

BRAVISA MUTUAL FUND

LAB MANUAL

Bravisa Mutual Fund

INTRODUCTION

3

INSTALL POSTGRESQL ON WINDOWS

DATABASE CREATION

SCHEMAS CREATION

TABLE CREATION

Gui.py

- I. Insert Data
- II. Delete Data
- III. Start Process
- IV. Download Data

Main.py

- I. Import libraries
- II. run_scripts
- III.BTT
- IV BTT
- V FB insert on linux
- VI FB Insert
- VII FB History Insert
- VIII OHLC Insert

INTRODUCTION

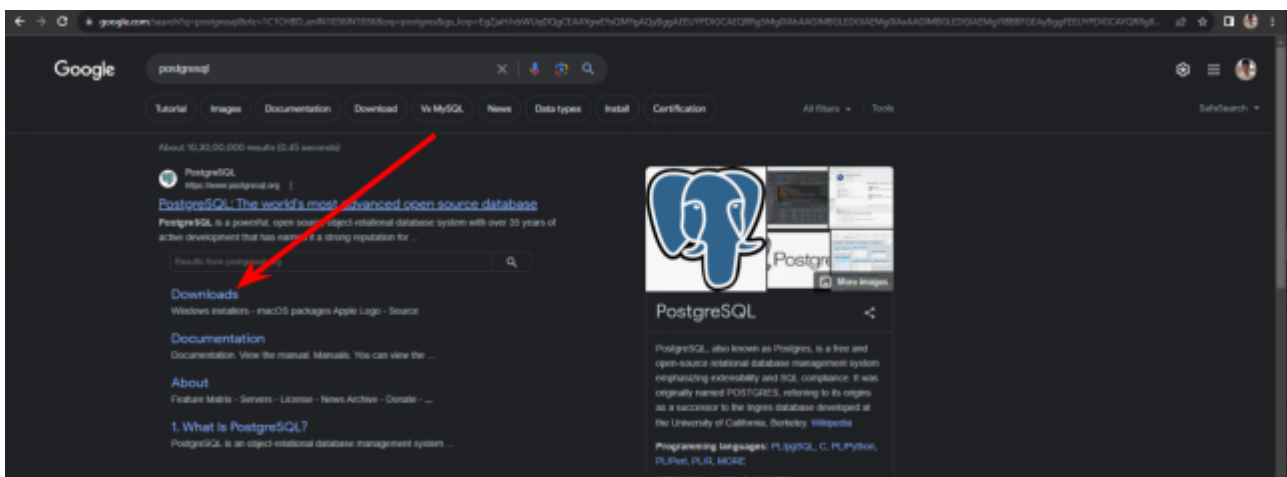
INSTALL POSTGRESQL ON WINDOWS:

There are three crucial steps for the installation of PostgreSQL as follows:

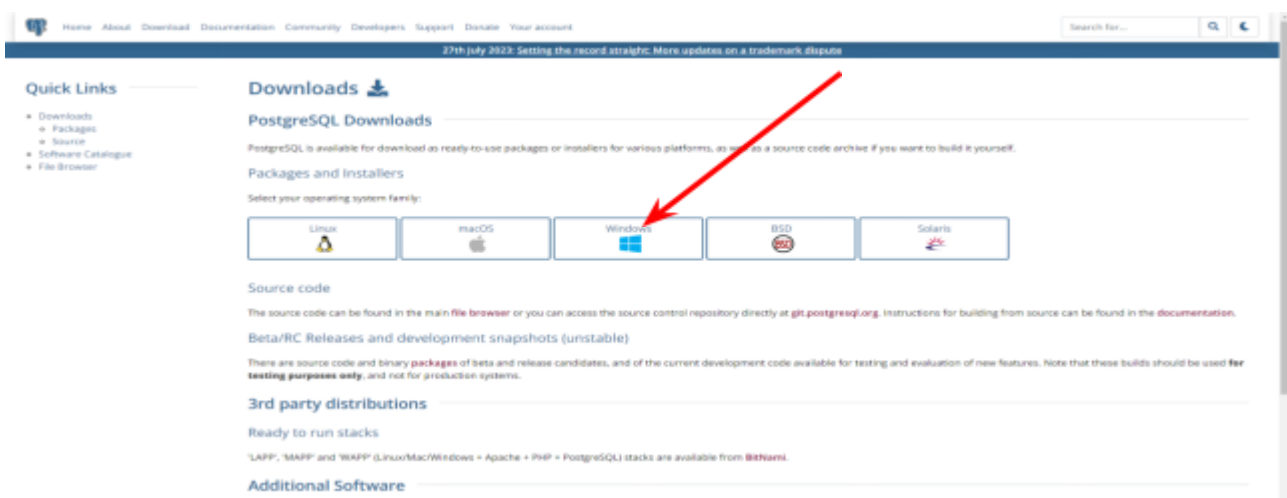
1. Download PostgreSQL installer for Windows
2. Install PostgreSQL
3. Verify the installation

1. Download PostgreSQL installer for Windows

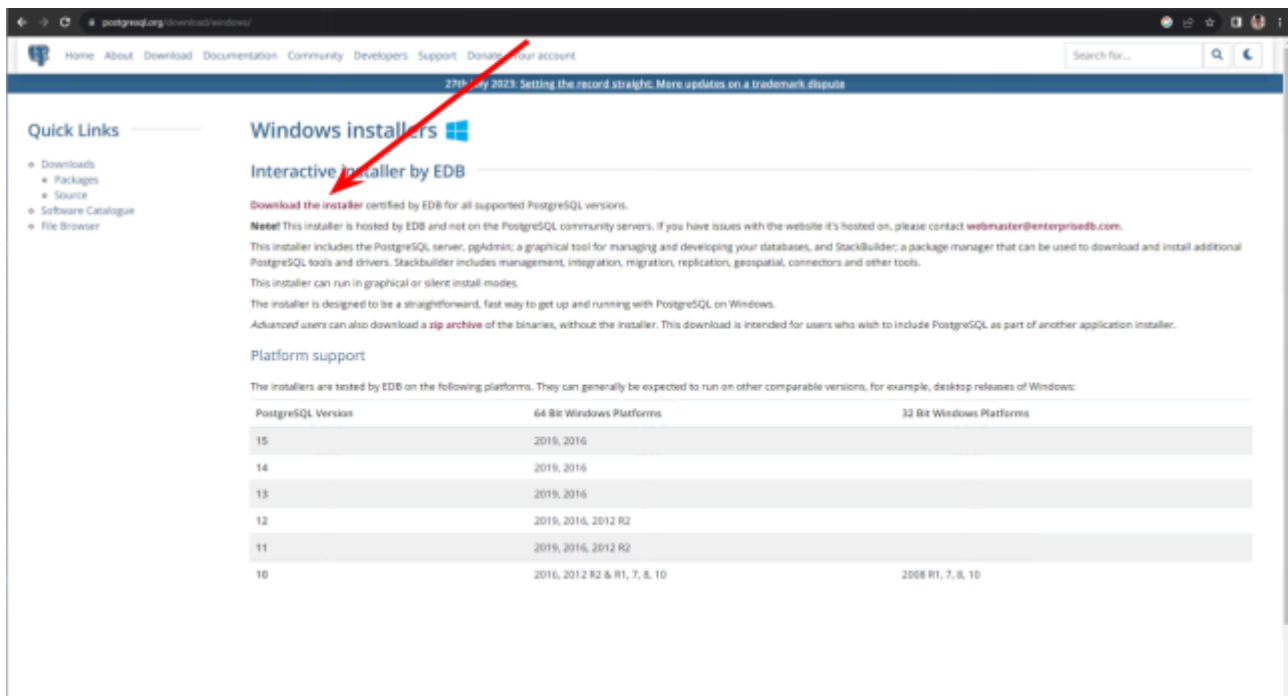
Step 1: Search PostgreSQL on Google and click on downloads.



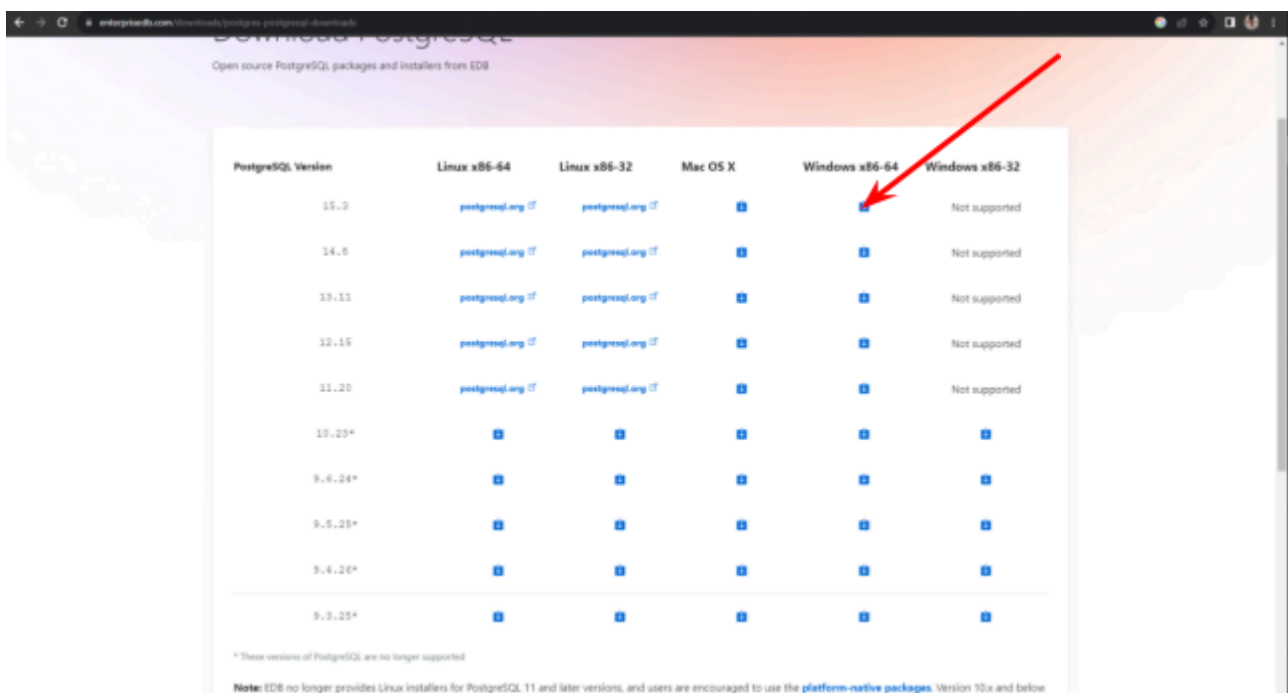
Step 2: Click on Windows button



Step 3: Click on Download the installer button



Step 4: Click on 15.3 version on Windows x86-64 (chosed based on system configuration)

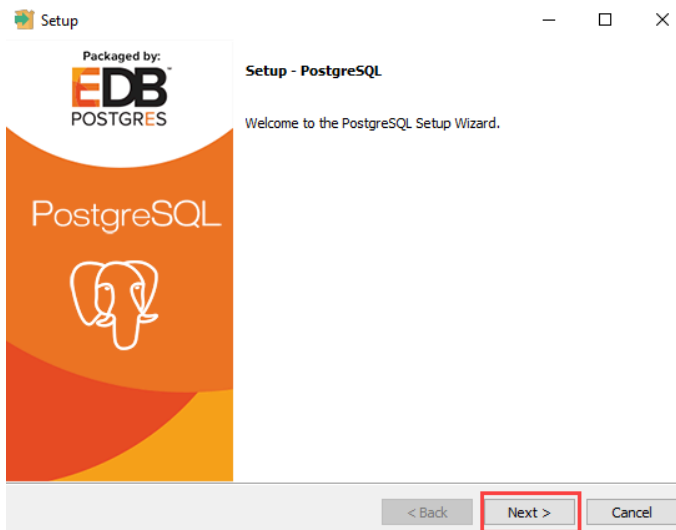


PostgreSQL installer for Windows is downloaded and make sure is downloaded check in downloads.

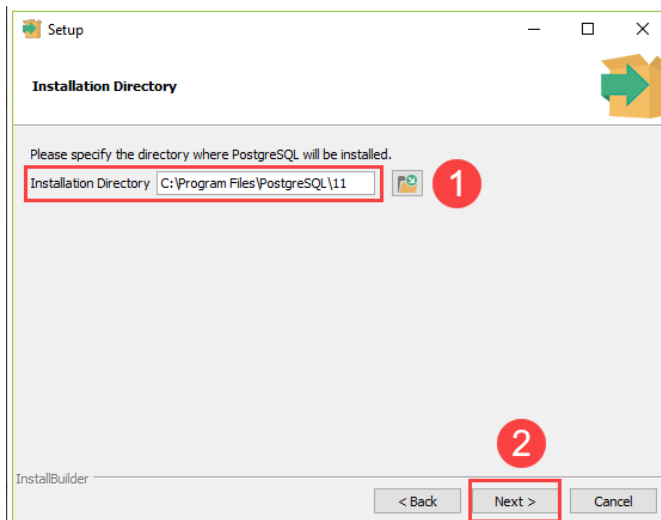
2. Install PostgreSQL

Step 1: After downloading the installer double click on it

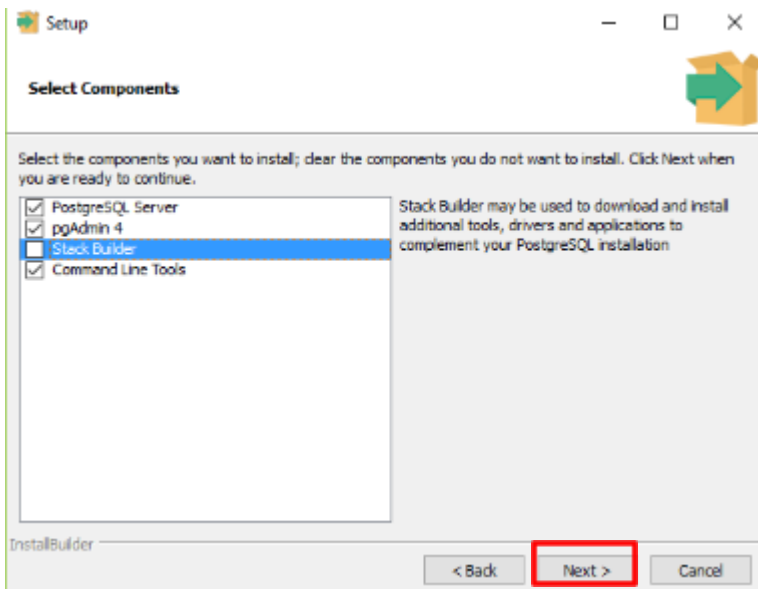
Step 2: Click the Next button



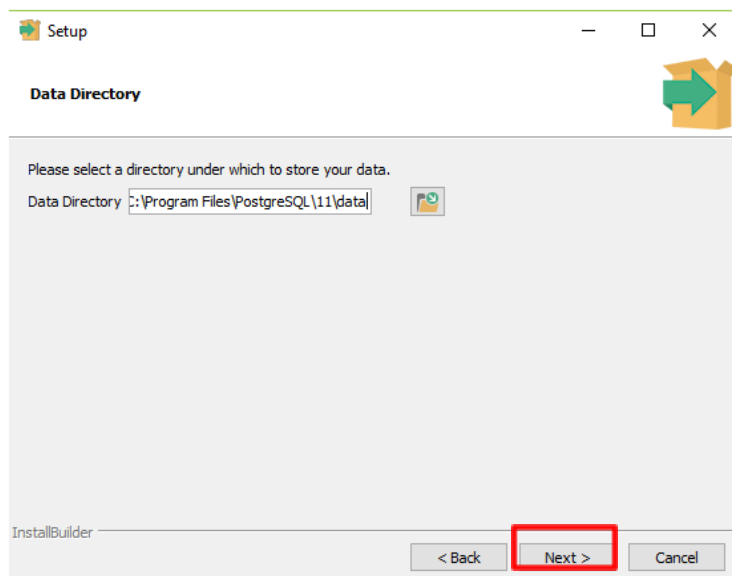
Step 3: Choose the installation folder, where you want PostgreSQL to be installed, and click on Next.



Step 4: Select the components as per your requirement to install and click the Next button.



Step 5: Select the database directory where you want to store the data and click on Next.



Step 6: Set the password for the database superuser (Postgres)

Setup

Password

Please provide a password for the database superuser (postgres).

Password:

Retype password:

InstallBuilder

< Back **Next >** Cancel

Step 7: Set the port for PostgreSQL. Make sure that no other applications are using this port. If unsure leave it to its default (5432) and click on Next.

Setup

Port

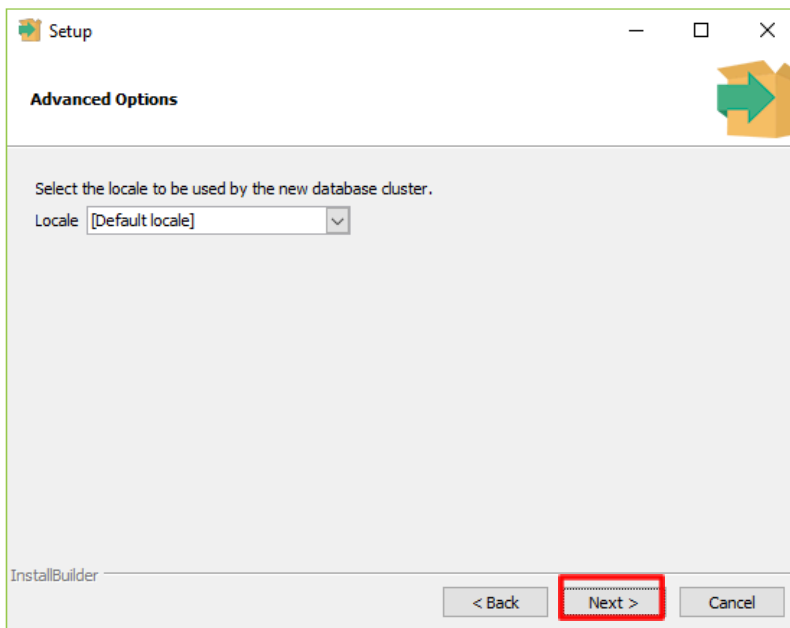
Please select the port number the server should listen on.

Port:

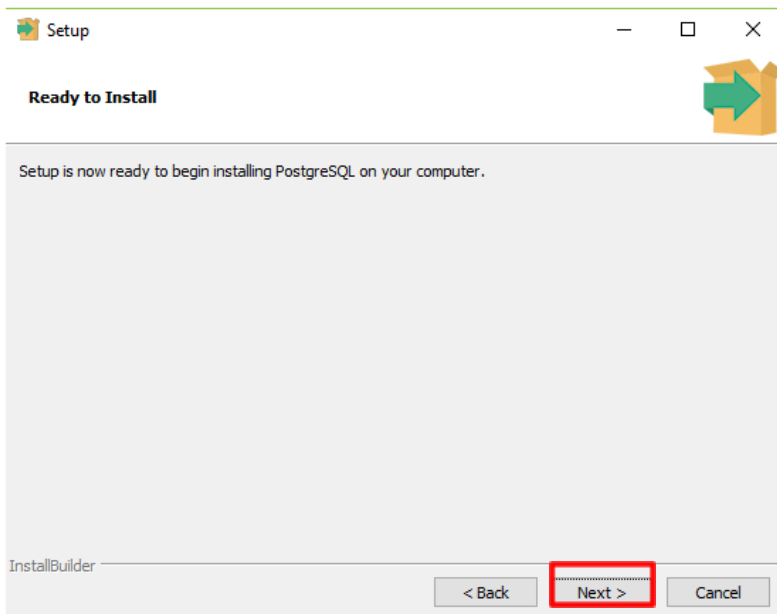
InstallBuilder

< Back **Next >** Cancel

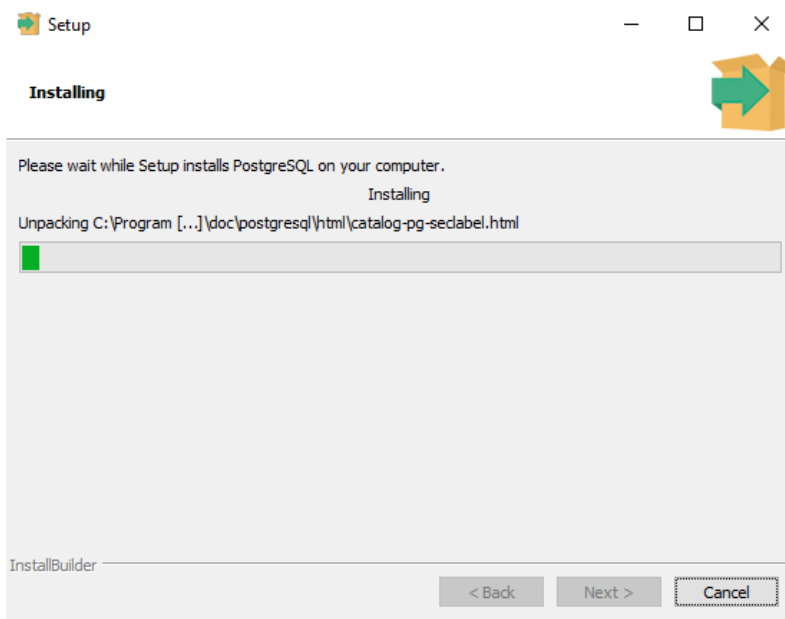
Step 8: Choose the default locale used by the database and click the Next button.



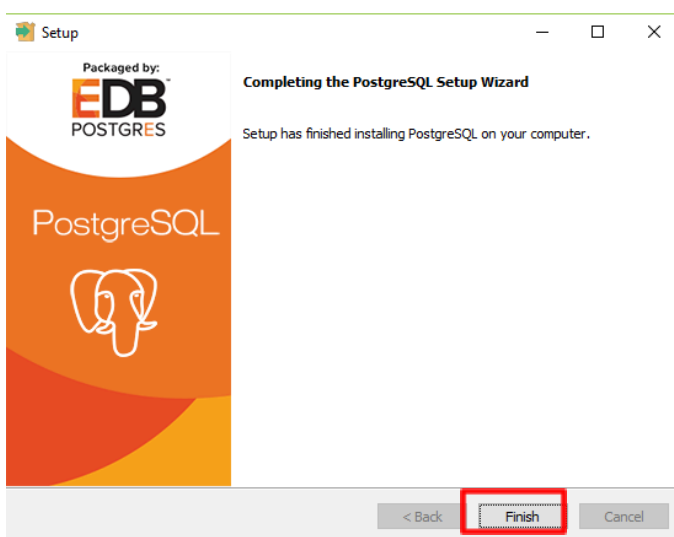
Step 9: Click the Next button to start the installation.



- Wait for the installation to complete, it might take a few minutes.



Step 10: Click the Finish button to complete the PostgreSQL installation.



3. Verify the installation

Verifying the Installation of PostgreSQL

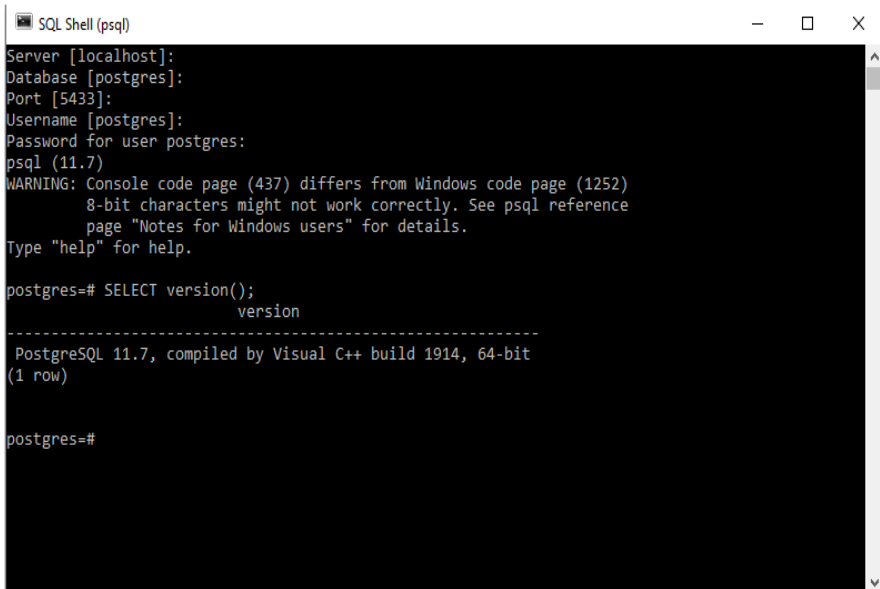
There are couple of ways to verify the installation of PostgreSQL like connecting to the database server using some client applications like pgAdmin or psql.

The quickest way though is to use the psql shell. For that follow the below steps:

Step 1: Search for the psql shell in the windows search bar and open it.

Step 2: Enter all the necessary information like the server, database, port, username, and password and press Enter.

Step 3: Use the command `SELECT version();` you will see the following result:



```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5433]:
Username [postgres]:
Password for user postgres:
psql (11.7)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

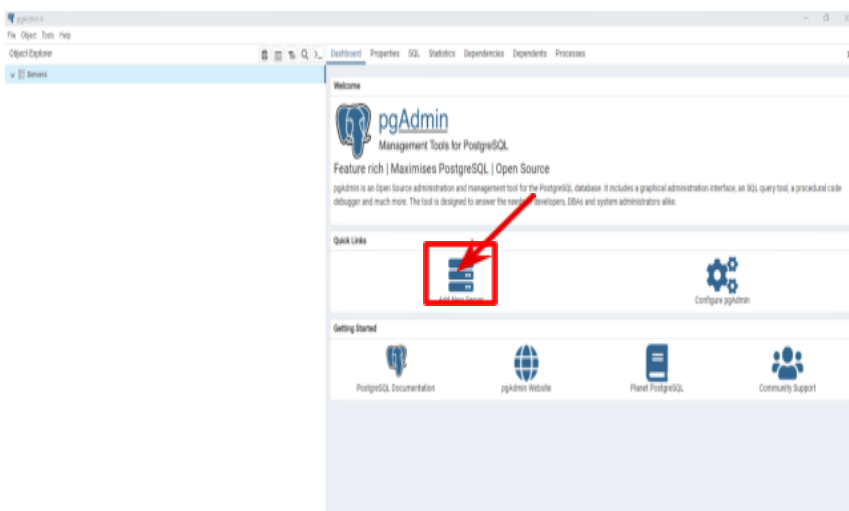
postgres=# SELECT version();
              version
-----
 PostgreSQL 11.7, compiled by Visual C++ build 1914, 64-bit
(1 row)

postgres=#
```

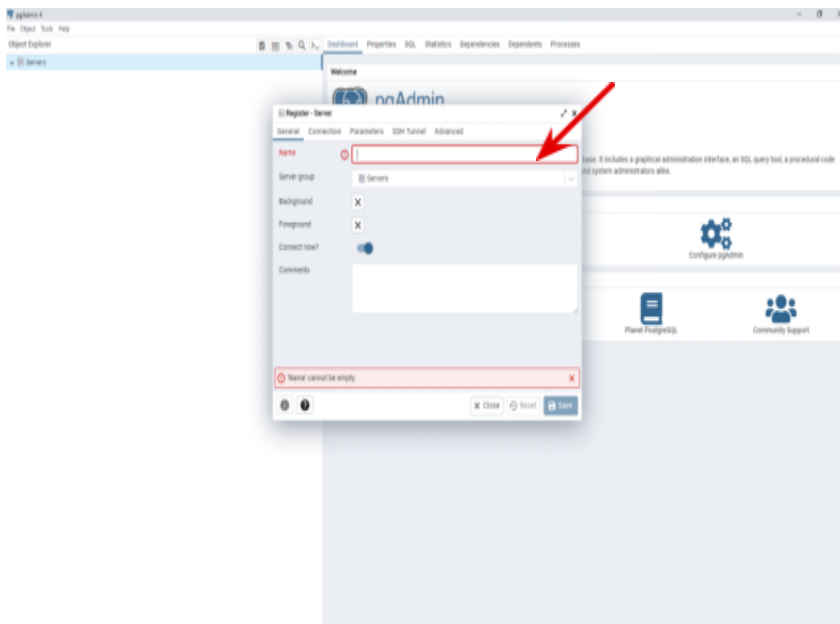
DATABASE CREATION:

1. Add new server in pg-Admin

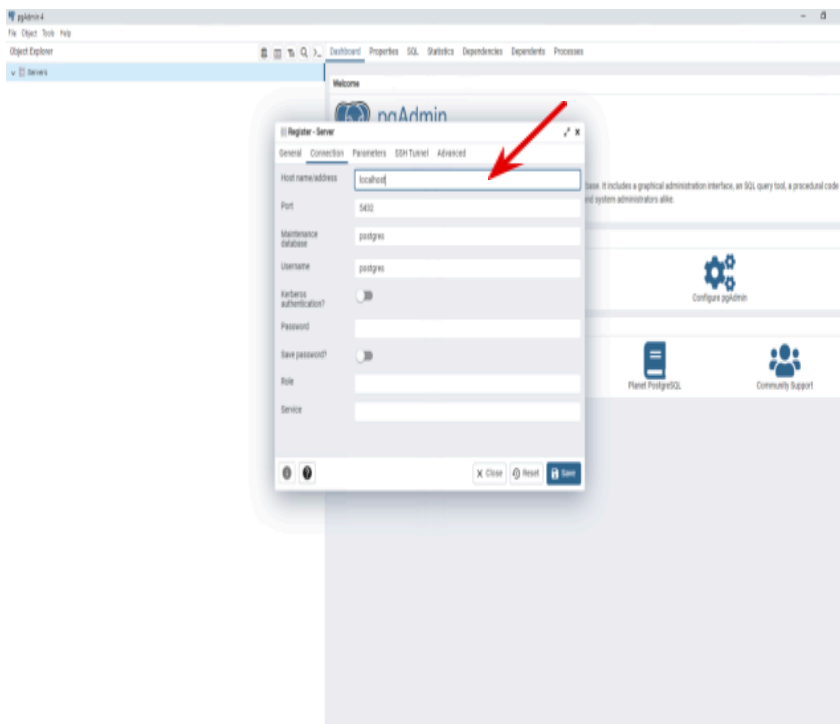
Step 1: Click on add new server button



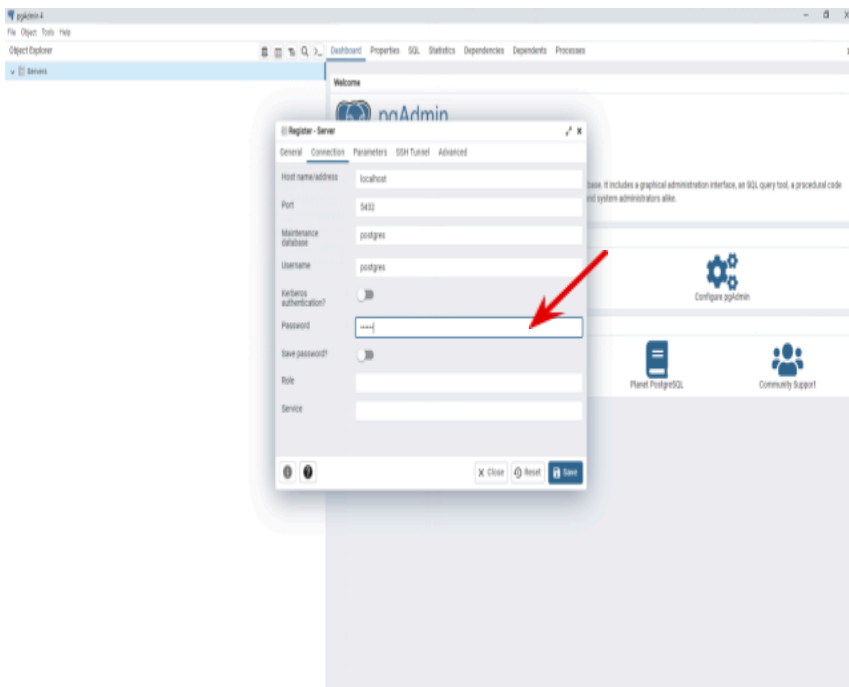
Step 2: Enter your server name



Step 3: Enter the Host name / Address (local PC host name is

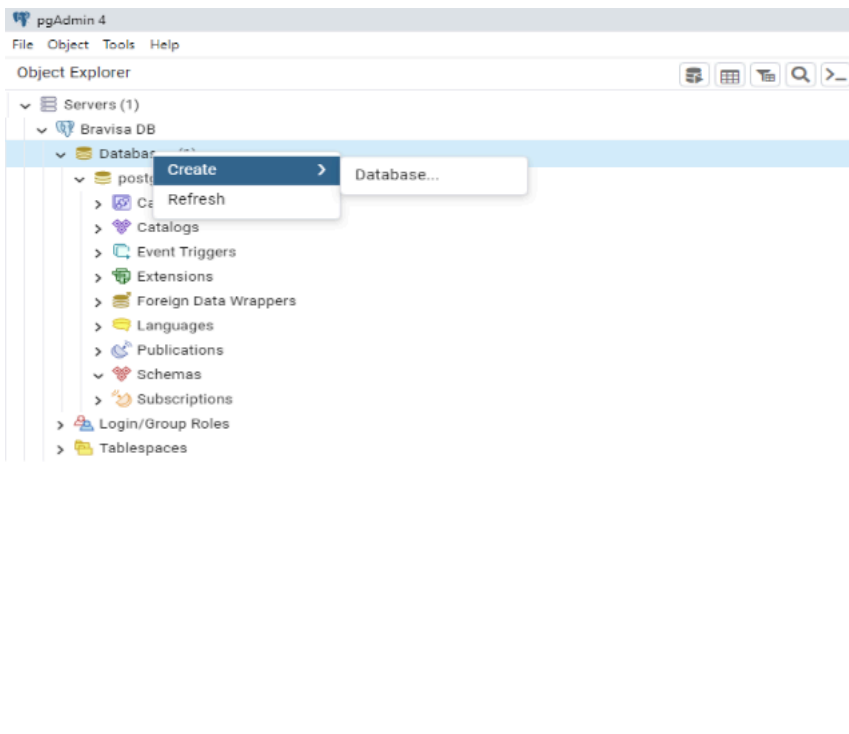


Step 4: Enter the password for database superuser

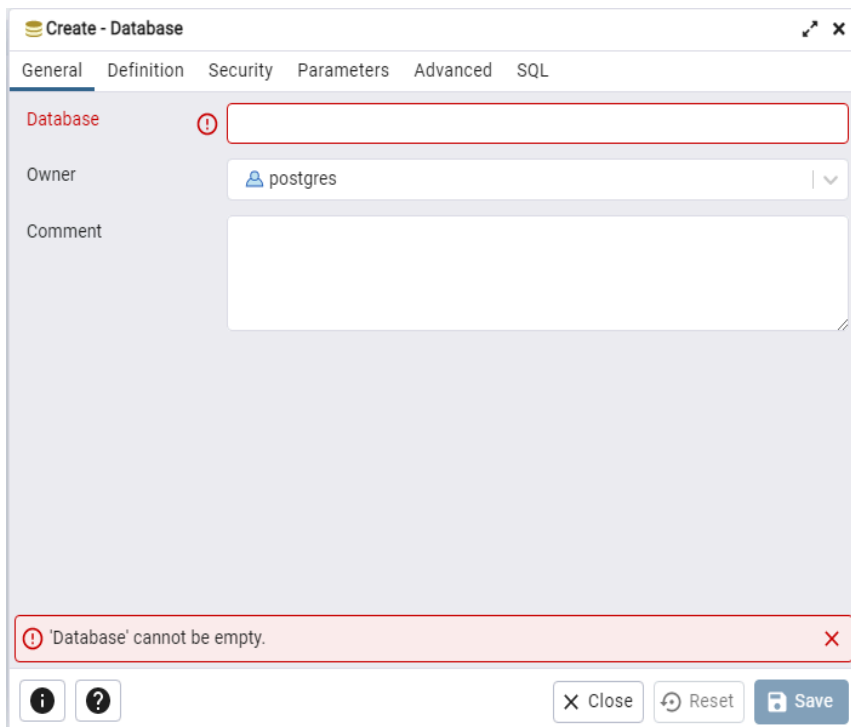


2. Create a Database

Step 1: Right click on Database ,click on create button and click on database.

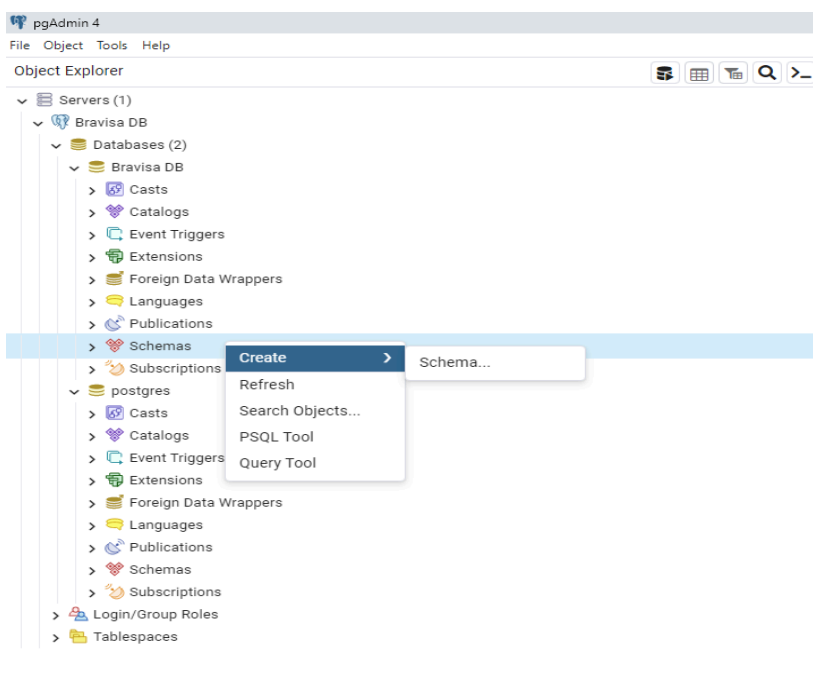


Step 3: Enter the database name



SCHEMAS CREATION:

Step 1: Right click on Schemas , click on Create and click on Schema



Step 2: Enter the Schema name



Create - Schema

General Security Default privileges SQL

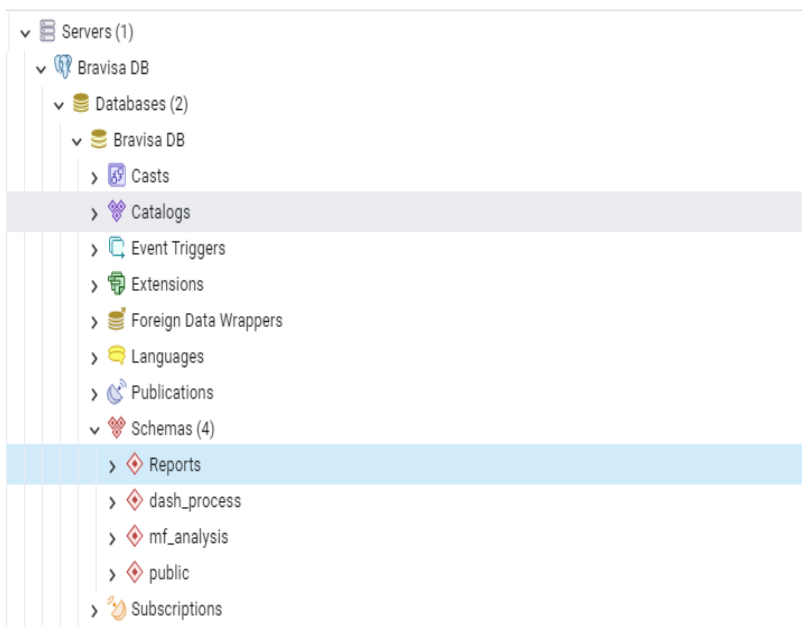
Name: Reports

Owner: postgres

Comment:

Close Reset Save

- Create three Schemas (Reports,dash_process,mf_analysis) following above steps.



TABLES CREATION:

Creating a table in PostgreSQL

In PostgreSQL, we can create a table in two ways:

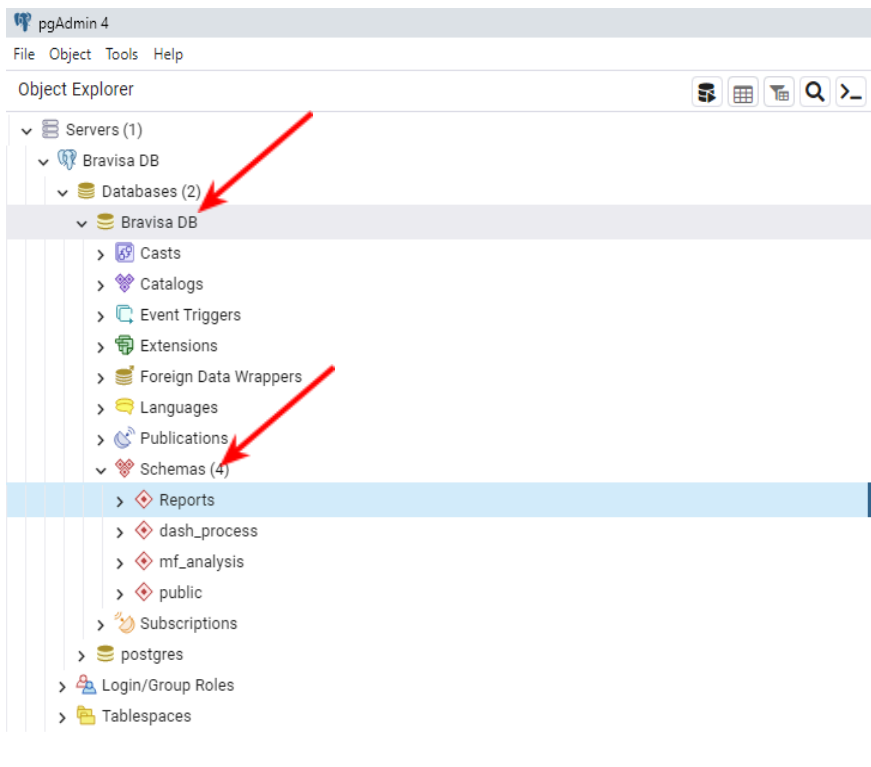
- PostgreSQL Create Table using pgAdmin
- PostgreSQL Create Table using SQL Shell

PostgreSQL Create Table using pgAdmin

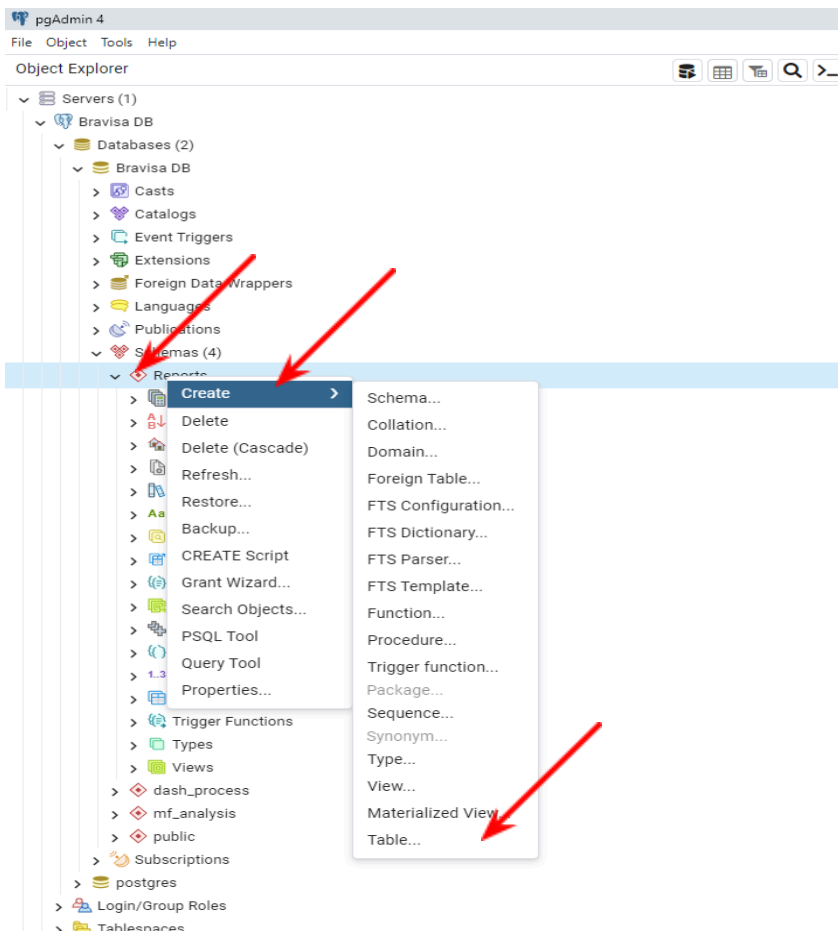
We are going to follow the below process to create a table in pgAdmin:

Step 1: Firstly, we will open the latest version pgAdmin in our local system, and we will go to the object tree and select the database, in which we want to create a table.

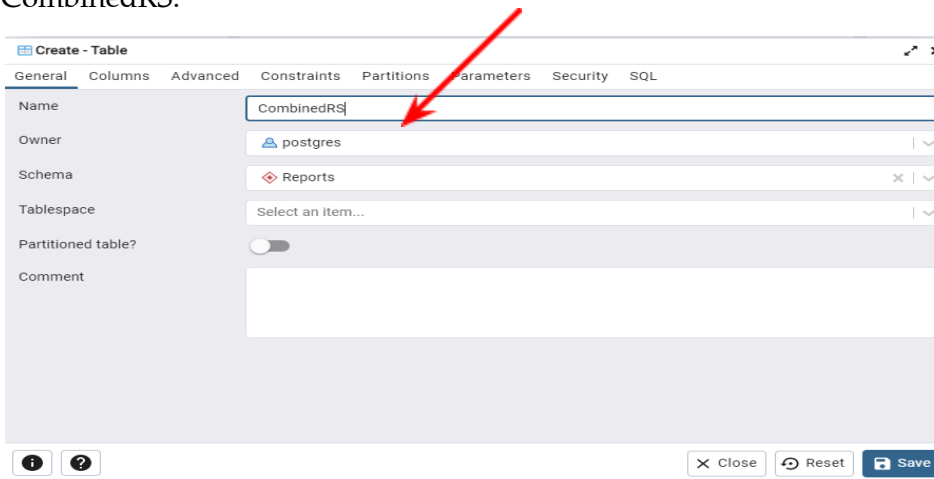
Step 2: After that, left-click on the selected database(Bravisa DB)and then we can see the Catalogs and Schemas



Step 3: Then we will right-click on the Reports under Schema section, select Create option from the given drop-down, and click on the Table from the given list.



Step 4: Once we click on the Table, the Create-table window will appear on the screen where we will enter all the necessary details like Table name. In our case, we will create a table called CombinedRS.



Step 5:

- After that, we will move to the Column tab in the same window then click on the + sign to add columns in a particular table
- And we can select the Data types from the given drop-down list as well as we can change the columns Not-null preference and also set the Primary key.
- And then click on Save to complete the process of creating a table as we can see in the below screenshot:

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
CompanyCode	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
CompanyName	character varying			<input type="checkbox"/>	<input type="checkbox"/>	
NSECode	character varying			<input type="checkbox"/>	<input type="checkbox"/>	
BSECode	bigint			<input type="checkbox"/>	<input type="checkbox"/>	
PRSRank	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
ERSRank	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
RRSRank	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
FRSRank	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
IRSRank	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
CombIRS	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
Rank	integer			<input type="checkbox"/>	<input type="checkbox"/>	
Value Average	double precision			<input type="checkbox"/>	<input type="checkbox"/>	
GenDate	date			<input type="checkbox"/>	<input type="checkbox"/>	

- And we can see that the CombinedRS table is created under the Table section.

```

-- Table: Reports.CombinedRS
-- DROP TABLE IF EXISTS "Reports"."CombinedRS";
CREATE TABLE IF NOT EXISTS "Reports"."CombinedRS"
(
    "CompanyCode" double precision,
    "CompanyName" character varying COLLATE pg_catalog."default",
    "NSECode" character varying COLLATE pg_catalog."default",
    "BSECode" bigint,
    "PRSRank" double precision,
    "ERSRank" double precision,
    "RRSRank" double precision,
    "FRSRank" double precision,
    "IRSRank" double precision,
    "CombIRS" double precision,
    "Rank" integer,
    "Value Average" double precision,
    "GenDate" date
)
TABLESPACE pg_default;
ALTER TABLE IF EXISTS "Reports"."CombinedRS"
OWNER to postgres;
  
```

PostgreSQL Create Table using psql:

We are going to follow the below process to create a table in psql:

Step1:

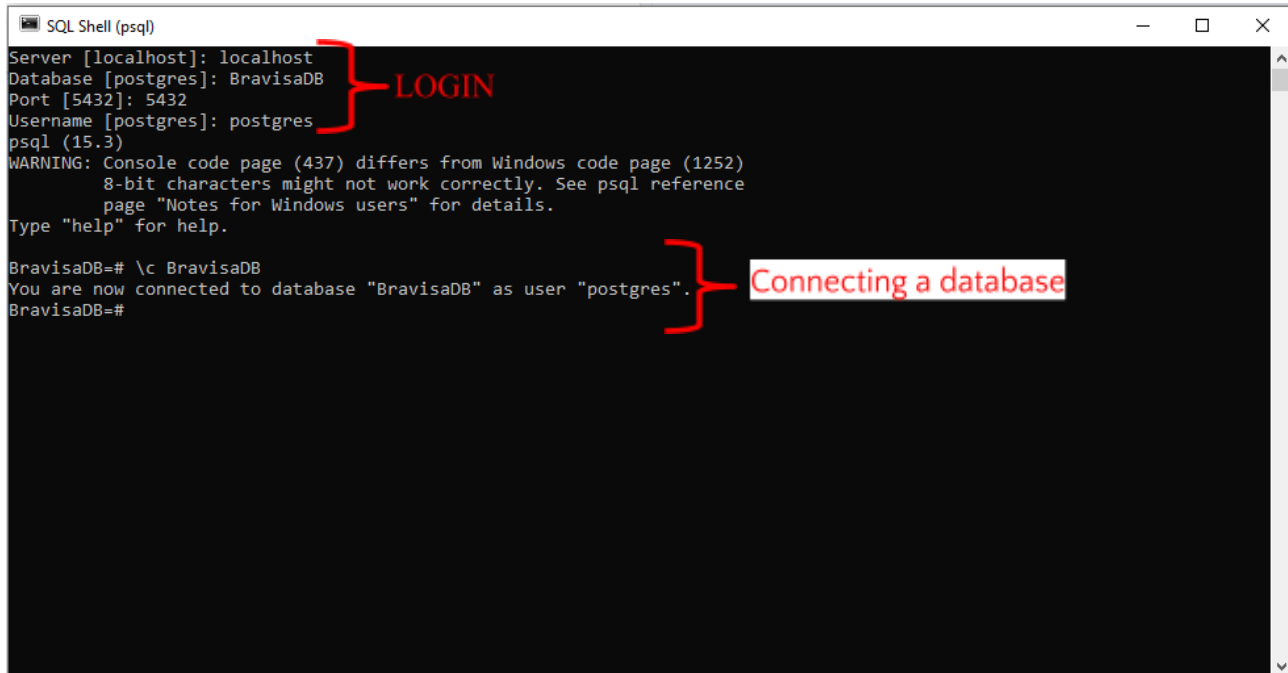
Firstly, we will open the psql in our local system, and we will connect to the database where we want to create a table.

We will create a table in the BravisaDB database, which we created earlier in the PostgreSQL tutorial.

Step 2:

- Login with your hostname, database, port, username and password
- For connecting a database, we will enter the below command:

\c BravisaDB



The screenshot shows a terminal window titled "SQL Shell (psql)". The prompt is "psql (15.3)". The user has entered the following commands and received the following responses:

```
Server [localhost]: localhost
Database [postgres]: BravisaDB
Port [5432]: 5432
Username [postgres]: postgres
psql (15.3)
WARNING: Console code page (437) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.

BravisaDB=# \c BravisaDB
You are now connected to database "BravisaDB" as user "postgres".
BravisaDB=#
```

Red annotations are present in the image:

- A red bracket on the right side of the first four input lines (Server, Database, Port, Username) is labeled "LOGIN".
- A red bracket on the right side of the command "\c BravisaDB" is labeled "Connecting a database".

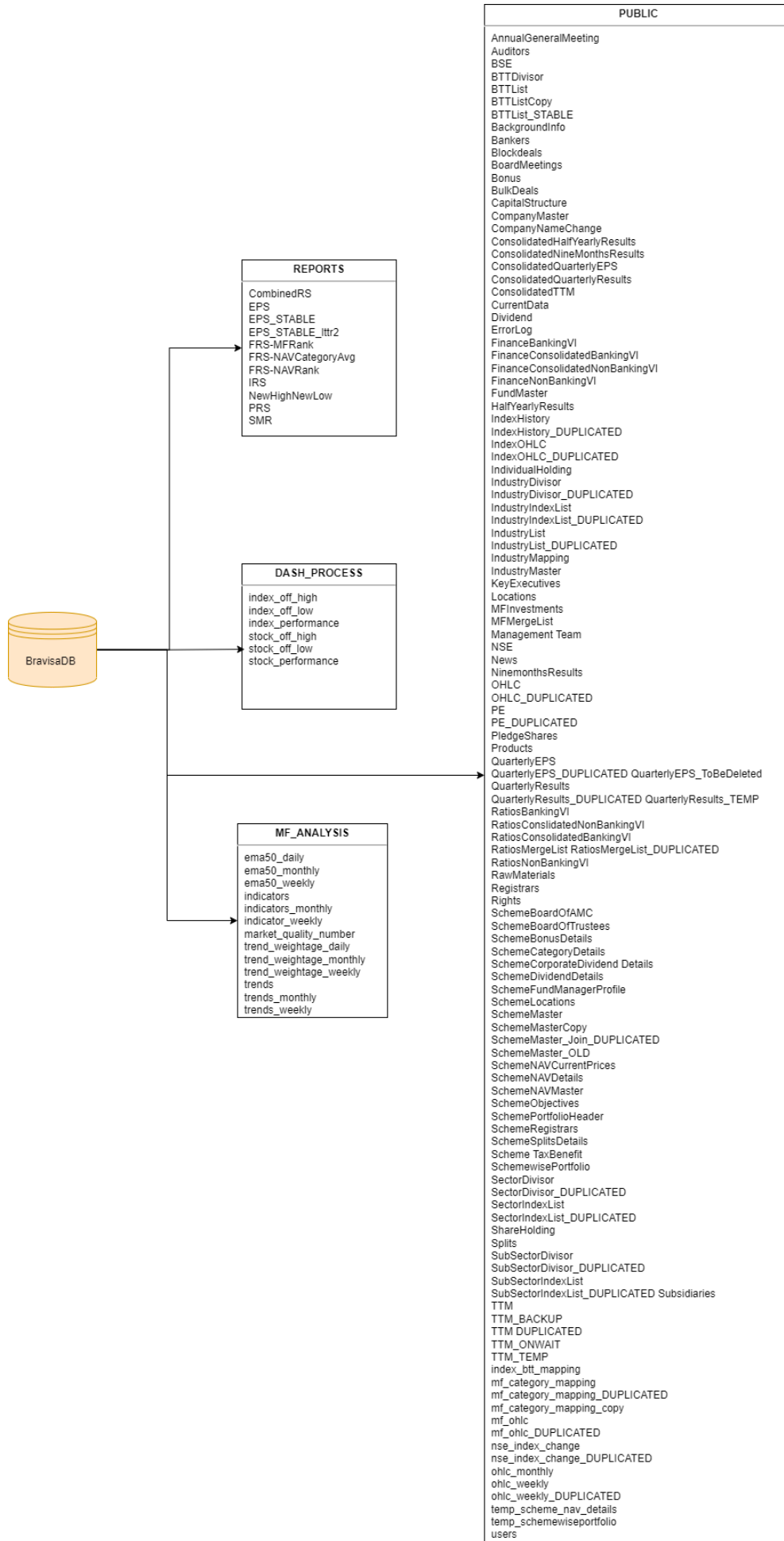
Step 3:

- Now, we will enter the below command to create a table in the BravisaDB database.

```
-- Table: "Reports"."EPS"
-- DROP TABLE "Reports"."EPS";
CREATE TABLE "Reports"."EPS"
(
    "CompanyCode" double precision,
    "NSECode" character varying COLLATE pg_catalog."default",
    "BSECode" bigint,
    "CompanyName" character varying COLLATE pg_catalog."default",
    "ISIN" character varying COLLATE pg_catalog."default",
```

```
"Months" smallint,  
"Quarter" smallint,  
"YearEnding" date,  
"Q1 EPS" double precision,  
"Q1 EPS Growth" double precision,  
"Q1 Sales" double precision,  
"Q1 Sales Growth" double precision,  
"Q2 EPS" double precision,  
"Q2 EPS Growth" double precision,  
"Q2 Sales" double precision,  
"Q2 Sales Growth" double precision,  
"TTM1 EPS Growth" double precision,  
"TTM1 Sales Growth" double precision,  
"TTM2 EPS Growth" double precision,  
"TTM2 Sales Growth" double precision,  
"TTM3 EPS Growth" double precision,  
"TTM3 Sales Growth" double precision,  
"NPM" double precision,  
"EPS Rating" double precision,  
"Ranking" double precision,  
"EPSDate" date  
)  
TABLESPACE pg_default;  
ALTER TABLE "Reports"."EPS"  
OWNER to postgres;
```

As we can see in the below screenshot that the table is created in BravisaDB database.



gui.py

```
import tkinter as tk
```

This line imports the tkinter module, which provides tools for creating graphical user interfaces (GUIs) in Python.

```
from tkinter import ttk
```

This line imports the ttk module from tkinter, which provides themed widgets that have a more modern and visually appealing appearance than the standard tkinter widgets.

```
from tracemalloc import start
```

This line imports the start function from the tracemalloc module, which is used for tracking memory usage in Python programs.

```
from tkcalendar import Calendar, DateEntry
```

This line imports the Calendar and DateEntry classes from the tkcalendar module, which provides widgets for selecting dates in a GUI.

```
from tooltip_warnings import ToolTip, CreateToolTip
```

This line imports the ToolTip and CreateToolTip classes from the tooltip_warnings module. These classes likely handle displaying tooltips (hints or descriptions that appear when you hover over a GUI element).

```
import datetime
```

This line imports the built-in datetime module, which provides classes and functions for working with dates and times.

```
from main import run_scripts_frompy
```

This line imports the `run_scripts_frompy` function from the main module. It seems like this function is intended to execute certain scripts.

```
from report_delete import run_delete
```

This line imports the `FB_Insert` class from the `fb_insert` module in the `lib.fb_process` package. This class likely handles inserting data into some database.

```
import psycopg2
```

This line imports the `psycopg2` module, which is a PostgreSQL adapter for Python. It's used for working with PostgreSQL databases.

```
import numpy as np
import pandas as pd
```

These lines import the `numpy` and `pandas` libraries, which are widely used for numerical computations and data manipulation, respectively.

```
from dataadd import add_full_fb_data, add_missing_data
```

This line imports the `add_full_fb_data` and `add_missing_data` functions from the `dataadd` module. These functions are likely used to add data to some dataset.

```
from lib.reports.EPS import EPS
```

This line imports the `EPS` class from the `lib.reports.EPS` module. This class likely deals with generating or handling EPS (Earnings Per Share) reports.

```
prodDBhost = "localhost"
prodDB = "BravisaDB"
prodDBuser = "postgres"
prodDBPass = "bravisatempletree"
prodDBPort = "5432"
```

These lines define various database connection parameters such as host, database name, username, password, and port. These parameters are used to connect to a PostgreSQL database.

We set up the necessary imports and defines variables for database connection parameters. It also creates a Tkinter GUI window with a notebook widget containing multiple frames for different functionalities like deleting reports, starting a process, inserting data, and downloading reports.

```
conn = psycopg2.connect(host=prodDBhost, dbname=prodDB,  
                        user=prodDBuser, password=prodDBPass,  
cur = conn.cursor()
```

These lines establish a connection to the PostgreSQL database using the connection parameters (prodDBhost, prodDB, prodDBuser, prodDBPass, and prodDBPort) defined earlier. The psycopg2.connect() function is used to establish the connection, and then a cursor (cur) is created for executing SQL queries on the database.

```
root = tk.Tk()  
root.geometry('720x480')  
root.title('Notebook Demo')
```

These lines create the main Tkinter window (root) with a specified geometry of 720x480 pixels and a title "Notebook Demo".

```
notebook = ttk.Notebook(root)  
notebook.pack(pady=5, expand=True)
```

Here, a ttk.Notebook widget is created and added to the root window. The pack() method is used to display the notebook in the window. The pady parameter adds a vertical padding of 5 pixels, and expand=True allows the notebook to expand within the available space.

```
delete_data_frame = ttk.Frame(notebook, width=720, height=400)  
start_main = ttk.Frame(notebook, width=720, height=400)  
insert_data = ttk.Frame(notebook, width=720, height=400)  
download_reports = ttk.Frame(notebook, width=720, height=400)
```

Four frames (delete_data_frame, start_main, insert_data, and download_reports) are created using the ttk.Frame class. Each frame is given a specific width and height within the notebook.


```
delete_data_frame.pack(fill='both', expand=True)
start_main.pack(fill='both', expand=True)
insert_data.pack(fill='both', expand=True)
download_reports.pack(fill='both', expand=True)
```

These lines use the pack() method to display each of the frames within the notebook. The fill='both' option ensures that the frames fill both horizontal and vertical space, and expand=True allows the frames to expand within the available space.

```
notebook.add(delete_data_frame, text='Delete Reports')
notebook.add(start_main, text='Start Process')
notebook.add(insert_data, text='Insert data')
notebook.add(download_reports, text='Download Reports')
```

The add() method is used to add each frame to the notebook. The text parameter specifies the text that appears as the tab label for each frame.

the code sets up a Tkinter GUI with a notebook containing frames for different functionalities. However, the specific functionality and logic of each frame's content are not included.

FUNCTION: INSERT DATA

```
starttext = tk.Label(insert_data, text="START DATE", width=10, height=1)
starttext.place(x=10, y=10)
```

This code creates a tk.Label widget within the insert_data frame. The label displays "START DATE" and is given a width and height. The place() method is used to position the label at coordinates (10, 10) within the frame.

```
start_date = tk.StringVar()
end_date = tk.StringVar()
```

These lines create two `tk.StringVar` instances named `start_date` and `end_date`. These string variables will be used to store the selected start and end dates.

```
calendar_startdate = DateEntry(insert_data, width= 15, height = 15, background= "magenta3",
                                foreground= "white",bd=2, set_date=Calendar.datetime.today(),
                                date_pattern="dd-mm-yyyy", textvariable=start_date)
calendar_startdate.place(x=120, y=10)
```

This code creates a `DateEntry` widget within the `insert_data` frame. It's used to allow the user to pick a date using a calendar. The widget is styled with a magenta background and white foreground, and the `set_date` parameter sets the default date to the current date. The `textvariable` parameter is set to the `start_date` variable to store the selected date.

The placement of the `calendar_startdate` widget is specified using the `place()` method.

Similar code is repeated for the "END DATE" widget:

```
endtext = tk.Label(insert_data, text="END DATE", width= 10, height = 1)
endtext.place(x=10,y=50)

calendar_enddate = DateEntry(insert_data, width= 15, height = 15, background= "magenta3",
                                foreground= "white",bd=2, set_date=Calendar.datetime.today(),
                                date_pattern="dd-mm-yyyy", textvariable=end_date)
calendar_enddate.place(x=120, y=50)
```

The code then defines a function `run_function()` that is executed when the "INSERT DATA" button is clicked. This function:

Retrieves the selected start and end dates from the `start_date` and `end_date` variables.
Converts the date strings into `datetime.date` objects.

A "INSERT DATA" button is also created, and when clicked, it triggers the `run_function()`.

```
def run_function():
    startdate, enddate = "", ""
    if(start_date.get() != ""):
        startdate = (start_date.get()).split("-")
        startdate = datetime.date(int(startdate[2]), int(startdate[1]), int(startdate[0]))
    if(end_date.get() != ""):
        enddate = (end_date.get()).split("-")
        enddate = datetime.date(int(enddate[2]), int(enddate[1]), int(enddate[0]))
    print(startdate, flush=True)
    print(enddate, flush=True)
```

- The function begins by initializing `startdate` and `enddate` as empty strings.

- It then checks whether the start_date (the user-selected start date) is not empty. If it's not empty, the date is split into day, month, and year components using the - delimiter. The components are then used to create a datetime.date object called startdate.
- The same process is performed for the end_date (the user-selected end date) to create the enddate.
- The start and end dates are then printed to the console using print().

```
time_warning = tk.Label(insert_data, text="This process will take close to 5 minutes. \nPLEASE DON'T CLOSE THE WINDOW", height = 2, foreground='Red')
time_warning.place(x=70,y=120)
# For Variable Label == "Normal Data Insert"
```

A label widget named time_warning is created within the insert_data frame. This label is used to provide a warning message to the user that the data insertion process will take about 5 minutes and advises not to close the window.

python

```
if (variable.get() == "Normal Data Insert"):
    add_full_fb_data(startdate, enddate, conn, cur, time_warning)
elif (variable.get() == "Missing Data Insert"):
    add_missing_data(conn, cur)
```

- Here, the code checks the value of the variable (dropdown selection) using variable.get().
- If the selected value is "Normal Data Insert," it calls the **add_full_fb_data()** function with the startdate, enddate, conn (database connection), cur (cursor), and time_warning as arguments. This function appears to be responsible for adding full data into the database for the specified date range.
- If the selected value is "Missing Data Insert," it calls the **add_missing_data()** function with the conn and cur arguments. This function appears to be responsible for adding missing data into the database.

the run_function() in the "INSERT DATA" functionality is responsible for extracting the selected start and end dates from the GUI, displaying a time warning label, and then triggering the appropriate data insertion functions based on the selected option. It's a key part of the GUI's interaction with the data insertion processes.

FUNCTION: DELETE DATA

The code snippet you've posted is responsible for creating date selection components in the "DELETE DATA" functionality of the GUI.

```
# For Delete Data Frame
starttext = tk.Label(delete_data_frame, text="START DATE", width= 10, height = 1)
starttext.place(x=10,y=10)
```

- This code creates a label widget named starttext within the delete_data_frame frame. The label displays the text "START DATE".
- The width and height parameters define the dimensions of the label widget.

- The place() method is used to position the label within the frame at coordinates (x=10, y=10).

```
calender_startdate = DateEntry(delete_data_frame, width= 15, height = 15, background= "magenta3",
                               foreground= "white",bd=2, set_date=Calendar.datetime.today(),
                               date_pattern="dd-mm-yyyy", textvariable=start_date)
calender_startdate.place(x=120, y=10)

endtext = tk.Label(delete_data_frame, text="END DATE", width= 10, height = 1)
endtext.place(x=10,y=50)

calender_enddate = DateEntry(delete_data_frame, width= 15, height = 15, background= "magenta3",
                              foreground= "white",bd=2, set_date=Calendar.datetime.today(),
                              date_pattern="dd-mm-yyyy", textvariable=end_date)
calender_enddate.place(x=120, y=50)
```

- This code creates a DateEntry widget named calender_startdate within the delete_data_frame frame. This widget allows the user to select a date.
- The width and height parameters define the dimensions of the date entry widget.
- The background parameter sets the background color of the widget.
- The foreground parameter sets the text color of the widget.
- The bd parameter sets the border width of the widget.
- The set_date parameter initializes the widget with the current date.
- The date_pattern parameter defines the format of the displayed date.
- The textvariable parameter is set to the start_date variable, which will store the selected start date.
- The place() method is used to position the date entry widget within the frame at coordinates (x=120, y=10).
- Similarly, the code creates a label and a date entry widget for selecting the end date with similar configurations.
- Overall, these lines of code create user interface components for selecting a start date and an end date within the "DELETE DATA" functionality of the GUI. The components allow users to pick specific dates to define the date range for data deletion.

```
def run_function():
    startdate, enddate = "", ""
    if(start_date.get() != ""):
        startdate = (start_date.get()).split("-")
        startdate = datetime.date(int(startdate[2]), int(startdate[1]), int(startdate[0]))
    if(end_date.get() != ""):
        enddate = (end_date.get()).split("-")
        enddate = datetime.date(int(enddate[2]), int(enddate[1]), int(enddate[0]))
    print(startdate, flush=True)
    print(enddate, flush=True)
    print(delete_variable.get(), flush=True)
```

- This part of the code retrieves the selected start and end dates from the GUI using the start_date.get() and end_date.get() methods.
- The selected dates are then split into day, month, and year components using the - delimiter and converted into datetime.date objects.
- The selected start and end dates, along with the chosen option from the delete_variable dropdown, are printed to the console using the print() function.

```
time_warning = tk.Label(delete_data_frame, text="This process will take close to 5 minutes. \nPLEASE DON'T CLOSE THE WINDOW", height = 2, foreground='Red')
time_warning.place(x=70,y=120)
```

- A label widget named time_warning is created within the delete_data_frame frame. This label is used to display a warning message to the user, indicating that the data deletion process may take up to 5 minutes and advising the user not to close the window.
- The label's text and appearance settings are defined.

```
if(delete_variable.get() == "Traditional Delete"):
    run_delete(startdate, enddate, time_warning)

elif(delete_variable.get() == "EPS"):
    print("Deleting EPS reports")
    sql = 'DELETE FROM "Reports"."EPS" WHERE "EPSDate" >= DATE(\'\' + str(startdate) + \'\' ) AND "EPSDate" < DATE(\'\' + str(enddate) + \'\' )'
    cur.execute(sql)
    conn.commit()

elif(delete_variable.get() == "PRS"):
    print("Deleting PRS reports")
    sql = 'DELETE FROM "Reports"."PRS" WHERE "Date" >= DATE(\'\' + str(startdate) + \'\' ) AND "Date" < DATE(\'\' + str(enddate) + \'\' )'
    cur.execute(sql)
    conn.commit()

elif(delete_variable.get() == "SMR"):
    print("Deleting SMR reports")
    sql = 'DELETE FROM "Reports"."SMR" WHERE "SMRDate" >= DATE(\'\' + str(startdate) + \'\' ) AND "SMRDate" < DATE(\'\' + str(enddate) + \'\' )'
    cur.execute(sql)
    conn.commit()

elif(delete_variable.get() == "IRS"):
    print("Deleting IRS reports")
    sql = 'DELETE FROM "Reports"."IRS" WHERE "GenDate" >= DATE(\'\' + str(startdate) + \'\' ) AND "GenDate" < DATE(\'\' + str(enddate) + \'\' )'
    cur.execute(sql)
    conn.commit()

elif(delete_variable.get() == "FRS-NavRank"):
    print("Deleting FRS-NavRank reports")
    sql = 'DELETE FROM "Reports"."FRS-NAVRank" WHERE "Date" >= DATE(\'\' + str(startdate) + \'\' ) AND "Date" < DATE(\'\' + str(enddate) + \'\' )'
    cur.execute(sql)
    conn.commit()

elif(delete_variable.get() == "CombinedRS"):
    print("Deleting CombinedRS reports")
    sql = 'DELETE FROM "Reports"."CombinedRS" WHERE "GenDate" >= DATE(\'\' + str(startdate) + \'\' ) AND "GenDate" < DATE(\'\' + str(enddate) + \'\' )'
    cur.execute(sql)
    conn.commit()

elif(delete_variable.get() == "BITT"):
    print("Deleting BITT reports")
    sql = 'DELETE FROM public."BITTList" WHERE "BITTDate" >= DATE(\'\' + str(startdate) + \'\' ) AND "BITTDate" < DATE(\'\' + str(enddate) + \'\' )'
    cur.execute(sql)
    conn.commit()
```

- This part of the code checks the selected option from the delete_variable dropdown using delete_variable.get().
- Depending on the selected option, the appropriate data deletion operation is performed.
- For example, if the selected option is "Traditional Delete," the function **run_delete()** is called with the selected start and end dates and the time_warning label. This function seems to be responsible for deleting reports based on the specified date range.
- Similar code blocks exist for other report types, each calling the appropriate **SQL query** to delete the specified reports.

```
delete_variable = tk.StringVar(delete_data_frame)
delete_variable.set("Traditional Delete") # default value
```

This creates a StringVar named delete_variable within the delete_data_frame frame. StringVar is used to store the selected value from a dropdown or combobox. The default value of the delete_variable is set to "Traditional Delete".

```
dropbox = ttk.Combobox(delete_data_frame, textvariable=delete_variable, values=["Traditional Delete", "EPS", "FRS", "SMR", "IRS", "FRS-NavRank", "CombinedRS", "BIT"], width=15)
dropbox.place(x=280, y=10)
```

- This code creates a combobox widget named dropbox within the delete_data_frame frame. The combobox provides options for users to choose the type of report deletion operation they want to perform.
- The textvariable parameter is set to the delete_variable, so the selected option will be stored in the delete_variable.
- The values parameter specifies the available options in the combobox.
- The width parameter defines the width of the combobox.
- The place() method is used to position the combobox within the frame at coordinates (x=280, y=10).

```
run_button = tk.Button(delete_data_frame, text ="DELETE REPORTS", command = run_function,
                        activebackground='black', activeforeground='white')
run_button.place(x = 70, y=90)
```

- This code creates a button widget named run_button within the delete_data_frame frame. The button is used to trigger the run_function() when clicked.
- The button's text is set to "DELETE REPORTS".
- The command parameter is set to run_function, which means when the button is clicked, the **run_function()** will be executed.
- The activebackground and activeforeground parameters define the colors of the button when it's in the active (pressed) state.
- The place() method is used to position the button within the frame at coordinates (x=70, y=90).

the run_function() within the "DELETE DATA" functionality is responsible for extracting selected start and end dates, displaying a time warning label, and performing data deletion operations based on the selected report type. It plays a key role in interacting with the GUI and initiating data deletion processes.

FUNCTION: START PROCESS

```
starttext = tk.Label(start_main, text="START DATE", width=10, height=1)
starttext.place(x=10, y=10)
```

- This code creates a label widget named starttext within the start_main frame. The label displays the text "START DATE".
- The width and height parameters define the dimensions of the label in terms of characters and lines.

```
calender_startdate = DateEntry(start_main, width= 15, height = 15, background= "magenta3",
                                foreground= "white", bd=2, set_date=Calendar.datetime.today(),
                                date_pattern="dd-mm-yyyy", textvariable=start_date)
calender_startdate.place(x=120, y=10)
```

- This code creates a DateEntry widget named calender_startdate within the start_main frame. The DateEntry widget is used to input a date.
- The width and height parameters define the dimensions of the widget.
- The background and foreground parameters define the colors of the widget.
- The bd parameter defines the border width of the widget.
- The set_date parameter sets the default date to the current date.
- The date_pattern parameter specifies the date format.
- The textvariable parameter is set to the start_date variable to store the selected date.

```
endtext = tk.Label(start_main, text="END DATE", width=10, height=1)
endtext.place(x=10, y=50)
```

This code creates another label widget named endtext within the start_main frame. The label displays the text "END DATE".

```
calender_enddate = DateEntry(start_main, width= 15, height = 15, background= "magenta3",
                                foreground= "white", bd=2, set_date=Calendar.datetime.today(),
                                date_pattern="dd-mm-yyyy", textvariable=end_date)
calender_enddate.place(x=120, y=50)
```

- This code creates another DateEntry widget named calender_enddate within the start_main frame. It's similar to the previous DateEntry widget but used for the end date.
- The positioning using place() is also defined.

```
CreateToolTip(calender_enddate, text = 'Keep it null for single date run')
```


This code uses the CreateToolTip function (presumably defined elsewhere in the code) to create a tooltip for the calender_enddate widget. The tooltip text is "Keep it null for single date run".

this code segment sets up user interface components for selecting start and end dates within the "START PROCESS" functionality of the GUI. Users can use DateEntry widgets to pick dates, and a tooltip provides guidance on using the end date input for single-date runs.

```
def run_function():
    startdate, enddate = "", ""
    if(start_date.get() != ""):
        startdate = (start_date.get()).split("-")
        startdate = datetime.date(int(startdate[2]), int(startdate[1]), int(startdate[0]))
    if(end_date.get() != ""):
        enddate = (end_date.get()).split("-")
        enddate = datetime.date(int(enddate[2]), int(enddate[1]), int(enddate[0]))
    print(startdate, flush=True)
    print(enddate, flush=True)
```

- This function run_function() retrieves the selected start and end dates from the respective start_date and end_date variables.
- The selected dates are split using the "-" separator and then converted into datetime.date objects.

```
if(start_variable.get() == "All Reports"):
    flag = run_scripts_frompy(startdate, enddate)
    time_warning = tk.Label(start_main, text="Processed Reports from {} to {}".format(str(startdate), str(enddate)), height = 2, foreground='Green')
    time_warning.place(x=70,y=120)
    print(flag)
```

- If the selected value in the dropdown (start_variable) is "All Reports", it calls the run_scripts_frompy function with the start and end dates.
- It then creates a label widget time_warning that displays a message indicating the range of processed reports and sets its position on the GUI.
- The flag value (result from run_scripts_frompy) is printed to the console.

```
if(start_variable.get() == "EPS"):
    date_list = [startdate+datetime.timedelta(days=x) for x in range((enddate-startdate).days)]
    for current_date in date_list:
        EPS().Generate_Daily_Report(current_date, conn, cur)
```

- If the selected value in the dropdown (start_variable) is "EPS", it generates a list of dates between the start and end dates.
- It then iterates through each date in the list and calls the Generate_Daily_Report function of the EPS class, passing the current date along with the database connection and cursor objects.

```
start_variable = tk.StringVar(start_main)
```

```
dropbox = ttk.Combobox(start_main, textvariable=start_variable, values=["All Reports", "EPS"], width=15)
dropbox.place(x=280, y=10)
```

```
run_button = tk.Button(start_main, text = "RUN PROCESS", command = run_function,
                        activebackground='black', activeforeground='white')
run_button.place(x = 70, y=90)
```


- This code sets up the dropdown (Combobox) widget named dropbox with two possible values: "All Reports" and "EPS".
- The textvariable parameter is set to the start_variable variable to store the selected value.
- The button run_button is created with the label "RUN PROCESS" and the command parameter is set to the run_function function.
- The button's active background and foreground colors are customized.
- Both the dropdown and the button are positioned using the place() method.

This code segment sets up user interface components for selecting a process type ("All Reports" or "EPS") and executing the corresponding process using the selected date range. The process type determines which function(s) will be called based on the selected start and end dates.

DOWNLOAD DATA

```
starttext = tk.Label(download_reports, text="START DATE", width= 10, height = 1)
starttext.place(x=10,y=10)
```

This line creates a label widget named starttext with the text "START DATE" and sets its width and height.

```
calender_startdate = DateEntry(download_reports, width= 15, height = 15, background= "magenta3",
                                foreground= "white",bd=2, set_date=Calendar.datetime.today(),
                                date_pattern="dd-mm-yyyy", textvariable=start_date)
calender_startdate.place(x=120, y=10)
```

- This code sets up a DateEntry widget named calender_startdate for selecting the start date.
- The widget's parameters, such as width, height, background color, and date format, are customized.
- The set_date parameter is used to set the default date to the current date.
- The textvariable parameter is set to the start_date variable, which is used to store the selected start date.

```
endtext = tk.Label(download_reports, text="END DATE", width= 10, height = 1)
endtext.place(x=10,y=50)
```

This line creates another label widget named endtext with the text "END DATE" and sets its width and height.

```
calender_enddate = DateEntry(download_reports, width= 15, height = 15, background= "magenta3",
                                foreground= "white",bd=2, set_date=Calendar.datetime.today(),
                                date_pattern="dd-mm-yyyy", textvariable=end_date)
calender_enddate.place(x=120, y=50)
```

Similar to previous DateEntry widgets, this code sets up a DateEntry widget named calender_enddate for selecting the end date.

It uses the same customization parameters as the previous widgets.

```
CreateToolTip(calender_enddate, text='Keep it null for single date run')
```

- This line is calling a function CreateToolTip to create a tooltip for the calender_enddate widget.
- The tooltip text is provided as the text parameter.

```
def run_function():
    startdate, enddate = "", ""
    if(start_date.get() != ""):
        startdate = (start_date.get()).split("-")
        startdate = datetime.date(int(startdate[2]), int(startdate[1]), int(startdate[0]))
    if(end_date.get() != ""):
        enddate = (end_date.get()).split("-")
        enddate = datetime.date(int(enddate[2]), int(enddate[1]), int(enddate[0]))
    print(startdate, flush=True)
    print(enddate, flush=True)
```

This section of code reads the selected start and end dates from the respective DateEntry widgets and converts them into Python datetime.date objects. It then prints these dates.

```
# Reports download scripts
if(variable.get() == "EPS" or variable.get() == "All Reports"):
    sql = 'SELECT * FROM "Reports"."EPS" WHERE "EPSDate" >= DATE(\'' + str(startdate) + '\') AND "EPSDate" < DATE(\'' + str(enddate) + '\') ORDER BY "EPSDate"'
    EPS = pd.read_sql_query(sql, con = conn)
    exportfilename = "C:\\Users\\dsram\\BravisaLocalDeploy\\BravisaFiles\\DownloadedFiles\\EPS-{}-{}.csv".format(str(startdate), str(enddate))
    exportfile = open(exportfilename, "w+")
    EPS.to_csv(exportfile, header=True, index=False, line_terminator='\r')
    exportfile.close()

if(variable.get() == "FRS" or variable.get() == "All Reports"):
    sql = 'SELECT * FROM "Reports"."FRS" WHERE "Date" >= DATE(\'' + str(startdate) + '\') AND "Date" < DATE(\'' + str(enddate) + '\') ORDER BY "Date"'
    FRS = pd.read_sql_query(sql, con = conn)
    exportfilename = "C:\\Users\\dsram\\BravisaLocalDeploy\\BravisaFiles\\DownloadedFiles\\FRS-{}-{}.csv".format(str(startdate), str(enddate))
    exportfile = open(exportfilename, "w+")
    FRS.to_csv(exportfile, header=True, index=False, line_terminator='\r')
    exportfile.close()

if(variable.get() == "SMR" or variable.get() == "All Reports"):
    sql = 'SELECT * FROM "Reports"."SMR" WHERE "SMRDate" >= DATE(\'' + str(startdate) + '\') AND "SMRDate" < DATE(\'' + str(enddate) + '\') ORDER BY "SMRDate"'
    SMR = pd.read_sql_query(sql, con = conn)
    exportfilename = "C:\\Users\\dsram\\BravisaLocalDeploy\\BravisaFiles\\DownloadedFiles\\SMR-{}-{}.csv".format(str(startdate), str(enddate))
    exportfile = open(exportfilename, "w+")
    SMR.to_csv(exportfile, header=True, index=False, line_terminator='\r')
    exportfile.close()

if(variable.get() == "IRS" or variable.get() == "All Reports"):
    sql = 'SELECT * FROM "Reports"."IRS" WHERE "GenDate" >= DATE(\'' + str(startdate) + '\') AND "GenDate" < DATE(\'' + str(enddate) + '\') ORDER BY "GenDate"'
    IRS = pd.read_sql_query(sql, con = conn)
    exportfilename = "C:\\Users\\dsram\\BravisaLocalDeploy\\BravisaFiles\\DownloadedFiles\\IRS-{}-{}.csv".format(str(startdate), str(enddate))
    exportfile = open(exportfilename, "w+")
    IRS.to_csv(exportfile, header=True, index=False, line_terminator='\r')
    exportfile.close()

if(variable.get() == "FRS-NAVRank" or variable.get() == "All Reports"):
    sql = 'SELECT * FROM "Reports"."FRS-NAVRank" WHERE "Date" >= DATE(\'' + str(startdate) + '\') AND "Date" < DATE(\'' + str(enddate) + '\') ORDER BY "Date"'
    FRSNAVRank = pd.read_sql_query(sql, con = conn)
    exportfilename = "C:\\Users\\dsram\\BravisaLocalDeploy\\BravisaFiles\\DownloadedFiles\\FRS-NAVRank-{}-{}.csv".format(str(startdate), str(enddate))
    exportfile = open(exportfilename, "w+")
    FRSNAVRank.to_csv(exportfile, header=True, index=False, line_terminator='\r')
    exportfile.close()

if(variable.get() == "CombinedRS" or variable.get() == "All Reports"):
    sql = 'SELECT * FROM "Reports"."CombinedRS" WHERE "GenDate" >= DATE(\'' + str(startdate) + '\') AND "GenDate" < DATE(\'' + str(enddate) + '\') ORDER BY "GenDate"'
    CombinedRS = pd.read_sql_query(sql, con = conn)
    exportfilename = "C:\\Users\\dsram\\BravisaLocalDeploy\\BravisaFiles\\DownloadedFiles\\CombinedRS-{}-{}.csv".format(str(startdate), str(enddate))
    exportfile = open(exportfilename, "w+")
    CombinedRS.to_csv(exportfile, header=True, index=False, line_terminator='\r')
    exportfile.close()
```

- This block of code checks the value of the variable (report type) selected from the dropdown menu.
- If the selected report type is "EPS" or "All Reports," it queries the database for the corresponding data based on the selected start and end dates and then exports the data to a CSV file.

```
exportfilename = "C:\\Users\\dsram\\BravisaLocalDeploy\\BravisaFiles\\DownloadedFiles\\CombinedRS-{}-{}.csv".format(str(startdate), str(enddate))
exportfile = open(exportfilename, "w+")
CombinedRS.to_csv(exportfile, header=True, index=False, line_terminator='\r')
exportfile.close()
```

- Inside each if block, the code constructs the filename for the CSV file based on the report type and selected start and end dates.

- It then opens the file, writes the data (retrieved from the database query) to the file, and closes the file.

```
run_button = tk.Button(download_reports, text ="DOWNLOAD REPORTS", command = run_function,  
                        activebackground='black', activeforeground='white')  
run_button.place(x=70, y=90)
```

This code creates a button labeled "DOWNLOAD REPORTS" and associates it with the `run_function()` you defined earlier. When this button is clicked, the `run_function()` will be executed.

```
variable = tk.StringVar(download_reports)|  
variable.set("All Reports") # default value  
  
dropbox = ttk.Combobox(download_reports, textvariable=variable, values=["All Reports", "EPS", "PRS", "SMR", "IRS", "FRS-NavRank", "CombinedRS"], width=15)  
dropbox.place(x=280, y=10)
```

- Here, a `StringVar` named `variable` is created to store the selected report type.
- The default value of the variable is set to "All Reports."
- A dropdown menu (`Combobox`) is created with the values ["All Reports", "EPS", "PRS", "SMR", "IRS", "FRS-NavRank", "CombinedRS"].
- The selected value from the dropdown menu will be stored in the variable.

```
root.mainloop()
```

This starts the main event loop of the GUI, which keeps the GUI running and responsive to user interactions until the user closes the application window.

```
from datetime import datetime
```

Imports the `datetime` module, which provides classes for working with dates and times.

```
import pandas.io.sql as sqlio
```

Imports the `sqlio` module from the `pandas.io.sql` package. It's used to work with SQL databases using Pandas DataFrames.

```
from flask import Flask
```

Imports the Flask class from the flask package, which is used to create and manage web applications.

```
from lib.db_helper import DB_Helper
from lib.fb_process.fb_helper import FB_Helper
from lib import btt_list
from lib import ohlc
from lib import ohlc_bravisa
from lib import index_ohlc
from lib import split_bonus
```

Imports various classes, modules, or variables from different files and directories. These are components of your application's functionality.

```
from lib.fb_process.fb_insert_linux import FBLinux
from lib.fb_process.fb_insert import FB_Insert
from lib.fb_process import fb_insert_history
```

Imports classes and modules related to inserting data into a database.

```
from lib.reports.PRS import PRS
from lib.reports.EPS import EPS
from lib.reports.PRS import PRS
from lib.reports.SMR import SMR
from lib.reports.FRS import FRS
from lib.reports.IRS import IRS
from lib.reports.combined_rank import CombinedRank
```

Imports various report-related classes from different modules.

```
from lib import PE
from lib.mf_ohlcv import MFOHLC
from lib.dash_process import index_change
from lib.dash_process import perstock_change
from lib.dash_process import perstock_offhighlow
from lib.dash_process import index_offhighlow
from lib.mf_analysis.rt_daily import DailyRTProcess
from lib.mf_analysis.rt_weekly import WeeklyRTProcess
```

Imports additional classes and modules related to various data analysis and processing tasks.

```
from datetime import datetime
```

Imports the datetime class again.

```
import psycopg2
```

Imports the psycopg2 module, which provides a PostgreSQL adapter for Python.

```
import csv
import os
import datetime
import time
```

Imports modules related to working with CSV files, operating system operations, date and time manipulation, and timing.

```
import sys
import pandas as pd
import json
```

Imports the sys module for system-specific functionality, the pandas library for data manipulation, and the json module for working with JSON data.

```
from flask import Response, request
```

Imports Response and request classes from the flask package, which are used to handle HTTP responses and requests.

```
from pandas.tseries.offsets import BMonthEnd,BMonthBegin
```

Imports specific date offset classes from the pandas.tseries.offsets module.

```
from lib.dash_process import index_offhighlow  
from lib.dash_process.index_offhighlow import IndexOffHighLow
```

Imports classes and modules related to dashboard processing and analysis.

```
from lib.mf_analysis.rt_daily import DailyRTProcess  
from lib.mf_analysis.rt_weekly import WeeklyRTProcess  
from lib.mf_analysis.rt_monthly import MonthlyRTProcess
```

Imports classes related to mutual fund analysis on daily, weekly, and monthly frequencies.

```
from lib.mf_analysis.ema50.ema50_daily import EMA50_daily  
from lib.mf_analysis.ema50.ema50_weekly import EMA50_weekly  
from lib.mf_analysis.ema50.ema50_monthly import EMA50_monthly
```

Imports classes related to Exponential Moving Average (EMA) analysis on daily, weekly, and monthly frequencies.

```
from lib.mf_analysis.market_quality_number.mqn_nse500 import MarketQualityNSE500  
from lib.mf_analysis.market_quality_number.mqn_nifty import MarketQualityNIFTY  
from lib.mf_analysis.market_quality_number.mqn_btt_index import MarketQualityBTT_Index
```

These lines import classes related to market quality number analysis on different indices.

```
from lib.BTTIndex import BTTIndex  
from lib.index_change import IndexCloseChange  
import scripts_helper
```

These lines import classes related to index analysis and other helper modules. The sys.tracebacklimit=0 line sets the limit of displayed traceback errors to 0, effectively hiding them.

```
# If 'entrypoint' is not defined in app.yaml, App Engine will look for an app
# called 'app' in 'main.py'.
app = Flask(__name__)
# flask_port = os.environ['flask_port']
flask_port=2000
```

This line of code assigns the value 2000 to the variable flask_port. This value seems to represent the port number that the Flask application will run on. Port numbers are used to identify specific processes on a server or host machine. In this case, the Flask application is likely configured to listen on port 2000 for incoming requests.

you might see more commonly used port numbers like 80 (HTTP), 443 (HTTPS), or custom ports for specific services. The choice of port number depends on the use case and whether there are any conflicting services using the same port.

```
def check(string, sub_str):
```

This defines a function named check that takes two parameters: string and sub_str. This function is meant to check if the sub_str exists within the string.

```
if (string.find(sub_str) == -1):
```

This line uses the find method to search for the sub_str within the string. If sub_str is not found (indicated by a return value of -1 from the find method), the condition is true.

```
return string, "501 Error in Insert, Check Logs"
```

If the condition in the previous line is true (meaning sub_str is not found in string), the function returns a tuple containing the original string and a string message indicating an error with the HTTP status code 501. This message suggests checking logs for further information.

```
else:
```

If the condition in the second line is false (meaning sub_str is found in string), the code within this block will be executed.

```
return string, "200 Successfully inserted"
```

In this case, the function returns a tuple containing the original `string` and a string message indicating success with the HTTP status code `200`.

```
@app.route('/')
```

This is a decorator that registers the following function as a route handler for the root URL ("/"). In Flask, route handlers are functions that are executed when a specific URL is accessed.

```
def hello():
```

This defines a function named `hello()` which will be executed when the root URL ("/") is accessed.

```
"""Return a friendly HTTP greeting."""
```

This is a docstring, providing a brief description of what the `hello()` function does. It's used to document the purpose of the function.

```
return 'Welcome to BravisaPy!'
```

This line is the actual response returned by the `hello()` function when the root URL is accessed. It sends a string containing the message "Welcome to BravisaPy!" as the response to the client.

This code defines a function `check()` for verifying the existence of a substring in a string and a Flask web application with a route handler `hello()` that responds with a welcoming message when the root URL is accessed.

```
#####run_script#####  
###
```

```
def run_scripts():  
    requested_date = request.get_json()  
    print("Requested date: ", requested_date, flush=True)
```

the function takes no arguments and starts by using the request object from Flask to retrieve JSON data sent by the client. It prints the requested date to the console.


```
print("In the runscript function", requested_date['date'], flush=True)
curr_date = requested_date['date'].split('-')
```

The function continues by printing a message and extracting the date from the JSON data. The date is expected to be in the format "YYYY-MM-DD". It splits the date using the '-' separator to create a list containing year, month, and day components.

```
curr_date = datetime.date(int(curr_date[0]), int(curr_date[1]), int(curr_date[2]))
print("Running for {}".format(str(curr_date)), flush=True)
```

The code then converts the extracted year, month, and day components into integers and uses them to create a datetime.date object. It prints the formatted date that will be processed.

```
end_offset = BMonthEnd()
month_end = end_offset.rollforward(curr_date)
month_end = month_end.to_pydatetime()
month_end = month_end.date()
```

the code calculates the last day of the current month using the BMonthEnd offset from the pandas library. It rolls the end date forward to the end of the month, converts it to a datetime object, and then extracts the date.

```
start_offset = BMonthBegin()
month_start = start_offset.rollback(curr_date)
month_start = month_start.to_pydatetime()
month_start = month_start.date()
```

this part of the code calculates the first working day of the month using the BMonthBegin offset. It rolls the start date back to the beginning of the month, converts it to a datetime object, and then extracts the date.

```
print("\t\tSTART", flush=True)
```

A message is printed to the console indicating the start of the processing.

- The following block of if and elif statements determines which scripts to run based on the current date and its day of the week:
- If it's Friday and also the first working day of the month, the function runs `friday_btt(curr_date)`.

- If it's the last day of the month, the function runs `month_end(curr_date)`.
- If it's the first working day of the month and the day of the week is less than 4 (indicating Monday to Wednesday), the function runs `daily_btt(curr_date)`.
- If it's Friday, the function runs `friday_scripts(curr_date)`.
- If it's a weekday from Monday to Thursday, the function runs `daily_scripts(curr_date)`.
- If it's Saturday, the function runs `saturday_fb(curr_date)`.

```
print("\t\tDONE", flush=True)
```

After the appropriate scripts have been run, a message is printed to the console indicating the completion of the processing.

```
return
```

```
#####run_scripts_history#####  
###
```

the `run_scripts()` function processes a requested date and determines which scripts to run based on the day of the week and the position within the month. It uses the pandas library's date offset functionality to calculate the first and last days of the month. The specific script functions are called based on the conditions mentioned.

```
@app.route('/run_scripts_history')  
def run_scripts_history():
```

This is a Flask route decorator that binds the `run_scripts_history()` function to the URL path `/run_scripts_history`.

```
sdate = datetime.date(2021, 5, 7)  
edate = datetime.date(2021, 7, 1)
```

These lines set the start date `sdate` and end date `edate` for the range of dates over which the scripts will be executed.

```
date_list = [sdate+datetime.timedelta(days=x) for x in range((edate-sdate).days)]  
print("Running script from {} to {}".format(str(sdate), str(edate)), flush = True)
```

This creates a list of dates within the specified range by using a list comprehension and the `datetime.timedelta` function. It prints a message indicating the range of dates that the scripts will be executed.

```
for curr_date in date_list:
    end_offset = BMonthEnd()
    month_end = end_offset.rollforward(curr_date)
    month_end = month_end.to_pydatetime()
    month_end = month_end.date()
```

Inside the loop, for each `curr_date` in the `date_list`, this block calculates the last day of the current month using the same method as in the previous function.

```
start_offset = BMonthBegin()
month_start = start_offset.rollback(curr_date)
month_start = month_start.to_pydatetime()
month_start = month_start.date()
```

his block calculates the first working day of the month using the same method as in the previous function.

The following block of `if` and `elif` statements determines which scripts to run based on the current date and its day of the week, similar to the `run_scripts()` function.

After each date is processed, a message is printed indicating the completion of the process for that date.

```
print("Returning from run_scripts_history")
return "Finished"
```

After all dates have been processed, a message is printed to indicate that the function is returning, and the string "Finished" is returned as the response.

The `run_scripts_history()` function processes a range of dates, calculates the first and last days of each month, and runs specific script functions based on the day of the week and position within the month. The scripts are similar to those in the `run_scripts()` function.

```
##### BTT
#####
```

```
@app.route('/btt_list_fetch')  
def btt_list_fetch():
```

This is a Flask route decorator that binds the `btt_list_fetch()` function to the URL path `/btt_list_fetch`.

```
    report = 'BTT_LIST_FETCH'  
    start = time.time()
```

Here, a variable `report` is assigned the string value `'BTT_LIST_FETCH'`, and the `start` variable is assigned the current timestamp in seconds using `time.time()` to track the start time of the operation.

```
    conn = DB_Helper().db_connect()  
    cur = conn.cursor()
```

These lines establish a connection to the database using the `db_connect()` method from the `DB_Helper` class and create a cursor object for executing SQL queries.

```
    try:  
        curr_date = datetime.date(2021, 3, 1)  
        btt_list.main(curr_date)
```

Inside the `try` block, the script is set to fetch data for the specified `curr_date` (March 1, 2021) using the `btt_list.main()` function. This function appears to retrieve some data related to a BTT list.

```
    end = time.time()  
    statuscode = "SUCCESS in " + str(end - start) + " seconds"  
    timestamp = datetime.datetime.now()  
    timestamp = timestamp.strftime("%c")
```

After the successful execution of the `btt_list.main()` function, the `end` variable is assigned the current timestamp using `time.time()`, and a success message with the elapsed time is created for the `statuscode` variable. Additionally, the current timestamp is formatted using `strftime("%c")` and stored in the `timestamp` variable.

```
cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES (' + str(timestamp) + ',' + str(statuscode) + ',' + str(report) + ')")
conn.commit()
```

This block inserts a new row into a table named "ErrorLog" with the timestamp, status code, and report name. The changes are committed to the database.

```
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
```

In case an exception occurs during the execution of the script, the except block is triggered. The exception message is stored in the statuscode variable, and the current timestamp is formatted and stored in the timestamp variable.

```
cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES (' + str(timestamp) + ',' + str(statuscode) + ',' + str(report) + ')")
conn.commit()
raise e
```

Similar to the try block, an error log entry is created in the database for the exception. Then, the exception is re-raised using raise e to propagate the error.

```
conn.close()
return str(statuscode)
```

the database connection is closed using conn.close(), and the status code (success or exception message) is returned as a string.

The btt_list_fetch() function appears to fetch data for a specific date, perform some operations on it, record the status and timestamp in a database, and return the status as a response.

```
@app.route('/btt_list_fetch_validate')
def btt_list_fetch_validate():
```

This is a Flask route decorator that binds the btt_list_fetch_validate() function to the URL path /btt_list_fetch_validate.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

let's go through the btt_list_fetch_validate() function line by line:

```
@app.route('/btt_list_fetch_validate')
def btt_list_fetch_validate():
```

This is a Flask route decorator that binds the btt_list_fetch_validate() function to the URL path /btt_list_fetch_validate.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines establish a connection to the database using the db_connect() method from the DB_Helper class and create a cursor object for executing SQL queries.

```
cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
details = cur.fetchone()
```

This line executes an SQL query to select the most recent STATUS value from the "ErrorLog" table, ordered by timestamp in descending order. The cur.fetchone() method retrieves the first result row from the query.

```
string = str(details)
status = check(string, 'SUCCESS')
```

The retrieved details value is converted to a string using str() and stored in the string variable. Then, the check() function is called with string and 'SUCCESS' as arguments to determine whether the string contains the substring 'SUCCESS'.

```
conn.close()
```

The database connection is closed using conn.close().

```
return str(statuscode)
```

The status value (either "501 Error in Insert, Check Logs" or "200 Successfully inserted") is returned as the response.

The `btt_list_fetch_validate()` function appears to fetch the most recent status from the database, validate whether it contains the string 'SUCCESS', and return the appropriate status response.

#####Fb insert on linux
#####

```
@app.route('/linux_fb_insert_01')
def linux_fb_insert_01():
```

This is a Flask route decorator that binds the `linux_fb_insert_01()` function to the URL path `/linux_fb_insert_01`.

```
report = 'FB01'
start = time.time()
```

This line initializes the `report` variable with the string 'FB01' and records the start time using the `time.time()` function.

```
fbname = FB_Helper().get_fb_name_one()
```

The `get_fb_name_one()` method from the `FB_Helper` class is called to retrieve.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines establish a database connection using the `db_connect()` method from the `DB_Helper` class and create a cursor object for executing SQL queries.

```
try:
    FBLinux().fb_insert_01(fbname,conn, cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

Inside the try block, the `fb_insert_01()` method from the `FBLinux` class is called with `fbname`, `conn`, and `cur` as arguments to insert data. If the insertion is successful, the end

time is recorded, and a success status message is generated. A new entry is inserted into the "ErrorLog" table in the database with the timestamp, status code, and report details. If an exception occurs during the process, the exception is caught, and an error entry is inserted into the "ErrorLog" table. The exception is then re-raised.

```
conn.close()
```

The database connection is closed using `conn.close()`.

```
return str(statuscode)
```

The `statuscode` (either a success message or an error message) is returned as the response.

The `linux_fb_insert_01()` function appears to insert data into the database and records the status of the operation in the error log table.

let's go through the `linux_fb_insert_02()` function line by line:

```
@app.route('/linux_fb_insert_02')
def linux_fb_insert_02():
```

This is a Flask route decorator that binds the `linux_fb_insert_02()` function to the URL path `/linux_fb_insert_02`.

```
report = 'FB02'
start = time.time()
```

This line initializes the `report` variable with the string 'FB02' and records the start time using the `time.time()` function.

```
fbname = FB_Helper().get_fb_name_two()
```

The `get_fb_name_two()` method from the `FB_Helper` class is called to retrieve.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines establish a database connection using the `db_connect()` method from the `DB_Helper` class and create a cursor object for executing SQL queries.


```
try:
    FBLinux().fb_insert_02(fbname, conn, cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")

    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:

    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

Inside the try block, the fb_insert_02() method from the FBLinux class is called with fbname, conn, and cur as arguments to insert data related. If the insertion is successful, the end time is recorded, and a success status message is generated. A new entry is inserted into the "ErrorLog" table in the database with the timestamp, status code, and report details. If an exception occurs during the process, the exception is caught, and an error entry is inserted into the "ErrorLog" table. The exception is then re-raised.

```
conn.close()
```

The database connection is closed using conn.close().

```
return str(statuscode)
```

The status code (either a success message or an error message) is returned as the response.

The linux_fb_insert_02() function appears to insert data related into the database and records the status of the operation in the error log table.

the linux_fb_insert_03() function:

```
@app.route('/linux_fb_insert_03')
def linux_fb_insert_03():
```

This is another Flask route decorator that associates the linux_fb_insert_03() function with the URL path /linux_fb_insert_03.

```
report = 'FB03'
start = time.time()
```

Here, report is set to 'FB03', and the start time is recorded using time.time().

```
fbname = FB_Helper().get_fb_name_three()
```

The `get_fb_name_three()` method from the `FB_Helper` class is called to retrieve.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

A database connection is established using `db_connect()` from the `DB_Helper` class, and a cursor is created for executing SQL queries.

```
try:
    FBLinux().fb_insert_03(fbname, conn, cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")

    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

Inside the try block, the `fb_insert_03()` method from the `FBLinux` class is called with the retrieved `fbname`, database connection (`conn`), and cursor (`cur`). If successful, the end time is noted, and a success status message is generated. An entry is inserted into the "ErrorLog" table with the timestamp, status code, and report details. If an exception occurs, it's caught, an error entry is added to the log, and the exception is re-raised.

```
conn.close()
```

The database connection is closed using `conn.close()`.

```
return str(statuscode)
```

Just like the previous functions, `linux_fb_insert_03()` seems to insert data related into the database and records the status of the operation in the error log table.

```
#####FB Insert #####
```

```
@app.route('/fb_history')
def fb_history():
```

This is a Flask route decorator that associates the `fb_history()` function with the URL path `/fb_history`.

```
    report = 'FB01'
    start = time.time()
```

Here, `report` is set to 'FB01', and the start time is recorded using `time.time()`.

```
    fbname_one = FB_Helper().get_fb_name_one()
    fbname_two = FB_Helper().get_fb_name_two()
    fbname_three = FB_Helper().get_fb_name_three()
```

Names or identifiers related to stock investment data (referred to as "fb") are retrieved using methods from the **FB_Helper** class.

```
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
```

A database connection is established using `db_connect()` from the **DB_Helper** class, and a cursor is created for executing SQL queries.

```
    try:
        FB_Insert().fb_insert_03(fbname_three,conn, cur)
        FB_Insert().fb_insert_01(fbname_one,conn, cur)
        FB_Insert().fb_insert_02(fbname_two,conn, cur)

        end = time.time()
        statuscode = "SUCCESS in " + str(end-start) + " seconds"
        timestamp = datetime.datetime.now()
        timestamp = timestamp.strftime("%c")

        cur.execute("INSERT INTO public.\\"ErrorLog\\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
        conn.commit()
    except Exception as e:

        statuscode = e
        timestamp = datetime.datetime.now()
        timestamp = timestamp.strftime("%c")
        cur.execute("INSERT INTO public.\\"ErrorLog\\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
        conn.commit()

    raise e

conn.close()
return str(statuscode)
```

Inside the try block, specific functions like `fb_insert_03()`, `fb_insert_01()`, and `fb_insert_02()` are called from the **FB_Insert** class, passing the appropriate fb names, database connection (`conn`), and cursor (`cur`). If successful, the end time is noted, and a success status message

is generated. An entry is inserted into the "ErrorLog" table with the timestamp, status code, and report details. If an exception occurs, it's caught, an error entry is added to the log, and the exception is re-raised.

```
conn.close()
```

The database connection is closed using `conn.close()`.

```
return str(statuscode)
```

The function returns the `statuscode` as a response.

It appears that the `fb_history()` function is involved in processing stock investment data and logging the results of various data insertions or operations related to that data.

```
@app.route('/fb_insert_01')
def fb_insert_01():
```

This Flask route decorator associates the `fb_insert_01()` function with the URL path `/fb_insert_01`.

```
report = 'FB01'
start = time.time()
```

The report is set to 'FB01', and the start time is recorded using `time.time()`.

```
fbname = FB_Helper().get_fb_name_one()
```

The `fbname` is assigned the value obtained from the `get_fb_name_one()` method of the `FB_Helper` class.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

A database connection is established using the `db_connect()` method of the `DB_Helper` class, and a cursor is created for executing SQL queries.

```
try:
    FB_Insert().fb_insert_01(fbname, conn, cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")

    cur.execute("INSERT INTO public.\\"ErrorLog\\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\\"ErrorLog\\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

Inside the try block, the fb_insert_01() method from the **FB_Insert** class is called, passing the fbname, database connection (conn), and cursor (cur). If successful, the end time is noted, and a success status message is generated. An entry is inserted into the "ErrorLog" table with the timestamp, status code, and report details. If an exception occurs, it's caught, an error entry is added to the log, and the exception is re-raised.

```
conn.close()
```

The database connection is closed using conn.close().

```
return str(statuscode)
```

The function returns the statuscode as a response.

```
@app.route('/fb_insert_01')
```

This is a route decorator in Flask. It specifies that this function should be triggered when the client accesses the URL path '/fb_insert_01'.

```
def fb_insert_01():
```

This function is defined to handle the logic associated with the route '/fb_insert_01'.

```
    report = 'FB01'
    start = time.time()
```

These lines set up a variable report with the value 'FB01' and record the current time in start.

```
    fbname = FB_Helper().get_fb_name_one()
```

This line retrieves the value of fbname by calling the get_fb_name_one() method from the FB_Helper class.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines establish a database connection using the db_connect method from the DB_Helper class and create a cursor to interact with the database.

```
try:
    FB_Insert().fb_insert_01(fbname,conn, cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")

    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:

    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

- This block of code tries to execute the fb_insert_01 method from the FB_Insert class, passing the fbname, conn, and cur as arguments. It seems to insert data related to 'FB01'.
- If successful, it calculates the time taken for the operation and constructs a success message with the time taken. It also logs this information in the database.
- If an exception occurs during the execution of the fb_insert_01 method, it captures the exception, logs an error message along with the current timestamp and the status code, and raises the exception again.

```
conn.close()
```

This line closes the database connection.

```
return str(statuscode)
```

Finally, the function returns a string representation of the statuscode variable, which indicates the success or error status of the operation.

Overall, this route seems to be responsible for inserting financial data related to 'FB01' into the database and recording the status of the operation in the database.

```
@app.route('/fb_insert_01_validate')
```

This route decorator specifies that this function will handle requests made to the URL path '/fb_insert_01_validate'.

```
def fb_insert_01_validate():
```

This function is defined to handle the logic associated with the route '/fb_insert_01_validate'.

```
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
```

These lines establish a database connection using the db_connect method from the DB_Helper class and create a cursor to interact with the database.

```
    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
    details = cur.fetchone()
```

This line executes a SQL query to retrieve the most recent status from the "ErrorLog" table in the database. It retrieves the value of the "STATUS" column for the latest entry.

```
    string = str(details)
    status = check(string, 'SUCCESS')
```

The fetched status value is converted to a string, and then the check function is called to determine if the status contains the substring 'SUCCESS' or not. The check function seems to return a status message based on whether the provided substring is found in the string or not.

```
    conn.close()
```

This line closes the database connection.

```
    return status
```

Finally, the function returns the value of the status variable, which is either a success message or an error message based on the result of the check function. Overall, this route is responsible for retrieving the most recent status from the database and checking if it contains the substring 'SUCCESS'. It then returns a status message based on the outcome of the check.

```
@app.route('/fb_insert_02')
```

This route decorator specifies that this function will handle requests made to the URL path '/fb_insert_02'.

```
def fb_insert_02():
```

This function is defined to handle the logic associated with the route '/fb_insert_02'.

```
    report = 'FB02'
    start = time.time()
```

These lines set up variables for the report name and the start time (for measuring execution time).

```
    fbname = FB_Helper().get_fb_name_two()
```

This line gets the financial data file name using the get_fb_name_two method from the FB_Helper class. It's either retrieved from a method or calculated based on your application's logic.

```
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
```

These lines establish a database connection using the db_connect method from the DB_Helper class and create a cursor to interact with the database.

```
try:
    FB_Insert().fb_insert_02(fbname,conn,cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "', '" + str(statuscode) + "', '" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "', '" + str(statuscode) + "', '" + str(report) + "')")
    conn.commit()
    raise e
```

This block of code contains a try-except structure to handle the main logic of inserting financial data into the database. The code inside the try block attempts to execute the fb_insert_02 method from the FB_Insert class with the provided financial data file name, database connection, and cursor. If successful, it calculates the execution time, prepares a success message, inserts an entry into the error log with a timestamp, status, and report information, and then commits the changes to the database. If an exception occurs during

this process, the except block captures the exception, logs an error entry into the error log, commits the changes, and raises the exception again.

```
conn.close()
```

This line closes the database connection.

```
return str(statuscode)
```

Finally, the function returns a string representation of the status code, which indicates whether the insertion process was successful or encountered an error.

Overall, this route is responsible for inserting financial data into the database, measuring execution time, logging the result in the error log, and returning a status message.

```
@app.route('/fb_insert_02_validate')
```

This route decorator specifies that this function handles requests to the URL path '/fb_insert_02_validate'.

```
def fb_insert_02_validate():
```

This function validates the status of the operation performed by the 'fb_insert_02' script.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a database connection and creates a cursor for executing SQL queries.

```
cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
details = cur.fetchone()
```

Executes an SQL query to fetch the most recent status from the 'ErrorLog' table, ordered by timestamp in descending order.

'fetchone()' retrieves the first row of the query result.

```
string = str(details)
```

Converts the fetched result into a string format.

```
status = check(string, 'SUCCESS')
```

Calls the 'check' function with the fetched status and 'SUCCESS' as parameters to determine if the status is a success or error message.

```
conn.close()
```

Closes the database connection.

```
return status
```

Returns the determined status (success or error) from the function.

this code fetches the most recent status of an operation (likely related to 'fb_insert_02') from the database, checks whether it indicates success or an error, and returns the corresponding status.

```
@app.route('/fb_insert_03')
```

This route decorator specifies that this function handles requests to the URL path '/fb_insert_03'.

```
def fb_insert_03():
```

This function is responsible for inserting data related to 'FB03' into the database.

```
report = 'FB03'  
start = time.time()
```

Initializes the report name and starts measuring the time for performance tracking.

```
fbname = FB_Helper().get_fb_name_three()
```

Obtains the file name or data source related to 'FB03' using the 'get_fb_name_three' method from the 'FB_Helper' class.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a database connection and creates a cursor for executing SQL queries.

```
try:
    FB_Insert().fb_insert_03(fbname,conn,cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" + str(report) + '\"')
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" + str(report) + '\"')
    conn.commit()
    raise e
```

- The code within the 'try' block attempts to insert data related to 'FB03' using the 'fb_insert_03' method of the 'FB_Insert' class.
- If the operation is successful, the status is recorded as a success along with the timestamp and report name in the 'ErrorLog' table.
- If an exception occurs, the status is recorded as an error, and the exception is raised again after recording the details.

```
conn.close()
```

Closes the database connection.

```
return str(statuscode)
```

Returns the status of the operation ('SUCCESS' or an error message) as a string.

this code handles the insertion of data related to 'FB03' into the database, records the status of the operation in the 'ErrorLog' table, and returns the status as a response.

```
@app.route('/fb_insert_03_validate')
```

This route decorator specifies that this function handles requests to the URL path '/fb_insert_03_validate'.

```
def fb_insert_03_validate():
```

This function is responsible for validating the status of the 'FB03' data insertion process.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a database connection and creates a cursor for executing SQL queries.

```
cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
details = cur.fetchone()
print(details)
string = str(details)
```

- Executes a SQL query to retrieve the most recent status entry from the 'ErrorLog' table.
- The 'fetchone()' method retrieves one row from the query result, which is a tuple containing the status value.
- The 'print(details)' statement prints the retrieved status tuple.

```
status = check(string, 'SUCCESS')
```

- Calls the 'check' function (not shown in the provided code) to compare the retrieved status value with the expected 'SUCCESS' value.
- The 'string' parameter contains the retrieved status value, and 'SUCCESS' is the expected status.
- The result of the comparison is assigned to the 'status' variable.

```
conn.close()
```

Closes the database connection.

```
return status
```

Returns the validation result ('SUCCESS' or not) as a response.

this code handles the validation of the status of the 'FB03' data insertion process by retrieving the most recent status entry from the 'ErrorLog' table and comparing it with the expected 'SUCCESS' value. The validation result is then returned as a response.

FB History Insert

let's go through the code for the /fb_insert_history_data route:

```
@app.route('/fb_insert_history_data')
def fb_insert_history_data():
```

This Flask route decorator associates the fb_insert_history_data() function with the URL path /fb_insert_history_data.

```
timer_start = time.time()
start = datetime.datetime.strptime("23-09-2019", "%d-%m-%Y")
end = datetime.datetime.strptime("24-09-2019", "%d-%m-%Y")
```

A timer is started using time.time(). The start and end dates are defined as datetime objects based on specific date strings.

```
date_generated = [start + datetime.timedelta(days=x) for x in range(0, (end-start).days)]
```

A list called date_generated is created using a list comprehension that generates dates between the start and end dates by incrementing them day by day using datetime.timedelta.

```
for date in date_generated:
    fb_insert_history.fb_list_date(date)
```

A loop iterates through the date_generated list, and for each date, the fb_list_date() function from the fb_insert_history module (presumably) is called. This function seems to insert historical data for a specific date into the database.

```
timer_end = time.time()
```

The timer is stopped using time.time().

```
return "Completed FB history data insert. Time Taken: " + str(timer_end-timer_start) + " seconds"
#####
```

The function returns a message indicating the completion of the historical data insertion and the time taken for the operation.

This route appears to be responsible for inserting historical data related to "fb" (possibly financial data) into the database for a range of dates.

OHLC Insert#####

This code defines a route handler for the URL path /ohlc_fetch. Let's go through it step by step:

```
@app.route('/ohlc_fetch')
def ohlc_fetch():
```

This route handles HTTP GET requests to the /ohlc_fetch URL path.

```
    report = 'OHLC'
    start = time.time()
```

- A variable report is assigned the value 'OHLC'. It seems to be used for logging purposes.
- The current time is captured using time.time() and assigned to the variable start. This will be used to measure the execution time of the code block.

```
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
```

- The code creates a database connection using DB_Helper().db_connect().
- A cursor is created using conn.cursor(). Cursors are used to execute SQL commands and retrieve results.

```
# Try catch here
try:
    ohlc.main()
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

- Inside a try-except block, the code attempts to execute the ohlc.main() function. This function seems to handle the main logic for fetching OHLC data.

If the ohlc.main() function executes successfully:

- The end time is captured using time.time() and assigned to the variable end.
- A status message is created with the execution time and assigned to statuscode.

- The current timestamp is captured using `datetime.datetime.now()` and formatted as a string in the variable `timestamp`.
- An SQL query is constructed to insert an entry into the "ErrorLog" table with the timestamp, status code, and report details.
- The changes are committed to the database using `conn.commit()`.
- If an exception occurs during the execution of the `ohlc.main()` function:
- The exception is caught and assigned to the variable `e`.
- The same process as above is repeated to create a status message, timestamp, and SQL query for logging the error.
- The changes are committed to the database.
- The exception is re-raised using **`raise e`**.

```
conn.close()
return str(statuscode)
```

- The database connection is closed using `conn.close()`.
- The function returns a string representation of the `statuscode`.

In summary, this route fetches OHLC data using the `ohlc.main()` function, logs the operation's status and execution time in the database, and handles exceptions by logging errors and re-raising exceptions.

```
@app.route('/ohlc_fetch_validate')
def ohlc_fetch_validate():
    conn = DB_Helper().db_connect()
    cur = conn.cursor()

    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

- This route handles HTTP GET requests to the `/ohlc_fetch_validate` URL path.
- It starts by establishing a database connection using the `db_connect()` method from the `DB_Helper` class. Then, it creates a cursor for database operations.
- The code executes an SQL query to fetch the most recent status from the "ErrorLog" table. The query fetches the status column and orders the results by timestamp in descending order, limiting the result to just one row using `LIMIT 1`.
- The fetched status details are obtained using `cur.fetchone()`, which retrieves the first row of the query result. The status details are then converted to a string representation.

- The check function is called with the status string and the substring 'SUCCESS'. The check function seems to return a status message indicating the success or failure of an operation.
- Once the status message is obtained, the database connection is closed using `conn.close()`.
- Finally, the status message is returned, which indicates the validation result of the last operation stored in the "ErrorLog" table.

This route serves as a validation mechanism to check the success or failure status of the most recent operation that has been logged in the "ErrorLog" table.

Here's a line-by-line explanation of the `/index_ohlc_fetch` route code:

```
@app.route('/index_ohlc_fetch')
def index_ohlc_fetch():
```

This line defines a new route using the `@app.route` decorator. It specifies that the route will handle HTTP GET requests to the URL path `/index_ohlc_fetch`

```
    report = 'IndexOHLC'
    start = time.time()
```

These lines initialize variables for later use. The `report` variable is assigned the string 'IndexOHLC', representing the name of the report being processed. The `start` variable records the current time using `time.time()` to measure the duration of the operation.

```
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
```

These lines establish a connection to the database using the `db_connect()` method from the `DB_Helper` class. The connection object is stored in the `conn` variable. Additionally, a cursor object is created for executing SQL queries, and it's stored in the `cur` variable

```
    try:
        index_ohlc.main()
```

This line starts a try block. Inside the block, the `index_ohlc.main()` function is called. Presumably, this function fetches or processes index OHLC data.

```
        end = time.time()
        statuscode = "SUCCESS in " + str(end - start) + " seconds"
        timestamp = datetime.datetime.now()
        timestamp = timestamp.strftime("%c")
```


These lines calculate the time taken for the operation by recording the current time (end). The statuscode variable is assigned a string indicating the success of the operation and the time taken. The current timestamp is obtained using datetime.datetime.now() and formatted as a string in a human-readable format using strftime("%c").

```
cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
conn.commit()
```

This line executes an SQL INSERT statement using the cursor (cur) to log the operation's status and details into the "ErrorLog" table in the database. The timestamp, status code, and report name are inserted into respective columns. The conn.commit() call is used to make the changes permanent in the database.

```
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

This block handles exceptions that might occur during the operation. If an exception is caught, the statuscode variable is assigned the exception details. Similar to the previous case, the timestamp, status code, and report name are logged into the "ErrorLog" table. The exception is then raised again to propagate it up the stack.

```
conn.close()
```

This line closes the database connection.

```
return str(statuscode)
```

Finally, the function returns the statuscode as a response. This status message indicates whether the operation was successful or if an exception occurred.

```
@app.route('/index_ohlc_fetch_validate')
def index_ohlc_fetch_validate():
    conn = DB_Helper().db_connect()
    cur = conn.cursor()

    cur.execute('SELECT \\"STATUS\\" FROM public.\"ErrorLog\" ORDER BY \\"TIMESTAMP\\" DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

This route decorator specifies that this function handles requests to the URL path '/index_ohlc_fetch_validate'.

This function is responsible for validating the status of the 'index_ohlc_fetch' process.

Establishes a database connection and creates a cursor for executing SQL queries.

- Executes a SQL query to retrieve the most recent status entry from the 'ErrorLog' table.
- The 'fetchone()' method retrieves one row from the query result, which is a tuple containing the status value.
- The retrieved status tuple is converted to a string using 'str(details)' and stored in the 'string' variable.

Closes the database connection.

Returns the validation result ('SUCCESS' or not) as a response.

this code handles the validation of the status of the 'index_ohlc_fetch' process by retrieving the most recent status entry from the 'ErrorLog' table and comparing it with the expected 'SUCCESS' value. The validation result is then returned as a response.

```
@app.route('/index_ohlc_backdate')
```

This is a route decorator in Flask. It indicates that this function should be triggered when a client accesses the URL path '/index_ohlc_backdate'.

```
def index_ohlc_backdate():
```

This function is defined to handle the logic associated with the route '/index_ohlc_backdate'.

```
start = time.time()
```

This line records the current time using the time.time() function. This will be used to calculate the time taken for the process.

```
index_ohlc.history_insert()
```

This line calls **the history_insert() function from the index_ohlc module**. It seems to be responsible for inserting historical index OHLC (Open, High, Low, Close) data into the database.

```
end = time.time()
```

This line records the current time again after the history_insert() function has completed.

```
return "Index OHLC history insert completed in " + str(end - start)
```

This line constructs a string containing the time taken for the historical data insertion. It calculates the difference between the end time and the start time, converts it to a string, and includes it in the returned message.

Overall, this route seems to trigger the insertion of historical index OHLC data into the database and returns a message indicating the time taken for the operation.

```
@app.route('/ohlc_backdate')
```

This is a route decorator in Flask. It indicates that this function should be triggered when a client accesses the URL path '/ohlc_backdate'.

```
def ohlc_backdate():
```

This function is defined to handle the logic associated with the route '/ohlc_backdate'.

```
start = datetime.datetime.strptime("04-02-2020", "%d-%m-%Y")
end = datetime.datetime.strptime("07-02-2020", "%d-%m-%Y")
```

These lines create two datetime objects, 'start' and 'end', representing the starting and ending dates respectively.

```
date_generated = [start + datetime.timedelta(days=x) for x in range(0, (end-start).days)]
```

This line generates a list of datetime objects using a list comprehension. It adds a 'datetime.timedelta' for each day in the range between 'start' and 'end'. This generates a list of dates within the specified range.

```
ohlc.ohlc_date_join(date_generated)
```

This line calls the function 'ohlc_date_join' from the 'ohlc' module, passing the list of generated dates as an argument. This function appears to perform some operation related to OHLC data for the given dates.

```
return "Completed ohlc backdate"
```

This line returns a simple message indicating that the backdating of OHLC data is completed. Overall, this route seems to perform a backdating operation for OHLC data based on the specified date range and then returns a completion message.

Split/Bonus

```
@app.route('/splitbonus')
```

This route decorator specifies that this function handles requests to the URL path '/splitbonus'.

```
def splitbonus():
```

This function is responsible for performing a split-bonus related operation and logging its status.

```
report = 'SplitBonus'
start = time.time()
```

- Initializes the 'report' variable with the value 'SplitBonus', which will be used for logging purposes.
- Records the start time using the 'time.time()' function to measure the time taken for the operation.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a database connection and creates a cursor for executing SQL queries.

```
try:
    split_bonus.main()
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

- A try-except block is used to handle exceptions that might occur during the operation.
- Inside the try block:
 - The 'split_bonus.main()' function is called to perform the split-bonus operation.
 - The end time is recorded and the time taken for the operation is calculated.
 - The current timestamp is obtained using 'datetime.datetime.now()' and formatted as a string.

- An SQL query is executed to insert a log entry into the 'ErrorLog' table with the timestamp, status, and report.
- The connection is committed to save the changes to the database.
- Inside the except block:
 - ◆ If an exception occurs, the status is set to the exception message.
 - ◆ The current timestamp is obtained and formatted.
 - ◆ An SQL query is executed to insert an error log entry with the timestamp, status, and report.
 - ◆ The connection is committed.
 - ◆ The exception is re-raised to propagate the error.

```
conn.close()
```

Closes the database connection.

```
return str(statuscode)
```

Returns the status of the operation (success or error) along with the time taken.

This code handles the split-bonus operation, records the start and end times, logs the operation's status and details, and returns the status along with the time taken for the operation.

```
@app.route('/splitbonus_validate')
```

This route decorator specifies that this function handles requests to the URL path '/splitbonus_validate'.

```
def splitbonus_validate():
```

This function is responsible for validating the status of a split-bonus operation.

```
conn = DB_Helper().db_connect()  
cur = conn.cursor()
```

Establishes a database connection and creates a cursor for executing SQL queries.

```
cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
details = cur.fetchone()
string = str(details)
```

- Executes an SQL query to select the most recent status from the 'ErrorLog' table in the database.
- Fetches the result using 'cur.fetchone()'.
- Converts the fetched details to a string representation using 'str(details)'.

```
status = check(string, 'SUCCESS')
```

- Calls a function named 'check' to compare the fetched status string with the string 'SUCCESS'.
- If the fetched status matches 'SUCCESS', the 'status' variable will be set to 'True', indicating success. Otherwise, it will be set to 'False'.

```
conn.close()
```

Closes the database connection.

```
return status
```

Returns the value of the 'status' variable, indicating whether the most recent split-bonus operation was successful or not.

This code fetches the most recent status of a split-bonus operation from the database, checks if the status is 'SUCCESS', and returns a boolean value indicating whether the operation was successful or not.

PE Insert#####

```
@app.route('/pe_backdate')
```

This route decorator specifies that this function handles requests to the URL path '/pe_backdate'.

```
def pe_backdate():
```

This function is responsible for inserting historical PE (Price-to-Earnings ratio) data.

```
start = time.time()
```

Records the current time as the start time for measuring the execution time of the following code.

```
PE.history_pe()
```

Calls the method `history_pe()` to perform the insertion of historical PE data.

```
end = time.time()
```

Records the current time as the end time after the execution of the `history_pe()` method.

```
return "PE history insert completed in " +str(end-start)+ " seconds"
```

Returns a string indicating the completion of the PE history insertion, along with the time taken for the process to complete.

this code defines a route to insert historical PE data, measures the time taken for the insertion process, and returns a message indicating the completion of the insertion along with the time taken.

```
@app.route('/pe')
```

This route decorator indicates that this function is intended to handle requests to the URL path `'/pe'`.

```
def pe_curr():
```

This function is responsible for fetching and inserting the current PE (Price-to-Earnings ratio) data.

```
report = 'PE'
```

A variable to store the report name, which is `'PE'`.

```
start = time.time()
```

Records the current time as the start time for measuring the execution time of the following code.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a connection to the database using the db_connect() method from the DB_Helper class and creates a cursor for executing SQL queries.

```
try:
    PE.current_pe()
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

- This try block attempts to execute the current_pe() method from the PE class, which fetches and inserts the current PE data.
- If the execution is successful, it records the end time, calculates the time taken, and constructs a status message.
- It also captures the current timestamp and constructs an SQL query to insert an entry into the error log table, indicating the success or failure of the process.
- If an exception occurs during execution, it captures the exception, captures the current timestamp, constructs an SQL query to insert an entry into the error log table indicating the failure, commits the transaction, and then re-raises the exception.

```
conn.close()
```

Closes the database connection.

```
return str(statuscode)
```

- Returns the status message indicating the success or failure of the process along with the time taken.
- This code defines a route to fetch and insert current PE data, measures the time taken for the process, logs the status of the process in the error log table, and returns a status message.


```
@app.route('/pe_validate')
```

This route decorator indicates that this function is intended to handle requests to the URL path '/pe_validate'.

```
def pe_validate():
```

This function is responsible for validating the success status of the 'PE' data insertion process.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a connection to the database using the db_connect() method from the DB_Helper class and creates a cursor for executing SQL queries.

```
cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
```

Executes an SQL query to retrieve the most recent status entry from the error log table.

```
details = cur.fetchone()
```

Fetches the result of the executed query, which should be the most recent status entry.

```
string = str(details)
```

Converts the fetched status details into a string representation

```
status = check(string, 'SUCCESS')
```

Calls the check function (not shown in the provided code) to determine if the fetched status matches the expected 'SUCCESS' status.

```
conn.close()
```

Closes the database connection.

```
return status
```

Returns the result of the status check, indicating whether the 'PE' data insertion process was successful or not.

This code defines a route to validate the success status of the 'PE' data insertion process by querying the error log table, checking the status entry, and returning the validation result.

```
##### PRS Insert #####
```

```
@app.route('/prs_backdate')
```

This route decorator specifies that this function will handle requests to the URL path '/prs_backdate'.

```
def prs_backdate():
```

This function is responsible for performing backdated PRS history insertion.

```
start = time.time()
```

Records the current time as the starting point for measuring the time taken for the process.

```
start_date = datetime.datetime.strptime("15-02-2019", "%d-%m-%Y")
end_date = datetime.datetime.strptime("16-02-2019", "%d-%m-%Y")
```

Defines the start and end dates for the date range over which the PRS history insertion will be performed.

```
date_generated = [start_date + datetime.timedelta(days=x) for x in range(0, (end_date-start_date).days)]
```

Generates a list of dates within the specified date range.

```
PRS().history_range_insert(date_generated)
```

Calls the history_range_insert method from the PRS class to perform PRS history insertion for the generated dates.

```
end = time.time()
```

Records the current time as the endpoint for measuring the time taken for the process.

```
return "Completed PRS History Insert. Time Taken: "+ str(end-start) + " seconds"
```

Returns a message indicating the completion of the PRS history insertion along with the time taken for the process.

This code defines a route that triggers the backdated insertion of PRS history data for a specific date range and then reports the completion time of the process.

```
@app.route('/prs')
```

This route decorator specifies that this function will handle requests to the URL path '/prs'.

```
def prs_fetch():
```

This function is responsible for fetching PRS data and generating PRS daily data for a specific date.

```
report = 'PRS'  
start = time.time()
```

Initializes variables to record the current report name and starting time for measuring the execution time.

```
conn = DB_Helper().db_connect()  
cur = conn.cursor()
```

Establishes a database connection and cursor to interact with the database.

```
try:
    curr_date = datetime.date(2021,3,1)
    PRS().generate_prs_daily(curr_date,conn,cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
raise e
```

Tries to execute the following actions:

- Sets the curr_date variable to a specific date (March 1, 2021).
- Calls the generate_prs_daily method from the PRS class to generate PRS daily data for the specified date.
- Records the end time and calculates the execution time.
- Inserts a status log entry into the database with the execution timestamp, status code, and report name in case of success.
- In case of an exception, records the exception as the status code and inserts an error log entry into the database.

```
conn.close()
```

Closes the database connection.

```
return str(statuscode)
```

Returns the status code as a string indicating the success or failure of the PRS data fetching and daily generation process.

This code defines a route that fetches PRS data and generates PRS daily data for a specific date. It records the execution time and logs the status of the process in the database.

```
@app.route('/prs_validate')
```

This route decorator specifies that this function will handle requests to the URL path '/prs_validate'.

```
def prs_validate():
```

This function is responsible for validating the status of the PRS data fetching and daily generation process.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a database connection and cursor to interact with the database.

```
cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
details = cur.fetchone()
```

Executes a SQL query to retrieve the most recent status entry from the error log table. FETCHONE() retrieves the first row of the result set.

```
string = str(details)
status = check(string, 'SUCCESS')
```

- Converts the retrieved status details to a string.
- Calls the check() function with the retrieved status details and the expected status string 'SUCCESS' to determine if the process was successful.

```
conn.close()
```

Closes the database connection.

```
return status
```

Returns the status of the PRS data fetching and daily generation process as determined by the check() function.

this code defines a route that fetches the most recent status from the error log, checks if it matches the expected 'SUCCESS' status, and returns the validation result. This route can be used to validate the success of the PRS data processing.

EPS & TTM Insert

```
@app.route('/eps_backdate')
def get_eps_history():
    start = time.time()
    EPS().Generate_History_Reports()
    end = time.time()
    return "Compiled quarterly EPS list. Time Taken: "+ str(end-start) + " seconds"
```

This code defines a route and function to generate historical EPS (Earnings Per Share) reports. Here's an explanation of the code:

This route decorator specifies that this function will handle requests to the URL path '/eps_backdate'.

This function is responsible for generating historical EPS reports.

Records the starting time of the EPS report generation process.

Calls the `Generate_History_Reports()` method of the `EPS` class to initiate the process of generating historical EPS reports.

Records the ending time of the EPS report generation process.

this code defines a route that triggers the generation of historical EPS reports using the **`Generate_History_Reports()`** method of the `EPS` class. The route returns a message indicating the successful compilation of the EPS reports along with the time taken for the process.

```
@app.route('/eps')
```

This route decorator specifies that this function will handle requests to the URL path '/eps'.

```
def get_eps_today():
```

This function is responsible for generating daily EPS reports.

```
report = 'EPS'
```

The variable `report` is assigned the value 'EPS', which indicates that this is the EPS report being generated.

```
start = time.time()
```

Records the starting time of the daily EPS report generation process.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a database connection and creates a cursor to interact with the database.

```
try:
    EPS().Generate_Daily_Report(conn,cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "', '" + str(statuscode) + "', '" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\\"TIMESTAMP\\", \\"STATUS\\", \\"REPORT\\") VALUES ('" + str(timestamp) + "', '" + str(statuscode) + "', '" + str(report) + "')")
    conn.commit()
    raise e
```

- The try-except block is used to handle exceptions that might occur during the EPS report generation process.
- Within the try block, the Generate_Daily_Report() method of the EPS class is called to generate the daily EPS report.
- If successful, the end time is recorded, and a success status message is generated.
- A timestamp is also created, and an entry is added to the database's error log with the relevant information.
- If an exception occurs, an error status code is assigned, and an error entry is added to the error log.

```
conn.close()
```

Closes the database connection.

```
return str(statuscode)
```

Returns the status code indicating the success or failure of the daily EPS report generation.

This code defines a route that triggers the generation of daily EPS reports using the Generate_Daily_Report() method of the EPS class. The route returns a status message indicating the success or failure of the report generation process.

```
@app.route('/eps_validate')
def eps_validate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()

    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

- This route decorator specifies that this function will handle requests to the URL path '/eps_validate'.
- This function is responsible for validating the status of the EPS report generation process.
- Establishes a database connection and creates a cursor to interact with the database.
- Executes an SQL query to retrieve the latest status entry from the error log table in the database.
- The status information is fetched using the fetchone() method.
- Converts the retrieved status details into a string.
- Calls the check() function (presumably defined elsewhere) to validate whether the status is 'SUCCESS'.
- Closes the database connection.
- Returns the status of the EPS report generation process (e.g., 'SUCCESS' or another status).

This code defines a route that allows you to validate the status of the EPS report generation process by querying the error log table in the database for the latest status entry. The route returns the status of the process.

```
@app.route('/ttm_backdate')
def get_ttm_history():
    start = time.time()
    EPS().ttm_history_insert()
    end = time.time()
    return "Compiled quarterly EPS list. Time Taken: " + str(end-start) + " seconds"
```

This code defines a route and function for backdating the calculation and insertion of TTM (Trailing Twelve Months) EPS history into the database. Here's an explanation of the code:

- This route decorator specifies that this function will handle requests to the URL path '/ttm_backdate'.
- This function is responsible for calculating and inserting the TTM EPS history.
- Records the current time before starting the TTM history insertion process.
- Calls the ttm_history_insert() method of the EPS class to calculate and insert the TTM EPS history into the database.
- Records the current time after completing the TTM history insertion process.
- Returns a message indicating that the quarterly EPS list has been compiled and provides the time taken for the process.

This code defines a route that triggers the calculation and insertion of TTM EPS history into the database. The route returns a message along with the time taken for the process to complete.

SMR Insert

```
@app.route('/smr_backdate')
def get_smr_backdate():
    start = time.time()
    SMR().history_daterange_smr()
    end = time.time()
    return "Generated SMR report for backdate. Time Taken: "+ str(end-start) + "
seconds"
```

function for generating SMR (Stock Market Returns) reports for a specified date range. Here's an explanation of the code:

- This route decorator specifies that this function will handle requests to the URL path '/smr_backdate'.
- This function is responsible for generating SMR reports for a specified date range.
- Records the current time before starting the SMR report generation process.
- Calls the history_daterange_smr() method of the SMR class to generate SMR reports for the specified date range.
- Records the current time after completing the SMR report generation process.
- Returns a message indicating that the SMR report has been generated for the specified date range and provides the time taken for the process.

this code defines a route that triggers the generation of SMR reports for a specified date range. The route returns a message along with the time taken for the process to complete.

```
@app.route('/smr')
```

This route decorator specifies that this function will handle requests to the URL path '/smr'.

```
def get_smr_current():
```

This function is responsible for generating SMR reports for the current date.

```
report = 'SMR'
```

Defines the name of the report as 'SMR'.

```
start = time.time()
```

Records the current time before starting the SMR report generation process.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

Establishes a connection to the database using the DB_Helper class and gets a cursor to interact with the database.

```
# Try catch here
try:
    SMR().generate_smr_current(conn, cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "', '" + str(statuscode) + "', '" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "', '" + str(statuscode) + "', '" + str(report) + "')")
    conn.commit()
    raise e
```

- This block of code tries to generate the SMR report for the current date using the generate_smr_current() method of the SMR class.
- If successful, it records the end time, calculates the time taken for the process, and inserts an entry into the error log table with a success status.
- If an exception occurs during the process, it records the exception details, inserts an entry into the error log table with an error status, and raises the exception again.

```
conn.close()
```

Closes the database connection.

```
return str(statuscode)
```

Returns a message indicating the status of the report generation process.

this code defines a route that triggers the generation of SMR reports for the current date. It records the process status, time taken, and any exceptions that occur during the process.

```
@app.route('/smr_validate')
def smr_validate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
```

```
cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\"  
DESC LIMIT 1')  
details = cur.fetchone()  
string = str(details)  
status = check(string, 'SUCCESS')  
conn.close()  
return status
```

the status of the SMR (Stock Market Returns) report generation process. Here's an explanation of the code:

- This route decorator specifies that this function will handle requests to the URL path '/smr_validate'.
- This function is responsible for validating the status of the SMR report generation process.
- Establishes a connection to the database using the DB_Helper class and gets a cursor to interact with the database.
- Executes a SQL query to retrieve the most recent status from the error log table.
- Converts the fetched details to a string and checks if the status is 'SUCCESS' using the check() function (which isn't shown in the provided code). The check() function likely evaluates whether the status matches the expected value.
- Closes the database connection.
- Returns the validation status, which could be 'SUCCESS' or some other relevant status.

This code defines a route that retrieves the most recent status from the error log table related to the SMR report generation process and validates whether it's a success status or not. The result is then returned as the validation status.

```
@app.route('/ratios_list_backdate')  
def get_ratios_list():  
  
    start = time.time()  
    SMR().history_daterange_ratioslist()  
    end = time.time()  
    return "Compiled Ratios List for backdate. Time Taken: "+ str(end-start) + "  
seconds"
```

- This route decorator specifies that this function will handle requests to the URL path '/ratios_list_backdate'.
- This function is responsible for generating a historical list of financial ratios.
- Records the current time before starting the process to calculate the time taken for execution.

- Calls the `history_daterange_ratioslist()` method of the SMR class. This method presumably generates a historical list of financial ratios within a specified date range.
- Records the time after the process is completed.
- Returns a string indicating that the ratios list has been compiled for the backdate period and also includes the time taken for execution.

this code defines a route that triggers the generation of a historical list of financial ratios using the SMR class's method. It then calculates and returns the time taken for the execution of this process.

MFList and FRS Insert

```
@app.route('/mf_list_backdate')
def get_mf_list_backdate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    start = time.time()
    FRS().generate_history_mflist(conn, cur)
    end = time.time()
    conn.close()
    return "Compiled MutualFund list for backdate. Time Taken: "+ str(end-start) +
" seconds"
```

function for compiling a historical list of mutual funds. Here's a step-by-step explanation of the code:

- This is a route decorator that specifies the URL path `'/mf_list_backdate'`. When a user accesses this path in a web browser or makes an HTTP request, the associated function will be executed.
- This is the function that will be executed when a request is made to the `'/mf_list_backdate'` URL path. It compiles a historical list of mutual funds.
- These lines establish a connection to the database using the `db_connect()` method from the `DB_Helper` class and create a cursor object to execute SQL queries.
- This records the current time before starting the process to calculate the time taken for execution.
- This line calls the `generate_history_mflist()` method from the `FRS` class. Presumably, this method compiles the historical list of mutual funds using the provided database connection and cursor.
- This records the time after the process is completed.
- This closes the database connection and cursor to release resources.
- this line returns a string indicating that the mutual fund list has been compiled for the backdated period. It also includes the time taken for the execution of the process.

the function compiles a historical list of mutual funds by calling a method from the `FRS` class, and it measures the time taken for this process to complete. The result is returned as a string response.

```
@app.route('/frs_backdate')
def get_frs_backdate():

    conn = DB_Helper().db_connect()
```

```
cur = conn.cursor()
start = time.time()
FRS().generate_history_mfrank(conn, cur)
end = time.time()
conn.close()
return "Generated FRS report for backdate. Time Taken: "+ str(end-start) + "
seconds"
```

generating a historical FRS (Financial Ranking System) report for a backdated period. Here's a breakdown of the code:

- This is a route decorator that specifies the URL path '/frs_backdate'. When this path is accessed in a web browser or via an HTTP request, the associated function will be executed.
- This is the function that will be executed when a request is made to the '/frs_backdate' URL path. It generates a historical FRS report for a backdated period.
- These lines establish a connection to the database using the db_connect() method from the DB_Helper class and create a cursor object to execute SQL queries.
- This records the current time before starting the process, which will be used to calculate the time taken for execution.
- This line calls the generate_history_mfrank() method from the FRS class. Presumably, this method generates the historical FRS report using the provided database connection and cursor.
- This records the time after the process is completed.
- This line closes the database connection and cursor to release resources.
- Finally, this line returns a string indicating that the FRS report has been generated for the backdated period. It also includes the time taken for the execution of the process.

the function generates a historical FRS report by calling a method from the FRS class and calculates the time taken for this process to complete. The result is returned as a string response.

```
@app.route('/frs')
```

This is a route decorator that specifies the URL path '/frs'. When this path is accessed in a web browser or via an HTTP request, the associated function will be executed.

```
def get_frs_current():
```

This is the function that will be executed when a request is made to the '/frs' URL path. It generates the current FRS report.

```
report = 'FRS'
start = time.time()
```

These lines define a variable report to store the name of the report ('FRS'). They also record the current time before starting the process, which will be used to calculate the time taken for execution.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines establish a connection to the database using the db_connect() method from the DB_Helper class and create a cursor object to execute SQL queries.

```
try:
    FRS().generate_current_mfrank(conn,cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('" + str(timestamp) + "','" + str(statuscode) + "','" + str(report) + "')")
    conn.commit()
    raise e
```

This try block contains the main logic of generating the current FRS report. It does the following:

- Calls the generate_current_mfrank() method from the FRS class, presumably to generate the current FRS report using the provided database connection and cursor.
- Records the time after the process is completed.
- Constructs a success status message along with the time taken for execution.
- Gets the current timestamp and formats it.
- Inserts an entry into the 'ErrorLog' table with the timestamp, status code, and report name.
- Commits the changes to the database.
- If an exception occurs during the process, it captures the exception, records the timestamp, inserts an entry into the 'ErrorLog' table, commits the changes, and then raises the exception.

```
conn.close()
```

This line closes the database connection and cursor to release resources.

```
return str(statuscode)
```

Finally, this line returns the status code as a string response.

The function generates a current FRS report by calling a method from the FRS class and records the time taken for this process. It also logs the status and report details in the 'ErrorLog' table. The result (status code) is returned as a string response.

```
@app.route('/frs_validate')
def frs_validate():
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    cur.execute('SELECT "STATUS" FROM public."ErrorLog" ORDER BY "TIMESTAMP"
DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

A route and function for validating the FRS (Financial Ranking System) report generation process. Here's a breakdown of the code:

- This is a route decorator that specifies the URL path '/frs_validate'. When this path is accessed in a web browser or via an HTTP request, the associated function will be executed.
- This is the function that will be executed when a request is made to the '/frs_validate' URL path. It performs the validation of the FRS report generation process.
- These lines establish a connection to the database using the db_connect() method from the DB_Helper class and create a cursor object to execute SQL queries.
- These lines execute an SQL query to retrieve the latest status from the 'ErrorLog' table. It selects the 'STATUS' column value from the row with the most recent timestamp. The result is fetched using cur.fetchone() and stored in the details variable. The result is then converted to a string using the str() function.
- This line calls the check() function (presumably defined elsewhere in the code) to compare the retrieved status (in string format) with the expected status 'SUCCESS'. The result of this comparison is stored in the status variable.
- This line closes the database connection and cursor to release resources.
- this line returns the status as a response. This status indicates whether the most recent FRS report generation was successful or not based on the retrieved status from the 'ErrorLog' table.

the function retrieves the latest status of the FRS report generation process from the database and compares it to the expected 'SUCCESS' status. The result of this comparison is returned as the response.

```
@app.route('/nav_rank_backdate')
def get_nav_rank_history():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    start = time.time()
    FRS().generate_history_nav_rank(conn, cur)
```

```
end = time.time()
conn.close()
return "Compiled Scheme NAV list for backdate. Time Taken: "+ str(end-start) +
" seconds"
```

- This is a route decorator that sets up a URL path '/nav_rank_backdate'. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the '/nav_rank_backdate' URL path. It generates historical NAV ranking reports.
- These lines create a database connection and cursor using the db_connect() method from the DB_Helper class. The cursor will be used to execute SQL queries.
- This line records the current time before the historical NAV ranking report generation process starts.
- This line calls the generate_history_nav_rank() method from the FRS class to generate historical NAV ranking reports. The method is passed the database connection (conn) and cursor (cur) as parameters.
- This line records the current time after the historical NAV ranking report generation process completes.
- This line closes the database connection and cursor to release resources.
- This line returns a message that indicates the completion of the historical NAV ranking report generation process. It also includes the time taken for the process to complete, calculated by subtracting the start time from the end time.

this route and function generate historical NAV ranking reports by calling the relevant method from the FRS class. The time taken for the process is calculated, and a message indicating the completion of the process is returned as the response.

```
@app.route('/nav_rank')
```

This is a route decorator that sets up a URL path '/nav_rank'. When a request is made to this path, the associated function will be executed.

```
def get_nav_rank_current():
```

This function will be executed when a request is made to the '/nav_rank' URL path. It generates current NAV ranking reports.

```
report = 'NAV_RANK'
```

This line defines a string 'NAV_RANK' which represents the report name.

```
start = time.time()
```


This line records the current time before the current NAV ranking report generation process starts.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines create a database connection and cursor using the `db_connect()` method from the `DB_Helper` class. The cursor will be used to execute SQL queries.

```
try:
    FRS().generate_current_nav_rank(conn,cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" + str(report) + '\"')")
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\", \"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" + str(report) + '\"')")
    conn.commit()
    raise e
conn.close()
return str(statuscode)
```

This block of code attempts to generate the current NAV ranking report using the `generate_current_nav_rank()` method from the `FRS` class. If successful, it records the end time and calculates the time taken for the process. It also inserts a log entry into the database's `ErrorLog` table to record the status of the report generation.

If an exception occurs during the process, it captures the exception, records an error log entry, and raises the exception again.

This line closes the database connection and cursor to release resources.

This line returns the status code, indicating the success or failure of the current NAV ranking report generation process.

This route and function generate current NAV ranking reports by calling the relevant method from the `FRS` class. The status of the process is recorded in the database's `ErrorLog` table, and the status code is returned as the response.

```
@app.route('/nav_rank_validate')
def nav_rank_validate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()

    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\" DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

- This is a route decorator that sets up a URL path `'/nav_rank_validate'` . When a request is made to this path, the associated function will be executed.

- This function will be executed when a request is made to the '/nav_rank_validate' URL path. It validates the status of the current NAV ranking report generation process.
- These lines create a database connection and cursor using the db_connect() method from the DB_Helper class. The cursor will be used to execute SQL queries.
- This line executes an SQL query to retrieve the most recent status entry from the ErrorLog table. The query retrieves the 'STATUS' column value from the row with the latest timestamp.
- This line converts the retrieved status details into a string and then checks if the status string contains the word 'SUCCESS'. It uses a check() function, which is likely defined elsewhere in your code, to perform this check. The result of the check is stored in the status variable.
- This line closes the database connection and cursor to release resources.
- This line returns the result of the status check as the response. The response will indicate whether the most recent current NAV ranking report generation process was successful (contains 'SUCCESS') or not.

This route and function validate the status of the current NAV ranking report generation process by querying the ErrorLog table for the most recent status entry and checking if it contains the word 'SUCCESS'. The validation result is returned as the response.

The code defines a route and function for generating and validating Mutual Fund Open-High-Low-Close (OHLC) data. Here's an overview of the code:

```
@app.route('/mf_ohlc')
```

This is a route decorator that sets up a URL path '/mf_ohlc'. When a request is made to this path, the associated function will be executed.

```
def get_mf_ohlc():
```

This function will be executed when a request is made to the '/mf_ohlc' URL path. It generates the Mutual Fund Open-High-Low-Close (OHLC) data and logs the status of the process in the ErrorLog table.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines create a database connection and cursor using the db_connect() method from the DB_Helper class. The cursor will be used to execute SQL queries.

```
try:
    mf_ohlc = MFOHLC()
    mf_ohlc.gen_mf_ohlc_current(conn, cur)
```

This section of code within the try block creates an instance of the MFOHLC class, which presumably contains methods for generating Mutual Fund OHLC data. It then calls the gen_mf_ohlc_current() method to generate the OHLC data and store it in the database.

```
end = time.time()
```

```
        statuscode = "SUCCESS in " + str(end-start) + " seconds"
        timestamp = datetime.datetime.now()
        timestamp = timestamp.strftime("%c")
        cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
        conn.commit()
```

After generating the OHLC data, this section logs the status of the process into the ErrorLog table. It records the timestamp, status code (indicating success), and the report name ('MF_OHLC').

```
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
    conn.commit()
    raise e
```

In case an exception occurs during the OHLC generation process, this block catches the exception, logs the error status with a timestamp and report name in the ErrorLog table, commits the changes, and then re-raises the exception.

```
conn.close()
return str(statuscode)
```

These lines close the database connection and cursor to release resources. Finally, the function returns the status code as a response, indicating whether the OHLC generation process was successful.

this route and function generate Mutual Fund Open-High-Low-Close (OHLC) data using methods from the MFOHLC class, log the status of the process in the ErrorLog table, and return the status code as a response. If the OHLC generation encounters an exception, the exception is caught, logged, and re-raised.

```
@app.route('/mf_ohlc_validate')
def mf_ohlc_validate():
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\"
DESC LIMIT 1')
    details = cur.fetchone()
```

```

string = str(details)
status = check(string, 'SUCCESS')
conn.close()
return status

```

The provided code defines a route and function for validating the Mutual Fund Open-High-Low-Close (OHLC) data generation process. Let's break down the code step by step:

- This is a route decorator that sets up a URL path '/mf_ohlc_validate'. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the '/mf_ohlc_validate' URL path. It validates the status of the Mutual Fund OHLC data generation process by checking the latest entry in the ErrorLog table.
- These lines create a database connection and cursor using the db_connect() method from the DB_Helper class. The cursor will be used to execute SQL queries.
- This code section queries the ErrorLog table to retrieve the latest status entry. It fetches the value of the 'STATUS' column for the latest record and converts it to a string. The check() function is then used to validate whether the status is 'SUCCESS'. It seems like the check() function is not defined in the provided code, but based on previous examples, it likely checks whether the provided status string matches the expected status ('SUCCESS').
- This line closes the database connection to release resources.
- Finally, the status, which is the result of the validation, is returned as the response to the requester. This response could be a simple string like "SUCCESS" or "FAILURE" indicating the validation result.

this route and function validate the status of the Mutual Fund OHLC data generation process by checking the latest entry in the ErrorLog table. It uses the retrieved status to determine whether the process was successful and then returns the validation status as the response.

```

@app.route('/mf_ohlc_backdate')
def get_mf_ohlc_backdate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    start = time.time()
    MFOHLC().gen_mf_ohlc_history(conn, cur)
    end = time.time()
    conn.close()
    return "Compiled MF OHLC list for backdate. Time Taken: " + str(end-start) + "
seconds"

```

The provided code defines a route and function for generating Mutual Fund Open-High-Low-Close (OHLC) historical data. Let's break down the code step by step:

- This is a route decorator that sets up a URL path '/mf_ohlc_backdate'. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the '/mf_ohlc_backdate' URL path. It generates historical Mutual Fund OHLC data.
- These lines create a database connection and cursor using the db_connect() method from the DB_Helper class. The cursor will be used to execute SQL queries.
- This section of code measures the time it takes to generate the historical Mutual Fund OHLC data. It records the starting time using time.time(), then calls the gen_mf_ohlc_history() method of the MFOHLC class to generate the historical data using the provided database connection and cursor. Afterward, it records the ending time.
- This line closes the database connection to release resources.
- this line returns a string as the response to the requester. The string indicates that the historical Mutual Fund OHLC list has been compiled for backdating and includes the time taken to complete the process.

This route and function generate historical Mutual Fund OHLC data for backdating. It measures the time taken to complete the generation process and returns a response string that includes information about the process and the time it took.

IRS

```
@app.route('/irs_history')
def irs_history():

    start = time.time()
    IRS().gen_irs_history()
    end = time.time()
    return "Generated PE High/Low for history IRS and inserted. Time Taken: "+
str(end-start) + " seconds"
```

The provided code defines a route and function for generating historical PE (Price-to-Earnings) High/Low data for IRS (Interest Rate Swaps). Let's break down the code step by step:

- This is a route decorator that sets up a URL path '/irs_history'. When a request is made to this path, the associated function will be executed.

- This function will be executed when a request is made to the '/irs_history' URL path. It generates historical PE High/Low data for IRS.
- This section of code measures the time it takes to generate the historical PE High/Low data for IRS. It records the starting time using `time.time()`, then calls the `gen_irs_history()` method of the IRS class to generate the historical data. Afterward, it records the ending time.
- Finally, this line returns a string as the response to the requester. The string indicates that the historical PE High/Low data for IRS has been generated and inserted, and it includes the time taken to complete the process.

this route and function generate historical PE High/Low data for IRS. It measures the time taken to complete the generation process and returns a response string that includes information about the process and the time it took.

The provided code defines a route and function for generating historical PE (Price-to-Earnings) data for IRS (Interest Rate Swaps) based on specific dates. Let's break down the code step by step:

```
@app.route('/irs_date_history')
```

This is a route decorator that sets up a URL path '/irs_date_history'. When a request is made to this path, the associated function will be executed.

```
def irs_date_history():
```

This function will be executed when a request is made to the '/irs_date_history' URL path. It generates historical PE data for IRS based on specific dates.

```
    report = 'IRS_Date_History'  
    start = time.time()
```

These lines define a variable `report` to hold the name of the report ('IRS_Date_History') and record the starting time using `time.time()`.

```
    conn = DB_Helper().db_connect()  
    cur = conn.cursor()
```

These lines establish a database connection using the `db_connect()` method of the `DB_Helper` class and create a cursor for executing SQL queries.

```
    try:  
        IRS().gen_irs_date_history(conn, cur)  
        # IRS().gen_irs_date_history_after_dec(conn, cur)  
        end = time.time()  
        statuscode = "SUCCESS in " + str(end-start) + " seconds"  
        timestamp = datetime.datetime.now()  
        timestamp = timestamp.strftime("%c")  
        cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",  
        \"REPORT\") VALUES ('\" + str(timestamp) + \"', '\" + str(statuscode) + \"', '\" +  
        str(report) + \"')")  
        conn.commit()  
    except Exception as e:
```

```

        statuscode = e
        timestamp = datetime.datetime.now()
        timestamp = timestamp.strftime("%c")
        cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
        conn.commit()

```

This section of code tries to generate historical PE data for IRS based on specific dates using the `gen_irs_date_history()` method of the IRS class. If the process is successful, it records the ending time, generates a status code string, and inserts an entry into the error log table with the timestamp, status code, and report name. If an exception occurs during the process, it captures the exception, records the timestamp, and inserts an entry into the error log table.

```
conn.close()
```

This line closes the database connection.

```
return str(statuscode)
```

Finally, this line returns the status code string as the response to the requester.

this route and function generate historical PE data for IRS based on specific dates. It measures the time taken to complete the generation process and records the result in the error log table. The status code is then returned as the response.

```

@app.route('/irs_daterange')
def irs_daterange():
    start = time.time()
    IRS().gen_irs_daterange()
    end = time.time()
    return "Generated IRS for daterange. Time Taken: " + str(end-start) + " seconds"

```

The provided code defines a route and function for generating IRS (Interest Rate Swaps) data for a specific date range. Let's break down the code step by step:

- This is a route decorator that sets up a URL path `'/irs_daterange'` . When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the `'/irs_daterange'` URL path. It generates IRS data for a specific date range.
- This line records the starting time using `time.time()` .
- This line calls the `gen_irs_daterange()` method of the IRS class to generate IRS data for the specific date range.
- This line records the ending time using `time.time()` .
- this line returns a string indicating that IRS data has been generated for the specified date range, along with the time taken to complete the process.

This route and function generate IRS data for a specific date range, record the time taken to complete the process, and return a message indicating the completion and time taken.

This code defines a route and function for generating current IRS (Interest Rate Swaps) data for a specific date. It supports both GET and POST methods. Let's break down the code step by step:

```
@app.route('/irs', methods=['GET', 'POST'])
```

This is a route decorator that sets up a URL path '/irs' and specifies that this route can handle both GET and POST requests. When a request is made to this path, the associated function will be executed.

```
def irs_current():
```

This function will be executed when a request is made to the '/irs' URL path.

```
curr_date = request.form['date']
```

This line retrieves the 'date' parameter from the form data of the request. It expects that the parameter is passed in the POST request.

```
curr_date = curr_date.split('-')
print("\nDATA: {}".format(curr_date))
curr_date = datetime.date(int(curr_date[0]), int(curr_date[1]),
int(curr_date[2]))
print("Running for {}".format(str(curr_date)))
```

These lines parse the retrieved 'date' parameter and convert it into a datetime.date object.

```
report = 'IRS'
start = time.time()
```

These lines define a variable 'report' as 'IRS' and record the starting time using time.time().

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines establish a database connection using the db_connect() method of the DB_Helper class and create a database cursor.

```
try:
    IRS().gen_irs_current(curr_date, conn, cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\"')
    conn.commit()
except Exception as e:
    statuscode = e
```



```

        timestamp = datetime.datetime.now()
        timestamp = timestamp.strftime("%c")
        cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
        conn.commit()
    pass

```

This block of code tries to generate current IRS data using the `gen_irs_current()` method of the IRS class. If successful, it records the end time and a success status. If an exception occurs, it records the exception and an error status. It also logs this information into the error log table.

```

    conn.close()
    return str(statuscode)

```

- This line closes the database connection.
- this line returns a string indicating the status of the operation, whether it was successful or if an error occurred.

This route and function generate current IRS data for a specific date, log the process and its status, and return a message indicating the outcome. It handles both GET and POST requests, expecting the 'date' parameter in the form data of a POST request.

```

@app.route('/irs_validate')
def irs_validate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\"
DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status

```

This code defines a route for validating the status of an IRS (Interest Rate Swaps) data generation process. Let's break down the code step by step:

- This is a route decorator that sets up a URL path '/irs_validate'. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the '/irs_validate' URL path.
- These lines establish a database connection using the `db_connect()` method of the `DB_Helper` class and create a database cursor.
- These lines execute an SQL query to retrieve the latest status entry from the error log table.
- These lines convert the fetched status details to a string and then pass it to the `check()` function (which seems to be a user-defined function) along with the expected status 'SUCCESS'. The purpose of this function seems to be checking whether the fetched status matches the expected status.

- This line closes the database connection.
- This line returns the status result, which indicates whether the latest status in the error log matches the expected 'SUCCESS' status.

This route and function retrieve the latest status from the error log table, compare it with the expected 'SUCCESS' status, and return the validation result. It provides a way to check whether the IRS data generation process was successful or not.

Combined Rank

This code defines a route for generating and inserting combined rank data in the database. Let's break down the code step by step:

```
@app.route('/combirs_current')
```

This is a route decorator that sets up a URL path '/combirs_current'. When a request is made to this path, the associated function will be executed.

```
def combined_rank_current():
```

This function will be executed when a request is made to the '/combirs_current' URL path.

```
report = 'CombinedRS'
```

The variable report is assigned the value 'CombinedRS'. This is used as a label to identify this specific operation in the error log.

```
start = time.time()
```

This records the current time as the start time of the data generation process.

```
conn = DB_Helper().db_connect()
cur = conn.cursor()
```

These lines establish a database connection using the db_connect() method of the DB_Helper class and create a database cursor.

```
try:
    CombinedRank().current_rank(conn,cur)
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
    conn.commit()
```

```
raise e
```

- This block of code encapsulates the main process of generating and inserting combined rank data.
- It calls the `current_rank()` method of the `CombinedRank` class, which seems to handle the actual generation and insertion process.
- It records the end time after the process is completed and calculates the time taken.
- It constructs the status message indicating success along with the timestamp and inserts this information into the error log table in the database.
- If an exception occurs during the process, it captures the exception, constructs an error status message with a timestamp, and inserts this information into the error log table. Then, it re-raises the exception.

```
conn.close()
return str(statuscode)
```

- This line closes the database connection.
- This line returns the status code indicating the success or failure of the combined rank data generation process.

This route and function are responsible for generating and inserting combined rank data into the database. It logs the status of the operation in the error log table and returns the status code indicating the success or failure of the operation.

```
@app.route('/combirs_validate')
def combirs_validate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\"
DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

This code defines a route for validating the status of the combined rank data generation process. Let's break down the code step by step:

- This is a route decorator that sets up a URL path `/combirs_validate`. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the `/combirs_validate` URL path.
- These lines establish a database connection using the `db_connect()` method of the `DB_Helper` class and create a database cursor.
- These lines execute a SQL query to retrieve the most recent status from the error log table.

- The fetchone() method retrieves one row of the result.
- The string variable is assigned the string representation of the retrieved status details.
- This line calls a function check() (which isn't defined in the provided code) to compare the retrieved status with the string 'SUCCESS'.
- If the status matches 'SUCCESS', it's assumed that the combined rank data generation process was successful.
- This line closes the database connection.
- this line returns the result of the status check, indicating whether the combined rank data generation process was successful or not.

This route and function are responsible for validating the status of the combined rank data generation process by querying the error log table, checking if the status is 'SUCCESS', and returning the result of the validation.

```
@app.route('/combirs_history')
def combirs_backdate():

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    start = time.time()
    CombinedRank().history_rank(conn, cur)
    end = time.time()
    conn.close()
    return "Generated Combined Rank for backdate. Time Taken: " + str(end-start) + "
seconds"
```

This code defines a route that triggers the generation of combined rank data for a historical period. Let's break down the code step by step:

- This is a route decorator that sets up a URL path '/combirs_history'. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the '/combirs_history' URL path.
- These lines establish a database connection using the db_connect() method of the DB_Helper class and create a database cursor.
- These lines record the starting time using time.time() before calling the history_rank() method of the CombinedRank class to generate historical combined rank data.
- After the data generation is complete, it records the ending time using time.time().
- This line closes the database connection.
- this line returns a message indicating that the combined rank data for a historical period has been generated, along with the time taken for the process.

This route and function are responsible for triggering the generation of combined rank data for a historical period. It records the time taken for the process and returns a message indicating its completion.

Dash daily process

```
@app.route('/stock_performance')
```

This is a route decorator that sets up a URL path '/stock_performance'. When a request is made to this path, the associated function will be executed.

```
def stock_performance():
```

This function will be executed when a request is made to the '/stock_performance' URL path.

```
    report = 'StockPerformance'
```

This variable holds the name of the report, which is 'StockPerformance' in this case.

```
    start = time.time()
```

This line records the starting time using time.time().

```
    conn = DB_Helper().db_connect()
    cur = conn.cursor()
```

These lines establish a database connection using the db_connect() method of the DB_Helper class and create a database cursor.

```
try:
    perstock_change.main()
    end = time.time()
    statuscode = "SUCCESS in " + str(end-start) + " seconds"
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
    conn.commit()
except Exception as e:
    statuscode = e
    timestamp = datetime.datetime.now()
    timestamp = timestamp.strftime("%c")
    cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +
str(report) + '\")')
    conn.commit()
    raise e
```

This block of code contains a try-except block:

- Inside the try block, the perstock_change.main() function is called to perform the daily process of calculating stock performance.
- If the process is successful, it records the ending time, status code, and timestamp of the completion in the error log table in the database.

- Inside the except block, if an exception occurs during the process, it records the exception details in the error log table, commits the changes to the database, and raises the exception again to ensure proper logging and handling.

```
conn.close()
```

This line closes the database connection.

```
return str(statuscode)
```

this line returns the status code message indicating the success of the daily stock performance calculation process.

This endpoint and function are responsible for triggering the daily process of calculating stock performance. It records the time taken for the process, updates the error log in the database, and returns a status code message.

```
@app.route('/stock_performance_validate')
def stock_performance_validate():
    """ Endpoint to validate if the process is successful.
    """

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\"
DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

- This is a route decorator that sets up a URL path '/stock_performance_validate'. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the '/stock_performance_validate' URL path.
- These lines establish a database connection using the db_connect() method of the DB_Helper class and create a database cursor.
- This code queries the error log table in the database to retrieve the most recent status entry. It selects the 'STATUS' column from the 'ErrorLog' table and orders the results by timestamp in descending order. It then fetches the first result using fetchone() and converts it to a string.
- This line uses the check() function (assuming it's defined elsewhere in your code) to compare the retrieved status with the expected status 'SUCCESS'. If the status matches, it returns 'SUCCESS'; otherwise, it returns some other status.

- This line closes the database connection.
- this line returns the status (which is either 'SUCCESS' or some other status) as the response of the endpoint.

this endpoint and function validate whether the daily stock performance process has been successful by querying the error log and comparing the status with 'SUCCESS'. It then returns the validation status as the response.

```
@app.route('/stock_offhigh_low')
```

This is a route decorator that sets up a URL path '/stock_offhigh_low'. When a request is made to this path, the associated function will be executed.

```
def stock_offhigh_low():
```

This function will be executed when a request is made to the '/stock_offhigh_low' URL path.

```
    report = 'Stock Off-High/Low'  
    start = time.time()
```

These lines define the report name and start a timer to measure the execution time of the process.

```
    conn = DB_Helper().db_connect()  
    cur = conn.cursor()
```

These lines establish a database connection using the db_connect() method of the DB_Helper class and create a database cursor.

```
    try:  
        perstock_offhighlow.main()  
        end = time.time()  
        statuscode = "SUCCESS in " + str(end-start) + " seconds"  
        timestamp = datetime.datetime.now()  
        timestamp = timestamp.strftime("%c")  
        cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",  
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +  
str(report) + '\"')  
        conn.commit()  
    except Exception as e:  
        statuscode = e  
        timestamp = datetime.datetime.now()  
        timestamp = timestamp.strftime("%c")  
        cur.execute("INSERT INTO public.\"ErrorLog\" (\"TIMESTAMP\", \"STATUS\",  
\"REPORT\") VALUES ('\" + str(timestamp) + '\", '\" + str(statuscode) + '\", '\" +  
str(report) + '\"')  
        conn.commit()  
        raise e
```

This code block runs the process to calculate the off-high/low of individual stocks using the `perstock_offhighlow.main()` function. If the process is successful, it records a 'SUCCESS' status along with the execution time and current timestamp in the error log table. If an exception occurs, it records the exception as the status in the error log and raises the exception again.

```
conn.close()
return str(statuscode)
```

- This line closes the database connection.
- This line returns the status of the process (either 'SUCCESS' or an error message) as the response of the endpoint.
- This endpoint and function run a daily process for calculating the off-high/low of individual stocks. It records the status, execution time, and timestamp in the error log table and returns the status as the response.

```
@app.route('/stock_offhigh_low_validate')
def stock_offhigh_low_validate():
    """ Endpoint to validate if process is successful.
    """

    conn = DB_Helper().db_connect()
    cur = conn.cursor()
    cur.execute('SELECT \"STATUS\" FROM public.\"ErrorLog\" ORDER BY \"TIMESTAMP\"
DESC LIMIT 1')
    details = cur.fetchone()
    string = str(details)
    status = check(string, 'SUCCESS')
    conn.close()
    return status
```

- This is a route decorator that sets up a URL path '/stock_offhigh_low_validate'. When a request is made to this path, the associated function will be executed.
- This function will be executed when a request is made to the '/stock_offhigh_low_validate' URL path.
- These lines establish a database connection using the `db_connect()` method of the `DB_Helper` class and create a database cursor.
- These lines retrieve the most recent status entry from the error log table and convert it to a string. Then, the `check()` function (which seems to be defined elsewhere) is used to determine if the status is 'SUCCESS' or not.
- This line closes the database connection.

- This line returns the validation status (either 'SUCCESS' or an error message) as the response of the endpoint.
- This endpoint and function validate whether the process of calculating stock off-high/low has been successful by checking the most recent status in the error log table. It returns the validation status as the response.