**STUDENT QUESTION**

Please guide me through SQL Connectivity in ASP.NET CORE projects.

**JPC RESPONSE**

This is a big question. I will start by rewording this as:  
"Working with SQL and databases in CORE projects".  
CORE has an "Entity Framework" object for this.   
This is "instantiated" as an object in "Startup.cs" with code which will usually be "generated" for us.  
We are finding that this sometimes does not happen and we need to write this in.  
Check this if you see an error message like "Service not registered".  
The following example is for a database named "Northwind" which is a common learning example.

services.AddDbContext<Models.DB.**NorthwindContext**>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

Now we turn our attention to the Controller.

The Controller code is usually "generated" for us. A new keyword for that is "scaffolding".  
It is a good idea to check it if we get error messages like "Service not registered".

We need a "using" statement like this at the top of the controller code:

using MVCManukauTech.Models.DB;

The "services" object created in "Startup.cs" gets "injected" via a constructor parameter.  
This example code is for Controller "ProductsController" that works with the "Products" table in the "Northwind" database.

public class ProductsController : Controller

{

private readonly NorthwindContext \_context;

public ProductsController(NorthwindContext context)

{

\_context = context;

}

//other methods go here

}

The creators of this framework like the keyword "context" for the way we setup and use a database.  
Also, it is a common C# language convention that class private fields have variable names starting with an underscore. For example "\_context" that we see here.  
We will see that other code in this class ProductsController will use methods of "\_context" for database operations.

If we are writing a custom controller from empty, we can copy and adapt from generated code in another controller.

**To READ data, we use the method "FromSql"**  
We need to use this as a method of a C# class that reflects the data that we are working with.  
The simplest of these C# classes are "Models". These "model" or "imitate" or "represent" tables in the database.  
We can generate them automatically with the "Entity Framework".

**Examples - simplified to literal SQL without placeholders:**

var products = \_context.Products.FromSql("SELECT \* FROM Products").ToList();

var products = \_context.Products.FromSql  
 ("SELECT \* FROM Products WHERE CategoryId = 'Travel'").ToList();

For one product using a list with one member.  
var products   
 = \_context.Products.FromSql("SELECT \* FROM Products WHERE ProductId = '1MOR4ME'").ToList();  
Product product = products[0];

Shorter version of "For one product" uses method SingleOrDefault()  
var product  
 = \_context.Products.FromSql  
 ("SELECT \* FROM Products WHERE ProductId = '1MOR4ME'").SingleOrDefault();

**ViewModel**

We also need to create custom C# classes for special cases like a JOIN between 2 database tables.  
These are called "ViewModels"

In our XSpy example which you will see later, we have the example "OrderDetailsQueryForCart.cs" which has properties from 2 tables: "Products" and "OrderDetails".

We need to use a "Model" or "ViewModel" class in a query so that data from the database gets copied into memory with formatting in memory similar to the database design. The class fields need to match the database query fields.

**Placeholders as parameters.**

We often have SQL with placeholders - user input goes into these places at runtime.  
We use parameter syntax with placeholders like @p0, @p1, @p2, @p3 ...  
We then substitute the runtime values with the FromSql method or the ExecuteSqlCommand method.  
Examples from "OrderDetailsController.cs" line 161

SQLUnitCostLookup = "SELECT \* FROM Products WHERE ProductId = @p0";

var products = \_context.Products.FromSql(SQLUnitCostLookup, ProductId).ToList();

decimal unitCost = Convert.ToDecimal(products[0].UnitCost);

**To WRITE data or carry out other non-Query work, the "low level" method is "ExecuteSqlCommand"**  
which is a child of "\_context.Database". Therefore this appears in code as **\_context.Database.ExecuteSqlCommand**

lineNumber = cart.Count + 1;

//150807 JPC Security improvement implementation of @p0 etc

SQLBuy = "INSERT INTO OrderDetails VALUES(@p0, @p1, @p2, @p3, @p4)";

rowsChanged  
 = \_context.Database.ExecuteSqlCommand(SQLBuy, orderId, lineNumber, ProductId, 1, unitCost);

Another ExecuteSqlCommand example from Xspy, OrderDetailsController.cs, line 215

SQLUpdateOrderDetails   
 = "UPDATE OrderDetails SET Quantity = @p0 WHERE OrderId = @p1 AND LineNumber = @p2";

rowsChanged  
 = \_context.Database.ExecuteSqlCommand(SQLUpdateOrderDetails, Quantity, orderId, LineNumber);

-----------------

**LINQ**

Microsoft have an alternative to SQL which we often see auto-generated by Visual Studio.

**LINQ EXAMPLE ONE - query from only one table:**

SQL version

var products = \_context.Products.FromSql("SELECT \* FROM Products");  
return View(products.ToList());

LINQ version

var products = \_context.Products;  
return View(products.ToList());

------------------------------------------------

**LINQ EXAMPLE TWO - find 1 product record by its primary key ProductId, example "1MOR4ME"**

SQL version

var products = \_context.Products.FromSql("SELECT \* FROM Products WHERE ProductId = '1MOR4ME'");  
Product product = products.ToList()[0];  
return View(product);

LINQ version

var product = \_context.Products.Find("1MOR4ME");  
return View(product);

------------------------------------------------

**LINQ EXAMPLE THREE - a query across 3 tables**

SQL version

string sql =

"SELECT ProductId, ProductName, Products.CategoryId AS CategoryId, "

+ "Products.SupplierId AS SupplierId, QuantityPerUnit, "

+ "UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued, "

+ "Categories.CategoryName AS Category, Suppliers.CompanyName AS Supplier "

+ "FROM (Products INNER JOIN Categories ON Products.CategoryId = Categories.CategoryId) "

+ "INNER JOIN Suppliers ON Products.SupplierId = Suppliers.SupplierId";

var products = \_context.Products.FromSql(sql);

return View(products.ToList());

LINQ version

var products = \_context.Products.Include(p => p.Category).Include(p => p.Supplier);  
return View(products.ToList());

I will stay with the Visual-Studio-generated LINQ if it is working well.  
When writing new queries I like to stay with standard SQL. In my experience SQL gives better error messages although the LINQ error messages may now be improving.   
LINQ is popular among some industry programmers so it is a good idea to try some LINQ.

----------------------------------------------------------------------------------------------------------------------------

**WRITING DATA**

Example adding a new record eg a new product - instead of SQL INSERT

\_context.Products.Add(product);

\_context.SaveChanges();

Example of LINQ for editing an existing record eg changing an existing product - instead of SQL UPDATE

\_context.Entry(product).State = EntityState.Modified;

\_context.SaveChanges();

More details at:

<https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/advanced>