Build an EF and ASP.NET Core 3.0 App HOL

Lab 3

This lab walks you through creating the SQL Server view, the custom exceptions, and support for change tracking events. Prior to starting this lab, you must have completed Lab 2.

Part 1: Create the SQL Server View

The SQL Server view will be created using the EF Core migration framework. This enables a single call to the EF Core command line to update the database to the necessary state.

The SQL calls are added into migrations using static helper classes that leverages an instance of the `MigrationBuilder` class. The `MigrationBuilder` `Sql` method executes raw SQL against the target database.

Step 1: Create the Helper Class

1) Create a new class named `MigrationHelpers.cs` under the `EfStructures` folder in the `AutoLot.Dal` project.
2) Make the class `public` and `static`, and add the following using statements to the top:

```
using Microsoft.EntityFrameworkCore.Migrations;
```

3) Add the following method that uses the migration builder to create the view. . This will be called by the `Up` method of the migration

```
public static void CreateView(MigrationBuilder migrationBuilder)
{
  var sql = @"
CREATE VIEW [dbo].[CustomerOrderView]
AS
SELECT dbo.Customers.FirstName, dbo.Customers.LastName, dbo.Inventory.Color,
dbo.Inventory.PetName, dbo.Makes.Name AS Make
FROM   dbo.Orders
INNER JOIN dbo.Customers ON dbo.Orders.CustomerId = dbo.Customers.Id
INNER JOIN dbo.Inventory ON dbo.Orders.CarId = dbo.Inventory.Id
INNER JOIN dbo.Makes ON dbo.Makes.Id = dbo.Inventory.MakeId ";
  migrationBuilder.Sql(sql);
}
```

4) Add another method to drop the view. This will be called by the `Down` method of the migration.

```
public static void DropView(MigrationBuilder migrationBuilder)
{
  var sql = @"DROP VIEW [dbo].[CustomerOrderView]";
  migrationBuilder.Sql(sql);
}
```

Step 2: Create the Migration for the SQL Server View

Even if nothing has changed in the model, migrations can still be created, but the `Up` and `Down` methods will be empty. To execute custom SQL, that is exactly what is needed.

1) Open a command prompt or Package Manager Console in the `AutoLot.Dal` directory.
2) Create an empty migration (but do **NOT** run `dotnet ef database update`) by running the following command. Note that the output directory will be the same as previous migrations for the same derived DbContext:

```
dotnet ef migrations add AddSQL -c AutoLot.Dal.EfStructures.ApplicationDbContext
```

3) Open up the new migration file (named `<timestamp>_AddSQL.cs`). Note that the `Up` and `Down` methods are empty. Add the following using statement to the top of the file:

```
using AutoLot.Dal.EfStructures.MigrationHelpers;
```

4) Change the Up method to the following:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
  MigrationHelpers.CreateView(migrationBuilder);
}
```

5) Change the Down method to the following code:

```
protected override void Down(MigrationBuilder migrationBuilder)
{
  MigrationHelpers.DropView(migrationBuilder);
}
```

6) SAVE THE MIGRATION FILE BEFORE RUNNING THE MIGRATION
7) Update the database by executing the migration:

```
dotnet ef database update
```

8) Check the database to make sure the view exists

Part 2: Add the Custom Exceptions

A common pattern in exception handling is to wrap system exceptions with custom exceptions. The AutoLot application uses three (3) custom exceptions, including a base custom exception.

Step 1: The Base Custom Exception

1) Create a new directory named Exceptions in the AutoLot.Dal project. In this directory add a new class named CustomException.cs. Add the following using statement to the top of the class:

```
using system;
```

2) Update the code to the following:

```
public class CustomException : Exception
{
  public CustomException()
  {
  }

  public CustomException(string message) : base(message)
  {
  }

  public CustomException(string message, Exception innerException)
    : base(message, innerException)
  {
  }
}
```

Step 2: The Custom Concurrency Exception

1) Add a new class named CustomConcurrencyException.cs. Add the following using statement to the top of the class:

```
using Microsoft.EntityFrameworkCore;
```

2) Update the code to the following:

```
public class CustomConcurrencyException : CustomException
{
  public CustomConcurrencyException()
  {
  }

  public CustomConcurrencyException(string message) : base(message)
  {
  }

  public CustomConcurrencyException(string message, DbUpdateConcurrencyException innerException)
    : base(message, innerException)
  {
  }
}
```

Step 3: The Custom Retry Limit Exceeded Exception

1) Add a new class named CustomRetryLimitExceededException.cs. Add the following using statement to the top of the class:

```
using Microsoft.EntityFrameworkCore;
```

2) Update the code to the following:

```
public class CustomConcurrencyException : CustomException
{
  public CustomConcurrencyException()
  {
  }

  public CustomConcurrencyException(string message) : base(message)
  {
  }

  public CustomConcurrencyException(string message, RetryLimitExceededException innerException)
    : base(message, innerException)
  {
  }
}
```

Step 4: The Custom Db Update Exception

3) Add a new class named CustomDbUpdateException.cs. Add the following using statement to the top of the class:

```
using Microsoft.EntityFrameworkCore;
```

4) Update the code to the following:

```
public class CustomDbUpdateException : CustomException
{
  public CustomDbUpdateException()
  {
  }

  public CustomDbUpdateException(string message) : base(message)
  {
  }

  public CustomDbUpdateException(string message, DbUpdateException innerException)
    : base(message, innerException)
  {
  }
}
```

Part 3: Add Support for Change Tracking Events

The ChangeTracker exposes two events that can be useful for auditing purposes.

Step 1: The Entity Tracked Event

This event is fired when an entity is added to change tracking.

1) Open the ApplicationDbContext class (in the AutoLot.Dal project) and add the following using statements:

```
Using  Microsoft.EntityFrameworkCore.ChangeTracking;
```

2) Add the following code to the constructor:

```
public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options): base(options)
{
  ChangeTracker.Tracked += ChangeTracker_Tracked;
}
```

3) Add the following event handler:

```
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}");
  }
}
```

Step 2: The Entity State Changed Event

This event is fired when an entity's state is changed.

1) Add the following code to the constructor:

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

2) Add the following event handler:

```
private void ChangeTracker_StateChanged(object? sender, EntityStateChangedEventArgs e)
{
  if (!(e.Entry.Entity is Car c))
  {
    return;
  }
  var action = string.Empty;
  Console.WriteLine($"Blog {c.PetName} was {e.OldState} before the state changed to
{e.NewState}");
  switch (e.NewState)
  {
    case EntityState.Added:
    case EntityState.Deleted:
    case EntityState.Modified:
    case EntityState.Unchanged:
      switch (e.OldState)
      {
        case EntityState.Added:
          action = "Added";
          break;
        case EntityState.Deleted:
          action = "Deleted";
          break;
        case EntityState.Modified:
          action = "Edited";
          break;
        case EntityState.Detached:
        case EntityState.Unchanged:
          break;
        default:
          throw new ArgumentOutOfRangeException();
      }
      Console.WriteLine($"The object was {action}");
      break;
    case EntityState.Detached:
      break;
    default:
      throw new ArgumentOutOfRangeException();
  }
}
```

Summary

This lab created the SQL Server view and custom exceptions, and support for the change tracker events.

Next steps

In the next part of this tutorial series, you will create the repositories.