

# Build an ASP.NET Core MVC App with EF Core

## One-Day Hands-On Lab

### Lab 4

This lab walks you through creating custom exceptions, overriding `SaveChanges`, implementing the event handlers for the Change Tracker, and adding a SQL Server view to the database. As a final step, you will allow the test project access to internal project items. Prior to starting this lab, you must have completed Lab 3.

## Part 1: Add the Custom Exceptions

A common pattern in exception handling is to wrap system exceptions with custom exceptions. The AutoLot Data Access Layer uses three (5) custom exceptions with a base custom exception.

### Step 1: Create the Base Custom Exception

- Create a new folder in the `AutoLot.Dal` project named `Exceptions` and add a new class to the folder named `CustomException.cs`. Update the code to the following:

```
namespace AutoLot.Dal.Exceptions;
```

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

### Step 2: Create the Remaining Exceptions

- Add three more files to the `Exceptions` directory: `CustomConcurrencyException.cs`, `CustomDbUpdateException.cs`, `CustomRetryLimitExceededException.cs`.

- Update the each of the exceptions to the following:

```
// CustomConcurrencyException.cs
namespace AutoLot.Dal.Exceptions;

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

// CustomDbUpdateException.cs
namespace AutoLot.Dal.Exceptions;
public class CustomDbUpdateException : CustomException
{
    public CustomDbUpdateException(){}
    public CustomDbUpdateException(string message) : base(message) { }
    public CustomDbUpdateException(string message, DbUpdateException innerException)
        : base(message, innerException) { }
}

// CustomRetryLimitExceededException.cs
namespace AutoLot.Dal.Exceptions;
public class CustomRetryLimitExceededException : CustomException
{
    public CustomRetryLimitExceededException() {}
    public CustomRetryLimitExceededException(string message) : base(message) { }
    public CustomRetryLimitExceededException(
        string message, RetryLimitExceededException innerException): base(message, innerException)
    {
    }
}
```

- Add the following global using statement to the GlobalUsings.cs class:

```
global using AutoLot.Dal.Exceptions;
```

## Part 2: Override Save Changes

Overriding save changes in the `ApplicationDbContext` class allows for encapsulation of error handling. Note that this example only overrides one of the `SaveChanges` methods, the other three would be handled in a similar fashion.

- Update the code to the following:

```
public override int SaveChanges()
{
    try
    {
        return base.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        //A concurrency error occurred
        //Should log and handle intelligently
        throw new CustomConcurrencyException("A concurrency error happened.", ex);
    }
    catch (RetryLimitExceededException ex)
    {
        //DbResiliency retry limit exceeded
        //Should log and handle intelligently
        throw new CustomRetryLimitExceededException("There is a problem with SQL Server.", ex);
    }
    catch (DbUpdateException ex)
    {
        //Should log and handle intelligently
        throw new CustomDbUpdateException("An error occurred updating the database", ex);
    }
    catch (Exception ex)
    {
        //Should log and handle intelligently
        throw new CustomException("An error occurred updating the database", ex);
    }
}
```

## Part 3: Create the SQL Server Objects

As a pattern, if all SQL Server objects are created using the EF Core migration framework, a single call to the EF Core command line updates the database to the necessary state.

### Step 1: Create the Helper Class to Create/Drop SQL Server Objects

- Add a class named `MigrationHelpers.cs` to the `EfStructures` folder, and make the class public and static:

```
public static class MigrationHelpers
{
}
```

### Step 2: Create the SQL Server View Create and Drop Functions

- The create will be called in the Up method of the migration:

```
public static void CreateCustomerOrderView(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql(@"exec (N'
        CREATE VIEW [dbo].[CustomerOrderView]
        AS
        SELECT c.FirstName, c.LastName, i.Color, i.PetName,
            i.DateBuilt, i.IsDrivable, i.Price, i.Display, m.Name AS Make
        FROM dbo.Orders o
        INNER JOIN dbo.Customers c ON c.Id = o.CustomerId
        INNER JOIN dbo.Inventory i ON i.Id = o.CarId
        INNER JOIN dbo.Makes m ON m.Id = i.MakeId')")
    );
}
```

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

```
public static void DropCustomerOrderView(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("EXEC (N' DROP VIEW [dbo].[CustomerOrderView] ')");
}
```

### Step 3: Create the SQL Server Stored Procedure Create and Drop Functions

- The create will be called in the Up method of the migration:

```
public static void CreateSproc(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql(@"exec (N'
        CREATE PROCEDURE [dbo].[GetPetName] @carID int, @petName nvarchar(50) output
        AS
        SELECT @petName = PetName from dbo.Inventory where Id = @carID')")
    );
}
```

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

```
public static void DropSproc(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("EXEC (N' DROP PROCEDURE [dbo].[GetPetName]')");
}
```

## Step 4: Create the SQL Server Functions Create and Drop Functions

- The create will be called in the Up method of the migration:

```
public static void CreateFunctions(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql(@"exec (N'
        CREATE FUNCTION [dbo].[udtf_GetCarsForMake] ( @makeId int )
        RETURNS TABLE
        AS
        RETURN
        (
            SELECT Id, IsDrivable, DateBuilt, Color, PetName, MakeId, TimeStamp, Display, Price
            FROM Inventory WHERE MakeId = @makeId
        )'")
);

migrationBuilder.Sql(@"exec (N'
        CREATE FUNCTION [dbo].[udf_CountOfMakes] ( @makeid int )
        RETURNS int
        AS
        BEGIN
            DECLARE @Result int
            SELECT @Result = COUNT(makeid) FROM dbo.Inventory WHERE makeid = @makeid
            RETURN @Result
        END'")
);
}
```

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

```
public static void DropFunctions(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("EXEC (N' DROP FUNCTION [dbo].[udtf_GetCarsForMake]')");
    migrationBuilder.Sql("EXEC (N' DROP FUNCTION [dbo].[udf_CountOfMakes]')");
}
```

## Step 5: Create the Migration for the SQL Server Objects

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

- Open a command prompt or Package Manager Console in the AutoLot.Dal directory. Create an empty migration (but do **NOT** run `dotnet ef database update`) by running the following command:

```
[Windows]dotnet ef migrations add CustomSql -c AutoLot.Dal.EfStructures.ApplicationDbContext
[Non-Windows]dotnet ef migrations add CustomSql -c AutoLot.Dal.EfStructures.ApplicationDbContext
```

**Note:** After the first migration for a context, the same output directory will be used in subsequent migrations, so it can be left off the command.

- Open the new migration file (named <timestamp>\_CustomSql.cs). Note that the Up and Down methods are empty. Change the Up method to the following:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    MigrationHelpers.CreateCustomerOrderView(migrationBuilder);
    MigrationHelpers.CreateSproc(migrationBuilder);
    MigrationHelpers.CreateFunctions(migrationBuilder);
}
```

- Change the Down method to the following code:

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    MigrationHelpers.DropCustomerOrderView(migrationBuilder);
    MigrationHelpers.DropSproc(migrationBuilder);
    MigrationHelpers.DropFunctions(migrationBuilder);
}
```

### • SAVE THE MIGRATION FILE BEFORE RUNNING THE MIGRATION

- Update the database by executing the migration:

```
dotnet ef database update
```

- Check the database to make sure the view, sproc, and functions exist

## Part 4: Map the SQL Functions to C# Functions

- Map the udf\_CountOfMakes SQL Server function to a C# function in the ApplicationDbContext class:

```
[DbFunction("udf_CountOfMakes", Schema = "dbo")]
public static int InventoryCountFor(int makeId) => throw new NotSupportedException();
```

- Map the udf\_GetCarsForMake SQL Server function to a C# function in the ApplicationDbContext class:

```
[DbFunction("udtf_GetCarsForMake", Schema = "dbo")]
public IQueryable<Car> GetCarsFor(int makeId) => FromExpression(() => GetCarsFor(makeId));
```

## Part 5: Allow the Test Project Access to Internals

- Add a new class named LibraryAttributes.cs to the project root folder. Add the following using statements to the class:

```
using System.Runtime.CompilerServices;
```

- Update the code to the following:

```
[assembly: InternalsVisibleTo("AutoLot.Dal.Tests")]
```

## Summary

This lab created the custom exceptions, implemented the `SaveChanges()` override, added SQL Server objects to the database, and allowed the `AutoLot.Dal.Tests` test project access to internal project items.

## Next steps

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