Hello and welcome to the Introduction to Object-Relational Mapping for DBAs. During this presentation I’ll create a simple website and use that as the basis to discuss ORM. The presentation was created for DBAs with little to no software development experience.

Let’s take a quick poll, does anyone have software development experience? Written code? Created a website or application?

Anyone have any experience with an ORM such as Entity Framework, Hibernate/nHibernate, Active Record (Ruby on Rails), etc? Was your experience as a DBA dealing with ORM queries? How about writing software using an ORM?

Let us start off with a definition of ORM. There is Wikipedia definition but to me an ORM is one way or reading and persisting data in my application. There are may ways to read and persist data and ORM is just one of the ways. When I first started programming I used ADO datasets and I’m sure there was some technique before that. ORM as grown in popularity over the years because it is better then previous efforts to read/persist data in object-orientated applications. It’s not perfect but it’s currently the best technique that we have.

<Add wiki ORM definition>

An ORM maps relations data, which all you DBAs are familiar with, to objects. What is an object? Again, lets present the wiki definition.

<Add wiki object definition. Object is an instance of a class.>

An instance of a class? What is a class? A class defines the initial state of the object and its behaviour. You can think of state as data and the data can change over time. For example, we might have a car class. The car will have states such as colour, horn sound and current speed. It will also have behaviour such as accelerate and honk horn.

<Show example of object, maybe a car object>

Now you can instantiate different car objects such as red car and a blue car.

When an object is first created the class defines what the initial state of the object is. For example, the speed of the car could be zero at when first created. The class can also leave some state variables undefined and rely on the instansiater the set the state. For example, colour could be set buy the object creator to red.

Make sense? Any questions about classes and objects?

<Save often slide>

The problem is objects only exist in memory and disappear when an application is closed. We need to persist the state of our objects. In our example we might want to persist the cars the blue and red cars we created above. We might only need to persist some information, such as the color and horn sound but not the speed.

Obviously, a popular data persistence tool is a relational database. The problem is getting the object state data into a relational format. This is where ORMs enter the picture. They do the translation for us. We could write code to do the translation ourselves but why not use an existing framework?

<Slide about not repeating ourselves?>

Good developers are similar to good DBAs in that we don’t like to repeat ourselves. Good DBAs don’t have duplicate data (normalization) and good developers don’t duplicate code.

Enough with the slides lets get to some examples. Sorry, one more slide before the example. This slide shows some of the features or ORMs we will cover during the example.

<Slide with things being covered>

* Code First
* Migrations
* Convention over configuration
* Repository pattern
* Object caching and Lazy Loading
* Concurrency
* Unit of Work

Our example is a simple website that tracks the games our friends have played. Let me bring load application. This is a blank website with some notes on what we want on the home page.

<show application>

Let’s also bring up our database and show that there is nothing in it. Since I’m running on a Mac I need to SQL Server in a Docker container and I can’t use SQL Server Management Studio. Instead I’ll be using DataGrip which is similar to SQL Server Management Studio.

<show empty database>

Notice we don’t have any databases? Just the master one.

Since we want to track players and the games we play we will need a Players and Games tables. Let’s start with the Players table.

ORM works by mapping tables to objects. To make this work we need to create objects that are similar to the database tables. We call these objects models. In most cases the name of the class is the table name and the properties on the object are the fields.

<Create player model>

Simple model for now.

Most modern ORMs have conventions to save developers having to configure everything. Convention over configuration. For example, Entity Framework mapping assumes a property called ID is a primary key of the table. It also assumes that the model is singular (Player) and the table name is plural (Players). In the past you used to have to manually setup all the mappings. Anyone work with Hibernate XML file? I didn’t work with them very much but they scared me with all their crazy XML mapping logic.

Before we can do anything with this model we need to link it to the database using a Database Context. We also need to set the connection string.

<Create DBContext and set connection string. Use the sa password for connection string.>

We are going to work a bit backwards then usual. Instead of creating the database first then creating the code. To do this we need to install some command line tools for Entity Framework.

* Add Microsoft.EntityFrameworkCore.Tools.DotNet nuget package.
* Add the following to the project file:

<ItemGroup>

<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"

Version="2.0.2" />

</ItemGroup>

* In the project folder run dotnet restore
* Then dotnet ef should work.

To create the migration run:

* dotnet ef migrations create InitialCreate

This will create the Migrations folder and create the first migration file. Now we can run the migration to create the database and the Player table.

* dotnet ef database update

Now let us look at the newly created database.

<show database by opening properties in DataGrip and selecting the new schema>

Notice it created the database automagically for us. It also created the player table and the migrations history table. The player table is what you would expect, a table with two fields in it. One for the ID primary key and one for the Name. One problem we see right off the bat is the name is nvarchar(max). We probably want to limit the size of the name column. Lets go in and fix this.

<Set the max length of the column to 100>

<run dotnet ef database update 0, dotnet ef migrations remove>

<Set the column max length>

<dotnet ef migrations add CreatePlayerTable>

<dotnet ef database update>

Now that we have a player model and table let us do up a quick CRUD form. No need to watch me type this. We will use a code generator to some scaffolding code.

<Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package>

<dotnet aspnet cmd>

<don’t forget to reload the project you will get a blank page on the Players page>

Let us take a look at the code generated. It has all the CRUD method we would expect. Each method interacts with DB context to access the database. In all cases this code is taking all the values from the database and mapping them to our Player model.

I’ll quickly create the Games table and CRUD methods in a similar fashion. Pretty much exactly the same as the Player.

<Create the Games migration and CRUD methods>

Now that we have the two base tables lets figure out how to track the games that have been played. Any ideas on how to do that?

<Show ERD>

Lets start by creating the GameResultTypes table. This table says if you won, lost, tied a game.

<Create enum class>

In the code the different Game Result Types are represented as type safe enums. We could use integers instead but they aren’t type safe. For example, if we assign 0 to win, 1 to loss, and 2 to tie but that does not prevent a developer from accidently assigning a value of 3 which is undefined. Since enums are type safe you can’t assign “3”. The application won’t compile.

We want to seed the data so new developers don’t have to manually add it. It’s automagically added to the database.

<Seed the data and show the seeded data.>

I’m not a fan of using the primary key for the enum because the primary key can get out of sync between database instances. For example, if someone manually added and then deleted a Game Result Type on staging then your auto-increment IDs no longer line up with production. That is why I use the KeyCode value.

Now the KeyCode is set to the enum value. If change the enum values to different values and get different KeyCodes.

<Show example of changing enum keycode values>

I’ll create the CRUD methods for the Game Result Types but in this case we don’t want the user to be able to add or delete the types. They can only edit the human readable part of the type. The fastest way to do this for demo purposes is to run the scaffolding then delete stuff we don’t want.

<Generate CRUD for Game Result Types and delete add and delete>

Next, I’ll create the GamesPlayed table. Remember this links to the Games table and only holds the date. The players are in a separate table.

<Create the GamesPlayed table>

Notice for the model we link to the to the Games table. This will create a foreign key between the two tables.

<Create the CRUD methods>

Notice that the drop-down links to the Game ID, and not the name. Let’s fix that.

<Fix drop down and list. Do this by updating the ViewData in the controller.>

Now that the game name is fixed let’s create the GamePlayers table. This table says which player was involved in the game and if they won or not.

<Create the model but not the CRUD>

In this case we won’t create CRUD scaffolding. We only want to show the game players when showing a game played.

<Create the Game Players CRUD>

Now we have all the items we need to create