Creating an ASP.NET MVC Project

Step by Step

Rick Bellows
August 10, 2023

Initial Information:

The name of the Web Application MVC 7, will be named:

`C:\Users\rgbel\source\repos\`**`BrokerTrac_050.`**`sln`

It will connect to a database that I have installed to a named instance on my machine:

Server\Instance: **FRODO\THE_SHIRE**
Database: **BrokerTrac050**

Notice that I've named the Project and the Database slightly differently. The database omits the underscore. I do this so that in the code, I can tell the namespace references (BrokerTrac_050) from database references (BrokerTrac050). This is just a matter of personal preference. In fact, don't even read much into the name "BrokerTrac" – this is just what I decided to call this. Apparently there is a company of that name. I wouldn't name it that if this were going to be a commercial system. I don't want to get a "cease and desist" letter.

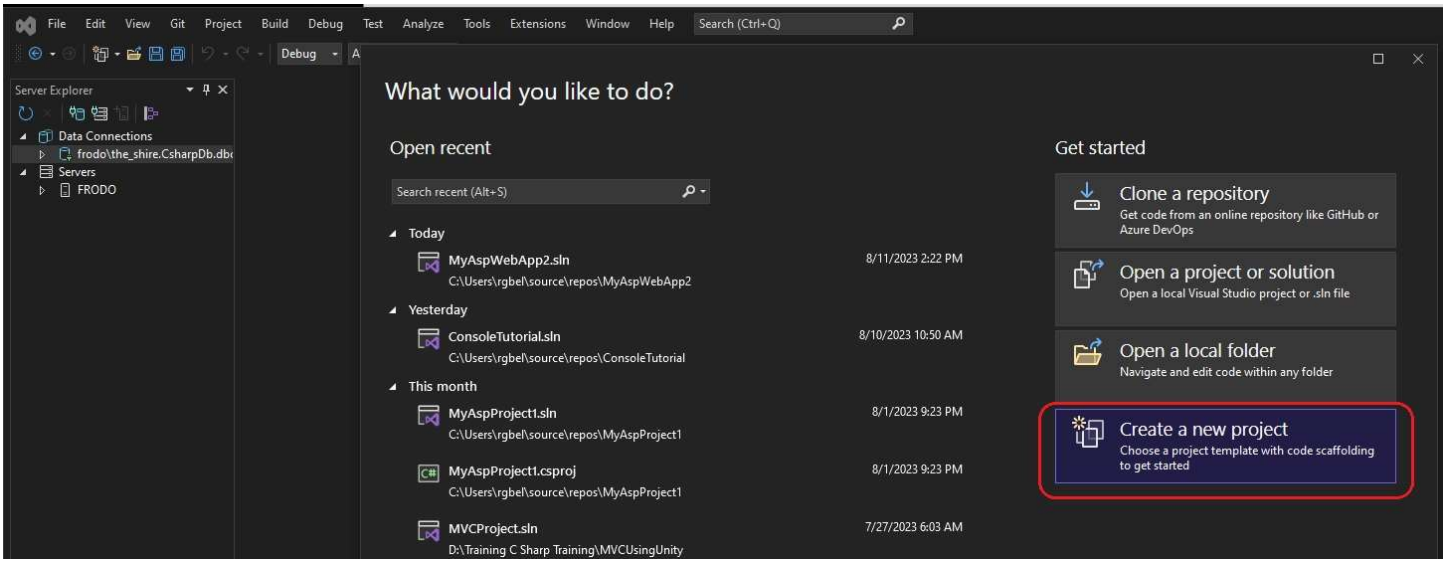
The database will have several tables already created with a few records in it.
See **Appendix A: The initial BrokerTrac050 Database**

Shortcut:
The first several steps (1-4) can be done in DOS: Open a CMD shell and:

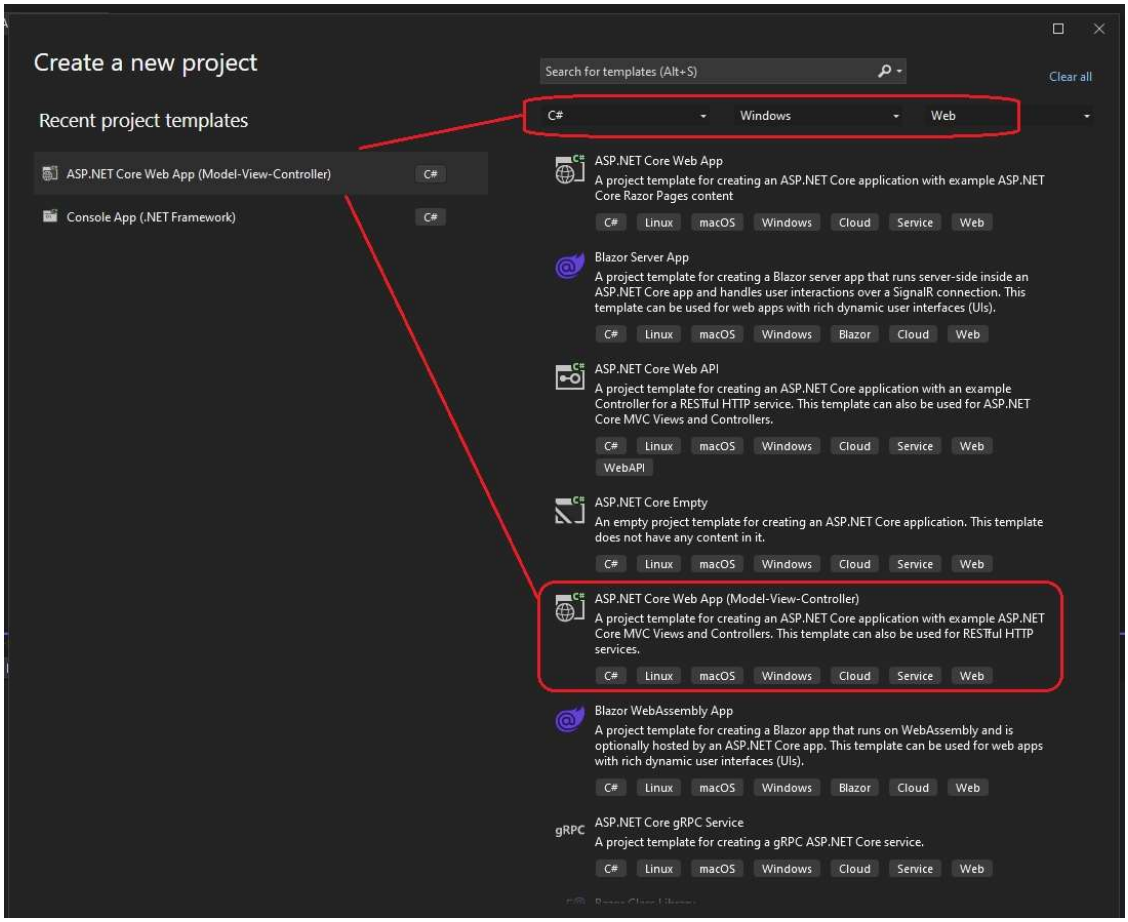
```
cd C:\Users\rgbel\source\repos\
mkdir BrokerTrac_050
cd BrokerTrac_050
dotnet add mvc

■ Then if you're in the project directory, you can
dotnet run --lanuch-profile http        or:
dotnet run BrokerTrac_050

dotnet watch             Ctrl-R  to restart
```

Begin at the beginning.

1) Open Visual Studio Create New Project C#, Windows, Web

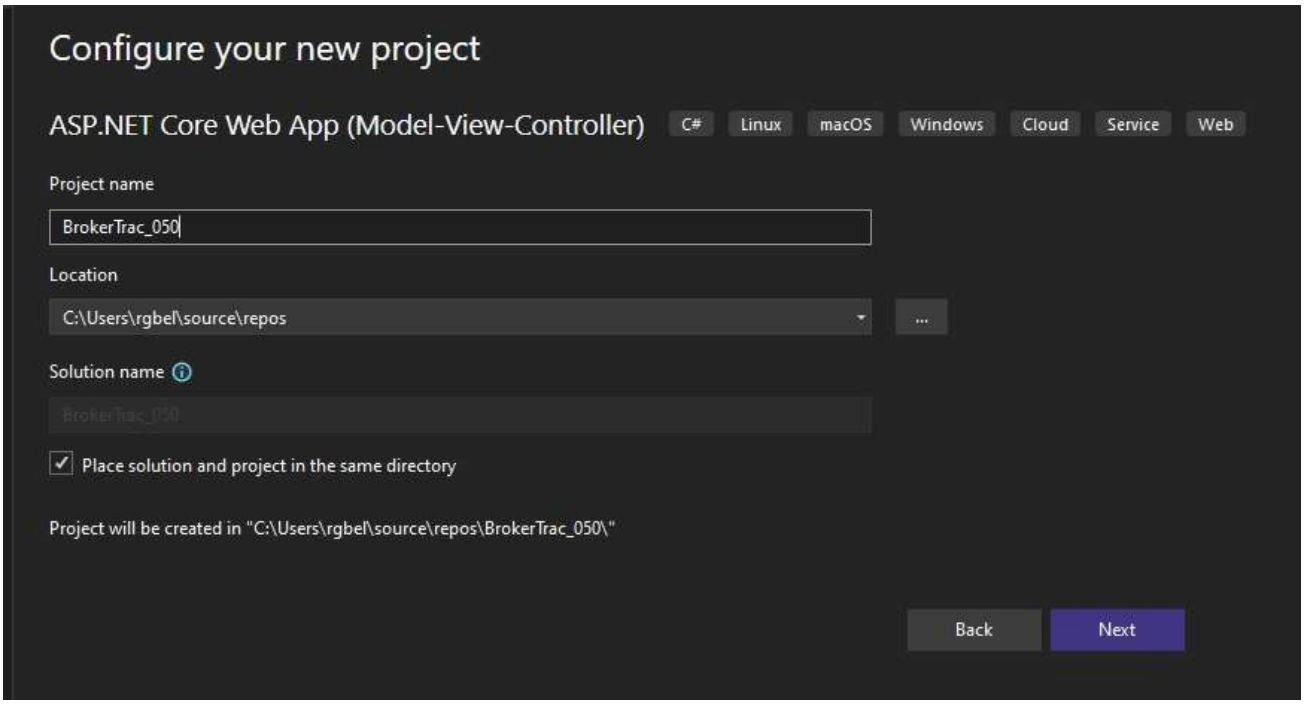


2) Choose template for: ASP.NET Core Web App (Model-View-Controller)



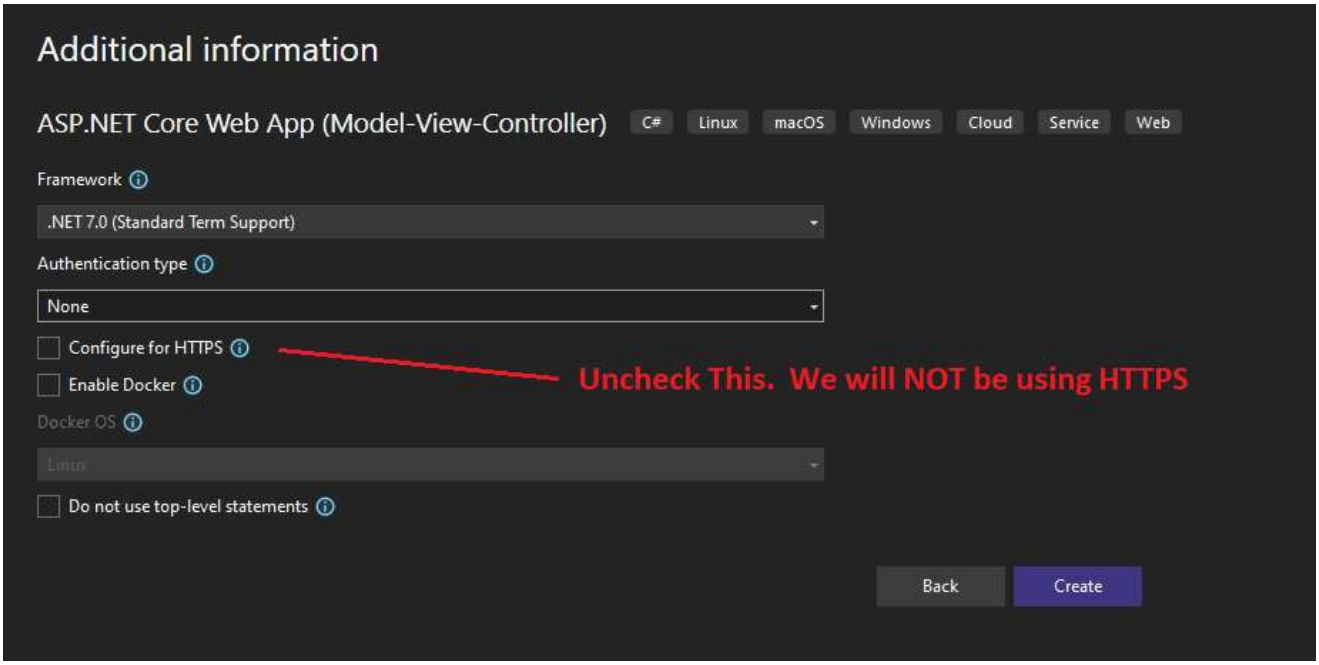
3) Name the project: **BrokerTrac_050**

Place solution and project in the same directory.

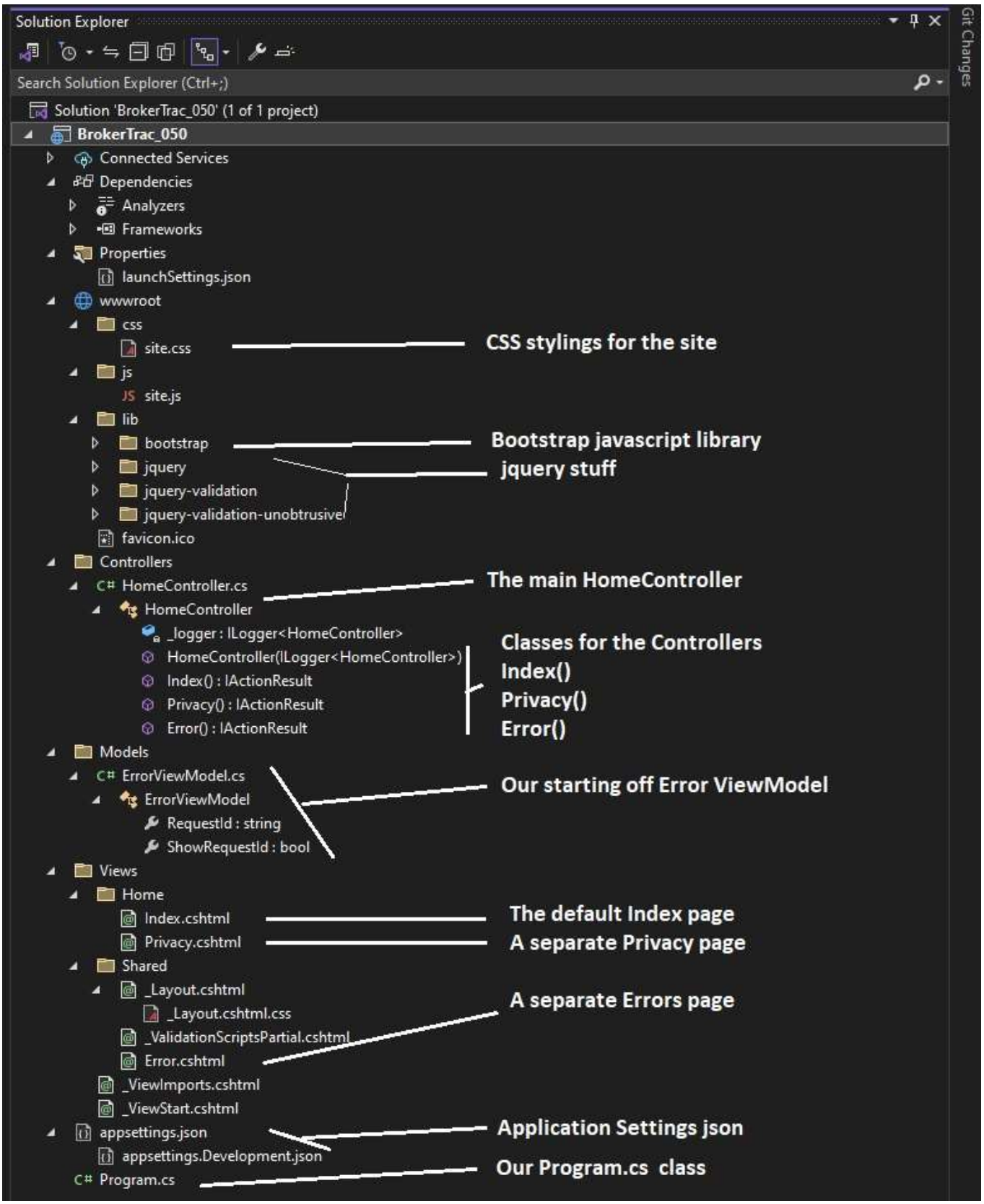


4 - Additional Information

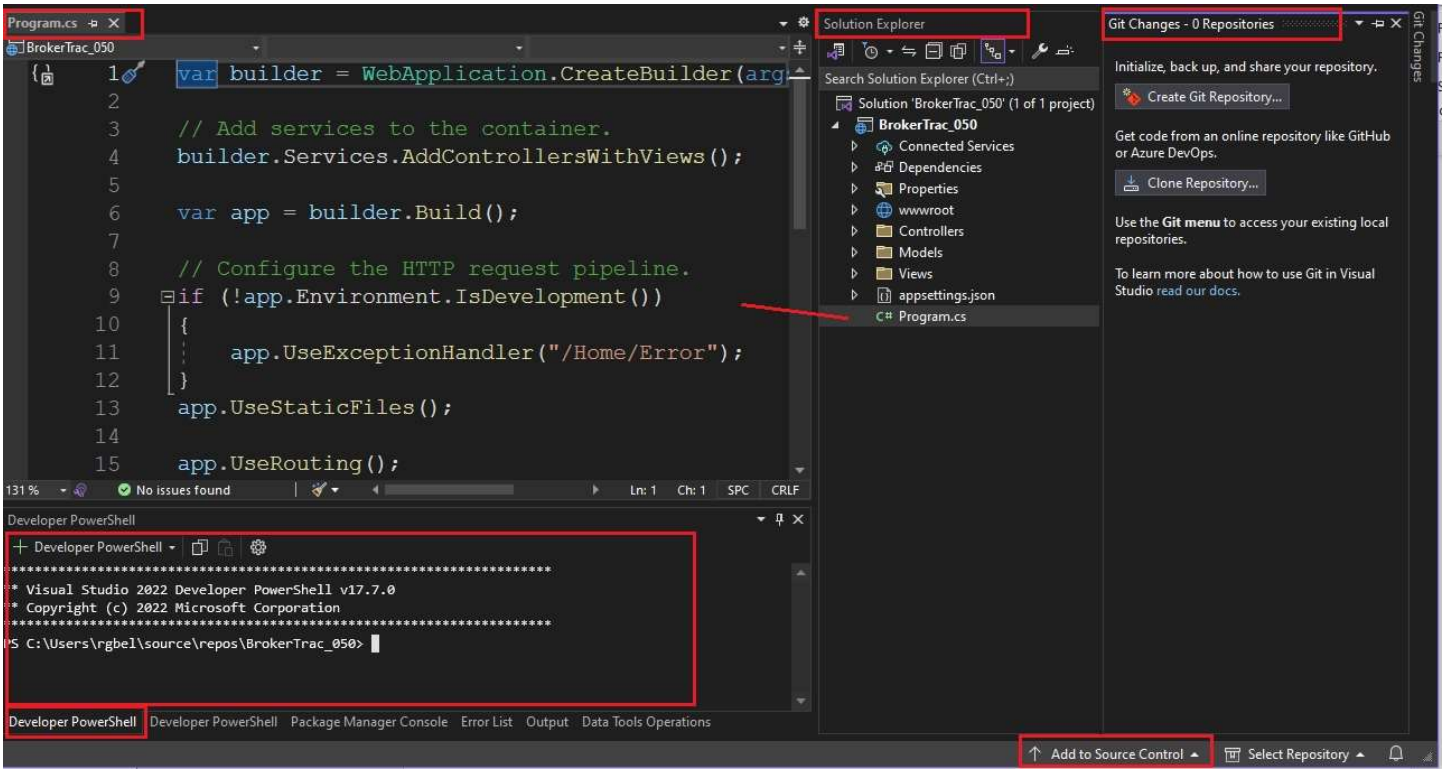
Framework: .NET 7.0 (Standard Term Support). Uncheck "Configure for HTTPS"
The reason I don't want to use https, is that the webserver will require a certificate to validate against. Although you can create a local certificate, I don't want to get into that at all.



5) Initial Solution Explorer



6) PowerShell, the Package Manager, and the dot net Command Line Interface (CLI)



There are two different "command line" interfaces we can use in Visual Studio when issuing commands to do things like install packages and a variety of other things. The lower left shows the **PowerShell** terminal. We will be using this **Developer PowerShell** (which I prefer), and has a prompt of "`PS>`" :

For me it is: **`PS`** `C:\Users\rgbel\source\repos\BrokerTrac_050>`

The second interface is the Package Manager Console, aka The Dot Net **Package Manager** It is a matter of preference as to which one you use, since you'll be issuing: **dotnet** commands to either one, however, the syntax is slightly different. In PowerShell, the "switches" switches are the same but the Options have slightly different spellings and capitalization.

The Package Manager CLI has a prompt of "PM>", But actually, without showing the path by default: So it will look like : `PM>`

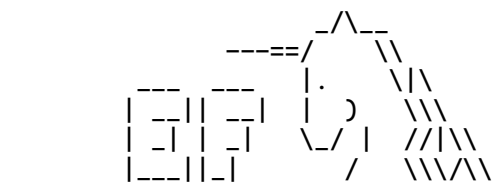
The Package Manager doesn't *show* what folder you are in by default. You ARE in a directory – type dir to confirm this.

I am doing Terminal using the **PowerShell** interface. No particular reason why. But be careful as some of the commands I show may not run with the spellings of the switches I'm using.

And there is another interface, the Entity Framework CLI. This is actually invoked from one of the two above (PowerShell or the Package Manager). It is done by issuing commands like **dotnet ef** (with parameters)

You can issue this from either the PowerShell or the Package Manager:

```
PM> dotnet ef
```



```
Entity Framework Core .NET Command-line Tools 7.0.10

Usage: dotnet ef [options] [command]

Options:
  --version        Show version information
  -h|--help        Show help information
  -v|--verbose     Show verbose output.
  --no-color       Don't colorize output.
  --prefix-output  Prefix output with level.

Commands:
  database    Commands to manage the database.
  dbcontext   Commands to manage DbContext types.
  migrations  Commands to manage migrations.

Use "dotnet ef [command] --help" for more information about a command.
PM>
```

This is the dot net entity framework Command Line Interface (CLI)

This is where we handle migrations mostly. More on migrations later.

[EF Core tools reference (.NET CLI) - EF Core | Microsoft Learn]

We won't need "dotnet ef" just yet. We're still setting up the project with the tools we need.

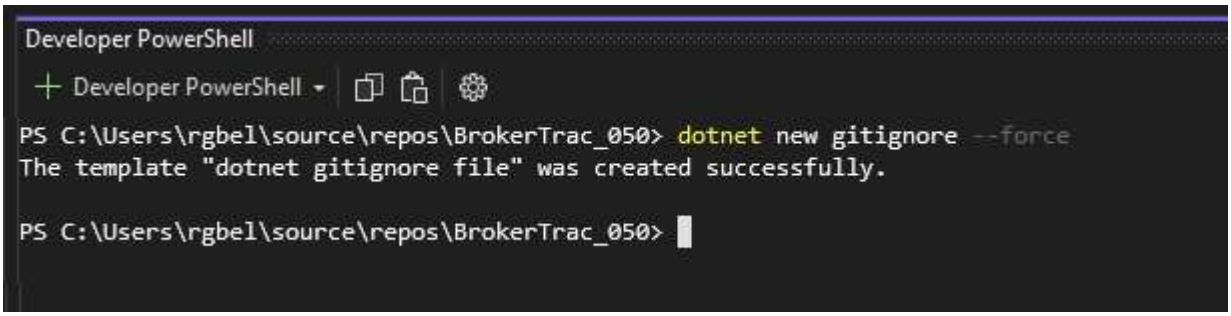
Source Control:

To the right, next to the Solution Explorer, is the **Git Changes** area. We will be setting up Source Control to GitHub next. The word "repository" generally means any source-control ssytem.

The gitignore file

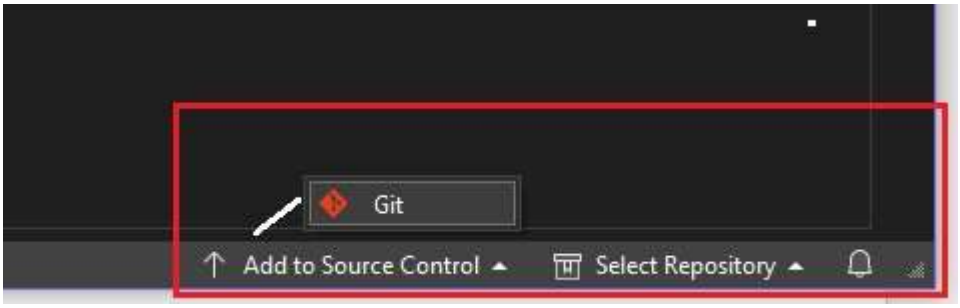
Certain of our files in the project are configuration related, and not really source code, so we will later create a "gitignore" file – and each time we push to the repository, it won't store that "environment-specific" stuff. Later, if we ever release this application, that kind of information will just get in the way.
In the PowerShell terminal, create a Gitignore file like this:

```
PS> dotnet new gitignore --force
```



7) Add this project to GitHub

Click on the "Add to Source Control" dropdown in the lower right section of Visual Studio.



I already have **Git** installed on this computer, and I am logged into my remote **GitHub** account.
These are two separate, but very related things.

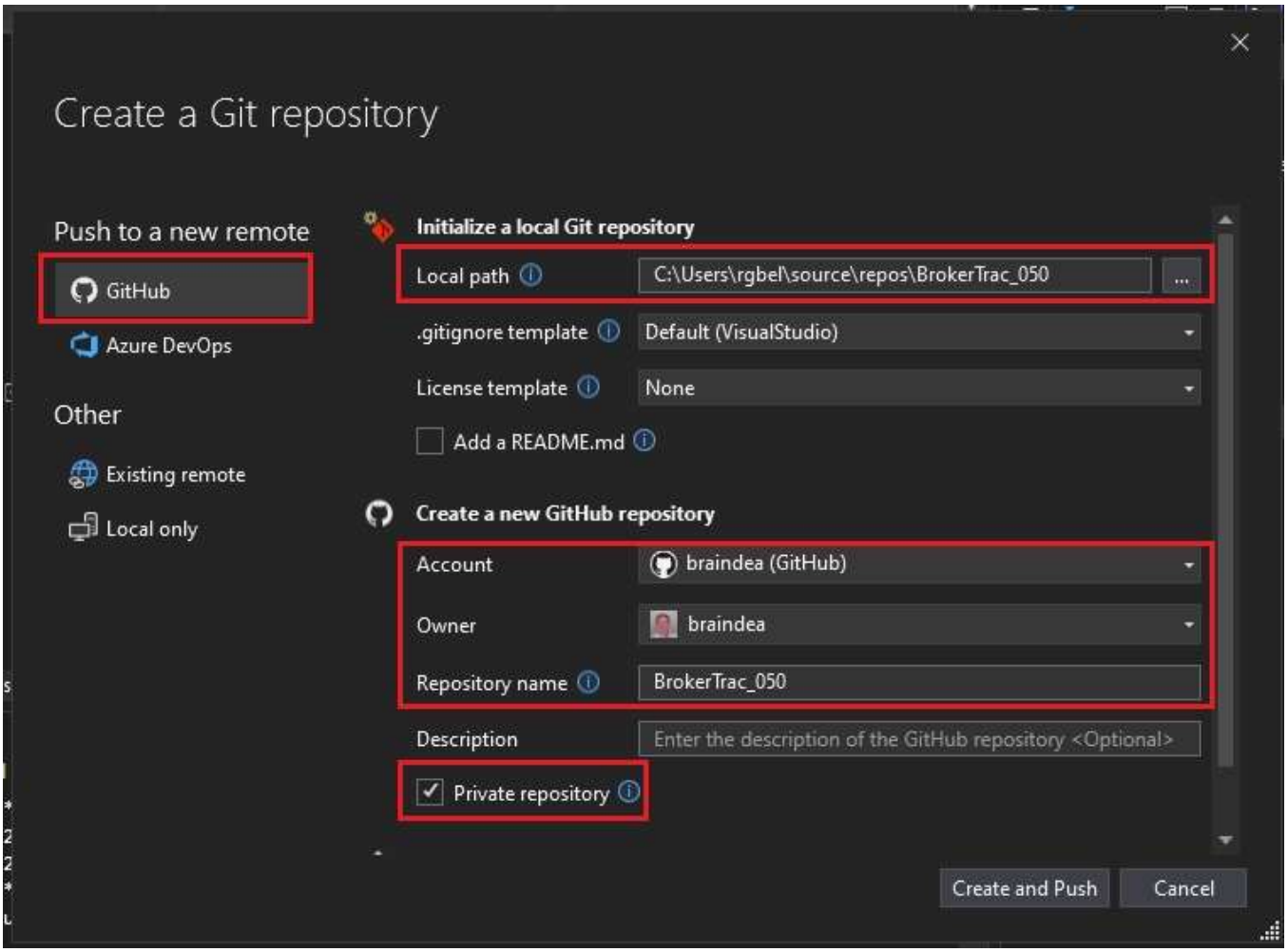
Git is a local repository (or "repo"), and GitHub is a remote, online repository. You can commit, undo, create branches, etc. locally in Git. But you can also create Pull Requests (which we won't be doing) or Pushing to GitHub. Since we are the only person (and on this one computer) using this repository – we won't need Pull Requests. We'll just periodically Push our changes to GitHub.

Clicking on "Git" here, brings up our dialog for the local Git and online GitHub configurations:

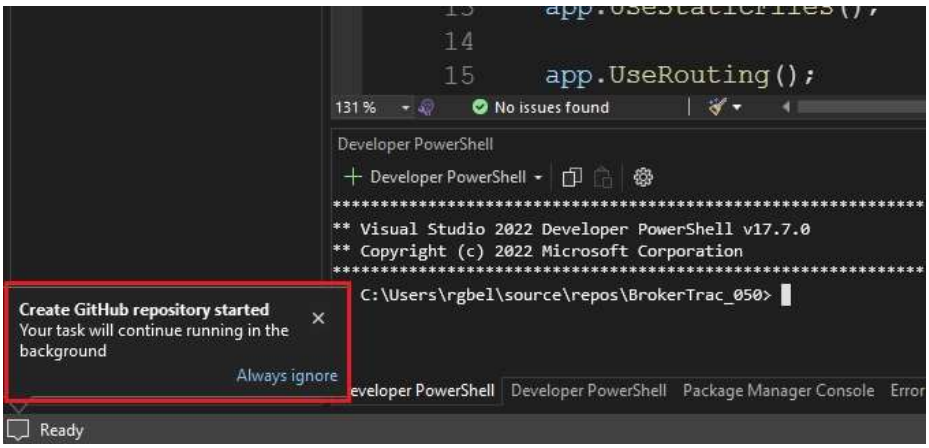
8) Configure Source Control Repositories

The defaults will already be populated correctly. We'll be using GitHub, not Azure DevOps.
You'll notice that Visual Studio, by default, stores projects off of the **C:\Users**\login**source\repos**\ folder.

Press button for: Create and Push



In the lower left of Visual studio, you should see:

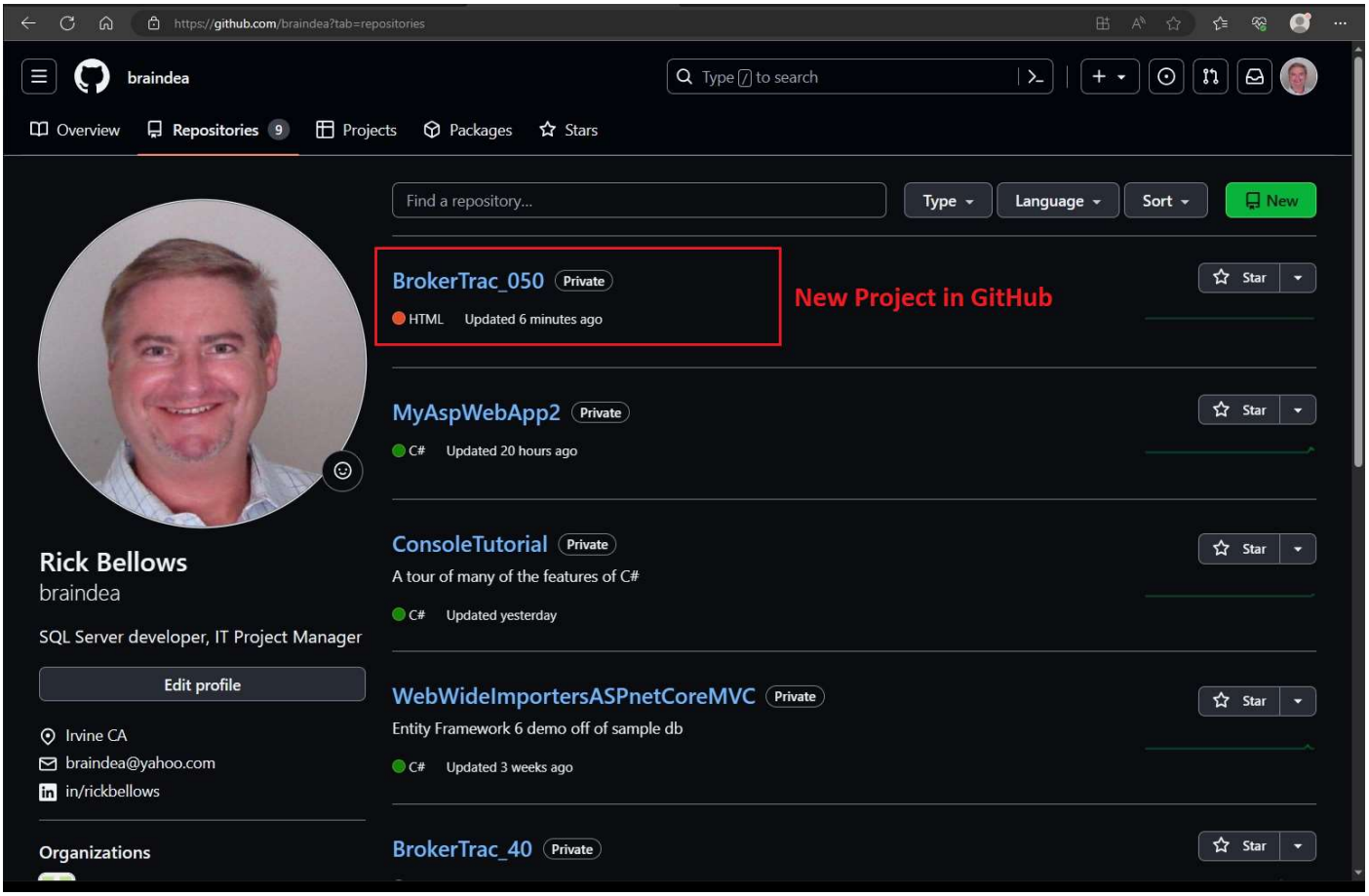


9) Verify a new Repository has been added to my GitHub account

This is a private repository. Nobody except me can see it.

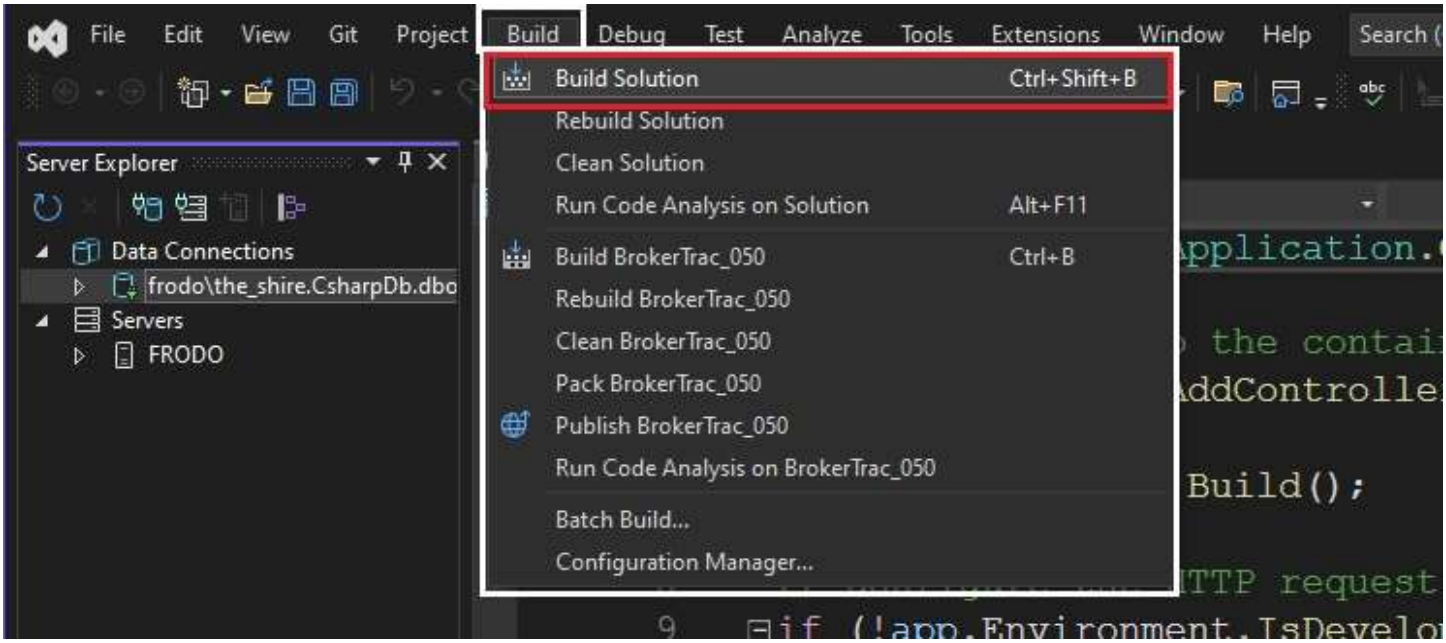
If it were shared with someone, or made Public, users would create a "Pull Request" to check-out the code so they could work on it, and then later commit, merge, and push changes into the branch.

Since this is private, I'll be the only one making changes, so Pull Requests are not really necessary – unless, say, I were going to do some work on a separate machine, in which case Pull Requests would be used to help coordinate changes between two development environments. Right now though, a local commit, and a remote push are all mostly what we'll be doing to synch changes to GitHub.

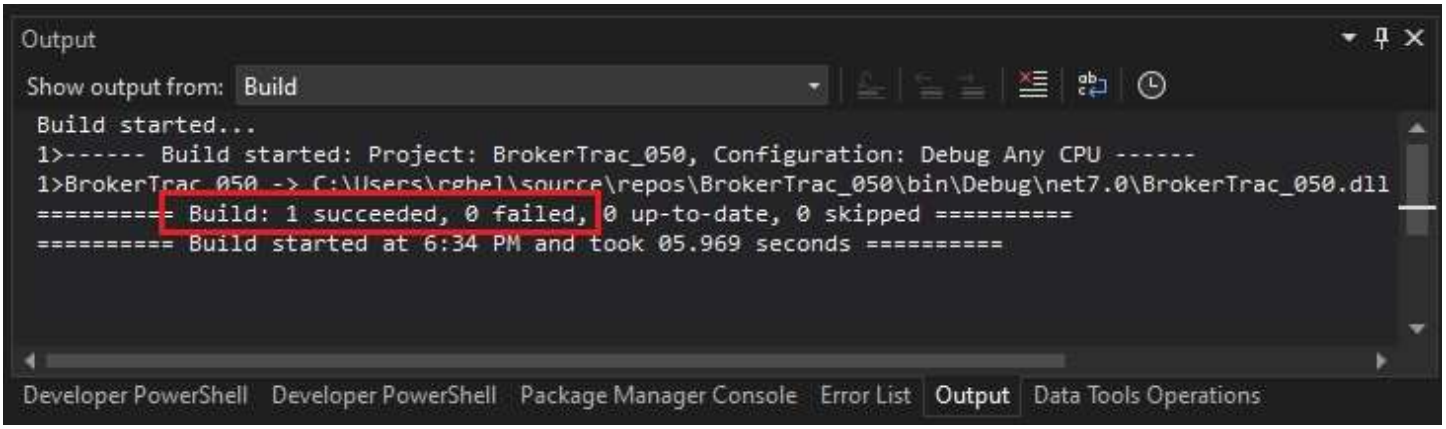


10) Build the Solution.

In Visual Studio, at the menu bar: Build – Build Solution or: CTRL-Shift-B



Note the **Output** window. Build: 1 succeeded, 0 failed

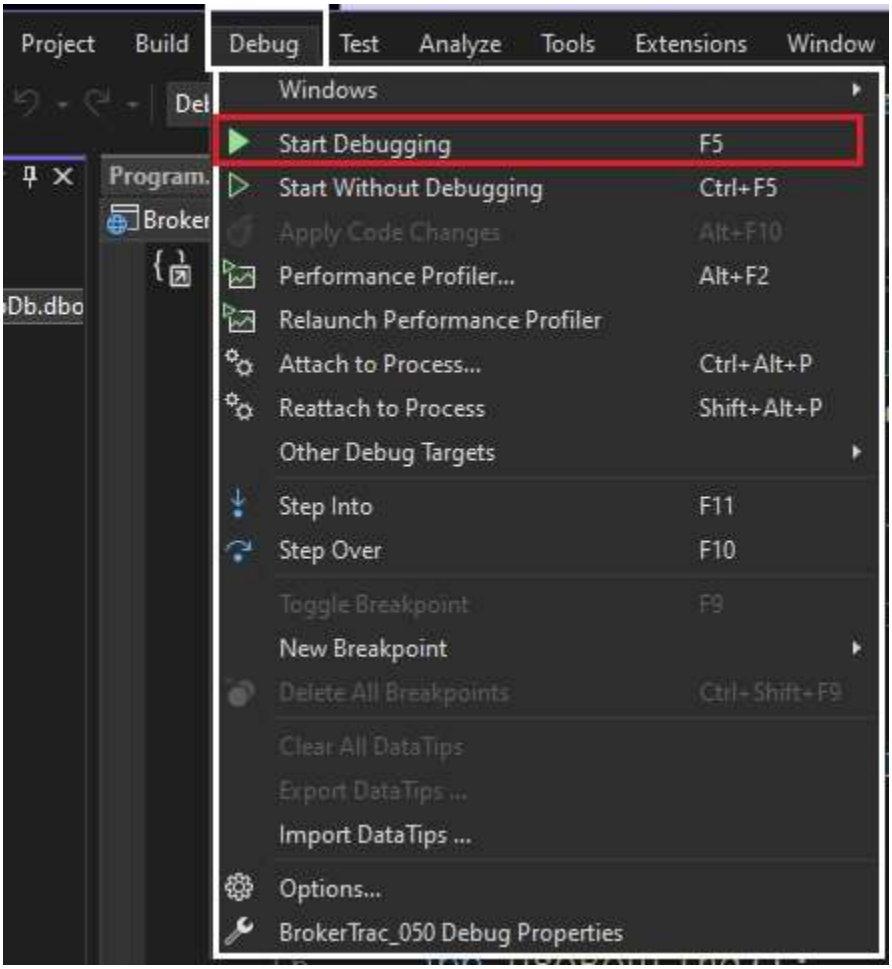


11) Run the Solution

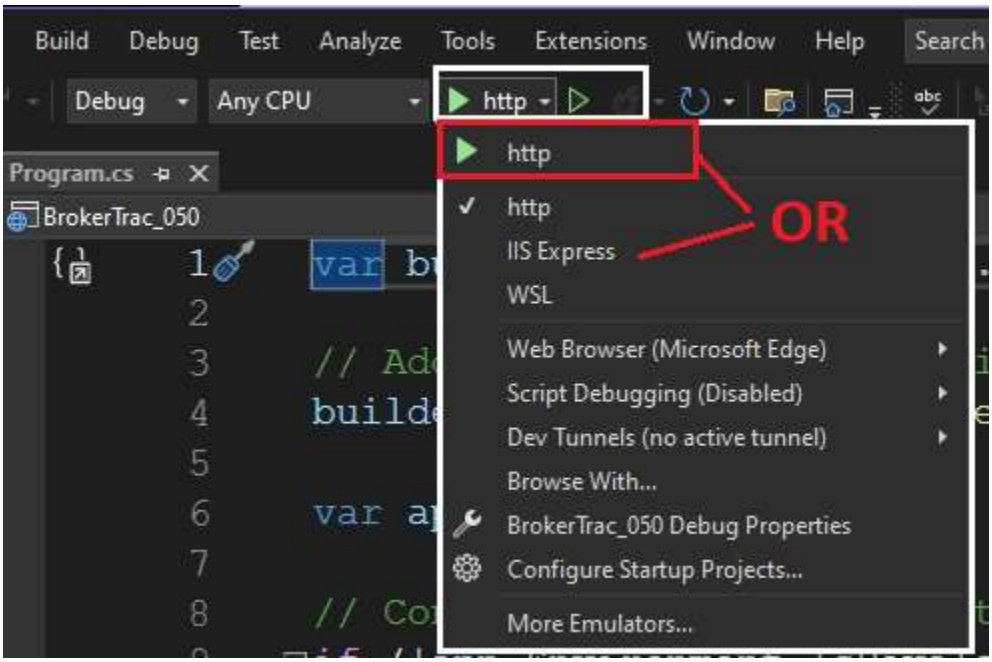
Note that ASP.NET projects don't behave the same as Console Applications. After running the first time – you either have to close them a certain way – to stop the running webserver processes, or do a "Hot Reload"

We will see this in a moment. To Run in Debug mode, you have a couple of options:

Using the first option: Debug or: F5



You can also Run using the second option:

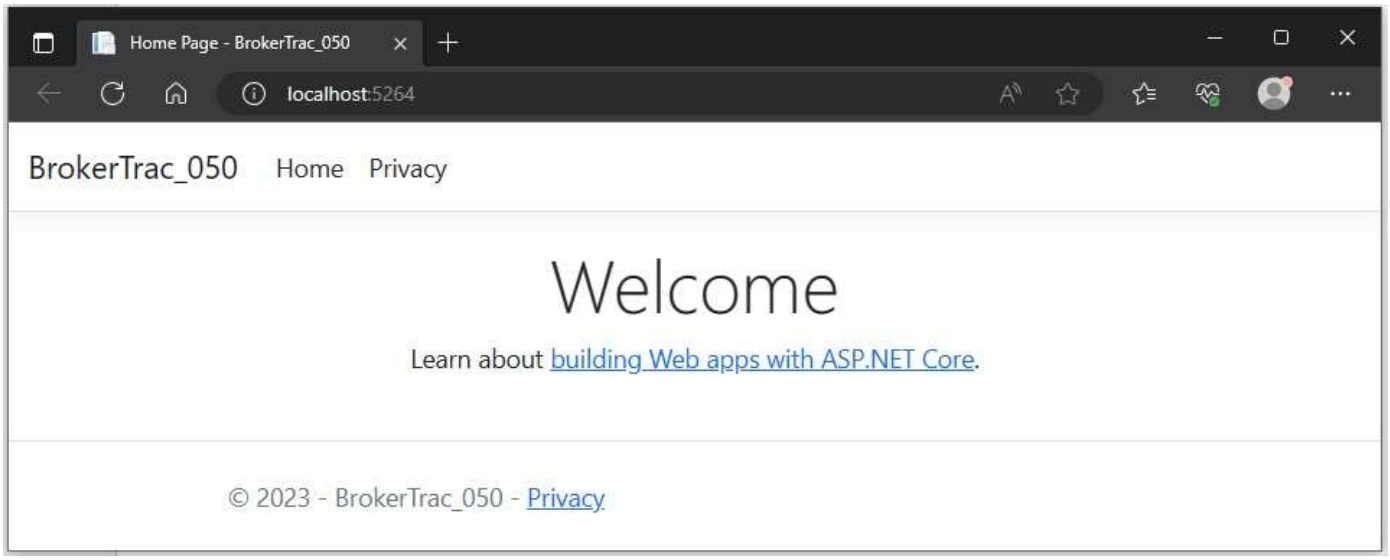


If we had set up the project to support HTTPS, you would have seen an https option here.

With https, you might run into an error saying that you don't have a trusted web certificate installed.

We're avoiding that altogether in this walk through.

What you should now get, is our first run of the application, as it was built by Visual Studio using the template project. It's just a welcome page, with two options at the top, one for a "Home" page (which is what's shown), and a "Privacy" policy page (just as a sample second page.) This web page should render in your default browser like this:

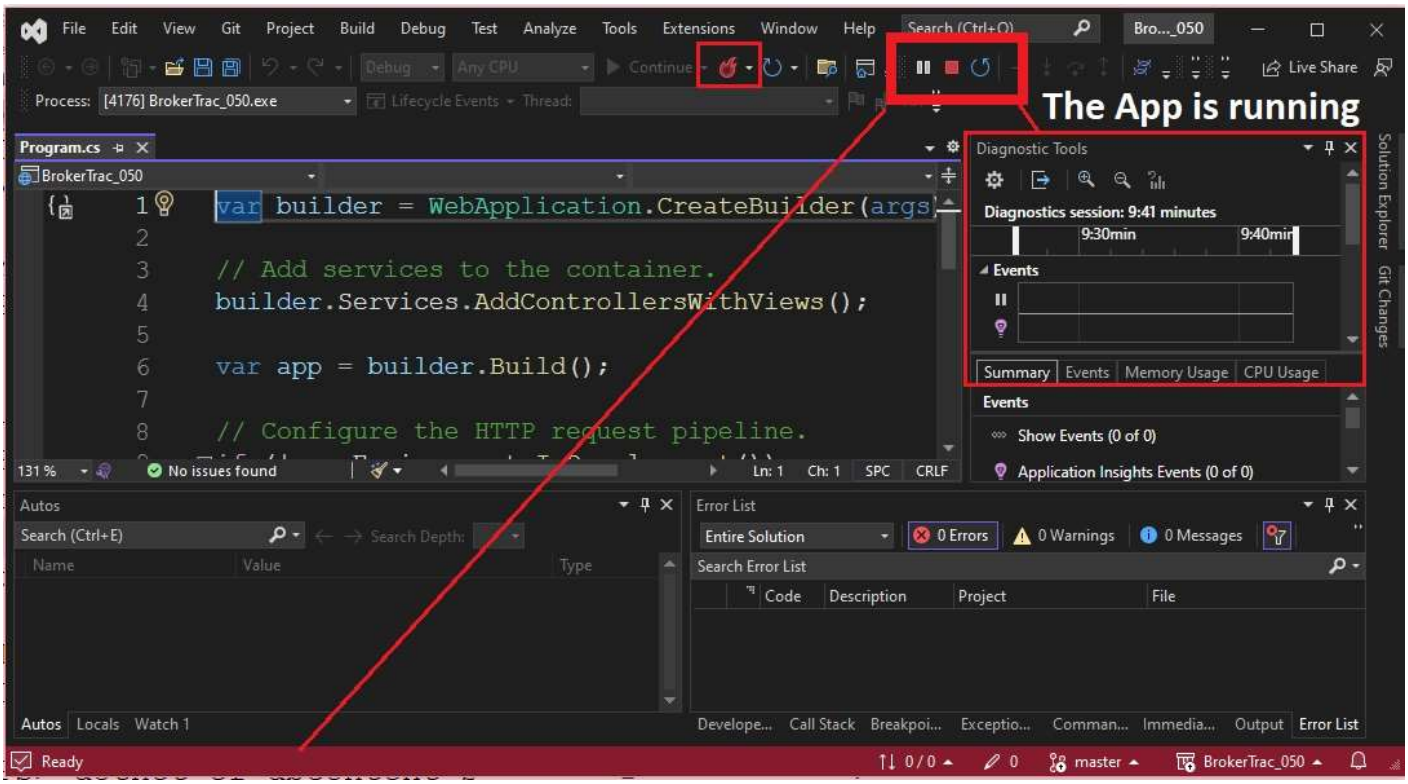


This indicates that we have successfully created the project according to the ASP.NET MVC template.

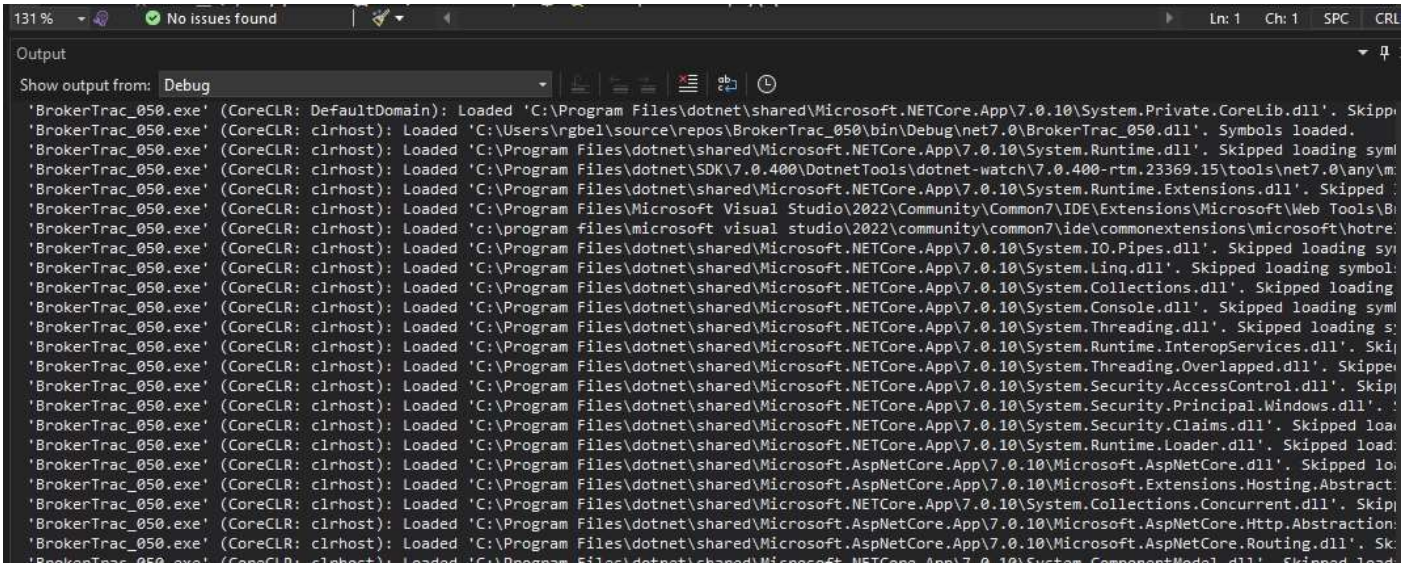
We will be changing this next. But first, note that Visual Studio is running this application.

12) Stop the Program

To close the application, DO NOT just close the browser. You will want to stop the ASP.NET program from running within Visual Studio. (Click the red square, or use **CTRL-C**) Otherwise, subsequent runs may fail, complaining that the process is still running in the background. This is also where the "Hot Reload" feature will prove useful.

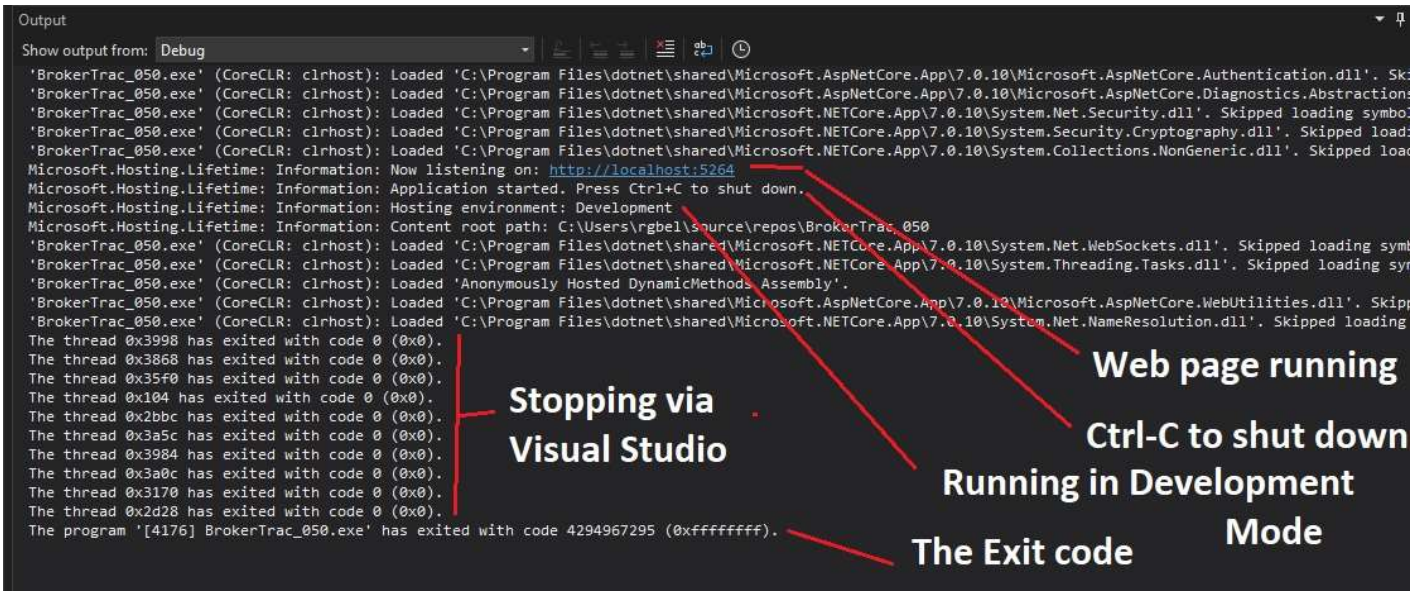


When you stop the Run, you will see the Output window again: There's a lot of messages, but if you scroll to the top portion, you will see:



Notice all the messages where it says it has "loaded" various components.

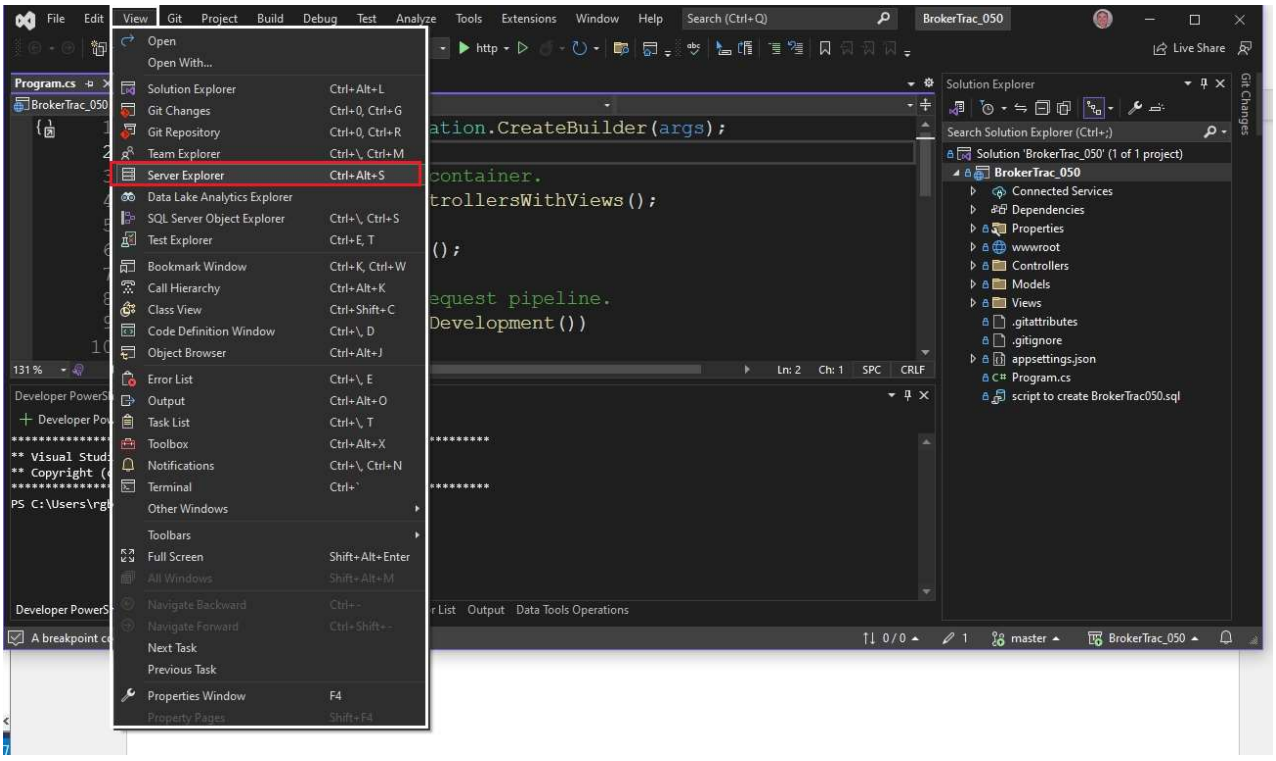
There's more, so scrolling to the bottom of the Output window shows:



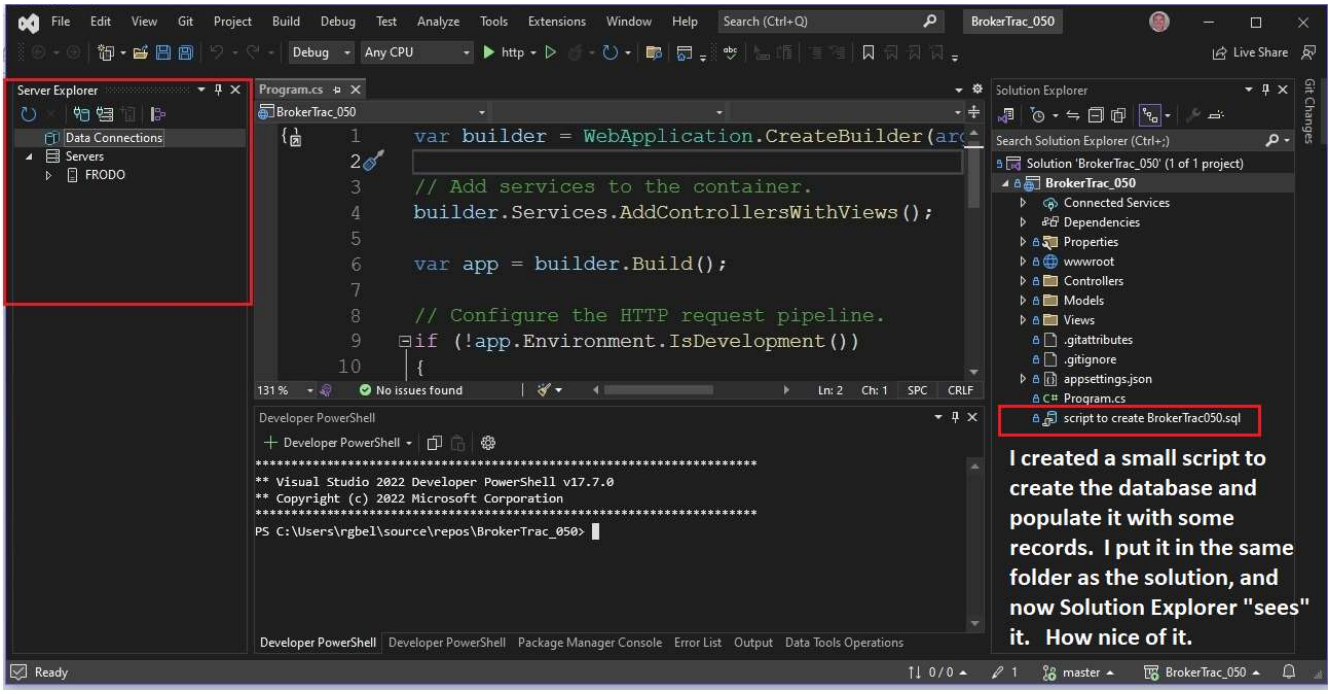
You might want to make note of what port the app is running on: **http://localhost:5264**

This might be different on your machine.

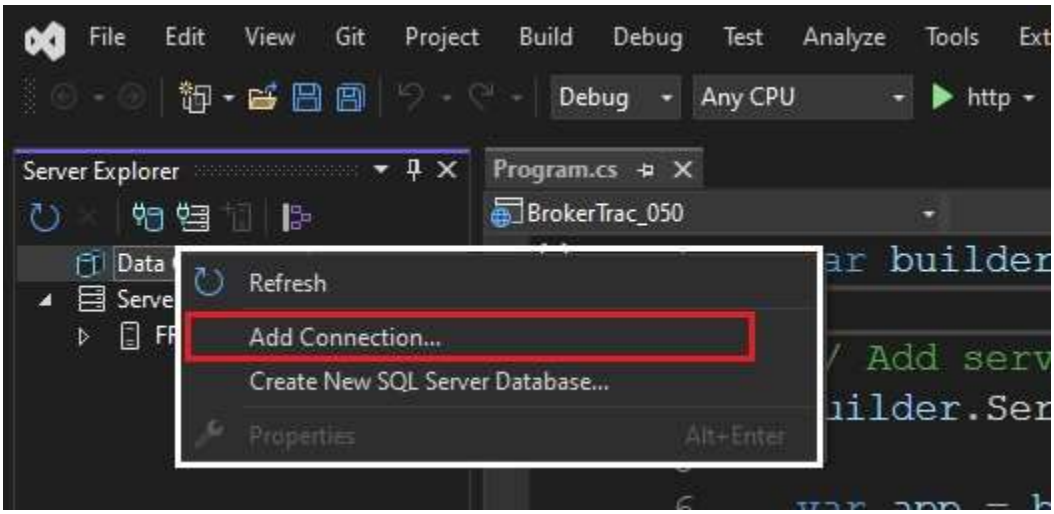
13) Setting up the Server Explorer in Visual Studio with a Data Connection



Notice we don't have any Data Connections set up yet.



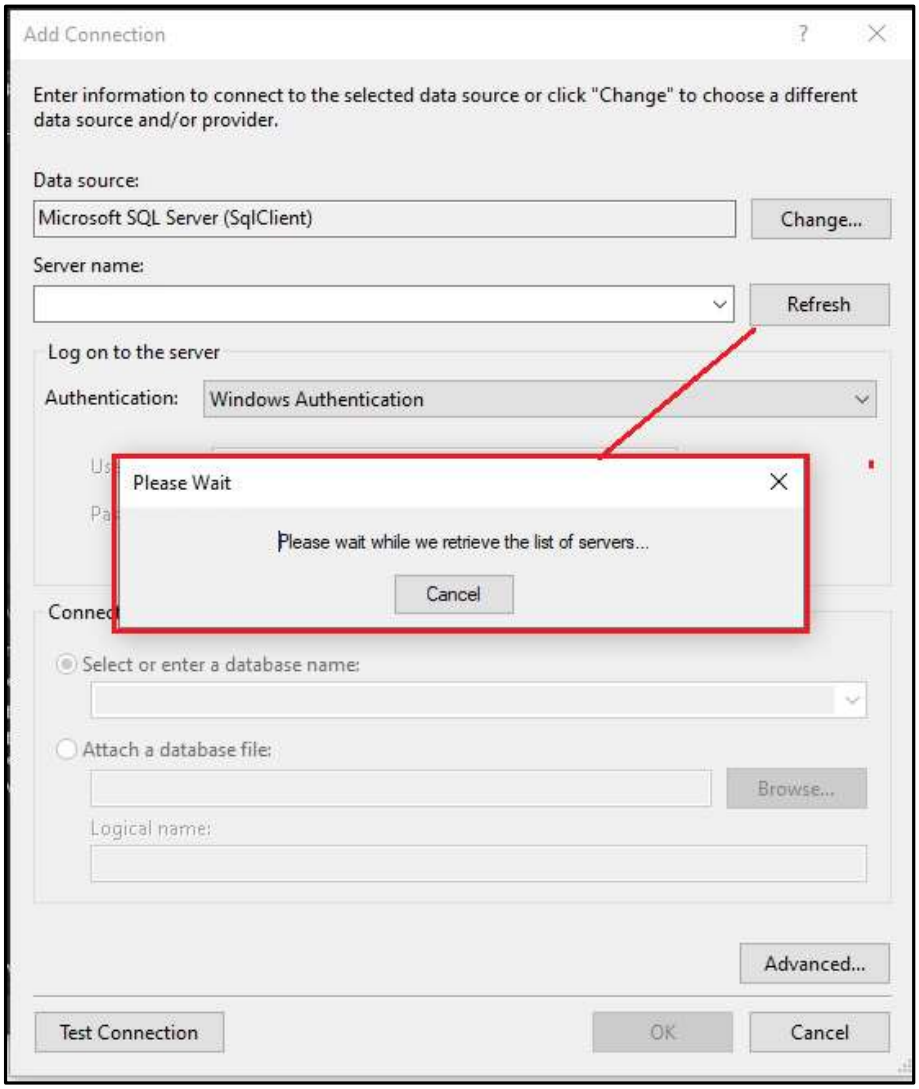
Right Click on **Data Connections**.



Choose **Add Connection**, and if your SQL Server instance is running, you should see this dialog box:

Adding a Connection

It needs to find what SQL Server Instances you have available:

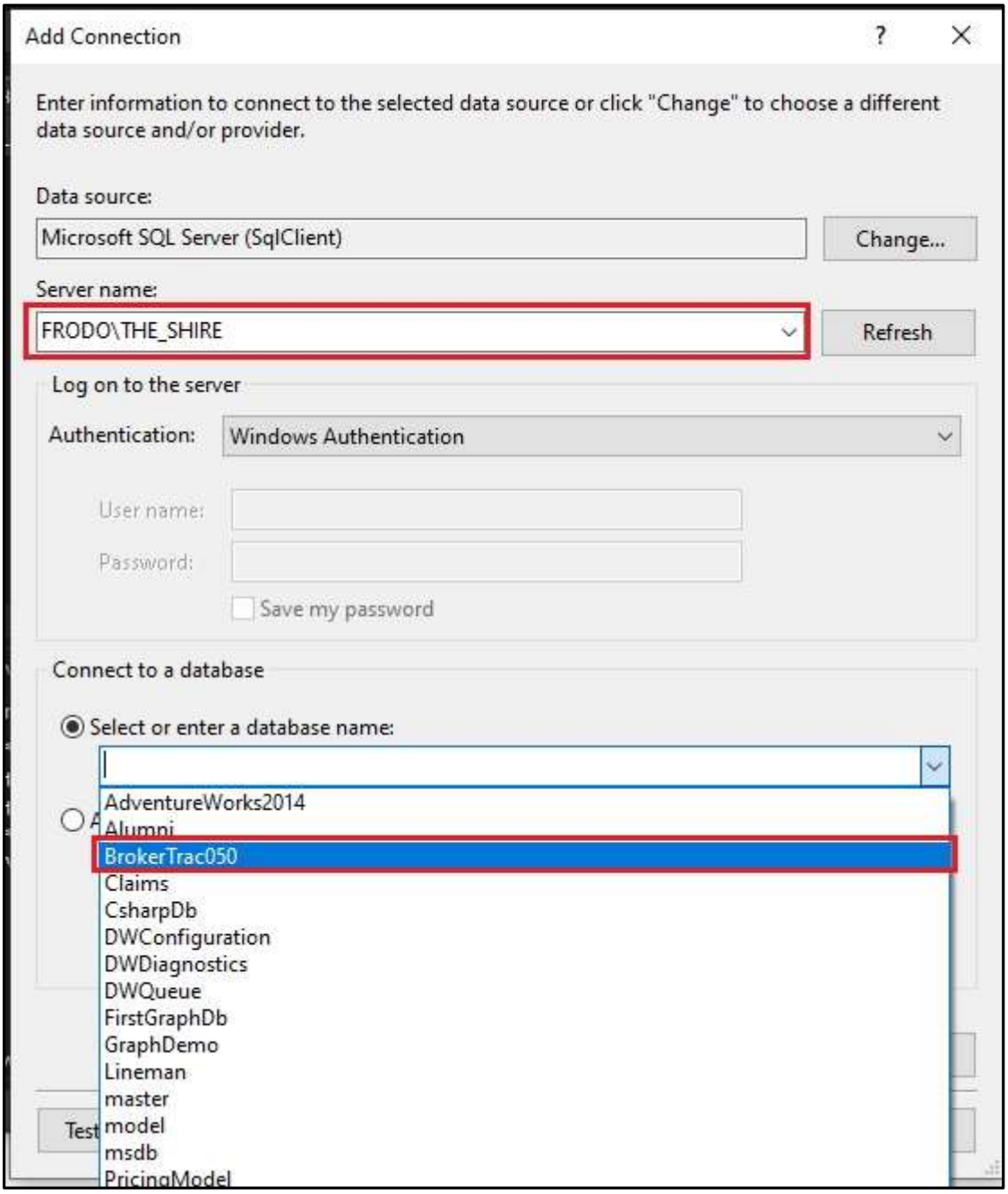


It may take a moment for Visual Studio to find your SQL Server instance.

You may not have to actually click the Refresh button. It seems to do that automatically.

And here I have chosen the SQL Server (named instance) that it found, and it gives me a list of Databases to choose from.

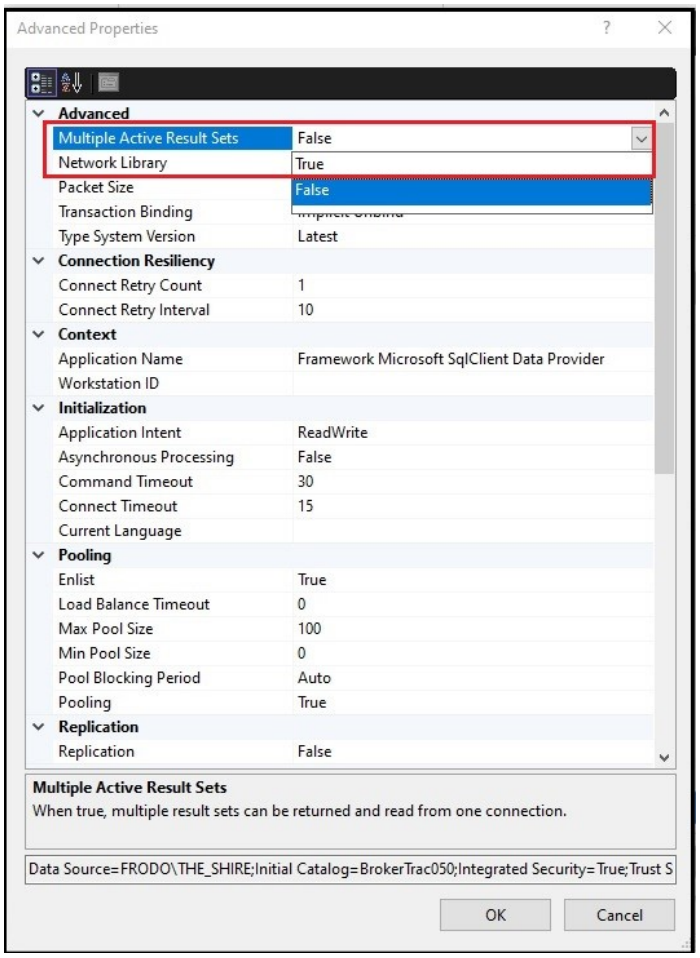
It finds our SQL Server. Now choose the Database.

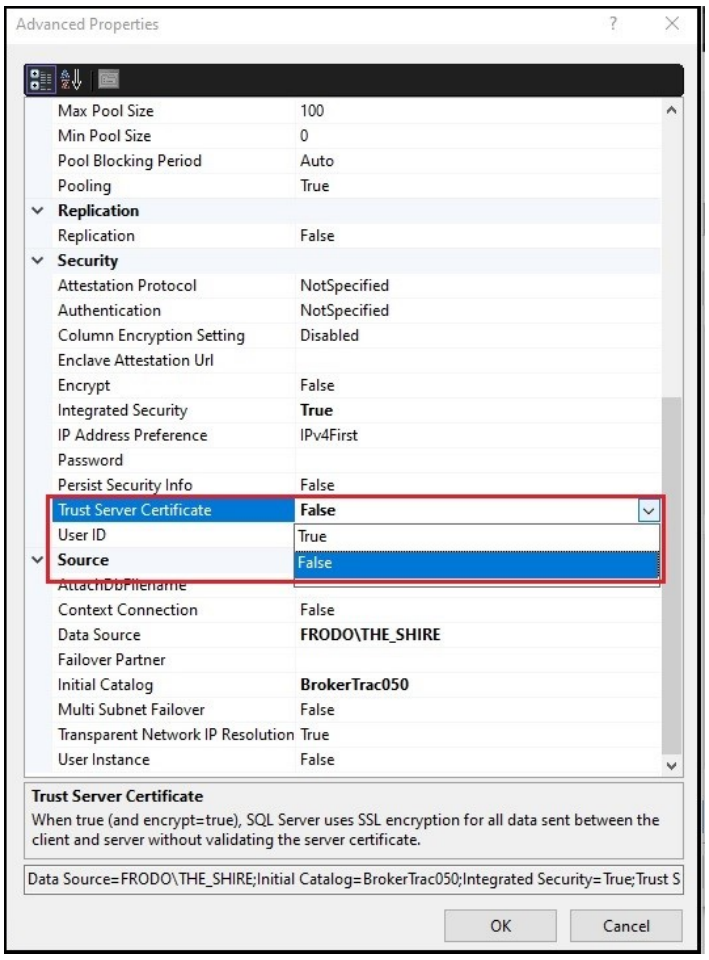


After choosing the Database, click the Advanced button in the lower right.

There are some settings on this Database Connection that we may need to set.

Database Connection – Advanced Settings





These two illustrations are of the same Advanced Settings dialog. Scroll the cursor down to see all the possible settings. Right now we're just going to change two of them to True.

The first setting to change won't affect anything if you're the only person connecting to the database at a time, but setting the **Multiple Active Result Sets** to True might be a good idea. This means more than one person can be receiving record sets from this connection. SQL Server is by default a multi-user system. Also, you might find yourself running the application in more than one browser, even in testing.

The second one is **Trust Server Certificate**. This will help the ASP.NET connection get around security problems. There is another setting on this, when we get to the **Connection String** – which we can also adjust in a moment.

Also, realize that I've set my SQL Server Database to use Windows Integrated Security, that is, my user in Windows is allowed to connect to the database without entering a password. This would not usually be the way you would configure it in a production environment. Setting up a Service Account, specific to the application, or group of users using the application would be a "best practice", and is more secure. Since this is on our own development machine, it's ok, and easier to do it with **Integrated Security** set it to True.

Also note that the **Encript** flag is set to False. Another thing you might not do if this were an enterprise application. We need to explititely set it to False in the connection below, or we'll have problems later.

The Connection String

Notice at the bottom of the Advanced Tab, in a text box, it has been assembling our Connection String

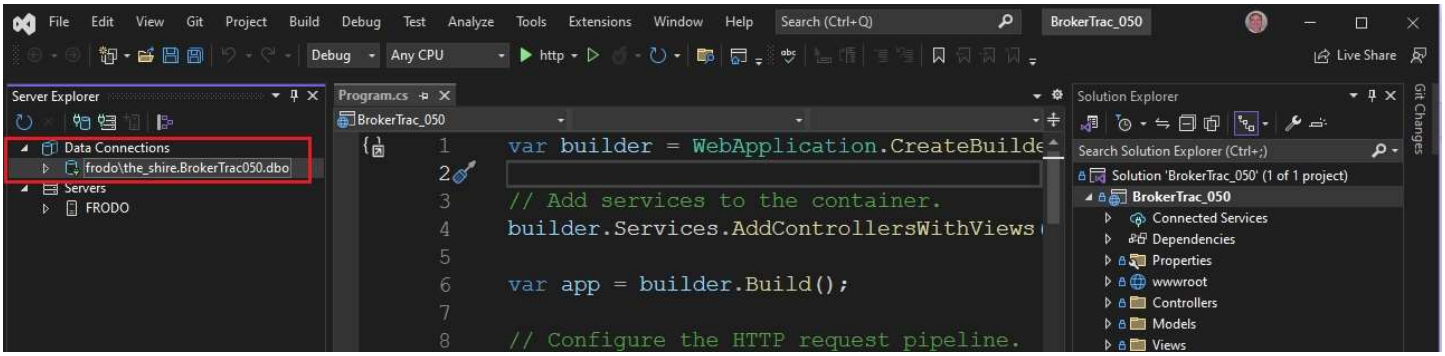
You might want to record this. You can get to it in the Properties of the Data Connection whenever you want, but it's something we're going to need a couple ot times. Here's what mine looks like now:

```
Data Source=FRODO\THE_SHIRE;
Initial Catalog=BrokerTrac050;
Integrated Security=True;
Multiple Active Result Sets=True;
Trust Server Certificate=True;
Encrypt=False;
```

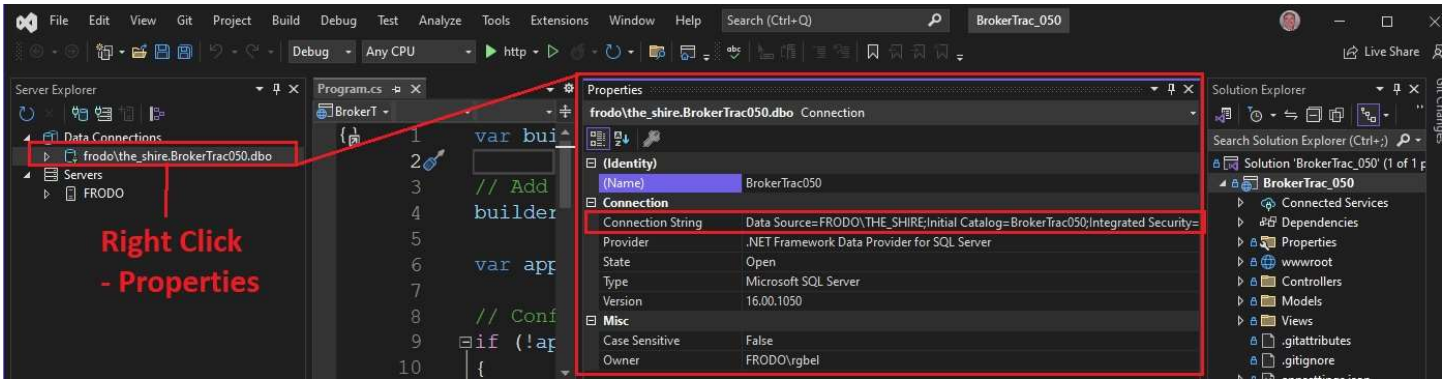
I put some carriage returns after each of the semi-colons, just to illustrate the settings more easily. In your application, you don't do that. It's just one long string.

Click on OK in the Advanced Settings, then OK to create the Data Connection. If you want to, you can click the "Test Connection" button to verify that we can connect successfully to the SQL Server and Database in it.

Back in Visual Studio, you can see the new Data Connection.



Right click on it, and chose Properties any time you want. Here we can also see the Connection String.



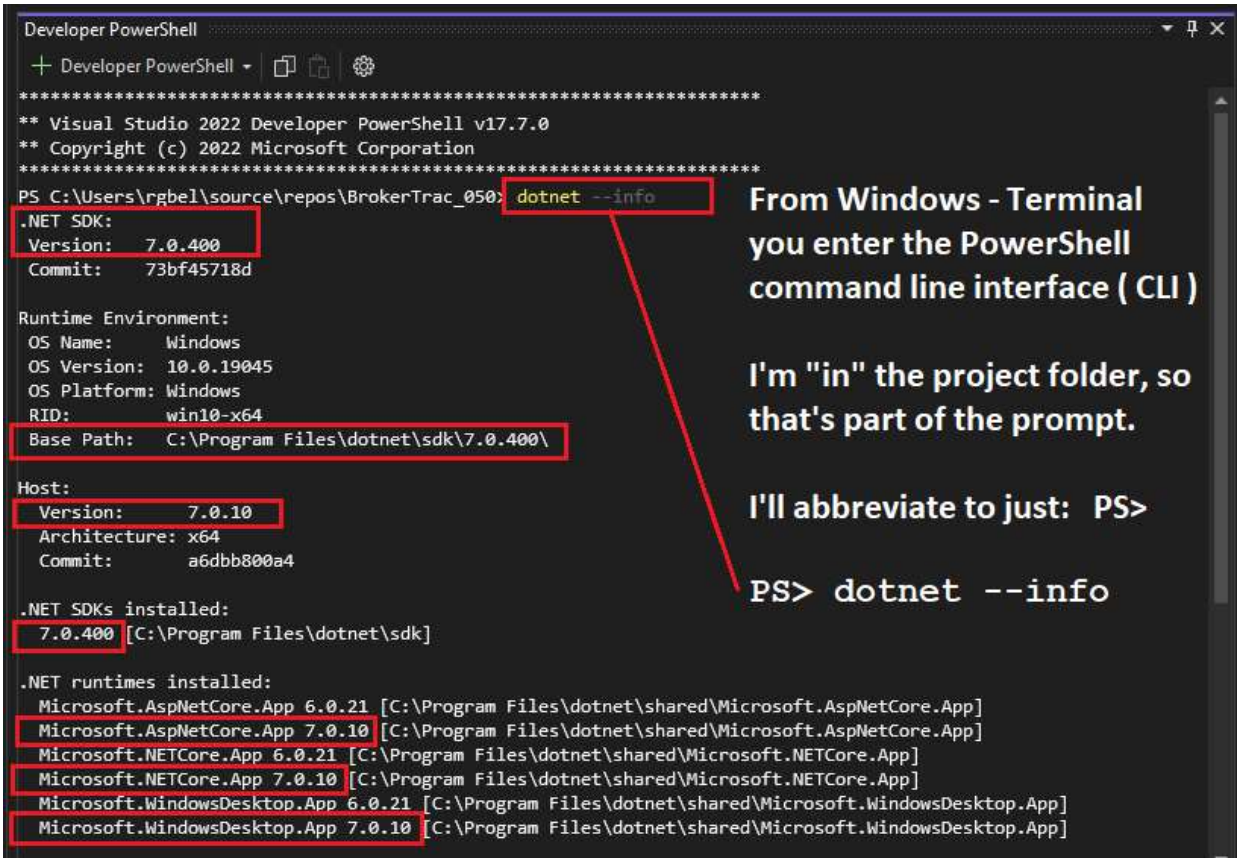
This Data Connection is now registered with Visual Studio, pointing to the BrokerTrac050 database. This connection in your Server Explorer will be available to other Visual Studio projects as well.

14) Setting up Entity Framework, it's Designer, and the SQL Server libraries

Note: You should have a database and the Data Connection set up with some tables and (optionally) some data in the tables by this point. If you haven't done that yet, see **Appendix A** for the scripts.

Let's check the "versons" of Microsoft .Net Runtimes, and SDK we have installed, type in the PowerShell window at the bottom of the Visual Studio IDE. It's down by the Output window. These are tabs, so click the one titled "Developer PowerShell".

PS> dotnet --info



It looks like I have an older version of .Net Runtimes installed (6.0.21), but we'll be using: 7.0.10
The .Net SDK version is 7.0.400

The rest of the output:

```
Other architectures found:
  x86   [C:\Program Files (x86)\dotnet]
    registered at [HKLM\SOFTWARE\dotnet\Setup\InstalledVersions\x86\InstallLocation]

Environment variables:
  Not set

global.json file:
  Not found

Learn more:
  https://aka.ms/dotnet/info

Download .NET:
  https://aka.ms/dotnet/download
PS C:\Users\rgbel\source\repos\BrokerTrac_050>
```

Developer PowerShell | Package Manager Console | Error List | Output | Data Tools Operations

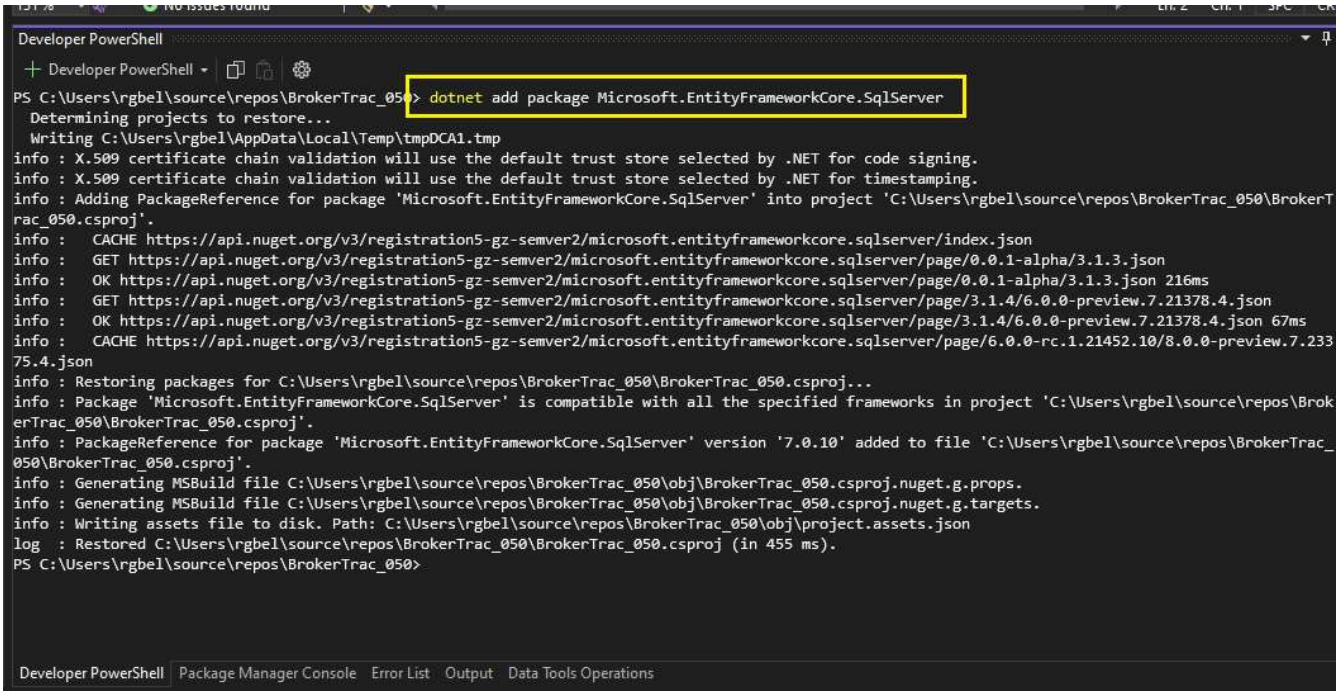
See **Appendix B** for a full list of the **dotnet** command options.

Install Entity Framework

```
PS> dotnet tool install --global dotnet-ef
```

Install Entity Framework Core – SQL Server

```
PS> dotnet install Microsoft.EntityFrameworkCore.SqlServer
```



This makes SQL Server available to this project.

Note: You do have to enter these in the right order. First, get entity framework installed, if it isn't already, then install Entity Framework Core SqlServer, then Entity Framework Core Design.

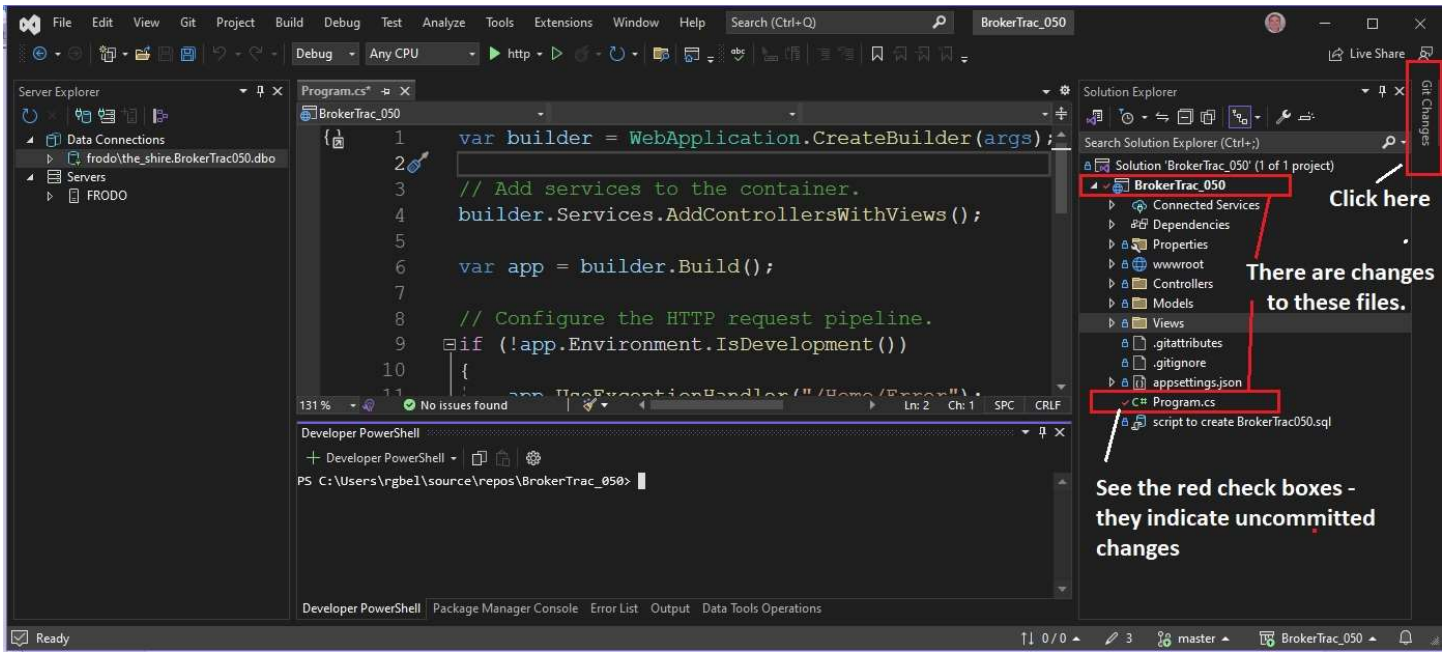
Design depends on SqlServer.

Install Entity Framework Design

`PS> dotnet add package` **`Microsoft.EntityFrameworkCore.Design`**

```
Developer PowerShell
Developer PowerShell
PS C:\Users\rgbel\source\repos\BrokerTrac_050> dotnet add package Microsoft.EntityFrameworkCore.Design
  Determining projects to restore...
  Writing C:\Users\rgbel\AppData\Local\Temp\tmpA7FC.tmp
info : X.509 certificate chain validation will use the default trust store selected by .NET for code signing.
info : X.509 certificate chain validation will use the default trust store selected by .NET for timestamping.
info : Adding PackageReference for package 'Microsoft.EntityFrameworkCore.Design' into project 'C:\Users\rgbel\source\repos\BrokerTrac_050\BrokerTrac
_050.csproj'.
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/index.json
info :   OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/index.json 227ms
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/page/0.0.1-alpha/3.1.4.json
info :   OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/page/0.0.1-alpha/3.1.4.json 66ms
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/page/3.1.5/6.0.0-rc.1.21452.10.json
info :   OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/page/3.1.5/6.0.0-rc.1.21452.10.json 68ms
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/page/6.0.0-rc.2.21480.5/8.0.0-preview.7.23375.4.j
son
info :   OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.entityframeworkcore.design/page/6.0.0-rc.2.21480.5/8.0.0-preview.7.23375.4.js
on 67ms
info : Restoring packages for C:\Users\rgbel\source\repos\BrokerTrac_050\BrokerTrac_050.csproj...
info : Package 'Microsoft.EntityFrameworkCore.Design' is compatible with all the specified frameworks in project 'C:\Users\rgbel\source\repos\BrokerT
rac_050\BrokerTrac_050.csproj'.
info : PackageReference for package 'Microsoft.EntityFrameworkCore.Design' version '7.0.10' added to file 'C:\Users\rgbel\source\repos\BrokerTrac_050
\BrokerTrac_050.csproj'.
info : Generating MSBuild file C:\Users\rgbel\source\repos\BrokerTrac_050\obj\BrokerTrac_050.csproj.nuget.g.props.
info : Generating MSBuild file C:\Users\rgbel\source\repos\BrokerTrac_050\obj\BrokerTrac_050.csproj.nuget.g.targets.
info : Writing assets file to disk. Path: C:\Users\rgbel\source\repos\BrokerTrac_050\obj\project.assets.json
log  : Restored C:\Users\rgbel\source\repos\BrokerTrac_050\BrokerTrac_050.csproj (in 237 ms).
PS C:\Users\rgbel\source\repos\BrokerTrac_050>
```

Now would be an excellent time to Commit and Push our changes to Git and GitHub…

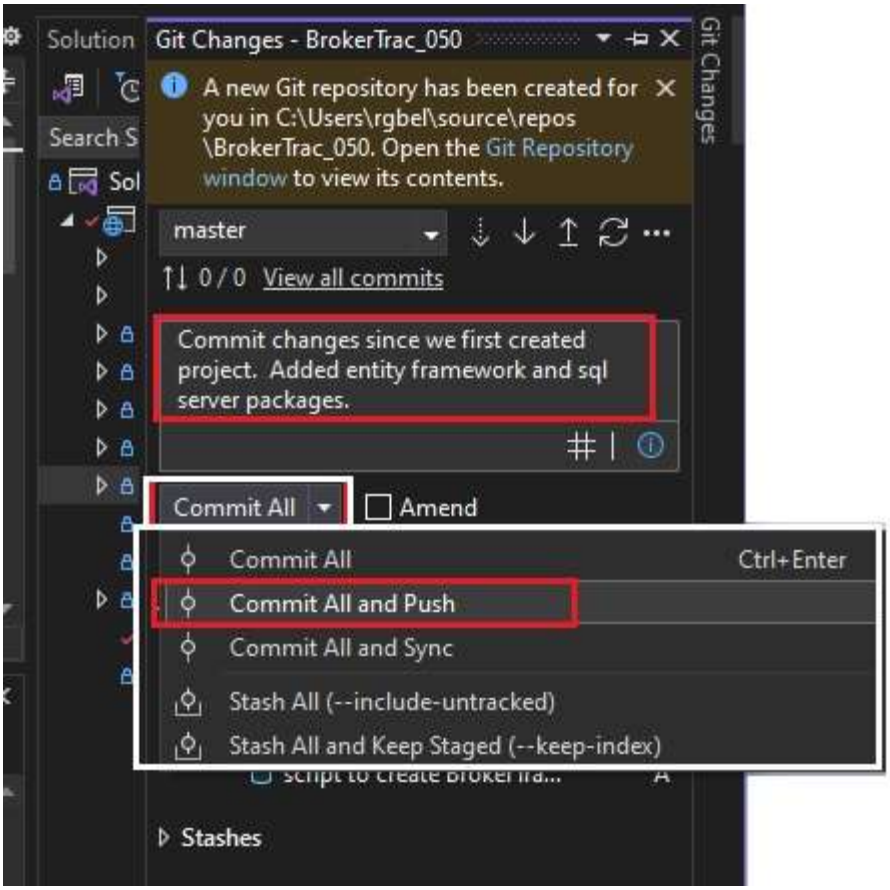


Commit All and Push

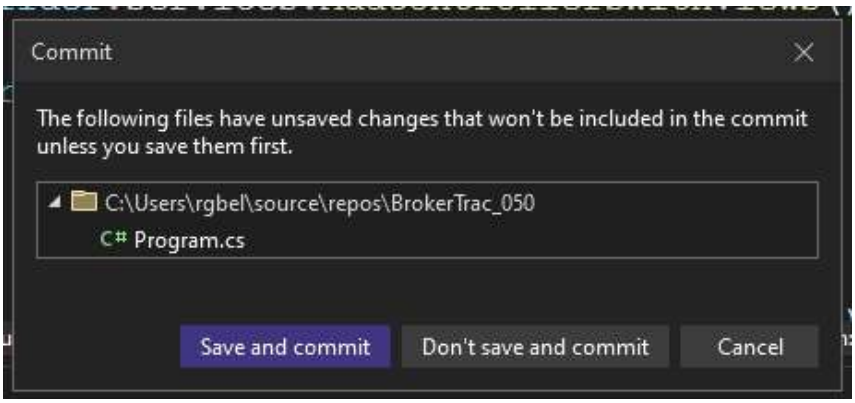
Click the "Git Changes" tab to open it over the Solution Explorer.

Then make any comments you want about the changes we've made.

Let's commit the changes to local Git, and pushes it to GitHub.



But I should have done a "Save All" first! It will notice this, and give you a chance to Save first, then continue on the Commit.



15) Scaffold the Database to Classes in Visual studio

Now to do the "Database First" migration. This is to have Visual Studio create a bunch of classes and a controller for our **Customer** and **Order** tables. Right now these simply exist in the database, but we want the project to have Entity Framework classes that mirror them. This will create class files for those tables. It will also create something known as a database context class. Later, we will do another operation to make functionality (classes with methods) to do Create, Read, Update, and Delete (CRUD) operations on the database, from the application. But in this first step, we're just going to get a database context and classes for the tables it finds in the database.

Here is the statement to run in the PowerShell terminal:

```
PS> dotnet ef dbcontext scaffold 'Data Source=FRODO\THE_SHIRE;Initial
Catalog=BrokerTrac050;Integrated Security=True;Multiple Active Result
Sets=True;Trust Server Certificate=True;Encrypt=False;'
Microsoft.EntityFrameworkCore.SqlServer
--context-dir Data --output-dir Models
```

For this, we do need the Connection String. Since we are "in" the project folder, we don't need to specify the Application name, it will know it's for this project. Here's a note on some options:

dbcontext directs it to create a Dbcontext class for the connection which we need in the project for database access. We'll be dealing with that more later. **Dbcontext** will be a base class name, and **BrokerTrac050Context** will be derived from that, and is the one we will use.

When the scaffolding occurs, the default will be for it to put our dbcontext class and class files for the tables into the main project folder. This usually isn't what we want. We usually create folders in the project to hold and group those kinds of things. We may have many tables. Developers have a few conventions on what to name them. Some people call it a Data Access Layer (DAL), others call it other things. A common pattern that most people won't complain too much about is to place the dbcontext classes into a folder named **\Data** or \DataAccess, and the classes which are for the entities (which equate to our tables) into a folder named **\Models**. So we're adding these switches::

```
--context-dir Data
--output-dir Models
```

Another switch at the end would be **`-force`** This would be to "force" it to re-build everything from scratch, even though it's already created them a first time. It won't let you over write the object's if they already exist, unless you specifically tell it to foce the scaffolding again. If you are doing a "force" to override and re-create the classes, you may also want to use a switch called: **`--no-onconfiguring`** This prevents the utility from re-creating dbcontext information concerning the Connection String if it's already been done once. It's commonly used when re-doing the scaffolding.

See **Appendix C** for a list of all the scaffolding switches.

It's not a problem to have it re-scaffold the classes it creates at this early point in development. We haven't customized anything yet. Later we probably will be – but by that time we won't be doing scaffolding FROM the database, but we'll take more full control of the Database objects. A little more detail on that next:

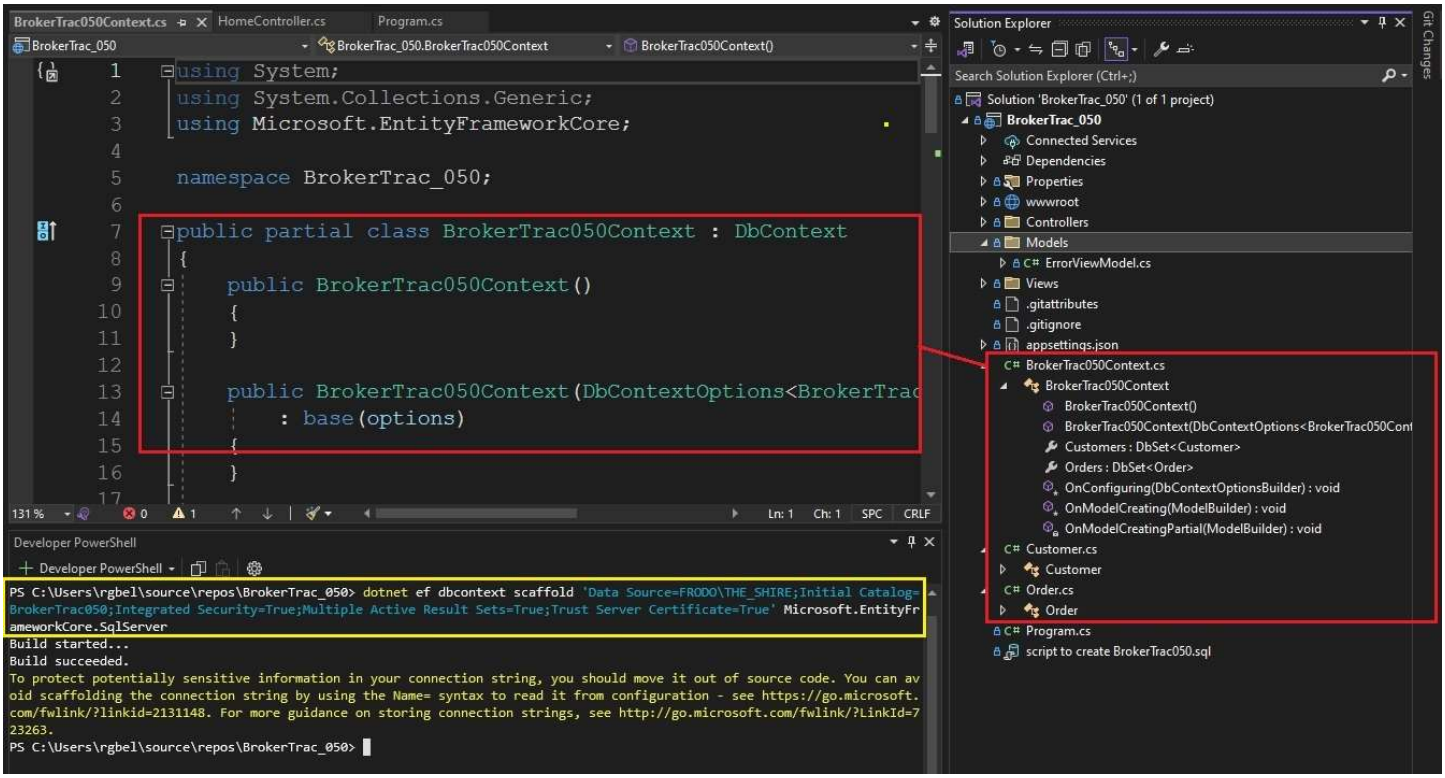
Package Manager (PM>) syntax is *slightly* different:

```
PM> Scaffold-DbContext "Data Source=FRODO\\THE_SHIRE;Initial
Catalog=BrokerTrac050;Integrated Security=True;Multiple Active Result
Sets=True;Trust Server Certificate=True;"
Microsoft.EntityFrameworkCore.SqlServer -ContextDir Data -outputDir Models
```

Database-First, Code-First, and Migrations:

When working with Entity Framework, you can either do a "Code First" approach, where you actually create the Database and Tables via creation of classes in Visual Studio – and incrementally update the database per our programming changes, using what are called "**migrations**"; or we can do a "Database First" approach where we already have the tables in the database, and then "scaffold" them into the Visual Studio project. I prefer a Database First approach, and that's why we set up the SQL Server items per Appendix A. A "Code First" approach would have made those steps unnecessary. After our initial use of "Database First" with scaffolding, we can then use migrations so most of the work we'll be doing which change tables (or adding tables) will be done in Visual Studio. Then we'll push those changes down to the database.

Issue the command shown above into Powershell to do the Database First migration, aka: Scaffolding from the database: Here's what the project looks like now:



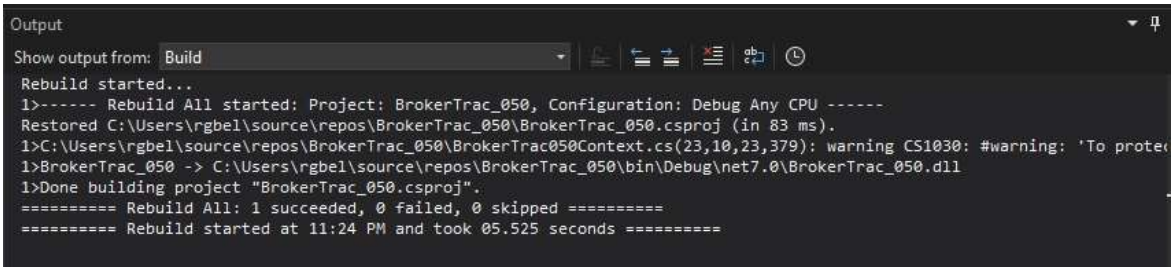
The PowerShell window shows that we entered the command correctly. There is a warning that Visual Studio continually will complain about – that our Connection String is visible – and that's not a best practice. Sometimes we would have a password embedded in that, but even the address of the server is considered sensitive information. We will ignore that.

But notice to the right that it has automatically created a class named: **BrokerTrac050Context.cs** with a bunch of accessor methods. Also classes for the Customer and Order tables. Because I didn't specify a specific folder to place these in, they end up in the project folder – but they can be easily moved. Or we could delete the class files and re-do the scaffold with the -o and -force switches. We will have to be careful though, as this automatically generated code is intact at this point. If we go messing around moving things, we'll want to use the Visual Studio tools for that. Otherwise things might break, as behind the scenes, we have further automatically created (intermediate language assemblies) that get created when we Build the project. Fixing that stuff can be a real chore if we don't do it right. But good for us though: We have check-ins in Git that we can roll back to if we want to get back to a known point, and try this stuff again.

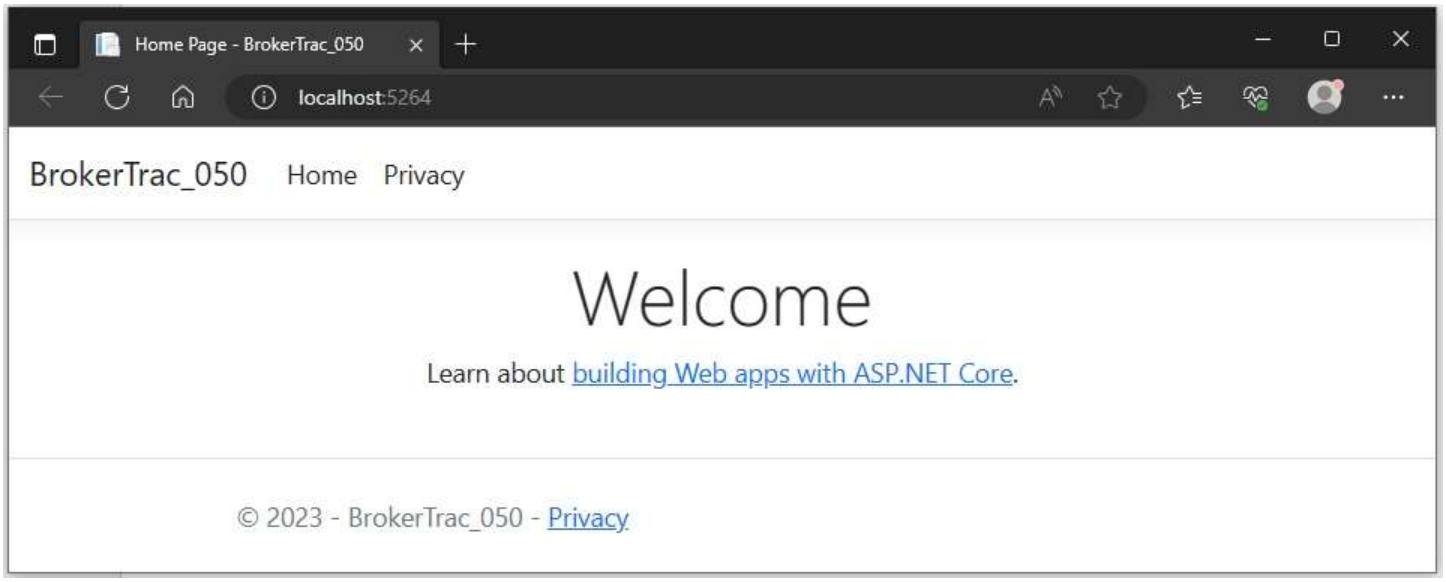
16) Build the Solution again and Run

Let's Build the Solution again (CTRL-Shift-B), and if successful, run it (F5) – to make sure everything still works – and if it does – we should Commit and Push again.

The build succeeded…



And the Web App still runs: (F5)



17) What we've done so far:

What we've done so far is to set up the project according to the Template for an ASP.NET MVC project, connected to a database, installed the NuGet packages (which could have been done through Visual Studio windows, instead of PowerShell), and scaffolded the database tables into the project. We even got everything into Source Control, so we can safely make changes, with the abilty to split off feature branches of the code, or roll back things that aren't working out.

What we will be doing next:

This is a great checkpoint – and gets us ready to do some "actual" development of creating new pages, interacting with the database, and making it do some useful things. Things like:

- Creating views and pages for listing Customers, Orders & Order Lines, and Products. We'll have .cshtml files to help create these lists where we can pick from existing records.

- Creating edit pages for each one of these where we can create new Customers, Products, and Orders – also to edit or delete them as needed. We'll interact with the database through the BrokerTrac050Context class that was created for us. Certain methods for doing this will also be created for us by using the tooling provided with Entity Framework, and the Model-View-Controller pattern which comes with the project.

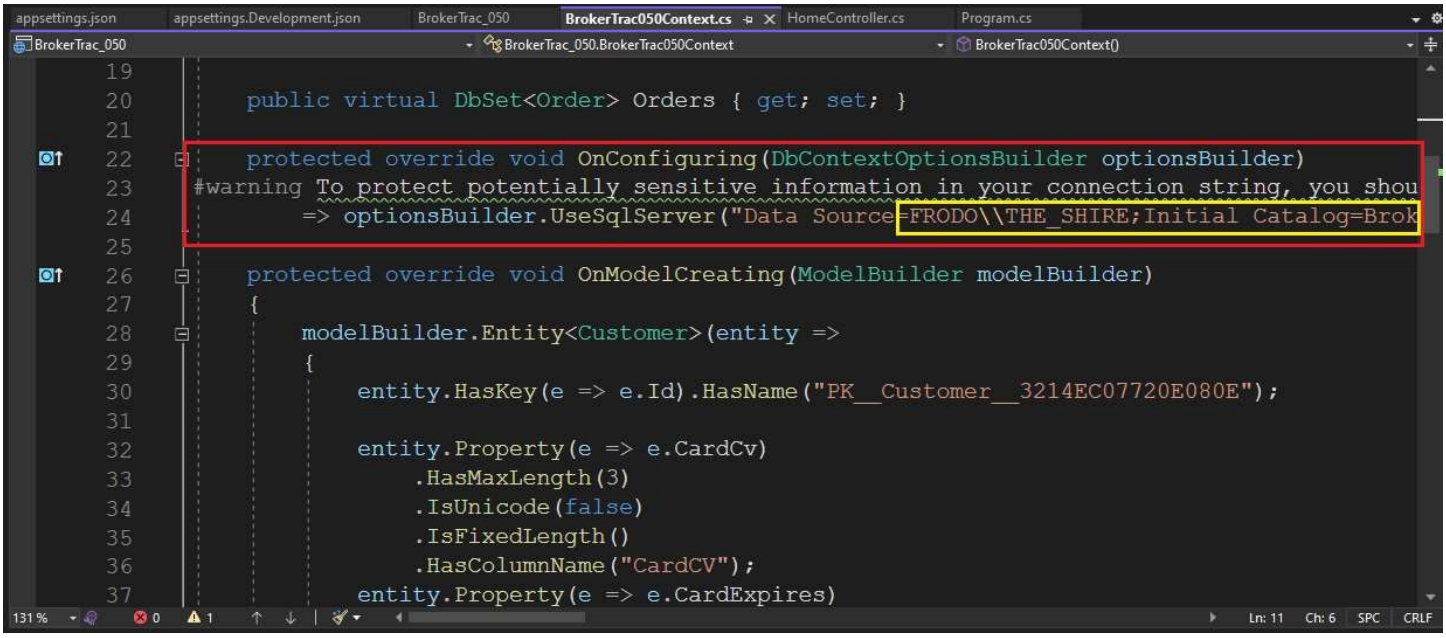
- We'll need suitable navigation between the pages. This will involve routes between the Views.

- Customizing the appearance of the pages, so it doesn't look like the template for the generic project. This will involve some work in the Cascading Style Sheets (CSS) and the JavaScript libraries like Bootstrap.

So far, all we've done is prepare the project environment with the appropriate tools. Now is where the actual work starts.

18) Change where the Connection String is located

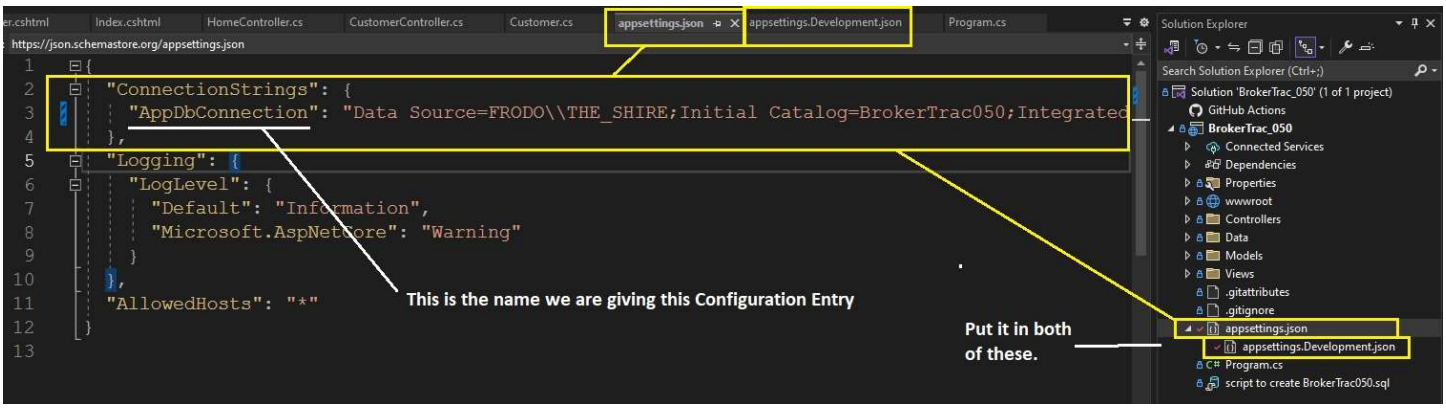
Notice in the class file for BrokerTrac050Context.cs, that the connection string is shown right there – with a warning from Microsoft:



Now you need to open the file **appsettings.json** from the Solution Explorer. You are going to create a new section at the top of the JSON that looks like this: I am naming this connection string setting: "**AppDbConnection**". Just to be safe, put it in **appsettings.Development.json** as well.

```
  "ConnectionStrings": {
    "AppDbConnection":"Data Source=FRODO\\THE_SHIRE;Initial Catalog=BrokerTrac050;Integrated
Security=True;Multiple Active Result Sets=True;Trust Server Certificate=True;Encrypt=False"
  },
```

The connection string cannot have carriage returns in it. It's one long string. Semi-colons need to be between each name=value pairs. The semicolons can have a space after them. Notice the slash between the host computer name (FRODO), and the name of the SQL instance (THE_SHIRE), does have to be "escaped" since usually one slash inside of a string means you're doing some formatting. Putting two slashes says "I really want one slash here".

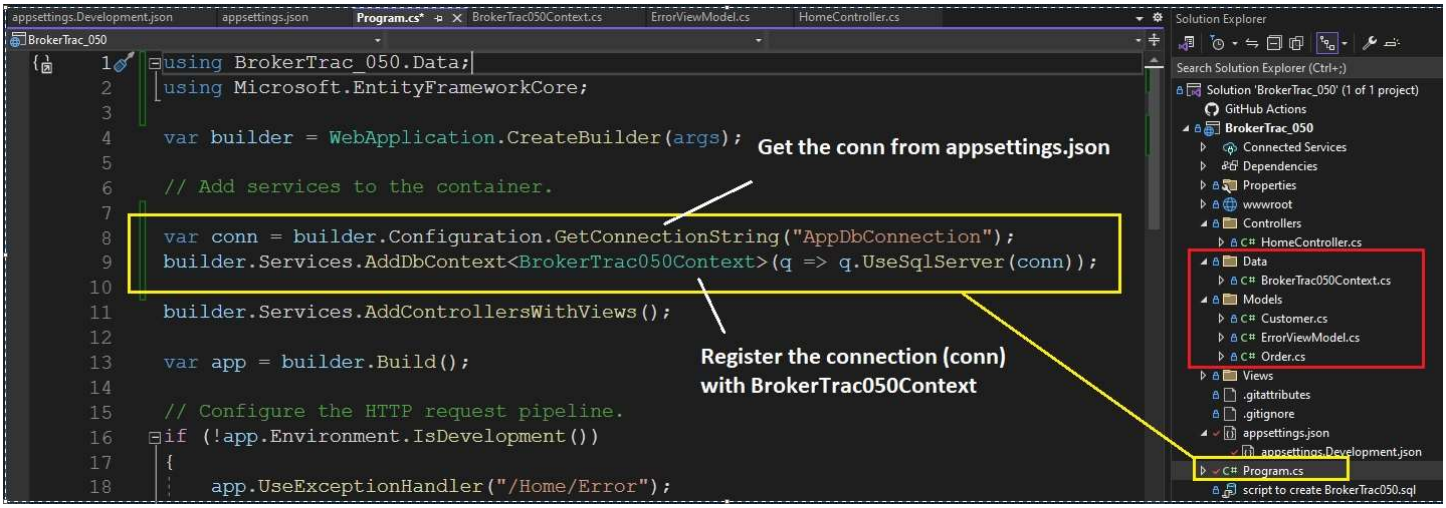


Now, in the main **Program.cs** class file – let's add two lines of code to grab the Connection String (which will be an object named conn, and register that conn object with the Database Context class, BrokerTrac050Context:

After this location at the top:
// Add services to the container.

add these two lines:

```
var conn = builder.Configuration.GetConnectionString("AppDbConnection");
builder.Services.AddDbContext<BrokerTrac050Context>(q => q.UseSqlServer(conn));
```



Now the connection string has been re-located to a better place, we're opening it from appsettings.json, and the database context class knows about it and can connect to the database to access Customers and Orders.

Well, not quite yet. We can see the tables, but now we have to create some functionality to Create, Read, Update, and Delete them. That's abbreviated as CRUD operations. The basic things you do to rows in a table.

You may notice in the Solution Explorer on the right, in red – I re-did the scaffolding, since the first time I did it, the classes ended up at the main project level, and not in the \Data and \Models folder. I simply deleted the Custome.cs, Order.cs and BrokerTrac050Context.cs files manually – and re-did the command. See that on the next page:

```
PS> dotnet ef dbcontext scaffold 'Data Source=FRODO\THE_SHIRE;Initial
Catalog=BrokerTrac050;Integrated Security=True;Multiple Active Result
Sets=True;Trust Server Certificate=True;Encrypt=False'
Microsoft.EntityFrameworkCore.SqlServer --force  --context-dir Data
--output-dir Models  --no-onconfiguring
```

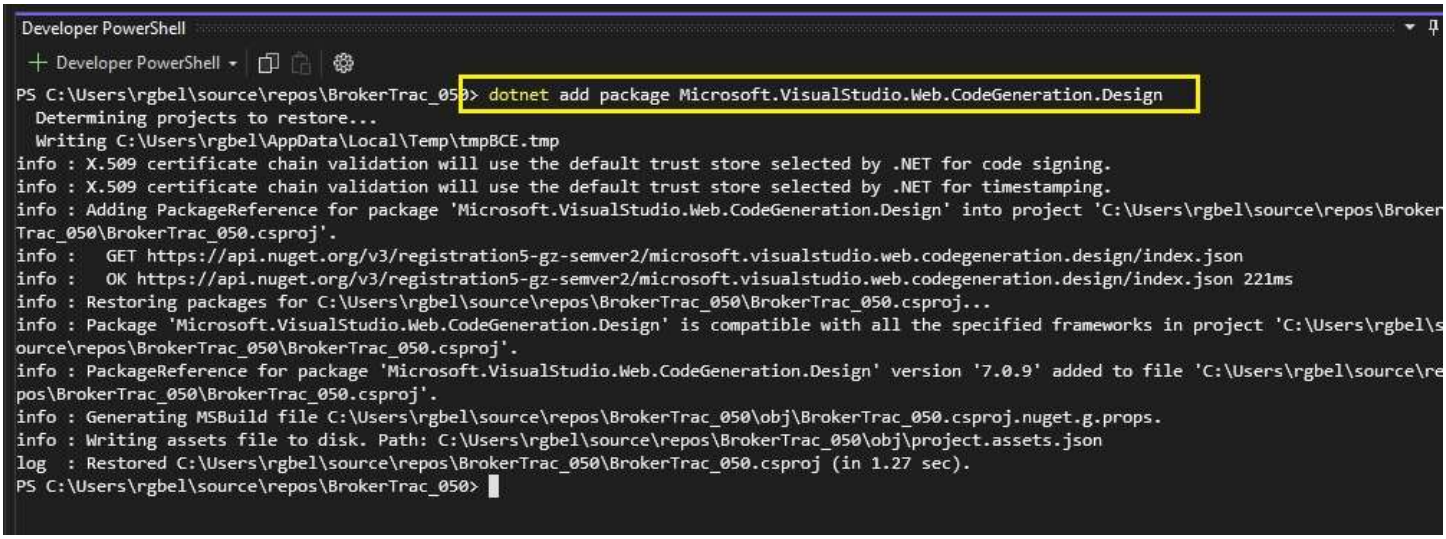
19) Add another tool through PowerShell or Package Manager

Back in the PowerShell terminal, we will install another tool that will help us generate code.

```
PS> dotnet tool install --global dotnet-aspnet-codegenerator
```

And we need yet another tool for web generation:

```
PS> dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
```



Note: If you were doing this in VS Code for Linux or MacOS, you would now need to export the PATH so it knows where the Path is: <u>We don't need to do this on Windows:</u>

```
Export PATH=$PATH/.dotnet/tools:$PATH
```

Just making note of this.

20) Scaffold a controller based on the Model

See Appendix D for a list of all the options on this command

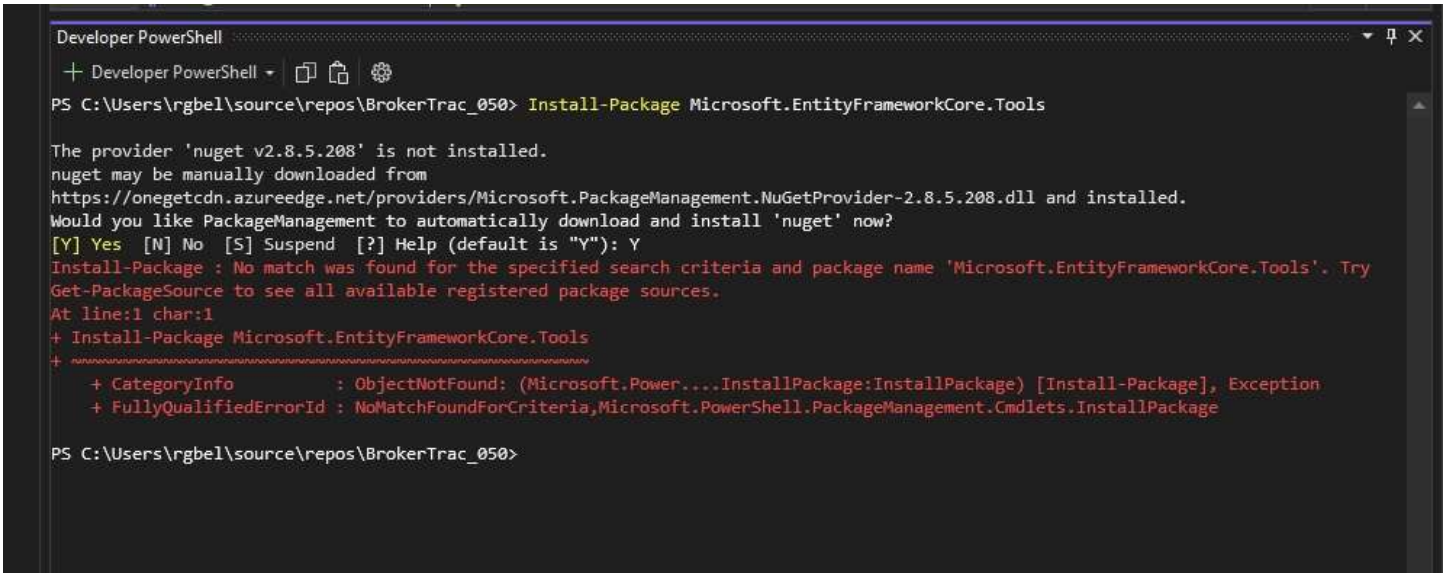
Or see: [dotnet aspnet-codegenerator command | Microsoft Learn]

```
dotnet aspnet-codegenerator controller
--controllerName      CustomerController
--model               Customer
--dataContext         BrokerTrac050Context
--relativeFolderPath  Controllers
--useDefaultLayout
--referenceScriptLibraries
--force
```

```
PS> dotnet aspnet-codegenerator controller --controllerName
CustomerController --model Customer --dataContext BrokerTrac050Context --
relativeFolderPath Controllers --useDefaultLayout --
referenceScriptLibraries  --force
```

We will, however need this first:

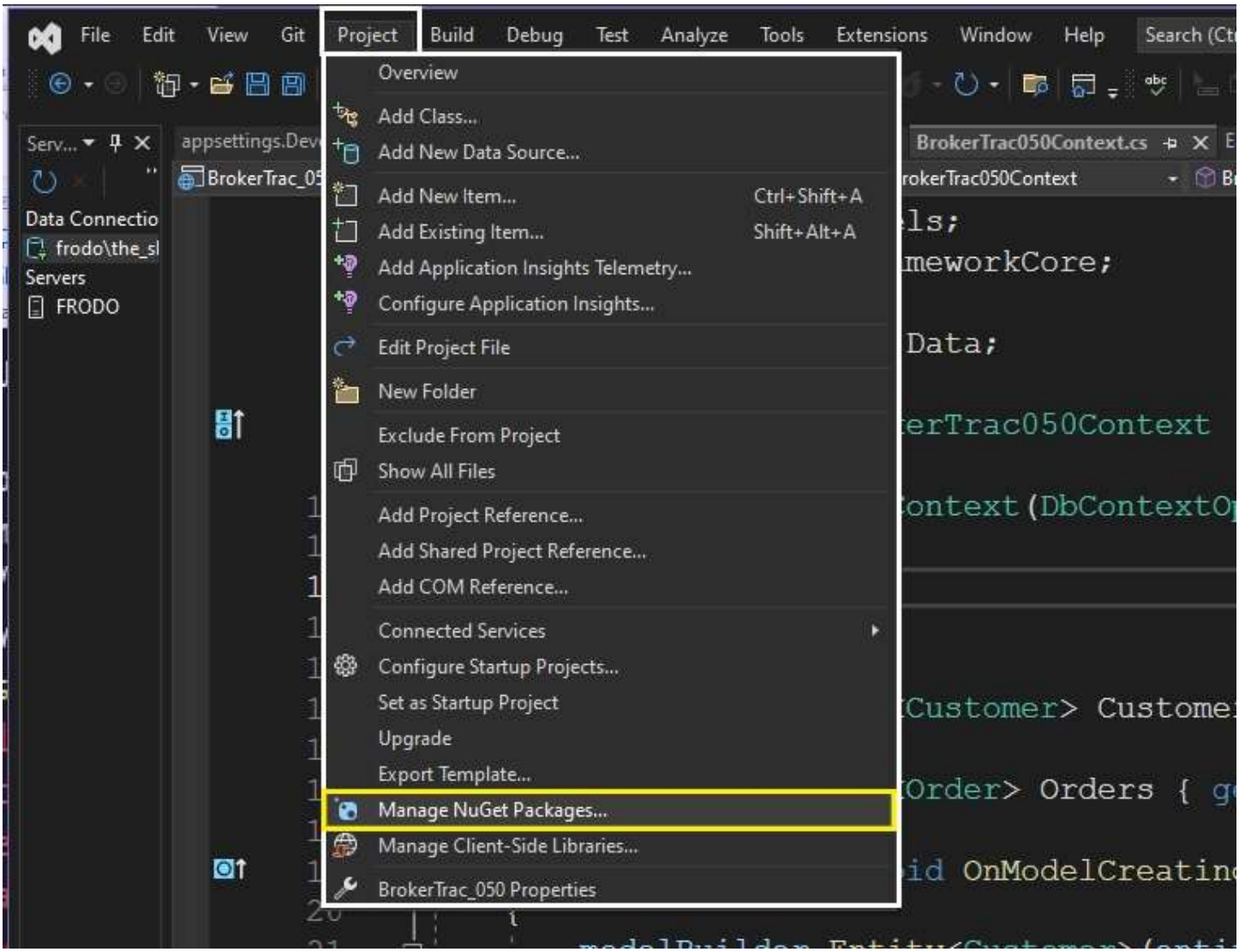
```
PS> Install-Package Microsoft.EntityFrameworkCore.Tools
```



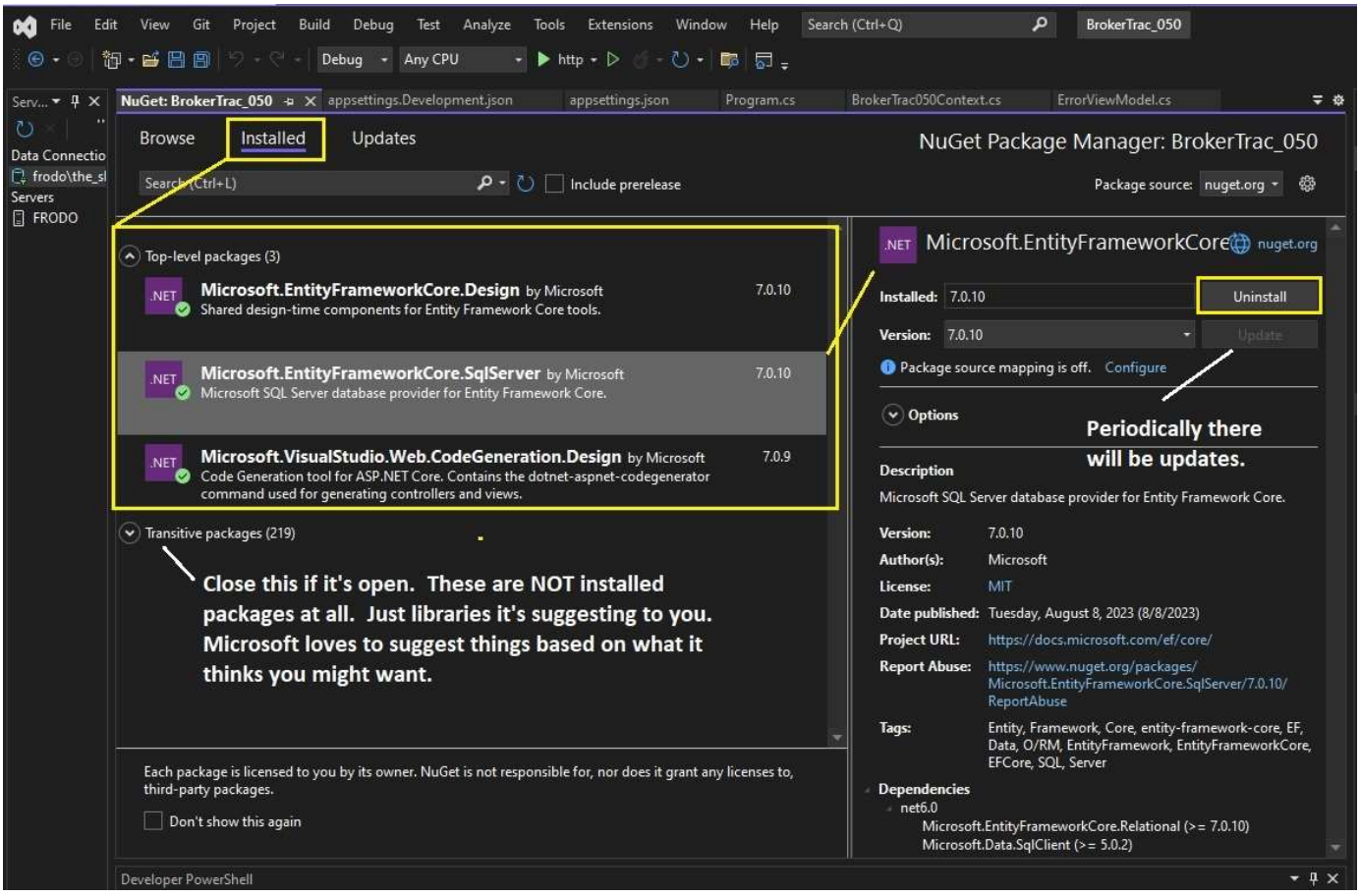
But I'm getting an error with NuGet. I answered "Y" to "get the new version", but it still fails.

Let's talk getting NuGet packages from the Visual Studio IDE though. This is actually how most people do it.

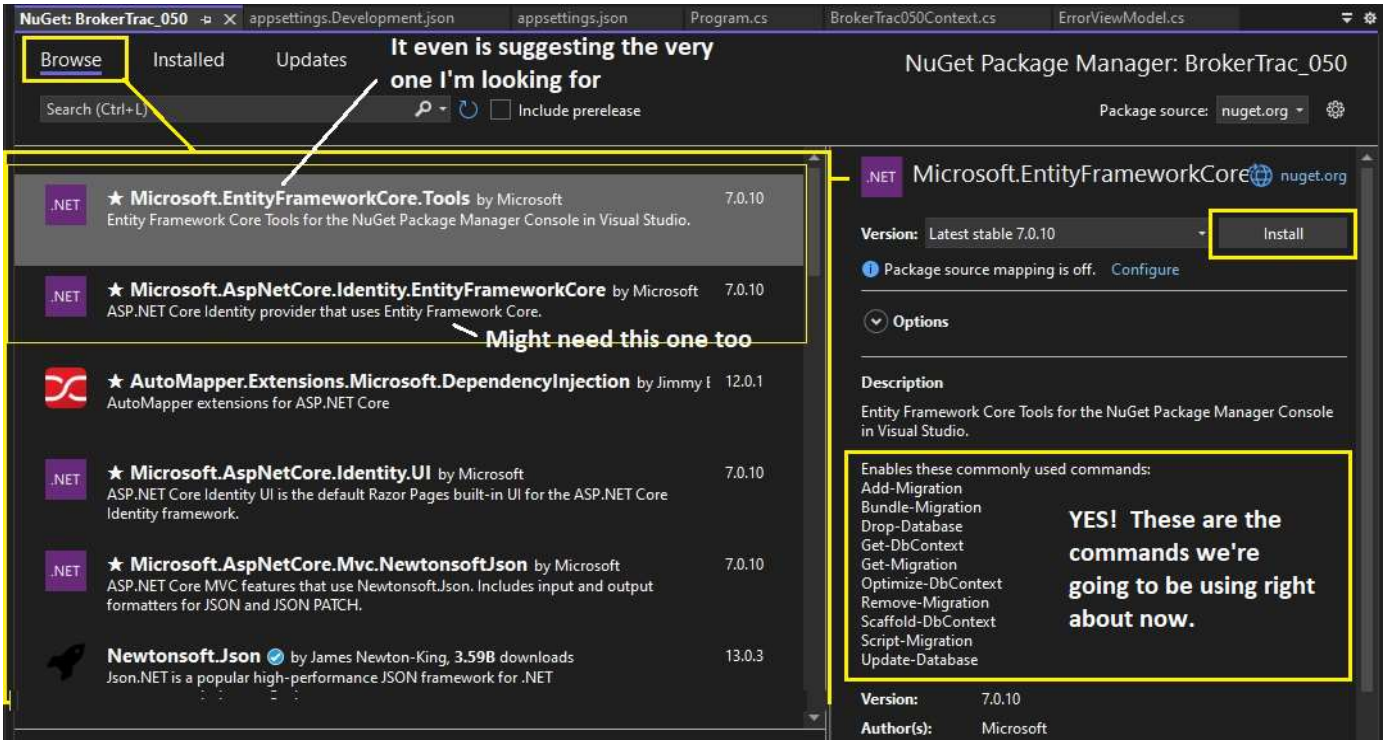
Visual Studio has the NuGet Package Manger built right into the Integrated Development Environment – which is under: **Project**… **Manage NuGet Packages**



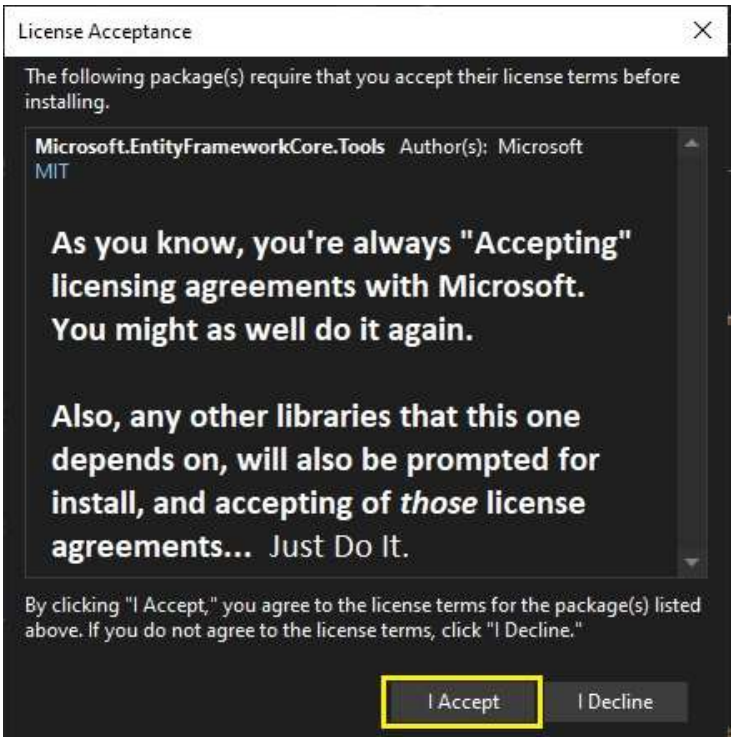
You will initially see a list of the installed Packages



Then go to Browse, and we can search for the packages we need.



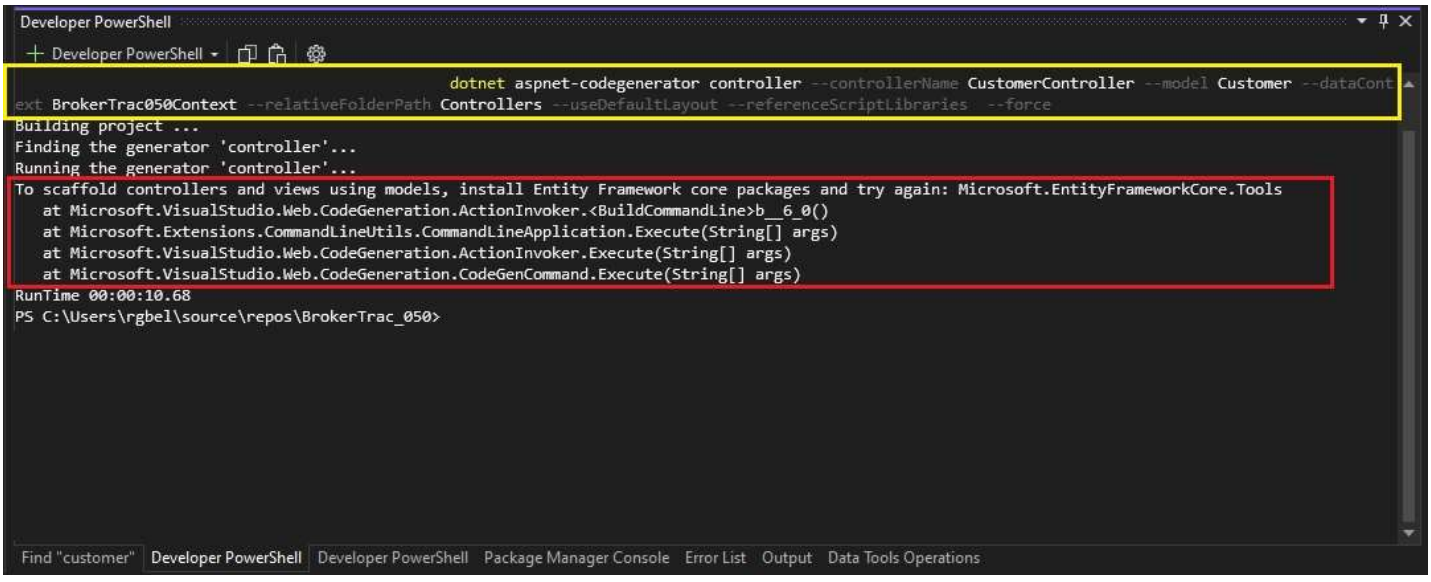
And of course, read the full license agreement, which changes every week or so…



Now let's try this again:

```
PS> dotnet aspnet-codegenerator controller --controllerName
CustomerController --model Customer --dataContext BrokerTrac050Context --
relativeFolderPath Controllers --useDefaultLayout --
referenceScriptLibraries  --force
```

And I am still getting an error on this:



The error reads:

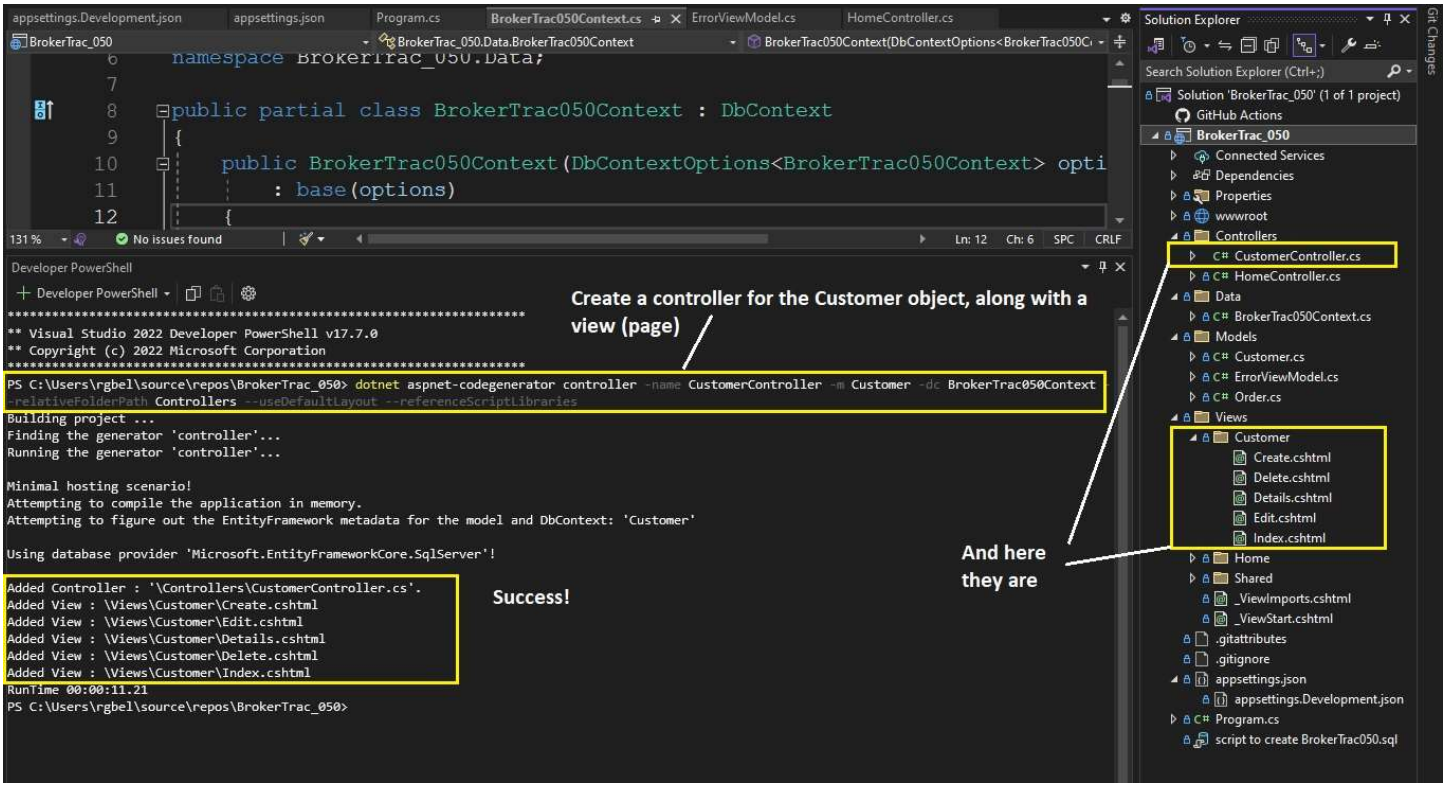
To scaffold controllers and views using models, install Entity Framework core packages and try again: Microsoft.EntityFrameworkCore.Tools at Microsoft.VisualStudio.Web.CodeGeneration.ActionInvoker.<BuildCommandLine>b__6_0() at Microsoft.Extensions.CommandLineUtils.CommandLineApplication.Execute(String[] args) at Microsoft.VisualStudio.Web.CodeGeneration.ActionInvoker.Execute(String[] args) at Microsoft.VisualStudio.Web.CodeGeneration.CodeGenCommand.Execute(String[] args)

Well – I had *just* installed the Tools package, so I saved everything, Re-Built the application, checked it in to GitHub, closed Visual Studio – then re-opened the project and tried it again.

This time it worked: I used the abreviated switches on the commands. I'll see if the verbose ones work when I do the Orders. I also took off the -f (--force option, which would not be needed this first time.)

This worked:

```
dotnet aspnet-codegenerator controller -name CustomerController -m
Customer -dc BrokerTrac050Context --relativeFolderPath Controllers --
useDefaultLayout –referenceScriptLibraries
```



OK – so now we actually have a Customer controller object, and some Customer pages. Index will list all Customers, then there's pages to Create, Delete, and Update, and Show them.

21) Adjust the Navigation Bar of the Index page.

We will need a way to navigate over to the Customers page. I'm going to do this by creating a sub-navigation, just to the Customers List page. Over in the Solution Explorer, there is a page under the **Shared** folder named _Layout.cshtml Open this up. You will see a bunch of HTML. This is the skeleton of the navigation for the main index page we saw when the application ran in the browser.

Around line 15, after the <body> and <header> declarations – there is a part that begins with <nav>

This is the navigation bar element at the top of the page that came with this canned installation of an ASP NET MVC web application. Now we're ready to start customizing things. This walk through I'm doing isn't about designing wonderful websites. It's to show how to interact with the database. The navigation we'll do here will be very basic. All I need right now is to extend that navigation so that it can go to a Customer Listing page. That is – I want it to use the CustomerController to connect to the database, pull up a listing of those customers sitting in that table – and render them as a list on the web page. So:

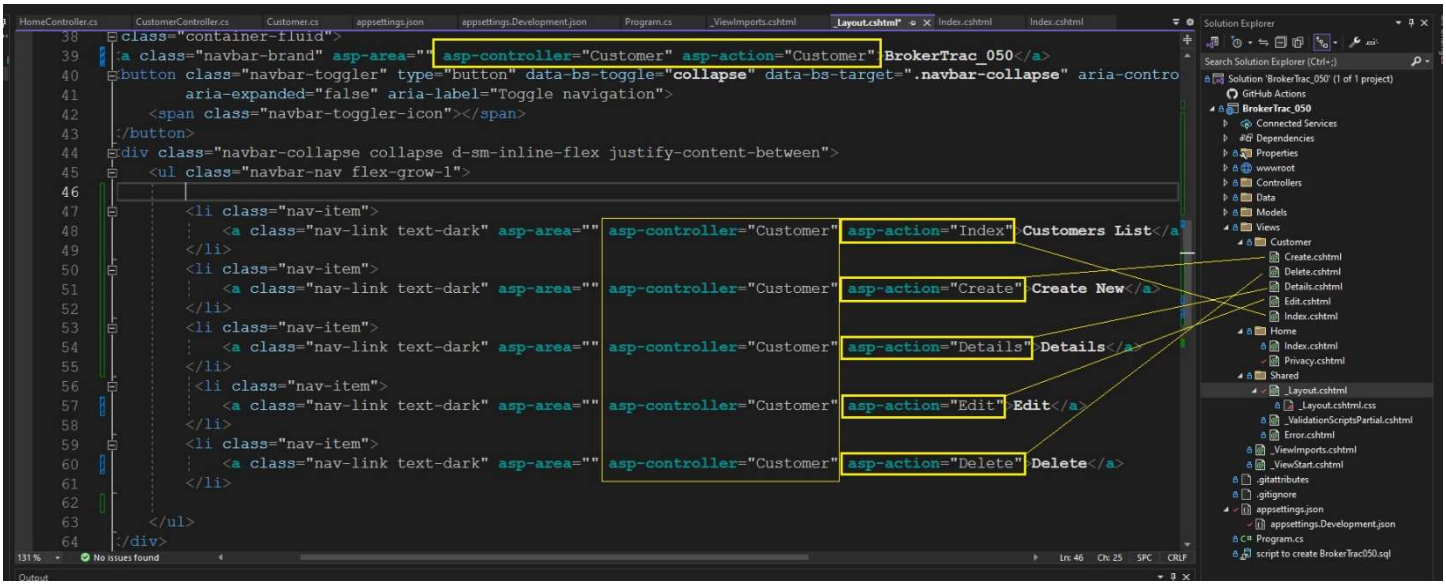
```
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-dark bg-dark border-
bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-
action="Index">BrokerTrac_050</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">

                        <li class="nav-item">
                            <a class="nav-link text-light" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-light" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
                        </li>

                    </ul>
                </div>
            </div>
        </nav>
```

What I'm going to do is Copy this entire chunk, and then make a copy of it – right after that bottom ending </nav> tag. It will be a second navigation bar, right under the first one (which I'll leave alone) – but this second one will be just for navigating between customer pages. Here's how I've edited the second navigation bar:

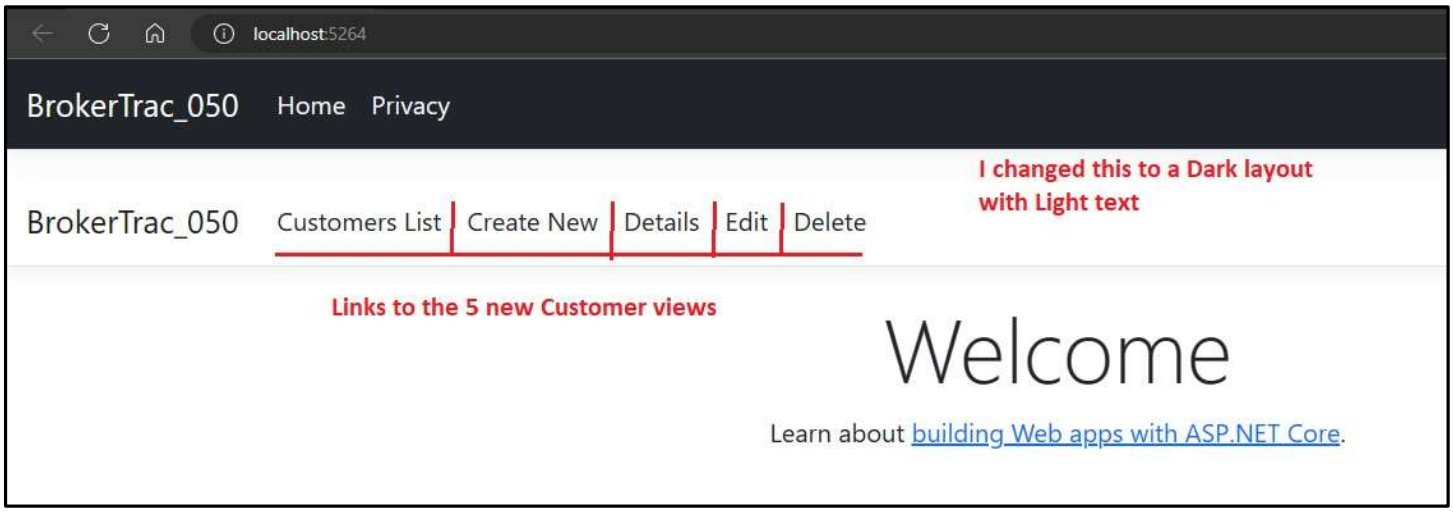
```
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-
bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Customer" asp-
action="Customer">BrokerTrac_050</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">

                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Customer"
asp-action="Index">Customers List</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Customer"
asp-action="Create">Create New</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Customer"
asp-action="Details">Details</a>
                        </li>
                         <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Customer"
asp-action="Edit">Edit</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Customer"
asp-action="Delete">Delete</a>
                        </li>

                    </ul>
                </div>
            </div>
        </nav>
```

This will create five new Navigation buttons in the second Nav Bar, one for each of the View (Pages) that were created for us in the prior section.

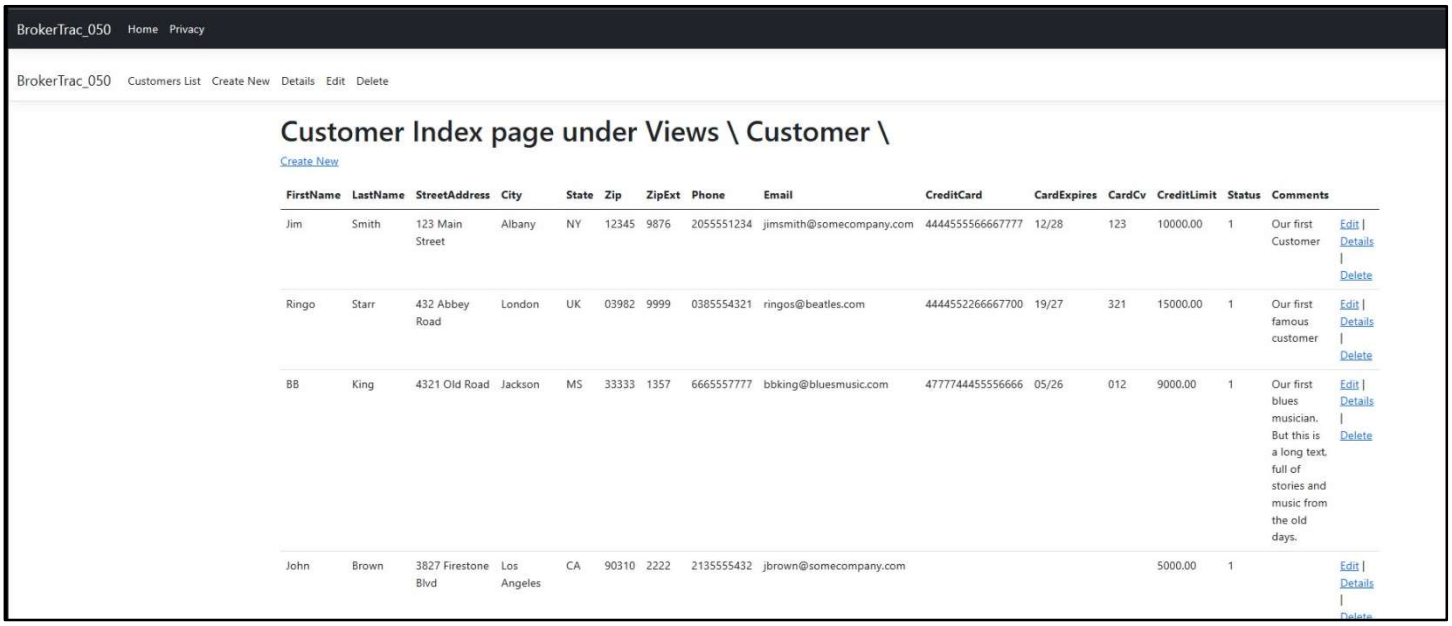
Here's how those entries map to the Customer Views:



Running the application now - It will show the two Navigation bars:



Now, click on the **Customer List**



FirstName	LastName	StreetAddress	City	State	Zip	ZipExt	Phone	Email	CreditCard	CardExpires	CardCv	CreditLimit	Status	Comments	
Jim	Smith	123 Main Street	Albany	NY	12345	9876	2055551234	jimsmith@somecompany.com	4444555566667777	12/28	123	10000.00	1	Our first Customer	Edit | Details | Delete
Ringo	Starr	432 Abbey Road	London	UK	03982	9999	0385554321	ringos@beatles.com	4444552266667700	19/27	321	15000.00	1	Our first famous customer	Edit | Details | Delete
BB	King	4321 Old Road	Jackson	MS	33333	1357	6665557777	bbking@bluesmusic.com	4777744455556666	05/26	012	9000.00	1	Our first blues musician. But this is a long text, full of stories and music from the old days.	Edit | Details | Delete
John	Brown	3827 Firestone Blvd	Los Angeles	CA	90310	2222	2135555432	jbrown@somecompany.com				5000.00	1		Edit | Details | Delete

As you can see, the application connected to the database and retrieved a list of the Customers we had in the database. Also notice that on the very right hand side, there are links to Edit, Details and Delete next to each row. This is the standard "off the shelf" Index page for a table.

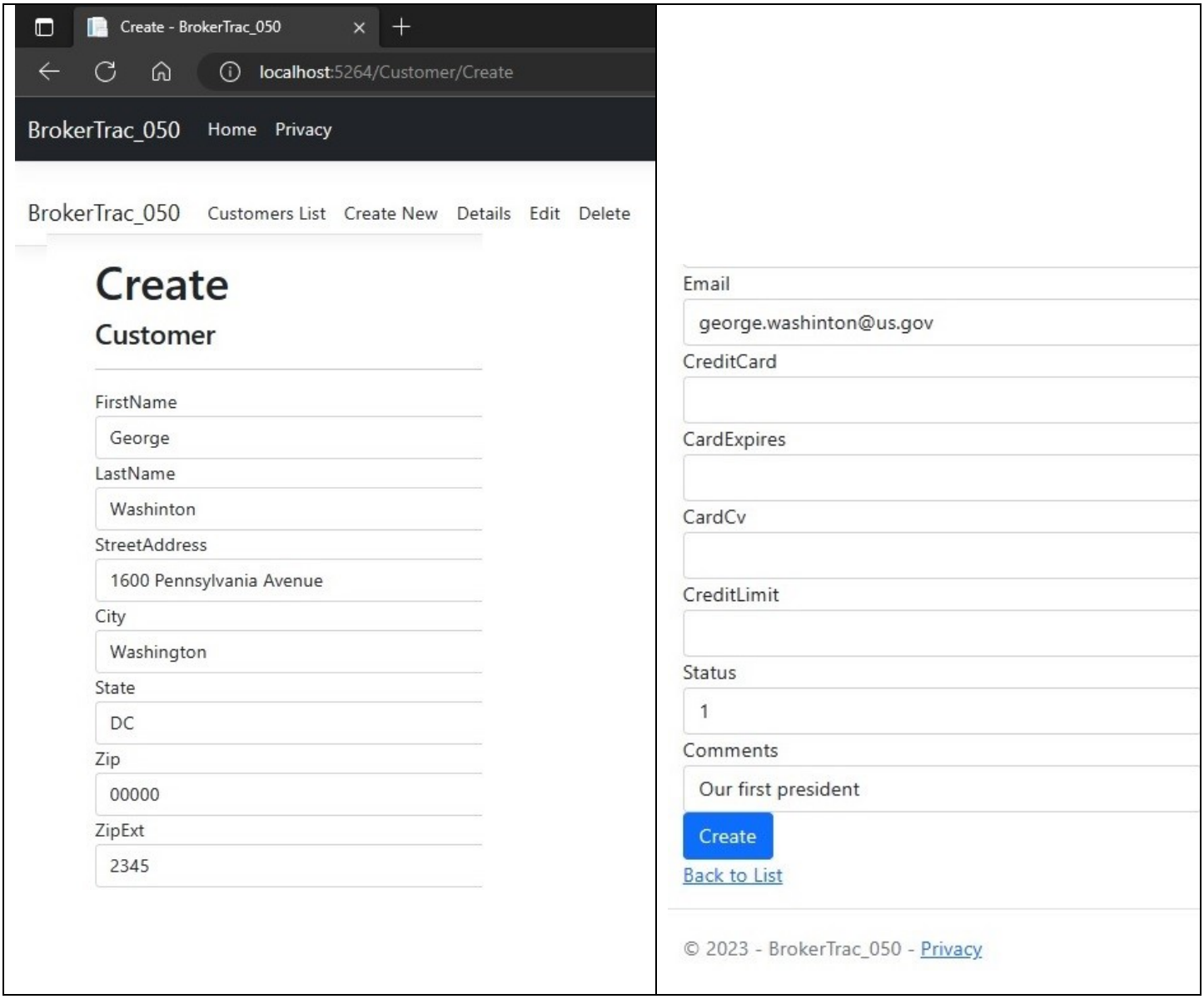
This is when we say "**Yes**!"

The Row headers could be shortened a little, and the Comments might not need to be on this list – or at least not showing the really long text that's in there – and we can even control which columns should even be shown on this listing – but we can adjust that later.

Now let's inspect the other pages. Both navigating from this listing – and the Nav Bar links:

The Create Customer page:

The screen is kind of long, so split into two sections:



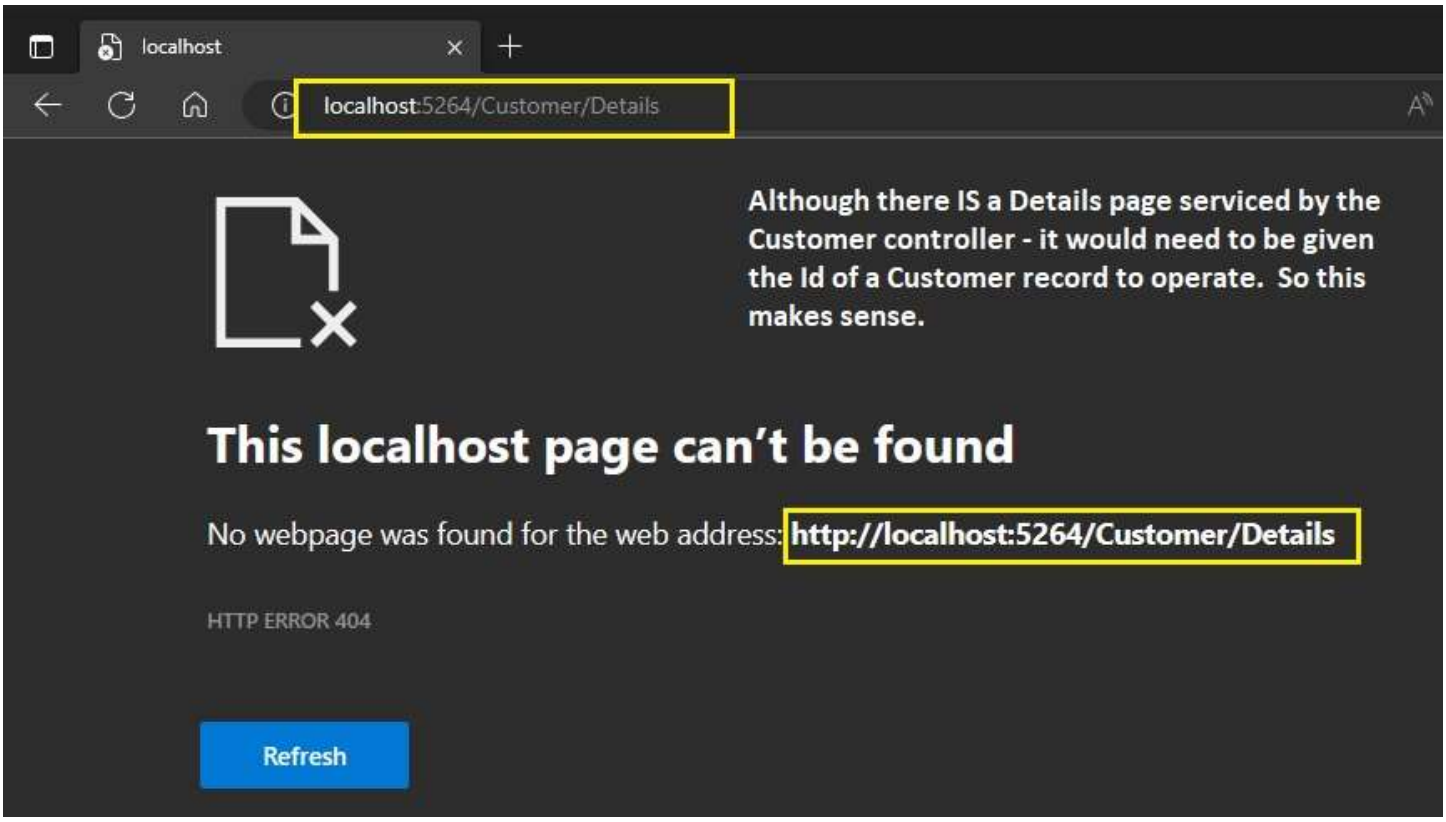
After hitting the Create button, it inserted a row to the table, requeries the table, and shows the list again.

John	Brown	3827 Firestone Blvd	Los Angeles	CA	90310	2222	2135555432	jbrown@somecompany.com			5000.00	1		Edit | Details | Delete
Andy	Bodie	543 Mockingbird Ave	Springfield	MI	43542	0001	4135551534	anybody@somecompany.com	4455455666667777	12/28	555	2	No Credit for Andy Bodie	Edit | Details | Delete
George	Washinton	1600 Pennsylvania Avenue	Washington	DC	00000	2345		george.washinton@us.gov				1	Our first president	Edit | Details | Delete

George Washinton is the last entry in the Index listing now.

I checked the "Edit" link from the Customer Listing – and it works – because it knows the ID of the record you want to edit. So does "Details" and "Delete"

But when we click Edit, Details or Delete from the Nav Bar – we get this error – because we have not yet selected a record, and it doesn't know which one we want to do the operation on. So maybe having those navigations from there doesn't make sense.



But at least we can do the basic operations on the Customer table now:

C Create – Insert new Customers – works

R Read – Show a list – and show the Details on a Customer – works

U Update - Modify an existing Customer – works

D Delete – Remove a record from the Customers table - works

After this – we would create a similar Controller and View pages for Orders.

But two tables would not really make much of an application. At least a few others would be necessary. Probably tables like: Products, Order Detail Lines (child records of an Order, showing what Products are being ordered.) Then things like: Shipping or Sales Completion information. Possibly Vendors, who we purchase things from. Materials we order from Vendors. If we were a manufacturer, we might need a Bill of Materials table to define what Materials we buy to produce the Products. And then maybe other tables that keep track of our Inventory on hand, or what we're planning on manufacturing, etc.

But those will have to be done with a bit more planning.

Also – the layout of the page, as it is; might not be how we would want an actual application to look like.

This is just a "cookie-cutter" template. But as you can see, Visual Studio provided us a great set of tools to get started. This is one of the wonderful set of tools offered by Microsoft to help people get started with:

ASP Active Server Pages – Web pages that can interact dynamically with a database

.NET The dot Net framework that makes languages such as C# available for our middle-tier

MVC The Model (database) View (web pages) Controller (C# code to coordinate the two)

Entity Framework The tooling that allows code generation to create workable code.

Remember – it wasn't too long ago when developers had to write all of this from scratch!

In the next module, we will talk about Code First, and Migrations

This will be where you don't have an existing database – but you construct it from Visual Studio.

Then, as you add tables and objects to the database, you incrementally add and modify the database through packages called Migrations – which get applied to the database, to keep it in synch with the development of classes being done in Visual Studio. It's kind of the opposite of "Database First", with scaffolding that we just did.

dotnet ef migrations add InitialMigration
dotnet ef migrations remove
dotnet ef database update

https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli&WT.mc_id=AZ-MVP-5003535

Thanks for joining me in this look at APS NET MVC with Entity Framework – Database First

Migrations after doing a Database First scaffold:

There is some mention online that you can't go from "Database First" and then to "Migrations" so easily.

Apparently if you're going to control the structure of the database from Visual Studio (Code First, then Migrations) – the chain of changes must be complete and contiguous

This Learn Microsoft pages discusses customizing Migrations – in EF 6

Customizing the migrations history table - EF6 | Microsoft Learn

This has the answer though:

entity framework core - In EFCore, how do I scaffold my context and then start doing migrations? - Stack Overflow

Code First Migrations uses a snapshot of the model stored in the **most recent migration** to detect changes to the model (you can find detailed information about this in Code First Migrations in Team Environments).

Run
`Add-Migration InitialCreate –IgnoreChanges`
to create the initial migration from an existing database. Then
`update-database`
to simply add the migration to the _EFMigrationsHistory table.

So YES, you can begin doing Migrations from Visual Studio after you've done a Database First scaffold. You just have to tell it to Ignore Changes that have taken place before the InitialCreate of the first migration.

Appendix A: The initial BrokerTrac050 Database

The current installation is a SQL Server 2022 instance.

Host (name of my PC) is: Frodo
The Named Instance is: The_Shire

The database will be set up to use Windows Authentication, and to trust the local connection. This is important if we don't want to obtain a trusted certificate, which for local development is a bit of overkill. We will not be encrypting anything, nor be using https protocol. Security is not a concern of this instruction set.

I have my database services set up to be launched manually. That is; none of the SQL Server processes launch when I start up the machine. I go to the SQL Server Configuration Manager, and "Start" the SQL Browser service, and the SQL Server engine whenever I wish to do database work. Make sure both of those are running, and that SQL Server Management Studio is available. These queries will be run from there.

Create the Database:

Note, you can do this in SQL Server Management Studio using the SSMS user interface as well.
If you do that, the FILENAME sections will be placed in your default locations you specified when you installed SQL Server. The LOCATIONs below are specific to my machine, as this is where I keep mdf and ldf files. You can specify wherever on your hard drives you wish them to be.

```
USE [master]
GO

CREATE DATABASE [BrokerTrac050]
 ON  PRIMARY ( NAME = N'BrokerTrac050',     FILENAME = N'D:\SQLServerData\Data\BrokerTrac050.mdf' )
 LOG ON      ( NAME = N'BrokerTrac050_log', FILENAME = N'D:\SQLServerData\Log\BrokerTrac050_log.ldf' )
GO

ALTER DATABASE [BrokerTrac050] SET RECOVERY SIMPLE
GO

USE [BrokerTrac050]
GO
```

Create several Tables:

```
USE [BrokerTrac050]
GO


CREATE TABLE [dbo].[Customers] (
       [Id]                [bigint]            PRIMARY KEY CLUSTERED IDENTITY,
       [FirstName]         [varchar](50)       NULL,
       [LastName]          [varchar](50)       NULL,
       [StreetAddress]     [varchar](50)       NULL,
       [City]              [varchar](50)       NULL,
       [State]             [char](2)           NULL,
       [Zip]               [char](5)           NULL,
       [ZipExt]            [char](4)           NULL,
       [Phone]             [char](10)          NULL,
       [Email]             [varchar](30)       NULL,
       [CreditCard]        [char](16)          NULL,
       [CardExpires]       [char](5)           NULL,
       [CardCV]            [char](3)           NULL,
       [CreditLimit]       [money]             NULL,
       [Status]            [int]               NULL,
       [Comments]          [text]              NULL  );

GO


CREATE TABLE [dbo].[Orders](
       [Id]                [bigint]            PRIMARY KEY CLUSTERED IDENTITY,
       [CustomerId]        [bigint]            NOT NULL,
       [OrderDate]         [datetime]          NOT NULL,
       [ShippedDate]       [datetime]          NULL,
       [Salesman]          [varchar](50)       NULL,
       [Store]             [varchar](50)       NULL,
       [Status]            [int]               NOT NULL,
       [Comments]          [text]              NULL  );
GO
```

Create Foreign Key Constraints:

Now let's populate some records:

This is optional, and does not affect scaffolding the database to Entity Framework. I just like to have sample records in my Customer table first, so that when we create the Customers List page, something will populate into it. Letting me know it's working.

```
USE [BrokerTrac050]
GO

SET NOCOUNT ON
GO

INSERT INTO dbo.Customers (FirstName, LastName, StreetAddress, City, State, Zip, ZipExt,
       Phone, Email, CreditCard, CardExpires, CardCV, CreditLimit, Status, Comments)
VALUES ('Jim', 'Smith', '123 Main Street', 'Albany', 'NY', '12345', '9876', '2055551234',
       'jimsmith@somecompany.com', '4444555566667777', '12/28', '123', 10000.00, 1,
       'Our first Customer');

INSERT INTO dbo.Customers (FirstName, LastName, StreetAddress, City, State, Zip, ZipExt,
       Phone, Email, CreditCard, CardExpires, CardCV, CreditLimit, Status, Comments)
VALUES ('Ringo', 'Starr', '432 Abbey Road', 'London', 'UK', '03982', '9999', '0385554321',
       'ringos@beatles.com', '4444552266667700', '19/27', '321', 15000.00, 1,
       'Our first famous customer');

INSERT INTO dbo.Customers (FirstName, LastName, StreetAddress, City, State, Zip, ZipExt,
       Phone, Email, CreditCard, CardExpires, CardCV, CreditLimit, Status, Comments)
VALUES ('BB', 'King', '4321 Old Road', 'Jackson', 'MS', '33333', '1357', '6665557777',
       'bbking@bluesmusic.com', '4777744455556666', '05/26', '012', 9000.00, 1,
       'Our first blues musician. But this is a long text, full of stories and music from the old
days.');

INSERT INTO dbo.Customers (FirstName, LastName, StreetAddress, City, State, Zip, ZipExt,
       Phone, Email, CreditCard, CardExpires, CardCV, CreditLimit, Status, Comments)
VALUES ('John', 'Brown', '3827 Firestone Blvd', 'Los Angeles', 'CA', '90310', '2222', '2135555432',
       'jbrown@somecompany.com', NULL, NULL, NULL, 5000.00, 1,
       NULL);

INSERT INTO dbo.Customers (FirstName, LastName, StreetAddress, City, State, Zip, ZipExt,
       Phone, Email, CreditCard, CardExpires, CardCV, CreditLimit, Status, Comments)
VALUES ('Andy', 'Bodie', '543 Mockingbird Ave', 'Springfield', 'MI', '43542', '0001', '4135551534',
       'anybody@somecompany.com', '4455455666667777', '12/28', '555', NULL, 2,
       'No Credit for Andy Bodie');

GO
```

These steps can be run to set up the database either before strarting this walk through, or after building and running the project off the template. You'll want to have this database set up before starting step 14, when we will create class structures for Entity Framework via "scaffolding" the database – i.e. a Database-First migration.

Appendix B: The dotnet commands:

C:\Users\rgbel> **dotnet –help**

Usage: dotnet [runtime-options] [path-to-application] [arguments]

Execute a .NET application.

runtime-options:

--additionalprobingpath <path>	Path containing probing policy and assemblies to probe for.
--additional-deps <path>	Path to additional deps.json file.
--depsfile	Path to <application>.deps.json file.
--fx-version <version>	Version of the installed Shared Framework to use to run the application.
--roll-forward <setting>	Roll forward to framework version (LatestPatch, Minor, LatestMinor, Major, LatestMajor, Disable).
--runtimeconfig	Path to <application>.runtimeconfig.json file.

path-to-application:
The path to an application .dll file to execute.

Usage: dotnet [sdk-options] [command] [command-options] [arguments]

Execute a .NET SDK command.

sdk-options:

-d|--diagnostics	Enable diagnostic output.
-h|--help	Show command line help.
--info	Display .NET information.
--list-runtimes	Display the installed runtimes.
--list-sdks	Display the installed SDKs.
--version	Display .NET SDK version in use.

SDK commands:

add	Add a package or reference to a .NET project.
build	Build a .NET project.
build-server	Interact with servers started by a build.
clean	Clean build outputs of a .NET project.
format	Apply style preferences to a project or solution.
help	Show command line help.
list	List project references of a .NET project.
msbuild	Run Microsoft Build Engine (MSBuild) commands.
new	Create a new .NET project or file.
nuget	Provides additional NuGet commands.
pack	Create a NuGet package.
publish	Publish a .NET project for deployment.

SDK commands (continued):

remove	Remove a package or reference from a .NET project.
restore	Restore dependencies specified in a .NET project.
run	Build and run a .NET project output.
sdk	Manage .NET SDK installation.
sln	Modify Visual Studio solution files.
store	Store the specified assemblies in the runtime package store.
test	Run unit tests using the test runner specified in a .NET project.
tool	Install or manage tools that extend the .NET experience.
vstest	Run Microsoft Test Engine (VSTest) commands.
workload	Manage optional workloads.

Additional commands from bundled tools:

dev-certs	Create and manage development certificates.
fsi	Start F# Interactive / execute F# scripts.
user-jwts	Manage JSON Web Tokens in development.
user-secrets	Manage development user secrets.
watch	Start a file watcher that runs a command when files change.

Appendix C: All the Switches on PowerShell "scaffolding" command:

```
ef dbcontext scaffold = "Data Source=The connectionstring;"
 Microsoft.EntityFrameworkCore.SqlServer  = The provider (could be Oracle or some other db type)
   --use-database-names                   = I want entity names same / similar to db - object names
   --output-dir  Models                   = output directory in my project for entities to be Models
   --context-dir Data                     = output directory in my project for context to be Data
   --no-pluralize                         = I want entity names same / similar to db - object names
   --no-onconfiguring                     = no OnCongiguring method with connect string
   --force                                = will override any existing model
```

Appendix D: Switches on the ASP NET Code Generator

To update to most recent version: dotnet tool update -g dotnet-aspnet-codegenerator

Usage: aspnet-codegenerator [generator] [options]

Generators:

`area`	: Generates an MVC Area.
`Controller`	: Generates a controller.
`identity`	: Generates identity files.
`Minimalapi`	: Generates an endpoints file (with CRUD API endpoints) given a model and optional DbContext.
`razorpage`	: Generates RazorPage(s).
`view`	: Generates a view.

Options:

`-p or --project`	Path to .csproj file in the project.
`-n or --nuget-package-dir`	
`-c or --configuration`	Configuration for the project (Debug/ Release)
`-tfm or --target-framework`	Target Framework to use. (Short name eg. net46)
`-b or --build-base-path`	
`--no-build`	

Options for Controllers and views	**Description**
`--model or -m`	Model class to use.
`--dataContext or -dc`	The `DbContext` class to use or the name of the class to generate.
`--bootstrapVersion or -b`	Specifies the bootstrap version. Valid values are `3` or `4`. Default is `4`. If needed and not present, a *wwwroot* directory is created that includes the bootstrap files of the specified version.
`--referenceScriptLibraries or -scripts`	Reference script libraries in the generated views. Adds `_ValidationScriptsPartial` to Edit and Create pages.
`--layout or -l`	Custom Layout page to use.
`--useDefaultLayout or -udl`	Use the default layout for the views.
`--force or -f`	Overwrite existing files.
`--relativeFolderPath or -outDir`	Specify the relative output folder path from project where the file needs to be generated, if not specified, file will be generated in the project folder
`--useSqlite or -sqlite`	Flag to specify if `DbContext` should use SQLite instead of SQL Server.

Option just for Views	**Description**
`--controllerName or -name`	Name of the controller.
`--useAsyncActions or -async`	Generate async controller actions.
`--noViews or -nv`	Generate **no** views.
`--restWithNoViews or -api`	Generate a Controller with REST style API. `noViews` is assumed and any view related options are ignored.
`--readWriteActions or -actions`	Generate controller with read/write actions without a model.

<END OF DOCUMENT>