

Build an ASP.NET Core Service and App with .NET (Core) 5.0 Two-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.
- Add a new class to the folder named CustomException.cs
- Update the code to the following:

```
using System;
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
using Microsoft.EntityFrameworkCore;
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
using Microsoft.EntityFrameworkCore;
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
using System;
using Microsoft.EntityFrameworkCore.Storage;
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)
        {
        }
    }
}
```

Part 2: Override Save Changes

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

- Add the following using statement to the ApplicationDbContext.cs class:

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

- 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: Implement the ChangeTracker Event handlers

Step 1: Assign and handle the Tracked event

- Add the following to the ApplicationDbContext constructor:

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

- 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: Assign and handle the StateChanged event

- Add the following to the ApplicationDbContext constructor:

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

- 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($"Car {c.PetName} was {e.OldState} before the state changed to {e.NewState}");
    switch (e.NewState)
    {
        case EntityState.Added:
        case EntityState.Deleted:
        case EntityState.Modified:
        case EntityState.Unchanged:
            switch (e.OldState)
            {
                case EntityState.Added:
                    action = "Added";
                    break;
                case EntityState.Deleted:
                    action = "Deleted";
                    break;
                case EntityState.Modified:
                    action = "Edited";
                    break;
                case EntityState.Detached:
                case EntityState.Unchanged:
                    break;
                default:
                    throw new ArgumentOutOfRangeException();
            }
            Console.WriteLine($"The object was {action}");
            break;
        case EntityState.Detached:
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

Part 4: Create the SQL Server View

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 for the View

- Add a class named `MigrationHelpers.cs` to the `EfStructures` folder.
- Make the class public and static, and add the following using statements to the top:

```
using Microsoft.EntityFrameworkCore.Migrations;
public static class MigrationHelpers
{
}
```

Step 2: Create the 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 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')")
    );
}
```

- 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 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. The must be executed all on one line:
Note: For subsequent migrations, the output directory will be the same as previous migration, so it can be left off the command. I leave it in for clarity.

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

- Open the new migration file (named `<timestamp>_CustomSql.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;
```

- Change the Up method to the following:

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

- Change the Down method to the following code:

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    MigrationHelpers.DropCustomerOrderView(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 exists

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 and the event handlers for the Change Tracker, added a SQL Server view 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.