Building A Modern Dashboard with Python and Taipy

A guide to building a front-end data application

Thomas Reid
Jun 23, 2025 11 min read

Image by AI (GPT-4o)

This is the third article in a short series on developing data dashboards using the latest Python-based development tools, Streamlit, Gradio and Taipy.

The source data set for each dashboard is the same, but stored in different formats. As much as possible, I'll try to make the actual dashboard layouts for each tool resemble each other and have the same functionality.

I've already written the Streamlit and Gradio versions. The Streamlit version gets its source data from a Postgres database. The Gradio and Taipy versions get their data from a CSV file. You can find links to those other articles at the end of this one.

What is Taipy?

Taipy is a relatively new Python-based web framework that became prominent a couple of years ago. According to its website, Taipy is …

> "… ***an open-source Python library for building production-ready front-end & back-end in no time. No knowledge of web development is required!***"

The target audience for Taipy is data scientists, machine learning practitioners and data engineers who may not have extensive experience developing front-end applications, but are typically fluent in Python. Taipy makes it reasonably easy to create front-ends using Python, so that's a win-win.

You can get started using Taipy for free. If you need to use it as part of an enterprise, with dedicated support and scalability, paid plans are available on a monthly or yearly basis. Their website provides more information, which I'll link to at the end of this article.

Why use Taipy over Gradio or Streamlit?

As I've shown in this and the other two articles, you can develop very similar output using all three frameworks, so it begs the question of why use one over the other.

While Gradio excels at quickly creating ML demos and Streamlit is brilliant for interactive data exploration, they both operate on a principle of simplicity that can become a limitation as your application's ambition grows. Taipy enters the picture when your project needs to graduate from a simple script or demo into a robust, performant, and maintainable application.

You should strongly consider choosing Taipy over Streamlit/Gradio if,

- Your app's performance is critical
- Your single script file is becoming unmanageably long and complex.
- You need to build multi-page applications with complex navigation.
- Your application requires "what-if" scenario analysis or complex pipeline execution.
- You are building a production tool for business users, not just an internal exploratory dashboard.
- You are working in a team and need a clean, maintainable codebase.

In short, choose **Gradio** for demos. Choose **Streamlit** for interactive exploration. Choose **Taipy** when you're ready to build high-performance, scalable, and production-grade enterprise data applications.

What we'll develop

We're developing a data dashboard. Our source data will be a single CSV file containing 100,000 synthetic sales records.

The actual source of the data isn't ***that*** important. It could just as easily be stored as a Parquet file, in SQLite or Postgres, or any database you can connect to.

This is what our final dashboard will look like.

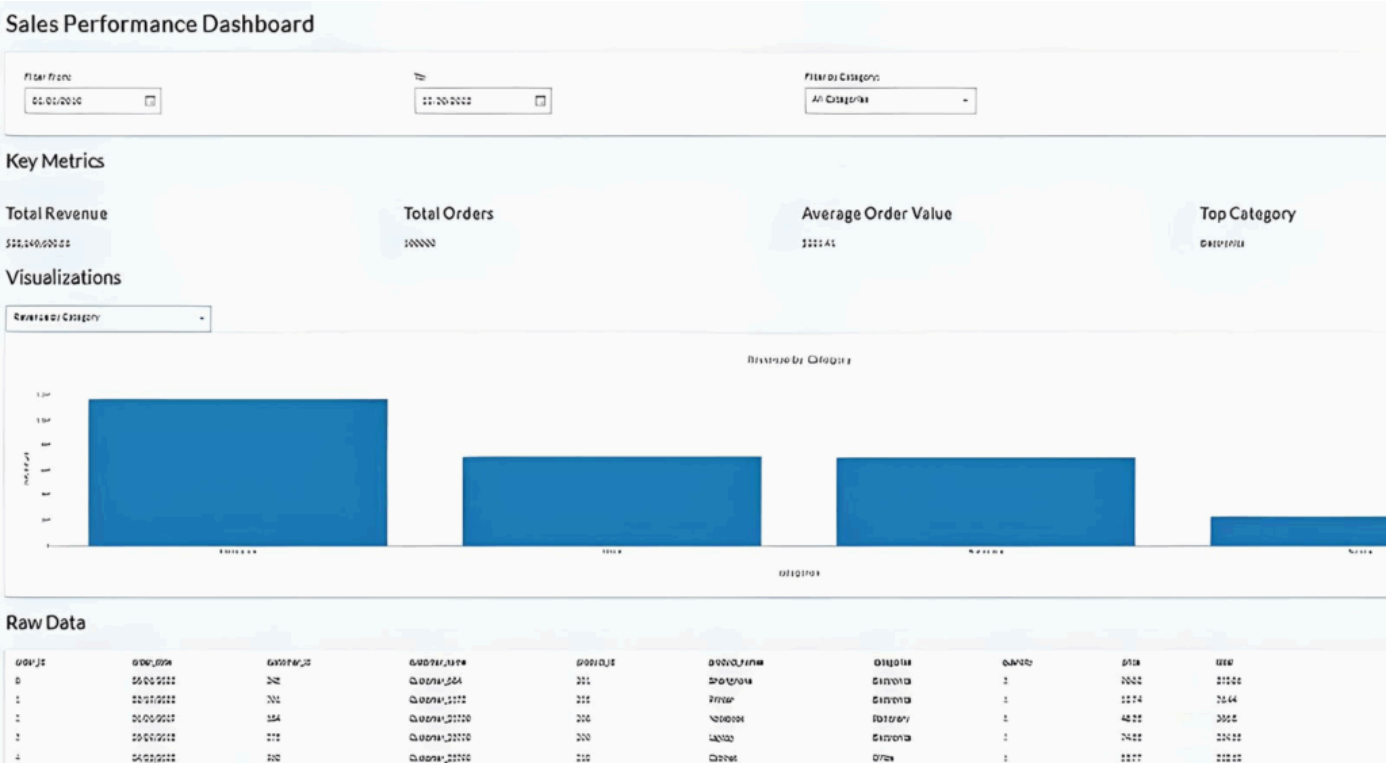

Image by Author

There are four main sections.

- The top row enables the user to select specific start and end dates and/or product categories using date pickers and a drop-down list, respectively.
- The second row, **“Key Metrics,**“ provides a top-level summary of the selected data.
- The **Visualisations** section allows the user to select one of three graphs to display the input dataset.
- The **Raw Data** section is exactly what it claims to be. This tabular representation of the chosen data effectively views the underlying CSV data file.

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 update to reflect the user's choices.

The source data

As mentioned, the dashboard's source data is contained in a single comma-separated values (CSV) file. The data consists of 100,000 synthetic sales-related records. Here are the first ten records of the file.

order_id	order_date	customer_id	customer_name	product_id	product
0	01/08/2022	245	Customer_884	201	Smartph
1	19/02/2022	701	Customer_1672	205	Printer
2	01/01/2017	184	Customer_21720	208	Noteboo
3	09/03/2013	275	Customer_23770	200	Laptop
4	23/04/2022	960	Customer_23790	210	Cabinet
5	10/07/2019	197	Customer_25587	202	Desk
6	12/11/2014	510	Customer_6912	204	Monitor
7	12/07/2016	150	Customer_17761	200	Laptop
8	12/11/2016	997	Customer_23801	209	Coffee
9	23/01/2017	151	Customer_30325	207	Pen

And here is some Python code you can use to generate a dataset. It utilises the NumPy and Pandas Python libraries, so ensure that both are installed before running the code.

```
# generate the 100000 record CSV file
#
import polars as pl
import numpy as np
from datetime import datetime, timedelta
```

```
    def generate(nrows: int, filename: str):
        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(len(names), size=nrows)
        quantity = np.random.randint(1, 11, size=nrows)
        price = np.random.randint(199, 10000, size=nrows) / 100
        # Generate random dates between 2010-01-01 and 2023-12-31
        start_date = datetime(2010, 1, 1)
        end_date = datetime(2023, 12, 31)
```

```
    date_range = (end_date - start_date).days
    # Create random dates as np.array and convert to string format
    order_dates = np.array([(start_date + timedelta(days=np.random.randint(0
    # Define columns
    columns = {
        "order_id": np.arange(nrows),
        "order_date": order_dates,
        "customer_id": np.random.randint(100, 1000, size=nrows),
        "customer_name": [f"Customer_{i}" for i in np.random.randint(2**15,
        "product_id": product_id + 200,
        "product_names": names[product_id],
        "categories": categories[product_id],
        "quantity": quantity,
        "price": price,
        "total": price * quantity,
    }
    # Create Polars DataFrame and write to CSV with explicit delimiter
    df = pl.DataFrame(columns)
    df.write_csv(filename, separator=',',include_header=True)  # Ensure comm
# Generate 100,000 rows of data with random order_date and save to CSV
generate(100_000, "/mnt/d/sales_data/sales_data.csv")
```

Installing and using Taipy

Installing Taipy is easy, but before coding, it's best practice to set up a separate Python environment for all your work. I use Miniconda for this purpose, but feel free to use whatever method suits your workflow.

If you want to follow the Miniconda route and don't already have it, you must first install Miniconda.

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

```
#create our test environment
(base) C:\Users\thoma>conda create -n taipy_dashboard python=3.12 -y
```

```
# Now activate it
(base) C:\Users\thoma>conda activate taipy_dashboard

# Install python libraries, etc ...
(taipy_dashboard) C:\Users\thoma>pip install taipy pandas
```

The Code

I'll break down the code into sections and explain each one as we proceed.

Section 1

```
from taipy.gui import Gui
import pandas as pd
import datetime

# Load CSV data
csv_file_path = r"d:\sales_data\sales_data.csv"

try:
    raw_data = pd.read_csv(
        csv_file_path,
        parse_dates=["order_date"],
        dayfirst=True,
        low_memory=False  # Suppress dtype warning
    )
    if "revenue" not in raw_data.columns:
        raw_data["revenue"] = raw_data["quantity"] * raw_data["price"]
    print(f"Data loaded successfully: {raw_data.shape[0]} rows")
except Exception as e:
    print(f"Error loading CSV: {e}")
    raw_data = pd.DataFrame()

categories = ["All Categories"] + raw_data["categories"].dropna().unique().t
```

```
# Define the visualization options as a proper list
chart_options = ["Revenue Over Time", "Revenue by Category", "Top Products"]
```

This script prepares sales data for use in our Taipy visualisation app. It does the following,

1. Imports the required external libraries and loads and preprocesses our source data from the input CSV.
2. Calculates derived metrics like revenue.
3. Extracts relevant filtering options (categories).
4. Defines available visualisation options.

Section 2

```
start_date = raw_data["order_date"].min().date() if not raw_data.empty else
end_date = raw_data["order_date"].max().date() if not raw_data.empty else da
selected_category = "All Categories"
selected_tab = "Revenue Over Time"  # Set default selected tab
total_revenue = "$0.00"
total_orders = 0
avg_order_value = "$0.00"
top_category = "N/A"
revenue_data = pd.DataFrame(columns=["order_date", "revenue"])
category_data = pd.DataFrame(columns=["categories", "revenue"])
top_products_data = pd.DataFrame(columns=["product_names", "revenue"])

def apply_changes(state):
    filtered_data = raw_data[
        (raw_data["order_date"] >= pd.to_datetime(state.start_date)) &
        (raw_data["order_date"] <= pd.to_datetime(state.end_date))
    ]
    if state.selected_category != "All Categories":
        filtered_data = filtered_data[filtered_data["categories"] == state.s

    state.revenue_data = filtered_data.groupby("order_date")["revenue"].sum(
```

```
    state.revenue_data.columns = ["order_date", "revenue"]
    print("Revenue Data:")
    print(state.revenue_data.head())

    state.category_data = filtered_data.groupby("categories")["revenue"].sum
    state.category_data.columns = ["categories", "revenue"]
    print("Category Data:")
    print(state.category_data.head())

    state.top_products_data = (
        filtered_data.groupby("product_names")["revenue"]
        .sum()
        .sort_values(ascending=False)
        .head(10)
        .reset_index()
    )
    state.top_products_data.columns = ["product_names", "revenue"]
    print("Top Products Data:")
    print(state.top_products_data.head())

    state.raw_data = filtered_data
    state.total_revenue = f"${filtered_data['revenue'].sum():,.2f}"
    state.total_orders = filtered_data["order_id"].nunique()
    state.avg_order_value = f"${filtered_data['revenue'].sum() / max(filtere
    state.top_category = (
        filtered_data.groupby("categories")["revenue"].sum().idxmax()
        if not filtered_data.empty else "N/A"
    )

def on_change(state, var_name, var_value):
    if var_name in {"start_date", "end_date", "selected_category", "selected
        print(f"State change detected: {var_name} = {var_value}")  # Debuggi
        apply_changes(state)

def on_init(state):
    apply_changes(state)

import taipy.gui.builder as tgb
```

```
def get_partial_visibility(tab_name, selected_tab):
    return "block" if tab_name == selected_tab else "none"
```

Sets the default start and end dates and initial category. Also, the initial chart will be displayed as **Revenue Over Time**. Placeholder and initial values are also set for the following:-

- **total_revenue**. Set to `"$0.00"`.
- **total_orders.** Set to `0`.
- **avg_order_value.** Set to `"$0.00"`.
- **top_category.** Set to `"N/A"`.

Empty DataFrames are set for:-

- **revenue_data.** Columns are `["order_date", "revenue"]`.
- **category_data.** Columns are `["categories", "revenue"]`.
- **top_products_data.** Columns are `["product_names", "revenue"]`.

The **apply_changes** function is defined. This function is triggered to update the state when filters (such as date range or category) are applied. It updates the following:-

- Time-series revenue trends.
- Revenue distribution across categories.
- The top 10 products by revenue.
- Summary metrics (total revenue, total orders, average order value, top category).

The **on_change** function fires whenever any of the user-selectable components is changed

The **on_init** function fires when the app is first run.

The **get_partial_visibility** function determines the CSS `display` property for UI elements based on the selected tab.

Section 3

```
with tgb.Page() as page:
    tgb.text("# Sales Performance Dashboard", mode="md")

    # Filters section
    with tgb.part(class_name="card"):
        with tgb.layout(columns="1 1 2"):  # Arrange elements in 3 columns
            with tgb.part():
                tgb.text("Filter From:")
                tgb.date("{start_date}")
            with tgb.part():
                tgb.text("To:")
                tgb.date("{end_date}")
            with tgb.part():
                tgb.text("Filter by Category:")
                tgb.selector(
                    value="{selected_category}",
                    lov=categories,
                    dropdown=True,
                    width="300px"
                )

    # Metrics section
    tgb.text("## Key Metrics", mode="md")
    with tgb.layout(columns="1 1 1 1"):
        with tgb.part(class_name="metric-card"):
            tgb.text("### Total Revenue", mode="md")
            tgb.text("{total_revenue}")
        with tgb.part(class_name="metric-card"):
```

```
            tgb.text("### Total Orders", mode="md")
            tgb.text("{total_orders}")
        with tgb.part(class_name="metric-card"):
            tgb.text("### Average Order Value", mode="md")
            tgb.text("{avg_order_value}")
        with tgb.part(class_name="metric-card"):
            tgb.text("### Top Category", mode="md")
            tgb.text("{top_category}")

    tgb.text("## Visualizations", mode="md")
    # Selector for visualizations with reduced width
    with tgb.part(style="width: 50%;"):  # Reduce width of the dropdown
        tgb.selector(
            value="{selected_tab}",
            lov=["Revenue Over Time", "Revenue by Category", "Top Products"]
            dropdown=True,
            width="360px",  # Reduce width of the dropdown
        )

    # Conditional rendering of charts based on selected_tab
    with tgb.part(render="{selected_tab == 'Revenue Over Time'}"):
        tgb.chart(
            data="{revenue_data}",
            x="order_date",
            y="revenue",
            type="line",
            title="Revenue Over Time",
        )

    with tgb.part(render="{selected_tab == 'Revenue by Category'}"):
        tgb.chart(
            data="{category_data}",
            x="categories",
            y="revenue",
            type="bar",
            title="Revenue by Category",
        )

    with tgb.part(render="{selected_tab == 'Top Products'}"):
        tgb.chart(
```

```
            data="{top_products_data}",
            x="product_names",
            y="revenue",
            type="bar",
            title="Top Products",
        )

    # Raw Data Table
    tgb.text("## Raw Data", mode="md")
    tgb.table(data="{raw_data}")
```

This section of code defines the look and behaviour of the overall page and is split up into several sub-sections

Page Definition

tgp.page(). Represents the dashboard's main container, defining the page's structure and elements.

Dashboard Layout

- Displays the title: **"Sales Performance Dashboard"** in Markdown mode (`mode="md"`).

Filters Section

- Placed within a **card-style part** that uses a 3-column layout – `tgb.layout(columns="1 1 2")` —to arrange the filters.

Filter Elements

1. **Start Date.** A date picker `tgb.date("{start_date}")` for selecting the start of the date range.

2. **End Date.** A date picker `tgb.date("{end_date}")` for choosing the end of the date range.
3. **Category Filter.**

- A dropdown selector `tgb.selector` to filter data by categories.
- Populated using `categories` e.g., `"All Categories"` and available categories from the dataset.

Key Metrics Section

Displays summary statistics in four **metric cards** arranged in a 4-column layout:

- **Total Revenue.** Shows the `total_revenue` value.
- **Total Orders.** Displays the number of unique orders (`total_orders`).
- **Average Order Value.** Shows the `avg_order_value` .
- **Top Category.** Displays the name of the category contributing the most revenue.

Visualizations Section

- A drop-down selector allows users to switch between different visualisations (e.g., “Revenue Over Time,” “Revenue by Category,” “Top Products”).
- The dropdown width is reduced for a compact UI.

Conditional Rendering of Charts

- **Revenue over time.** Displays the line chart `revenue_data` showing revenue trends over time.

- **Revenue by category.** Shows the bar chart `category_data` to visualise revenue distribution across categories.
- **Top products.** Displays the bar chart `top_products_data` showing the top 10 products by revenue.

Raw Data Table

- Displays the raw dataset in a tabular format.
- Dynamically updates based on user-applied filters (e.g., date range, category).

Section 4

```
Gui(page).run(
    title="Sales Dashboard",
    dark_mode=False,
    debug=True,
    port="auto",
    allow_unsafe_werkzeug=True,
    async_mode="threading"
)
```

This final, short section renders the page for display on a browser.

Running the code

Collect all the above code snippets and save them to a file, e.g taipy-app.py. Make sure your source data file is in the correct location and referenced correctly in your code. You then run the module just like any other Python code by typing this into a command-line terminal.

```
python taipy-app.py
```

After a second or two, you should see a browser window open with your data app displayed.

Summary

In this article, I've attempted to provide a comprehensive guide to building an interactive sales performance dashboard with Taipy using a CSV file as its source data.

I explained that Taipy is a modern, Python-based open-source framework that simplifies the creation of data-driven dashboards and applications. I also provided some suggestions on why you might want to use TaiPy over the other two popular frameworks, Gradio and Streamlit.

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 visualisations like revenue trends and top products, and navigate through raw data with pagination.

This guide provides a comprehensive implementation, covering the entire process from creating sample data to developing Python functions for querying data, generating plots, and handling user input. This step-by-step approach demonstrates how to leverage Taipy'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 a CSV file for my data source, modifying the code to use another data source, such as a relational database management system (RDBMS) like SQLite, should be straightforward.

For more information on Taipy, their website is https://taipy.io/

To view my other two TDS articles on building data dashboards using Gradio and Streamlit, click the links below.

Gradio dashboard

Streamlit dashboard