# Using TOM to Create a DirectLake Dataset

This repository contains C# console application named **TOM\_CreateFabricDataset** which demonstrates how to create a DirectLake data model for Fabric and Power BI using the Tabular Object Model (TOM). This repository also contains a Fabric notebook named **CreateLakehouseTables.ipynb** with Python code which must be used to create tables in a Fabric Lakehouse that will be used as the underlying datasource for the DirectLake data model.

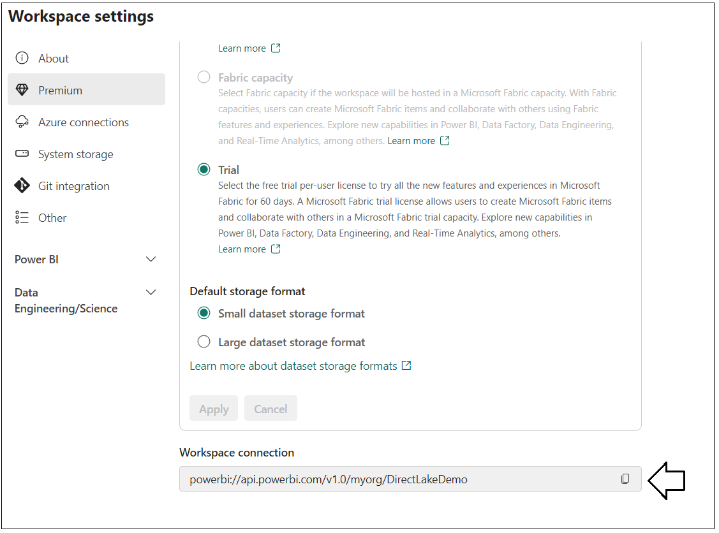
Here are the high-level steps to completing this demonstration:

* Create workspace associated with Fabric capacity
* Create a new Lakehouse in the new workspace
* Create Lakehouse tables using a pre-provided Fabric notebook
* Run the custom C# application to create DirectLake data model using TOM

## Create 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 trial capacity with Fabric capabilities.

Get URL to Workspace Connection

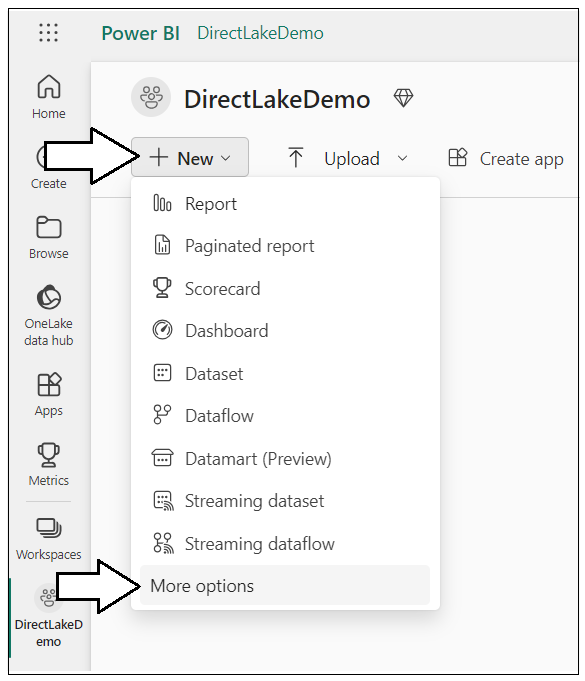


The new

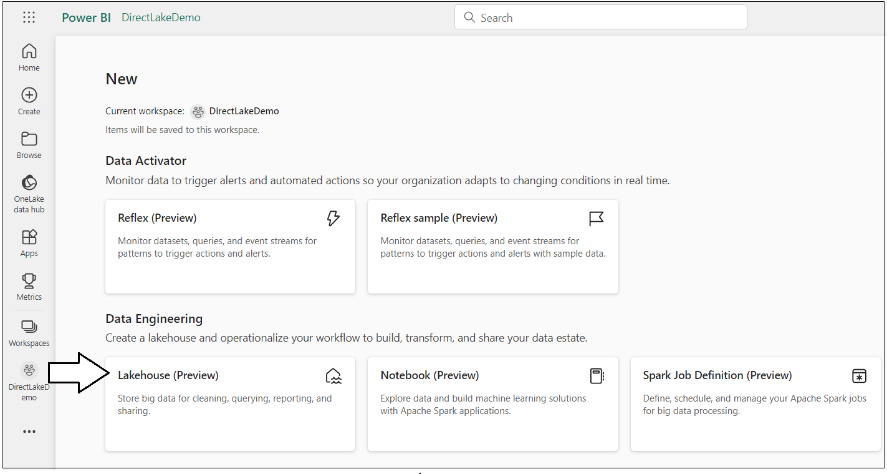
powerbi://api.powerbi.com/v1.0/myorg/DirectLakeDemo

## Create a new Lakehouse in the new workspace

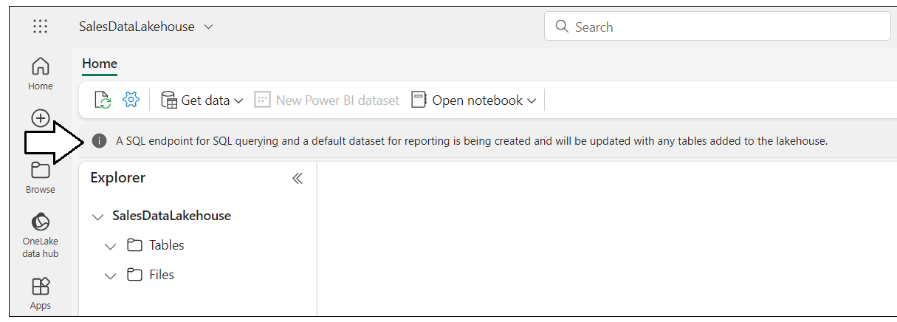
Inside the new workspace, create a new Lakehouse named **SalesDataLakehouse**.



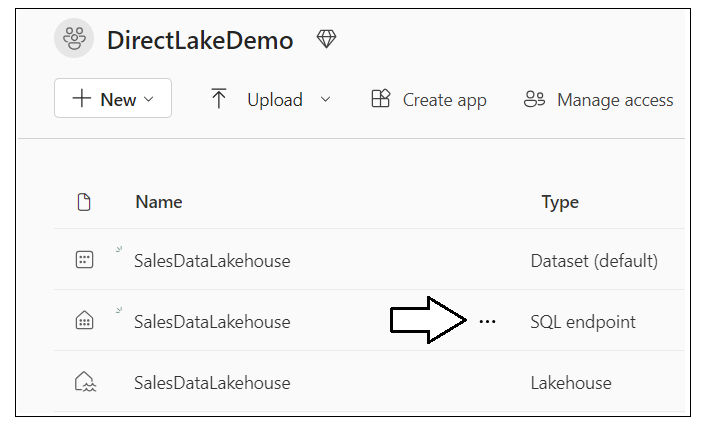
Ssss



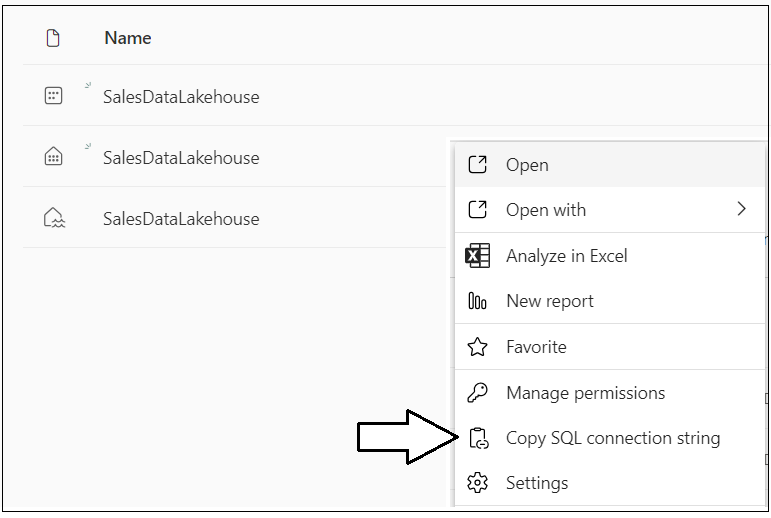
Sss



Ssss



Sssssss



Get Lakehouse SQL Endpoint

A screenshot of a computer

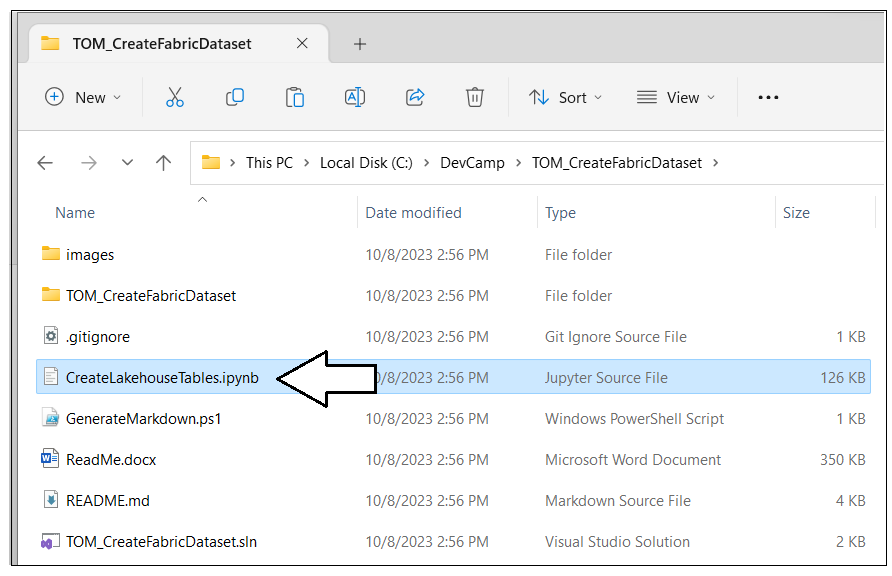
Description automatically generated

Ssss

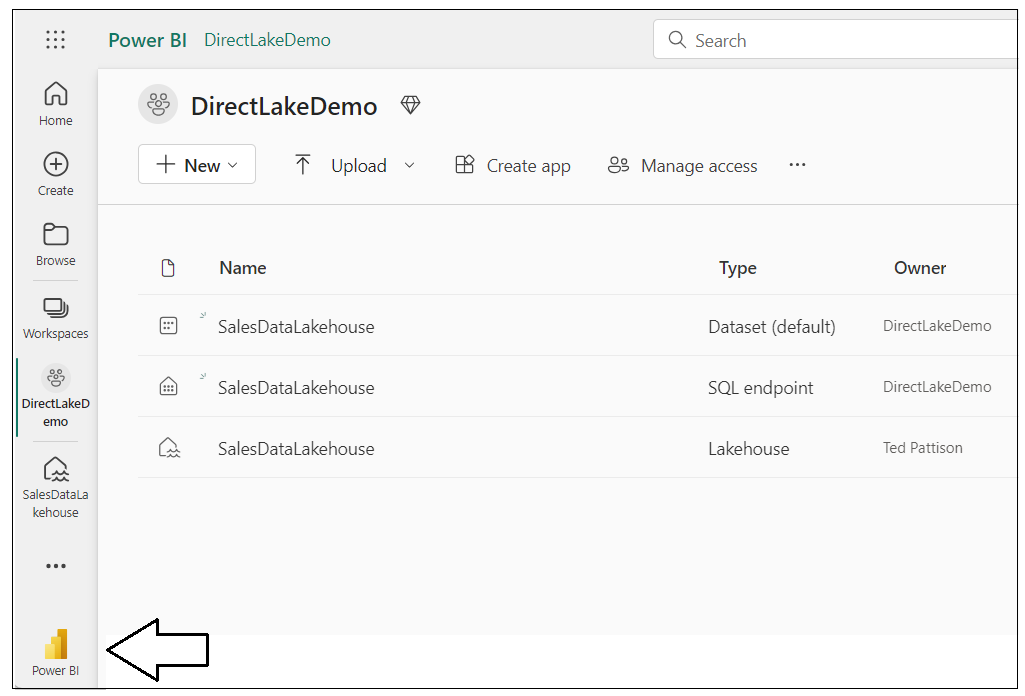
5lcsgl3vll3edero2m4sge7gdu-nya26urqtgsejoagwutwdoogl4**.datawarehouse.pbidedicated.windows.net**

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

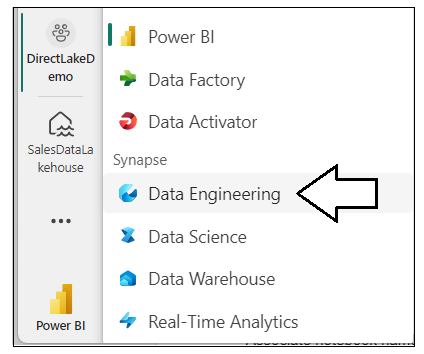
Download all the sources files from this repository as a single ZIP archive using [**this link**](https://github.com/PowerBiDevCamp/TOM_CreateFabricDataset/archive/refs/heads/main.zip). When you look inside the ZIP archive, you should see several files inside. Extract the files into a local folder on your machine.



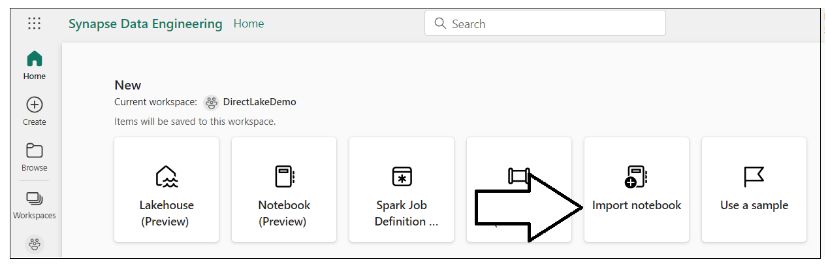
Back to Fabric UI.



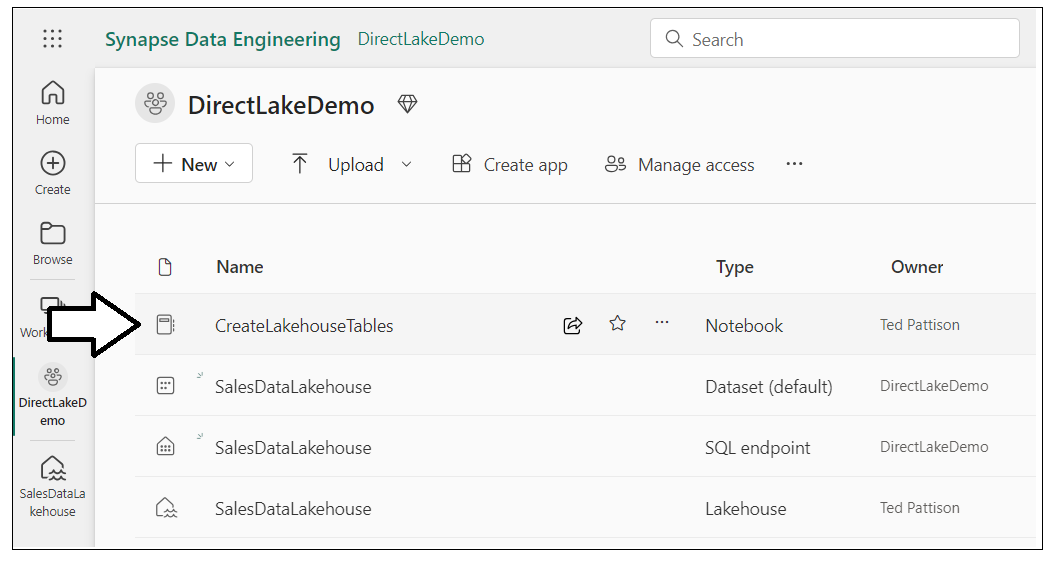
Sssss



Aaaaa

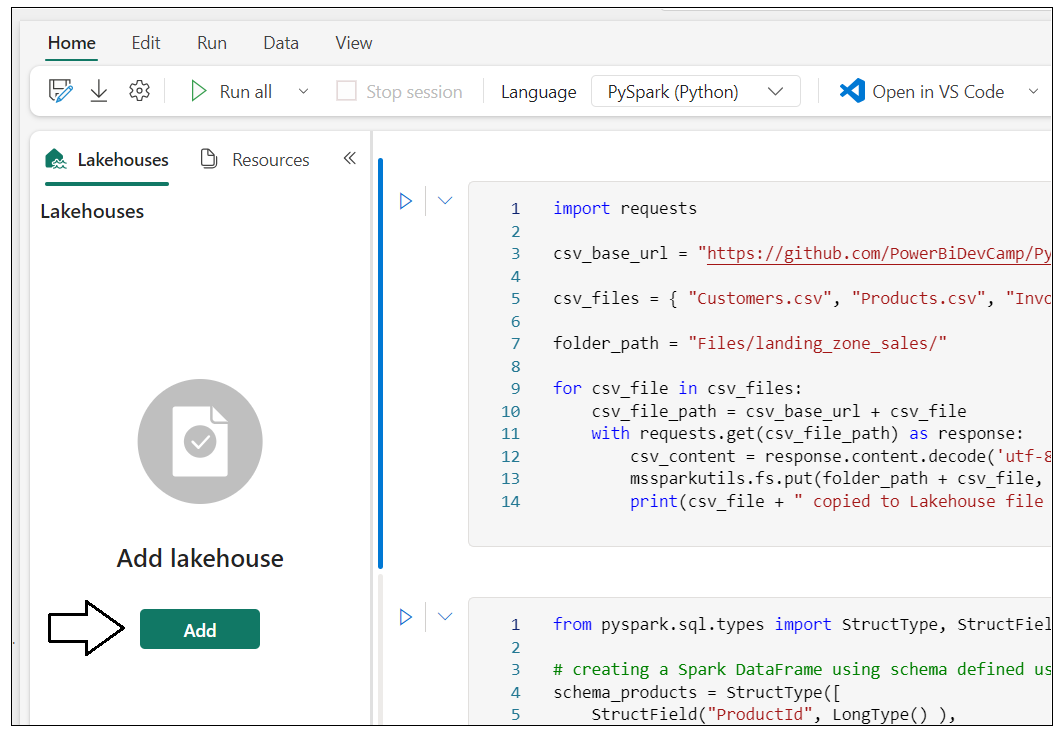


Upload Python notebook named **CreateLakehouseTables.ipynb**.

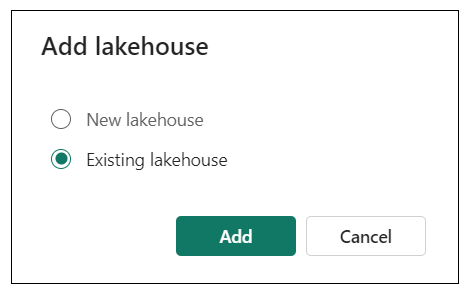


### Associate the Fabric Notebook with the Lakehouse named SalesDataLakehouse

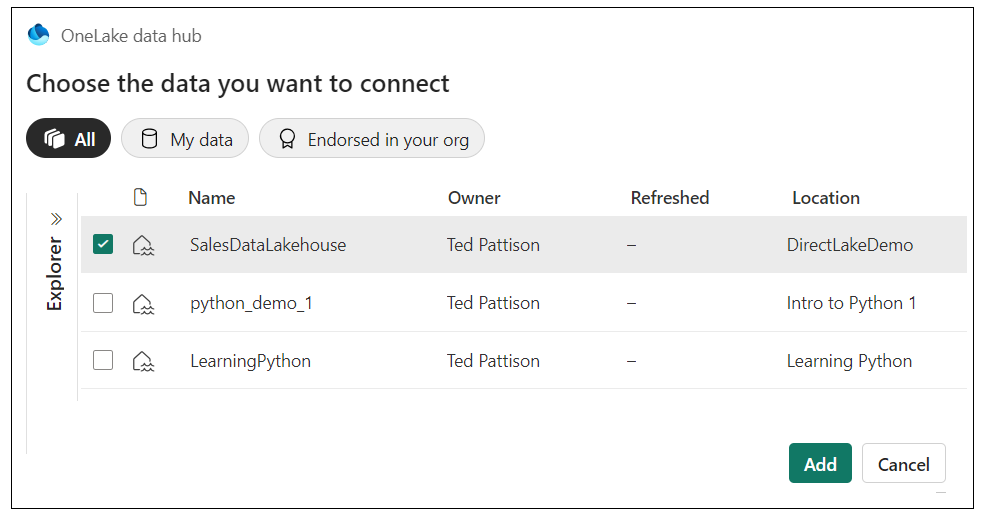
ddddd



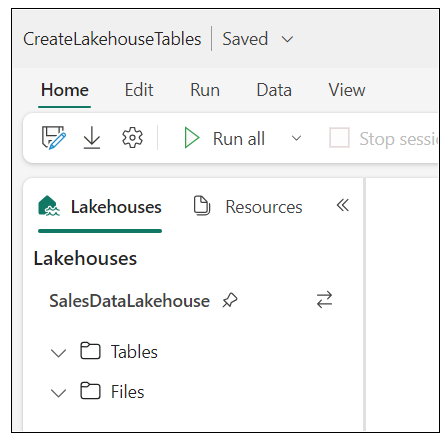
Associate notebook named **CreateLakehouseTables.ipynb** with Lakehouse



Sss



Ssss



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

Execute code in notebook to copy CSV files from GitHib repository into Lakehouse file system

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')

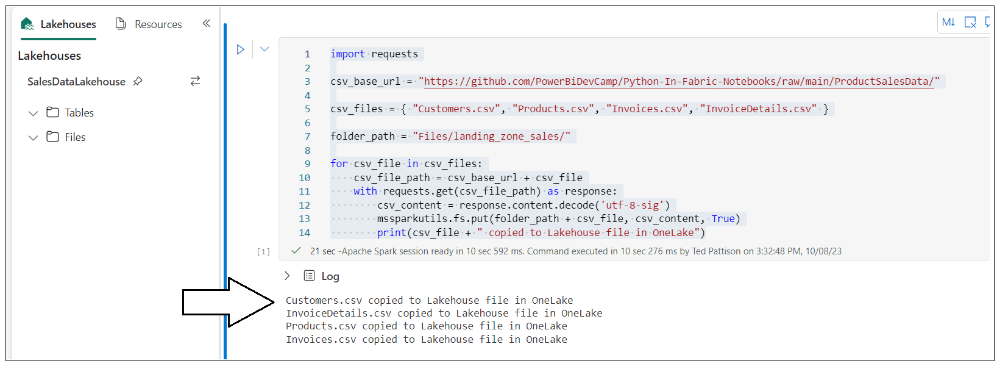
mssparkutils.fs.put(folder\_path + csv\_file, csv\_content, True)

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

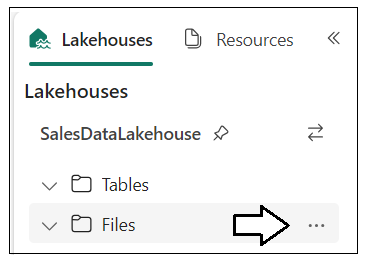
xx



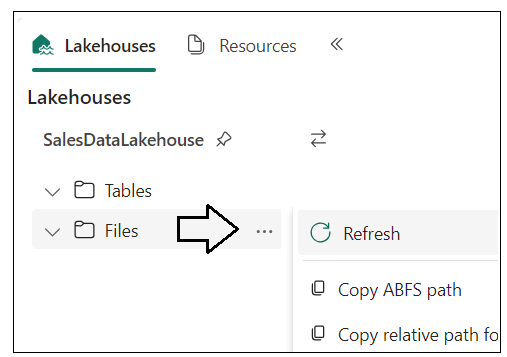
Sssss



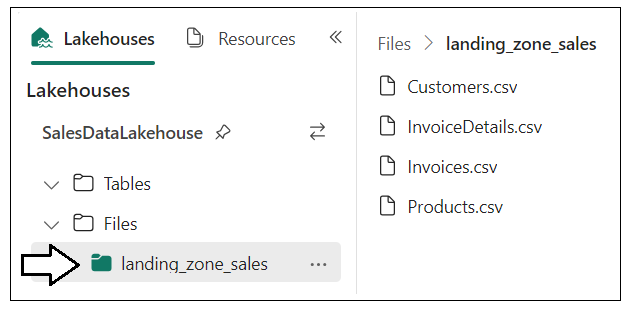
Sssss



Sss



Sss



xxx

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

Examine the code

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)

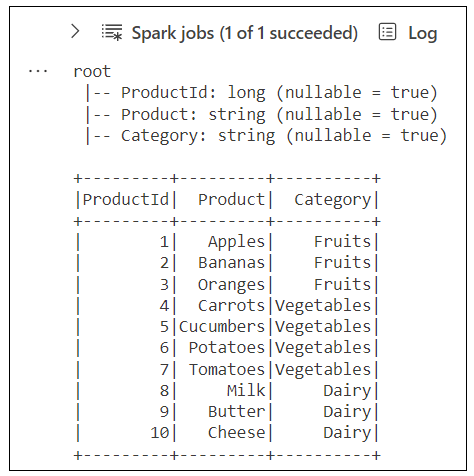
.load("Files/landing\_zone\_sales/Products.csv")

)

df\_products.printSchema()

df\_products.show()

ssssss



Xx

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")

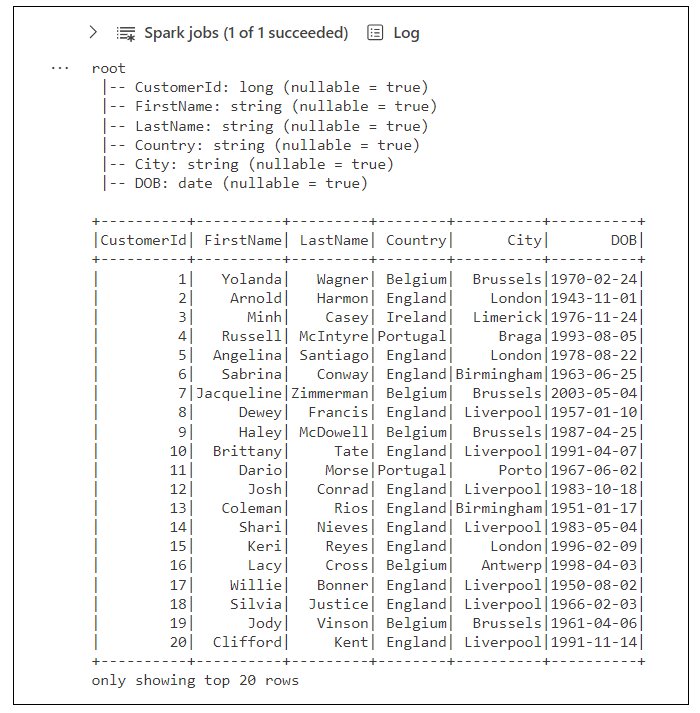
.load("Files/landing\_zone\_sales/Customers.csv")

)

df\_customers.printSchema()

df\_customers.show()

xxx



Xxx

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")

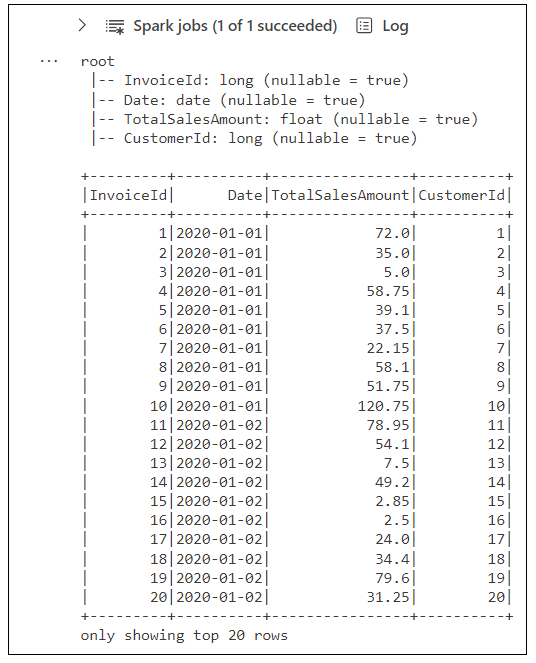
.load("Files/landing\_zone\_sales/Invoices.csv")

)

df\_invoices.printSchema()

df\_invoices.show()

xx



Xx

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")

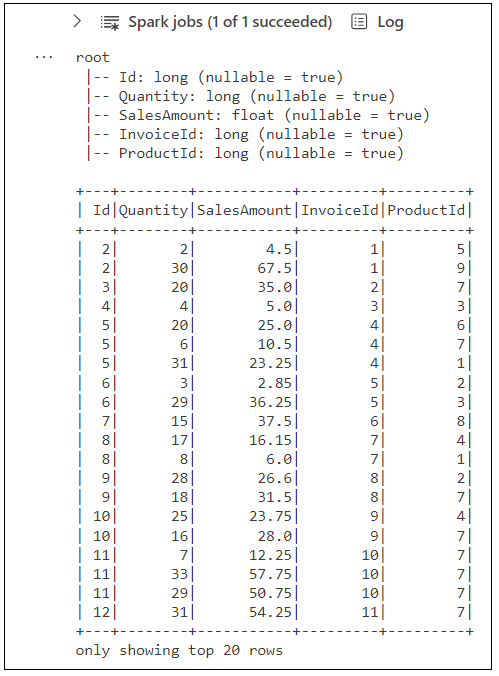
.load("Files/landing\_zone\_sales/InvoiceDetails.csv")

)

df\_invoice\_details.printSchema()

df\_invoice\_details.show()

xxx



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

Xxxx

# save all bronze layer tables

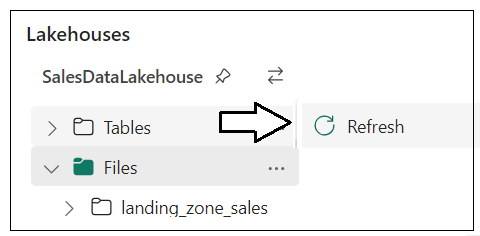
df\_products.write.mode("overwrite").format("delta").save(f"Tables/bronze\_products")

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

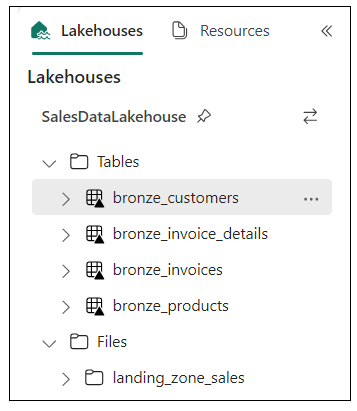
df\_invoices.write.mode("overwrite"). format("delta").save(f"Tables/bronze\_invoices")

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

xxxx



Sssss



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

Ssss

# create silver layer products table

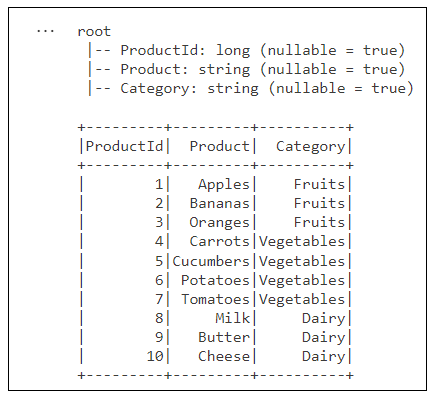
df\_silver\_products = spark.read.format("delta").load("Tables/bronze\_products")

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')

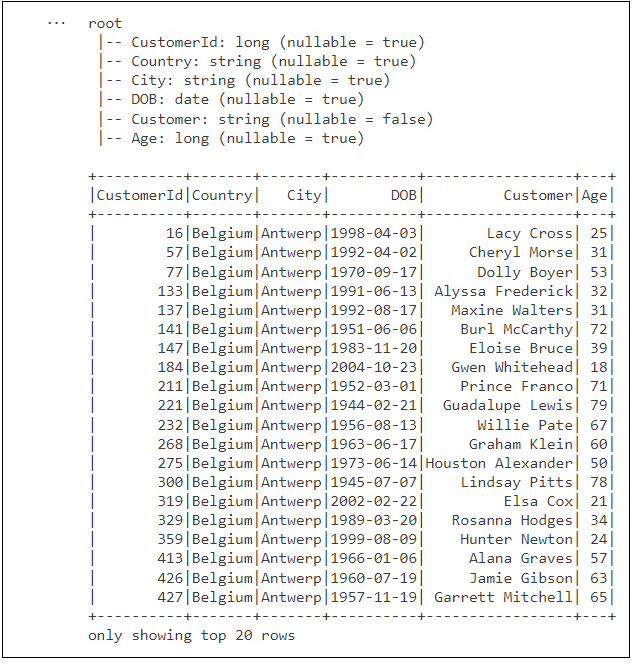
)

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")

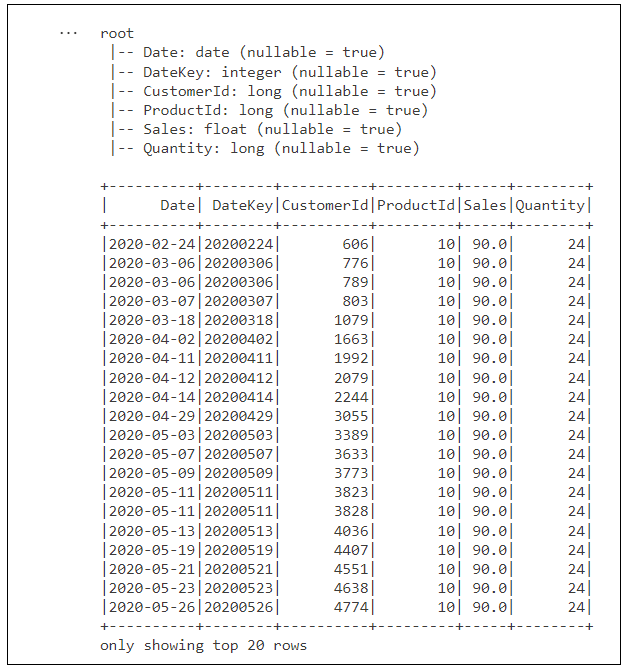
)

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')

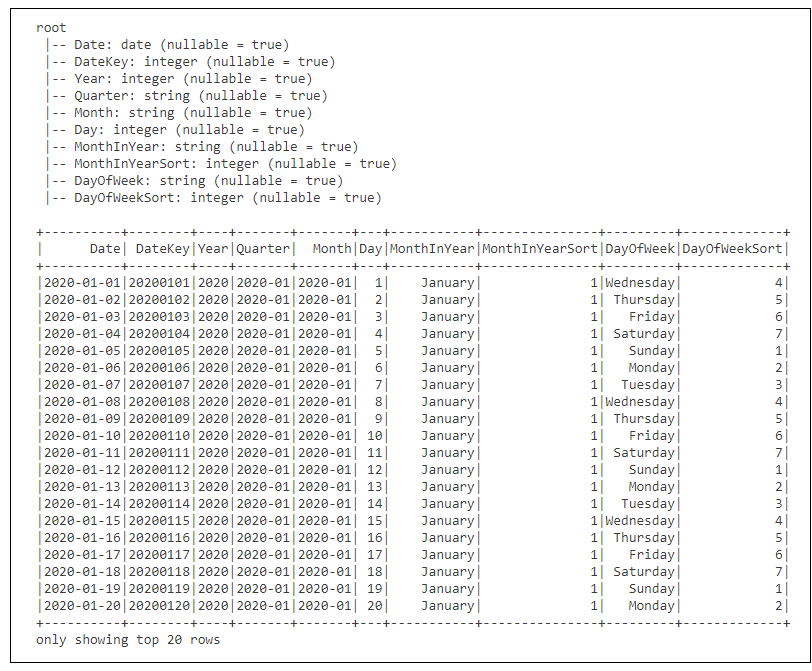
)

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

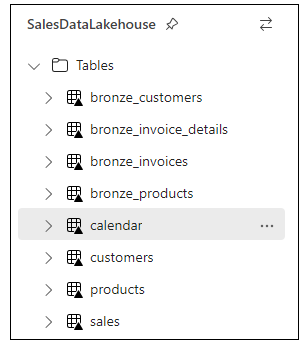
df\_calendar\_spark.printSchema()

df\_calendar\_spark.show()

xx

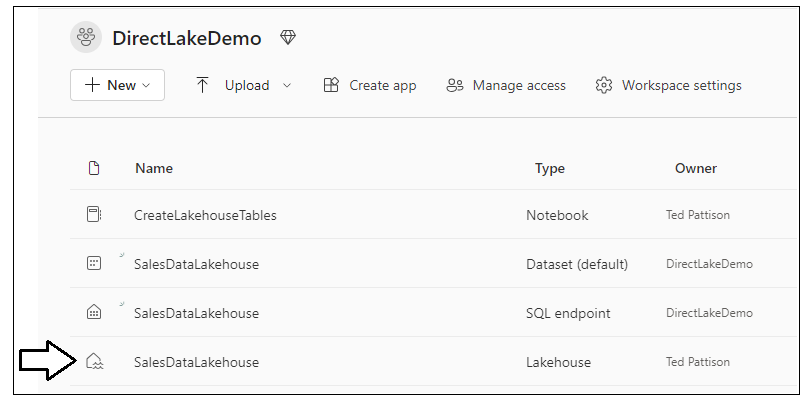


Xx

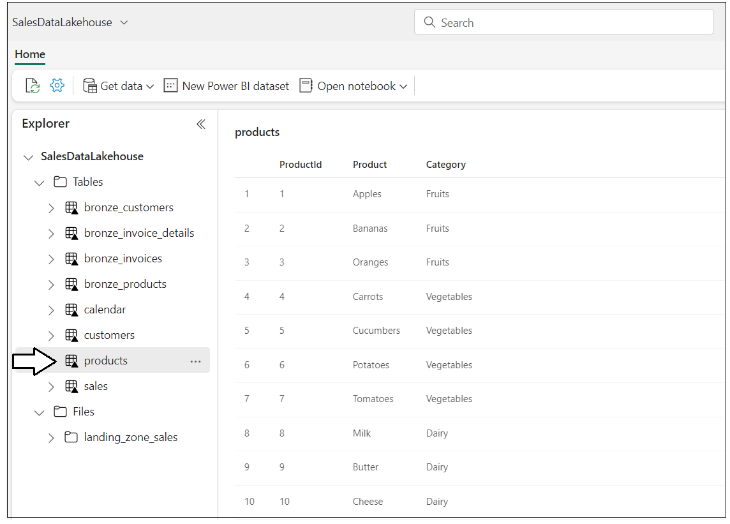


### Inspect the tables that have been created in the Lakehouse

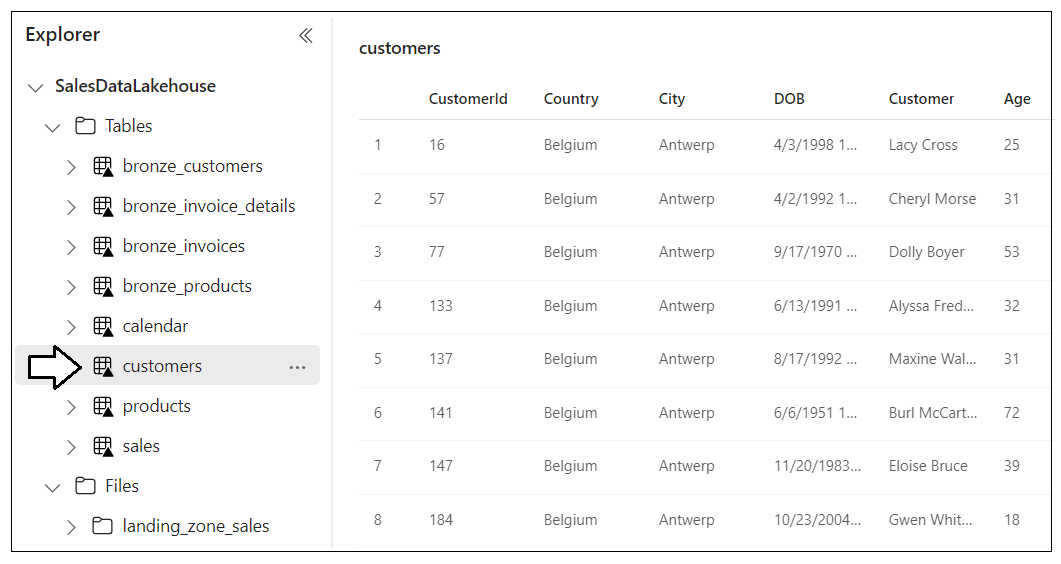
Xxxx



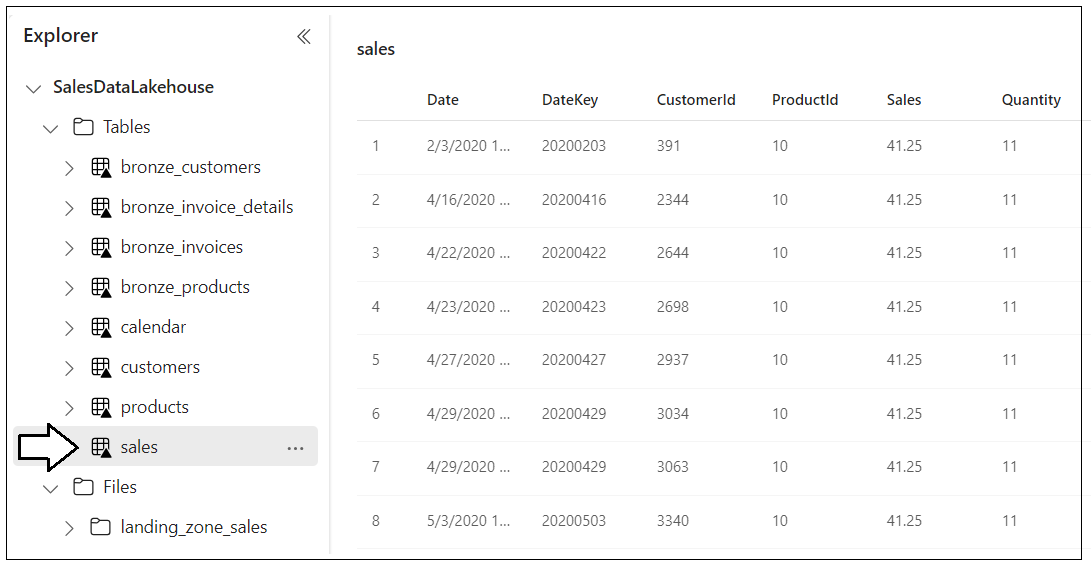
Xxxx



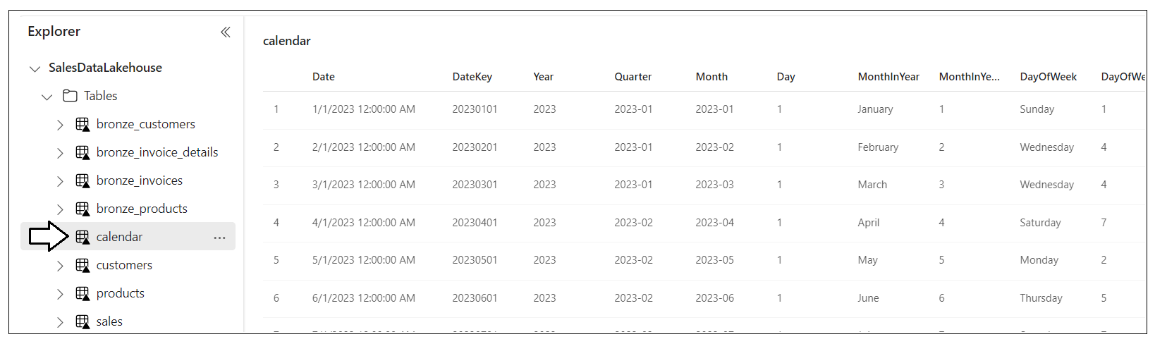
Xxx



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 Azure AD application
  1. Create a native/public application with redirect URI of <http://localhost>
  2. Record Application ID for use in console application.
* Download C# console application source code and open project in Visual Studio 2022
* Open **AppSettings.cs** and updae the following:
  1. ApplicationID of Azure AD application
  2. Workspace Connection
* SQL Endpoint
  1. Lakehouse Name
  2. UserID and Password to prevent interactive login
  3. Save changes
* Run application
  1. It should run without error
  2. When done, verify you can see new data model and use it to create new report