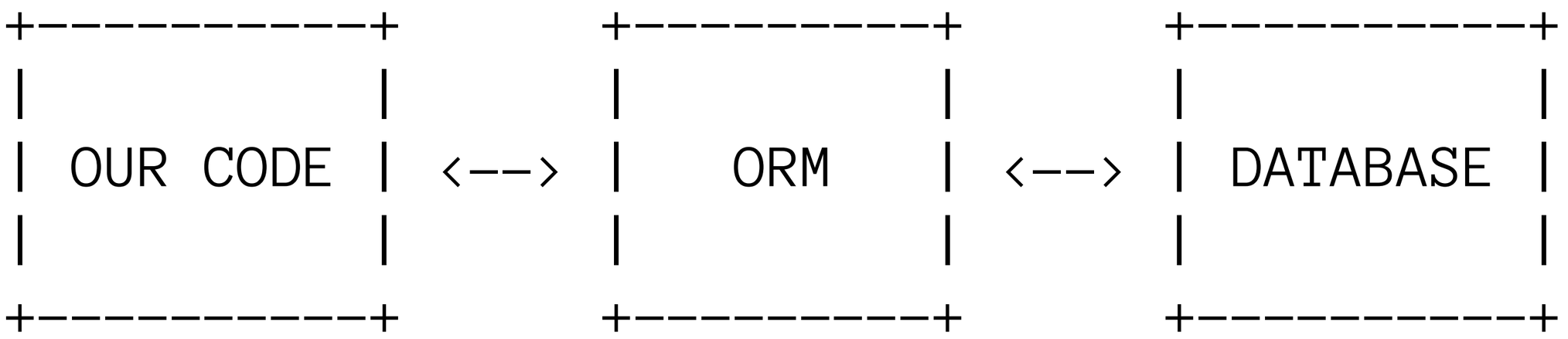
However, our systems are Object Oriented. We aren't coding in the landscape of tables and rows, but lists and objects.

Enter the Object Relational Mapper (ORM)

Luckily we have a concept called an Object Relational Mapper (ORM).

The purpose of an ORM is to seamlessly translate between the language of lists/objects to tables/rows.

With an ORM we can continue to use all of our familiar `C#` ideas and allow the ORM to take care of the details of the appropriate *SQL* and *relational database* work.

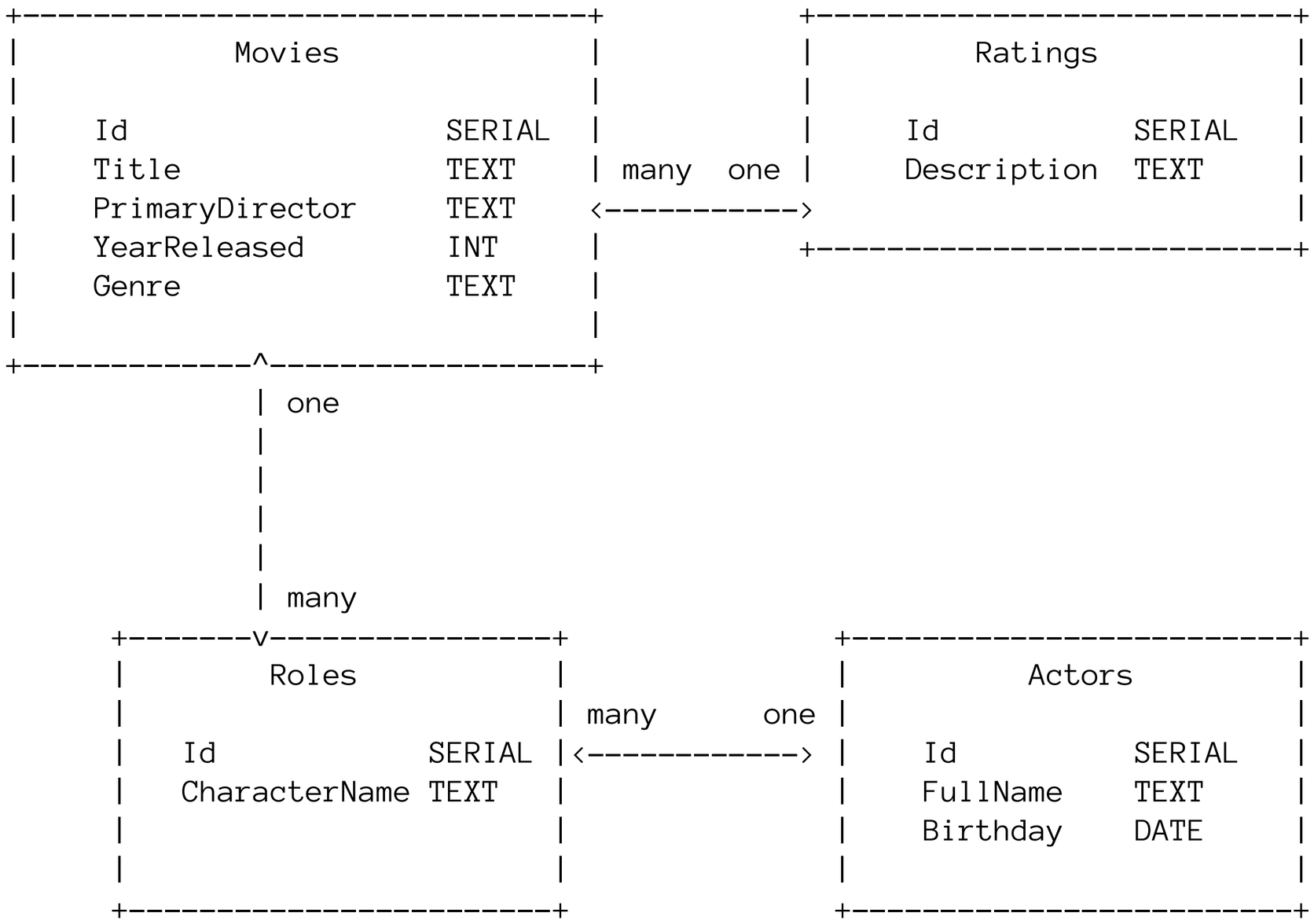

OUR CODE
<-->
ORM
<-->
DATABASE

Enter Entity Framework Core for .NET

In `.NET Core` we will be using the `Entity Framework` as our ORM of choice.

This will allow us to setup relationships between our database tables and our `C#` classes.

We will refer to these as `POCOs` -- Plain Old C-Sharp Objects.


Movies
Id SERIAL
Title TEXT
PrimaryDirector TEXT
YearReleased INT
Genre TEXT
many one
Ratings
Id SERIAL
Description TEXT
one
many
Roles
Id SERIAL
CharacterName TEXT
many one
Actors
Id SERIAL
FullName TEXT
Birthday DATE

Creating our first EF Core application

Start by making a plain console application so we can prompt the user if needed as well as output information.

```
dotnet new sdg-console -o SuncoastMovies
```

We will also need to add the Entity Framework library to our project.

This adds PostgreSQL support for Entity Framework.

```
cd SuncoastMovies
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
```

Define our first Model

We will begin by accessing the list of *movies* from our database from our `C#` code.

Before we can do that we need to teach `C#` about the structure of a movie.

We can do that by defining a `class` named `Movie` (singular) and give it properties corresponding to the same names we find in our table.

In our `Movie.cs` we can add `class Movie` as such:

```
using System;

namespace SuncoastMovies
{
  class Movie
  {
    public int Id { get; set; }
    public string Title { get; set; }
    public string PrimaryDirector { get; set; }
    public int YearReleased { get; set; }
    public string Genre { get; set; }
  }
}
```

Conventions

Notice how we have modeled the property names and types to match the column names and types in the database.

This is how `EF Core` will know how to **MAP** the information between our table and our objects.

Here we are following a *convention* that allows us to keep the same ideas between our object world and our database world.

Connecting to the database through a context

To connect our POCO models to the database we use a `Database Context`, or `DbContext` for short.

The purpose of the database context is to tell our code what models in the database are accessible as well as how to connect to the database.

SuncoastMoviesContext.cs

```
class SuncoastMoviesContext : DbContext
{
  public DbSet<Movie> Movies { get; set; }

  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    optionsBuilder.UseNpgsql("server=localhost;database=SuncoastMovies");
  }
}
```

Walk through

It is derived from `DbContext` which is provided by EF Core.

The `DbContext` is what allows us to connect to our database and relate our models to tables.

Walk through

We then define a property `Movies` (plural) of type `DbSet<Movie>`.

The `Movie` here is the model we wish to relate and `Movies` corresponds to the `Movies` table in our database.

`DbSet` will act much like our `List` collection but has much more knowledge of how to read and write from the database.

Walk through

Finally, we **override** a method required by `EF Core` that tells us how to connect to the database.

EF Core will call this method to setup the connection to the database.

The code in `OnConfiguring` tells EF to use a postgres database (`UseNpgsql`) named `SuncoastMovies` located on our local machine.

Using the context to get at data

```
// Get a new context which will connect to the database
var context = new SuncoastMoviesContext();
```

With this object we access our `Movies` property:

```
// This would represent the "movies" table in our database
context.Movies;
```

Using our Movies

Now with this `movies` object we can use some of our familiar `LINQ` capabilities.

We start by seeing how we would count the number of movies in the table.

```
var movieCount = context.Movies.Count();
Console.WriteLine($"There are {movieCount} movies!");
```

This should seem familiar as the `movies.Count()` statement eventually becomes a `SELECT COUNT(*) FROM MOVIES` in our database.

This is a perfect example of the idea of an ORM.

At the level of `C#` we are working with a `DbSet` and the `Count()` method from LINQ.

However, EF Core is translating this to the appropriate SQL statements and returning us the data we need.

Getting a list of all the movies

To see all of the movies, we can use a `foreach` loop:

```
foreach (var movie in context.Movies)
{
  Console.WriteLine($"There is a movie named {movie.Title}");
}
```

Again, translated to SQL this would be `SELECT * FROM MOVIES`. However, here we receive instances of our `Movie` class we can use to output information such as each movie object's title: `movie.Title`.

Bonus: Seeing what SQL our code is generating.

Before we can use the logger we must add another package to our application:

```
dotnet add package Microsoft.Extensions.Logging.Console
```

Then we can add this code to our `OnConfiguring` method:

```
var loggerFactory = LoggerFactory.Create(builder => builder.AddConsole());
optionsBuilder.UseLoggerFactory(loggerFactory);
```

Adding in related tables.

Our database has other tables we could add to our system: `Ratings`, `Roles`, and `Actors`.

Let's add a model for our ratings.

```
class Rating
{
  public int Id { get; set; }
  public string Description { get; set; }
}
```

Add the model to the context

```
public DbSet<Rating> Ratings { get; set; }
```

Update the Movie model

```
public int RatingId { get; set; }
public Rating Rating { get; set; }
```

Fetch movies and ratings

This tells the `Movie` model that it can use the `Rating` property to return a `Rating` object.

Now when we access the `context.Movies` we can also tell it to fetch the related `Rating` via the `Include` method.

```
var moviesWithRatings = context.Movies.Include(movie => movie.Rating);
```

Showing movies and ratings

```
var moviesWithRatings = context.Movies.Include(movie => movie.Rating);
foreach (var movie in moviesWithRatings)
{
  if (movie.Rating == null)
  {
    Console.WriteLine($"Movie {movie.Title} - no rating");
  }
  else
  {
    Console.WriteLine($"Movie {movie.Title} - {movie.Rating.Description}");
  }
}
```

Add in other relationships.

Finally, we can add the relationships for `Role` and `Actor`.

```
class Movie
{
  public int Id { get; set; }
  public string Title { get; set; }
  public string PrimaryDirector { get; set; }
  public int YearReleased { get; set; }
  public string Genre { get; set; }

  public int RatingId { get; set; }
  public Rating Rating { get; set; }

  public List<Role> Roles { get; set; }
}
```

Add in other relationships.

```
class Role
{
  public int Id { get; set; }
  public string CharacterName { get; set; }

  public int MovieId { get; set; }
  public Movie Movie { get; set; }

  public int ActorId { get; set; }
  public Actor Actor { get; set; }
}
```

```
// Makes a new collection of movies but each movie knows the associated Rating object
var moviesWithRatingsRolesAndActors = context.Movies.
                                      // from our movie, please include the associated Rating object
                                      Include(movie => movie.Rating).
                                      // ... and from our movie, please include the associated Roles LIST
                                      Include(movie => movie.Roles).
                                      // THEN for each of roles, please include the associated Actor object
                                      ThenInclude(role => role.Actor);

foreach (var movie in moviesWithRatingsRolesAndActors)
{
  if (movie.Rating == null)
  {
    Console.WriteLine($"{movie.Title} - not rated");
  }
  else
  {
    Console.WriteLine($"{movie.Title} - {movie.Rating.Description}");
  }

  foreach (var role in context.Movie.Roles)
  {
    Console.WriteLine($" - {role.CharacterName} played by {role.Actor.FullName}");
  }
}
```

Adding data

To add a `Movie` to the record we can make a new POCO like we always do.

```
var newMovie = new Movie {
  Title = "SpaceBalls",
  PrimaryDirector = "Mel Brooks",
  Genre = "Comedy",
  YearReleased = 1987,
  RatingId = 2
};
```

Saving

Then we can add this POCO to the `Movies` context and tell the context to save the changes

```
context.Movies.Add(newMovie);
context.SaveChanges();
```

Updating

```
var existingMovie = context.Movies.FirstOrDefault(movie => movie.Title == "SpaceBalls");

if (existingMovie != null) {
  existingMovie.Title = "SpaceBalls - the best movie ever";

  context.SaveChanges();
}
```

Deleting a movie

To remove a movie we find the movie and then remove it.

```
var existingMovieToDelete = context.Movies.FirstOrDefault(movie => movie.Title == "Cujo");

if (existingMovieToDelete != null) {
  context.Movies.Remove(existingMovieToDelete);

  context.SaveChanges();
}
```