Profit Scan Guide

Set up & Run the Application

1. Install Python

First, ensure Python 3 is installed on your system. Streamlit requires Python 3.6 or later.

- **Windows:**
  - Download Python from python.org.
  - Run the installer and make sure to check the box to Add Python to PATH.
- **macOS:**
  - Use Homebrew by running `brew install python3` in the Terminal.
  - Python can also be installed from python.org.

2. Set Up a Virtual Environment

- Open a command prompt (Windows) or Terminal (macOS).
- Navigate to the project directory where you want to store your files.
- Run the following command to create a virtual environment:

```
python3 -m venv myenv
```

• Replace `myenv` with your desired environment name.

Activate the virtual environment:

Windows:

```
myenv\Scripts\activate
```

macOS:

```
source myenv/bin/activate
```

3. Install Required Packages

With your virtual environment activated, install the required packages using pip. The primary packages include Streamlit, pandas, NumPy, and SQLite for database interactions.

```
pip install streamlit pandas numpy openpyxl base64 io sqlite3
```

4. Prepare Your Application

- Save the provided code into a file named `app.py` (or another name of your choice) in your project directory.

5. Run the Streamlit Application

With everything set up, you can now run your Streamlit application.

- In your command prompt or Terminal, navigate to your project directory.
- Run the application with Streamlit by executing:

```
streamlit run app.py
```

- Replace `app.py` with your file name if different.
- Streamlit will start the server, and your default web browser should automatically open a new tab pointing to the local URL of the application.

Additional Notes:

- **Data Files:** The application requires CSV or Excel files for customers, invoices, products, and expenses data. Ensure these files are prepared according to the application's requirements (specific columns as mentioned in the instructions within the app).
- **Database:** The application utilizes SQLite. No additional setup is required since SQLite operates directly off the file system and interacts through Python's `sqlite3` module.

How the code works:

Database Connection and Pandas Setup

SQLite Database Connection

```
conn = sqlite3.connect("my_database.db")
c = conn.cursor()
```

- The `sqlite3.connect("my_database.db")` function establishes a connection to an SQLite database. If `my_database.db` does not exist, SQLite creates it
- The `conn.cursor()` method returns a cursor object. Cursors allow you to execute SQL commands through Python, enabling operations like creating tables, inserting data, and querying the database.

Pandas Display Options:

```
pd.set_option("display.max_rows", 500)
pd.set_option("display.max_columns", 500)
```

By setting `display.max_rows` and `display.max_columns` to 500, you ensure that up to 500 rows and columns can be shown in the output. This is particularly useful when working with large datasets and want to avoid truncation of data in your output

Constants for Database Tables and Columns

```
TABLE_CUSTOMERS = "customers"
TABLE_INVOICES = "invoices"
TABLE_PRODUCTS = "products"
TABLE_EXPENSES = "expenses"
COLUMN_CUSTOMER = "Customer"
# and so on
```

Purpose: Defines string constants for the names of database tables and key columns. Using constants instead of directly typing the string values helps prevent errors from typos and makes the code more maintainable. If we need to change a table or column name, we can do it in one place, and it updates everywhere.

Data Loading Section

`load_csv` Function

```
def load_csv(file, table_name):
    df = pd.read_csv(
        file,
        thousands=",",
        decimal=".",
        na_values=["NA", "na", "N/A", "n/a", ""],
    )
    for col in df.columns:
        try:
            df[col] = pd.to_numeric(df[col])
        except ValueError:
            pass  # column can't be converted to a number, le
ave as is

    if "Product" in df.columns:
        df["Product"] = df["Product"].str.strip()

    df.to_sql(table_name, conn, if_exists="replace", index=Fa
lse)
    return df
```

Purpose: This function reads a CSV file uploaded by the user, performs initial processing, and stores the data into the specified SQLite table.

`pd.read_csv`: Loads the CSV file into a Pandas DataFrame. It uses parameters to handle common data issues:

- `thousands=","` interprets commas as thousands separators, useful for correctly parsing numeric values formatted as strings in many locales.
- `decimal="."` specifies the character used for the decimal point.
- `na_values=["NA", "na", "N/A", "n/a", ""]` defines a list of strings to be recognized as NaN (missing values), ensuring the data is clean and consistent.

⇒ Next, it Iterates over all columns attempting to convert them to numeric types (`int` or `float`). This is crucial for ensuring that operations such as sorting, filtering, and mathematical computations work correctly later on

⇒ If a "Product" column exists, it trims whitespace from its values.

⇒ Uses `df.to_sql` to store the DataFrame in an SQLite table. The `if_exists="replace"` parameter means that if the table already exists, it will be replaced with the new data, ensuring that the database always reflects the most recent upload.

⇒ The function returns the processed DataFrame, allowing for further manipulation or inspection if needed

`load_excel` Function

```
def load_excel(file, table_name):
    df = pd.read_excel(file)

    if "Product" in df.columns:
        df["Product"] = df["Product"].str.strip()

    df.to_sql(table_name, conn, if_exists="replace", index=Fa
lse)
    return df
```

Purpose: Similar to `load_csv`, but specifically for Excel files. It reads data from an Excel file into a DataFrame, cleans it, and stores it in the specified SQLite table.

Directly reads the Excel file without needing to specify thousands separators or decimal points since Excel formats are more standardized than CSV. However, it still handles the "Product" column similarly by trimming whitespace

Both `load_csv` and `load_excel` serve as entry points for data into the app, handling initial cleaning and standardization to ensure that subsequent operations can proceed smoothly

Data Retrieval process

`load_data` Function

```
def load_data(file, table_name, required_columns):
    try:
        if file is not None and f"{table_name}_data" not in st.session_state:
            try:
                c.execute(f"DROP TABLE IF EXISTS {table_name}")
                conn.commit()
            except sqlite3.Error as e:
                st.error(f"An error occurred: {e.args[0]}")

            file_extension = file.name.split(".")[-1]
            if file_extension == "csv":
                df = load_csv(file, table_name)
            elif file_extension in ["xlsx", "xls"]:
                df = load_excel(file, table_name)

            if table_name == "expenses":
                if ("Allocate_To" in df.columns and "Allocate_By_Tran" in df.columns and "Allocate_By_Value" in df.columns):
                    st.session_state["allocation_columns_pre_filled"] = True
                else:
                    st.session_state["allocation_columns_pre_filled"] = False

            if not set(required_columns).issubset(df.columns):
                st.error(f"The following required columns are missing from the Excel file: {set(required_columns) - set(df.columns)}")
                return False
            st.session_state[f"{table_name}_data"] = df
            return True
        else:
            return False
    except Exception as e:
        st.error(f"Error when trying to read the file: {e}")
        return False
```

Purpose and Flow

This function aims to load data from a file (CSV or Excel) into a specified database table and perform initial validation against required columns.

The entire function is wrapped in a `try` block to catch and handle any exceptions that may occur during file processing.

Step By Step

It first checks if the file is not `None` (indicating that a file has been uploaded) and if the data for this table hasn't already been loaded into the session state.

Executes a SQL command to drop the table if it already exists: `c.execute(f"DROP TABLE IF EXISTS {table_name}")`. This ensures that the database always contains the latest data uploaded by the user.

Determines the file extension and calls the appropriate function (`load_csv` or `load_excel`) based on the file type.

Special Handling for Expenses Table:

- If the table being loaded is "expenses", it checks for the presence of specific columns related to expense allocation.

Verifies that the uploaded file contains all required columns.

If all checks pass, the loaded DataFrame is stored in the session state under a key named after the table

`load_data_from_db` Function

```
def load_data_from_db(table_name):
    if f"{table_name}_data" in st.session_state:
        return st.session_state[f"{table_name}_data"]
    else:
        df = pd.read_sql_query(f"SELECT * FROM {table_name}", 
conn)
        st.session_state[f"{table_name}_data"] = df
        return df
```

This function fetches data from a specified table in the SQLite database and returns it as a Pandas DataFrame. It also utilizes Streamlit's session state to cache data, reducing unnecessary database queries and improving application performance.

Step-by-Step Explanation

1. **Session State Check**:

- `if f"{table_name}_data" in st.session_state:` checks if the data for the specified table is already stored in Streamlit's session state. Streamlit's session state is a feature that allows data and variables to persist across reruns of the app, **which happens whenever the user interacts with the UI.**
- If the data is present, it's returned immediately, bypassing the need to query the database again.

2. **Database Query Execution**:

- `df = pd.read_sql_query(f"SELECT * FROM {table_name}", conn)` executes a SQL query to fetch all records (`SELECT *`) from the specified table. This command utilizes Pandas' `read_sql_query` function, which takes a SQL query and a connection object (`conn`) as arguments, returning the query results as a DataFrame.
- This step ensures that the latest data is fetched from the database, allowing for up-to-date analysis and reporting within your application.

`merge_data` Function

```
def merge_data(customers_table_name, invoices_table_name, pro
ducts_table_name):
    if "merged_data" in st.session_state:
        return st.session_state["merged_data"]
    else:
        # Get the columns from the tables
        customers_columns = pd.read_sql_query(f"PRAGMA table_
info({TABLE_CUSTOMERS})", conn)["name"].tolist()
        products_columns = pd.read_sql_query(f"PRAGMA table_i
nfo({TABLE_PRODUCTS})", conn)["name"].tolist()

        if COLUMN_CUSTOMER in customers_columns:
            customers_columns.remove(COLUMN_CUSTOMER)
        if COLUMN_PRODUCT in products_columns:
            products_columns.remove(COLUMN_PRODUCT)

        customers_columns_str = ", ".join([f'customers."{co
l}"' for col in customers_columns])
        products_columns_str = ", ".join([f'products."{col}"'
for col in products_columns])
```

```
        # Combine all column strings together with additional
commas
        all_columns_str = ", ".join(filter(None, ["invoices.
*", customers_columns_str, products_columns_str]))

        query = f"""
        SELECT
            {all_columns_str}
        FROM {invoices_table_name} AS invoices
        LEFT JOIN {customers_table_name} AS customers
        ON invoices.Customer = customers.Customer
        INNER JOIN {products_table_name} AS products
        ON invoices.Product = products.Product
        """

        df = pd.read_sql_query(query, conn)
        st.session_state["merged_data"] = df
        return df
```

The function's **primary goal** is to merge data from the customers, invoices, and products tables into a single dataset.

Step-by-Step

Session State Check

Checks if the merged dataset already exists in Streamlit's session state

Retrieve Table Columns

Queries the SQLite database for the structure (column names) of the customers and products tables using the `PRAGMA table_info` SQL command. This information is used to dynamically construct the SQL query for merging data

▶ **PRAGMA** dynamically fetches column names from the database tables, ensuring that the script automatically adapts to any changes in the table structure.

Column Filtering:

Prepares strings of column names for inclusion in the SQL query. This involves wrapping column names in quotes and prefixing them with their respective table names (e.g., `customers."Customer"`) to avoid ambiguity in the SQL query.

SQL Query Construction:

Constructs an SQL query to select all columns from the invoices table and the relevant columns from the customers and products tables. The query uses LEFT JOIN to combine customers with invoices and INNER JOIN to combine products with invoices, ensuring that all invoice records are included and enriched with customer and product details.

Execute Query and Merge Data:

Executes the constructed SQL query using `pd.read_sql_query`, which fetches and merges the data from the database into a single Pandas DataFrame. This DataFrame represents the consolidated dataset for analysis.

Cache and Return Merged Data:

Stores the merged DataFrame in Streamlit's session state under the key "merged_data" to cache the result for future use. Returns the merged DataFrame for subsequent processing

`calculate_cost` Function

```
def calculate_cost(df):
    try:
        if "Product_Cost" in df.columns:
            df["Cost_Amount"] = df["Quantity"] * df["Product_
Cost"]
        elif "Cost_%" in df.columns:
            df["Cost_Amount"] = df["Sales_Amount"] * df["Cost
_%"]
        else:
            st.warning(
                "Neither 'Product_Cost' nor 'Cost_%' column e
xists in the dataframe. The 'Cost_Amount' column will not be
calculated."
            )
            df["Cost_Amount"] = 0
    except Exception as e:
        st.error(f"Cost calculation failed. Error: {e}")
    return df
```

Step By Step:

Try-Except Block:

- The function is wrapped in a try-except block to handle any potential errors gracefully during the cost calculation process.

Cost Calculation Logic:

- **Product Cost Based Calculation**: If the `Product_Cost` column exists in the dataframe, the function calculates the `Cost_Amount` by multiplying the `Quantity` of each product sold by its `Product_Cost`.
- **Percentage Based Calculation**: If the dataframe contains a `Cost_%` column instead, the `Cost_Amount` is calculated as a percentage of the `Sales_Amount`. This

method is useful when costs are defined as a percentage of sales, a common scenario for products with variable cost structures.

- **Fallback**: If neither column is present, the function issues a warning and sets the `Cost_Amount` to 0.

Return Value:

- The function returns the dataframe with the newly calculated `Cost_Amount` column appended or modified.

`get_all_columns` Function

```
def get_all_columns(df_list):
    all_columns = []
    for df in df_list:
        if df is not None:
            non_numeric_cols = df.select_dtypes(exclude="numb
er").columns.tolist()
            non_empty_cols = [
                col for col in non_numeric_cols if (df[col].d
ropna() != pd.Timestamp(0)).any()
            ]
            all_columns.extend(non_empty_cols)
    return list(set(all_columns))
```

Initialize a Collector List (`all_columns`):

- A list to accumulate the names of all non-numeric, non-empty columns found in the `df_list`

The function iterates over each DataFrame within the provided list `non_numeric_cols = df.select_dtypes(exclude="number").columns.tolist()` selectively extracts column names that do not store numeric data

Filter Out Entirely Empty Columns:

- The comprehension `[col for col in non_numeric_cols if (df[col].dropna() != pd.Timestamp(0)).any()]` further refines the selection by excluding columns that are entirely empty.

`all_columns.extend(non_empty_cols)` appends the filtered, non-empty, non-numeric column names to the `all_columns` list.

Eliminate Duplicates and Return the Result:

- `return list(set(all_columns))` converts the `all_columns` list to a set to remove any duplicate entries, then converts it back to a list.

`process_expenses` Function

```
@st.cache_data
def process_expenses(df, expenses_df):
    try:
        unique_expenses = expenses_df["Expense"].unique()
        zero_data = pd.DataFrame(0, index=df.index, columns=u
nique_expenses)
        df = pd.concat([df, zero_data], axis=1)

        for expense_name in unique_expenses:
            expense_rows = expenses_df[expenses_df["Expense"]
== expense_name]
            for _, row in expense_rows.iterrows():
                df = allocate_expense(df, row)

        return df
    except Exception as e:
        st.error(f"Expense processing failed. Error: {e}")
    return df
```

`@st.cache_data` is a decorator provided by Streamlit to cache the function's output. This caching mechanism prevents the function from re-executing and recalculating expense allocations every time the app reruns.

Unique Expense Categories:

- `unique_expenses = expenses_df["Expense"].unique()` extracts a list of unique expense names from the `expenses_df`. This step is crucial for identifying all different types of expenses present in the dataset, which will each be allocated across the relevant transactions in `df`.

Initializing Expense Columns with Zeroes:

- `zero_data = pd.DataFrame(0, index=df.index, columns=unique_expenses)` creates a new DataFrame filled with zeroes, having the same row index as `df` but with one column for each unique expense. This DataFrame is then concatenated to `df`, effectively adding a column for each expense type, initialized to zero. This prepares us for allocating actual expense amounts to these newly created columns.

Iterating Over Each Expense Type:

- The loop `for expense_name in unique_expenses:` iterates through each unique expense, selecting the rows in `expenses_df` that correspond to the current expense type. For each type of expense (e.g., rent, utilities), the function iterates over the expense entries. It uses criteria defined (in the `allocate_expense` function, to be discussed later) to distribute these expenses across the sales transactions.

Allocating Expense Amounts:

- Within each expense type iteration, another loop iterates over the rows (`expense_rows`) of `expenses_df` that pertain to the current expense category. For each row, `df = allocate_expense(df, row)` calls the `allocate_expense` function, passing the main DataFrame (`df`) and the current expense row (`row`). This function call is where the actual allocation of expense amounts to the appropriate transactions or products in `df` occurs, based on predefined rules or criteria specified within each row of `expenses_df`.

`allocate_expense` Function

```
def allocate_expense(df, row):
    try:
        expense_name = row["Expense"]
        allocations = row["Allocate_To"].split(";")
        by_tran = row["Allocate_By_Tran"] / len(allocations)
        by_value = row["Allocate_By_Value"] / len(allocation
s)
        total_amount = row["Amount"]
        amount_by_tran = total_amount * by_tran
        amount_by_value = total_amount * by_value
        for allocation in allocations:
            df = allocate_based_on_rules(
                df, allocation.strip(), amount_by_tran, amoun
t_by_value, expense_name
            )
    except Exception as e:
        st.error(f"Expense allocation failed. Error: {e}")
    return df
```

▶ This function is called by `process_expenses` for each row in the expenses DataFrame, applying specific allocation rules to distribute each expense appropriately.

Breakdown

- **Initializing Variables**: The function begins by extracting information from the current expense row:
  - `expense_name`: The name of the expense being processed.
  - `allocations`: A list of criteria for how this expense should be allocated across transactions. This could be based on specific categories, products, or even spread evenly across all transactions.
  - `by_tran` and `by_value`: These variables determine how much of the expense is allocated based on transaction count versus transaction value. They are derived by dividing the allocation methods (`Allocate_By_Tran` and `Allocate_By_Value`) by the number of allocations, ensuring that the expense amount is distributed according to the specified ratios.
- **Calculating Allocation Amounts**:
  - `amount_by_tran`: The portion of the total expense amount to be allocated based on the number of transactions.
  - `amount_by_value`: The portion of the total expense amount to be allocated based on the value of transactions.
- **Allocating Expenses**:
  - The function iterates over each allocation criterion within the `allocations` list.
  - For each criterion, it calls `allocate_based_on_rules`, passing the current DataFrame, the allocation criterion, the amounts to be allocated by transaction and by value, and the name of the expense.
  - This process ensures that each part of the expense is allocated according to its specific rules, which could vary widely depending on the nature of

the expense and the business's accounting practices.

`allocate_based_on_rules` Function

```
def allocate_based_on_rules(df, allocation, amount_by_tra
n, amount_by_value, expense_name):
    try:
        if allocation.lower() == "all":
            if not df.empty:
                df[expense_name] += amount_by_tran / len(d
f) + amount_by_value * (
                    df["Sales_Amount"] / df["Sales_Amoun
t"].sum()
                )
        else:
            conditions = allocation.split(";")
            temp_df = df.copy()
            for condition in conditions:
                if "=" in condition:
                    key, value = condition.split("=")
                    if key.strip() in df.columns:
                        temp_df = temp_df[temp_df[key.stri
p()] == value.strip()]
            if not temp_df.empty:
                total_sales = temp_df["Sales_Amount"].sum
() or 1  # Avoid division by zero
                df.loc[temp_df.index, expense_name] += amo
unt_by_tran / len(temp_df) + amount_by_value * (
                    temp_df["Sales_Amount"] / total_sales
                )
            else:
                st.warning(f"No rows met the condition for
{expense_name}")
```

```
    except Exception as e:
        st.error(f"Allocation rule application failed. Err
or: {e}")
    return df
```

Universal Allocation:

- If the allocation criterion is `"all"`, the function distributes the expense evenly across all transactions (`amount_by_tran / len(df)`) and also proportionally based on the sales amount. This is a straightforward method used when an expense should be evenly spread out or when it scales with sales activity.

Conditional Allocation:

- For more specific allocation rules, the function splits the allocation criteria (`allocation.split(";")`) and filters the DataFrame (`temp_df`) to match these conditions.

`calculate_totals` Function

```
def calculate_totals(df, expenses_df):
    try:
        expense_columns = []

        # Iterate through the expenses and use the weighte
d column if available
        for expense_name in expenses_df["Expense"].unique
():
            weighted_column_name = f"{expense_name}_Weight
ed"
            if weighted_column_name in df.columns:
                expense_columns.append(weighted_column_nam
e)
            else:
                expense_columns.append(expense_name)

        # Calculate total expenses using the selected colu
mns
        total_expense = df[expense_columns].sum(axis=1)
        df["Total_Expense"] = total_expense
        df["Net_Profit"] = df["Sales_Amount"] - df["Total_
Expense"] - df["Cost_Amount"]
    except Exception as e:
        st.error(f"Total calculation failed. Error: {e}")
    return df
```

A list named `expense_columns` is initialized. This list will hold the names of all columns in `df` that represent expenses, either directly or as weighted calculations

The function iterates over unique expense names from `expenses_df`. For each expense, it checks if a weighted column (e.g., `Rent_Weighted`) exists in `df`. Weighted columns are preferred because they provide a more refined allocation of expenses based on specific rules or proportions. If a weighted column is present, it's added to `expense_columns`; otherwise, the regular expense column name is added.

Calculating Total Expenses: Using the identified `expense_columns`, the function calculates the sum of these columns for each row (`axis=1`), representing the total expense for each transaction or item. This sum is then assigned to a new column, `Total_Expense`, in `df`. This step aggregates all individual expense allocations into a single figure.

The function calculates `Net_Profit` for each item by subtracting `Total_Expense` and `Cost_Amount` from `Sales_Amount`

The function is wrapped in a `try-except` block to gracefully handle any errors that might occur during the calculation process.

`append_totals` Function

```
def append_totals(df):
    try:
        totals = df.select_dtypes(np.number).sum().rename
("Total")
        df.index = df.index.astype(str)  # Convert index t
o string
        df = pd.concat([df, pd.DataFrame(totals).T])
    except Exception as e:
        st.error(f"Appending totals failed. Error: {e}")
    return df
```

Summarizing Numeric Data:

- `totals = df.select_dtypes(np.number).sum().rename("Total")`: This line performs a few tasks:
    - `df.select_dtypes(np.number)`: Selects columns in the DataFrame `df` that contain numeric data types.
    - `.sum()`: Computes the sum of each selected numeric column across all rows.
    - `.rename("Total")`: Renames the Series resulting from the sum operation to "Total," indicating that these values represent the total sums of their respective columns.

Appending the Totals Row:

- `df = pd.concat([df, pd.DataFrame(totals).T])`: This line appends the totals row to the DataFrame `df`:
    - `pd.DataFrame(totals).T`: Converts the `totals` Series into a DataFrame and then transposes it (`T`). Transposing is necessary because `totals` is a Series where each element represents a total sum for a column; transposing turns it into a row that can be appended to the DataFrame.
    - `pd.concat([df, ...])`: Concatenates the original DataFrame `df` with the newly created DataFrame containing the totals row. This effectively appends the totals row to the bottom of `df`

`remove_empty_columns` Function

```
def remove_empty_columns(df):
    df = df.dropna(how="all", axis=1)
    return df
```

Breakdown

- `dropna()`: A Pandas DataFrame method used to remove missing values. The method can be configured to drop rows or columns based on various criteria.
- `how="all"`: This parameter tells `dropna()` to only remove the rows/columns where all values are `NaN`. In this context, since `axis=1` specifies column-wise operation, it means a column will be dropped if every cell within that column is missing data.

After removing columns that are entirely empty, the cleaned DataFrame `df` is returned.

`file_uploader_with_session_state` Function

```
def file_uploader_with_session_state(name, type):
    uploaded_file = st.file_uploader(f"Upload {name} fil
e", type=type)
    file_extension = None
    if uploaded_file is not None:
        file_extension = uploaded_file.name.split(".")[-1]
# Get extension
        st.session_state.uploaded_files[name] = uploaded_f
ile
    return st.session_state.uploaded_files.get(name, Non
e), file_extension
```

`uploaded_file = st.file_uploader(f"Upload {name} file", type=type)`: This line leverages Streamlit's `file_uploader` widget, allowing users to upload files directly through the application's user interface.

Handling the Uploaded File:

- The conditional block checks if a file has been successfully uploaded (`if uploaded_file is not None:`). If so, it proceeds to extract the file extension:
    - `file_extension = uploaded_file.name.split(".")[-1]`: Extracts the file extension from the uploaded file's name. This information is crucial for processing the file correctly, whether it's a CSV, Excel, or another supported format.

`st.session_state.uploaded_files[name] = uploaded_file`: This line stores the uploaded file in Streamlit's session state under a dynamically generated key based on the `name` variable.Utilizing session state in this manner ensures that the uploaded file persists across reruns of the app, preventing the need for users to re-upload files after every interaction.

The function returns two values: the uploaded file object (if present) and its file extension.

`to_csv_xlsx` Function

```
def to_csv_xlsx(df, extension):
    if extension == "csv":
        return df.to_csv(index=False)
    elif extension in ["xlsx", "xls"]:
        output = io.BytesIO()
        with pd.ExcelWriter(output, engine="openpyxl") as 
writer:
            df.to_excel(writer, index=False)  # Exclude in
dex
        return output.getvalue()
```

This function plays a critical role in the data export feature of the application, allowing users to download processed data in their preferred format, either as CSV or Excel (XLSX)

It is pretty straightforward:

CSV Format:

we start with a condition that checks if the desired output format is CSV.If so, the DataFrame is converted to a CSV string with `index=False` to exclude the

DataFrame index from the output. The CSV string is then returned.

Excel Format

- `elif extension in ["xlsx", "xls"]:`: This condition checks if the desired output format is either XLSX or XLS (Excel).
- `output = io.BytesIO()`: Creates an in-memory bytes buffer where the Excel file will be written. This is necessary because the Excel file content needs to be stored as bytes data before it can be encoded and downloaded.
- `with pd.ExcelWriter(output, engine="openpyxl") as writer:`: Initializes an Excel writer object that can write to the specified bytes buffer using the `openpyxl` engine, which is suitable for writing XLSX files.
- `df.to_excel(writer, index=False)`: Converts the DataFrame to Excel format, writing it to the buffer through the Excel writer. The `index=False` parameter is used here as well, to exclude the DataFrame index from the output.

`create_download_link` Function

```
def create_download_link(df, filename, extension):
    file_data = to_csv_xlsx(df, extension)
    if isinstance(file_data, str):
        file_data = file_data.encode()
    b64 = base64.b64encode(file_data).decode()
    href = f"<a href='data:file/{extension};base64,{b64}'
download='{filename}'><input type='button' value='Click to
Download {filename}'></a>"
    return href
```

This function generates a hyperlink for the user to download the processed data file, which was converted into either a CSV or Excel format by the `to_csv_xlsx` function.

`check_uploaded_files` Function

```python
def check_uploaded_files():
    if "uploaded_files" not in st.session_state:
        st.session_state.uploaded_files = {}

    customers_file, file_extension = file_uploader_with_session_state(
        TABLE_CUSTOMERS, ["csv", "xlsx"]
    )
    invoices_file, _ = file_uploader_with_session_state(TABLE_INVOICES, ["csv", "xlsx"])
    products_file, _ = file_uploader_with_session_state(TABLE_PRODUCTS, ["csv", "xlsx"])
    expenses_file, _ = file_uploader_with_session_state(TABLE_EXPENSES, ["csv", "xlsx"])

    return customers_file, invoices_file, products_file, expenses_file, file_extension
```

The main goal of this function is to facilitate and manage the upload of the customer, invoice, product, and expense data.

It does so by:

- Providing a Streamlit sidebar for uploading files.
- Storing references to these uploaded files in Streamlit's session state, allowing the application to access these files across reruns without requiring the user to re-upload them.

Step-by-Step

1. **Checking and Initializing Session State for Uploaded Files**:

- The function starts by checking if there is an entry named `"uploaded_files"` in Streamlit's session state. This entry is intended to keep track of files uploaded during the session.
- If `"uploaded_files"` does not exist (meaning no files have been uploaded yet or the session state is fresh), it initializes an empty dictionary under this name. This dictionary will store references to the uploaded files, keyed by their respective data categories (customers, invoices, etc.).

2. **Uploading Files Through Custom Wrapper**:
   - **Custom File Uploader Invocation**: For each category of data needed by the application (customers, invoices, products, expenses), the function calls `file_uploader_with_session_state`. This custom wrapper around Streamlit's file uploader adds functionality to track uploaded files via session state.
   - **File Types**: It specifies the types of files that users can upload for each category (CSV or Excel), ensuring that only compatible file formats are processed.

At the end of the function, it returns the file objects for each data category. This makes the uploaded files accessible to other parts of the application for data processing and analysis.

In Practice:

User uploads their customer, invoice, product, and expense files once. As they interact with the application—filtering data, generating reports, etc.—the application does not ask for these files again, thanks to the session state keeping track of the uploads. This seamless experience is made possible by the `check_uploaded_files` function.

`generate_report` Function

```
def generate_report(df, allocation_factor, report_type, ex
penses_df):
    with st.spinner("Generating the report..."):
        if report_type == "Summary":
            if allocation_factor.lower() == "all":
                report_df = pd.DataFrame(
                    df.loc["Total"].copy()
                ).transpose()  # Only include the "Total"
row
                report_df.index = ["Grand Total"]  # Chang
e "Total" to "Grand Total"
            else:
                df = df[
                    [
                        allocation_factor,
                        "Sales_Amount",
                        "Cost_Amount",
                        "Total_Expense",
                        "Net_Profit",
                    ]
                ]
                report_df = df.groupby(allocation_factor).
sum()

                # Add the grand total row
                report_df.loc["Grand Total"] = report_df.s
um()

                # Rename columns for the report
                report_df.rename(
                    columns={
                        "Sales_Amount": "Sum of Sales_Amou
nt",
                        "Cost_Amount": "Sum of Cost_Amoun
t",
                        "Total_Expense": "Sum of Total Exp
enses",
                        "Net_Profit": "Sum of Net Profit",
                    },
                    inplace=True,
                )
                display_df = report_df.copy()
                if allocation_factor.lower() != "all":
                    report_df = report_df.reset_index(
                        drop=False
                    )  # Reset index for downloading

        elif report_type == "Detailed":
            if allocation_factor.lower() == "all":
                report_df = df[df.select_dtypes(np.numbe
r).columns.tolist()].copy()
                report_df = report_df.groupby(df.index).su
m()
            else:
                report_df = df[
                    [allocation_factor] + df.select_dtypes
(np.number).columns.tolist()
                ].copy()
                report_df = report_df.groupby(allocation_f
actor).sum()

                # grand total row
                if "Total" not in report_df.index:
                    report_df.loc["Grand Total"] = report_
df.sum()

                # columns renamed for the report
                report_df.rename(
                    columns={
                        "Sales_Amount": "Sum of Sales_Amou
nt",
                        "Cost_Amount": "Sum of Cost_Amoun
t",
                        "Net_Profit": "Sum of Net Profit",
                    },
                    inplace=True,
                )
                # Rename individual expense columns
                for expense in expenses_df["Expense"].uniq
ue():
                    report_df.rename(
                        columns={expense: f"Sum of {expens
e}"}, inplace=True
                    )

                display_df = report_df.copy()  # copy for
displaying
                if allocation_factor.lower() != "all":
                    report_df = report_df.reset_index(
                        drop=False
                    )  # Reset index for downloading

    return report_df, display_df
```

The `generate_report` function is designed to aggregate the processed data into meaningful insights, formatted as either a "**Detailed**" or "**Summary**" report, depending on the user's selection.

The function begins with a `with st.spinner("Generating the report..."):` block, indicating to the user that their report is being processed.

Based on the `report_type` parameter, the function branches into two paths: one for generating a "Summary" report and another for a "Detailed" report.

Summary Report Generation:

- **All Data Aggregation**: If the `allocation_factor` is set to "all", the function creates a report that aggregates all data into a single "Grand Total" row. This provides a high-level overview of the entire dataset.
- **Grouped Data Aggregation**: If specific `allocation_factor` is chosen, the function groups the data by this factor and calculates sums for key metrics. This allows users to see how different categories (e.g., customers, products) contribute to overall metrics.
- **Grand Total Row Addition**: Regardless of the grouping, a "Grand Total" row is appended to provide a summary of the entire dataset.

Detailed Report Generation:

Similar to the summary report but retains more granularity, allowing users to drill down into specific data points. The detailed report still aggregates data but does so without collapsing it into high-level categories unless specified by the `allocation_factor`

Interaction and Output

- **Displaying the Report**: The summary report is displayed in the application, allowing you to see aggregated financial metrics by product category alongside the grand total figures.
- **Downloading the Report**: A download link is generated, enabling you to download the report for offline analysis, sharing with team members, or for record-keeping purposes.
-

A Scenario: Generating a Summary Report by Product

Imagine you're a business owner who has uploaded data to our ProfitScan application, and now you want to generate a summary report based on product categories to see how each category contributes to sales, costs, expenses, and net profits.

Step 1: User Selections

- **Report Type**: You choose "Summary" from the report options.
- **Allocation Factor**: You select "Product" as the allocation factor, aiming to group the report's financial metrics by different product categories.

Step 2: Preparing Data for Reporting

- The application has already processed your data, merging customer, invoice, and product information, calculating costs, allocating expenses, and appending totals.
- This processed dataset is ready to be aggregated further into the report format based on your selections.

Step 3: Generating the Report

- **Aggregation**: The function groups the data by the "Product" category since you've chosen it as the allocation factor. It then calculates the sum of "Sales_Amount", "Cost_Amount", "Total_Expense", and "Net_Profit" for each product category.
- **Grand Total Calculation**: After grouping and summing up the metrics by product, a "Grand Total" row is added to the report. This row represents the sum of all financial metrics across all product categories, providing a quick overview of the overall financial health related to your products.
- **Preparing for Display and Download**: Two versions of the report are prepared:
  - `display_df`: Formatted specifically for display within the ProfitScan application. It allows you to visually inspect the report on the web interface.
  - `report_df`: Structured for download.

Step 4: Interaction and Output

- **Displaying the Report**: The summary report is displayed in the app, allowing you to see aggregated financial metrics by product category alongside the grand total figures.
- **Downloading the Report**: A download link is generated, enabling you to download the report.

`add_missing_products_to_products_table` Function

```
def add_missing_products_to_products_table(products_table_nam
e, missing_products):
    # dataframe with the missing products and a cost of zero
    missing_products_df = pd.DataFrame(
        {
            "Product": list(missing_products),
            "Product_Cost": 0,
        }  # we put 0 here, we could've put np.nan
    )

    # Append the dataframe to the products table
    missing_products_df.to_sql(
        products_table_name, conn, if_exists="append", index=
False
    )

    # Update the products data stored in the session state
    products_df = load_data_from_db(products_table_name)
    st.session_state[f"{products_table_name}_data"] = product
s_df
```

This function is designed to address a specific scenario: after merging invoice data with product data, there might be products referenced in invoices that do not exist in the products table. This discrepancy would result in incomplete or incorrect financial analysis, especially when calculating costs and profits. The function ensures that every product sold, as per the invoices, has a corresponding entry in the products table

Identify Missing Products:

- After loading and processing the invoices and products data, the application identifies any products listed in the invoices that are not found in the products table. These are considered "missing" products.

Create a DataFrame for Missing Products:

- A new DataFrame is constructed, containing two columns: `Product` and `Product_Cost`. Each missing product is listed with a default `Product_Cost` of 0. This step assumes that the cost for these missing products is initially unknown or unavailable.

Append Missing Products to the Products Table:

- The `missing_products_df` DataFrame is appended to the existing products table in the SQLite database using the `.to_sql` method. The `if_exists='append'` parameter ensures that the new entries are added to the table without altering the existing data.

`apply_expense_weights` Function

```
def apply_expense_weights(df, expenses_df):
    weighted_df = df.copy()
    for i, row in expenses_df.iterrows():
        if "Weights" in row and pd.notna(row["Weights"]) and
st.session_state.enabled_expenses[i]:
            weight_info = eval(row["Weights"])
            column_name = weight_info["column_name"]
            weight_mapping = weight_info["weights"]
            # Calculate the weight for each row based on the
mapping
            weighted_df[f"{row['Expense']}_Weight"] = weighte
d_df[column_name].map(weight_mapping).fillna(0)

            # Calculate xWeight for each row using the specif
ic value from the Expense column
            weighted_df[f"{row['Expense']}_xWeight"] = weight
ed_df[row["Expense"]] * weighted_df[f"{row['Expense']}_Weigh
t"]

            # Calculate Total xWeight for the entire datafram
e
            total_xWeight = weighted_df[f"{row['Expense']}_xW
eight"].sum()

            # Calculate Share for each row
            weighted_df[f"{row['Expense']}_Share"] = weighted
_df[f"{row['Expense']}_xWeight"] / total_xWeight if total_xWe
ight != 0 else 0

            # Adjust for rounding errors
            difference = 1.0 - weighted_df[f"{row['Expense']}
_Share"].sum()
            weighted_df.loc[weighted_df[f"{row['Expense']}_Sh
are"].idxmax(), f"{row['Expense']}_Share"] += difference

            # Calculate weighted expense
            weighted_df[f"{row['Expense']}_Weighted"] = weigh
ted_df[f"{row['Expense']}_Share"] * row["Amount"]

            # Replace original expense column with weighted v
alue
            weighted_df.drop(columns=[row["Expense"]], inplac
e=True)
```

```
            weighted_df.rename(columns={f"{row['Expense']}_We
ighted": row["Expense"]}, inplace=True)

            # Drop other temporary columns used in the calcul
ation
            weighted_df.drop(columns=[f"{row['Expense']}_Weig
ht", f"{row['Expense']}_xWeight", f"{row['Expense']}_Share"],
inplace=True)

    return weighted_df
```

Step-by-Step

1. **Copy Original DataFrame**: Creates a copy of the provided DataFrame (`df`) to preserve the original data and ensure that all manipulations are performed on a separate object.
2. **Iterate Over Expenses**: Loops through each row in the `expenses_df` DataFrame, which contains details about each expense, including its name and the weight rules for allocation.
3. **Check for Weight Information**: For each expense, it checks if weight information is provided and valid. This includes verifying that the expense is enabled for weighting as indicated by the `st.session_state.enabled_expenses[i]` flag.
4. **Weight Application**:
    - **Mapping Weights**: Based on the `weight_info`, which is extracted using `eval(row["Weights"])`, it maps these weights to the corresponding rows in `weighted_df` based on a specified column (`column_name`). This process assigns a weight to each transaction or product.
    - **Calculating Weighted Expense**: Multiplies the expense amount by the calculated weights to distribute the expense according to the predefined rules.
    - **Adjusting for Total and Share**: Calculates the total weighted expense (`total_xWeight`) and the share of each expense (`_Share`). It also adjusts for any rounding errors to ensure the total shares sum up to 1.

- **Replacing Expense Column**: Updates the original expense column with the newly calculated weighted expense values and removes temporary columns used for calculations.

5. **Returning the Weighted DataFrame**: After applying weights to all relevant expenses and cleaning up temporary columns, the function returns the modified DataFrame with expenses allocated according to the specified weights.

Integration with the Application

This function plays a critical role in the financial analysis process by allowing for more sophisticated expense allocation based on business logic or analysis needs. For instance, if certain products or categories are known to have a higher burden of specific expenses, this function can allocate expenses accordingly, leading to a more accurate reflection of net profit or loss.

`main` Function

we'll break down the process into smaller sections, explaining each part with its corresponding code block and logic.

Step 1: Setting Up the Page

```
def main():
    st.set_page_config(page_title="ProfitScan", layout="wid
e")
    hide_streamlit_style = """
            <style>
            footer {visibility: hidden;}
            </style>
```

```
        """
    st.markdown(hide_streamlit_style, unsafe_allow_html=True)
```

- `set_page_config`: This function configures the page, setting the title to "ProfitScan" and opting for a wide layout for more space.
- **CSS Styling**: The `hide_streamlit_style` variable contains CSS to hide the default Streamlit footer.

Step 2: Displaying Instructions

```
with st.expander("Instructions", expanded=False):
    st.markdown("""
        <div style="background-color: #464e56; padding: 30px;
border-radius: 5px;">
            <h2 style='color: white;'>Instructions:</h2>
            <!-- Instructions content goes here -->
        </div>
        """, unsafe_allow_html=True)
```

- **Expander**: Creates an expandable section titled "Instructions". It's initially collapsed (`expanded=False`), keeping the UI neat while still accessible.
- **Instructions**: Detailed instructions are provided within a styled div. This helps users understand the data requirements and how to interact with the app.

Step 3: File Upload Interface

```
with st.sidebar:
        st.markdown(
            "<h1 style='text-align: center; font-size: 24p
x;'>Upload your files</h1>",
            unsafe_allow_html=True,
        )

        if "uploaded_files" not in st.session_state:
            st.session_state.uploaded_files = {}

        st.subheader("Customer Data")
        customers_file, file_extension = file_uploader_with_s
ession_state(
            "Customers", ["csv", "xlsx"]
        )

        st.subheader("Transaction Data")
        invoices_file, _ = file_uploader_with_session_state
("Invoices", ["csv", "xlsx"])
        products_file, _ = file_uploader_with_session_state
("Products", ["csv", "xlsx"])

        st.subheader("Expense Data")
        expenses_file, _ = file_uploader_with_session_state
("Expenses", ["csv", "xlsx"])
```

- **Sidebar**: Utilizes Streamlit's sidebar feature to organize file upload widgets neatly, prompting users to upload their CSV or Excel files.
- **Checking and Initializing Uploaded Files in Session State**

```
 if "uploaded_files" not in st.session_state:
           st.session_state.uploaded_files = {}
```

This line checks if `uploaded_files` exists in the session state. If not, it initializes `uploaded_files` as an empty dictionary. This dictionary will be used to store references to the files uploaded by the user.

File Uploaders for Data Categories:

Each file uploader is created for different data categories: Customers, Invoices, Products, and Expenses

```
customers_file, file_extension = file_uploader_with_session_s
tate("Customers", ["csv", "xlsx"])
```

This line, and its variants for invoices, products, and expenses, sets up file uploaders specific to each data category.

Step 4: Data Processing and Validation

```
if customers_file and invoices_file and products_file and exp
enses_file:
```

This line checks if all necessary files have been uploaded by the user. It ensures that the process only continues if all four datasets are available

```
customers_required_columns = ["Customer"]
        invoices_required_columns = [
            "Invoice_No",
            "Customer",
            "Product",
            "Quantity",
            "Sales_Amount",
        ]
        products_required_columns = ["Product"]

        expenses_required_columns = ["Expense", "Amount"]
```

For each dataset, a list of required columns is defined. These lists represent the minimum schema needed for the application to work correctly, ensuring that the uploaded files contain all necessary information for processing.

- **Customers Data:**
  - Required to contain a "Customer" column.
- **Invoices Data:**
  - Must include "Invoice_No", "Customer", "Product", "Quantity", and "Sales_Amount".
- **Products Data:**
  - A "Product" column is needed, with either "Product_Cost" or "Cost_%" being optional but necessary for cost calculations.
- **Expenses Data:**
  - Should have "Expense" and "Amount" columns.

Loading Data:

```
load_data(customers_file, "customers", customers_required_columns)
load_data(invoices_file, "invoices", invoices_required_columns)
load_data(products_file, "products", products_required_columns)
load_data(expenses_file, "expenses", expenses_required_columns)
```

Each `load_data()` call is responsible for loading the uploaded files into the application, applying the schema validation according to the lists of required columns previously defined.

Data Retrieval from Database:

Once the data is validated and loaded, it's retrieved from the database for further processing.

```
invoices_df = load_data_from_db("invoices")

products_df = load_data_from_db("products")
```

Data Integrity and Consistency Checks:

```
invoices_products = set(invoices_df["Product"].unique())
products_products = set(products_df["Product"].unique())
```

These sets collect unique product names from both invoices and products data. This step is crucial for identifying any discrepancies, such as products mentioned in invoices that don't have corresponding entries in the products dataset.

Merging Data and Calculating Costs:

The `merge_data()` function combines customer, invoice, and product data into a single dataframe. Following this, `calculate_cost()` computes the cost amount for each transaction based on product cost or cost percentage.

```
merged_df = merge_data("customers", "invoices", "products")

merged_df = calculate_cost(merged_df)
```

Preparation for Expense Allocation:

- **Retrieving All Columns:**
    - Retrieves all unique column names from the customers, invoices, and products data for potential use in expense allocation.
- **Loading Expenses Data:**
    - Expenses data is loaded for subsequent processing.

```
all_columns = get_all_columns(
    [
        load_data_from_db("customers"),
        load_data_from_db("invoices"),
        load_data_from_db("products"),
    ]
)

all_columns = ["All"] + [col for col in all_columns if col.lo
wer() != "date"]

# Load expenses data into a DataFrame
expenses_df = load_data_from_db("expenses")
```

Expense Allocation and Data Processing

```
processed_df = merged_df.copy()
```

This line creates a copy of the `merged_df`, which contains the merged data from customers, invoices, and products. The copy is stored in `processed_df` for further processing without altering the original merged data.

```
if "enabled_expenses" not in st.session_state:
            st.session_state.enabled_expenses = [True] * len
(expenses_df)

if "allocation_columns_pre_filled" not in st.session_state:
    st.session_state["allocation_columns_pre_filled"] = False
```

These lines ensure that the session state contains flags for tracking which expenses are enabled for processing and whether the allocation columns were pre-filled. This setup facilitates dynamic interaction with the user's inputs and preferences throughout the session.

User-Defined Expense Allocation Rules

```
if st.session_state["allocation_columns_pre_filled"]:
    st.success("Allocation columns have been pre-filled in th
e uploaded Expenses file. The settings will be used as-is.")
else:
    for i, row in expenses_df.iterrows():
        # User inputs for defining allocation rules...
```

This segment checks if the allocation columns in the uploaded expenses file were pre-filled. If so, it uses those settings directly. Otherwise, it iterates through each expense entry, prompting the user to define allocation rules.

Defining Allocation Rules and Applying Weights

```
for i, row in expenses_df.iterrows():
    amount_rounded = round(row["Amount"], 2)
    st.markdown(f"Allocation rules for {row['Expense']}, Amou
nt: {amount_rounded}")
```

```
    # Inputs for number of allocation rules, factors, and val
ues...
    # Slider for determining allocation percentage between tr
nsaction & value...
    # Checkbox for deciding whether to apply weight to an exp
ense...
```

For each expense, the application collects user inputs to define how the expense should be allocated across transactions or categories. This includes specifying the number of allocation rules, selecting factors for allocation, and deciding the allocation percentage between transaction amounts and fixed values.

Processing Expenses and Applying Weights

```
processed_df = process_expenses(merged_df, expenses_df)
weighted_final_df = apply_expense_weights(processed_df, expen
ses_df)
```

- After gathering all user-defined rules and preferences, `process_expenses` allocates expenses according to those rules. Then, `apply_expense_weights` adjusts the allocated expenses based on any additional weights applied by the user. The result is stored in `weighted_final_df`, representing the final processed data ready for analysis and reporting.

Displaying the Updated Expenses Table

```
st.markdown(
    "<h3 style='text-align:left; margin-top: 2rem'>Updated Ex
penses Table</h3>",
    unsafe_allow_html=True,
)
```

```
any_weights_applied = any(st.session_state.enabled_expenses)

expenses_to_display = (
    expenses_df
    if any_weights_applied
    else expenses_df.drop(columns="Weights", errors="ignore")
)
st.write(expenses_to_display)
```

This section presents the updated expenses table that reflects the allocations and adjustments made. It provides transparency on how expenses have been distributed across different categories or transactions.

Final Data Presentation

```
st.markdown(
    "<h3 style='text-align: left; margin-top: 2rem'>Final Tab
le</h3>",
    unsafe_allow_html=True,
)
st.dataframe(processed_df)
```

The final processed DataFrame, which might include `weighted_final_df` if weights were applied, is displayed. This table offers a comprehensive view of the financial data after all processing has been completed.

Report Generation Based on User Selections

```
if allocation_factor:
      # show the progress while the page is loading
      with st.spinner("Generating report..."):
          report_df, display_df = generate_report(
              processed_df, allocation_factor, report_type, e
xpenses_df
          )
      st.markdown(
          f"<h3 style='text-align:left;'>{report_type} Report
</h3>",
          unsafe_allow_html=True,
      )
      st.dataframe(display_df)  # display_df to display in th
e app

      report_filename = (
          f"{report_type}_Report_{allocation_factor}.{file_ex
tension}"
      )
      report_download_link = create_download_link(
          report_df, report_filename, file_extension
      )  # Use report_df for downloading
      st.markdown(report_download_link, unsafe_allow_html=Tru
e)

      conn.close()
```

This section allows users to generate and view reports based on the processed data. Users can select an allocation factor and report type (e.g., Summary or Detailed) to tailor the report to their needs.

A download link for the report is provided, enabling users to save the report for further analysis, sharing, or archival purposes.

Conclusion and Application Cleanup

```
conn.close()
```

Finally, the application closes the database connection with `conn.close()`, ensuring that all resources are properly released, and the application ends its execution cleanly.

What’s next ?

The goal it to take this Streamlit application and transform it into a production-grade application, that is able to handle larger amounts of data.

Below is a guide to navigate through the transformation process, and help you pick the technology stack that fits you the most

**

If there is a technology you are confortable with that isn’t included here, that’s not a problem.

**

These are just some suggestions and comparisons based on my experience.

Choosing a Backend Framework

- **Django**: Offers a robust, full-featured framework with an ORM, authentication, and more. Great for applications with complex data models and a need for rapid development with its "batteries-included" approach.
- **Flask**: A lightweight WSGI web application framework that's easy to get started with for simple applications but can be scaled with extensions. Flask is ideal for microservices architecture.

- **FastAPI**: Asynchronous support makes it suitable for high-performance applications dealing with real-time data. It's modern and fast, with automatic API documentation.

Frontend Technologies

- **React**: A JavaScript library for building user interfaces, React is great for creating single-page applications (SPAs) with dynamic content. It has a vast ecosystem and community support.
- **Vue.js**: Known for its ease of integration and incrementally adoptable ecosystem, Vue.js is good for projects where you want simplicity without sacrificing flexibility.
- **Angular**: Provides a full-fledged framework for SPAs, including a powerful CLI, but comes with a steeper learning curve.

Database Selection

- **PostgreSQL**: An advanced open-source relational database, ideal for complex queries and data integrity. It supports JSON and is extensible.
- **MySQL/MariaDB**: Widely used relational databases, known for their reliability and ease of use. They are great for a wide range of applications.
- **MongoDB**: A NoSQL database that's suitable for applications needing to handle large volumes of unstructured data, offering scalability and flexibility.

Security and Performance

- Implement HTTPS, use JWT for authentication, and follow OWASP guidelines to secure your application.
- Use caching, database indexing, and optimize queries for performance.

An example stack you can follow could be: React.js, FastAPI and PostgreSQL.

(Pick the technologies you have most confidence and experience in)