Projektbeskrivning

I detta projekt har jag utvecklat ett automatiserat system som hämtar väderdata från ett API, bearbetar denna data och lagrar den i en SQL-databas. Systemet är uppdelat i flera delar: ett huvudskript (**Weather.py**) som ansvarar för datahämtning och -lagring, ett GUI-baserat skript (**WeatherViewerApp.py**) för att visa och interagera med datan, och ett testskript (**test_weather.py**) för att verifiera att funktionaliteten fungerar korrekt. Systemet är automatiserat och körs enligt ett schemalagt schema via Windows Task Scheduler.

Genomgång av de olika filerna

1. Weather.py

Översikt

Denna file som läsa värden från Open-Meteo API, Spara data i SQL-Database.Det har tre huvudfunktioner: att hämta data, bearbeta data och infoga den i databasen.

Kodstruktur och Förklaring

Importer och Loggning

```
import pyodbc
import requests
import pandas as pd
from sqlalchemy import create_engine
import logging
```

- **requests**: Används för att hämta data från API
  .
- **pandas**: Används för att manipulera och bearbeta data i tabellformat.
- **sqlalchemy**: Används för att ansluta till SQL-databasen och infoga data.
- **logging**: Används för att logga händelser och fel under körning.

```
# Set up logging
logging.basicConfig(filename='historical_weather_task.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')
```

- Konfigurerar loggning så att alla loggar sparas i filen

Funktion: fetch_and_process_weather_data

```python
def fetch_and_process_weather_data():
    """Fetches and processes weather data from the API."""
    # Parameters for the API request
    latitude = 59.33
    longitude = 18.07
    start_date = "2021-01-01"
    end_date = "2021-12-31"
    url = f"https://archive-api.open-meteo.com/v1/era5?latitude={latitude}&longitude={longitude}&start
```

- **Syfte**: Denna funktion ansvarar för att hämta och bearbeta väderdata.
- **API-anrop**: Använder parametrar för att hämta väderdata för Stockholm, Sverige under 2021.

```python
# Make the API request
response = requests.get(url)
response.raise_for_status()
data = response.json()

# Convert data to DataFrame
hourly_data = data['hourly']
df = pd.DataFrame(hourly_data)
```

Hämtar data från API
konverterar svaret till JSON och skapar en DataFrame från timvis data.

```python
df['time'] = pd.to_datetime(df['time'])

if df.isnull().values.any():
    logging.warning("Data contains missing values. Filling with default values.")
    df.fillna({'temperature_2m': 0, 'relative_humidity_2m': 50, 'wind_speed_10m': 0}, inplace=
```

Omvandlar tidsformatet och fyller i saknade värden med standardvärden.

```
df['temperature_2m'] = df['temperature_2m'] * 9/5 + 32

logging.info(f"Transformed DataFrame Columns and Types:\n{df.dtypes}")
logging.info(f"Sample Transformed Data:\n{df.head()}")

return df
```

- Omvandlar temperaturen från Celsius till Fahrenheit och loggar datatyperna samt exempeldata.

Funktion:

```
def insert_data_to_db(df):
    """Inserts the DataFrame into the database."""
    try:
        engine = create_engine("mssql+pyodbc://@GEORGE/WeatherINSweden?trusted_connection=yes&driver=O
        df.to_sql('weather_forecast', con=engine, if_exists='append', index=False)
        logging.info("Historical weather data inserted successfully.")
        print("Data inserted successfully.")
    except Exception as e:
        logging.error(f"Error inserting data: {e}")
        raise e
```

Koppla till data basen som heter weather_forecast och det printa ut meddelande att den är kopplat

Exeption om det något fel så ska program printa ut Error

Huvudfunktion (__main__)

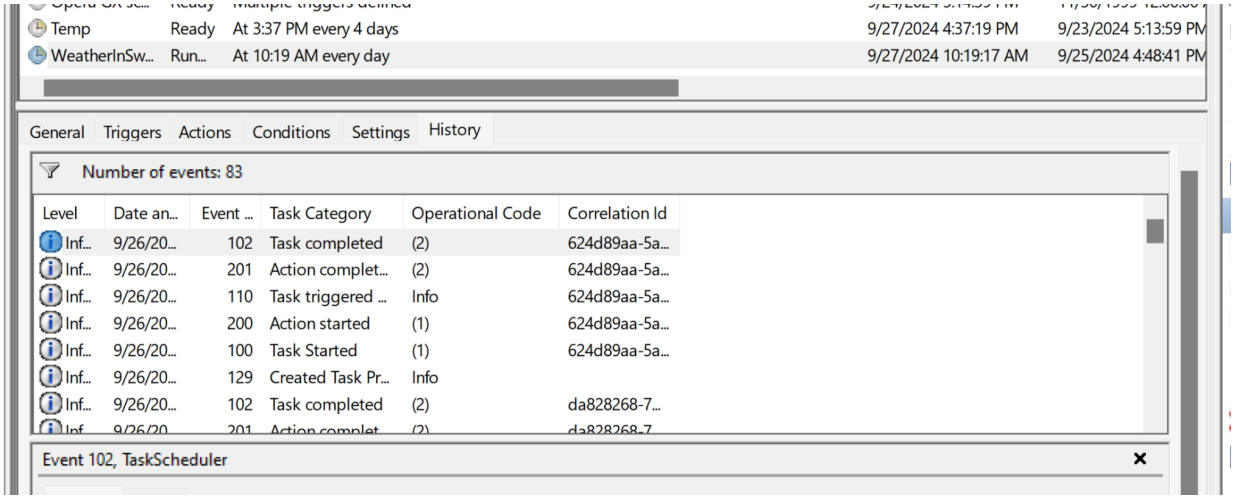
```
if __name__ == '__main__':
    df = fetch_and_process_weather_data()
    insert_data_to_db(df)
```

Kör datainsamlings- och infogningsexekvering när skriptet körs direkt.

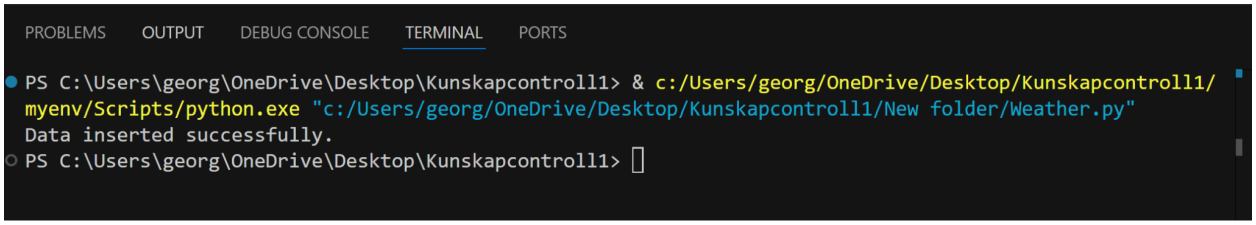
Schemaläggning av Skriptet

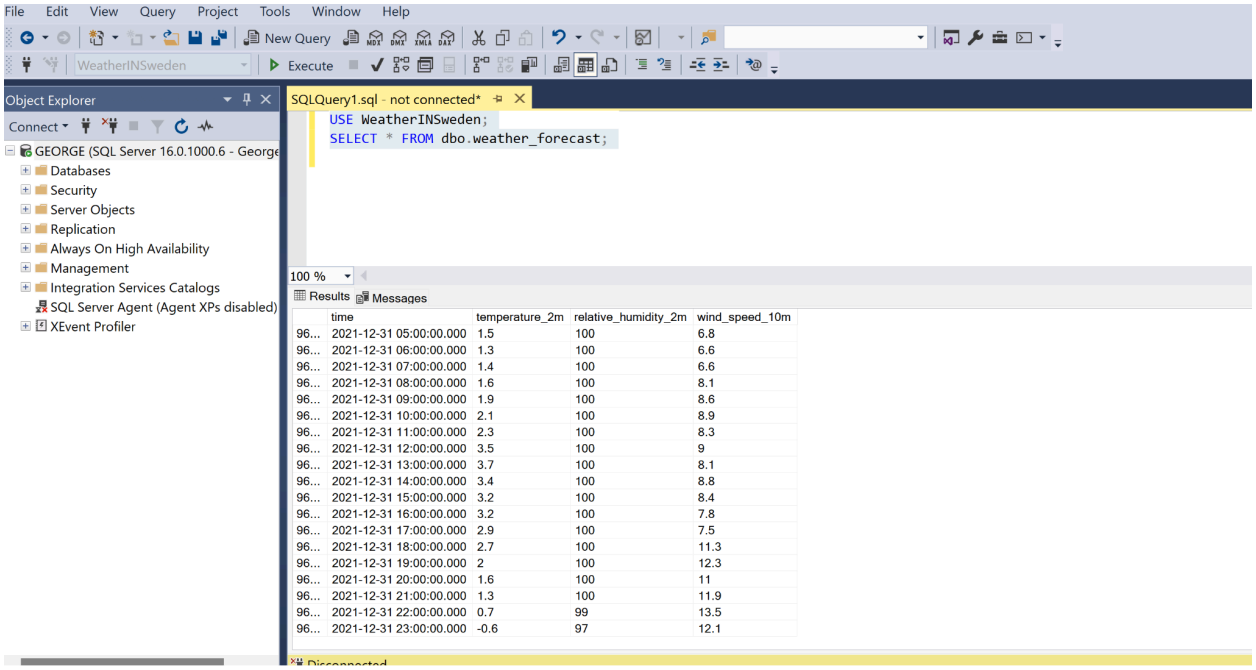
För att köra skriptet automatiskt vid en specifik tidpunkt, använd Windows Task Scheduler:

1. Öppna Task Scheduler och skapa en ny grundläggande uppgift.
2. Välj när du vill att skriptet ska köras.
3. Välj "Starta ett program" och ange sökvägen till Python-exekverbara filen och skriptet Weather.py.
4. Spara och aktivera uppgiften.



Data skicat till databasen





2. test_weather.py

Detta skript använder unittest för att säkerställa att huvudskriptet (Weather.py) fungerar korrekt. Det testar datahämtning, databasanslutning och datainförsel.

Kodstruktur och Förklaring:

Testklass: TestWeatherData

```python
class TestWeatherData(unittest.TestCase):

    def test_data_fetching(self):
        df = fetch_and_process_weather_data()
        self.assertIsInstance(df, pd.DataFrame, "Data fetched is not a DataFrame.")
        self.assertGreater(len(df), 0, "DataFrame is empty.")

    def test_database_connection(self):
        try:
            engine = create_engine("mssql+pyodbc://@GEORGE/WeatherINSweden?trusted_connection=yes&driv
            conn = engine.connect()
            conn.close()
        except Exception as e:
            self.fail(f"Database connection failed: {e}")

    def test_data_insertion(self):
        df = fetch_and_process_weather_data()
        try:
            insert_data_to_db(df)
        except Exception as e:
            self.fail(f"Data insertion failed: {e}")
```

- **test_data_fetching**: Testar att fetch_and_process_weather_data() returnerar en giltig DataFrame.
- **test_database_connection**: Testar anslutningen till databasen.
- **test_data_insertion**: Testar infogningen av data i databasen.

Huvudblock för att köra tester

```python
if __name__ == '__main__':
    unittest.main()
```

- Kör alla definierade tester när skriptet exekveras.

Test fungerar jätte bra.

```
yenv/Scripts/python.exe "c:/Users/georg/OneDrive/Desktop/Kunskapcontroll1/New folder/test_weather.py"
.Data inserted successfully.
..
----------------------------------------------------------------------
Ran 3 tests in 1.722s

OK
```

3- WeatherViewerApp.py

```
# Skapa anslutning till SQL-databasen
DATABASE_URL = "mssql+pyodbc://@GEORGE/WeatherINSweden?trusted_connection=yes&driver=ODBC+Driver+17+fo
engine = create_engine(DATABASE_URL)

# Läs in data från SQL-databasen
query = "SELECT * FROM weather_forecast"
df = pd.read_sql(query, con=engine)
```

Denna koppla till data base och börja läsa från min table

```
class WeatherViewerApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Weather Data Viewer")
        self.filtered_df = df.copy()

        # Skapa GUI-komponenter
        self.create_gui()

        # Visa all data initialt
        self.display_results(self.filtered_df)
```

Class som integrear programmet med gui

```python
def create_gui(self):

    style = ttk.Style()
    style.configure("Treeview", rowheight=30, font=('Arial', 10))
    style.configure("Treeview.Heading", font=('Arial', 12, 'bold'))

    main_frame = tk.Frame(self.root)
    main_frame.pack(fill="both", expand=True)

    filter_frame = tk.Frame(main_frame)
    filter_frame.pack(fill="x", pady=10)

    self.filter_label = tk.Label(filter_frame, text="Filter (Column:Value):")
    self.filter_label.pack(side="left", padx=5)

    self.filter_entry = tk.Entry(filter_frame)
    self.filter_entry.pack(side="left", padx=5)

    self.filter_button = tk.Button(filter_frame, text="Apply Filter", command=self.apply_filter)
    self.filter_button.pack(side="left", padx=5)

    self.tree = ttk.Treeview(main_frame, columns=[col for col in df.columns], show='headings')
    self.tree.pack(fill="both", expand=True)
    for col in df.columns:
        self.tree.heading(col, text=col)
        self.tree.column(col, width=100, stretch=tk.NO)  # Initial width, stretch turned off

    self.auto_adjust_column_width()
```

Skapar Gui komponenter

```python
def auto_adjust_column_width(self):
    # Skapa en label för textmätning
    label = tk.Label(self.root, font=('Arial', 10))
    max_width = {col: len(self.tree.heading(col, 'text')) * 10 for col in self.tree["columns"]}

    # Kontrollera alla kolumnvärden
    for item in self.tree.get_children():
        row_values = self.tree.item(item, 'values')
        for col, value in zip(self.tree["columns"], row_values):
            label.config(text=value)
            text_width = label.winfo_reqwidth()
            max_width[col] = max(max_width[col], text_width)

    # Ställ in kolumnbredd baserat på det bredaste innehållet
    for col, width in max_width.items():
        self.tree.column(col, width=width + 20)  # Lägg till extra padding

def display_results(self, df_to_display):
```

Automatisk Justering av Kolumnbredd
Auto_adjust_column_width(): Används för att automatiskt justera bredden på kolumnerna i Treeview-widgeten baserat på det bredaste innehållet i varje kolumn.

```
def display_results(self, df_to_display):
    # Rensa tidigare resultat
    for item in self.tree.get_children():
        self.tree.delete(item)

    # Visa resultat
    for _, row in df_to_display.iterrows():
        self.tree.insert("", "end", values=row.tolist())
```

Visa Data i Treeview-widgeten som Tar en DataFrame som argument och visar innehållet.

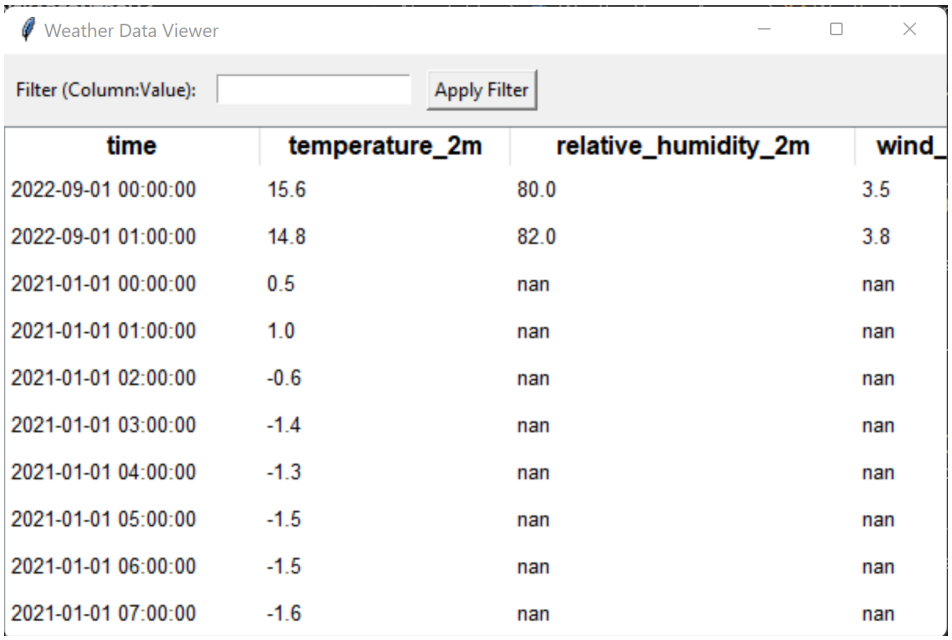
```
def apply_filter(self):
    filter_text = self.filter_entry.get()
    if not filter_text:
        self.filtered_df = df.copy()
    else:
        try:
            column, value = filter_text.split(':', 1)
            column = column.strip()
            value = value.strip()
            if column in df.columns:
                self.filtered_df = df[df[column].astype(str).str.contains(value, case=False, na=Fa
            else:
                messagebox.showwarning("Invalid Column", f"Column '{column}' does not exist.")
                self.filtered_df = df.copy()
        except ValueError:
            messagebox.showwarning("Invalid Filter", "Filter format should be 'Column:Value'.")
            self.filtered_df = df.copy()

    # Uppdatera visningen med filtrerad data
    self.display_results(self.filtered_df)
```

Används för att filtrera data baserat på ett specifikt filter (kolumn och värde).

```
# Skapa applikationsfönstret
root = tk.Tk()
app = WeatherViewerApp(root)
root.mainloop()
```

Starta Application



time	temperature_2m	relative_humidity_2m	wind_
2022-09-01 00:00:00	15.6	80.0	3.5
2022-09-01 01:00:00	14.8	82.0	3.8
2021-01-01 00:00:00	0.5	nan	nan
2021-01-01 01:00:00	1.0	nan	nan
2021-01-01 02:00:00	-0.6	nan	nan
2021-01-01 03:00:00	-1.4	nan	nan
2021-01-01 04:00:00	-1.3	nan	nan
2021-01-01 05:00:00	-1.5	nan	nan
2021-01-01 06:00:00	-1.5	nan	nan
2021-01-01 07:00:00	-1.6	nan	nan

Filstruktur:

- Weather.py: Huvudskriptet för att hämta och infoga väderdata i SQL-databasen.
- WeatherViewerApp.py: GUI-applikationen för att visa och interagera med väderdata.
- test_weather.py: Testskriptet som verifierar huvudskriptets funktionalitet.
- README.md: Dokumentation och instruktioner för projektet.
- requirements.txt: Lista över beroenden för att köra projektet.

Självutvärdering:
Utmaningar: Som IOT utvecklare att jobba med API är inte svårt för mig och har redan jobbat med fler project förut och spara de i databasen.

Gui, jag har jobbat med Gui i programmering nätverk förut men Java språket och har bygget local chat program.

Test_weather.py: var ny för mig och jag lärt mig hur man fixa det

Betygsbedömning: jag har gjort allt som behövs och jag tror att jag ska få VG.