# Using TOM to Create a DirectLake Dataset

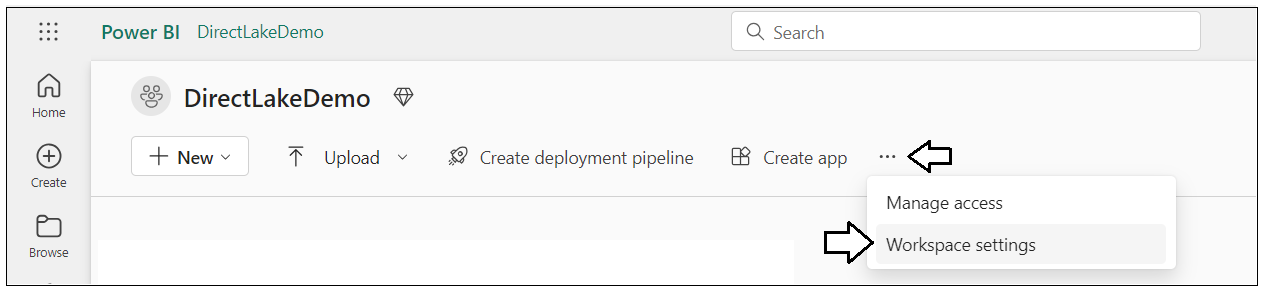
This GitHub repository contains a sample C# console application named **TOM\_CreateFabricDataset** which demonstrates how to automate the creation of a DirectLake mode dataset for Power BI using the Tabular Object Model (TOM). Inside this repository, there is also a Fabric notebook named **CreateLakehouseTables.ipynb** with Python code which can be used to generate Lakehouse tables in delta format that will be used the underlying datasource for the DirectLake dataset.

Here are the high-level steps to set up and complete the demonstration:

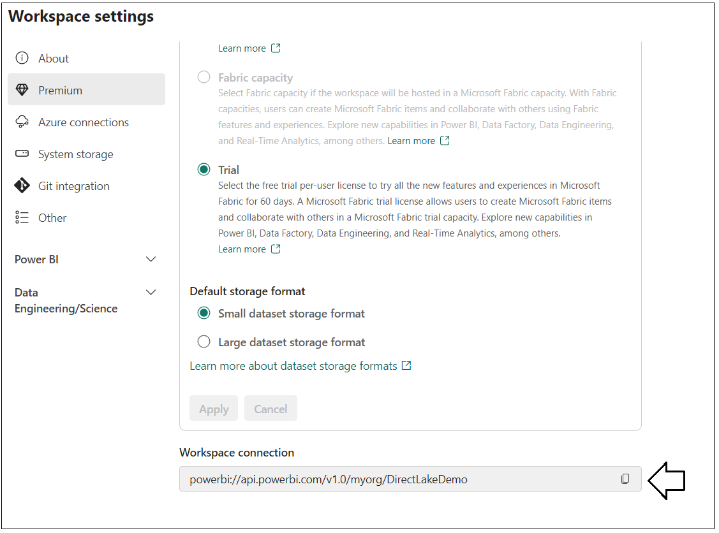
* Create a new workspace associated with Fabric capacity
* Create a new Lakehouse in the new workspace
* Generate Lakehouse tables using pre-provided Python code in Fabric notebook
* Set up and run the custom C# application to automate the creation of a DirectLake datatset

## Create a new workspace associated with Fabric capacity

Create a new workspace with a name such as **DirectLakeDemo**. Make sure the workspace is associated with a Premium capacity or a Fabric trial capacity which provides Fabric capabilities. After creating the new workspace named **DirectLakeDemo**, select the **Workspace settings** menu command to display the **Workspace settings** pane.



Select the **Premium** tab in the left navigation and scroll down to locate the **Workspace connection** setting. Click the **Copy** button to copy the **Workspace connection** value to the Windows clipboard.



The **Workspace connection** string starts with **powerbi://** and ends with the workspace name.

powerbi://api.powerbi.com/v1.0/myorg/DirectLakeDemo

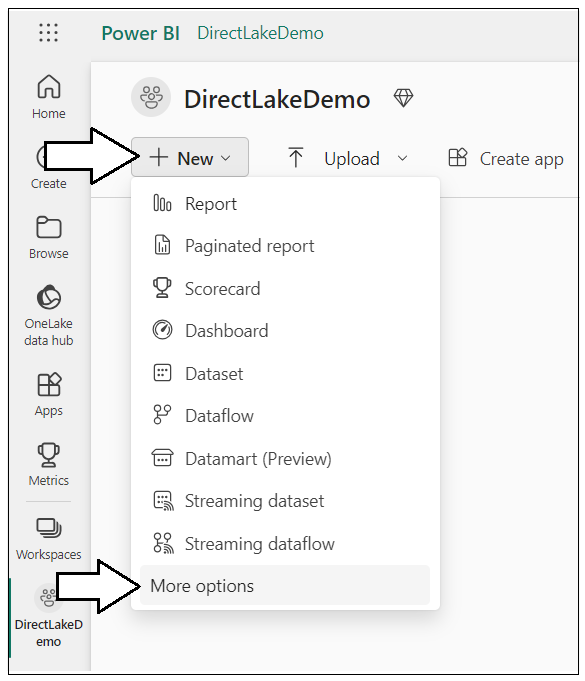
Create a new text file using Notepad.exe to save configuration data you will need later when configuring the C# console application. Copy the **Workspace connection** string into the text file as shown in the following screenshot.

A screenshot of a computer

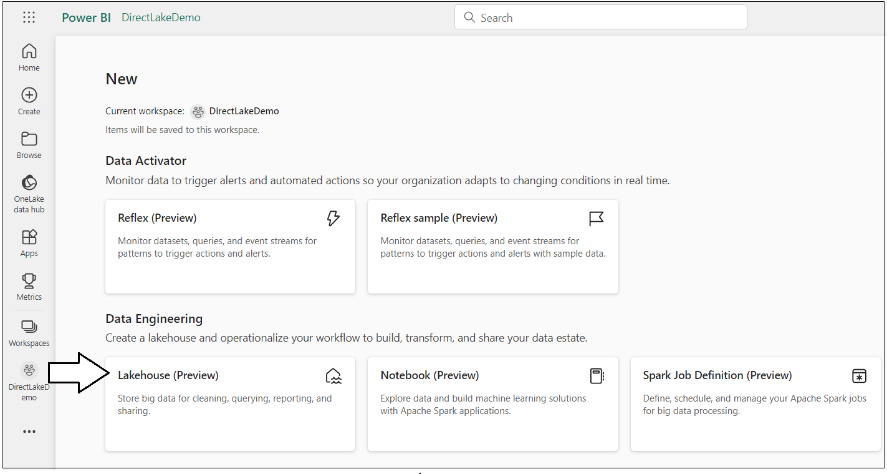
Description automatically generated

## Create a new Lakehouse in the new workspace

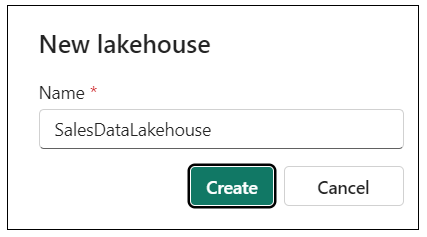
Inside the new workspace you just created, create a new Lakehouse named **SalesDataLakehouse**. Start by dropping down the **+ New** menu button and select **More options**.



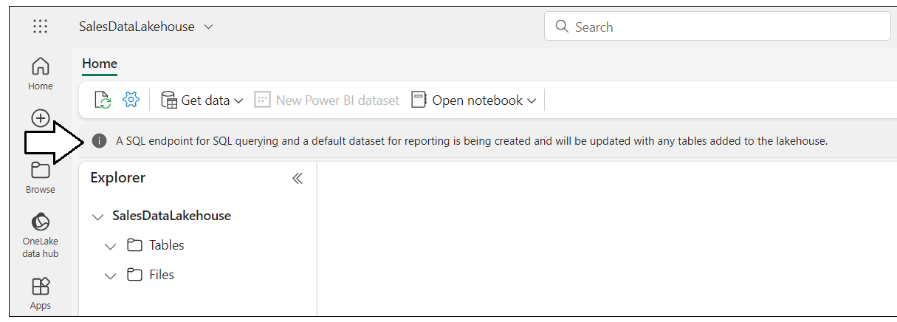
On the **New** page select **Lakehouse (Preview)** from **Data Engineering** section.



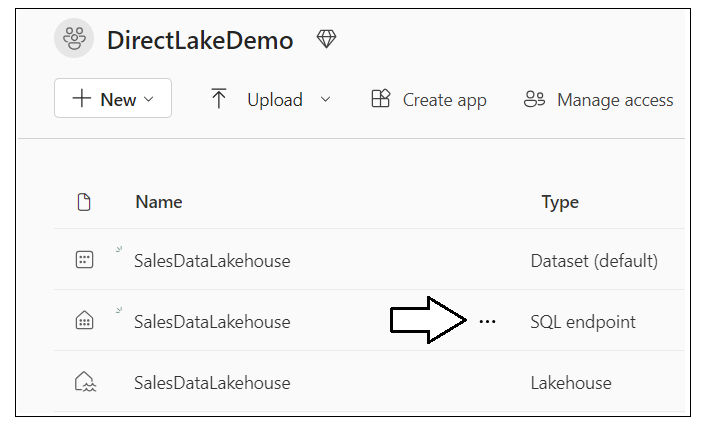
When prompted, enter a name of **SalesDataLakehouse** for the new Lakehouse and click **Create**.



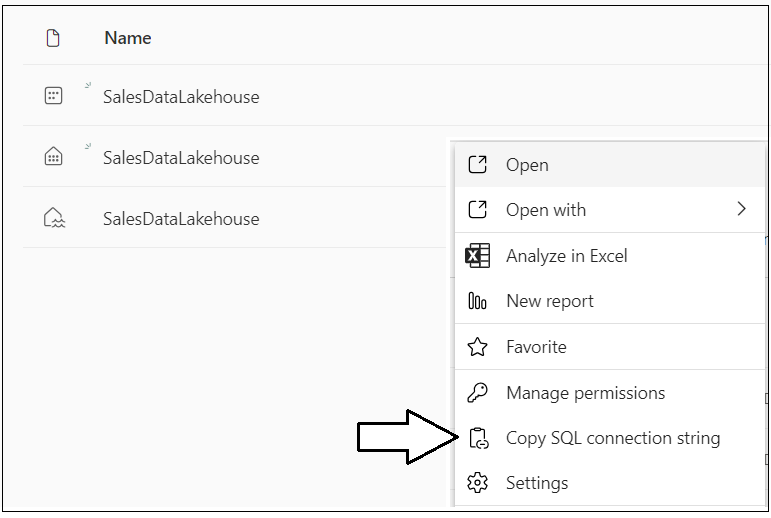
Once you have created the Lakehouse, you will notice a message indicating a SQL Endpoint is also being created.



Navigate back to the main page for the **DirectLakeDemo** workspace. In the list of workspace items, you should see a new item for the new **SQL endpoint** in addition to the item for the Lakehouse. Both the Lakehouse and the SQL endpoint both have the same name of **SalesDataLakehouse**. Drop down the context menu for the SQL endpoint named **SalesDataLakehouse**.



Select the **Copy SQL connection string** menu command from the context menu of the SQL endpoint.



In the **Copy SQL connection string** dialog, click **Copy** to copy the connection string value to the Windows clipboard.

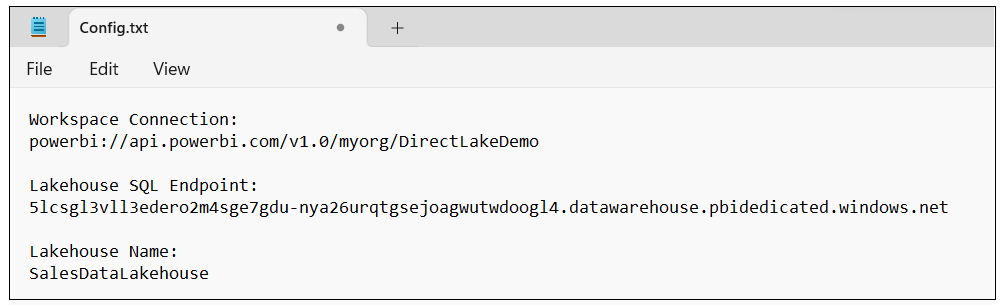
A screenshot of a computer

Description automatically generated

The connection string starts with a unique value and end with **.datawarehouse.pbidedicated.windows.net**.

5lcsgl3vll3edero2m4sge7gdu-nya26urqtgsejoagwutwdoogl4.datawarehouse.pbidedicated.windows.net

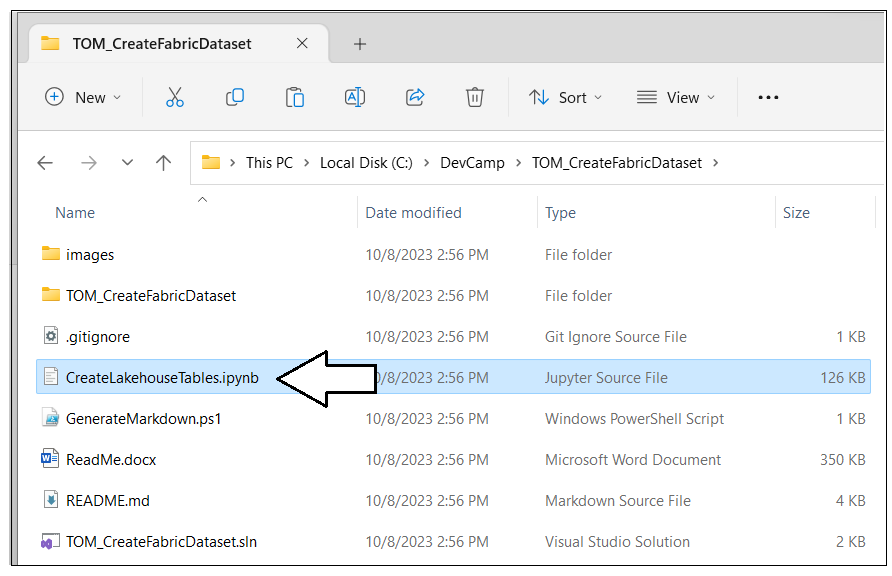
Copy the values for the SQL endpoint and the name of the Lakehouse into the text file with configuration values.



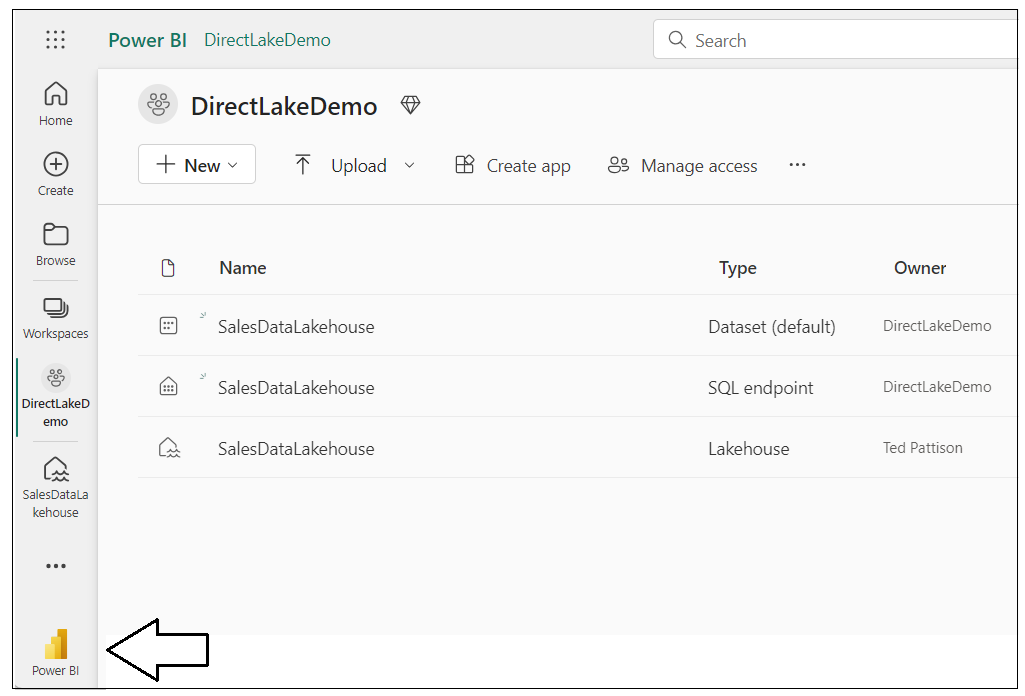
Once again, you will use these configuration values later when setting up the C# console application.

## Create Lakehouse tables using a pre-provided Fabric notebook

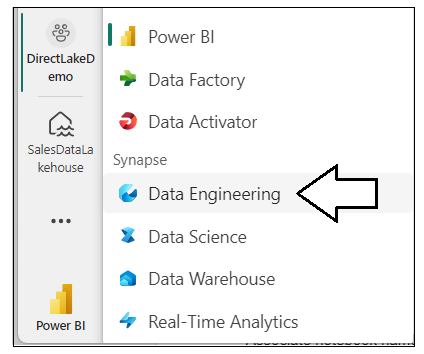
Download all the sources files from this GitHub repository as a single ZIP archive named **TOM\_CreateFabricDataset.zip** using [**this link**](https://github.com/PowerBiDevCamp/TOM_CreateFabricDataset/archive/refs/heads/main.zip). Extract the files from inside **TOM\_CreateFabricDataset.zip** into a local folder on your machine. The first file you will use from these files is a Fabric notebook with Python code named **CreateLakehouseTables.ipynb**.



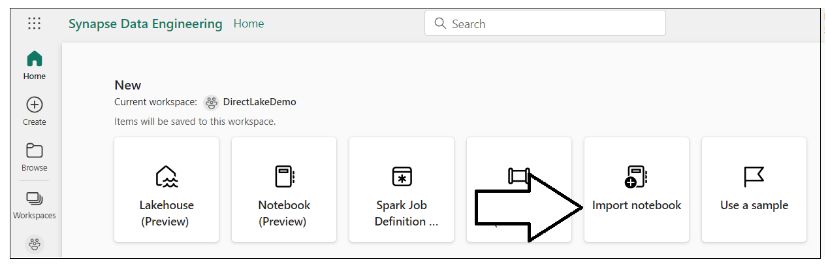
Return to the browser and navigate to the main page of the **DirectLakeDemo** workspace. Locate and click on the ***Fabric Experience Switcher*** menu on the bottom right of the main workspace page.



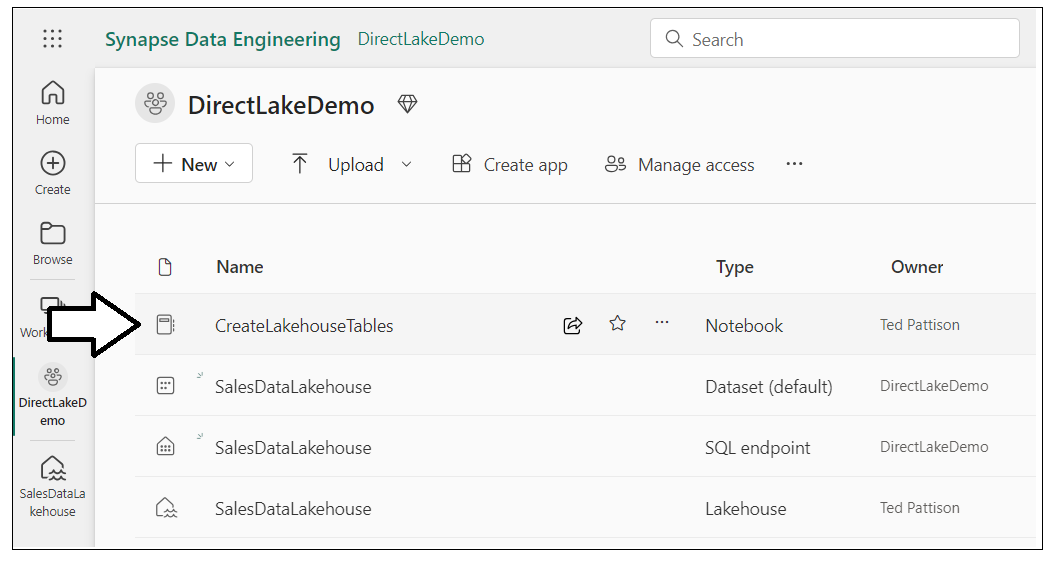
Select Data Engineering from the ***Fabric Experience Switcher*** menu.



Once you switch to **Data Engineering** experience, locate and click on the **Import notebook** button.

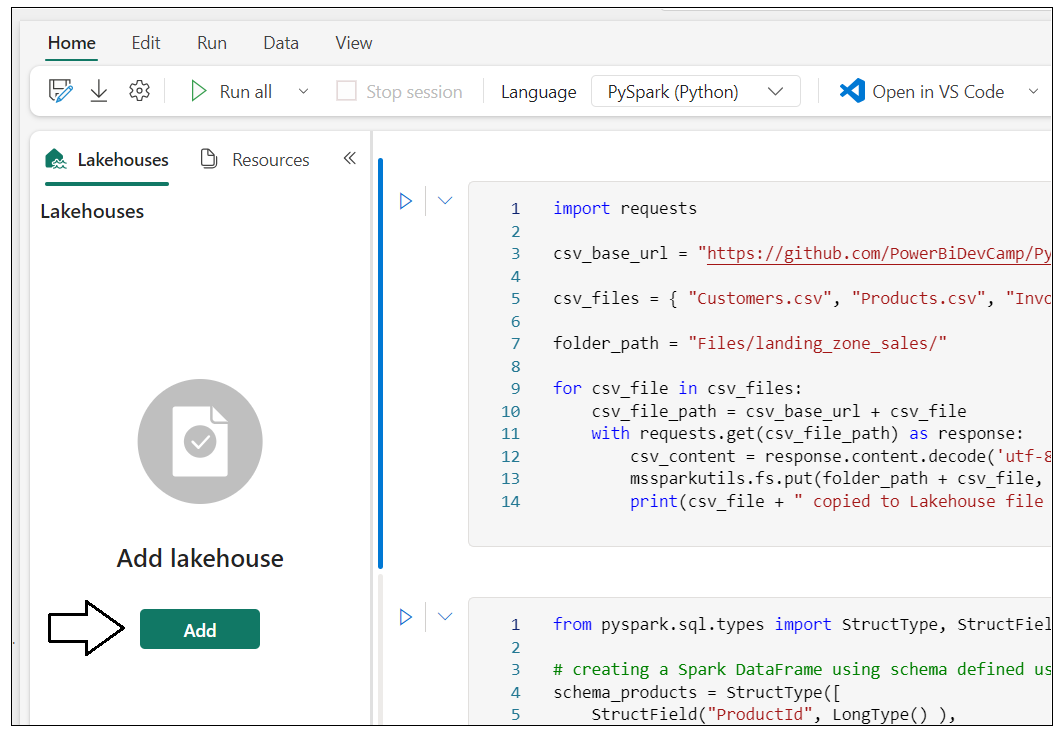


Upload Python notebook named **CreateLakehouseTables.ipynb**. After the notebook has been imported, you should be able to see an item for it on main workspace page. Click on **CreateLakehouseTables.ipynb** to open it in the browser.

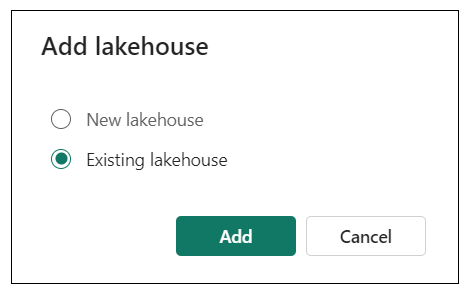


### Associate the Fabric Notebook with the Lakehouse named SalesDataLakehouse

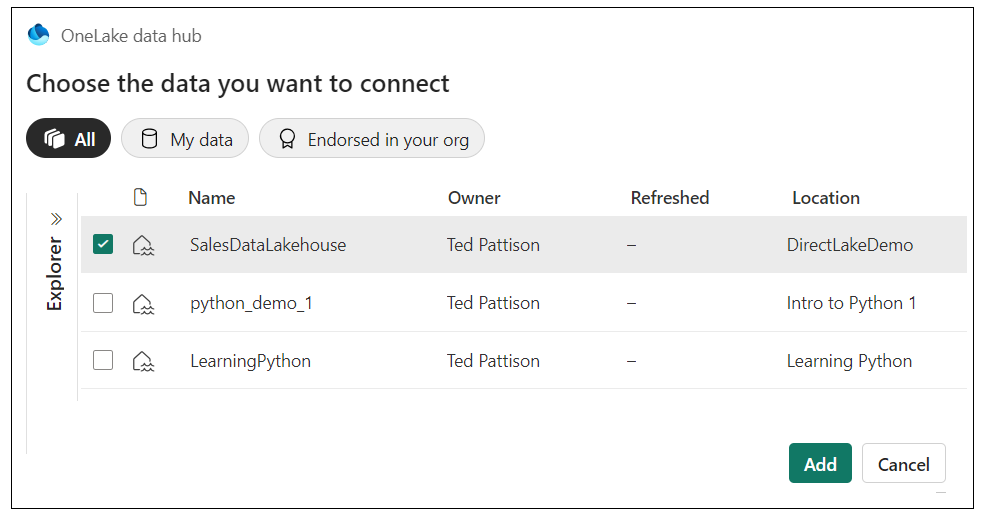
Once the notebook opens, you should see that it is not yet associated with any Lakehouses. Click the **Add** button in the **Lakehouses** pane.



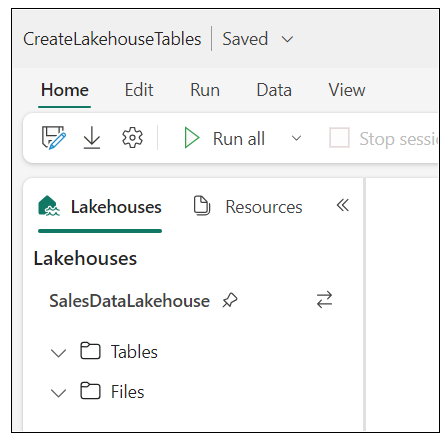
In the **Add lakehouse** dialog, select **Existing lakehouse** and click **Add**.



Select the lakehouse you created earlier named **SalesDataLakehouse** and click **Add**.



Once you have associated the notebook with **SalesDataLakehouse**, you should see the **Tables** folder and the **Files** folder which are both initially empty.



### Copy CSV files from this repository into the file system of your Fabric Lakehouse

Now you will execute the Python code from the cell of the workbook to populate the lakehouse with tables. You will execute the Python code in each cell in this workspace one by one from top to bottom. You will begin by executing Python code to copy four CSV files from this GitHub repository into file system of this lakehouse. Examine the following Python code from the first cell of the notebook.

import requests

csv\_base\_url = "https://github.com/PowerBiDevCamp/Python-In-Fabric-Notebooks/raw/main/ProductSalesData/"

csv\_files = { "Customers.csv", "Products.csv", "Invoices.csv", "InvoiceDetails.csv" }

folder\_path = "Files/landing\_zone\_sales/"

for csv\_file in csv\_files:

csv\_file\_path = csv\_base\_url + csv\_file

with requests.get(csv\_file\_path) as response:

csv\_content = response.content.decode('utf-8-sig')

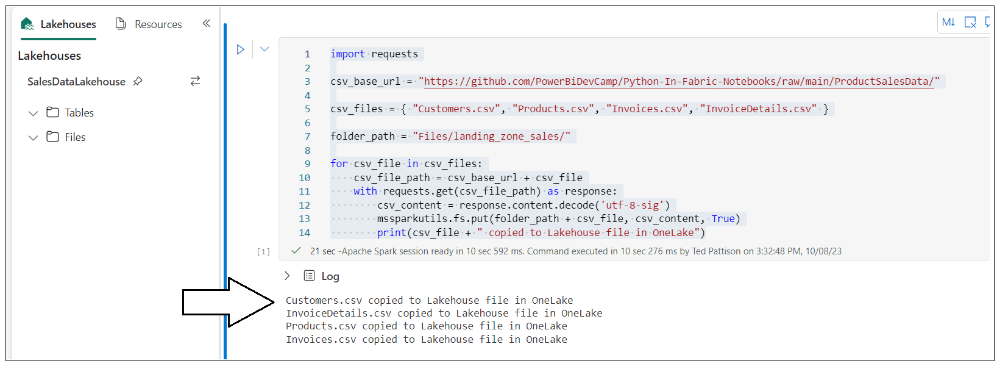
mssparkutils.fs.put(folder\_path + csv\_file, csv\_content, True)

print(csv\_file + " copied to Lakehouse file in OneLake")

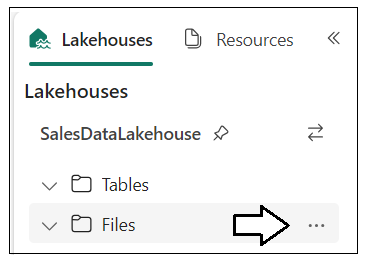
Execute the code in the top notebook cell by clicking the **Execute** button located on top just to the left of the cell.



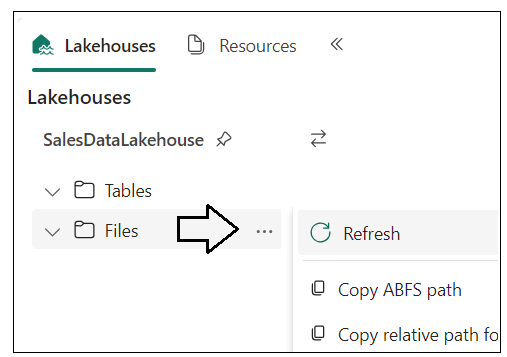
The first time you execute Python code from a Fabric notebook, it typically takes 10-20 seconds to start up and initialize the Spark pool which is used to process the notebook code execution requests. When the Python code completes its execution, you should see a message for each of the CSV files that have been copied into the lakehouse file system.



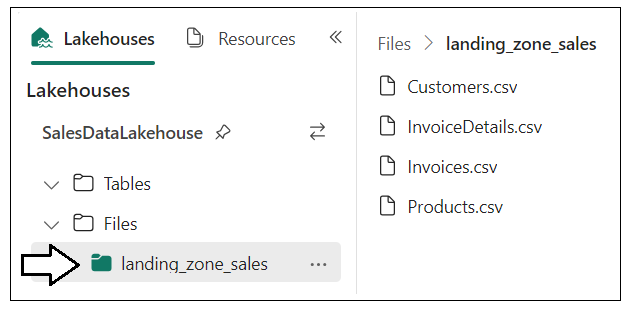
In the **Lakehouses** pane on the left, drop down the context menu for the **Files** folder.



Select the **Refresh** command from the context menu of the **Files** folder.



After the refresh operation completes, you should see a new child folder named **landing\_zone\_sales** inside the **Files** folder. If you select the **landing\_zone\_sales** folder on the left, you should be able to see four new CSV files named **Customers.csv**, **InvoiceDetails.csv**, **Invoices.csv** and **Products.csv**.



You have now copied the CSV files with the raw data into the lakehouse file system. Now you will use Spark to load this data into memory as DataFrames where the data can be manipulated and saved as lakehouse tables.

### Execute code in notebook to load CSV files into Spark DataFrames for the bronze layer

Examine the following Python code from the second cell in the notebook which loads product data from Products.csv into a Spark DataFrame and then displays the DataFrame schema and samples rows of data.

from pyspark.sql.types import StructType, StructField, StringType, LongType, FloatType

# creating a Spark DataFrame using schema defined using StructType and StructField

schema\_products = StructType([

StructField("ProductId", LongType() ),

StructField("Product", StringType() ),

StructField("Category", StringType() )

])

df\_products = (

spark.read.format("csv")

.option("header","true")

.schema(schema\_products)

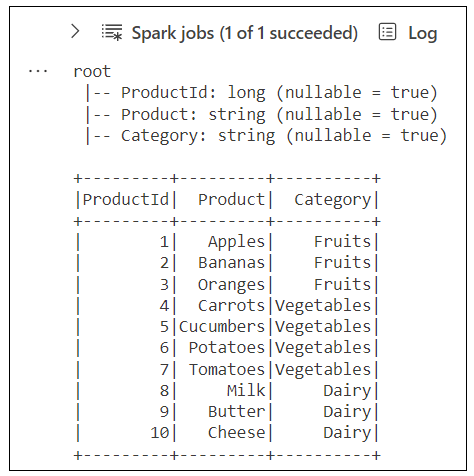
.load("Files/landing\_zone\_sales/Products.csv")

)

df\_products.printSchema()

df\_products.show()

Execute the code in the second cell to load product data into a Spark DataFrame. After the code completes, you should see output which display the DataFrame schema and displays 10 rows of data.



Examine the following Python code from the third cell in the notebook which loads customer data from **Customers.csv** into a Spark DataFrame and then displays the DataFrame schema and samples rows of data.

from pyspark.sql.types import StructType, StructField, StringType, LongType, FloatType, DateType

# creating a Spark DataFrame using schema defined with StructType and StructField

schema\_customers = StructType([

StructField("CustomerId", LongType() ),

StructField("FirstName", StringType() ),

StructField("LastName", StringType() ),

StructField("Country", StringType() ),

StructField("City", StringType() ),

StructField("DOB", DateType() ),

])

df\_customers = (

spark.read.format("csv")

.option("header","true")

.schema(schema\_customers)

.option("dateFormat", "M/d/yyyy")

.option("inferSchema", "true")

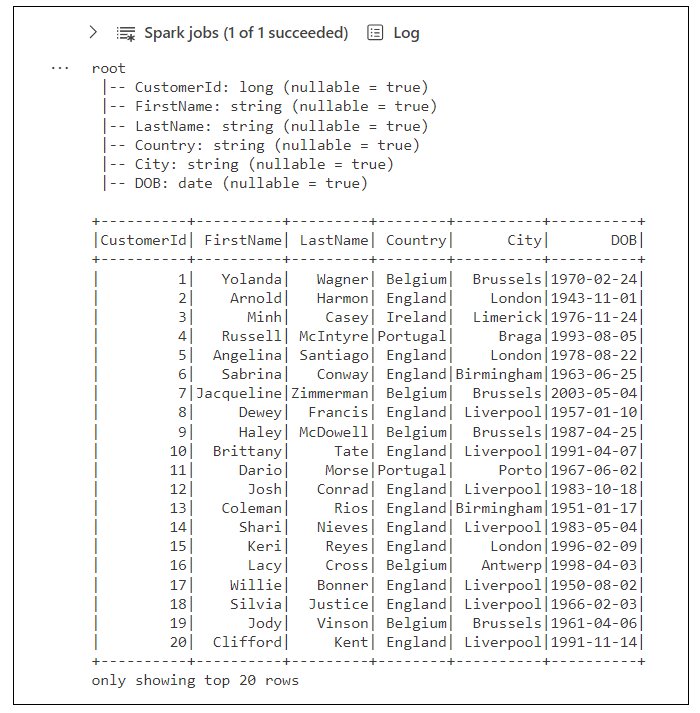
.load("Files/landing\_zone\_sales/Customers.csv")

)

df\_customers.printSchema()

df\_customers.show()

Execute the code in the third cell to load customer data into a Spark DataFrame. After the code completes, you should see output which display the DataFrame schema and displays the top 20 rows of data.



Examine the Python code in the next cell which loads customer data from **Invoices.csv** into a Spark DataFrame and then displays the DataFrame schema and samples rows of data.

from pyspark.sql.types import StructType, StructField, StringType, LongType, FloatType, DateType

# creating a Spark DataFrame using schema defined using StructType and StructField

schema\_invoices = StructType([

StructField("InvoiceId", LongType() ),

StructField("Date", DateType() ),

StructField("TotalSalesAmount", FloatType() ),

StructField("CustomerId", LongType() )

])

df\_invoices = (

spark.read.format("csv")

.option("header","true")

.schema(schema\_invoices)

.option("dateFormat", "MM/dd/yyyy")

.option("inferSchema", "true")

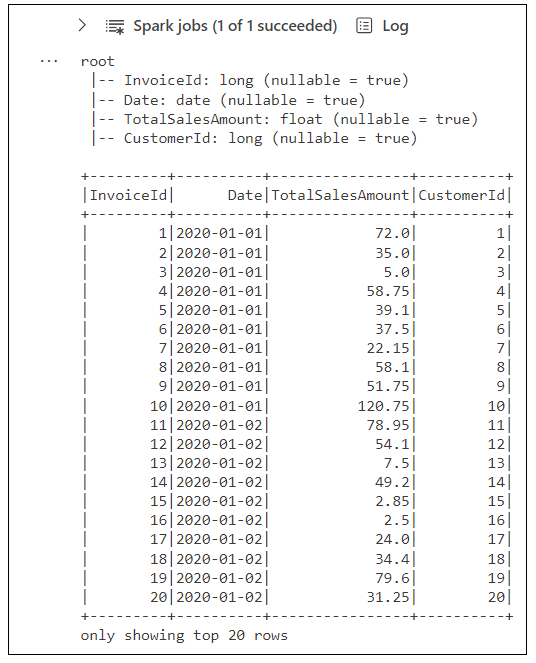
.load("Files/landing\_zone\_sales/Invoices.csv")

)

df\_invoices.printSchema()

df\_invoices.show()

Execute the code to load invoice data into a Spark DataFrame. After the code completes, you should see output which display the DataFrame schema and displays the top 20 rows of data.



Examine the Python code in the next cell which loads customer data from **InvoiceDetails.csv** into a Spark DataFrame and then displays the DataFrame schema and samples rows of data.

from pyspark.sql.types import StructType, StructField, StringType, LongType, FloatType, DateType

# creating a Spark DataFrame using schema defined using StructType and StructField

schema\_invoice\_details = StructType([

StructField("Id", LongType() ),

StructField("Quantity", LongType() ),

StructField("SalesAmount", FloatType() ),

StructField("InvoiceId", LongType() ),

StructField("ProductId", LongType() )

])

df\_invoice\_details = (

spark.read.format("csv")

.option("header","true")

.schema(schema\_invoice\_details)

.option("dateFormat", "MM/dd/yyyy")

.option("inferSchema", "true")

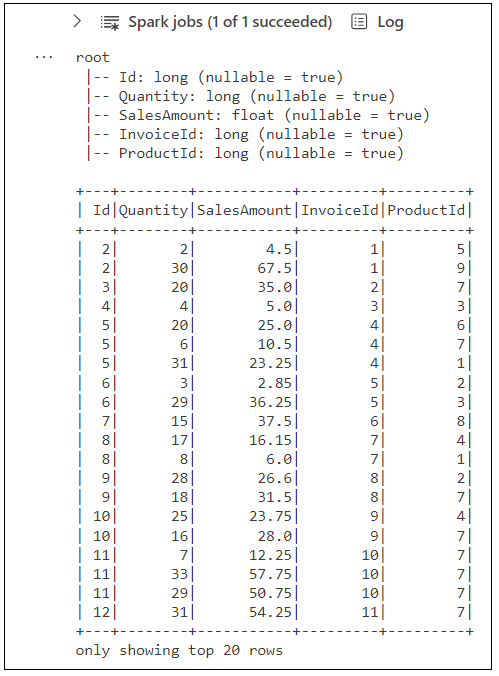
.load("Files/landing\_zone\_sales/InvoiceDetails.csv")

)

df\_invoice\_details.printSchema()

df\_invoice\_details.show()

Execute the code to load invoice detail data into a Spark DataFrame. After the code completes, you should see output which display the DataFrame schema and displays the top 20 rows of data.



You have now create four DataFrames. However, you have just loaded data into memory. Now it’s time to actually persist your work by saving each of these four DataFrames into lakehouse tables using the delta format.

### Execute code to Save the Four DataFrames as Delta Tables in the Lakehouse

Locate and execute the next cell with the following Python code which saves all DataFrames as lakehouse tables with delta format.

# save all bronze layer tables

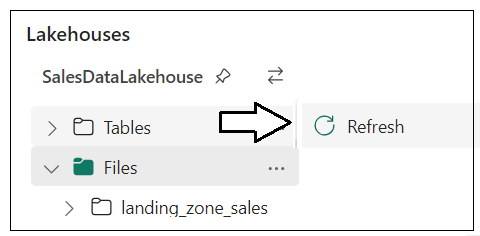
df\_products.write.mode("overwrite").format("delta").save(f"Tables/bronze\_products")

df\_customers.write.mode("overwrite").format("delta").save(f"Tables/bronze\_customers")

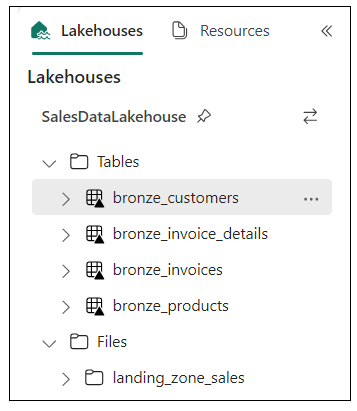
df\_invoices.write.mode("overwrite"). format("delta").save(f"Tables/bronze\_invoices")

df\_invoice\_details.write.mode("overwrite")format("delta").save(f"Tables/bronze\_invoice\_details")

Once the code which creates the lakehouse tables completes, click the **Refresh** context menu of the **Tables** folder.



Once the refresh operation completes, you should be able to see four tables created for the Bronze layer.



### Reshape and Transform Data in Bronze Layer Tables to Create Silver Layer Tables

Ssss

# create silver layer products table

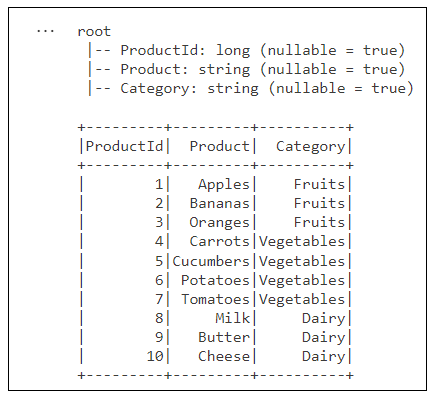
df\_silver\_products = spark.read.format("delta").load("Tables/bronze\_products")

df\_silver\_products.write.mode("overwrite").format("delta").save(f"Tables/products")

df\_silver\_products.printSchema()

df\_silver\_products.show()

cccc



Xx

# create silver layer customers table

from pyspark.sql.functions import concat\_ws, floor, datediff, current\_date, col

df\_silver\_customers = (

spark.read.format("delta").load("Tables/bronze\_customers")

.withColumn("Customer", concat\_ws(' ', col('FirstName'), col('LastName')) )

.withColumn("Age",( floor( datediff( current\_date(), col("DOB") )/365.25) ))

.drop('FirstName', 'LastName')

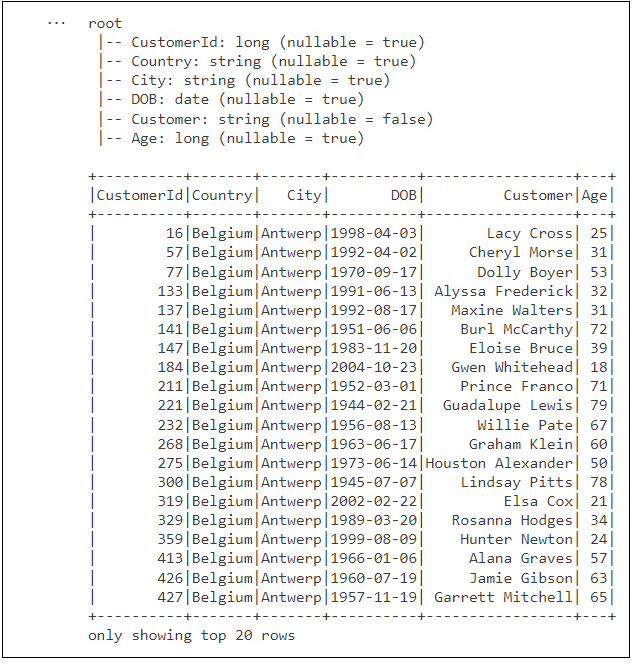
)

df\_silver\_customers.write.mode("overwrite").format("delta").save(f"Tables/customers")

df\_silver\_customers.printSchema()

df\_silver\_customers.show()

ssss



Xx

# create silver layer sales table

from pyspark.sql.functions import col, desc, concat, lit, floor, datediff

from pyspark.sql.functions import date\_format, to\_date, current\_date, year, month, dayofmonth

df\_bronze\_invoices = spark.read.format("delta").load("Tables/bronze\_invoices")

df\_bronze\_invoice\_details = spark.read.format("delta").load("Tables/bronze\_invoice\_details")

df\_silver\_sales = (

df\_bronze\_invoice\_details

.join(df\_bronze\_invoices, df\_bronze\_invoice\_details['InvoiceId'] == df\_bronze\_invoices['InvoiceId'])

.withColumnRenamed('SalesAmount', 'Sales')

.withColumn("DateKey", (year(col('Date'))\*10000) +

(month(col('Date'))\*100) +

(dayofmonth(col('Date'))) )

.drop('InvoiceId', 'TotalSalesAmount', 'InvoiceId', 'Id')

.select('Date', "DateKey", "CustomerId", "ProductId", "Sales", "Quantity")

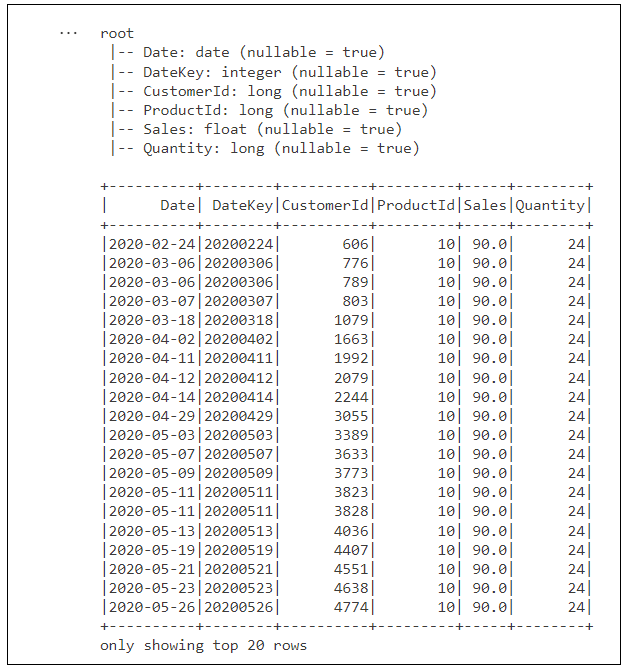
)

df\_silver\_sales.write.mode("overwrite").format("delta").save(f"Tables/sales")

df\_silver\_sales.printSchema()

df\_silver\_sales.show()

xxxx



Xx

# create silver layer calendar table

import pandas as pd

from datetime import datetime, timedelta, date

import os

from pyspark.sql.functions import to\_date, year, month, dayofmonth, quarter, dayofweek

first\_sales\_date = df\_silver\_sales.agg({"Date": "min"}).collect()[0][0]

last\_sales\_date = df\_silver\_sales.agg({"Date": "max"}).collect()[0][0]

start\_date = date(first\_sales\_date.year, 1, 1)

end\_date = date(last\_sales\_date.year, 12, 31)

os.environ["PYARROW\_IGNORE\_TIMEZONE"] = "1"

df\_calendar\_ps = pd.date\_range(start\_date, end\_date, freq='D').to\_frame()

df\_calendar\_spark = (

spark.createDataFrame(df\_calendar\_ps)

.withColumnRenamed("0", "timestamp")

.withColumn("Date", to\_date(col('timestamp')))

.withColumn("DateKey", (year(col('timestamp'))\*10000) +

(month(col('timestamp'))\*100) +

(dayofmonth(col('timestamp'))) )

.withColumn("Year", year(col('timestamp')) )

.withColumn("Quarter", date\_format(col('timestamp'),"yyyy-QQ") )

.withColumn("Month", date\_format(col('timestamp'),'yyyy-MM') )

.withColumn("Day", dayofmonth(col('timestamp')) )

.withColumn("MonthInYear", date\_format(col('timestamp'),'MMMM') )

.withColumn("MonthInYearSort", month(col('timestamp')) )

.withColumn("DayOfWeek", date\_format(col('timestamp'),'EEEE') )

.withColumn("DayOfWeekSort", dayofweek(col('timestamp')))

.drop('timestamp')

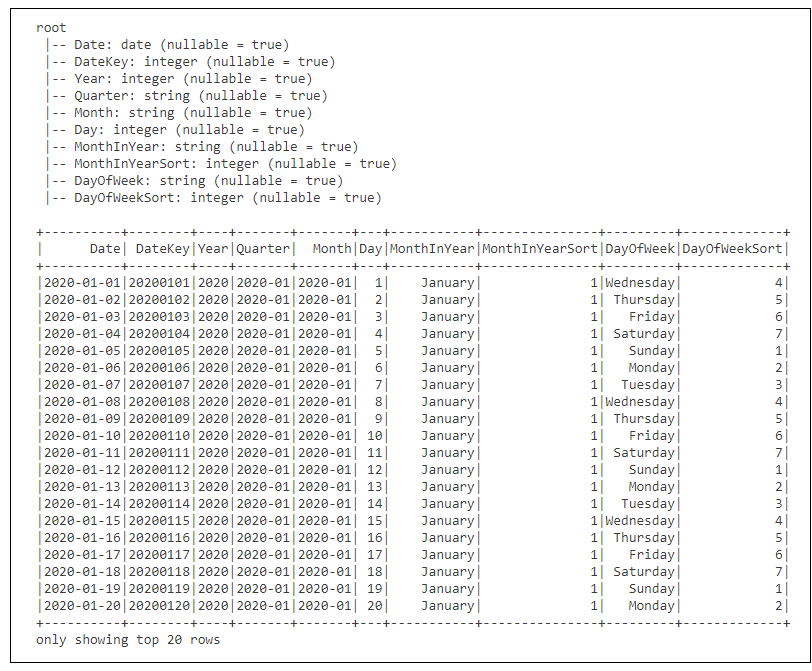
)

df\_calendar\_spark.write.mode("overwrite").format("delta").save(f"Tables/calendar")

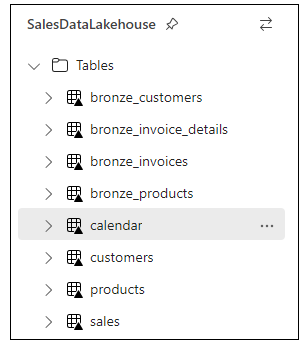
df\_calendar\_spark.printSchema()

df\_calendar\_spark.show()

xx

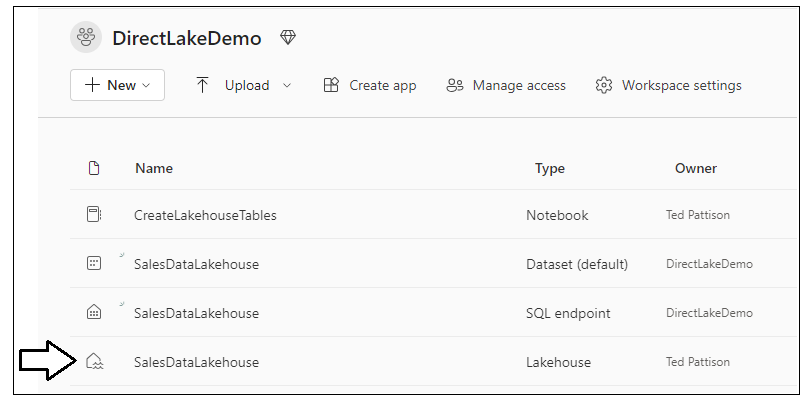


Xx

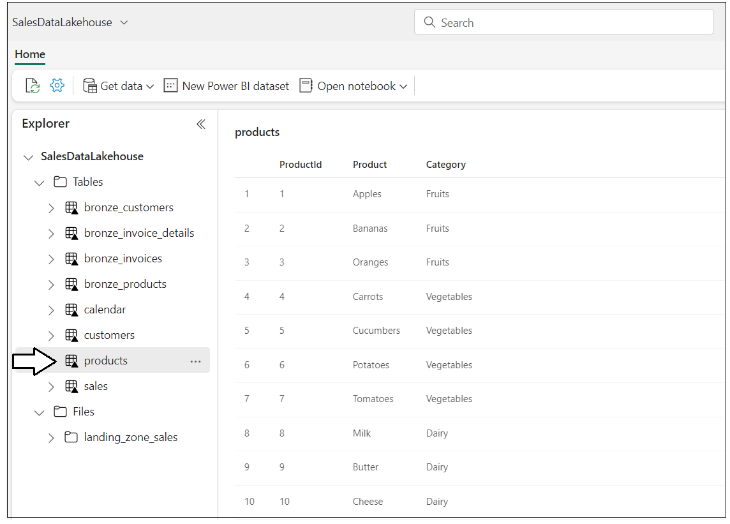


### Inspect the tables that have been created in the Lakehouse

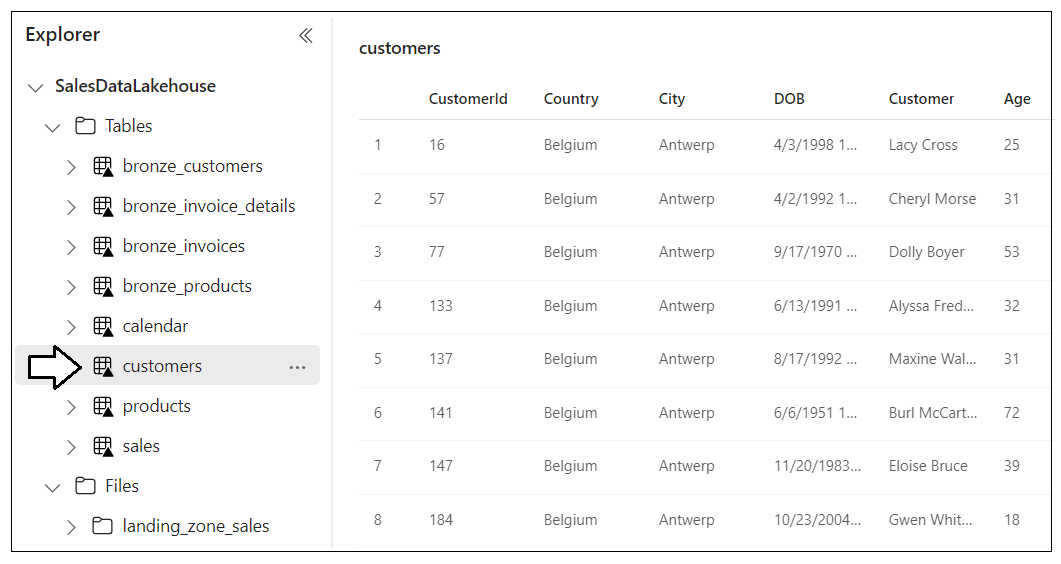
Xxxx



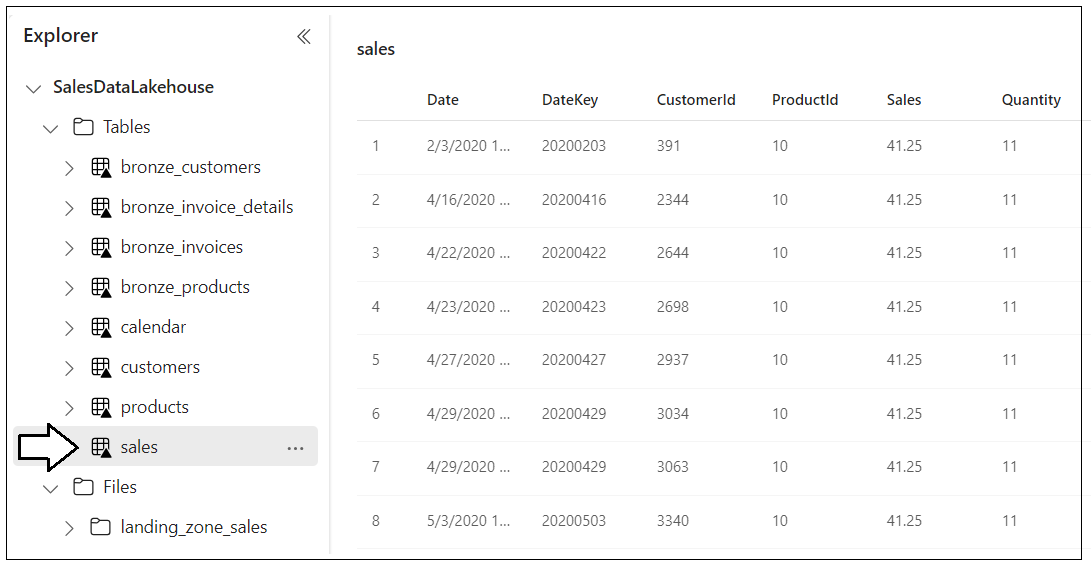
Xxxx



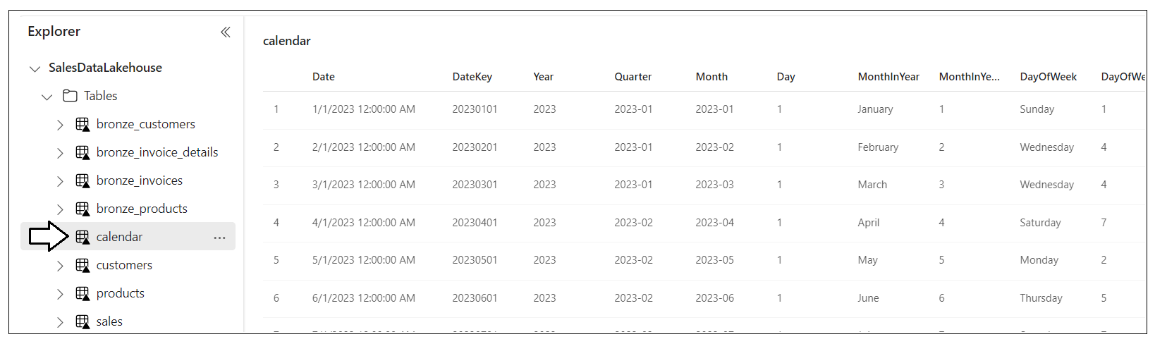
Xxx



Dddd



Xxxx



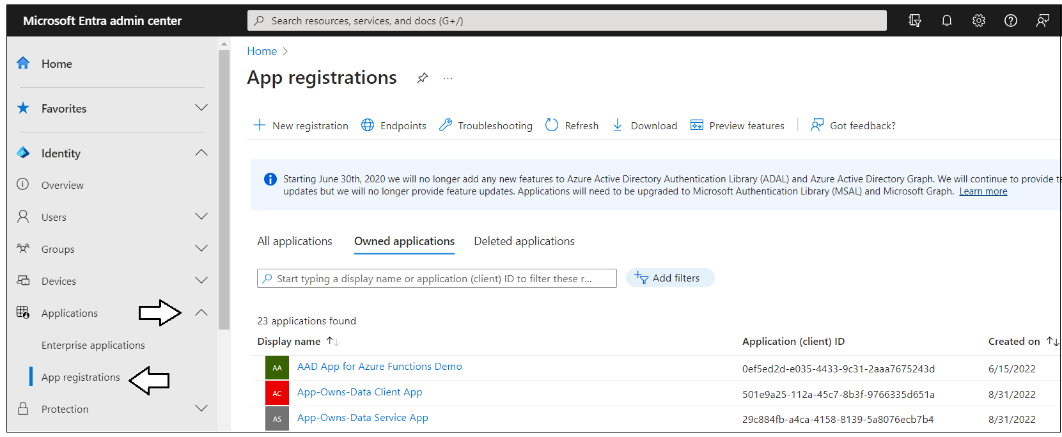
Now all Lakehouse tables have been created and you can move on top the step where you create the DirectLake dataset using the customer application.

## Run the custom C# application to create DirectLake data model using TOM

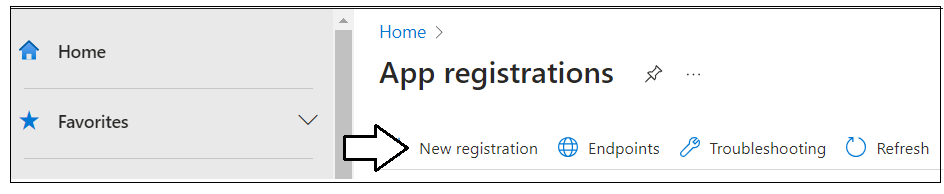
Create Entra ID application. In case you have not heard, Microsoft renamed Azure AD to Microsoft Entra.

Start by going to Azure AD section of the Azure portal.

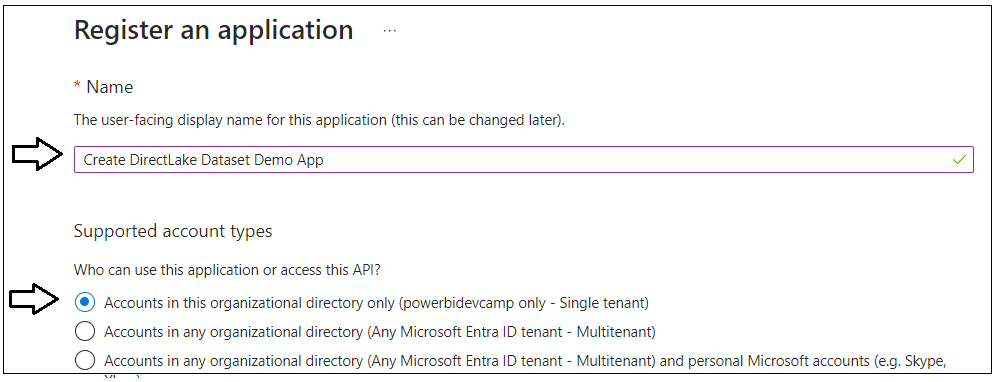
<https://entra.microsoft.com/>



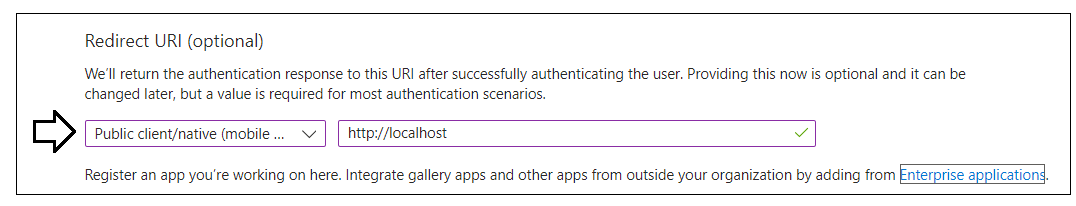
Ss

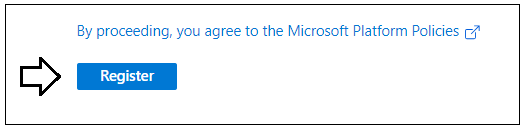


Give it a name such as **Create DirectLake Dataset Demo App**.

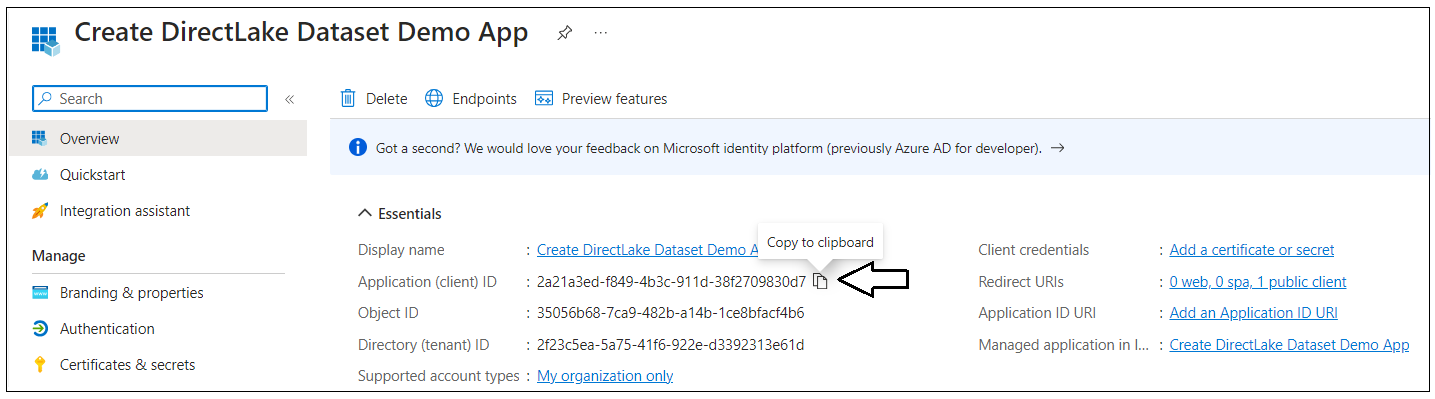


Create a native/public application with redirect URI of <http://localhost>





Record Application ID for use in console application.



Get Application ID

2a21a3ed-f849-4b3c-911d-38f2709830d7

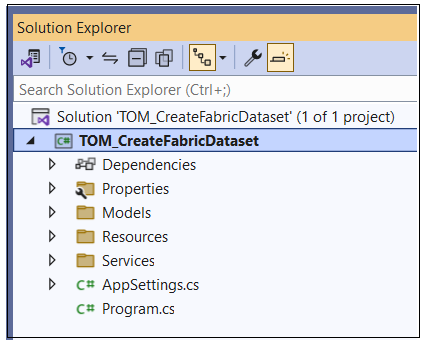
sss

A screenshot of a computer

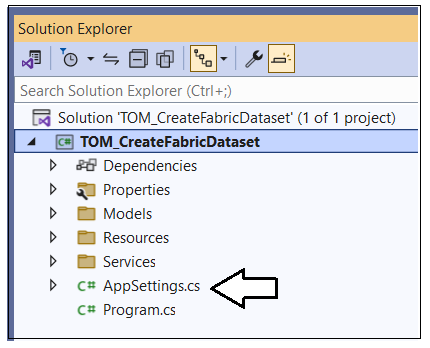
Description automatically generated

### Open C# console application project in Visual Studio 2022

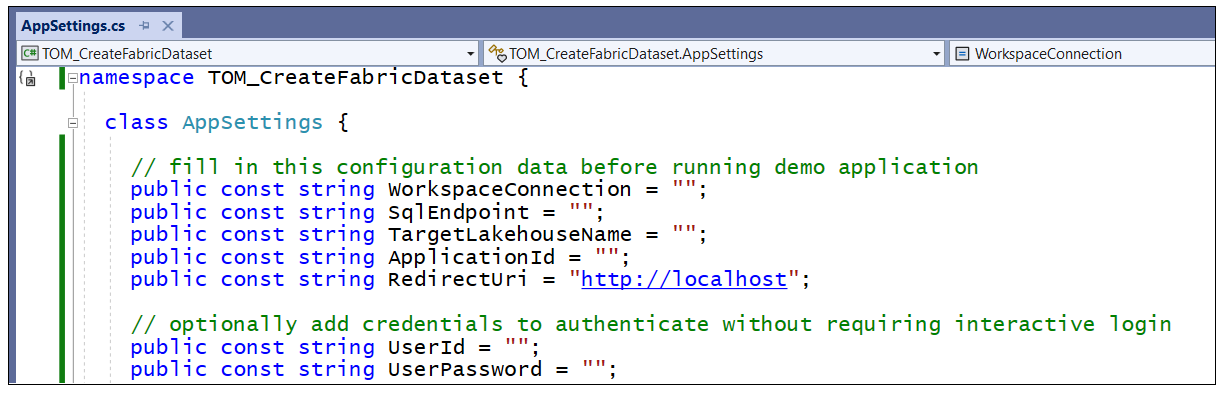
Xxxx



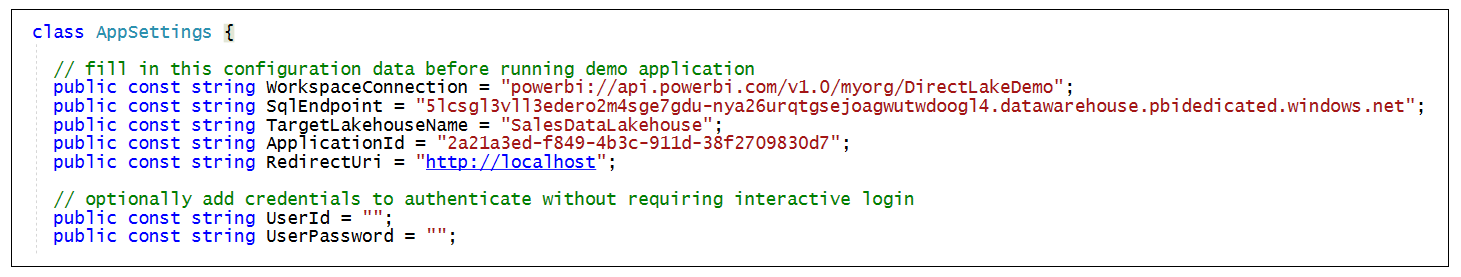
Open **AppSettings.cs** and update configuration value:



Xxxx



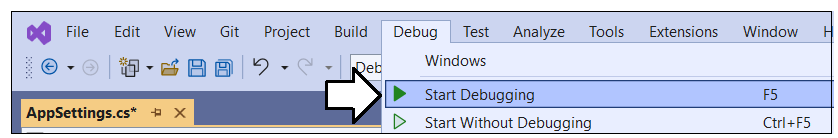
Add Workspace Connection, SQL Endpoint, Lakehouse Name, Application ID and endure RedirectUrl is connect.



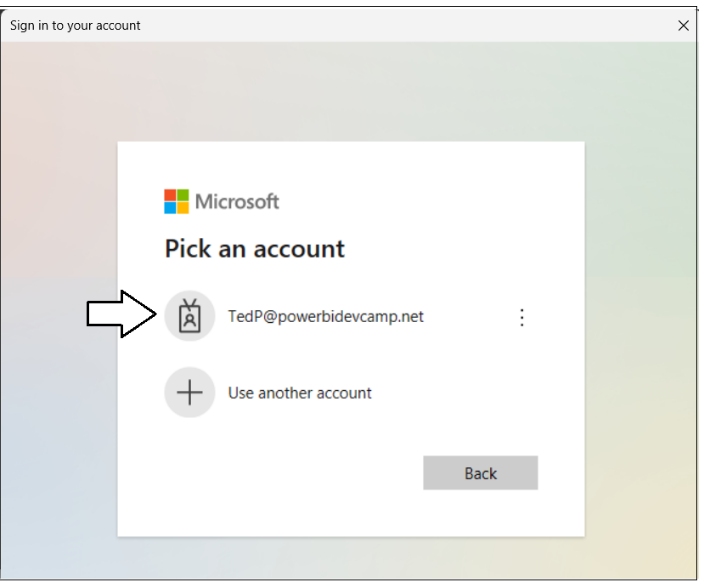
Save changes

### Run application

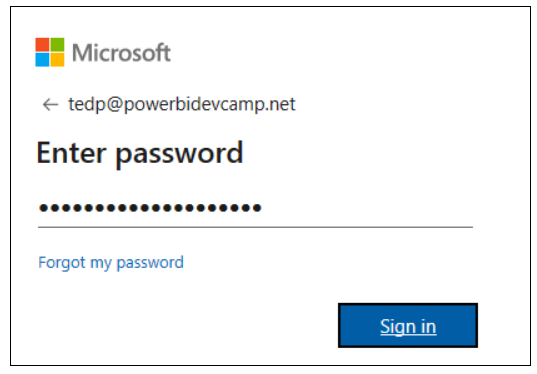
Xxx



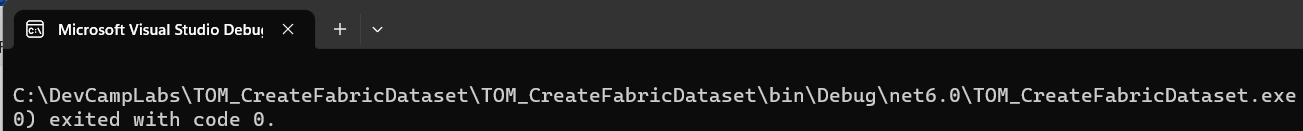
Xxx



Ssss

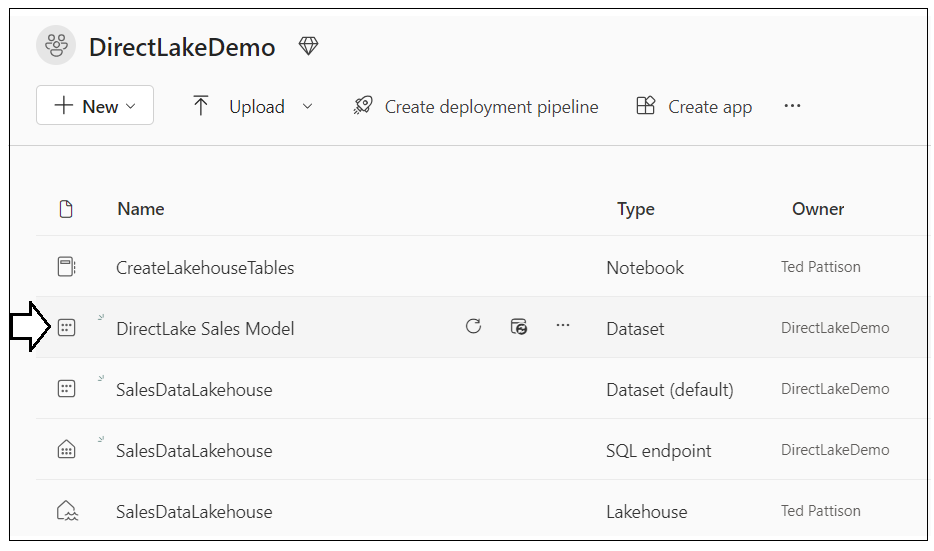


Sssss

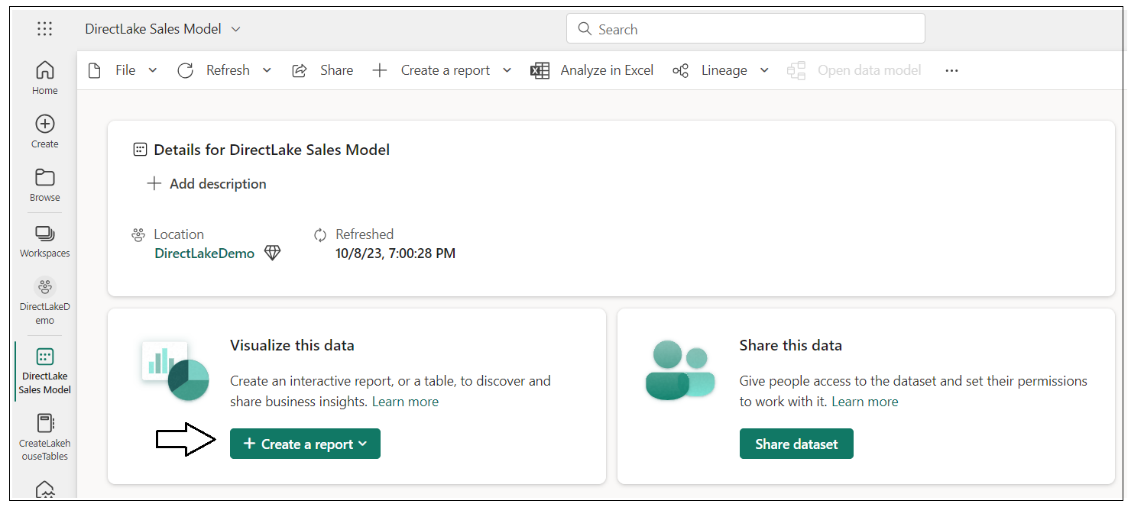


It should run without error

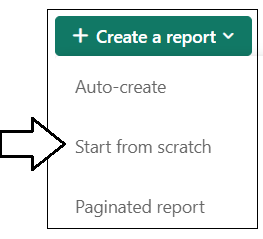
When done, verify you can see new data model and use it to create new report



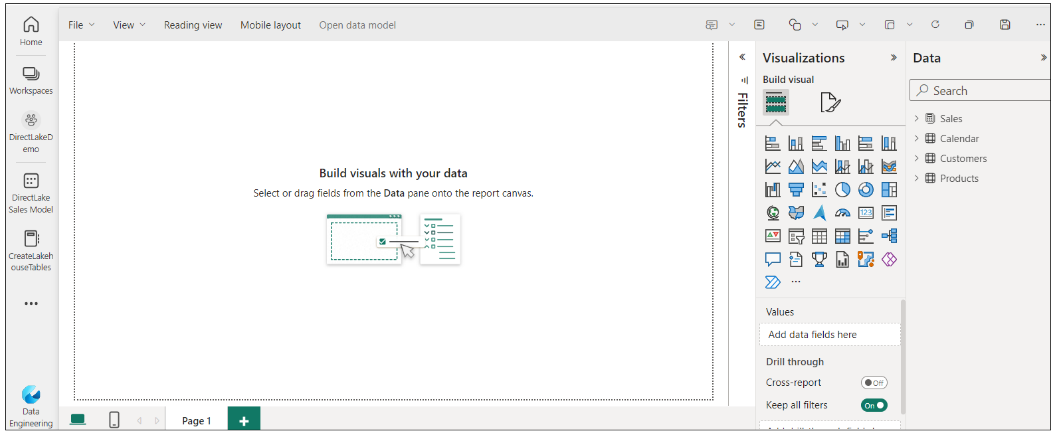
Xxxx



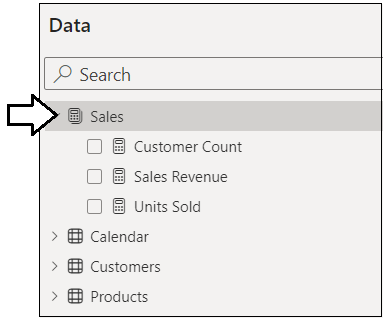
Xxxxx



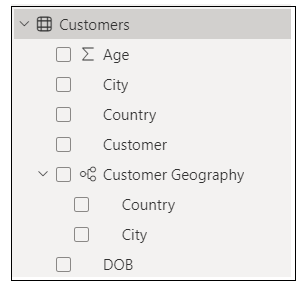
Xx



Xx



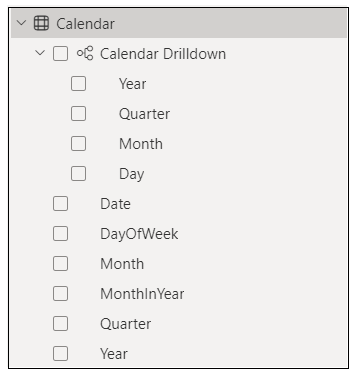
Xx



Xxx



Xxxxx



Xxxx