DATA SCIENCE

Building a Data Dashboard

Using the streamlit Python library

Thomas Reid

Jan 20, 2025 15 min read

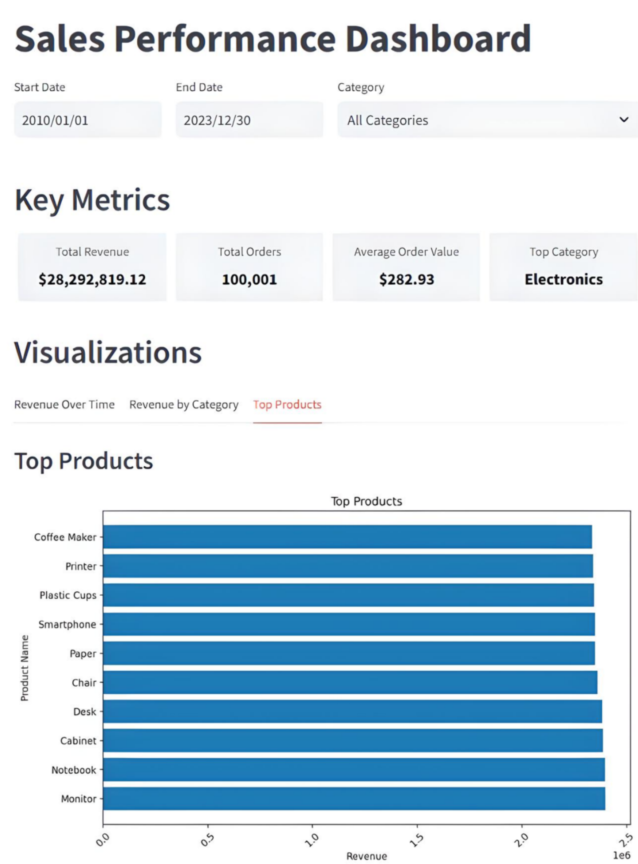

Image by Author

With source data from a Postgres database

As a Python data engineer for many years, one area I was not very involved in was the production of data dashboards. That all changed when Python-based libraries such as Streamlit, Gradio and Taipy came along.

With their introduction, python programmers had no excuses ***not*** to use them to craft nice-looking front-ends and dashboards.

Until then, the only other options were to use specialised tools like Tableau or AWS's Quicksight or—horror of horrors—get your hands dirty with CSS, HTML, and Javascript.

So, if you've never used one of these new Python-based graphical front-end libraries before, this article is for you as I'll be taking you through how to code up a data dashboard using one of the most popular libraries for this purpose called `Streamlit`.

> My intention is that this will be the first part of a series of articles on developing a data dashboard using three of the most popular Python-based GUI libraries. In addition to this one, I also plan to release articles on Gradio and Taipy, so look out for those. As much as possible I'll try to replicate the same layout and functionality in each dashboard. I'll use the exact same data for all three too, albeit in different formats e.g. a CSV, database etc …

> Please also note that I have no connection or affiliation with Streamlit/Snowflake, Postgres or any other company or tool mentioned in this post.

What is Streamlit?

Founded in 2018 by Adrien Treuille,

Amanda Kelly, and Thiago Teixeira, Streamlit quickly gained popularity among data scientists and machine learning engineers when it introduced its open-source Python framework to simplify the creation of interactive data applications.

In March 2022, Snowflake, a Data Cloud company, acquired Streamlit and its capabilities were integrated into the Snowflake ecosystem to enhance data application development.

Streamlit's open-source framework has been widely adopted, with over 8 million downloads and more than 1.5 million applications built using the platform. An active community of developers and contributors continues to play a significant role in its ongoing development and success.

What we'll develop

We're going to develop a data

dashboard. Our source data for the dashboard will be in a single Postgres database table and contain 100,000 synthetic sales records.

To be honest, the actual source of the data isn't ***that*** important. It could just as easily be a text or CSV file, SQLite, or any database you can connect to. I chose Postgres because I have a copy on my local PC, and it's convenient for me to use.

This is what our final dashboard will look like.

Image by Author

There are four main sections.

- The top row allows the user to choose specific start and end dates and/or product categories via date pickers and a drop-down list, respectively.

- The second row – Key metrics – shows a top-level summary of the chosen data.
- The Visualisation section allows the user to select one of three graphs to display the input data set.
- The raw data section is exactly what it says. This is a tabular representation of the chosen data, effectively viewing the underlying Postgres database table data.

Using the dashboard is easy. Initially, stats for the whole data set are displayed. The user can then narrow the data focus using the 3 choice fields at the top of the display. The graphs, key metrics and raw data sections dynamically change to reflect what the user has chosen.

The underlying data

As mentioned, the dashboard's source data is contained in a single Postgres database table. The data is a set of 100,000 synthetic sales-related data records. Here is the Postgres table creation script for reference.

```
CREATE TABLE IF NOT EXISTS public.sa
(
    order_id integer NOT NULL,
    order_date date,
    customer_id integer,
    customer_name character varying(
    product_id integer,
    product_names character varying(
    categories character varying(10(
    quantity integer,
    price numeric(10,2),
    total numeric(10,2)
)
```

And here is some Python code you can use to generate a data set for yourself. Make sure both numpy and polars libraries are installed first

```
# generate the 1m record CSV file
#
import polars as pl
```

```python
import numpy as np
from datetime import datetime, timed

def generate(nrows: int, filename: s
    names = np.asarray(
        [
            "Laptop",
            "Smartphone",
            "Desk",
            "Chair",
            "Monitor",
            "Printer",
            "Paper",
            "Pen",
            "Notebook",
            "Coffee Maker",
            "Cabinet",
            "Plastic Cups",
        ]
    )

    categories = np.asarray(
        [
            "Electronics",
            "Electronics",
            "Office",
            "Office",
            "Electronics",
            "Electronics",
            "Stationery",
            "Stationery",
            "Stationery",
            "Electronics",
            "Office",
            "Sundry",
        ]
    )
```

```
    product_id = np.random.randint(1
    quantity = np.random.randint(1,
    price = np.random.randint(199, 1

    # Generate random dates between
    start_date = datetime(2010, 1, 1
    end_date = datetime(2023, 12, 31
    date_range = (end_date - start_d

    # Create random dates as np.arra
    order_dates = np.array([(start_d

    # Define columns
    columns = {
        "order_id": np.arange(nrows)
        "order_date": order_dates,
        "customer_id": np.random.ran
        "customer_name": [f"Customer
        "product_id": product_id + 2
        "product_names": names[produ
        "categories": categories[pro
        "quantity": quantity,
        "price": price,
        "total": price * quantity,
    }

    # Create Polars DataFrame and wr
    df = pl.DataFrame(columns)
    df.write_csv(filename, separator

# Generate 100,000 rows of data with
generate(100_000, "/mnt/d/sales_data
```

Setting up our development

environment

Before we get to the example code, let's set up a separate development environment. That way, what we do won't interfere with other versions of libraries, programming, etc… we might have on the go for other projects we're working on.

I use Miniconda for this, but you can use whatever method suits you best.

If you want to go down the Miniconda route and don't already have it, you must install Miniconda first. Get it using this link,

> **Miniconda – Anaconda documentation**

Once the environment is created, switch to it using the `activate` command, and then `pip install` our required Python libraries.

```
#create our test environment
```

```
(base) C:Usersthoma>conda create -n


# Now activate it
(base) C:Usersthoma>conda activate s


# Install python libraries, etc ...
(streamlit_test) C:Usersthoma>pip ir
```

The Code

I’ll split the code up into sections and explain each one along the way.

```
#
# Streamlit equivalent of final Grac
#
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import psycopg2
from psycopg2 import sql
from psycopg2 import pool

# Initialize connection pool
try:
    connection_pool = psycopg2.pool.
        minconn=5,
        maxconn=20,
        dbname="postgres",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432"
```

```
    )
except psycopg2.Error as e:
    st.error(f"Error creating connec

def get_connection():
    try:
        return connection_pool.getco
    except psycopg2.Error as e:
        st.error(f"Error getting cor
        return None

def release_connection(conn):
    try:
        connection_pool.putconn(conr
    except psycopg2.Error as e:
        st.error(f"Error releasing c
```

We start by importing all the external libraries we'll need. Next, we set up a **ThreadedConnectionPool** that allows **** multiple threads to share a pool of database connections. Two helper functions follow, one to get a database connection and the other to release it. This is overkill for a simple single-user app but essential for handling multiple simultaneous users or threads accessing the database in a web app environment.

```
def get_date_range():
    conn = get_connection()
    if conn is None:
        return None, None
    try:
        with conn.cursor() as cur:
            query = sql.SQL("SELECT
            cur.execute(query)
            return cur.fetchone()
    finally:
        release_connection(conn)

def get_unique_categories():
    conn = get_connection()
    if conn is None:
        return []
    try:
        with conn.cursor() as cur:
            query = sql.SQL("SELECT
            cur.execute(query)
            return [row[0].capitaliz
    finally:
        release_connection(conn)

def get_dashboard_stats(start_date,
    conn = get_connection()
    if conn is None:
        return None
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                WITH category_totals
                    SELECT
                        categories,
                        SUM(price *
                    FROM public.sale
```

```
                WHERE order_date
                AND (%s = 'All C
                GROUP BY categor
            ),
            top_category AS (
                SELECT categorie
                FROM category_to
                ORDER BY categor
                LIMIT 1
            ),
            overall_stats AS (
                SELECT
                    SUM(price *
                    COUNT(DISTIN
                    SUM(price *
                FROM public.sale
                WHERE order_date
                AND (%s = 'All C
            )
            SELECT
                total_revenue,
                total_orders,
                avg_order_value,
                (SELECT categori
            FROM overall_stats
        """)
        cur.execute(query, [star
                            star
        return cur.fetchone()
finally:
    release_connection(conn)
```

The **get_date_range** function executes the SQL query to find the range of dates (`MIN` and `MAX`) in the `order_date` column and returns the two dates as a tuple: `(start_date, end_date)`.

The **get_unique_categories** function runs an SQL query to fetch unique values from the `categories` column. It capitalizes the category names (first letter uppercase) before returning them as a list.

The **get_dashboard_stats** function executes a SQL query with the following parts:

- **`category_totals`**: Calculates total revenue for each category in the given date range.
- **`top_category`**: Finds the category with the highest revenue.
- **`overall_stats`**: Computes overall statistics:
- Total revenue (`SUM(price * quantity)`).
- Total number of unique orders (`COUNT(DISTINCT order_id)`).
- Average order value (total

revenue divided by total orders).

It returns a single row containing:

- `total_revenue`: Total revenue in the specified period.
- `total_orders`: Number of distinct orders.
- `avg_order_value`: Average revenue per order.
- `top_category`: The category with the highest revenue.

```
def get_plot_data(start_date, end_da
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT DATE(order_da
                       SUM(price * c
                FROM public.sales_da
                WHERE order_date BET
                  AND (%s = 'All Cat
                GROUP BY DATE(order_
                ORDER BY date
            """)
            cur.execute(query, [star
            return pd.DataFrame(cur.
    finally:
```

```
        release_connection(conn)

def get_revenue_by_category(start_da
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT categories,
                       SUM(price * c
                FROM public.sales_da
                WHERE order_date BET
                  AND (%s = 'All Cat
                GROUP BY categories
                ORDER BY revenue DES
            """)
            cur.execute(query, [star
            return pd.DataFrame(cur.
    finally:
        release_connection(conn)

def get_top_products(start_date, enc
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT product_names
                       SUM(price * c
                FROM public.sales_da
                WHERE order_date BET
                  AND (%s = 'All Cat
                GROUP BY product_nam
                ORDER BY revenue DES
                LIMIT 10
```

```
            """)
            cur.execute(query, [star
            return pd.DataFrame(cur.
    finally:
        release_connection(conn)

def get_raw_data(start_date, end_dat
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT
                    order_id, order_
                    product_id, proc
                    (price * quantit
                FROM public.sales_da
                WHERE order_date BET
                  AND (%s = 'All Cat
                ORDER BY order_date,
            """)
            cur.execute(query, [star
            return pd.DataFrame(cur.
    finally:
        release_connection(conn)

def plot_data(data, x_col, y_col, ti
    fig, ax = plt.subplots(figsize=(
    if not data.empty:
        if orientation == 'v':
            ax.bar(data[x_col], data
        else:
            ax.barh(data[x_col], dat
        ax.set_title(title)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
```

```
        plt.xticks(rotation=45)
    else:
        ax.text(0.5, 0.5, "No data a
    return fig
```

The **get_plot_data** function fetches daily revenue within the given date range and category. It retrieves data grouped by the day (`DATE(order_date)`) and calculates daily revenue (`SUM(price * quantity)`), then returns a Pandas DataFrame with columns: `date` (the day) and `revenue` (total revenue for that day).

The **get_revenue_by_category** function fetches revenue totals grouped by category within the specified date range. It groups data by `categories` and calculates revenue for each category (`SUM(price * quantity)`), orders the results by revenue in descending order and returns a Pandas DataFrame with columns: `categories` (category name) and `revenue` (total revenue for the category).

The **get_top_products** function retrieves the top 10 products by revenue within the given date range and category. It groups data by `product_names` and calculates revenue for each product (`SUM(price * quantity)`), orders the products by revenue in descending order and limits results to the top 10 before returning a Pandas DataFrame with columns: `product_names` (product name) and `revenue` (total revenue for the product).

The **get_raw_data** function fetches raw transaction data within the specified date range and category.

The **plot_data** function takes in some data (in a pandas DataFrame) and the names of the columns you want to plot on the x- and y-axes. It then creates a bar chart – either vertical or horizontal, depending on the chosen orientation – labels the axes, adds a title, and returns the finished chart (a Matplotlib Figure). If the data is empty, it just displays a "No data available" message instead of trying to plot anything.

```
# Streamlit App
st.title("Sales Performance Dashboar

# Filters
with st.container():
    col1, col2, col3 = st.columns([1
    min_date, max_date = get_date_ra
    start_date = col1.date_input("St
    end_date = col2.date_input("End
    categories = get_unique_categori
    category = col3.selectbox("Categ

# Custom CSS for metrics
st.markdown("""
    <style>
    .metric-row {
        display: flex;
        justify-content: space-betwe
        margin-bottom: 20px;
    }
    .metric-container {
        flex: 1;
        padding: 10px;
        text-align: center;
        background-color: #f0f2f6;
        border-radius: 5px;
        margin: 0 5px;
    }
    .metric-label {
```

```
        font-size: 14px;
        color: #555;
        margin-bottom: 5px;
    }
    .metric-value {
        font-size: 18px;
        font-weight: bold;
        color: #0e1117;
    }
    </style>
""", unsafe_allow_html=True)

# Metrics
st.header("Key Metrics")
stats = get_dashboard_stats(start_da
if stats:
    total_revenue, total_orders, avg
else:
    total_revenue, total_orders, avg

# Custom metrics display
metrics_html = f"""
<div class="metric-row">
    <div class="metric-container">
        <div class="metric-label">To
        <div class="metric-value">${
    </div>
    <div class="metric-container">
        <div class="metric-label">To
        <div class="metric-value">{t
    </div>
    <div class="metric-container">
        <div class="metric-label">Av
        <div class="metric-value">${
    </div>
    <div class="metric-container">
        <div class="metric-label">To
```

```
        <div class="metric-value">{t
    </div>
</div>
"""
st.markdown(metrics_html, unsafe_all
```

This code section creates the main structure for displaying the key metrics in the Streamlit dashboard. It:

1. **Sets up the page title**: "Sales Performance Dashboard."
2. **Presents filters** for start/end dates and category selection.
3. **Retrieves metrics** (such as total revenue, total orders, etc.) for the chosen filters from the database.
4. **Applies custom CSS** to style these metrics in a row of boxes with labels and values.
5. **Displays the metrics** within an HTML block, ensuring each metric gets its own styled container.

```
# Visualization Tabs
st.header("Visualizations")
tabs = st.tabs(["Revenue Over Time",

# Revenue Over Time Tab
with tabs[0]:
    st.subheader("Revenue Over Time'
    revenue_data = get_plot_data(sta
    st.pyplot(plot_data(revenue_data

# Revenue by Category Tab
with tabs[1]:
    st.subheader("Revenue by Categor
    category_data = get_revenue_by_c
    st.pyplot(plot_data(category_dat

# Top Products Tab
with tabs[2]:
    st.subheader("Top Products")
    top_products_data = get_top_proc
    st.pyplot(plot_data(top_products
```

This section adds a header titled "Visualizations" to this part of the dashboard. It creates three tabs, each of which displays a different graphical representation of the data:

Tab 1: Revenue Over Time

- Fetches **revenue data grouped**

by date for the given filters using `get_plot_data()`.

- Calls `plot_data()` to generate a **bar chart** of revenue over time, with dates on the x-axis and revenue on the y-axis.
- Displays the chart in the first tab.

Tab 2: Revenue by Category

- Fetches **revenue grouped by category** using `get_revenue_by_category()`.
- Calls `plot_data()` to create a **bar chart** of revenue by category.
- Displays the chart in the second tab.

Tab 3: Top Products

- Fetches **top 10 products by revenue** for the given filters using `get_top_products()`.
- Calls `plot_data()` to create a **horizontal bar chart** (indicated

by `orientation='h'`).

- Displays the chart in the third tab.

```
st.header("Raw Data")

raw_data = get_raw_data(
    start_date=start_date,
    end_date=end_date,
    category=category
)

# Remove the index by resetting it a
raw_data = raw_data.reset_index(drop

st.dataframe(raw_data,hide_index=Tru

# Add spacing
st.write("")
```

The final section displays the raw data in a dataframe. The user is able to scroll up and down as required to see all records available.

An empty `st.write("")` is added at the end to provide spacing for better visual alignment.

Running the App

Let’s say you save your code into a file called app.py. You can run it using this from the command line,

```
(streamlit_test) C:Usersthoma> pytho
```

If everything works as expected, you will see this after you run the above command.

```
  You can now view your Streamlit ap

  Local URL: http://localhost:8501
  Network URL: http://192.168.0.59:8
```

Click on the Local URLs shown, and a browser screen should appear with the Streamlit app running.

Summary

In this article, I’ve attempted to provide a comprehensive guide to building an interactive sales performance dashboard using Streamlit with a Postgres database table as its source data.

Streamlit is a modern, Python-based open-source framework that simplifies the creation of data-driven dashboards and applications. The dashboard I developed allows users to filter data by date ranges and product categories, view key metrics such as total revenue and top-performing categories, explore visualizations like revenue trends and top products, and navigate through raw data with pagination.

This guide includes a complete implementation, from setting up a Postgres database with sample data to creating Python functions for querying data, generating plots, and handling user input. This step-by-step approach demonstrates how to leverage Streamlit's capabilities to create user-friendly and dynamic dashboards, making it ideal for data engineers and scientists who want to build interactive data applications.

Although I used Postgres for my data, it should be straightforward to modify the code to use a CSV file or any other relational database management system (RDBMS), such as SQLite, as your data source.

> _That's all from me for now. I hope you found this article useful. If you did, please check out my profile page at this link. From there, you can see my other published stories and subscribe to get notified when I post new content._

If you liked this content, Medium thinks you'll find these articles interesting, too.

> **Speed up Pandas code with Numpy**

> **Introducing Deepseek Artifacts**

• • •

WRITTEN BY

Thomas Reid

See all from Thomas Reid

Topics:

Dashboard

Data Science

Python

Streamlit

Visualization

Share this article:

Related Articles

ARTIFICIAL INTELLIGENCE

Implementing Convolutional Neural Networks in TensorFlow

Step-by-step code guide to building a Convolutional Neural Network

Shreya Rao

August 20, 2024 6 min read

DATA SCIENCE

Hands-on Time Series Anomaly Detection using Autoencoders, with Python

Here's how to use Autoencoders to detect signals with anomalies in a few lines of...

Piero Paialunga

August 21, 2024 12 min read

DATA SCIENCE

Solving a Constrained Project Scheduling Problem with Quantum Annealing

Solving the resource constrained project scheduling problem (RCPSP) with D-Wave's hybrid constrained quadratic model (CQM)

Luis Fernando PÉREZ ARMAS, Ph.D.

August 20, 2024 29 min read

DATA SCIENCE

Back To Basics, Part Uno: Linear Regression and Cost Function

An illustrated guide on essential machine learning concepts

Shreya Rao

February 3, 2023 6 min read

DATA SCIENCE

Must-Know in Statistics: The Bivariate Normal Projection Explained

Derivation and practical examples of this powerful concept

Luigi Battistoni

August 14, 2024 7 min read

DATA SCIENCE

Our Columns

Columns on TDS are carefully curated collections of posts on a particular idea or category...

TDS Editors

November 14, 2020 4 min read

DATA SCIENCE

Optimizing Marketing Campaigns with Budgeted Multi-Armed Bandits

With demos, our new solution, and a video

Vadim Arzamasov

August 16, 2024 10 min read

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

Subscribe to Our Newsletter

ABOUT

·

ADVERTISE

·

PRIVACY POLICY

·

TERMS OF USE

COOKIES SETTINGS