POLITECNICO
MILANO 1863

iAPP4P

Design Document

Lorenzo Carlassara
Angelica Iseni
Emma Lodetti
Virginia Valeri

Software Engineering for Geoinformatics

Academic year 2022/23

Title:	Design Document
Authors:	Carlassara L. (10601118), Iseni A. (10668862), Lodetti E. (10619244), Valeri V. (10639607)
Date:	June 18, 2023
Professor:	Giovanni E. Quattrocchi

Contents

1 Introduction

1.1 Purpose

Before diving into coding and implementing a software project, it is crucial to have a clear understanding of the design goals that iAPP4P seeks to achieve. This is where the "Design Document" comes in.
The purpose of this document is to describe the requirements for the development of an interactive client-server application that enables users to access, query and visualize air quality data retrieved from existing public digital archives. The system consists of three main components: a database to ingest and store the data, psycopg2 and SQLAlchemy packages for querying the databases and retrieve the data through a python script, and a jupyter dashboard to provide means for requesting, processing (descriptive statistics), and visualizing data (maps, dynamic graphs).
The Design Document is intricately connected to the Requirements Analysis Specification Document (RASD), authored Carlassara L., Iseni A., Lodetti E., Valeri V.
You can access the RASD on GitHub at the following link:
https://github.com/carls31/SE4GEO-Lab/blob/main/GRUPPO_GEO_RASD.pdf

1.2 Context and motivations

The European Commission's agreement on electronic fuels will enable the sale of heat-powered vehicles beyond 2035, when the ban on gasoline and diesel cars comes into effect. This is provided that these vehicles run on synthetic, climate-neutral fuels. Additionally, European energy ministers have approved the regulation to phase out gasoline and diesel engines by 2035 through a majority vote.

Public authorities collect air quality and weather observations in near-real time from ground sensor stations and store them in digital archives. Ground sensors data are composed of long time series of observations and sensors metadata, including coordinates, type of measured variable, etc. Often, observations from different sensors and/or providers require different patterns for data accessing, harmonization, and processing. Interactive applications and dashboards, capable of facilitating such tasks, are key for supporting both public authorities as well as ordinary citizens in data processing and visualization [1].

Our app, **iAPP4P** (interactive Application for Air Pollution monitoring) aims to provide people with a clear and scientifically based view of the current state of knowledge about air pollutants and their potential environmental impacts on human health.

1.3 Definitions, acronyms, abbreviations

- **RASD:** Requirements Analysis and Specification Document.
- **OSM:** Open Street Map.
- **DBMSs:** DataBase Management Systems.
- **SQL:** Strcutured Query Language.
- **EEA:** European Environment Agency.

1.4 Scope and limitations

The primary goal of developing this product is to raise awareness and engage users in understanding the air pollution situation in the most polluted cities in Europe. Our objective is to create an Open-Source application that enables users to access, visualize, analyze, and save data related to air pollution. Through this application, users will have the opportunity to gain insights into the chosen data.

The cities available for the analysis belongs to the European countries shown in the Table 1[2].

Andorra	Austria
Belgium	Bosnia
Czech Republic	Denmark
Estonia	Finland
Germany	Spain
Sweden	Switzerland

Table 1: European Countries with dataset of interest

This cities will be evaluated in order to discern the differences in the main pollution indices.
Pollutants can have negative impacts on human health, as they can be inhaled deep into the lungs and bloodstream and cause respiratory problems as asthma, bronchitis and cardiovascular problems. Long-term exposure can increase the risk of chronic diseases such as lung cancer and heart diseases. Furthermore, they can also affect visibility, air quality, and the health of the environment around us.
The design document is primarily a high-level description of the application and its functionality.
It focuses on providing a conceptual understanding of the system rather than exploring specific implementation details. Therefore, finer technical aspects and are not covered in this document. It also does not address potential issues or challenges that may arise during the development process. For a deeper understanding of the software's scope and limitations, it is recommended to refer to the complementary Requirements Analysis Specification Document (RASD). The RASD provides more detailed information about the software requirements, including its capabilities and constraints.

2 Architectural design

2.1 Overview

The architectural design section of the Design Document provides an overview of the overall structure and organization of the software system being developed. It outlines the major components within the system and their interactions. This section focuses on the high-level design choices and principles that will form the architecture of the system.

2.2 Component diagrams

Component diagrams provide a visual representation of the different components of a system and how they interact with each other. They help to understand the structure and organization of the system's architecture.

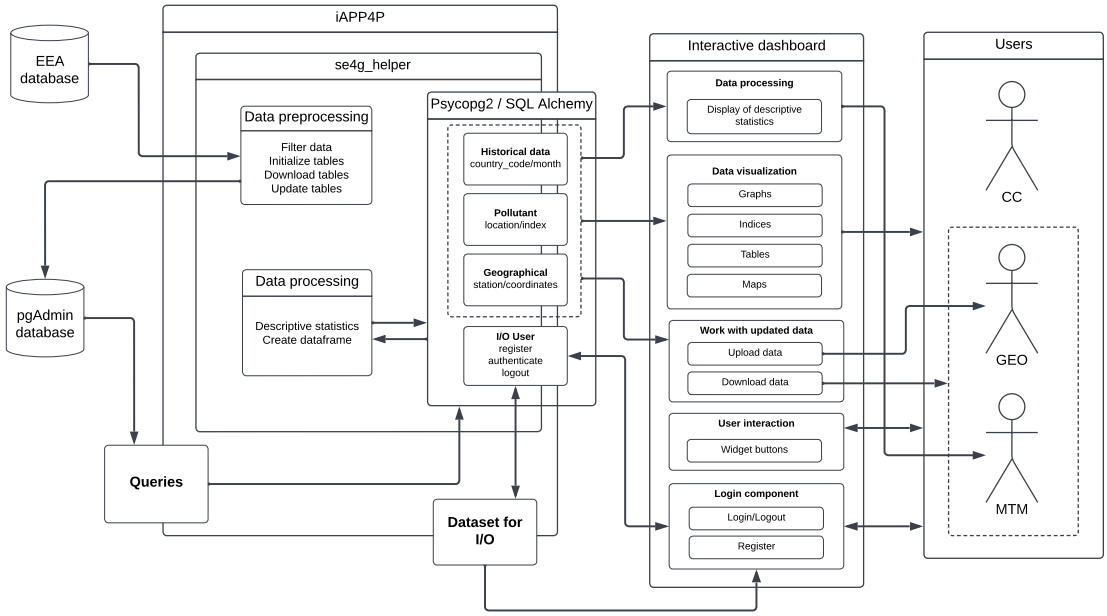

Figure 1: Architectural components diagram

The system consists of a database to ingest and store the data, psycopg2 and SQLAlchemy packages for querying the databases and retrieve the data through a python script, and a jupyter dashboard.
The software is also divided into three main programs:

Database Configuration Script: This script is responsible for creating the database structure. Setting up the database configuration is essential to establish a server that can store, retrieve, and manipulate the necessary data according to user requirements. In our configuration script, we develop the code that enables us to fetch and manipulate the data that needs to be stored in the database.
We are utilizing the EEA database for data preprocessing tasks. The EEA database contains a wealth of environmental data and information that is relevant to the project. It is initialized to have four tables: *pollution_detection, stations, se4g_pollution_main*. Additionally, we are using the pgAdmin database, an administrator and management tool for PostgreSQL, to execute queries and perform operations.

Python Script: These Python scripts serve as an intermediary layer between the data stored in the database and the visualization components of the dashboard. The Python scripts include code that connects to the database, retrieves the relevant data, and processes it to prepare it for visualization. They leverage various libraries and tools to handle data manipulation. The scripts extract the necessary data from the database, apply any required data transformations or calculations, and organize it in a format suitable for the dashboard. They ensure that the data is structured and formatted correctly for visualization purposes.

Interactive Dashboard: Finally, we have developed an interactive dashboard with Jupyter NoteBook that provides a user-friendly and visually appealing interface for data exploration and analysis. The interactive dashboard allows users to:

- Visualize basic statistics of the pollution data
- Select a day to visualize data
- Visualize histograms
- Select pollutant and shows time series of all the cities
- Visualize correlation graphs

To improve the user experience, the interactive dashboard combines numerous data processing features. It offers descriptive statistics to deliver insightful analyses of the data. Through interactive visualizations such as graphs, indices and tables, users may explore and analyze the data. The dashboard also has a map feature that shows the most recent information for particular cities, enabling users to visually analyze regional trends and patterns. The dashboard includes a login component that supports login, logout, and registration features to guarantee secure access.
In the Figure 1, you can get a visual overview of everything that was just explained.

3 Software Design

3.1 Database design

The software will use and interact with DBMS. PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. Databases will incorporate primary keys, which are unique identifiers assigned to each entity. This primary key will allow for clear identification of individual entities.
Setting up the database configuration is essential in order to have a server where the required data can be stored, retrieved, and from which it is possible to be manipulated at the user's need. To interact with PostgreSQL effectively, we need to perform a series of operations that involve establishing a connection, sending SQL commands or making changes, and finally, closing the connection to the server.
To facilitate this interaction between our software and PostgreSQL, we rely on specific libraries and functions that optimize the process:

Libraries:

- Psycopg2
- SQLAlchemy

In this and then they are filled with the appropriate data to start with.

Table	Usage	Columns
station	Stores data about stations	station_code, station_name, station_altitude, network_countrycode, samplingpoint_x, samplingpoint_y
pollution_detection	Stores coordinates	station_code, pollutant, value_datetime_begin, value_datetime_end, value_numeric
se4g_pollution_main	Attributes about dataset	station_code, station_name, station_altitude, network_countrycode, pollutant, value_datetime_begin, value_datetime_end, value_datetime_updated, value_numeric, samplingpoint_x, samplingpoint_y

Table 2: Database structure

3.2 Database connection

In order to work with the same up-to-date data these functions have been defined to connect to an unique database, which for a matter of efficiency, is managed by a single administrator. In this way other administrators can directly draw on the current data in the main administrator's database.

The first function establishes a connection to a PostgreSQL database using the psycopg2 library. It takes as parameters the file containing username and password of the administrator, the db_user (postgres), the IP of the administrator device, and the database name.

The resulting connection object is assigned to the variable 'conn' and if the connection is successfully established, the 'conn' object is returned. If an exception of type 'psycopg2.Error' occurs during the connection attempt, an error message is printed, indicating the failure to connect to the database.

```
file = 'bin.txt'
db_user = "postgres"
ip = '192.168.30.19'
ip = 'localhost'
database = "se4g"
port = "5432"
```

```
import psycopg2

def connect_right_now(
    file: str = file,
    db_user: str = db_user,
    ip: str = ip,
    database: str = database,
):
    try:
        with open('code/'+file, 'r') as f:
            conn = psycopg2.connect(
                host = ip,
                database = database,
                user = db_user,
                password = f.read()
            )
        #print('connected with ',ip, ' through psycopg2')
        return conn
    except psycopg2.Error as e:
        print(f"Error connecting to the database: {e}")
```

Instead, the second function establishes a connection to a PostgreSQL database using the SQLAlchemy library. Firstly, the code imports the 'create_engine' function from the SQLAlchemy module. Then the function 'connect_with_sqlalchemy' takes as parameters 'file', 'db_user', 'ip', 'database', and 'port', with the same meaning of the previous code. 'engine' represents the connection to the database. It uses a formatted string to construct the connection URL, which includes the PostgreSQL credentials (username and password), IP address, port, and database name. If the engine is successfully created, the 'engine' object is returned. If an exception of type 'create_engine.Error' occurs during the connection attempt, an error message is printed, indicating the failure to connect to the database.

```
from sqlalchemy import create_engine

def connect_with_sqlalchemy(
    file: str = file,
    db_user: str = db_user,
    ip: str = ip,
    database: str = database,
    port: str = port
```

```
    ):
    try:
        with open('code/'+file, 'r') as f:
            engine = create_engine(f'postgresql://{db_user}:{f.read()}@{ip}:{port}/{
                database}')
        #print('connected with ',ip, ' through sqlalchemy')
        return engine
    except create_engine.Error as e:
        print(f"Error connecting to the database: {e}")
```

3.3 Dashboard design

When designing a dashboard for a project, it is important to consider both functional and aesthetic aspects. Here are some important elements to include in our dashboard:

- Dropdowns and Radio items
- Scatter Plot
- Time Series Graphs
- Slider
- Callback Functions

In general, the dashboard design provides an interactive data exploration interface where users can select X-axis and Y-axis variables,filter the data by year and view the corresponding scatter plot and time series graphs. The following code defines a class called Dashboard that represents a data visualization dashboard.
Our project leverage two key classes,Folium Map and Dashboard, to enhance data visualization and interaction capabilities.

Class FoliumMap: This code defines a class with an initializer (init) and a method (get_columns). In the init method: It sets the table_name and db_columns instance variables to the provided values or default values if none are provided and it initializes the selected_pollutant and selected_datetime variables to None.
The get_columns method: establishes a database connection,creates a cursor to execute SQL queries, constructs an SQL query to select distinct values from the specified column (db_columns) in the specified table (table_name_clmns),executes the SQL query using the cursor,retrieves the column names and rows fetched from the database ,closes the cursor and the database connection. It constructs a Pandas DataFrame from the retrieved data, with column names taken from the cursor description and rows from the fetched data and returns the resulting DataFrame.

```
def __init__(self, table_name='se4g_pollution_main', db_columns='value_numeric,
    samplingpoint_x, samplingpoint_y'):
        self.table_name = table_name
        self.db_columns = db_columns
        self.selected_pollutant = None
        self.selected_datetime = None

    def get_columns(self, table_name_clmns='se4g_pollution_main', db_columns='
        value_datetime_end'):
```

```
        conn = connect_right_now()
        cursor = conn.cursor()

        # Generate the SQL statement to select data from the source table
        select_data_query = f"SELECT DISTINCT {db_columns} FROM {table_name_clmns};"

        # Execute the SELECT command
        cursor.execute(select_data_query)

        clmns = [desc[0] for desc in cursor.description]
        # Fetch all the rows
        rows_clmns = cursor.fetchall()

        cursor.close()
        conn.close()

        return pd.DataFrame(rows_clmns, columns=clmns)
```

The code defines two methods:

The select_filters method
It takes two optional parameters, pollutants and datetime_list, which default to a list of pollutants and a list of datetime values fetched from the database, creates dropdown widgets for selecting a pollutant and datetime,displays the dropdown widgets and it sets the selected pollutant and datetime to the values chosen in the dropdowns.

The update_maps method:
It establishes a database connection,creates a cursor to execute SQL queries,constructs an SQL query to select data from the table specified in self.table_name with the chosen pollutant and datetime values,executes the SQL query using the cursor. The code retrieves the column names and rows fetched from the database,constructs a DataFrame from the fetched data, including geometric information for mapping,creates a GeoDataFrame using the DataFrame and assigns a coordinate reference system. Subsequently it updates the marker map by creating markers on the Folium map for each location in the GeoDataFrame and updates the heat map by generating a heat map using the coordinates from the GeoDataFrame. It formats the legend HTML content based on the selected pollutant and datetime and it clears the previous legend and adds the updated legend to the marker and heat maps.At the end it displays the updated marker and heat maps.

```
def select_filters(self, pollutants=None, datetime_list=None):
        if pollutants is None:
            pollutants = ['SO2', 'NO', 'NO2', 'CO', 'PM10']
        if datetime_list is None:
            datetime_list = self.get_columns()['value_datetime_end'].tolist()

        pollutant_dropdown = widgets.Dropdown(
            options=pollutants,
            description='Select pollutant:'
        )
        datetime_dropdown = widgets.Dropdown(
            options=datetime_list,
            description='Select datetime:'
```

```
        )

        display(pollutant_dropdown)
        display(datetime_dropdown)

        self.selected_pollutant = pollutant_dropdown
        self.selected_datetime = datetime_dropdown

        return pollutant_dropdown, datetime_dropdown

    def update_maps(self, change):
        conn = connect_right_now()
        cursor = conn.cursor()

        # Generate the SQL statement to select data from the source table
        select_data_query = f"SELECT {self.db_columns} FROM {self.table_name} WHERE
            pollutant = '{self.selected_pollutant.value}' AND value_datetime_end = '{
            self.selected_datetime.value}';"

        # Execute the SELECT command
        cursor.execute(select_data_query)

        columns = [desc[0] for desc in cursor.description]
        # Fetch all the rows
        rows = cursor.fetchall()

        df = pd.DataFrame(rows, columns=columns)

        geom_list = [Point(xy) for xy in zip(df['samplingpoint_x'], df['samplingpoint_y
            '])]
        crs = 'epsg:4979'
        gdf = gpd.GeoDataFrame(df, crs=crs, geometry=geom_list)

        # Update the MARKER MAP
        m_folium = folium.Map(location=[57, 15], zoom_start=2.5, tiles='CartoDB
            positron')

        for index, row in gdf.iterrows():
            folium.Marker(
                location=[row['geometry'].y, row['geometry'].x],
                popup=row['value_numeric'],
                icon=folium.map.Icon(color='red')
            ).add_to(m_folium)

        # Update the HEAT MAP
        m_heat = folium.Map(location=[55, 15], tiles='Cartodb dark_matter', zoom_start
            =2.5)

        heat_data = [[point.xy[1][0], point.xy[0][0]] for point in gdf.geometry]

        plugins.HeatMap(heat_data).add_to(m_heat)

        # Update the legend HTML content
        pollutant_selected = self.selected_pollutant.value
        datetime_formatted = pd.to_datetime(self.selected_datetime.value).strftime('%y
```

```
            -%m-%d %H:%M'
                        "<div style='position:fixed; top:10px; left:10px; background-
            color:white; padding:5px; border:1px solid gray; z-index:9999; font-size:12
            px;'>"
                    "<b>Selected Filters:</b><br>"
                    "Pollutant: {pollutant_selected}<br>"
                    "Datetime: {datetime_formatted}"
                    "</div>"

        # Clear previous legend and add the updated legend to the map



        # Clear previous legend and add the updated legend to the map



        # Display the updated maps


```

The last function of this class called load_data that loads data from a specified database table. The function connects to the database,executes an SQL query to select all data from the specified table. It fetches the column names and rows from the executed query and then closes the database connection. Afterwards it creates a pandas DataFrame from the fetched rows, using the column names as column labels and performs some data transformations and manipulations on the DataFrame, including creating a new column for the time series, filtering out rows with a specific country, and converting a month-day string to a datetime object.

```
def load_data(table_name):
    # Connect to the database and fetch the data
    conn = connect_right_now()
    cursor = conn.cursor()

    # Generate the SQL statement to select data from the source table
    select_data_query = f"SELECT * FROM {table_name};"

    # Execute the SELECT command
    cursor.execute(select_data_query)

    columns = [desc[0] for desc in cursor.description]

    # Fetch all the rows
    rows = cursor.fetchall()

    cursor.close()
    conn.close()

    # Create a pandas DataFrame from the fetched rows
    df = pd.DataFrame(rows, columns=columns)

    unique_month_day = df['month_day'].unique()
```

```
    month_day_dict = {day: index + 1 for index, day in enumerate(unique_month_day)}

    df['time_series'] = df['month_day'].map(month_day_dict)
    df = df[df['country'] != 'Bosnia and Herzegovina']

    df['month_day_date'] = '2023' + df['month_day'].astype(str)
    df['month_day_date'] = pd.to_datetime(df['month_day_date'], format='%Y%m%d')

    return df
```

Class Dashboard: In this code, a class called Dashboard is defined and initializer methods are provided. When an instance of this class is created,the app attribute is initialized with an instance of Jupyter Dash and the df attribute with an empty DataFrame. The class also has a load_data method that connects to the database, executes an SQL query,retrieves data from the table specified by self.table_name, and stores the retrieved data as a DataFrame in the df attribute. It also retrieves the column names from the cursor description. The code fetches data, creates a DataFrame, performs data manipulations, and prepares the data for visualization in a few concise steps.

```
def __init__(self, table_name='se4g_dashboard'):
        self.app = JupyterDash(__name__)
        self.df = load_data(table_name) # Initialize an empty DataFrame

    def load_data(self):
        # Connect to the database and fetch the data
        conn = connect_right_now()
        cursor = conn.cursor()

        # Generate the SQL statement to select data from the source table
        select_data_query = f"SELECT * FROM {self.table_name};"

        # Execute the SELECT command
        cursor.execute(select_data_query)

        columns = [desc[0] for desc in cursor.description]
# Fetch all the rows
        rows = cursor.fetchall()

        cursor.close()
        conn.close()

        # Create a pandas DataFrame from the fetched rows
        self.df = pd.DataFrame(rows, columns=columns)

        unique_month_day = self.df['month_day'].unique()
        month_day_dict = {day: index + 1 for index, day in enumerate(unique_month_day)}

        self.df['time_series'] = self.df['month_day'].map(month_day_dict)
        self.df = self.df[self.df['country'] != 'Bosnia and Herzegovina']

        self.df['month_day_date'] = '2023' + self.df['month_day'].astype(str)
        self.df['month_day_date'] = pd.to_datetime(self.df['month_day_date'], format='%
            Y%m%d')
```

The create_dashboard method is responsible for setting up the layout of the dashboard. It includes dropdown menus, radio items, graphs, and a slider for interactive data visualization. Div sections, and the dropdowns and radio items are populated with dynamic options. The graphs display scatter plots and time series data. The slider allows selecting a specific time range based on the available data. Overall, the code sets up a visually appealing and interactive dashboard for data exploration and analysis.

```
def create_dashboard(self):
      available_indicators = self.df['pollutant'].unique()

      external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

      self.app = JupyterDash(__name__, external_stylesheets=external_stylesheets)

      self.app.layout = html.Div([
          html.Div([
              html.Div([
                  dcc.Dropdown(
                      id='crossfilter-xaxis-column',
                      options=[{'label': i, 'value': i} for i in available_indicators],
                      value='SO2'
                  ),
                  dcc.RadioItems(
                      id='crossfilter-xaxis-type',
                      options=[{'label': i, 'value': i} for i in ['Linear', 'Log']],
                      value='Linear',
                      labelStyle={'display': 'inline-block'}
                  )
              ],
                  style={'width': '49%', 'display': 'inline-block'}),

              html.Div([
                  dcc.Dropdown(
                      id='crossfilter-yaxis-column',
                      options=[{'label': i, 'value': i} for i in available_indicators],
                      value='CO'
                  ),
                  dcc.RadioItems(
                      id='crossfilter-yaxis-type',
                      options=[{'label': i, 'value': i} for i in ['Linear', 'Log']],
                      value='Linear',
                      labelStyle={'display': 'inline-block'}
                  )
              ], style={'width': '49%', 'float': 'right', 'display': 'inline-block'})
          ], style={
              'borderBottom': 'thin lightgrey solid',
              'backgroundColor': 'rgb(250, 250, 250)',
              'padding': '10px 5px'
          }),

          html.Div([
              dcc.Graph(
                  id='crossfilter-indicator-scatter',
```

```
                hoverData={'points': [{'customdata': 'Andorra'}]}
            )
        ], style={'width': '49%', 'display': 'inline-block', 'padding': '0 20'}),
        html.Div([
            dcc.Graph(id='x-time-series'),
            dcc.Graph(id='y-time-series'),
        ], style={'display': 'inline-block', 'width': '49%'}),

        html.Div(dcc.Slider(
            id='crossfilter-year--slider',
            min=self.df['time_series'].min(),
            max=self.df['time_series'].max(),
            value=self.df['time_series'].max(),
            marks={str(time): str(time) for time in self.df['month_day_date'].unique
                ()},
            step=None
        ), style={'width': '49%', 'padding': '0px 20px 20px 20px'})
    ])
```

The @self.app.callback decorator links a callback function to an output element and specifies the input elements that trigger the callback. When the input elements change, the callback function is executed, and its result updates the specified output element in the Dash application's user interface. It updates the 'crossfilter-indicator-scatter' figure in a Dash application based on changes in several input elements: 'crossfilter-xaxis-column', 'crossfilter-yaxis-column', 'crossfilter-xaxis-type', 'crossfilter-yaxis-type', and 'crossfilter-year–slider'. Overall this callback function updates the scatter plot's data and layout based on the selected inputs, allowing the plot to dynamically respond to user interactions in the Dash application.

```
@self.app.callback(
        dash.dependencies.Output('crossfilter-indicator-scatter', 'figure'),
        [dash.dependencies.Input('crossfilter-xaxis-column', 'value'),
         dash.dependencies.Input('crossfilter-yaxis-column', 'value'),
         dash.dependencies.Input('crossfilter-xaxis-type', 'value'),
         dash.dependencies.Input('crossfilter-yaxis-type', 'value'),
         dash.dependencies.Input('crossfilter-year--slider', 'value')])
    def update_graph(xaxis_column_name, yaxis_column_name,
                     xaxis_type, yaxis_type,
                     year_value):
        dff = self.df[self.df['time_series'] == year_value]

        return {
            'data': [dict(
                x=dff[dff['pollutant'] == xaxis_column_name]['value_numeric_mean'],
                y=dff[dff['pollutant'] == yaxis_column_name]['value_numeric_mean'],
                text=dff[dff['pollutant'] == yaxis_column_name]['country'],
                customdata=dff[dff['pollutant'] == yaxis_column_name]['country'],
                mode='markers',
                marker={
                    'size': 25,
                    'opacity': 0.7,
                    'color': 'orange',
                    'line': {'width': 2, 'color': 'purple'}
```

```
                }
            )],
            'layout': dict(
                xaxis={
                    'title': xaxis_column_name,
                    'type': 'linear' if xaxis_type == 'Linear' else 'log'
                },
                yaxis={
                    'title': yaxis_column_name,
                    'type': 'linear' if yaxis_type == 'Linear' else 'log'
                },
                margin={'l': 40, 'b': 30, 't': 10, 'r': 0},
                height=450,
                hovermode='closest'
            )
        }
```

The create_time_series function generates data and layout properties for a line chart. It takes a subset of filtered data (dff), the type of axis(linear or logarithmic), and a title.The function returns a dictionary containing the line chart data and layout properties. The self.app.callback decorator defines a call back function that is triggered by specific events in the Dash application.They specify the input component that will trigger the callback and the output component that will be updated. When the value of the specified input component changes, the associated callback function is executed. The callback functions update_x_timeseries and update_y_timeseries update the numerical values in the x-and y-time series plots, respectively, based on user manipulations of the scatter plots and selected axis options.

```
def create_time_series(dff, axis_type, title):
        return {
            'data': [dict(
                x=dff['time_series'],
                y=dff['value_numeric_mean'],
                mode='lines+markers'
            )],
            'layout': {
                'height': 225,
                'margin': {'l': 20, 'b': 30, 'r': 10, 't': 10},
                'annotations': [{
                    'x': 0, 'y': 0.85, 'xanchor': 'left', 'yanchor': 'bottom',
                    'xref': 'paper', 'yref': 'paper', 'showarrow': False,
                    'align': 'left', 'bgcolor': 'rgba(255, 255, 255, 0.5)',
                    'text': title
                }],
                'yaxis': {'type': 'linear' if axis_type == 'Linear' else 'log'},
                'xaxis': {'showgrid': False}
            }
        }

    @self.app.callback(
        dash.dependencies.Output('x-time-series', 'figure'),
        [dash.dependencies.Input('crossfilter-indicator-scatter', 'hoverData'),
         dash.dependencies.Input('crossfilter-xaxis-column', 'value'),
         dash.dependencies.Input('crossfilter-xaxis-type', 'value')])
```

```
        def update_x_timeseries(hoverData, xaxis_column_name, axis_type):
            country_name = hoverData['points'][0]['customdata']
            dff = self.df[self.df['country'] == country_name]
            dff = dff[dff['pollutant'] == xaxis_column_name]
            title = '<b>{}</b><br>{}'.format(country_name, xaxis_column_name)
            return create_time_series(dff, axis_type, title)

        @self.app.callback(
            dash.dependencies.Output('y-time-series', 'figure'),
            [dash.dependencies.Input('crossfilter-indicator-scatter', 'hoverData'),
             dash.dependencies.Input('crossfilter-yaxis-column', 'value'),
             dash.dependencies.Input('crossfilter-yaxis-type', 'value')])
        def update_y_timeseries(hoverData, yaxis_column_name, axis_type):
            dff = self.df[self.df['country'] == hoverData['points'][0]['customdata']]
            dff = dff[dff['pollutant'] == yaxis_column_name]
            return create_time_series(dff, axis_type, yaxis_column_name)

    def run(self):
        #self.load_data()
        self.create_dashboard() # Set up the layout of the application
        self.app.run_server(mode='inline') # Change mode to 'external' if using Jupyter
            Notebook
```

References

[1] G. E. Quattrocchi, “Se4geo - project,” 2023.

[2] “Download of utd air quality data,” 2023-04-05. [Online]. Available: https://discomap.eea.europa.eu/map/fme/AirQualityUTDExport.htm