

# Getting Started with the Tabular Object Model (TOM)

The purpose of this hands-on tutorial is to provide campers with a guide how to get started using the Tabular Object Model (TOM) to program against data models created using Power BI Desktop. This lab will get you started with Visual Studio Code and the latest version of .NET known as .NET 5 that was released in November of 2020.

There lab exercises were derived from an awesome set of blog posts by Phil Seamark (see more: [post1](#), [post2](#), [post3](#) and [post4](#)).

## Setup: Install Visual Studio Code and the .NET 5 SDK

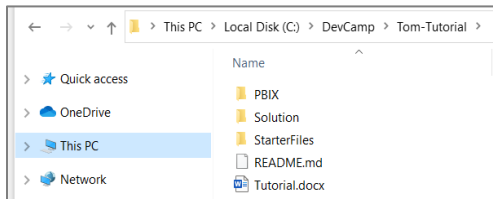
In this initial setup, you will download install the .NET 5 SDK and Visual Studio Code if you haven't done so already.

1. Install the .NET 5 SDK.
  - a) Download and run the installer: <https://dotnet.microsoft.com/download/dotnet/thank-you/sdk-5.0.100-windows-x64-installer>
  - b) Here is more info on .NET 5 downloads if you need it: <https://dotnet.microsoft.com/download>
2. Install Visual Studio Code
  - a) Open a browser and navigate to [code.visualstudio.com](https://code.visualstudio.com)
  - b) download and run the installer for the current version for Windows.
  - c) Use default settings when prompted during the install.

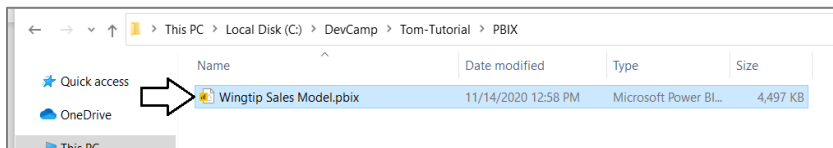
## Exercise 1: Connect to Power BI Desktop using the Tabular Object Model

In this exercise, you will create a new .NET console application and connect to a data model running in Power BI Desktop.

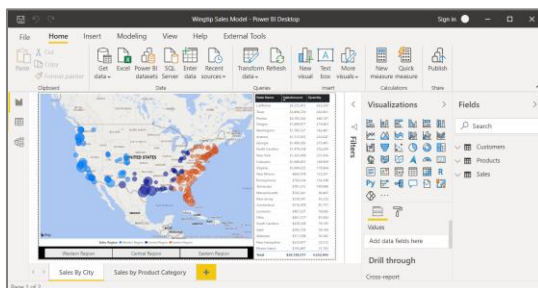
1. Create a new folder for the tutorial.
  - a) Create a new folder on your local hard drive named **Tom-Tutorial**.
  - b) Download student ZIP from this URL: <https://github.com/PowerBIDevCamp/Tabular-Object-Model-Tutorial/archive/main.zip>.
  - c) Extract the contents of the downloaded ZIP archive into the **Tom-Tutorial** folder.



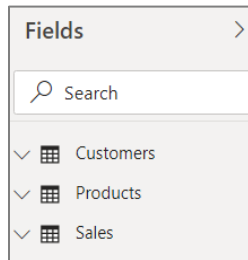
2. Launch Power BI Desktop project and open the project named **Wingtip Sales Model.pbix**.
  - a) Inside the **PBIX** folder, locate the project file named **Wingtip Sales Model.pbix** and open it in Power BI Desktop



- b) You should now see report and data model for **Wingtip Sales Model.pbix** loaded in Power BI Desktop.

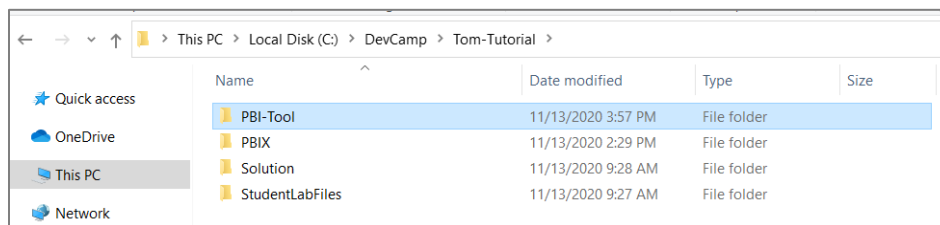


- c) Quickly review the four tables named **Customers**, **Products** and **Sales** that are visible in the **Fields** list

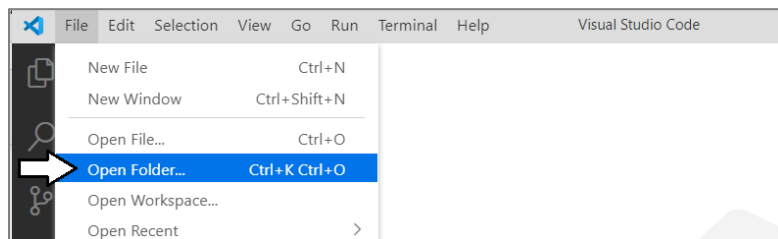


Leave this project open in Power BI desktop as you move on to the next step. Later in this exercises, you will connect to the data model of the **Wingtip Sales Model** project using the Tabular Object Model by connecting through a localhost address.

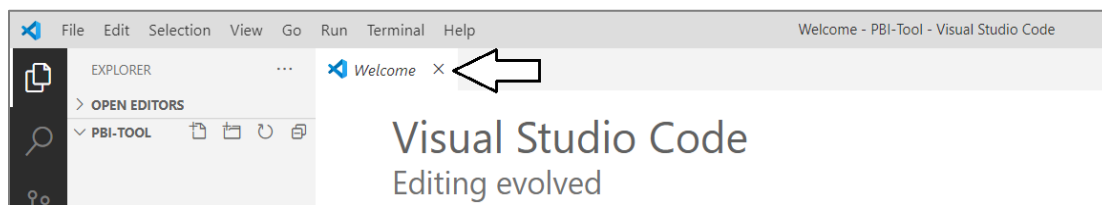
3. Create a new folder for your new project and name it **PBI-Tool**.
- a) Create a child inside the **Tom-Tutorial** folder named **PBI-Tool**.



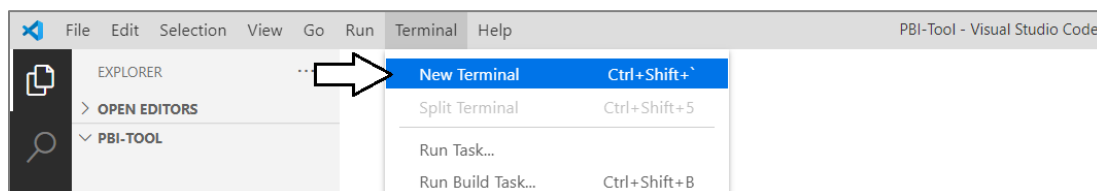
4. Launch Visual Studio Code and open the **PBI-Tool** folder.
- a) Launch Visual Studio Code and use the **Open Folder...** command to open the **PBI-Tool** folder created in the previous step.



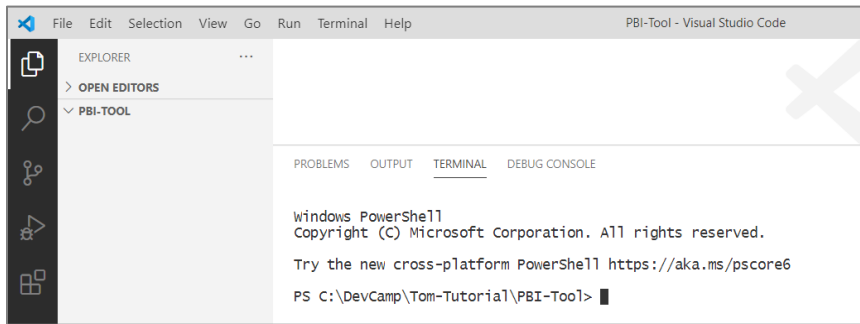
- b) Once you have open the **PBI-Tool** folder, close the **Welcome** page.



5. Use the Terminal console to verify the current version of .NET
- a) Use the **Terminal > New Terminal** command or the **[Ctrl+Shift+`]** keyboard shortcut to open the Terminal console.



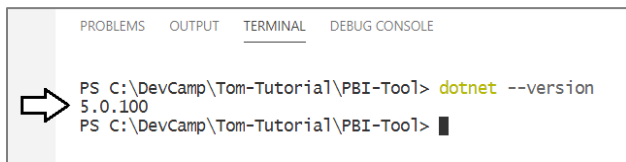
- b) You should now see a Terminal console with a cursor where you can type and execute command-line instructions.



- c) Type the following **dotnet** command-line instruction into the console and press **Enter** to execute it.

```
dotnet --version
```

- d) When you run the command, the **dotnet** CLI should respond by display the .NET version number.

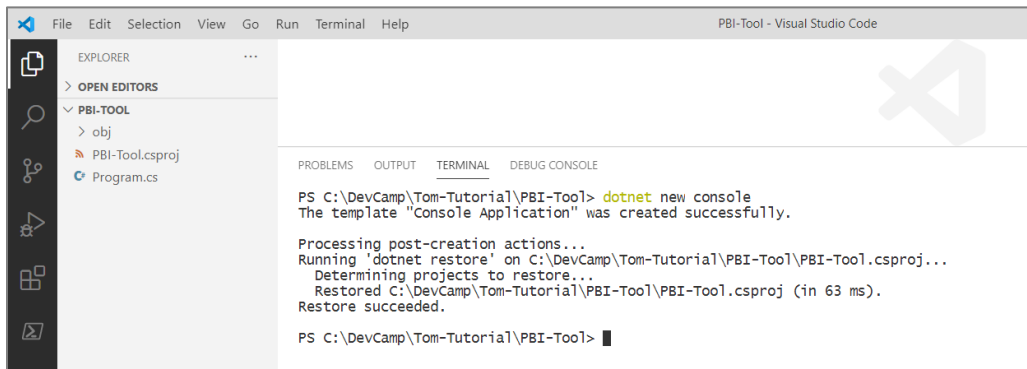


6. Create a new .NET console application in the PBI-Tool folder using the **dotnet new** command.

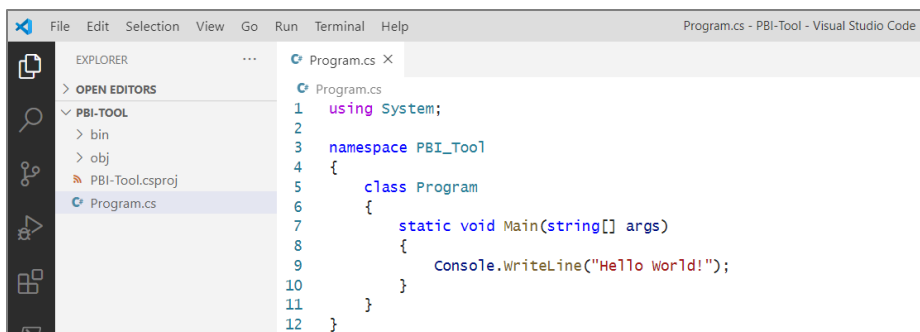
- a) In the Terminal console, type and execute the following command to generate a new .NET console application.

```
dotnet new console
```

- b) After running the **dotnet new console** command, you should see new files have been added to the project.

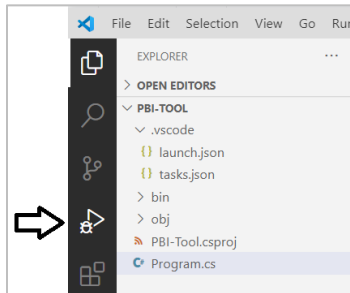


- c) Open the **Program.cs** file and inspect the C# starter code inside which displays the tradition **Hello World!** greeting.

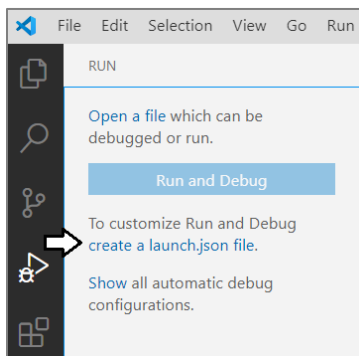


## 7. Add support for running the new console project in the .NET debugger.

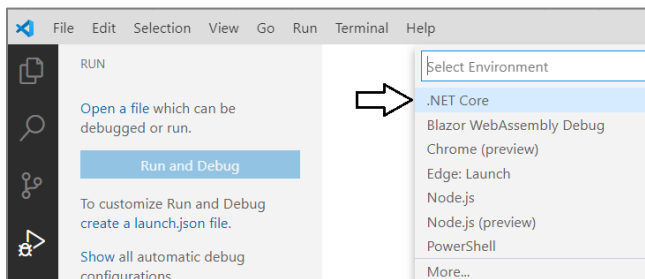
- a) Using the left navigation in Visual Studio Code, click the arrow icon with the bug to move to **Run and Debug** view.



- b) Click the **create a launch.json file** link.

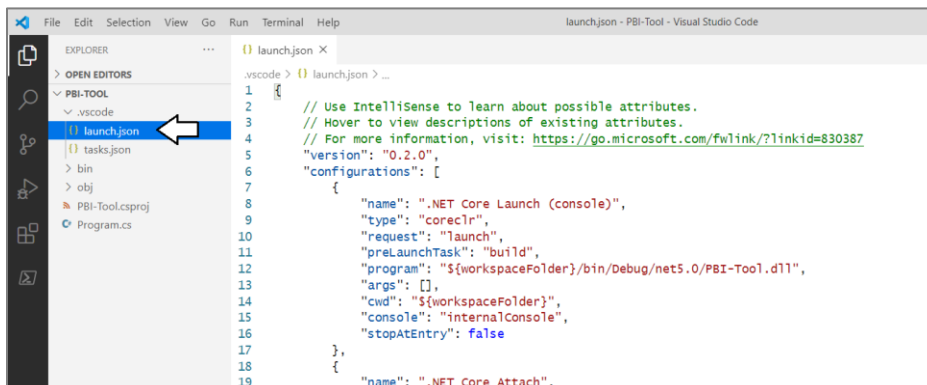


- c) When prompted to **Select Environment**, select **.NET Core**.



When you select an environment, Visual Studio Code responds by generating two files named **launch.json** and **tasks.json**.

- d) Examine the contents of the **launch.json** file.



- e) Inside **launch.json**, locate the following line of code.

```
"stopAtEntry": false
```

- f) Add a comma at the end of this line and then add another line of code below as shown in the following listing.

```
"stopAtEntry": false ,
"logging": { "moduleLoad": false }
```

- g) The **launch.json** file in your project should now match the following screenshot.

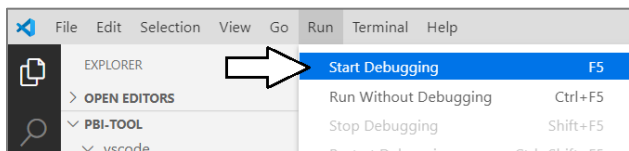
```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": ".NET Core Launch (console)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "build",
      "program": "${workspaceFolder}/bin/Debug/net5.0/Exercise01.dll",
      "args": [],
      "cwd": "${workspaceFolder}",
      "console": "internalConsole",
      "stopAtEntry": false,
      "logging": { "moduleLoad": false }
    },
    {
      "name": ".NET Core Attach",
```

- h) Save your changes and close **launch.json**.

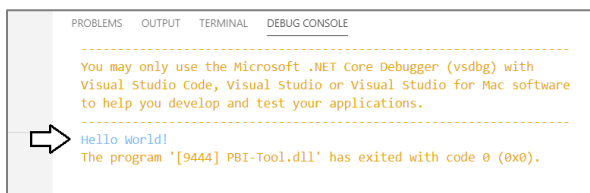
The reason for disabling **moduleLoad** is that it will eliminate unnecessary messages shown in the console during debugging sessions.

8. Run a debugging session to test the application and see the Hello World! Message.

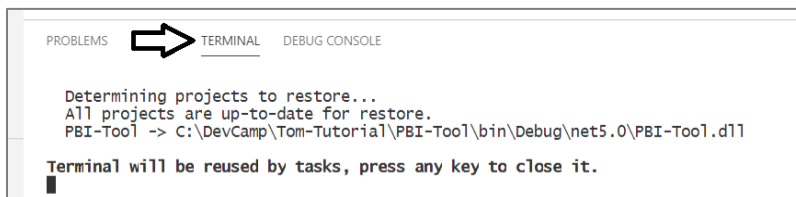
- a) Run a debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.



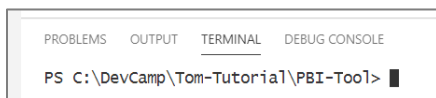
- b) The application should run and display **Hello World!** In the debug console window as shown in the following screenshot.



- c) Navigate to the **TERMINAL** console and press **ENTER** to stop the debugger.



- d) You have now run and completed a simple debugging session.



OK, you've now gotten 'Hello World' out of the way. Remember that a journey of a thousand miles begins with a single step!

9. Add the AMO NuGet package to your project using the so you can program against the Tabular Object Model.
  - a) Return to the Terminal console.
  - b) Type and execute the following **dotnet add package** command to add the **AMO** NuGet package

```
dotnet add package Microsoft.AnalysisServices.NetCore.retail.amd64 --version 19.12.7.2-Preview
```

At the time that lab exercise was written, the latest version of this package was **19.12.7.2-Preview**. You can check out the information at the following URL to determine the latest release: <https://www.nuget.org/packages/Microsoft.AnalysisServices.NetCore.retail.amd64>

- c) Once you have typed the following command into the **Terminal** console, press **Enter** to execute it.

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\DevCamp\Tom-Tutorial\PBI-Tool> dotnet add package Microsoft.AnalysisServices.NetCore.retail.amd64 --version 19.12.7.2-Preview

```

- d) The **dotnet add package** command should run without any errors.

```

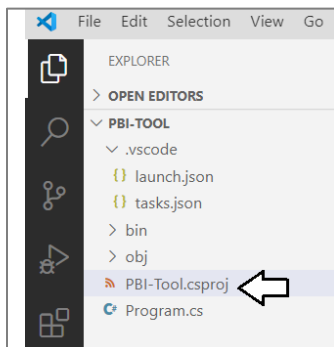
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  2: PowerShell Integrat... + [ ] [ ] ^

PS C:\DevCamp\Tom-Tutorial\PBI-Tool> dotnet add package Microsoft.AnalysisServices.NetCore.retail.amd64 --version 19.12.7.2-Preview

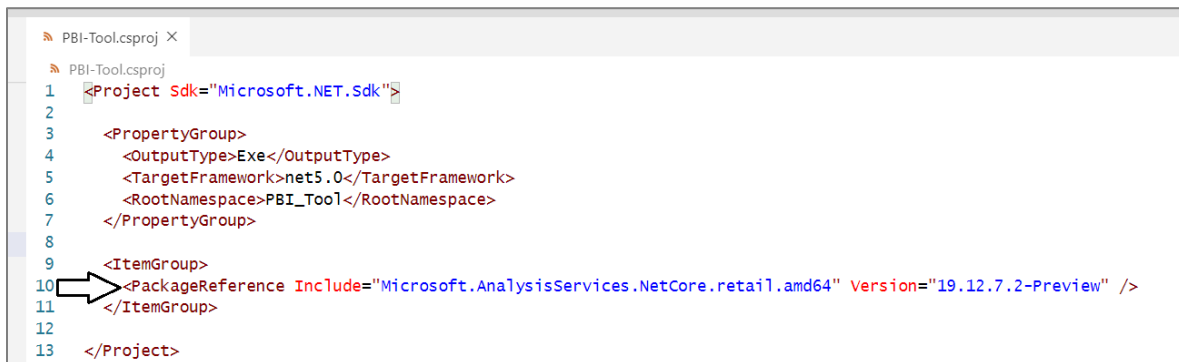
Determining projects to restore...
Writing C:\Users\Ted\AppData\Local\Temp\tmp4130.tmp
info : Adding PackageReference for package 'Microsoft.AnalysisServices.NetCore.retail.amd64' into project 'C:\DevCamp\Tom-Tutorial\PBI-Tool\PBI-Tool.csproj'.
info : Restoring packages for C:\DevCamp\Tom-Tutorial\PBI-Tool\PBI-Tool.csproj...
info : Package 'Microsoft.AnalysisServices.NetCore.retail.amd64' is compatible with all the specified frameworks in project 'C:\DevCamp\Tom-Tutorial\PBI-Tool\PBI-Tool.csproj'.
info : PackageReference for package 'Microsoft.AnalysisServices.NetCore.retail.amd64' version '19.12.7.2-Preview' added to file 'C:\DevCamp\Tom-Tutorial\PBI-Tool\PBI-Tool.csproj'
info : Committing restore...
info : Writing assets file to disk. Path: C:\DevCamp\Tom-Tutorial\PBI-Tool\obj\project.assets.json
log  : Restored C:\DevCamp\Tom-Tutorial\PBI-Tool\PBI-Tool.csproj (in 254 ms).
PS C:\DevCamp\Tom-Tutorial\PBI-Tool>

```

- e) After the **dotnet add package** command has completed, open the project file named **PBI-Tool.csproj**.



- f) You should see that the **PBI-Tool.csproj** file now contains a **PackageReference** element to track the newly installed package.



- g) Close **PBI-Tool.csproj** without saving any changes.

Now that you have added this NuGet package, you can program against the AMO library which includes the Tabular Object Model.

10. Modify the C# code in **Program.cs**.

- Open **Program.cs** and delete all the existing code inside.
- Copy and paste the following code in **Program.cs**.

```
using System;
using Microsoft.AnalysisServices.Tabular;

class Program {

    const string connectionString = "localhost:50000"; // update with port number on your machine

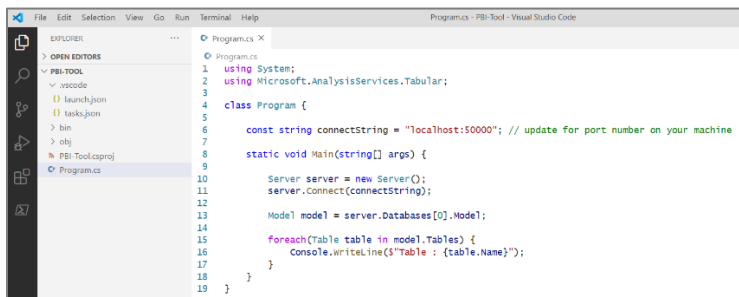
    static void Main(string[] args) {

        Server server = new Server();
        server.Connect(connectionString);

        Model model = server.Databases[0].Model;

        foreach(Table table in model.Tables) {
            Console.WriteLine($"Table : {table.Name}");
        }
    }
}
```

- The **Program.cs** file in your project should now match the following screenshot.



There is one more important thing you need to do before you can run this program. You must determine the port number that Power BI Desktop is using to run the project you opened named **Wingtip Sales Model.pbix**. The issue is that Power BI Desktop selects a random port number whenever you open a PBIX project file. Once you determine what the port number is on your PC, you can use that port number to modify the **connectString** which will allow you to connect to the data model of **Wingtip Sales Model.pbix**.

## 11. Determine the port number used by Power BI Desktop using command-line instructions.

- Return the Terminal console. Type in and execute the following **TASKLIST** command.

```
TASKLIST /FI "imagename eq msmdsrv.exe" /FI "sessionname eq console"
```

- When you run the **TASKLIST** command, it displays its output in a table format

Image Name	PID	Session Name	Session#	Mem Usage
msmdsrv.exe	41612	Console	1	516,492 K

- Determine the PID (i.e. Process ID) for the image named **msmdsrv.exe**.

Image Name	PID	Session Name	Session#	Mem Usage
msmdsrv.exe	41612	Console	1	516,492 K

You're half way there at this point. Now that you have found the process ID (PID), you will use that to find the port number.

- d) Type in the following **netstat** command and replace the PID in quotes with the one you discovered on your computer.

```
netstat /ano | findstr "41612"
```

- e) Execute the **netstat** command and examine its output.

```
PS C:\DevCamp\Tom-Tutorial\PBI-Tool> TASKLIST /FI "imagename eq msmdsrv.exe" /FI "sessionname eq console"

Image Name                PID Session Name        Session#    Mem Usage
=====
msmdsrv.exe               41612 Console                1         516,492 K
PS C:\DevCamp\Tom-Tutorial\PBI-Tool>

PS C:\DevCamp\Tom-Tutorial\PBI-Tool> netstat /ano | findstr "41612"

TCP    127.0.0.1:50468        0.0.0.0:0            LISTENING   41612
TCP    [::1]:50468           [::]:0               LISTENING   41612
TCP    [::1]:50468           [::1]:50469          ESTABLISHED 41612
```

- f) You should be able to discover the port being used by Power BI Desktop as shown in the following screenshot.

```
PS C:\DevCamp\Tom-Tutorial\PBI-Tool> netstat /ano | findstr "41612"

TCP    127.0.0.1:50468        0.0.0.0:0            LISTENING   41612
TCP    [::1]:50468           [::]:0               LISTENING   41612
TCP    [::1]:50468           [::1]:50469          ESTABLISHED 41612
```

Using the **TASKLIST** command and the **netstat** command is just one way to determine the Power BI Desktop port number. You can also find the current port number using tools such as DAX Studio and the Tabular Editor. Check out [this blog post](#) to learn more.

## 12. Run the program and connect to Power BI Desktop.

- a) Open **Program.cs** and update the **connectString** constant with the correct port number for your computer.

```
Program.cs
1  using System;
2  using Microsoft.AnalysisServices.Tabular;
3
4  class Program {
5
6      const string connectionString = "localhost:50468";
7
8      static void Main(string[] args) {
9
10         Server server = new Server();
11         server.Connect(connectionString);
12
13         Model model = server.Databases[0].Model;
14
15         foreach (Table table in model.Tables) {
16             Console.WriteLine($"Table : {table.Name}");
17         }
18     }
19 }
20 }
```

- b) Run a debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.
- c) When the program runs, it enumerates through the tables in the data model and displays their names in the Debug Console

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

-----
You may only use the Microsoft .NET Core Debugger (vsdbg) with
Visual Studio Code, Visual Studio or Visual Studio for Mac software
to help you develop and test your applications.
-----
Table : Customers
Table : Sales
Table : Products
The program '[38208] PBI-Tool.dll' has exited with code 0 (0x0).
```

Congratulations. You can now say you that have programmed using the **Tabular Object Model**. Your friends will be so jealous!



13. Write code to add a new measure to the **Sales** table.

- a) In **Program.cs**, comment out the lines of code for the **foreach** loop and place the cursor below to add more code.

```
static void Main(string[] args) {
    Server server = new Server();
    server.Connect(connectString);

    Model model = server.Databases[0].Model;

    // foreach(Table table in model.Tables) {
    //     Console.WriteLine($"Table : {table.Name}");
    // }
    ➡
```

- b) Add the following code at the end of the **Main** function.

```
Table table = model.Tables["Sales"];

if (table.Measures.ContainsName("VS Code Measure")) {
    Measure measure = table.Measures["VS Code Measure"];
    measure.Expression = "\"Hello Again World\"";
}
else {
    Measure measure = new Measure() {
        Name = "VS Code Measure",
        Expression = "\"Hello World\""
    };
    table.Measures.Add(measure);
}

model.SaveChanges();
```

The first time you run this code, it will add a new measure to the **Sales** table named **VS Code Measure**. The second time you run this code, it will determine that a measure named **VS Code Measure** already exists and it will modify the measure's expression. The main point of this code is to demonstrate that you can use TOM to create new measures and to modify existing measures.

- c) The code at the bottom of the **Main** function in **Program.cs** should now match the following screenshot.

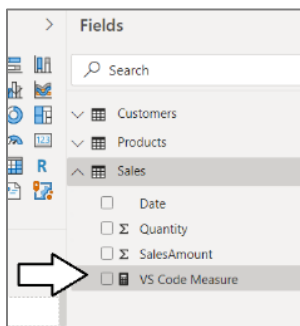
```
// foreach(Table table in model.Tables) {
//     Console.WriteLine($"Table : {table.Name}");
// }

Table table = model.Tables["Sales"];

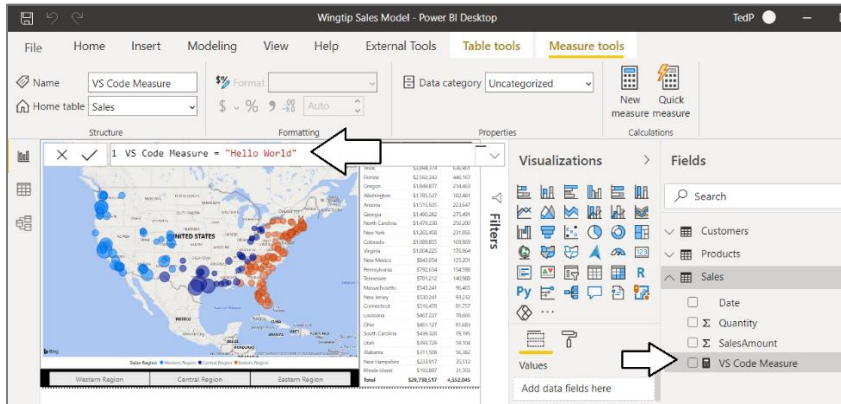
if (table.Measures.ContainsName("VS Code Measure")) {
    Measure measure = table.Measures["VS Code Measure"];
    measure.Expression = "\"Hello Again World\"";
}
else {
    Measure measure = new Measure() {
        Name = "VS Code Measure",
        Expression = "\"Hello World\""
    };
    table.Measures.Add(measure);
}

model.SaveChanges();
```

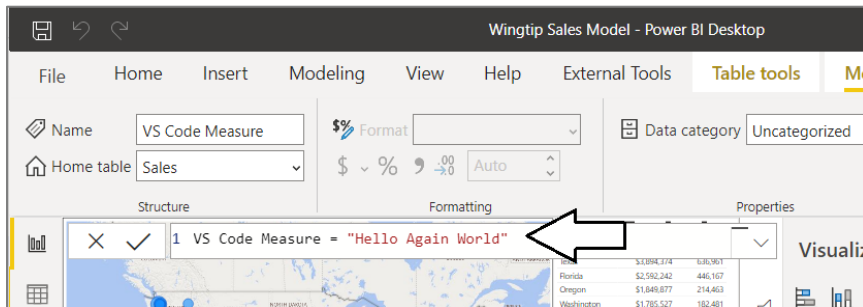
- d) Save your changes to **Program.cs**.  
 e) Run a debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.  
 f) Return to Power BI Desktop. The **Sales** table should contain **VS Code Measure**.



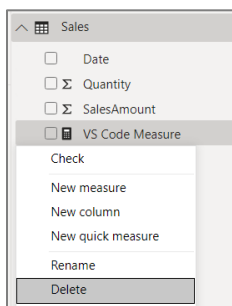
- g) Inspect the expression for **VS Code Measure**. The should expression should be a simple string "Hello World".



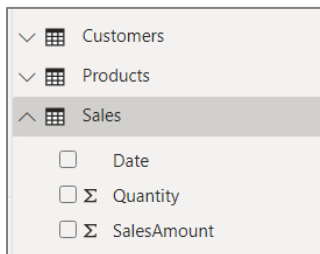
- h) Return to Visual Studio Code.  
 i) Run a second debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.  
 j) Return to Power BI Desktop and verify that **VS Code Measure** now has an updated expression "Hello Again World".



- k) Delete the measure named **VS Code Measure** by right-clicking it in the **Fields** list and select the **Delete** command.



- l) The **Sales** table should no longer contain a measure named **VS Code Measure**.

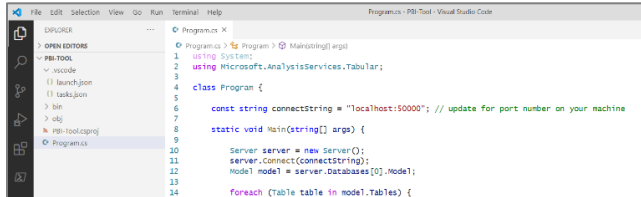


You have used the Tabular Object Model to connect to a Power BI Desktop project and perform some simple read and write operations. Now it's time to move ahead and write code using the Tabular Object Model that performs real-world data modeling tasks.

## Exercise 2: Writing TOM Code to Add Auto Measures to Tables

In this exercise, you will modify the **PBI-Tool** console application to automatically add measures to tables whenever it finds that a table has numeric columns that are not hidden from report view. You will add C# code that iterates through every **Table** object in a data model and then iterates through the **Column** objects for each **Table** object. The code will automatically add a new measure each time it find a numeric column which has an **IsHidden** property value of false.

1. Copy and paste the Exercise 2 starter code into **Program.cs**.
  - a) Open **Program.cs** and delete all the code that is currently inside.
  - b) Copy the Exercise 2 starter code from the [Exercise02-Program.cs](#) file in the **StarterFiles** folder to the Windows clipboard.
  - c) Return to Visual Studio Code and paste in the contents of the Windows clipboard into **Program.cs**.



2. Walk through the code in **Program.cs** to understand what it does.
  - a) The code contains an outer **foreach** loop to iterate through tables and an inner **foreach** loop to iterate through columns.

```
Server server = new Server();
server.Connect(connectString);
Model model = server.Databases[0].Model;

foreach (Table table in model.Tables) {
    foreach (Column column in table.Columns) {
        // iterate through each column of every table
    }
}
```

- b) Inside the inner **foreach** loop, there is an **if** statement to check whether each column is both non-hidden and numeric,

```
// determine if column is visible and numeric
if ((column.IsHidden == false) &
    (column.DataType == DataType.Int64 ||
     column.DataType == DataType.Decimal ||
     column.DataType == DataType.Double)) {

    // add auto measure for each visible, numeric column
}
```

- c) Inside the **if** statement, there is code to generate a new measure any time it finds a non-hidden, numeric column.

```
// add automeasure for this column new measure
string measureName = $"Sum of {column.Name} ({table.Name})";
string expression = $"SUM('{table.Name}'[{column.Name}])";
string displayFolder = "Auto Measures";

Measure measure = new Measure() {
    Name = measureName,
    Expression = expression,
    DisplayFolder = displayFolder
};

measure.Annotations.Add(new Annotation() { Value = "This is an Auto Measure" });

if (!table.Measures.ContainsName(measureName)) {
    table.Measures.Add(measure);
}
else {
    table.Measures[measureName].Expression = expression;
    table.Measures[measureName].DisplayFolder = displayFolder;
}
```

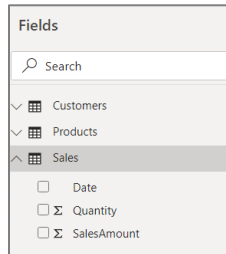
- d) After the outer **foreach** loop, there's a call to **model.SaveChanges** to push the changes to Power BI Desktop.

```
foreach (Table table in model.Tables) {
    foreach (Column column in table.Columns) {
        // code to add auto measures omitted for brevity
    }
}

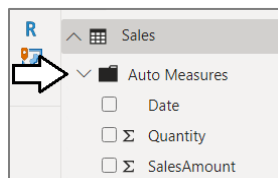
// save changes back to data model in Power BI Desktop
model.SaveChanges();
```

### 3. Run and test the code in **Program.cs**.

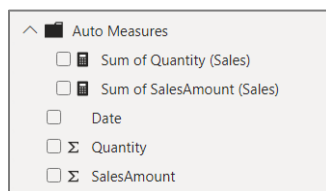
- a) Before running the program, return to Power BI Desktop and verify what fields are in the **Sales** table.



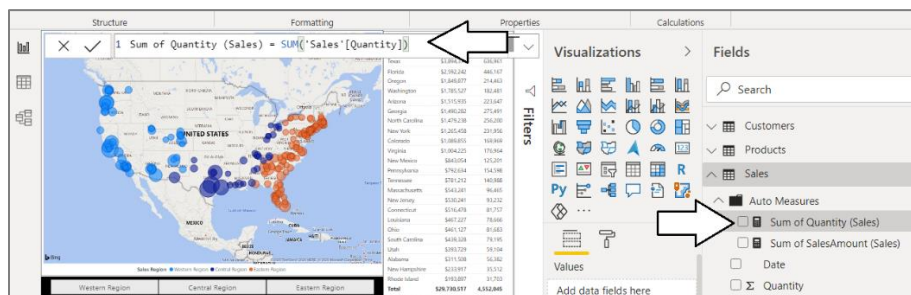
- b) Return to Visual Studio Code.  
 c) Run a debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.  
 d) Wait for the program to complete.  
 e) Return to Power BI Desktop and verify you can see a new display folder named **Auto Measures**.  
 f) Expand the **Auto Measures** folder.



- g) Inside the **Auto Measure** folder, you should see two new measures as shown in the following screenshot.



- h) Examine the DAX expression generated for each of the measures in the **Auto Measures** display folder.



You have now reached the end of Exercise 2. In the next exercise, you'll learn how to execute a DAX query and handle query results.

## Exercise 3 Executing DAX Queries using the AMOMD Client Library

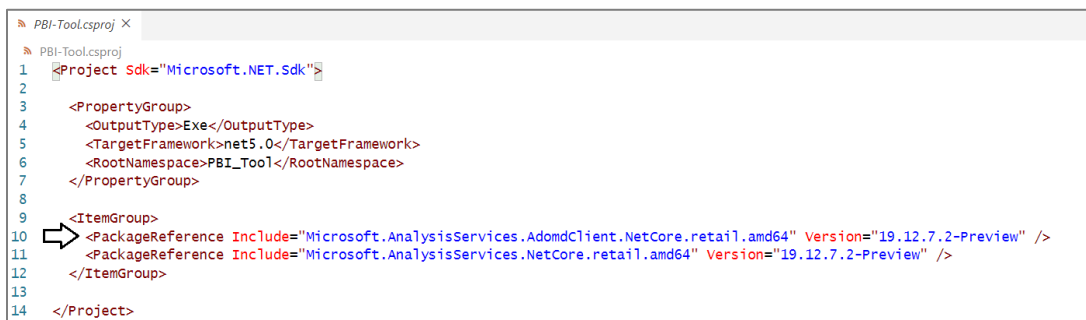
In this exercise, you will learn how to execute DAX queries against a data model in Power BI Desktop using the AMOMD Client library. You will begin by adding a new package for AMOMD. Next, you will add and test C# code that executes DAX queries and converts the query results into a standard CSV file format. In the second part of this exercise, you will mix TOM programming together with AMOMD in order to write code which determines what measures to create by examining the results of a DAX query.

1. Add the AMOMD Client NuGet package to the project so you can program against the library that allows for DAX query execution.
  - a) Return to the Terminal console.
  - b) Type and execute the following **dotnet add package** command to add the **AMOMD Client** NuGet package

```
dotnet add package Microsoft.AnalysisServices.AdomdClient.NetCore.retail.amd64 --version 19.12.7.2-Preview
```

At the time that lab exercise was written, the latest version of this package was **19.12.7.2-Preview**. You can check out the information at the following URL to determine the latest release: <https://www.nuget.org/packages/Microsoft.AnalysisServices.AdomdClient.NetCore.retail.amd64>.

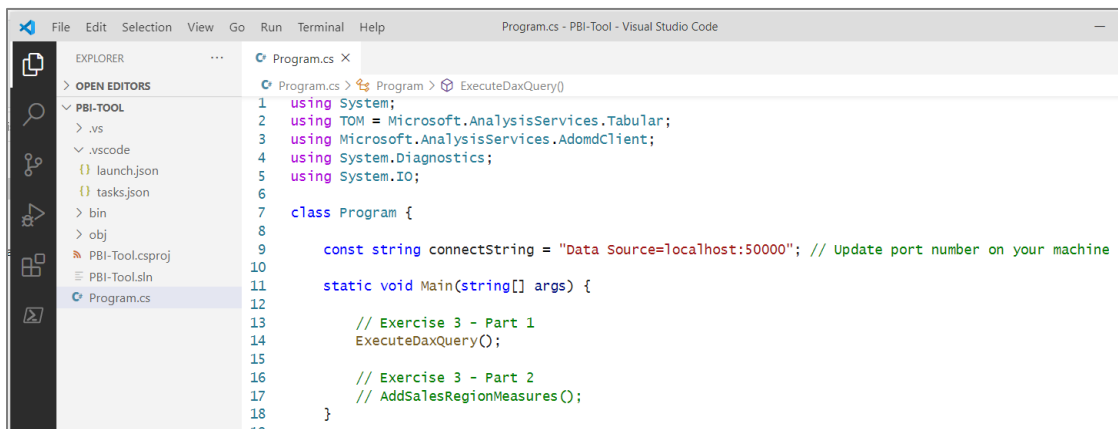
- c) If you look at the **PBI-Tool.csproj** file, you will see there is a new package reference for the AMOMD Client library.



```

1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <PropertyGroup>
4     <OutputType>Exe</OutputType>
5     <TargetFramework>net5.0</TargetFramework>
6     <RootNamespace>PBI_Tool</RootNamespace>
7   </PropertyGroup>
8
9   <ItemGroup>
10    <PackageReference Include="Microsoft.AnalysisServices.AdomdClient.NetCore.retail.amd64" Version="19.12.7.2-Preview" />
11    <PackageReference Include="Microsoft.AnalysisServices.NetCore.retail.amd64" Version="19.12.7.2-Preview" />
12   </ItemGroup>
13
14 </Project>
  
```

- d) Close the **PBI-Tool.csproj** file without saving any changes.
2. Copy and paste the Exercise 3 starter code into **Program.cs**.
  - a) Open **Program.cs** and delete all the code that is currently inside.
  - b) Copy the Exercise 3 starter code from the [Exercise03-Program.cs](#) file in the **StarterFiles** folder to the Windows clipboard.
  - c) Return to Visual Studio Code and paste in the contents of the Windows clipboard into **Program.cs**.



```

1 using System;
2 using TOM = Microsoft.AnalysisServices.Tabular;
3 using Microsoft.AnalysisServices.AdomdClient;
4 using System.Diagnostics;
5 using System.IO;
6
7 class Program {
8
9     const string connectionString = "Data Source=localhost:50000"; // Update port number on your machine
10
11     static void Main(string[] args) {
12
13         // Exercise 3 - Part 1
14         ExecuteDaxQuery();
15
16         // Exercise 3 - Part 2
17         // AddSalesRegionMeasures();
18     }
19 }
  
```

- d) Update the **connectString** constant with the same port number you have used in previous exercises.

```

class Program {
    const string connectionString = "Data Source=localhost:50000"; // Update port number on your machine
}
  
```

- e) Save your changes and close **Program.cs**.

3. Walk through the code in **Program.cs** to understand what it does.

a) Inside the **Main** function, there is a call to **ExecuteDaxQuery**.

```
// Exercise 3 - Part 1
ExecuteDaxQuery();
```

b) **ExecuteDaxQuery** executes a DAX query and then calls **ConvertReaderToCsv** passing an **AdomdDataReader** object.

```
static void ExecuteDaxQuery() {
    // DAX query to be submitted totabuar database engine
    String query = @"
        EVALUATE
            SUMMARIZECOLUMNS(
                //GROUP BY
                Customers[State],

                //FILTER BY
                TREATAS( {""Western Region""} , 'Customers'[Sales Region] ) ,

                // MEASURES
                ""Sales Revenue"" , SUM(Sales[SalesAmount]) ,
                ""Units Sold"" , SUM(Sales[Quantity])
            )
    ";

    AdomdConnection adomdConnection = new AdomdConnection(connectionString);
    adomdConnection.Open();

    AdomdCommand adomdCommand = new AdomdCommand(query, adomdConnection);
    AdomdDataReader reader = adomdCommand.ExecuteReader();

    ConvertReaderToCsv(reader);

    reader.Dispose();
    adomdConnection.Close();
}
```

c) **ConvertReaderToCsv** uses **AdomdDataReader** to process query results and create a CSV file which is opened in Excel.

```
static void ConvertReaderToCsv(AdomdDataReader reader, bool openinExcel = true) {
    string csv = string.Empty;

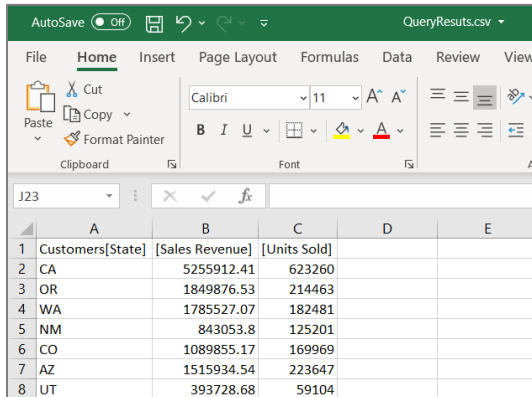
    for (int col = 0; col < reader.FieldCount; col++) {
        csv += reader.GetName(col);
        csv += (col < (reader.FieldCount - 1)) ? "," : "\n";
    }

    // Create loop to iterate each row in the resultset
    while (reader.Read()) {
        // Create loop to iterate each column in the current row
        for (int i = 0; i < reader.FieldCount; i++) {
            csv += reader.GetValue(i);
            csv += (i < (reader.FieldCount - 1)) ? "," : "\n";
        }
    }

    // write CVS data to new file in project directly named QueryResults.csv
    string filePath = System.IO.Directory.GetCurrentDirectory() + @"\QueryResuts.csv";
    StreamWriter writer = File.CreateText(filePath);
    writer.Write(csv);
    writer.Flush();
    writer.Dispose();

    if (openinExcel) {
        OpenInExcel(filePath);
    }
}
```

4. Test your work by executing a DAX query.
  - a) Run a debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.
  - b) When the application runs, it should generate a new CSV file with the query results.
  - c) After generating a CSV file, the application should then open the new CSV file in Microsoft Excel.

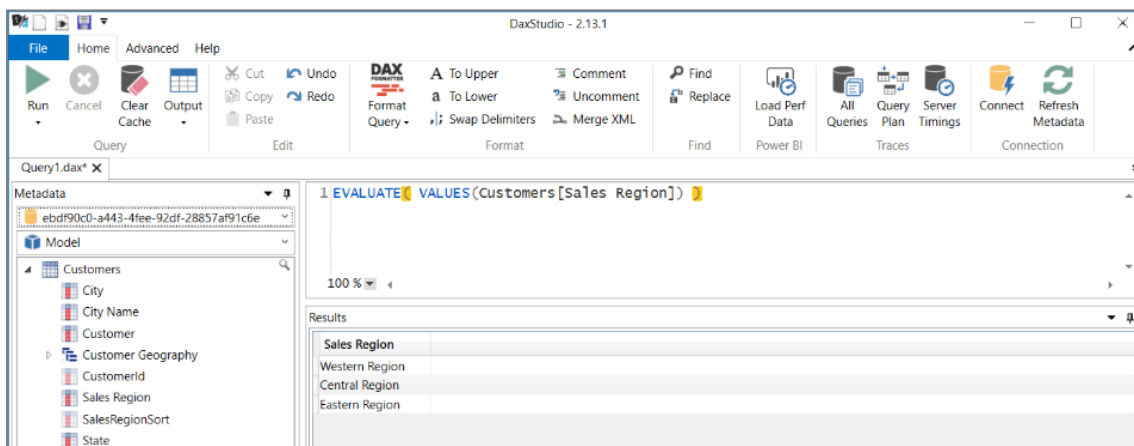


	A	B	C	D	E
1	Customers[State]	[Sales Revenue]	[Units Sold]		
2	CA	5255912.41	623260		
3	OR	1849876.53	214463		
4	WA	1785527.07	182481		
5	NM	843053.8	125201		
6	CO	1089855.17	169969		
7	AZ	1515934.54	223647		
8	UT	393728.68	59104		

- d) Close Microsoft Excel without saving any changes and return to Visual Studio Code.
5. Write a DAX query to return the distinct values that exist for the **Sales Region** column of the **Customers** table
  - a) Consider the following DAX query which returns distinct values for the **Sales Region** column of the **Customers** table.

```
EVALUATE( VALUES(Customers[Sales Region]) )
```

- b) If you run this query in DAX Studio, **Sales Region** has values of **Western Region**, **Central Region** and **Eastern Region**.



DAX Studio - 2.13.1

Query1.dax\* X

Metadata

- ebdf90c0-a443-4fee-92df-28857af91c6e
  - Model
    - Customers
      - City
      - City Name
      - Customer
      - Customer Geography
      - Customerid
      - Sales Region
      - SalesRegionSort
      - State

1 EVALUATE VALUES(Customers[Sales Region])

Results

Sales Region
Western Region
Central Region
Eastern Region

You do not need to use DAX Studio to complete this lab. This screenshot is primarily included so you can see the results of the query to enhance your conceptual understanding. However, you should keep in mind that DAX Studio is an awesome tool for testing DAX queries and optimizing performance. If you have never used this essential tools before, check it out at <https://daxstudio.org/>.

6. Update the Main function in **Program.cs**.
  - a) In **Main**, comment out the call to **ExecuteDaxQuery** and uncomment **AddSalesRegionMeasures** as shown in this listing.

```
static void Main(string[] args) {
    // Exercise 3 - Part 1
    // ExecuteDaxQuery();

    // Exercise 3 - Part 2
    AddSalesRegionMeasures();
}
```



- b) Walk through the code in **AddSalesRegionMeasures** to understand what it does.

```
static void AddSalesRegionMeasures() {

    // DAX query to be submitted to tabular database engine
    String query = "EVALUATE( VALUES(Customers[Sales Region]) )";

    AdomdConnection adomdConnection = new AdomdConnection(connectString);
    adomdConnection.Open();

    AdomdCommand adomdCommand = new AdomdCommand(query, adomdConnection);
    AdomdDataReader reader = adomdCommand.ExecuteReader();

    // open connection use TOM to create new measures
    TOM.Server server = new TOM.Server();
    server.Connect(connectString);
    TOM.Model model = server.Databases[0].Model;
    TOM.Table salesTable = model.Tables["Sales"];

    String measureDescription = "Auto Measures";

    // delete any previously created "Auto" measures
    foreach (TOM.Measure m in salesTable.Measures) {
        if (m.Description == measureDescription) {
            salesTable.Measures.Remove(m);
            model.SaveChanges();
        }
    }

    // Create new automeasure measures
    while (reader.Read()) {

        String SalesRegion = reader.GetValue(0).ToString();
        String measureName = $"{SalesRegion} Sales";

        TOM.Measure measure = new TOM.Measure() {
            Name = measureName,
            Description = measureDescription,
            DisplayFolder = "Auto Measures",
            FormatString = "$#,##0",
            Expression = $"@CALCULATE(SUM(Sales[SalesAmount]), Customers[Sales Region] = \"{SalesRegion}\")"
        };

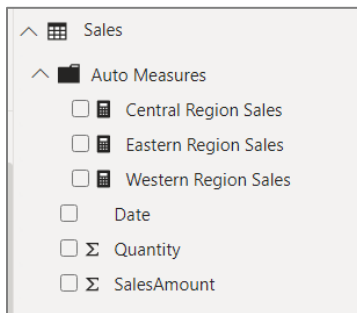
        salesTable.Measures.Add(measure);
    }

    model.SaveChanges();
    reader.Dispose();
    adomdConnection.Close();
}
```

- c) Save your changes to **Program.cs**. Once again, the only change you should make it to the **Main** function.

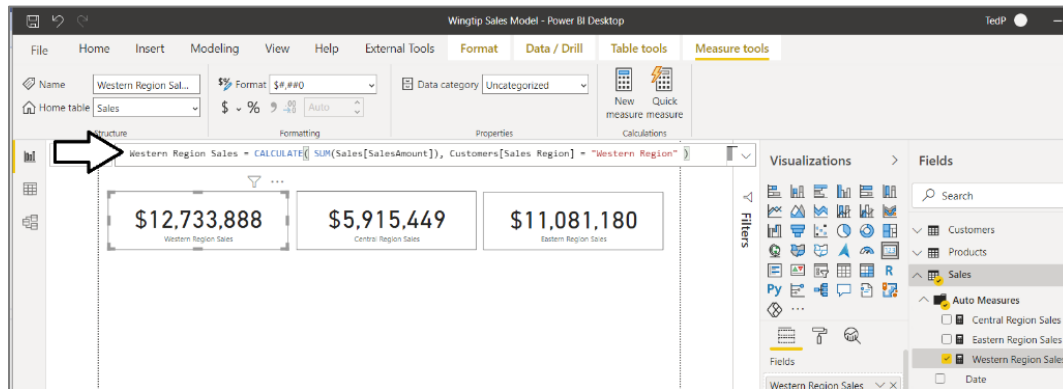
7. Run the program in the .NET debugger.

- a) Run a debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.
- b) Once the program has completed, you should see that a new measure has been created for each sales region.





- c) You should be able to verify that each measure filters by a specific sales region value in its evaluation.



Now you have seen how TOM and the AMOMD Client library can be used together for build creative data modeling solutions.

## Exercise 4: Integrate a Console Application as an External Tool in Power BI Desktop

In this final exercise, you will integrate your console application with Power BI Desktop as an external tool. Once nice benefit is that your application doesn't need to determine the port number for a Power BI Desktop project. That's because Power BI Desktop will automatically pass the port number for the current PBIX project to your console application when it's launched as an external tool.

- Copy and paste the Exercise 4 starter code into **Program.cs**.
  - Open **Program.cs** and delete all the code that is currently inside.
  - Copy the Exercise 4 starter code from the [Exercise04-Program.cs](#) file in the **StarterFiles** folder to the Windows clipboard.
  - Return to Visual Studio Code and paste in the contents of the Windows clipboard into **Program.cs**.

```

1 using System;
2 using Microsoft.AnalysisServices.Tabular;
3 using Microsoft.AnalysisServices.AdomdClient;
4 using System.Diagnostics;
5 using System.IO;
6 using System.Collections.Generic;
7 using System.Text.RegularExpressions;
8
9 class Program {
10
11     static Server server;
12     static string consoleDelimeter = "*****";
13     static string consoleHeader = "*****";
14
15     static string appFolder = Environment.CurrentDirectory + "\\ ";
16

```

- Save and close **Program.cs**.
- Configure the .NET 5 debugger to run the console application in an external console window.
    - Open the **launch.json** file and locate the **console** setting.


```

1 {
2     // Use IntelliSense to learn about possible attributes.
3     // Hover to view descriptions of existing attributes.
4     // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
5     "version": "0.2.0",
6     "configurations": [
7         {
8             "name": ".NET Core Launch (console)",
9             "type": "coreclr",
10            "request": "launch",
11            "preLaunchTask": "build",
12            "program": "${workspaceFolder}/bin/Debug/net5.0/PBI-Tool.dll",
13            "args": [],
14            "cwd": "${workspaceFolder}",
15            "console": "internalConsole",
16            "stopAtEntry": false,
17            "logging": { "moduleLoad": false }
18        }
19    ]
20 }

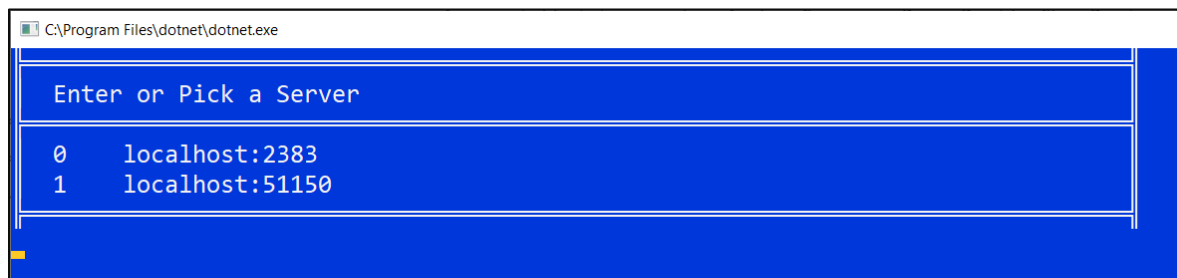
```

- b) Update the **console** setting from **internalConsole** to **externalConsole**.

```
"program": "${workspaceFolder}/bin/Debug/net5.0/PBI-Tool.dll",
"args": [],
"cwd": "${workspaceFolder}",
"console": "externalTerminal",
"stopAtEntry": false,
"logging": { "moduleLoad": false }
```

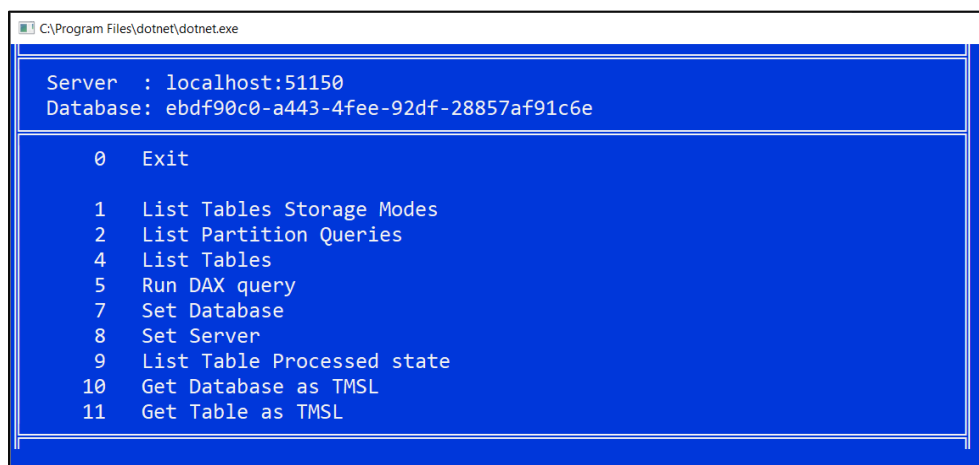


- c) Save and close **launch.json**.
3. Run the program in the .NET debugger.
- a) Run a debugging session by running the **Run > Start Debugging** command or pressing the **{F5}** key.
- a) When the program runs, it should launch in a separate, external console window.

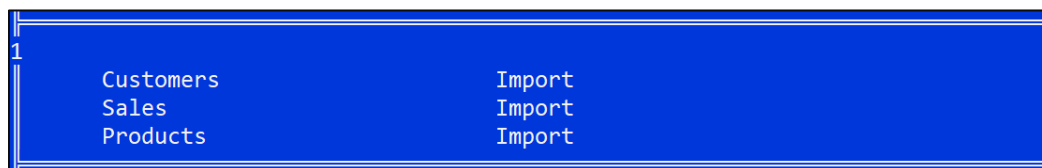


If you are running a local instance of SQL Server Analysis Service (SSAS), you will see that instance with a port number of **2383**.

- b) Select one of the servers by typing 0 or 1 and pressing **ENTER**.
- c) You should see a menu of options.



- d) Type 1 and then press **ENTER** to display a list of tables with their storage mode.
- e) You should see a list of tables with an indication of the storage mode used by each table.



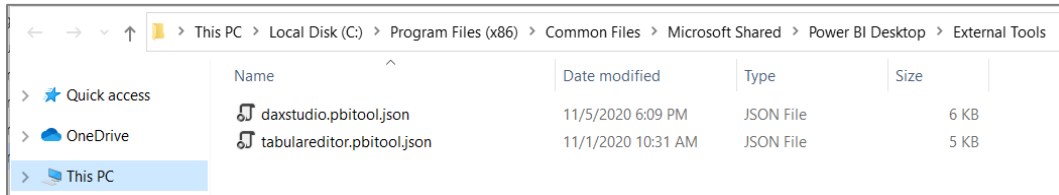
- f) Close console window.

4. Integrate the **PBI-Tool** console application with Power BI Desktop as an external tool

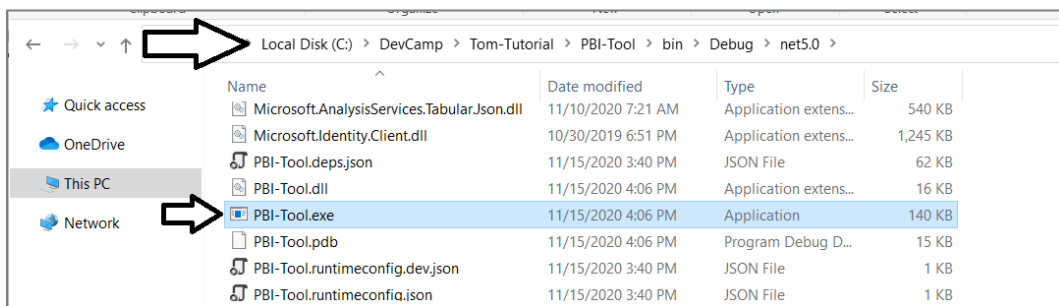
- a) Using Windows Explorer, examine the folder at the following path which holds a JSON file for each external tool.

[C:\Program Files \(x86\)\Common Files\Microsoft Shared\Power BI Desktop\External Tools](C:\Program Files (x86)\Common Files\Microsoft Shared\Power BI Desktop\External Tools)

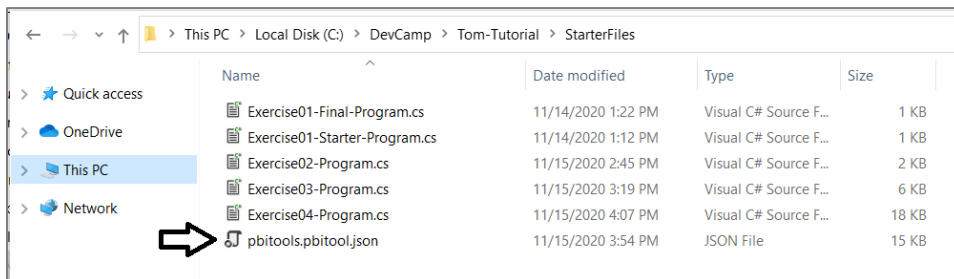
- b) This folder might be empty on your PC. However, it might already have one or more JSON files if you have installed other external tools such as DAX Studio or the Tabular Editor.



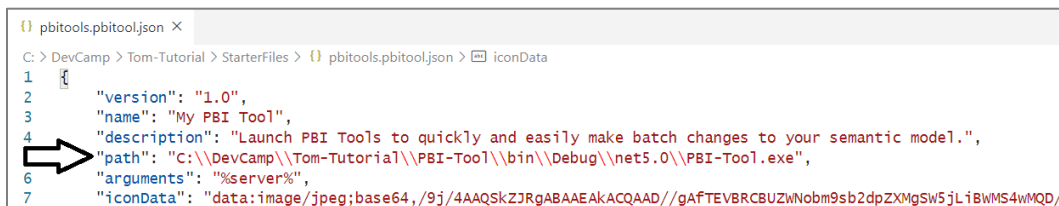
- c) Determine the file path to
- PBI-Tools.exe**
- which should be in the
- bin/Debug/net5.0**
- folder of your project.



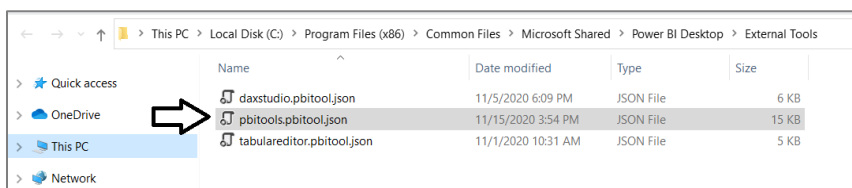
- d) Locate the file named
- pbtools.pbtool.json**
- in the
- StarterFiles**
- folder and open it in a text editor.



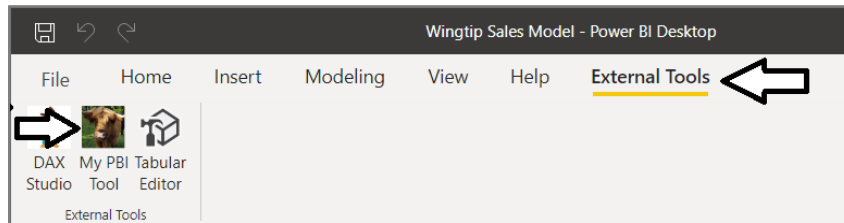
- e) Make sure the
- path**
- property has the correct value for
- PBI-Tools.exe**
- for your machine.



- f) Save your changes to
- pbtools.pbtool.json**
- and then copy it to Power BI Desktop external tools folder.



5. Test your work in Power BI Desktop.
- If Power BI Desktop is already running, shut down all running instances.
  - Restart Power BI Desktop and open the project named **Wingtip Sales Model.pbix**.
  - Navigate to the **External Tools** tab.
  - Click the button with the caption **My PBI Tool**.



- e) Experiment with the **PBI-Tool** console application to see how allows you to interact with the data model in Power BI Desktop.

```
C:\DevCamp\Tom-Tutorial\PBI-Tool\bin\Debug\net5.0\PBI-Tool.exe

Server : localhost:52809
Database: d847dce8-a1ba-470b-bec0-2473eaf1a6c9

0 Exit
1 List Tables Storage Modes
2 List Partition Queries
4 List Tables
5 Run DAX query
7 Set Database
8 Set Server
9 List Table Processed state
10 Get Database as TMSL
11 Get Table as TMSL
```

Congratulations, you have completed this lab. At this point, you are free to experiment with using the PBI-Tool console application. If you are curious (and you *should* be), you can take a deeper look at the code inside **Program.cs** to see how this tools was written.

Special thanks again to **Phil Seamark** of the Power BI CAT team for his blog posts and the code he wrote which was used to build out these lab exercises. For more from Phil, you can follow his blog at <https://dax.tips/>.