# SAMSUNG INNOVATION CAMPUS

## CODING AND PROGRAMMING

---

**Project Title:**

## "RETAIL STORE SALES TRENDS ANALYSIS USING TIME-SERIES DATA"

**Description:**

- The document outlines a **Retail Store Sales Trend Analysis using Time-Series Data** with Pandas.
- It defines the **objective** as analysing daily sales records to identify trends, seasonality, and performance.
- The project is broken into **step-by-step tasks**: loading data, cleaning, time-series manipulation, filtering, grouping, and aggregation.
- It includes creating **derived columns** like cumulative sales and sales categories (High, Medium, Low).
- Finally, results are **exported** into cleaned datasets and summary CSV files for store-wise and weekday-wise sales.

**Member Details:**



| | |
|---|---|
| Name: | Swasti M Petkar |
| USN: | 4GW23CI055 |
| Department: | CSE(AI&ML) |
| Year: | Pre-final Year |
| Submission date: | 02/09/2025 |
| College: | GSSS Institute of Engineering and Technology for Women, Mysuru |

## Index:

## Detailed Explanation on Tech-Stack And their Features :

**1. Flask (Python Web Framework)**

- **Why it is  Used:**
  Flask is a lightweight Python web framework that lets us  quickly build web applications with routing, templates, and file handling.
- **Features in your project:**
  - Handles **routes** like / (home page), /upload (CSV upload & analysis), and /download/<filename> (download processed files).
  - Provides integration with **Jinja2 templating engine** to pass processed data into HTML templates (dashboard.html).
  - Easy to scale with extensions (authentication, database integration if needed later).

---

**2. Pandas (Python Data Analysis Library)**

- **Why it is Used:**
  Pandas is perfect for **time-series and tabular data analysis** like our retail sales dataset.
- **Features in my  project:**
  - **Data Loading & Cleaning**
    - Reads CSV file.
    - Handles missing values.
    - Removes duplicates.
    - Converts dates to datetime.
  - **Time-Series Manipulation**
    - Sets Date as index.
    - Extracts **Weekday** and **Month** columns for grouping.
  - **Filtering & Conditions**
    - Get data for specific StoreID.
    - Filter sales above thresholds.

- **Aggregations & Grouping**
  - Store-wise total sales.
  - Month-wise average daily sales.
  - Weekday average sales.
- **Derived Columns**
  - CumulativeSales
  - SalesCategory
- **Exports CSVs** for cleaned data and summaries (to_csv).

---

**3. HTML + CSS + Bootstrap (Frontend UI)**

- **Why Used:**
  To provide a **clean, responsive, and professional dashboard** UI without writing a lot of CSS from scratch.
- **Features in my project:**
  - **index.html**
    - Upload form (CSV file upload).
    - Modern card-based layout.
    - Bootstrap styling (btn, card, form-control).
  - **dashboard.html**
    - Tables showing **store totals**, **weekday averages**, **top 3 stores**.
    - Responsive layout with Bootstrap grid system.
    - Buttons to **download processed CSVs** (cleaned data, summaries).

---

**4. Chart.js (JavaScript Charting Library)**

- **Why Used:**
  For **interactive and modern data visualizations** on the frontend.
- **Features in my project:**

- **Bar Chart** → Store-wise total sales.
- **Line Chart** → Average sales by weekday.
- **Pie Chart** → Store contribution to total sales.
- Fully integrated with Flask/Jinja → Data dynamically passed from backend (tables dictionary).

---

**5. Jinja2 Templating (Flask's Template Engine)**

- **Why Used:**
  Allows embedding Python data directly into HTML.
- **Features in my project:**
  - Loops ({% for row in tables.store_totals %}) to dynamically generate table rows and chart labels.
  - Makes dashboard **dynamic** based on uploaded file.

---

**6. File Handling (OS + Flask Send_file)**

- **Why Used:**
  To let users upload their own CSV, process it, and download results.
- **Features in my project:**
  - uploads/ → Raw uploaded CSV files.
  - processed/ → Cleaned & summary CSVs.
  - /download/<filename> → Route to download results.

---

**End-to-End Flow (Features Together)**

1. User uploads a CSV file via index.html.
2. Flask (/upload) saves the file → Pandas reads it.
3. Data is cleaned, manipulated, and analysed with Pandas.
4. Results (summaries, derived columns) are saved as CSVs in processed/.
5. Flask renders dashboard.html:

- Tables show results.
- Charts visualize insights (store totals, weekday patterns, contributions).
- Buttons let users download processed data.

6. Everything runs in a clean UI with **Bootstrap styling + Chart.js interactivity**.

## Procedure:

Step 1: Start the Application

1. Launch Flask backend server (app.py).
2. Open the web page (frontend) at http://127.0.0.1:5000.

---

Step 2: Upload Dataset

3. User selects a CSV file (retail_sales.csv) from local system using the upload form on the frontend.
4. Frontend sends the file to the Flask backend using an HTTP POST request to /analyze.

---

Step 3: Save and Process Data

5. Flask receives the file and saves it in the backend/ folder.
6. Flask calls analyze_sales(filepath) function from analysis.py.
7. Inside analyze_sales():
    - Load CSV into Pandas DataFrame.
    - Clean the data:
        - Fill missing Sales values with 0.
        - Remove duplicates.
        - Convert Date column to datetime.
    - Enhance the data:
        - Set Date as index.
        - Add new columns:

        - Weekday → Day name (Monday, Tuesday, etc.).
        - Month → Extract month number.
    - Perform Analysis:
        - Group by StoreID → Calculate total sales per store.
        - Group by Weekday → Calculate average sales per weekday.
    - Add Derived Columns:
        - CumulativeSales → Running total per store.
        - SalesCategory → Categorize sales as High, Medium, or Low.
    - Export Results:
        - cleaned_retail_sales.csv
        - store_sales_summary.csv
        - weekday_sales_summary.csv

---

Step 4: Send Response to Frontend

8. After processing:
    - Flask returns a JSON response with download links for all three output CSV files.

---

Step 5: Display Results

9. Frontend (JavaScript):
    - Reads the JSON response.
    - Dynamically shows Download Buttons for:
        - Cleaned Data
        - Store-wise Sales Summary
        - Weekday Sales Summary

---

Step 6: User Downloads Results

10. User clicks on buttons → Flask sends the requested CSV file for download.

---

## Algorithm in Pseudocode

START

Display upload form on frontend

WAIT for user to upload CSV

ON file upload:

SEND file to Flask /analyze route

Flask:

Save file

Call analyze_sales(file_path)

Read CSV

Clean and transform data

Perform analysis (aggregations)

Add new columns (Weekday, Month, CumulativeSales, Category)

Save results to CSV files

RETURN JSON with file download links

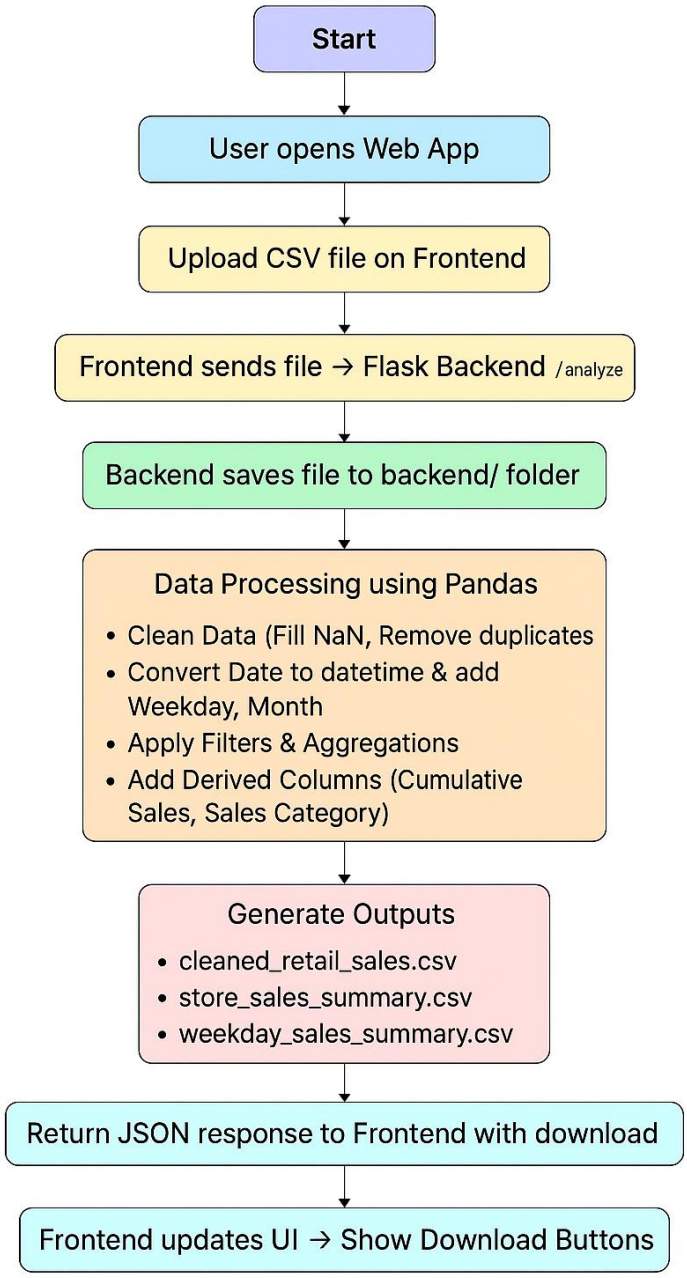
Frontend:

Show download buttons

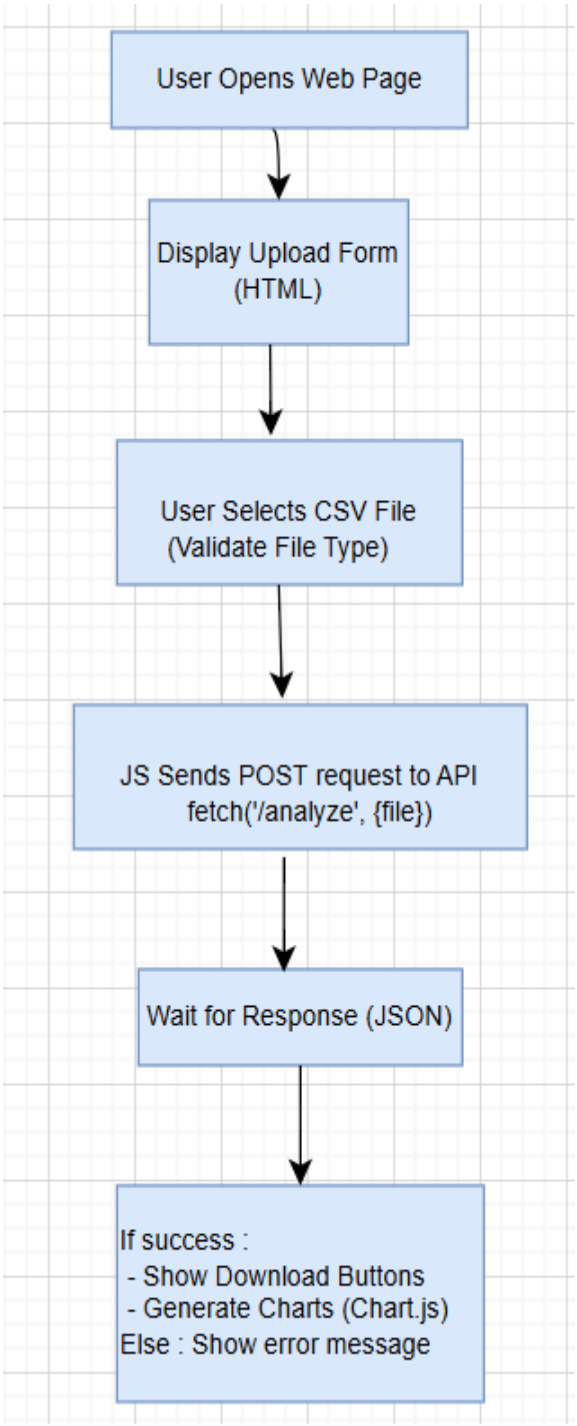
(Optional: Render visual charts)

END

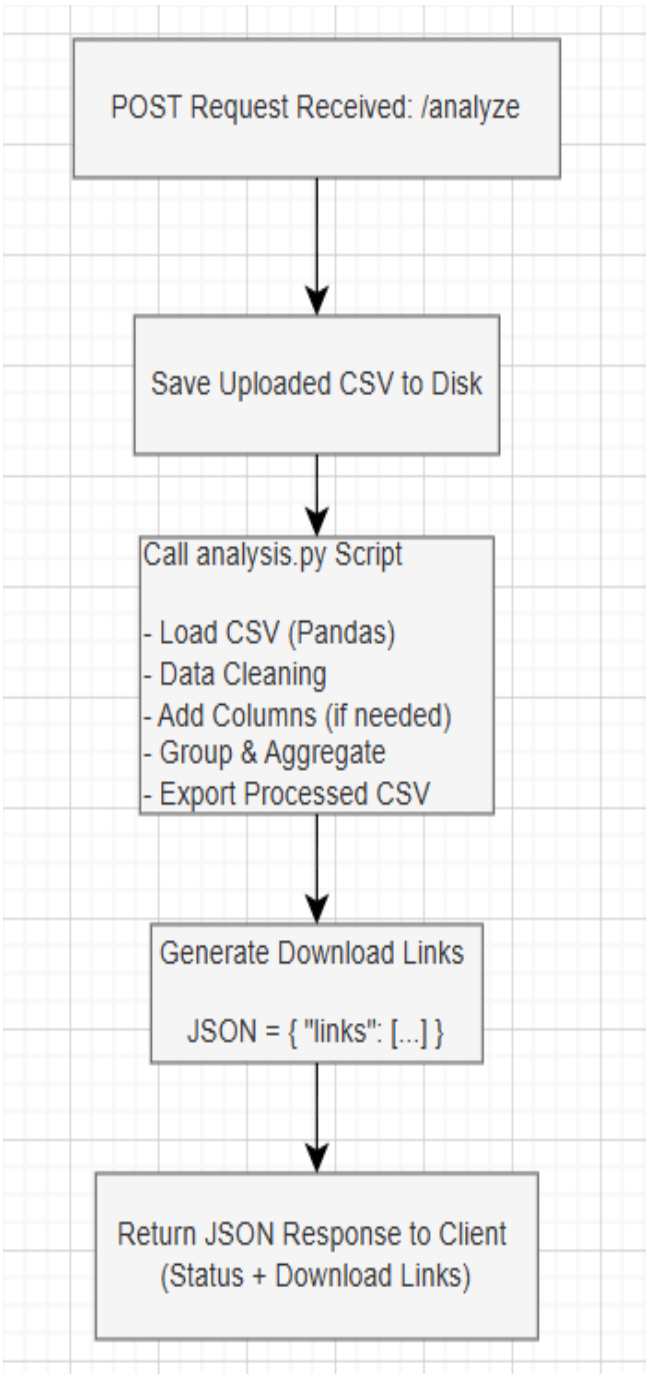
## Flowchart of the working of entire project :



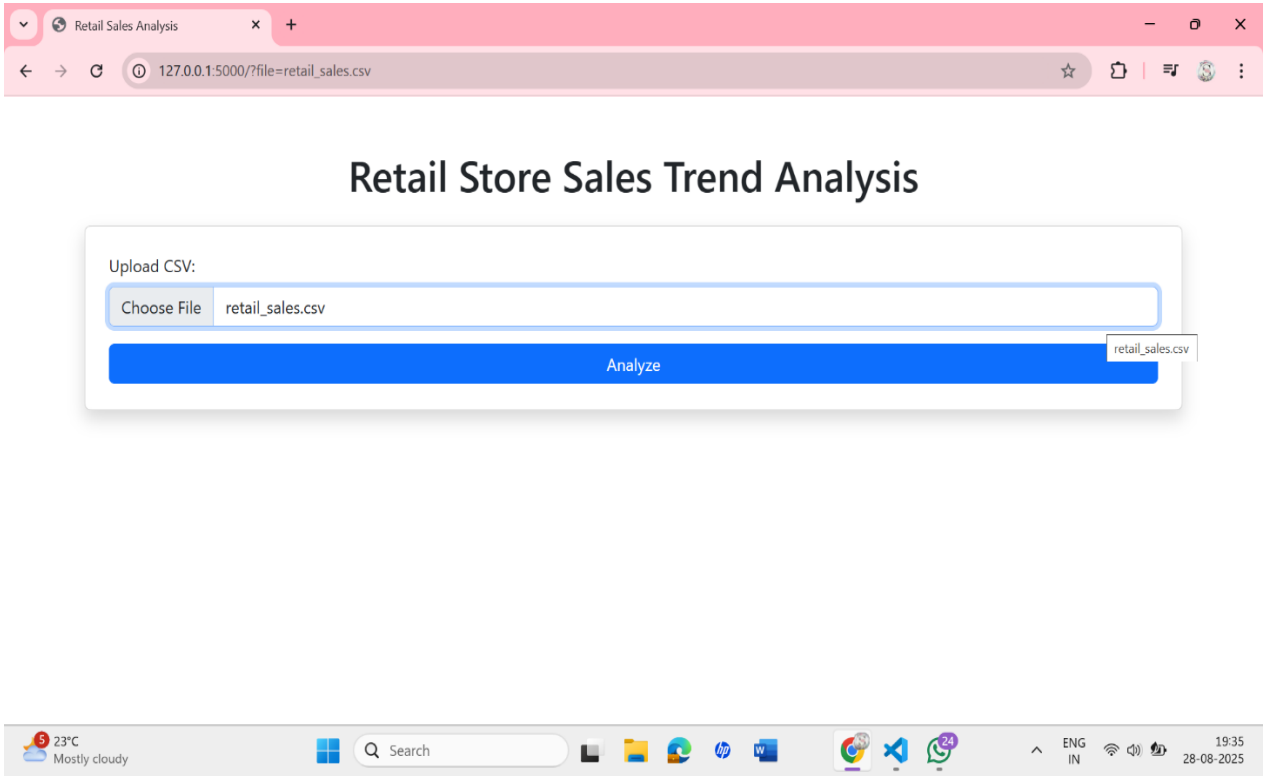
## Flowchart of Frontend:




## Flowchart of Backend:

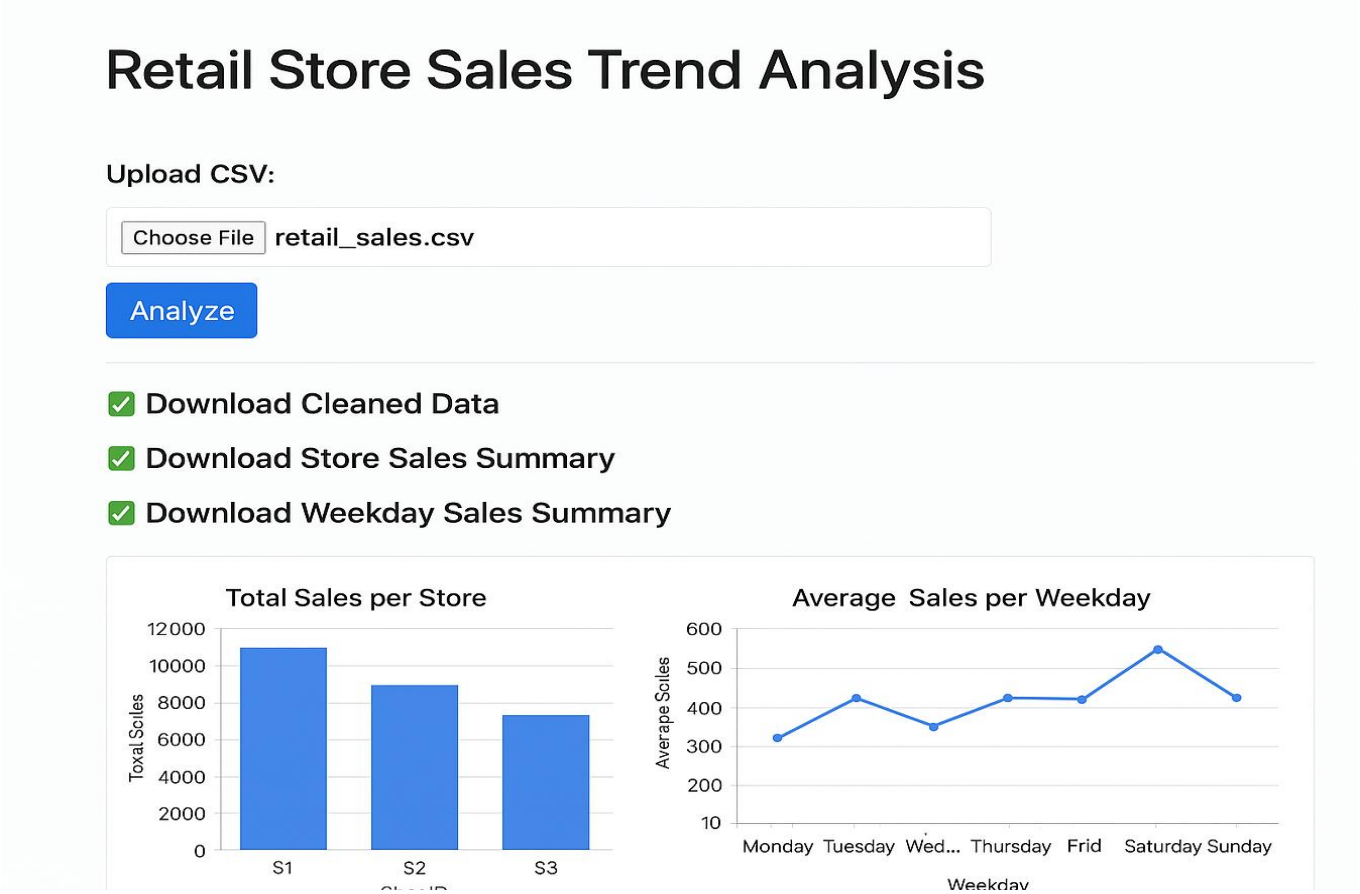


## Interface:




This interface is a **Flask web app** for **Retail Store Sales Trend Analysis**. It features a file upload section where users can select a **CSV file** (e.g., retail_sales.csv) for analysis. A prominent **"Analyze" button** submits the file to the backend for processing. The design is clean and minimal, with a title header and responsive input styling.

## Screenshots of output with explanation:




This output shows the **results of sales analysis** after uploading retail_sales.csv. It provides three download options: **Cleaned Data**, **Store Sales Summary**, and **Weekday Sales Summary**. Below, two charts display insights: **Total Sales per Store** (S1 highest, S3 lowest) and **Average Sales per Weekday** (Saturday has the peak sales). The interface combines file upload, data processing, and visualization effectively.

## Code Explanation:

### app.py

```
backend > app.py > home
1  from flask import Flask, request, jsonify, send_file, render_template
2  import os
3  from analysis import analyze_sales
4
5  app = Flask(__name__, static_folder="../frontend", template_folder="../frontend")
6
7  UPLOAD_FOLDER = 'backend/'
8  os.makedirs(UPLOAD_FOLDER, exist_ok=True)
9
10 @app.route('/')
11 def home():
12     return render_template('index.html')
13
14 @app.route('/analyze', methods=['POST'])
15 def analyze():
16     file = request.files['file']
17     filepath = os.path.join(UPLOAD_FOLDER, file.filename)
18     file.save(filepath)
19
20     # Call analysis function
21     results = analyze_sales(filepath)
22
23     return jsonify({
24         "message": "Analysis completed",
25         "cleaned_file": f"/download/{os.path.basename(results['cleaned'])}",
26         "store_summary": f"/download/{os.path.basename(results['store_summary'])}",
27         "weekday_summary": f"/download/{os.path.basename(results['weekday_summary'])}"
28     })
29
```

This is a **Flask backend application** for handling **file uploads, running data analysis, and returning results**.

**Imports (Lines 1–4)**

from flask import Flask, request, jsonify, send_file, render_template

import os

from analysis import analyze_sales

- **Flask**: Web framework used to create APIs and web apps.
    - Flask → Initializes the app.
    - request → Access incoming HTTP request data (like uploaded files).
    - jsonify → Converts Python dictionaries to JSON responses.
    - send_file → Sends files back to the client for download.
    - render_template → Renders HTML templates.
- **os**: For file handling and path operations.
- **from analysis import analyze_sales**:

- Imports a **custom function** analyze_sales() from analysis.py (probably processes the uploaded CSV and returns results).

---

**App Initialization (Line 6)**

```
app = Flask(__name__, static_folder="../frontend", template_folder="../frontend")
```

- Creates a Flask app instance.
- **static_folder**: Points to ../frontend (likely for CSS/JS files).
- **template_folder**: Also points to ../frontend (for HTML files like index.html).

---

**Upload Folder Setup (Lines 8–9)**

```
UPLOAD_FOLDER = 'backend/'
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
```

- UPLOAD_FOLDER → Directory where uploaded files will be stored.
- os.makedirs(..., exist_ok=True) → Creates the folder if it doesn't exist.

---

**Home Route (Lines 11–13)**

```
@app.route('/')
def home():
    return render_template('index.html')
```

- Defines the **root route** (/).
- When accessed, it **renders index.html** (front-end page for file upload).

---

**Analyze Route (Lines 15–29)**

```
@app.route('/analyze', methods=['POST'])
def analyze():
    file = request.files['file']
    filepath = os.path.join(UPLOAD_FOLDER, file.filename)
```

file.save(filepath)

- **Route /analyze**:
  - Accepts only **POST requests**.
  - Expects a **file upload** with the key 'file' (like from an <input type="file"> in HTML).
- file.save(filepath) → Saves the uploaded file to backend/ folder.

---

**Call Analysis Function (Line 21)**

results = analyze_sales(filepath)

- Calls **analyze_sales()** with the uploaded file path.
- Likely **cleans data, generates summaries, and returns a dictionary** with paths to processed files.

---

**Return JSON Response (Lines 23–28)**

```
return jsonify({
  "message": "Analysis completed",
  "cleaned_file": f"/download/{os.path.basename(results['cleaned'])}",
  "store_summary": f"/download/{os.path.basename(results['store_summary'])}",
  "weekday_summary": f"/download/{os.path.basename(results['weekday_summary'])}"
})
```

- Sends a **JSON response** back to the client.
- Includes:
  - "message" → Status message.
  - "cleaned_file" → Download link for cleaned data.
  - "store_summary" → Download link for store-wise summary.
  - "weekday_summary" → Download link for weekday summary.

os.path.basename() → Extracts **filename only** from the full path.

```
@app.route('/download/<filename>')
def download(filename):
    return send_file(os.path.join('backend/outputs/', filename), as_attachment=True)

if __name__ == '__main__':
    app.run(debug=True)
```

@app.route('/download/<filename>')

def download(filename):

return send_file(os.path.join('backend/outputs/', filename), as_attachment=True)

- @app.route('/download/<filename>'):

Defines a route like /download/report.csv. Whatever appears in the URL where <filename> is becomes the function argument filename. (Because you used <filename>—not <path:filename>—Flask will not accept slashes here, so only plain filenames are allowed.)

- send_file(...) : Sends a file back to the browser.
- ✓ os.path.join('backend/outputs/', filename) : builds the path to the file inside backend/outputs/.
- ✓ as_attachment = True: sets a Content-Disposition: attachment header so the browser downloads the file instead of trying to display it. The downloaded file name will be filename.

if __name__ == '__main__':

app.run(debug=True)

Runs the Flask dev server when you execute this script directly.

debug=True enables:

- auto-reload on code changes,
- the interactive debugger on errors.

## Analysis.py


### Top-level imports & folder setup

import pandas as pd

import os

- pandas is imported as pd — the main data library used here.
- os is used for filesystem operations (making directories, joining paths, etc).

OUTPUT_FOLDER = 'backend/outputs/'

os.makedirs(OUTPUT_FOLDER, exist_ok=True)

- OUTPUT_FOLDER is a string constant with the path where output files will be written.
- os.makedirs(..., exist_ok=True) creates that directory if it doesn't already exist. exist_ok=True prevents an error if the folder already exists.

---

### Function start & loading the CSV

def analyze_sales(filepath):

# Load dataset

df = pd.read_csv(filepath)

- Defines a function analyze_sales that accepts filepath (path to the uploaded CSV).
- pd.read_csv(filepath) loads the CSV into a DataFrame named df.

---

**Data cleaning**

df['Sales'].fillna(0, inplace=True)

df.drop_duplicates(inplace=True)

df['Sales'].fillna(0, inplace=True):

- Replaces NaN values in the Sales column with 0.
- inplace=True modifies df directly (no new object returned).

df.drop_duplicates(inplace=True):

- Removes duplicate rows (all columns identical).
- If you only want to dedupe based on certain columns (e.g., ['Date','StoreID']), use df.drop_duplicates(subset=[...]).

df['Date'] = pd.to_datetime(df['Date'])

df.set_index('Date', inplace=True)

df['Weekday'] = df.index.day_name()

df['Month'] = df.index.month

pd.to_datetime(df['Date']):

- Converts the Date column to datetime64[ns] dtype.

df.set_index('Date', inplace=True):

- Sets the Data Frame index to the Date column — useful for time-based grouping and indexing.
- After this, access timestamps via df.index.

df['Weekday'] = df.index.day_name():

- Creates a new column Weekday containing weekday names like 'Monday', 'Tuesday'.

df['Month'] = df.index.month:

- Adds a Month column as an integer 1–12.
- If you prefer names, use df.index.month_name().

---

**Grouping & aggregations**

store_sales = df.groupby('StoreID')['Sales'].sum()

avg_weekday_sales = df.groupby('Weekday')['Sales'].mean()

df.groupby('StoreID')['Sales'].sum():

- Groups rows by StoreID and sums the Sales column per store.
- **Result:** a Series with index StoreID and values = total sales for that store.

df.groupby('Weekday')['Sales'].mean():

- Groups by weekday name and computes the average sales for each weekday.
- **Result:** a Series indexed by weekday strings.

weekdays = ['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday']

df['Weekday'] = pd.Categorical(df['Weekday'], categories=weekdays, ordered=True)

avg_weekday_sales = df.groupby('Weekday')['Sales'].mean()

---

**Derived columns**

df['CumulativeSales'] = df.groupby('StoreID')['Sales'].cumsum()

df['SalesCategory'] = df['Sales'].apply(lambda x: 'High' if x>=5000 else 'Medium' if x>=3000 else 'Low')

df.groupby('StoreID')['Sales'].cumsum():

- Calculates a running cumulative sum of Sales **within each StoreID group**.
- This returns a Series aligned with df rows; assigning it to df['CumulativeSales'] gives each row the cumulative total up to that row's date for that store.

df['SalesCategory'] = df['Sales'].apply(lambda x: 'High' if x>=5000 else 'Medium' if x>=3000 else 'Low'):

- Creates a categorical label per-row based on sales thresholds:
  - >= 5000 → 'High'
  - >= 3000 → 'Medium' (this is only reached if <5000 but >=3000)
  - else → 'Low'.

import numpy as np

conditions = [df['Sales'] >= 5000, df['Sales'] >= 3000]

choices = ['High','Medium']

df['SalesCategory'] = np.select(conditions, choices, default='Low')

---

```
26
27      # Export results
28      cleaned_path = os.path.join(OUTPUT_FOLDER, 'cleaned_retail_sales.csv')
29      store_sales_path = os.path.join(OUTPUT_FOLDER, 'store_sales_summary.csv')
30      weekday_sales_path = os.path.join(OUTPUT_FOLDER, 'weekday_sales_summary.csv')
31
32      df.to_csv(cleaned_path)
33      store_sales.to_csv(store_sales_path)
34      avg_weekday_sales.to_csv(weekday_sales_path)
35
36      return {
37          "cleaned": cleaned_path,
38          "store_summary": store_sales_path,
39          "weekday_summary": weekday_sales_path
40      }
41
```

cleaned_path = os.path.join(OUTPUT_FOLDER, 'cleaned_retail_sales.csv')

- Builds a file path string by joining the OUTPUT_FOLDER value with the filename 'cleaned_retail_sales.csv'.
- os.path.join ensures platform-correct separators (e.g. \ on Windows, / on Unix).
- cleaned_path will be something like 'backend/outputs/cleaned_retail_sales.csv' (relative path).

store_sales_path = os.path.join(OUTPUT_FOLDER, 'store_sales_summary.csv')

- Same as above but the target file is the store-level summary CSV.

weekday_sales_path = os.path.join(OUTPUT_FOLDER, 'weekday_sales_summary.csv')

- Same for the weekday summary CSV.

df.to_csv(cleaned_path)

- Writes the DataFrame df to CSV at cleaned_path.

store_sales.to_csv(store_sales_path)

- store_sales was created earlier with df.groupby('StoreID')['Sales'].sum() — that returns a Series indexed by StoreID.
- Series.to_csv(...) will produce a two-column CSV: the index (StoreID) and the values (sum of sales). The value column will be unnamed unless you rename it or convert to a DataFrame.

avg_weekday_sales.to_csv(weekday_sales_path)

- Similar to the store summary: avg_weekday_sales is a Series (weekday → mean sales).
- That CSV will have weekday names in the index column and average sales values in the value column.

**35–39 return { ... }**

- The function returns a Python dictionary mapping keys ("cleaned", "store_summary", "weekday_summary") to the respective file path variables.
- The Flask code you showed earlier uses these returned paths (it takes the basename via os.path.basename(results['cleaned'])) to generate download links, so returning the full paths here is fine — the consumer extracts the filename for download.

**1. index.html (Main UI Layout)**

**Key Lines:**

<input type="file" id="csvFile">

<button id="analyzeBtn">Analyze</button>

**Explanation:**

- <input> allows the user to upload the CSV file.
- <button> triggers the analysis process via JavaScript.
-  serves as the container for displaying charts and output summaries.

**2. style.css (Styling the Page)**

**Key Lines:**

```
body { font-family: Arial, sans-serif; text-align: center; }

button { background-color: #007bff; color: white; padding: 10px; border: none; cursor: pointer; }
```

**Explanation:**

- Sets a clean and readable font with cantered content.
- Styles the button with blue background and white text for better UI/UX.

---

**3. script.js (Frontend Logic)**

**Key Lines:**

```
document.getElementById('analyzeBtn').addEventListener('click', () => {
  const file = document.getElementById('csvFile').files[0];
  let formData = new FormData();
  formData.append('file', file);
  fetch('/analyze', { method: 'POST', body: formData })
    .then(res => res.json())
    .then(data => console.log(data));
});
```

**Explanation:**

- Adds a click event listener to the Analyse button.
- Collects the uploaded CSV file and sends it to the Flask backend using fetch().
- Displays the response (like download links and summaries) on the frontend.

## Practical Use Cases of Retail Sales Analysis Application

1. Upload Retail Sales Dataset:

A store manager at a retail chain wants to analyse sales performance for the past year. They export sales data from their POS system (as a CSV) and upload it to this application.

2. Data Preprocessing:

The uploaded data might have:

Missing sales values for some dates (e.g., store closed).

Incorrect date formats.

Duplicate rows.

The backend cleans this automatically, just like a data analyst would do before analysis.

3. Trend Analysis & Visualization:

The system generates:

A line chart showing daily sales trends.

A bar chart showing monthly totals.

The manager uses this to see which months performed best and identify seasonal trends (e.g., higher sales during Diwali or Christmas).

4. Forecast Future Sales:

The manager wants to prepare inventory for the next month.

The system predicts that sales will increase by 20% next month (due to festive season).

The manager orders more stock in advance.

5. Generate Downloadable Reports:

After analysis:

The manager downloads a summary CSV and charts.

Shares them in a business meeting with the regional head to explain sales performance and forecast.

6. Interactive Dashboard (Frontend) :

A regional manager wants quick insights without digging into raw data:

They log in, select date range = last 6 months.

See charts for store-wise performance and weekday sales patterns.

Decide which stores need more marketing campaigns.

7. Error Handling & Notifications:

If the manager uploads a wrong file (e.g., customer data instead of sales data),

the system immediately shows:

“Invalid file format. Please upload a CSV with columns: Date, StoreID, Sales.”

This avoids confusion and wasted time.

## Bibliography:


1. Hyndman, R. J., & Athanasopoulos, G. (2018). Forecasting: Principles and Practice. OTexts. - https://otexts.com/fpp3/

2. Box, G. E., Jenkins, G. M., & Reinsel, G. C. (2015). Time Series Analysis: Forecasting and Control. Wiley.

3. Pandas Development Team. (2025). Pandas Documentation. https://pandas.pydata.org/docs/

4. Flask Documentation. (2025). Flask Web Framework. https://flask.palletsprojects.com/

5. Plotly Technologies Inc. (2025). Plotly Python Graphing Library. https://plotly.com/python/

6. Facebook Prophet. (2025). Forecasting at Scale. https://facebook.github.io/prophet/

7. Kaggle Datasets. (2025). Retail Sales Data for Time Series Analysis. https://www.kaggle.com/

8. Investopedia. (2025). Seasonality in Retail Sales. https://www.investopedia.com/