.NET App Dev Hands-On Lab

EF Lab 3 – DbContext, EF Core Migrations

This lab walks you through creating the `DbContext` and the `DbContextFactory` as well as running your first migration. Before starting this lab, you must have completed EF Lab 2. The lab works on the `AutoLot.Dal` project.

Begin by renaming the generated `Class1.cs` file to `GlobalUsings.cs`, and replace the scaffolded code with the following:

```
global using AutoLot.Models.Entities;
global using AutoLot.Models.Entities.Base;
global using AutoLot.Models.Entities.Configuration;
global using AutoLot.Models.ViewModels;
global using AutoLot.Models.ViewModels.Configuration;

global using Microsoft.Data.SqlClient;
global using Microsoft.EntityFrameworkCore;
global using Microsoft.EntityFrameworkCore.ChangeTracking;
global using Microsoft.EntityFrameworkCore.Design;
global using Microsoft.EntityFrameworkCore.Diagnostics;
global using Microsoft.EntityFrameworkCore.Metadata;
global using Microsoft.EntityFrameworkCore.Migrations;
global using Microsoft.EntityFrameworkCore.Query;
global using Microsoft.EntityFrameworkCore.Storage;
global using Microsoft.Extensions.DependencyInjection;

global using System.Data;
global using System.Linq.Expressions;
```

Part 1: Create the derived DbContext Class

The derived `DbContext` class is the hub of using EF Core with C#. This part builds the `ApplicationDbContext`.

Step 1: Create the ApplicationDbContext.cs file and its constructor

- Create a new folder named `EfStructures` in the `AutoLot.Dal` project. Add a new class to the folder named `ApplicationDbContext.cs`.
- Make the class `public` and inherit from `DbContext`. Add in a constructor that takes an instance of `DbContextOptions` and passes it to the base class:

```
namespace AutoLot.Dal.EfStructures;

public sealed class ApplicationDbContext : DbContext
{
  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
  {
  }
}
```

Step 2: Add the DbSet<T> properties

- Add a `DbSet<T>` for each of the model classes.

```
public DbSet<CreditRisk> CreditRisks { get; set; }
public DbSet<Customer> Customers { get; set; }
public DbSet<CustomerOrderViewModel> CustomerOrderViewModels { get; set; }
public DbSet<Car> Cars { get; set; }
public DbSet<Driver> Drivers { get; set; }
public DbSet<CarDriver> CarsToDrivers { get; set; }
public DbSet<Make> Makes { get; set; }
public DbSet<Order> Orders { get; set; }
public DbSet<Radio> Radios { get; set; }
public DbSet<SeriLogEntry> SeriLogEntries { get; set; }
```

Step 3: Add the OnModelCreating method and Register the Configuration Classes

- Add the override for `OnModelCreating`. This method is where the Fluent API code provides additional model information and where the configuration classes are registered.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
}
```


- Register the configuration classes:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  new CarConfiguration().Configure(modelBuilder.Entity<Car>());
  new DriverConfiguration().Configure(modelBuilder.Entity<Driver>());
  new CarDriverConfiguration().Configure(modelBuilder.Entity<CarDriver>());
  new RadioConfiguration().Configure(modelBuilder.Entity<Radio>());
  new CustomerConfiguration().Configure(modelBuilder.Entity<Customer>());
  new MakeConfiguration().Configure(modelBuilder.Entity<Make>());
  new CreditRiskConfiguration().Configure(modelBuilder.Entity<CreditRisk>());
  new OrderConfiguration().Configure(modelBuilder.Entity<Order>());
  new SeriLogEntryConfiguration().Configure(modelBuilder.Entity<SeriLogEntry>());
  new CustomerOrderViewModelConfiguration()
     .Configure(modelBuilder.Entity<CustomerOrderViewModel>());
  new CarViewModelConfiguration().Configure(modelBuilder.Entity<CarViewModel>());
}
```

Step 4: Add the Save Changes Event Handlers

- Update the constructor to handle the events for `SavingChanges`, `SavedChanges`, `SaveChangesFailed`:

```
public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
{
  SavingChanges += (sender, args) =>
  {
    string cs = ((ApplicationDbContext)sender)!.Database!.GetConnectionString();
    Console.WriteLine($"Saving changes for {cs}");
  };
  SavedChanges += (sender, args) =>
  {
    string cs = ((ApplicationDbContext)sender)!.Database!.GetConnectionString();
    Console.WriteLine($"Saved {args!.EntitiesSavedCount} changes for {cs}");
  };
  SaveChangesFailed += (sender, args) =>
  {
     Console.WriteLine($"An exception occurred! {args.Exception.Message} entities");
  };
}
```

Step 5: Add the ChangeTracker Event Handlers

- Update the constructor to assign handlers for the `Tracked` and `StateChanged` events:

```
public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
{
  //omitted for brevity
  ChangeTracker.Tracked += ChangeTracker_Tracked;
  ChangeTracker.StateChanged += ChangeTracker_StateChanged;
}
```


- Add the event handlers:

```
private void ChangeTracker_Tracked(object sender, EntityTrackedEventArgs e)
{
  var source = (e.FromQuery) ? "Database" : "Code";
  if (e.Entry.Entity is Car c)
  {
    Console.WriteLine($"Car entry {c.PetName} was added from {source}");
  }
}
private void ChangeTracker_StateChanged(object sender, EntityStateChangedEventArgs e)
{
  if (e.Entry.Entity is not Car c)
  {
    return;
  }
  var action = string.Empty;
  Console.WriteLine($"Car {c.PetName} was {e.OldState} before the state changed to {e.NewState}");
  switch (e.NewState)
  {
    case EntityState.Unchanged:
      action = e.OldState switch
          {
            EntityState.Added => "Added",
            EntityState.Modified => "Edited",
            _ => action
          };
      Console.WriteLine($"The object was {action}");
      break;
  }
}
```

Step 6: Update the GlobalUsings.cs file

- Add the following to the `GlobalUsings.cs` file:

```
global using AutoLot.Dal.EfStructures;
```

Part 2: Create the ApplicationDbContextFactory Class

The `IDesignTimeDbContextFactory` is used by the design time tools to instantiate a new instance of the `ApplicationDbContext`.

- Add a new class named `ApplicationDbContextFactory.cs` to the `EfStructures` folder. Make the class public and inherit from `ApplicationDbContextFactory<T>` where T is the `ApplicationDbContext` class and implement the interface (the `CreateDbContext()` method):

```
namespace AutoLot.Dal.EfStructures;

public class ApplicationDbContextFactory : IDesignTimeDbContextFactory<ApplicationDbContext>
{
  //class implementation goes here
}
```

- The `CreateDbContext()` method creates a new instance of `ApplicationDbContext` using a hard-coded, development connection string (**NOTE:** Update your connection string to fit your environment):

```
public ApplicationDbContext CreateDbContext(string[] args)
{
  var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
  var cs = @"server=(localdb)\MsSqlLocalDb;Database=AutoLot_Hol;Integrated Security=true";
  //var cs = @"Server=(localdb)\ProjectModels;Database=AutoLot_Hol;Trusted_Connection=True;";
  //var cs = @"server=.,5433;Database=AutoLot_Hol;User Id=sa;Password=P@ssw0rd;Encrypt=false;";
  optionsBuilder.UseSqlServer(cs);
  //optionsBuilder.UseSqlServer(cs, options => options.EnableRetryOnFailure());
  optionsBuilder.ConfigureWarnings(cw => cw.Ignore(RelationalEventId.BoolWithDefaultWarning));
  Console.WriteLine(cs);
  return new ApplicationDbContext(optionsBuilder.Options);
}
```


Part 3: Update the Database Using EF Core Migrations

Migrations are created and executed using the .NET Core EF Command Line Interface. The commands must be executed from the same directory as the `AutoLot.Dal.csproj` file.

The NuGet style commands can be used in the Package Manager Console in Visual Studio if the `Microsoft.EntityFrameworkCore.Tools` package was installed.

Step 1: Create and Execute the Initial Migration

- Run the following command if you currently have a previous version of the EF Core Global Tool installed. This will uninstall the version on your machine:

```
dotnet tool uninstall --global dotnet-ef
```

- Run the following command to install the EF Core Global Tooling version 6.0:

```
dotnet tool install --global dotnet-ef --version 6.0.0
```

- Alternately, you can update the tooling to the latest version (including prelease versions) with the following command:

```
dotnet tool update --global dotnet-ef --prerelease
```

Step 2: Create and Execute the Initial Migration

- Open a command prompt in the same directory as the `AutoLot.Dal` project
  OR
  [Visual Studio]Open Package Manager Console (View -> Other Windows -> Package Manager Console) and navigate to the correct directory using:

```
[Windows]cd .\AutoLot.Dal
[Non-Windows]cd ./AutoLot.Dal
```

- Create the initial migration with the following command (-o = output directory, -c = Context File):

```
[Windows]
NOTE: The following lines must be entered as one line – copying and pasting from this document doesn't work without removing the line break
dotnet ef migrations add Initial -o EfStructures\Migrations -c
AutoLot.Dal.EfStructures.ApplicationDbContext
NOTE: The above lines must be entered as one line – copying and pasting from this document doesn't work without removing the line break
[Non-Windows]
NOTE: The following lines must be entered as one line – copying and pasting from this document doesn't work without removing the line break
dotnet ef migrations add Initial -o EfStructures/Migrations -c
AutoLot.Dal.EfStructures.ApplicationDbContext
NOTE: The above lines must be entered as one line – copying and pasting from this document doesn't work without removing the line break
```

- This creates three files in the `EfStructures\Migrations (EfStructures/Migrations)` Directory:

```
A file named YYYYMMDDHHmmSS_Initial.cs (where date time is UTC)
A file named YYYYMMDDHHmmSS _Initial.Designer.cs (same numbers)
ApplicationDbContextModelSnapshot.cs
```

- Open up the `YYYYMMDDHHmmSS _Initial.cs` file. Check the Up and Down methods to make sure the database and table/column creation code is there
- Update the database with the following command:

```
dotnet ef database update
```

- Examine your database in SQL Server Management Studio to make sure the tables were created

Summary

In this lab, you created the `ApplicationDbContext` and the `ApplicationDbContextFactory`. The final step was creating the initial migration and updating the database.

Next steps

In the next part of this tutorial series, you will create the SQL Server objects, including a stored procedure, two views, and a user defined function.

