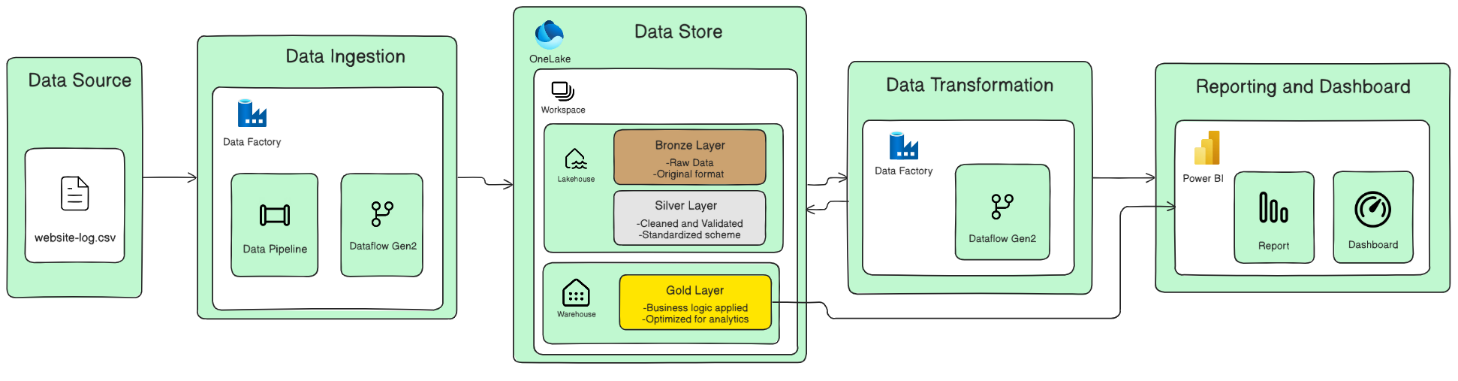
**Simple Website Analytics Dashboard**

**Problem Statement:** A small business wants to track website traffic and user engagement to understand which pages are most popular and how users are navigating their site.

**Objective:**

1. Monitor website traffic pattern
2. Identify most visited pages
3. Understand user navigation path

**Solution Overview:** Build a basic analytics pipeline in Microsoft Fabric to ingest website log data, transform it, and visualize key metrics in a Power BI dashboard.

**Architecture: **

Just for reference actual architecture might me little different.

We’ll now work on a multi-tenant website analytics dashboard with the following features:

1. Dynamic ingestion of website logs with tenant specific (simulate data accordingly)
2. Schema and row count validation
3. Centralized logging per tenant and step
4. Power BI dashboard showing traffic trends, navigation paths, session times, and peak hours according to tenant.
5. **Data Curation:**

Since multi-tenant website log data is hard to find. I have generated dataset from co-pilot with similar table structure containing the following columns/fields:

* Session ID
* Tenant
* IP Address
* Time Stamp
* Http Method
* URI/Page Path
* Http Status
* Bytes Sent
* Referrer/Previous Page
* User Agent/Browser Information

The datasets are different and separate for both tenants. We have the dataset of both the tenants as a csv file. Using a python script I have inserted it in my on-premises MySQL database.

Table Schema:

CREATE TABLE tenantb (

`Session ID` VARCHAR(255),

`Tenant` VARCHAR(255),

`IP Address` VARCHAR(45),

`Time Stamp` DATETIME,

`Http Method` VARCHAR(10),

`URI/Page Path` TEXT,

`Http Status` INT,

`Bytes Sent` INT,

`Referrer/Previous Page` TEXT,

`User Agent/Browser Information` TEXT

);

1. **Data Architecture:** We will be using a similar kind of data storage architecture for the second phase as well.

We will have a few changes on the Data Pipeline Activities that we use.

1. **Fabric Implementation:**

Create a Workspace (project\_secondphase)

Create a Lakehouse for Broze and Silver layer. (MyLakehouse)

Create a Warehouse for Gold layer. (Warehouse)

Create a Data Pipeline (Data Ingestion) for the initial data loading from the source which in our case is MySQL database.

Since we want an incremental loading for our usecase we will create a Registry/Watermark table in our warehouse.  
Control\_Ingestion: table\_name , last\_run\_timestamp

In our pipeline we will add a lookup activity (GetTableList) to get the list of tables.

SELECT table\_name FROM Control\_Ingestion;

We will connect the LookUp to a ForEach activity [@activity('GetTableList').output.value]

So our forEach activity will run for each of the table.

Inside our for each we have another LookUp activity (GetTime).

SELECT last\_run\_timestamp FROM Control\_Ingestion WHERE table\_name = '@{item().table\_name}'

Next for Column level Validation:

Connect it to a Copy Data Activity with source as the database Query it to get only the column names and destination as a schema table(table\_name\_schema).

@{trim(concat(item().table\_name, '\_schema'))}

We will use a Notebook activity to validate the schema from aur database that is stored as a table and the existing table in our lakehouse.

We have two parameters: table\_name (@item().table\_name) and

Schematable\_name (@{trim(concat(item().table\_name, '\_schema'))})

The Notebook (Data Validation) Script is:

table\_name = "tenanta"

schematable\_name = "tenanta\_schema"

from notebookutils import mssparkutils

df = spark.table(table\_name)

df2 = spark.table(schematable\_name)

df\_column\_names = set(df.columns)

df2\_column\_names = set(row['column\_name'] for row in df2.select("column\_name").collect())

# Compare and return result

match\_result = df\_column\_names == df2\_column\_names

# Exit the notebook with the result as a string

mssparkutils.notebook.exit(str(match\_result))

We will connect it to an If Condition which will take the exit value as the condition.

If the exit value is true (Schema matches):

Then we will use a copy data activity to incrementally load data using the Registry table in our warehouse and the getTime lookup we used earlier.

Source: SELECT \* FROM `@{item().table\_name}`

WHERE Time\_stamp > '@{activity('GetTime').output.value[0].last\_run\_timestamp}'

For the destination we will have @{item().table\_name} with append action.

After copy data we will use a script (UpdateTime) to update the time in the registry table.

UPDATE Control\_Ingestion

SET last\_run\_timestamp = '@{utcNow()}'

WHERE table\_name = '@{item().table\_name}'

If the exit value is false (schema mismatch):

We will use an outlook activity to send a mail to notify about the schema mismatch.

Our Initial Data Ingestion is now done.

Next for Data cleaning we will use a Dataflow Gen2 to clean and remove the noise from our dataset.

We will have two sources: tenanta and tenantb

Cleaning transformations:

Remove duplicates

Filter Httpstatus column row to have only 200

Remove noise from url and referrer where if it contains ‘?v’ and something then we remove it from the url or referrer

Table.AddColumn(#"Added custom", "Custom", each Text.BeforeDelimiter([Url], "?"))

Then we will select columns and have only those with use.

Table.SelectColumns(#"Added custom 1", {"Session\_ID", "Tenant", "IP\_Address", "Time\_stamp", "UserAgent", "Referrer new", "Custom"})

Column renaming:

Table.RenameColumns(#"Choose columns", {{"Referrer new", "Referrer"}, {"Custom", "URI"}})

We will perform these steps in both the tables and then we will append them to get a single Silver\_sessionTable.

Create a Data Pipeline (Data Cleaning) and add the dataflow in the pipeline for scheduling and orchestration.

Now we have our silver level data ready.

For Gold layer we will create a Data Pipeline (Data Enrichment):

Add a notebook (Data Enrich):

**#1.** We will read the silver layer data int a data frame and add custom column of date and hour

df = spark.read.table("Silver\_SessionData")

from pyspark.sql.functions import to\_date, hour

df = df.withColumn("Date", to\_date("Time\_stamp"))

df = df.withColumn("Hour", hour("Time\_stamp"))

**#2. We will use Data Wrangler for traffic\_trends table:**

# Code generated by Data Wrangler for PySpark DataFrame

from pyspark.sql import functions as F

def traffic\_trends(df):

    # Performed 1 aggregation grouped on columns: 'URI', 'Tenant'

    df = df.groupBy('URI', 'Tenant').agg(F.count('URI').alias('URI\_count'))

    df = df.dropna()

    df = df.sort(df['URI'].asc(), df['Tenant'].asc())

    return df

df\_traffic\_trends = traffic\_trends(df)

df\_traffic\_trends.write.mode("overwrite").saveAsTable("Gold\_Traffic\_trends")

**#3. Navigation Path**

# Code generated by Data Wrangler for PySpark DataFrame

from pyspark.sql import functions as F

def Navigation\_path(df):

    # Remove rows where Referrer is null, empty, or NaN

    df = df.filter(

        (F.col('Referrer').isNotNull()) &

        (F.col('Referrer') != '') &

        (~F.isnan(F.col('Referrer')))

    )

    # Perform aggregation

    df = df.groupBy('Tenant', 'URI', 'Referrer').agg(F.count('IP\_Address').alias('IP\_Address\_count'))

    # Sort the DataFrame

    df = df.sort(df['Tenant'].asc(), df['URI'].asc(), df['Referrer'].asc())

    return df

df\_Nav\_Path = Navigation\_path(df)

df\_Nav\_Path.write.mode("overwrite").saveAsTable("Gold\_Navigation\_Path")

**#4. Session\_time:**

import pandas as pd

from pyspark.sql import functions as F

def Session\_time(df):

    # Sort by columns: 'Tenant', 'Session\_ID', 'Time\_stamp'

    df = df.sort(df['Tenant'].asc(), df['Session\_ID'].asc(), df['Time\_stamp'].asc())

    # Group by 'Tenant' and 'Session\_ID' and get first and last timestamps

    df = df.groupBy('Tenant', 'Session\_ID').agg(

        F.last('Time\_stamp').alias('Time\_stamp\_last'),

        F.first('Time\_stamp').alias('Time\_stamp\_first')

    )

    df = df.dropna()

    pandas\_df = df.toPandas()

    pandas\_df['Time\_stamp\_last'] = pd.to\_datetime(pandas\_df['Time\_stamp\_last'])

    pandas\_df['Time\_stamp\_first'] = pd.to\_datetime(pandas\_df['Time\_stamp\_first'])

    pandas\_df['Session\_Time'] = (pandas\_df['Time\_stamp\_last'] - pandas\_df['Time\_stamp\_first']).dt.total\_seconds(

    # Filter out sessions longer than 30 minutes (1800 seconds)

    pandas\_df = pandas\_df[pandas\_df['Session\_Time'] <= 1800]

    return pandas\_df

df\_Session\_time = Session\_time(df)

spark\_df\_Session\_time = spark.createDataFrame(df\_Session\_time)

# Overwrite the existing table with updated schema and data

spark\_df\_Session\_time.write.mode("overwrite").option("overwriteSchema", "true").saveAsTable("Gold\_AvgSession\_time")

**#5. Peak\_hours**  
from pyspark.sql import functions as F

def Peak\_hours(df):

    df = df.dropna()

    # Convert integer hour to time string and rename column

    df = df.withColumn('Formatted\_Hour', F.date\_format(F.to\_timestamp(df['Hour'].cast('string'), 'H'), 'h:mm a'))

    # Drop original Hour column

    df = df.drop('Hour')

    # Rename formatted column to Hour

    df = df.withColumnRenamed('Formatted\_Hour', 'Hour')

    # Sort by Hour and Tenant

    df = df.sort(df['Hour'].asc(), df['Tenant'].asc())

    return df

df\_Peak\_hours = Peak\_hours(df)

df\_Peak\_hours.write.mode("overwrite").option("overwriteSchema", "true").saveAsTable("Gold\_Peak\_hours")

Connect the Notebook to SetVariable and create a variable named Tables: ["gold\_traffic\_trends","gold\_navigation\_path","gold\_avgsession\_time","gold\_peak\_hours"]

Hardcoded variable for gold layer table names stored in our lakehouse

Add a ForEach statement and use the variable to run it for each of the tables and use copy data activity to copy the gold tables to warehouse.

In the warehouse

Create a dimension table using SQL Query  
CREATE TABLE Gold.Tenant (

    Tenant VARCHAR(255),

    TenantName VARCHAR(255)

);

INSERT into Gold.Tenant VALUES ('TenantA','A'),('TenantB','B');

create a semantic model with all the gold layer tables and create a report.

Use Silcer for a Select tenant drop down.

And separate charts for each of the tables.

Create hierarchy in peak\_hours for drill down implementations.

Also create a slicer for date filter in peak hours

In the navigation\_path:

X axis: Referrer and URI (automatically creates hierarchy and drill down implemented)

Y axis: IP address count

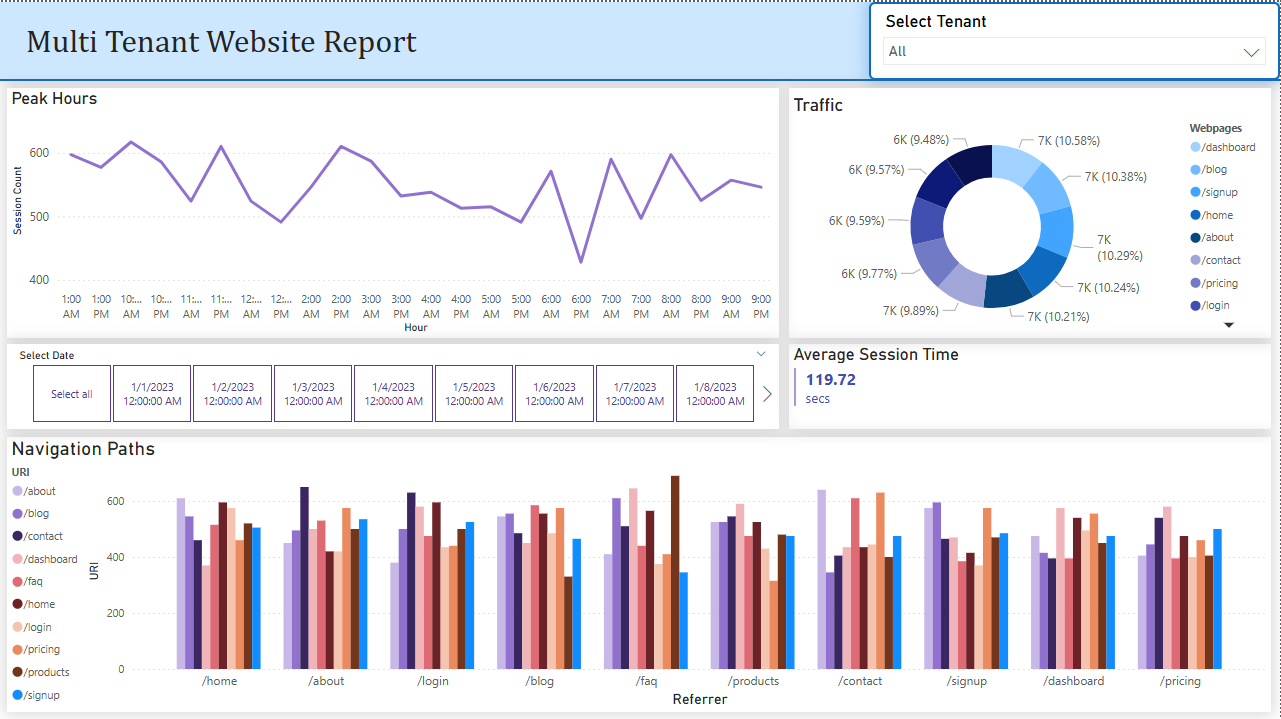
Or

X axis: referrer

Y axis: IP address count

Legend: URI

For clustered column chart



Final report