.NET 10 App Dev Hands-On Lab

EF Lab 4 – Custom Exceptions, SaveChanges(), SQL Server Objects

This lab walks you through creating custom exceptions, overriding the `SaveChanges` method, adding a SQL Server view to the database, and exposing the internal methods to the test project. Before starting this lab, you must have completed EF Lab 3.

Part 1: Add the Custom Exceptions

A typical exception-handling pattern is wrapping system exceptions with custom exceptions.

Copilot Agent Mode

```
Setup Prompt: Always use file scoped namespaces. Always combine attributes on a single line when
possible. The project does not use nullable reference types. There is a GlobalUsings.cs file that
includes common usings, don't include using statements in new files if they are already in the
globalusings.cs file. I prefer expression bodied members when possible. Single line if statements
should still use braces. Use ternary operators when appropriate. Use internal over private. All
classes and methods are public unless told otherwise. Don't add a constructor unless instructed to
do so. Use primary constructors when possible and don't declare a class level variable if the
parameter from the constructor can be used. Don't initialize properties unless instructed to do
so.

Prompt: In the Autolot.Models project, create a new folder named Exceptions. In that folder,
create a folder named Base. In that folder, create a class named CustomException, inherit from
Exception, and implement all of the public constructors.

Prompt: Add the following global usings to the GlobalUsings.cs file in the AutoLot.Dal and the
AutoLot.Models projects if they do not already exist (sorted alphabetically. Don't remove any
existing global using statements).
global using AutoLot.Models.Exceptions;
global using AutoLot.Models.Exceptions.Base;

Prompt: In the Autolot.Dal project, create a new folder named Exceptions. In that folder, create
three new files, CustomConcurrencyException, CustomDbUpdateException, and
CustomRetryLimitExceededException. All of them need to inherit from CustomException.
Implement the default constructors. The constructor that takes an inner exception should use
DbUpdateConcurrencyException, DbUpdateException, and RetryLimitExceededException, respectively as
the inner exception type.

Prompt: Add the following global usings to the GlobalUsings.cs file in the AutoLot.Dal and the
AutoLot.Models projects if they do not already exist (sorted alphabetically. Don't remove any
existing global using statements).
global using AutoLot.Dal.Exceptions;
```

Manual

Step 1: Create the Base Custom Exception in the Models Project

- Create a new folder in the AutoLot.Models project named Exceptions, and in that folder, add a new folder named Base. In the Base folder, add a new class named CustomException.cs. and update the code to the following:
  NOTE: The base exception should be in a project referenced by all other projects. In this workshop, that is the AutoLot.Models project.

```
namespace AutoLot.Models.Exceptions.Base;
public class CustomException : Exception
{
  public CustomException() {}
  public CustomException(string message) : base(message) {}
  public CustomException(string message, Exception innerException)
    : base(message, innerException) {}
}
```

- Add the following to the GlobalUsings.cs class in the AutoLot.Models and AutoLot.Dal projects:

```
global using AutoLot.Models.Exceptions;
global using AutoLot.Models.Exceptions.Base;
```

Step 2: Create the DAL Specific Exceptions

- Create a new folder in the AutoLot.Dal project named Exceptions and add three files to the folder: CustomConcurrencyException.cs, CustomDbUpdateException.cs, CustomRetryLimitExceededException.cs.
  Update 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: Save Changes and Concurrency Errors

- Add a method to explore the `DbUpdateConcurrencyException` to the `ApplicationDbContext` class:

```
internal void DemoConcurrencyException(DbUpdateConcurrencyException ex)
{
  //A concurrency error occurred - Should log and handle intelligently
  Console.WriteLine(ex.Message);
  EntityEntry entryEntity = ex.Entries[0];
  //Kept in DbChangeTracker
  PropertyValues originalValues = entryEntity.OriginalValues;
  PropertyValues currentValues = entryEntity.CurrentValues;
  IEnumerable<PropertyEntry> modifiedEntries = entryEntity.Properties.Where(e => e.IsModified);
  foreach (var itm in modifiedEntries)
  {
    //Console.WriteLine($"{itm.Metadata.Name},");
  }
  //Needs to call to database to get values
  PropertyValues databaseValues = entryEntity.GetDatabaseValues();
  //Discards local changes, gets database values, resets change tracker
  //entryEntity.Reload();
}
```

- Overriding save changes in the `ApplicationDbContext` class allows for error-handling encapsulation. Add the following to the `ApplicationDbContext` class:

```
public override int SaveChanges()
{
  try
  {
    return base.SaveChanges();
  }
  catch (DbUpdateConcurrencyException ex)
  {
    DemoConcurrencyException(ex);
    throw new CustomConcurrencyException("A concurrency error happened.", ex);
  }
  catch (RetryLimitExceededException ex)
  {
    throw new CustomRetryLimitExceededException("There is a problem with SQL Server.", ex);
  }
  catch (DbUpdateException ex)
  {
    throw new CustomDbUpdateException("An error occurred updating the database", ex);
  }
  catch (Exception ex)
```

```
  {
    throw new CustomException("An error occurred updating the database", ex);
  }
}
```

Part 3: Create the SQL Server Objects

In general, if all SQL Server objects are created using EF Core migrations, a single call to the EF Core command-line tool updates the database to the correct 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`:

```
namespace AutoLot.Dal.EfStructures;
public static class MigrationHelpers
{
  //implementation goes here
}
```

Step 2: 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 procedure. This will be called in the `Down` method of the migration.

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


Step 3: 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 functions. This will be called in 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 4: 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. **MAKE SURE ALL FILES ARE SAVED**

- 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.CreateSproc(migrationBuilder);
  MigrationHelpers.CreateFunctions(migrationBuilder);
}
```

- Change the Down method to the following code:

```
protected override void Down(MigrationBuilder 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 sproc, and functions exist
- You can create a script of the migrations by running the following CLI command:

```
dotnet ef migrations script -o allmigrations.sql -i
```


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 The FromExpression call in the CLR function body allows for the function to be used instead of a regular DbSet:

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

- The MakeTests.cs class in Lab 7 demonstrates using these functions.

Part 5: Allow the Test Project Access to Internals

- Add a new class named LibraryAttributes.cs to the project root folder. Clear out the scaffolded code and add the following to the class:

```
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("AutoLot.Dal.Tests")]
```

Summary

This lab created custom exceptions, implemented the SaveChanges() override, added SQL Server objects to the database, and enabled 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.