Understanding the MoMo Dashboard: A Complete Walkthrough

Thi document is written to explain how this this dashboard is created and diving deeper into the code , to the extent if steps are properly followed to can build similar or alot more improved dashboard just like we did . this document explain how all the parts of this mobile money (MoMo) dashboard work together, from processing the data to displaying it in your web browser. Imagine you're building a machine that takes raw transaction messages, organizes them, and shows you beautiful charts that's exactly what this system does.

Files and main functionality purpose

- **Data Processing Plant (*process.py and db.py*)** : Where raw messages get cleaned and organized
- **Delivery Truck (*server.py*)** : That brings the organized data to your web browser
- **Display Showroom (*HTML/CSS/JS files)*** : Where you see and interact with the information

Step 1 **: <u>Processing the Raw Data</u>**

I. *process.py* The Cleaning Machine

This is where everything begins. The file process.py works like a cleaning machine for transaction messages:

```
def main():
    init_db() # Prepares the database
    sms_data = parse_xml('modified_sms_v2.xml') # Reads the
message file
    processed = process_data(sms_data) # Cleans each message
    store_data(processed) # Saves to database
```

Real-life example: imagine you have 100 book which contain the name of student where each book contain each student's name and nickname and other information related to that exact same student.

This will be very hard to read each and every student name and again rewrite each student data as you got them from the book. By using of the programming approach we will eed to design regex which can extract data from the xml file which contain those big number of message , this is just example we gave of great number of student , againyou will need to look for pattern and design feasible regex which will be used to extract properly data from all provided book

- Storage of all book (*modified_sms_v2.xml*)
- Reads all books (*parse_xml*)
- Fixes dates to one format (*extract_date*) #any detail regardless to the students data that he/she maybe entered
- Identifies store names correctly (*extract_name*)
- Puts them in neat folders by type (*categorize_transaction*)

Data extraction : How Regular Expressions (Regex) Extract Data

Regular expressions (regex) are like smart search patterns that help find specific text in messages. In this MoMo dashboard, regex is used mainly in *process.py* to pull out important information from transaction SMS messages

1. **Extracting Amounts from Messages**

```
def extract_amount(body):
    match = re.search(r'(\d{1,3}(?:,?\d{3})*) RWF', body)
    if match:
        return int(match.group(1).replace(',', ''))
    return 0
```

What it does:

- Looks for money amounts like "1,000 RWF" or "500 RWF"
- \d{1,3} = finds 1 to 3 digits (like "500")
- (?:,?\d{3})* = optionally finds commas followed by 3 digits (like ",000")
- RWF = must end with " RWF"

Example:

- Message: "You received 1,500 RWF from John"
- Finds: "1,500 RWF"
- Returns: 1500 (as a number)

2. **Finding Names in Messages**

There are two patterns - one for companies and one for people.

Company Names:

company_pattern = r'(?:from|to|by)\s+([A-Za-z\s\.\-]+?(?:Ltd|LLC|Inc|Corp|Co)\b)'

What it looks for:

- Words after "from", "to", or "by"
- That end with business words like "Ltd", "LLC" etc.

Example:

- Message: "Payment to Green Foods Ltd"
- Finds: "Green Foods Ltd"

Personal Names:

personal_pattern = r'(?:from|to|by|transferred to|received from)\s+([A-Za-z\s\.\-]+?)'

What it looks for:

- 1-2 word names after transaction words
- Stops when it sees numbers or transaction terms

Example:

- Message: "Received from John Doe Ref123"
- Finds: "John Doe" (stops at "Ref123")

3. **Cleaning Up Names**

name = re.sub(r'[^a-zA-Z\s\.\-]', '', name)

What it does:

- Removes anything that's not:
    - Letters (a-z, A-Z)
    - Spaces (\s)
    - Periods or hyphens (.-)

Example:

- "John Doe (MTN)" → "John Doe MTN"
- Then later removes "MTN" (as it's a common word)

4. **Removing Transaction Words**

name = re.sub(r'\s(?:Has Been Completed|At|From|To|By|Ref|Txn...)\s*.*$', '', name)*

What it removes:

- Common SMS phrases like:
    - "Has Been Completed"
    - "Ref: 12345"
    - "Txn ID"
    - And many others
- Example:
    - "John Doe Ref12345" → "John Doe"
    - "Payment to MTN Airtime" → "" (empty because "MTN Airtime" is removed)

5. **Categorizing Transactions**

The code looks for keywords to decide the transaction type:

```
if "received" in body and "rwf from" in body:
    return "incoming"
elif "payment of" in body and "to" in body:
    return "payment"
```

Examples:

- "You received 500 RWF from Alice" → "incoming"
- "Payment of 200 RWF to Shop" → "payment"
- "Airtime purchase 100 RWF" → "airtime"

Why Regex is Perfect Here

Transaction SMS messages follow common patterns but with small differences. Regex helps by:

- Finding amounts regardless of formatting ("1,000" or "1000")
- Pulling names even when hidden in longer messages
- Ignoring unimportant parts of messages
- Handling variations in wording

II. *db.py* The Filing , structuring database

This file handles all database operations:

```
def init_db():
    # Creates a new organized filing system
    conn = sqlite3.connect(DB_NAME)
    c = conn.cursor()
    c.execute('CREATE TABLE IF NOT EXISTS transactions...')

def store_data(data):
    # Files each transaction in the right place
    for item in data:
        c.execute('INSERT INTO transactions...', item)
```

Key point: After running ***process.py***, you'll have a new file called ***corrected_data.db*** containing all your cleaned transactions like a perfectly organized database structuring file “db.py”.

Step 2 : The Web Server

server.py : The Bridge Between Data and Display
This Flask server does three important jobs:

1. **Serves the webpage** (like a waiter bringing you a menu)

```
@app.route('/')
def index():
    return send_from_directory('.', 'index.html')
```

2. **Provides data API** (like a kitchen preparing your order)

```
@app.route('/data')
def get_data():
    # Gets filter requests from browser
    tx_type = request.args.get('type', 'all')
    # Gets matching data from database
    transactions = conn.execute(query, params).fetchall()
    # Sends back organized response
    return jsonify(response_data)
```

3. **Handles errors** (like a manager solving problems)

```
except Exception as e:
    print(f"Error: {str(e)}")
    return jsonify({'error': str(e)}), 500
```

Real-life flow:

- You open http://127.0.0.1:5000 in your browser
- Browser asks for index.html (the menu)
- JavaScript in the page asks for /data (your food order)
- Server queries database (kitchen prepares food)
- Server sends JSON response (waiter brings food)
- JavaScript displays it beautifully (you enjoy the meal)

Key connection: Notice it links to:

```
<link rel="stylesheet" href="style.css" />
<script src="script.js"></script>
```

style.css : The Interior Designer

This makes everything look pretty with:

- Card designs
- Responsive layouts (works on phone/computer)

- Loading animations
- Table styling

Example feature:

```
.card {
  background: #3498db;
  color: white;
  padding: 20px;
  border-radius: 8px;
}
```

This makes those blue summary cards you see.

script.js : The Interactive Brain

This JavaScript file does all the magic you see:

1. **Fetching Data** (like a shop assistant getting stock)

```
async function fetchData() {
    const response = await
fetch(`/data?type=${type}&name=${name}`);
    const data = await response.json();
    updateTable(data.transactions);
    updateChart(data.chart_data);
}
```

2. **Updating Chart** (like changing display window)

```
function updateChart(chartData) {
    if (chart) chart.destroy();
    chart = new Chart(ctx, { type: 'pie', data: {...} });
}
```

3. **Handling Filters** (like sorting products by category)

```
const debouncedFetchData = debounce(fetchData, 250);
filterType.addEventListener('change', debouncedFetchData);
```

Special technique: The **debounce** function prevents too many requests when you quickly change filters - like waiting until you stop typing to search.

How Everything Works Together

Let's follow a typical user journey:

1. **User opens dashboard**
    - Browser loads index.html
    - HTML loads style.css (design) and script.js (functionality)
    - JavaScript calls /data API
2. **Server handles request**
    - Flask receives /data request

- Queries SQLite database with filters
- Returns JSON with transactions, chart data, and names

3. **Browser displays data**
    - JavaScript updates summary cards
    - Draws pie chart with Chart.js
    - Fills transaction table
    - Updates name filter options

4. **User changes filter**
    - JavaScript detects dropdown change
    - Waits 250ms (debounce) then fetches new data
    - Updates display with filtered results

Key Features Explained

The system reads SMS texts and guesses the type:

```
if "received" in body and "rwf from" in body:
    return "incoming"
elif "payment of" in body and "to" in body:
    return "payment"
```

Like sorting mail into "bills", "letters", "packages".

Why This Architecture Works Well

1. **Separation of Concerns:**
    - Processing (Python)
    - Data storage (SQLite)
    - Presentation (HTML/JS)

    Like having different departments in a company.

2. **Efficient Data Flow:**
    - Process data once (process.py)
    - Query efficiently (server.py)
    - Update only what changes (script.js)
3. **Responsive Design:**
    - CSS makes it work on phones and computers
    - Loading states prevent confusion

Troubleshooting Tips

If something isn't working:

1. **Check the order:**

- First run *python3 process.py* (prepare data)
- Then *python3 server.py* (start server)

2. Common issues:

- Forgot to install Flask? Run *pip install flask*
- Missing XML file? Need *modified_sms_v2.xml*
- Database problems? Delete *corrected_data.db* and reprocess