(Linq, EF)

Eng. Mahmoud Ouf

Lecture 3

Role of Entities

Entities are a conceptual model of a physical database that maps to your business domain.

This model is termed an entity data model (EDM).

The EDM is a client-side set of classes that are mapped to a physical database by Entity Framework convention and configuration.

The entities did not map directly to the database schema in so far as naming conventions go.

You are free to restructure your entity classes to fit your needs, and the EF runtime will map your unique names to the correct database schema.

The Role of the DbContext Class

The DbContext class represents a combination of the Unit of Work and Repository patterns that can be used to query from a database and group together changes that will be written back as a single unit of work.

DbContext provides a number of core services to child classes, including

- The ability to save all changes (which results in a database update),
- Tweak the connection string, delete objects, call stored procedures,
- Handle other fundamental details

Create a class that derives from DbContext for your specific domain.

In the constructor, pass the name of the connection string for this context class to the base class

The Role of DbSet<T>

To add tables into your context, you add a DbSet<T> for each table in your object model.

To enable lazy loading, the properties in the context need to be virtual.

Ex: public virtual DbSet<Customer> Customers { get; set; }

Each DbSet<T> provides a number of core services to each collection, such as creating, deleting, and finding records in the represented table.

The Role Navigation Properties

As the name suggests, navigation properties allow you to capture JOIN operations in the Entity Framework programming model

To account for these foreign key relationships, each class in your model contains virtual properties that connect your classes together

Ex: `public virtual ICollection<Order> Orders { get; set; }`

Lazy, Eager, and Explicit Loading

There are three ways that EF loads data into models. Lazy and Eager fetching are based on settings on the context, and the third, Explicit, is developer controlled.

Lazy Loading

The virtual modified allows EF to lazy load the data. This means that EF loads the bare minimum for each object and then retrieves additional details when properties are asked for in code.

Eager Loading

Sometimes you want to load all related records.

Lazy, Eager, and Explicit Loading

Explicit Loading

Explicit loading loads a collection or class that is referenced by a navigation property.

By default, it is set to Lazy Loading and we can re-enable it by:

```
context.Configuration.LazyLoadingEnabled = true;
```

To use Eager loading, set the LazyLoadingEnabled = false;

To use Explicit loading, use the Load method

Code First from Existing Database

Generating the Model

1. Create the solution for new application
2. (R.C.)Project=> Add New Item =>Select ADO.NET Entity Data Model
3. Then choose Add. This will launch the “ADO.NET Entity Data Model” Wizard
4. The wizard has 4 template:
    1. EF Designer from Database
    2. Empty EF Designer Model
    3. Empty Code First Model
    4. Code First from Database
5. Choose “Code First From Database”

Code First from Existing Database

new classes in your project:

one for each table that you selected in the wizard

one named ……Entities (the same name that you entered in the first step of the wizard).

By default, the names of your entities will be based on the original database object names; however, the names of entities in your conceptual model can be anything you choose.

You can change the entity name, as well as property names of the entity, by using special .NET attributes referred to as **data annotations**.

You will use data annotations to make some modifications to your model.

Code First from Existing Database

Data annotations:

Data annotations are series of attributes decorating the class and properties in the class

They instruct EF how to build your tables and properties when generating the database.

They also instruct EF how to map the data from the database to your model classes.

At the class level, the Table attribute specifies what table the class maps to.

At the property level, there are two attributes in use.

The Key attribute, this specifies the primary key for the table.

The StringLength attribute, which specifies the string length when generating the DDL for the field. This attribute is also used in validations,

Code First from Existing Database

Changing the default mapping

The [Table("Inventory")] attribute specifies that the class maps to the Inventory table. With this attribute in place, we can change the name of the class to anything we want.

Change the class name (and the constructor) to Car.

In addition to the Table attribute, EF also uses the Column attribute.

By adding the [Column("PetName")] attribute to the PetName property, we can change the name of the property to CarNickName.

Code First from Existing Database

Insert a Record (example)

```
private static int AddNewRecord()
{
        // Add record to the Inventory table of the AutoLot database.
        using (var context = new AutoLotEntities())
        {
                // Hard-code data for a new record, for testing.
                var car = new Car() { Make = "Yugo", Color = "Brown",
                CarNickName="Brownie"};
                context.Cars.Add(car);
                context.SaveChanges();
        }
        return car.CarId;
}
```

Code First from Existing Database

Selecting Record (example)

```
private static void PrintAllInventory()
{
        using (var context = new AutoLotEntities())
        {
                foreach (Car c in context.Cars)
                {
                        Console.WriteLine(“Name: “+ c.CarNickName);
                }
        }
}
```

Code First from Existing Database

Query with LINQ (example)

```
private static void PrintAllInventory()
{
        using (var context = new AutoLotEntities())
        {
                foreach (Car c in context.Cars.Where(c => c.Make == "BMW"))
                {
                        WriteLine(c);
                }
        }
}
```

Code First from Existing Database

Deleting Record (example)

```
private static void RemoveRecord(int carId)
{
        // Find a car to delete by primary key.
        using (var context = new AutoLotEntities())
        {
                Car carToDelete = context.Cars.Find(carId);
                if (carToDelete != null)
                {
                        context.Cars.Remove(carToDelete);
                        context.SaveChanges();
                }
        }
}
```

Code First from Existing Database

Updating Record (example)

```
private static void UpdateRecord(int carId)
{
        using (var context = new AutoLotEntities())
        {
                Car carToUpdate = context.Cars.Find(carId);
                if (carToUpdate != null)
                {
                        carToUpdate.Color = "Blue";
                        context.SaveChanges();
                }
        }
}
```