**Calling Stored Procedures with Entity Framework in .Net 6**

Working with stored procedures using the Entity Framework (EF) is a challenge because depending on what the stored procedure does determines how you call it using EF. This inconsistency, along with the names of the methods you call to submit a stored procedure changing from one version of EF to the next, can lead to much frustration for developers. This article won't solve these issues, but it will show you how to make calls to stored procedures using the version of Entity Framework in .NET 5.x. You're going to learn how to retrieve data from a stored procedure, how to pass parameters, return a scalar value, modify data, and handle multiple result sets.

Of course, as I've been preaching for the past 20+ years, you should create your own classes as a wrapper around EF to make calls to stored procedures. If you do, as you move from one version of .NET to the next, all you have to do is to change your methods and not all of the code in your application. For an example of how to put a wrapper around the Entity Framework, see my blog post entitled “Simplify Calling Stored Procedures with the Entity Framework” at <https://www.pdsa.com/blog>.

## Follow Along with This Article

You may follow along with the steps in this article to build an MVC application using .NET 5.x if you wish. The simple Web application shows you how to interact with a product table in a SQL Server database. I'm going to use Visual Studio Code, MVC, .NET 5.x, C#, the Entity Framework, and the Adventure Works Lite sample database for SQL Server. You can get the Adventure Works Lite database on my GitHub at <https://github.com/PaulDSheriff/AdventureWorksLT>.

In the GitHub repository is a file named AdventureWorksLT.bak that you can use to restore the SQL Server database. If you're unable to restore the backup, there's a SalesLT-Product.sql file that you can use to create the product table with data in any SQL Server database.

## Create the MVC Core Project

Create an MVC application using the .NET CLI from within Visual Studio Code. Startup Visual Studio Code and open a terminal window by clicking the **Terminal > New Terminal** menu. Navigate to the folder where you normally place your development projects. For example, I'm going to create my project under the D:\Samples folder. Once I open a terminal window, I type in the command:

#### 

CD D:\Samples

Once you're in your development folder, create a new folder named EFSample using the MKDIR (or the MD) command.

#### 

MKDIR EFSample

Change to the new directory using the CD command in the terminal window, as shown in the command below.

#### 

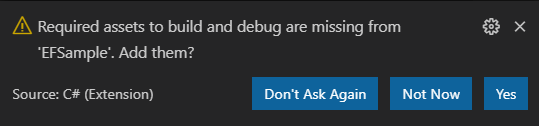
CD EFSample

Create a new MVC Core project in the EFSample folder by executing the **dotnet new mvc** command.

#### 

dotnet new mvc

Once the command has run, open the folder in VS Code by clicking the **File > Open Folder…** menu. A few seconds after opening the folder, you should be prompted with a dialog that looks like **Figure 1**. Click on the Yes button to add the required assets to the new project.

  
**Figure 1:** Add required assets to the project.

### Try It Out

To ensure that everything is created correctly, click the **Run > Start Debugging** menu and a Web page should be displayed in your default browser. If you get an error about your connection not being private, open the \Properties\launchSettings.json file and under the “EFSample” object, modify the applicationUrl property to look like the following: “http://localhost:5000”.

## Set Up EF Classes

To work with a table in a database, you need to add EF into your project and set up an entity class to map each field in your table to a property in a class. In addition, you need to create a DbContext class, add a connection string to point to your database table and configure some DbSet properties in your DbContext class. Finally, you need to create an instance of the DbContext class in the Startup class to allow this instance to be used with Dependency Injection (DI).

### Add Entity Framework

To use the Entity Framework in the .NET 5.x application, add a package to your EFSample project. Go back to the terminal window that should still be open in the EFSample folder. Type in the following command to add the Entity Framework to the project.

#### 

dotnet add package

Microsoft.EntityFrameworkCore.SqlServer

### Add a Product Entity Class

An entity class is a representation of a table or view contained in your database. In **Listing 1**, you can see an example of a Product class that has a property for each field in the SalesLT.Product table contained in the AdventureWorksLT sample database. You need to have a [Table] attribute to identity the table and schema in which this table is located. You may have additional data annotations such as the [Key] or [Required] attributes above various properties, however, I've left most of them off for this sample.

In your project, right mouse-click on the Models folder and add a new file named Product.cs. Into this file, add the code shown in **Listing 1**. In addition to the properties, add an override of the ToString() method. This method is useful when looking at lists of data in the debugger as you're going to do throughout this article.

#### Listing 1: Create a Product entity class to represent a single row of data in the Product table.

using System;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace EFSample.Models

{

[Table("Product", Schema = "SalesLT")]

public partial class Product

{

[Key]

public int ? ProductID { get; set; }

public string Name { get; set; }

public string ProductNumber { get; set; }

public string Color { get; set; }

public decimal StandardCost { get; set; }

public decimal ListPrice { get; set; }

public string Size { get; set; }

public decimal ? Weight { get; set; }

public DateTime SellStartDate { get; set; }

public DateTime ? SellEndDate { get; set; }

public DateTime ? DiscontinuedDate { get; set; }

public override string ToString()

{

return Name;

}

}

}

### Create DbContext Class

An essential piece of using EF is an instance of a DbContext class. Right mouse-click on the Models folder and add a new file named AdvDbContext.cs. Add the code shown in the code snippet below. You can think of a DbContext class as a database that contains properties of the type DbSet<T>. Each property relates to a single table and/or view in the database. For now, you only need a single DbSet<Product> property named Products.

#### 

using Microsoft.EntityFrameworkCore;

namespace EFSample.Models

{

public class AdvDbContext: DbContext

{

public AdvDbContext(DbContextOptions<AdvDbContext> options) : base(options) {}

public DbSet<Product> Products { get; set; }

}

}

### Add Connection String

The Entity Framework needs a connection string in order to connect to your database. Open the appSettings.json file and add a property named ConnectionStrings within the literal object in this file. This property's value is a literal object with a single property named DefaultConnection. The value for this property is the actual connection string. Change the values for Server, **Initial Catalog** and **Integrated Security** as appropriate for your database and server.

#### 

"ConnectionStrings": {

"DefaultConnection": "Server=Localhost;

Initial Catalog=AdventureWorksLT;

Integrated Security=True;"

}

### Add DbContext to the Startup Class

Instead of creating an instance of the AdvDbContext class each time you need to use it, it's best to allow .NET to use DI to pass in an instance of the DbContext to any class that needs it. Open the Startup.cs file and add two using statements at the top of this file in order to add the AdvDbContext class into the collection of services for this application.

#### 

using EFSample.Models;

using Microsoft.EntityFrameworkCore;

Within the Startup class, locate the ConfigureServices() method. Just after the one line that's already in this method, add an instance of the AdvDbContext to the services collection using the AddDbContext() method. To this method, you pass an options object where you tell it you're using SQL Server as your database, and you retrieve the connection string from the appSettings.json file using the Configuration class.

#### 

// Setup DB

Contextservices.AddDbContext<AdvDbContext>options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

## Call a Stored Procedure

You now have everything configured correctly in order to connect to your database and call any stored procedures within this database. In this first sample, build a stored procedure without any parameters and call that stored procedure to load a collection of product objects.

### Create a Stored Procedure

Go to your SQL Server database that contains the SalesLT.Product table and add a new stored procedure named Product\_GetAll. Place this stored procedure in the SalesLT schema. This stored procedure retrieves all rows of product data and returns them to your application.

#### 

CREATE PROCEDURE [SalesLT].[Product\_GetAll]ASBEGIN SELECT \* FROM SalesLT.Product;END

### Modify the Home Controller

To use the AdvDbContext class and the services of EF in your controller, add two using statements at the top of the HomeController.cs file just like you did in the Startup class.

#### 

using EFSample.Models;

using Microsoft.EntityFrameworkCore;

Tell .NET to inject the instance of the AdvDbContext object you created in the Startup class by adding a readonly field of the type AdvDbContext.

#### 

private readonly AdvDbContext \_db;

Next, add another parameter to the HomeController constructor. .NET looks at all parameters being passed into the HomeController and if they're in the collection of services, automatically fills in the parameters with an instance from the services collection object.

#### 

public HomeController(ILogger<HomeController> logger, AdvDbContext dbContext)

{

\_logger = logger;

\_db = dbContext;

}

Now that you have the instance of the AdvDbContext class, you simply need to call the appropriate method with the name of the stored procedure you just created. Add a new method named GetAll() to the HomeController class and make it look like the following code snippet.

#### 

public IActionResult GetAll()

{

List<Product> list;

string sql = "EXEC SalesLT.Product\_GetAll";

list = \_db.Products.FromSqlRaw<Product>(sql).ToList();

Debugger.Break();

return View("Index");

}

In the above code, you create a variable named list of the type List<Product>. Create a variable to hold the SQL statement with the name of the stored procedure to call on the database server. Use the FromSqlRaw<Product>(sql) method on the DbSet<Product> collection you defined in the AdvDbContext class to invoke the stored procedure. After the stored procedure returns the rows, this method maps each row to a Product object and builds the collection of products in the variable named list. I put the Debugger.Break() method after the call to the database so you can hover over the list variable and see all the data returned.

### Modify the Index Page

To call the GetAll() method you just created in the HomeController class, open the index.cshtml file located in the \Views\Home folder. Modify the code in this file to match the code shown below.

#### 

@ {

ViewData["Title"] = "Home Page";

}

<h1>Call Stored Procedures</h1>

<a asp-action="GetAll" class="btn btn-primary">Get All</a>

### Try It Out

After typing in all of the above code, try your changes by clicking on the Run > Start Debugging menu. When the home page loads, click the Get All hyperlink. The debugger should now come up and you can hover over the list variable to view the list of products returned. The ToString() method you added to the Product class is what displays each product's name when you look at the variable in the debugger.

## Pass Parameters to Stored Procedures

Many stored procedures have one or more parameters required for it to function correctly. EF allows you to pass in parameters to your stored procedures.

### Create a Stored Procedure with a Parameter

In the AdventureWorksLT database, add a new stored procedure named Product\_Get that accepts a single integer parameter named ProductID. This parameter is used to select a single product using the primary key field (ProductID) in the Product table.

#### 

CREATE PROCEDURE [SalesLT].[Product\_Get] @ProductID intAS

BEGIN

SELECT \* FROM SalesLT.Product

WHERE ProductID = @ProductID;

END

### Modify the Home Controller Class

If you've been using .NET and SQL Server for any length of time, you're familiar with the SqlClient namespace. This namespace contains various classes needed to interact with the SQL Server database using ADO.NET. EF is simply a wrapper around ADO.NET, so it makes sense that at some point, you're going to need to use some specific ADO.NET classes to use some of the services of EF. Open the HomeController.cs file and add a Using statement at the top of the file.

#### 

using Microsoft.Data.SqlClient;

Add new method (**Listing 2**) named GetAProduct() to this class in order to call the Product\_Get stored procedure. In this method, create a variable named parms of the type List<SqlParameter>. Create as many SQL parameter objects as you need for the stored procedure you're calling. In this simple sample, I only need to set the ParameterName and Value properties of the SqlParameter object. There are many other properties you can set such as Direction, IsNullable, DbType, etc., however, they're not needed for calling this stored procedure.

#### Listing 2: Use the SqlParameter class to pass parameters to your stored procedures.

public IActionResult GetAProduct()

{

List<Product> list;

string sql = "EXEC SalesLT.Product\_Get @ProductID";

List<SqlParameter> parms = new List<SqlParameter>

{

// Create parameter(s)

new SqlParameter { ParameterName = "@ProductID", Value = 706 }

};

list = \_db.Products.FromSqlRaw<Product>(sql, parms.ToArray()).ToList();

Debugger.Break();

return View("Index");

}

### Modify the Index Page

Open the index.cshtml file and add a new anchor tag immediately below the previous one you added that looks like the following code snippet.

#### 

<a asp-action="GetAProduct" class="btn btn-primary">Get A Product</a>

### Try It Out

Try out this new method by clicking the Run > Start Debugging menu. Click on the Get A Product hyperlink on the home page. The debugger should now come up and you can hover over the list variable to view the single product returned.

## Return a Scalar Value

Some stored procedures you write may not return a set of data, but rather a single value such as a string or integer value. This is called a scalar value and needs to be handled in a special way when using the Entity Framework. Create a stored procedure named Product\_CountAll in your database that looks like the following code.

#### 

CREATE PROCEDURE [SalesLT].[Product\_CountAll] AS

BEGIN

SELECT Count(\*) As [Value]

FROM SalesLT.Product

END

### Add Class to Get Integer Value

Notice in the Product\_CountAll stored procedure you just created the scalar value from the Count(\*) function is assigned to a variable named Value. Everything in EF needs to be assigned to a property in a class, so create a new class named *ScalarInt*. Right mouse-click on the Models folder and add a new file named ScalarInt.cs and add the following code into this new file. The property you create in this class must match the name of the variable you're using in your stored procedure for the scalar value you're returning.

#### 

namespace EFSample.Models

{

public partial class ScalarInt

{

public int Value { get; set; }

public override string ToString()

{

return Value.ToString();

}

}

}

### Modify the AdvDbContext Class

To use this ScalarInt class to call the Product\_CountAll stored procedure, create a property in the AdvDbContext class of the type DbSet<ScalarInt>. Open the AdvDbContext.cs file and add the following property.

#### 

public DbSet<ScalarInt> ScalarIntValue { get; set; }

When you use a DbSet<T> property, EF assumes that you're following good relational database practices such as the use of primary keys. However, the ScalarInt class doesn't correspond to a table and thus doesn't have a primary key. It's up to you to tell EF that there's no primary key by adding a line of code in the OnModelCreating() event. Override the following event in your AdvDbContext class and inform EF that that the ScalarInt class has no primary key.

#### 

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<ScalarInt>().HasNoKey();

}

### Modify the Home Controller

When calling a stored procedure that simply returns a single scalar value, you use the FromSqlRaw() method just like you've done before. However, you need to turn this single value into an enumerable list, so call the AsEnumerable() method after the call to the FromSqlRaw() method to create this list. Invoke the FirstOrDefault() method to return a single object. In this way, you get the single instance of the ScalarInt class you are looking for with the Value property in this instance filled in with the number of rows counted by this stored procedure.

#### 

public IActionResult CountAll()

{

ScalarInt value;

string sql = "EXEC SalesLT.Product\_CountAll";

value = \_db.ScalarIntValue.FromSqlRaw<ScalarInt>(sql).AsEnumerable().FirstOrDefault();

Debugger.Break();

return View("Index");

}

### Modify the Index Page

Open the index.cshtml file and add a new anchor tag immediately below the previous one you added that looks like the following code snippet.

#### 

<a asp-action="CountAll" class="btn btn-primary">Count All Products</a>

### Try It Out

Try out this new method by clicking the Run > Start Debugging menu. Click on the Count All Products hyperlink on the home page. The debugger should now come up and you can hover over the value variable to view the total count of product records.

## Modify Data Using a Stored Procedure

So far in this article, you've learned to retrieve data from stored procedures both with and without parameters. You also learned to retrieve a single scalar value from a stored procedure. To call a data modification stored procedure such as one that performs an INSERT, UPDATE, or DELETE, you use a different method on a different object in the DbContext object. To illustrate an UPDATE, create a stored procedure named Product\_UpdateListPrice.

#### 

CREATE PROCEDURE [SalesLT] .[Product\_UpdateListPrice] @ProductID int, @ListPrice money AS

BEGIN

UPDATE SalesLT.Product

SET ListPrice = @ListPrice

WHERE ProductID = @ProductID;

END

### Modify the Home Controller

When calling any data modification stored procedure use the ExecuteSqlRaw() method on the Database property of your DbContext object. This method returns an integer value of the number of rows affected by the statement in that stored procedure. In your HomeController class, add a new method named UpdateListPrice() to call the Product\_UpdateListPrice stored procedure you just created, as shown in **Listing 3**.

#### Listing 3: Use the ExecuteSqlRaw() method on the Database object to call a data modification stored procedure.

public IActionResult UpdateListPrice()

{

int rowsAffected;

string sql = "EXEC SalesLT.Product\_UpdateListPrice @ProductID, @ListPrice";

List<SqlParameter> parms = new List<SqlParameter>

{

// Create parameters

new SqlParameter { ParameterName = "@ProductID", Value = 706 },

new SqlParameter { ParameterName = "@ListPrice", Value = 1500 }

};

rowsAffected = \_db.Database.ExecuteSqlRaw(sql, parms.ToArray());

Debugger.Break();

return View("Index");

}

### Modify the Index Page

Open the index.cshtml file and add a new anchor tag immediately below the previous one you added that looks like the following code snippet.

#### 

<a asp-action="UpdateListPrice" class="btn btn-primary">Update List Price</a>

### Try It Out

Try out this new method by clicking the Run > Start Debugging menu. Click on the Update List Price hyperlink on the home page. The debugger should now come up and you can hover over the rowsAffected variable to view the total number of records affected. If you've done everything correctly, the value should be one (1). If you look in the Product table, you should also see the ListPrice field for the Product 706 has been modified to the value 1500.

## Handle Multiple Result Sets

A common performance-improving technique is to return multiple result sets from a single stored procedure. Doing this requires less client- and server-side resources than submitting multiple calls to the database and returning multiple result sets individually. To illustrate how to retrieve multiple result sets using EF, create the following stored procedure in the Adventure Works 2019 database.

#### 

CREATE PROCEDURE [Production].[MultipleResultsColors] AS

BEGIN

SELECT \* FROM Production.Product WHERE Color = 'Black';

SELECT \* FROM Production.Product WHERE Color = 'Red';

END

### Modify the Home Controller

When working with multiple result sets, you are going to get close to pure ADO.NET code rather than EF code. As such, you need to open the HomeController.cs file and add two using statements that once again you may be familiar with as they are used frequently with ADO.NET.

#### 

using System.Data;

using System.Data.Common;

Next, add a new method named MultipleResultSets() as shown in **Listing 4**. In this method, you create two different List<Product> variables: one to hold all black products and one to hold all red products. Using the Database object on your DbContext object, you call the GetDbConnection() method to retrieve the ADO.NET database object. From this object, you can create a command object using the CreateCommand() method. Fill in the CommandText property of the command object with the SQL statement you created with the call to the stored procedure. Open the connection on the database object and you're now ready to call the stored procedure.

#### Listing 4: When retrieving multiple result sets, you are almost writing pure ADO.NET code.

public IActionResult MultipleResultSets()

{

List<Product> black = new List<Product>();

List<Product> red = new List<Product>();

DbCommand cmd;

DbDataReader rdr;

string sql = "EXEC Production.MultipleResultsColors";

// Build command object

cmd = \_db.Database.GetDbConnection().CreateCommand();

cmd.CommandText = sql;

// Open database connection

\_db.Database.OpenConnection();

// Create a DataReader

rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

// Build collection of Black products

while (rdr.Read())

{

black.Add(new Product

{

ProductID = rdr.GetInt32(0),

Name = rdr.GetString(1),

ProductNumber = rdr.GetString(2)

});

}

// Advance to the next result set

rdr.NextResult();

// Build collection of Red products

while (rdr.Read())

{

red.Add(new Product

{

ProductID = rdr.GetInt32(0),

Name = rdr.GetString(1),

ProductNumber = rdr.GetString(2)

});

}

Debugger.Break();

// Close Reader and Database Connection

rdr.Close();

return View("Index");

}

Call the stored procedure using the ExecuteReader() method of the command object. Pass in an enumeration to tell ADO.NET to close the connection when the reader is closed. Once you have the data reader object, call the Read() method to loop through each record in the first result set. This method returns a false when there are no more records in the first result set. Each time through the loop, create a new Product object with any properties you want, and add them to the black list of products.

When you're done with the first result set, call the NextResult() method on the data reader object to advance to the next result set. Loop through all these records until the Read() method returns a false value. Each time through the loop, create a new Product object and add any properties you want. Add each new Product object to the red list of products. When you are done with all the results, be sure to call the Close() method on the reader object to close it and the database connection. Note that I didn't use any exception handling in this method, but you definitely should!

*Advertisement*

### Modify the Index Page

Open the index.cshtml file and add a new anchor tag immediately below the previous one you added that looks like the following code snippet.

#### 

<a asp-action="MultipleResultSets" class="btn btn-primary"> Multiple Result Sets</a>

### Try It Out

Try out this new method by clicking the Run > Start Debugging menu. Click on the Multiple Result Sets hyperlink on the home page. The debugger should now come up and you can hover over the black and the red variables to view the different lists of products.

## Summary

In this article, you learned how to make calls to stored procedures from the Entity Framework in .NET 6.x. There are some differences from previous versions of the Entity Framework, but nothing too dramatic that you can't figure out readily enough. Calling stored procedures with and without parameters can be a little different. Retrieving data and retrieving scalar values is done differently, so it's good to keep these samples around so you can look them up when you need to. In some cases, such as multiple result sets, you use almost pure ADO.NET code.

# Getting the Sample Code

You can download the sample code for this article by visiting [www.CODEMag.com](http://www.codemag.com/) under the issue and article, or by visiting [www.pdsa.com/downloads](http://www.pdsa.com/downloads). Select “Fairway/PDSA Articles” from the Category drop-down. Then select “Calling Stored Procedures with the Entity Framework in .NET 5” from the Item drop-down.