

Technical Documentation for Leave Management System



Table of Contents

Introduction.....	3
Brief Overview of the Project.....	3
Purpose of the Document.....	3
Scope and Objectives.....	3
Project Structure.....	5
Technology Stacks used.....	6
Functional Requirements.....	7
Data Upload and API Consumption.....	7
Employee Profile Management.....	10
Comprehensive Leave Data Visualization.....	12
Real-time Data Updates.....	14
Customizable Visualization Options.....	16
Reporting and Insights.....	18
Fiscal Year Configuration.....	18
Technical Requirements.....	19
Backend Technologies.....	19
Visualization Tool.....	20
Database.....	21
API Documentation.....	25
Error Mechanism.....	29
Security Considerations.....	34
Performance Optimization.....	36
Load Testing With Locust.....	38
Running the Test.....	38
Monitoring Results.....	39
Database Migration.....	40

Architecture Diagram.....	42
C4 Diagram.....	43
Context Diagram.....	43
Container Diagram.....	43
Component Diagram.....	44
Testing and Quality Assurance.....	45
Pytest DB Migration.....	45
Pytest Fast API.....	45
Flow Chart Diagram.....	46
Appendix.....	47
Glossary of Terms and Abbreviations.....	47
References to External Documentation, APIs, Libraries, or Frameworks Used.....	48
Additional Resources or Supporting Materials.....	48
Coding Standards Followed in the Project.....	48

Introduction

Brief Overview of the Project

The Leave Management System is a comprehensive solution designed to streamline and automate the process of managing employee leave requests and allocations.

Developed using FastAPI for the backend, PostgreSQL for the database, and Streamlit for the visualization dashboard, this system leverages modern web technologies and asynchronous programming to provide efficient and scalable leave management capabilities. The project also includes Docker for containerization, ensuring consistent deployment across different environments.

Purpose of the Document

The purpose of this document is to provide detailed technical documentation for the Leave Management System. It serves as a guide for developers, system administrators, and other stakeholders involved in the development, maintenance, and deployment of the system. The document aims to:

- Explain the overall architecture and structure of the system.
- Detail the implementation and configuration of various components.
- Provide guidance on setting up and running the system.
- Outline testing and deployment procedures.

Scope and Objectives

The scope of this document includes:

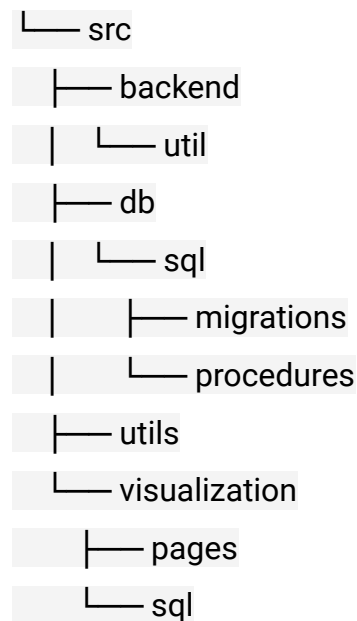
- Detailed explanation of the project's architecture and code structure.
- Step-by-step guidance on setting up the development environment and configuring the system.
- Instructions on how to use and extend the system's backend API.

- Procedures for managing the database, including migrations and stored procedures.
- Guidelines for creating and customizing the visualization dashboard.
- Strategies for testing the system's performance and load handling capabilities.

The primary objectives are to:

- Provide a clear and comprehensive reference for the development and maintenance of the Leave Management System.
- Facilitate understanding and usage of the system's features and functionalities.
- Ensure smooth deployment and operation of the system in various environments.
- Enable easy extension and customization of the system to meet specific requirements.

Project Structure



The project structure resembles a directory hierarchy commonly found in software projects, particularly those developed using certain frameworks or following certain design principles. While it's not directly indicative of a specific design pattern, it does exhibit characteristics of modular organization and separation of concerns, which are key principles in software engineering.

- **src:** This is typically short for "source" and contains the source code of the project.
- **backend:** This directory holds code related to the backend logic of the application, such as API endpoints, business logic, etc.
- **util:** It contains utility functions or classes that may be used across the backend codebase.
- **db:** This directory houses code related to database operations, including database schemas, queries, migrations, etc.
- **sql:** This subdirectory under "db" contains SQL files, perhaps representing database schemas, queries etc.

- migrations: Within the "sql" directory, this subdirectory holds database migration scripts used to manage changes to the database schema over time.
- procedures: This directory under "sql" contains stored procedures and other database-specific logic.
- utils: Similar to the "util" directory under "backend", this contains utility functions but is more generally applicable across the project.
- visualization: This directory contains code related to data visualization, such as frontend components, templates, or scripts for rendering charts, graphs, etc.
- pages: Within the "visualization" directory, this subdirectory contains specific pages or views for the visualization component of the application.

Overall, this structure reflects a modular organization of code, with separate directories for different functional components or layers of the software. While it doesn't map directly to a single design pattern, it aligns with principles of modular design, separation of concerns, and possibly aspects of layered architecture.

Technology Stacks used

- **Backend Framework:** FastAPI
- **Database:** PostgreSQL
- **Dashboard Development:** Streamlit
- **Data Visualization:** Plotly
- **API Testing:** Locust

The details for the technology stack is explained more in detail in the architecture diagram and the functional and non-functional requirements.

Functional Requirements

Data Upload and API Consumption

The sample json data we receive from the external application i.e. Vyaguta is given below:

```
{
  "id": 2653,
  "userId": 449,
  "emplId": "131",
  "teamManagerId": 400,
  "designationId": 3,
  "designationName": "Software Engineer",
  "firstName": "Srijana",
  "middleName": null,
  "lastName": "Shrestha",
  "email": "srijanashrestha123@lfttechnology.com",
  "isHr": true,
  "isSupervisor": false,
  "allocations": null,
  "leaveIssuerId": 697,
  "currentLeaveIssuerId": 697,
  "leaveIssuerFirstName": "Susanna",
  "leaveIssuerLastName": "Gurung",
  "currentLeaveIssuerEmail": "mahimagurung@lfttechnology.com",
  "departmentDescription": "HR and Administration",
  "startDate": "2024-04-23",
  "endDate": "2024-04-23",
  "leaveDays": 1,
  "reason": "x",
  "status": "REJECTED",
  "remarks": "Project health concern",
  "fiscalId": 102,
  "fiscalStartDate": "2023-07-17T00:00:00.000Z",
  "fiscalEndDate": "2024-07-15T00:00:00.000Z",
  "fiscalIsCurrent": true,
  "createdAt": "2024-04-14T05:33:10.000Z",
  "updatedAt": "2024-04-14T05:33:10.000Z",
  "isConverted": 0
}
```


The external Application has provided us an api endpoint to fetch data from their system i.e. <https://dev.vyaguta.lftechnology.com.np/api/leave/leaves?fetchType=all&startDate=2021-07-17&endDate=2024-04-23&size=10 000&roleType=issuer>.

We have an api endpoint http://localhost:8080/insert_leave_info served through the **fast-api** in the backend which retrieves data from vyaguta API every 30 seconds and stores it in the postgres database. The raw json data from the vyaguta API should be strictly in the above sample format. In case the raw data format changes we will need to modify raw table columns in our database. A **Bearer Token** and **URL** is required to authenticate with the vyaguta API. The bearer token is retrieved through environment variables for security purpose. Also, while ingesting we have specified data type for all the raw columns to be ingested as text which makes hassle free for ingestion except the **allocations** column which is a **json** datatype. These data validation and authentication token should be taken into consideration for data upload into the system.

The API also authenticates with the postgres database for which we have a separate database module i.e. **db** which has the logic to connect with the database through **asyncpg**. All the credentials required are acquired through environment variables. A **.env.example** file is included for the example in both backend and database module.

The system implements error handling mechanisms primarily through the use of exceptions. Let's go through the error handling mechanisms:

1. HTTPException

- The `HTTPException` is raised when there's an issue with the HTTP request or response, such as unauthorized access or server errors.
- It's used in various places throughout the code to return appropriate HTTP status codes and error messages to the client.
- For example, in the `get_leave_info` function, if the HTTP response status code is not 200, an `HTTPException` is raised with a status code of 401 (Unauthorized) and an appropriate error message.

2. Generic Exception Handling

- In several places, the code uses a generic `Exception` class to catch unexpected errors that may occur during execution.

- For instance, in the ``run_insert_leave_info`` function, a generic ``Exception`` is caught to handle any unexpected errors that might occur during the insertion of leave data into the database.
- When such exceptions occur, an appropriate error message is logged, and an ``HTTPException`` is raised with a status code of 500 (Internal Server Error) to indicate a server-side issue.

3. Logging

- The ``logger.error()`` function is used to log error messages with appropriate details whenever an error occurs.
- Error messages logged include details about the nature of the error, such as why leave data insertion failed or why leave information retrieval failed.

4. Graceful Handling of Errors

- The code attempts to gracefully handle errors by logging them and returning appropriate error messages, allowing the client to understand what went wrong.
- It distinguishes between different types of errors, such as unauthorized access, server-side errors, or unexpected errors, and responds accordingly with suitable HTTP status codes and error messages.

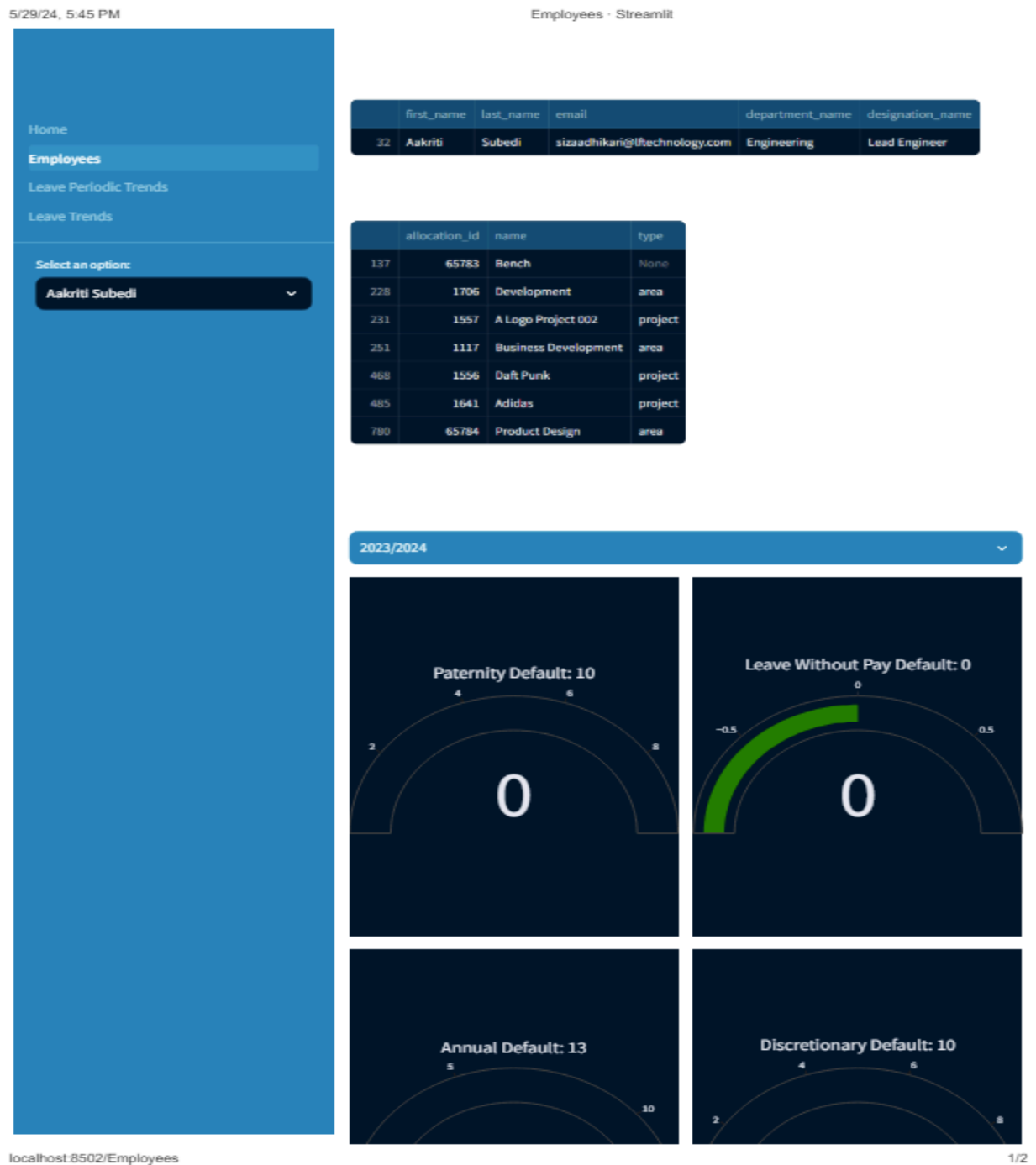
5. Asyncio Exception Handling

- Since the code extensively uses asynchronous programming with `asyncio`, errors within asynchronous tasks are handled properly.
- If an exception occurs within an asynchronous task, it is caught within the task itself, logged, and an appropriate response is returned.

Overall, the error mechanism in this code ensures that errors are logged for debugging purposes and that clients receive informative error messages in case of failures, helping them understand and address issues effectively.

Employee Profile Management

The code for employee profile management is in **pages/Employees.py** inside the visualization folder. It utilizes Streamlit to display employee data.



Here's a breakdown of the process:

1. Data Fetching:

- The code fetches employee details, allocation details, and leave balance data using SQL queries stored in separate files (``employee_details.sql``, ``allocation_details.sql``, ``leave_balance.sql``). This data retrieval is done through the ``fetch_data`` function.

2. Streamlit UI

- The Streamlit UI is initialized with ``st.subheader`` and ``st.sidebar.selectbox`` to create a dropdown menu for selecting an employee.
- Upon selecting an employee, the associated employee details and allocation details are retrieved and displayed in separate sections using ``st.write``.
- The employee details include first name, last name, email, department name, and designation name.
- The allocation details include allocation ID, name, and type.

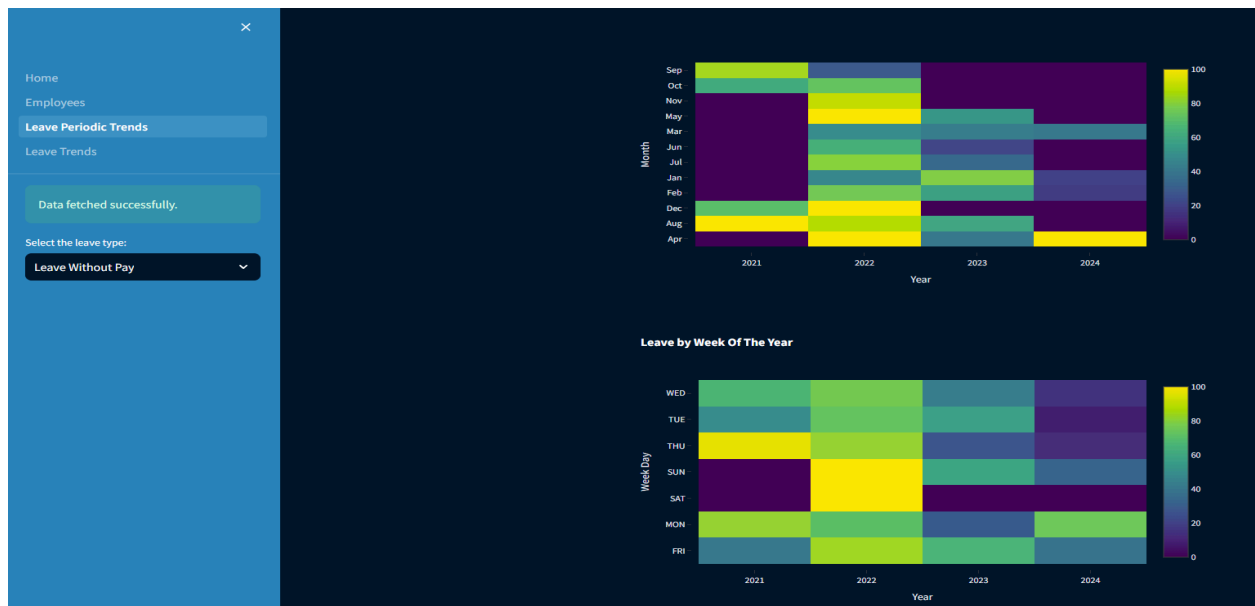
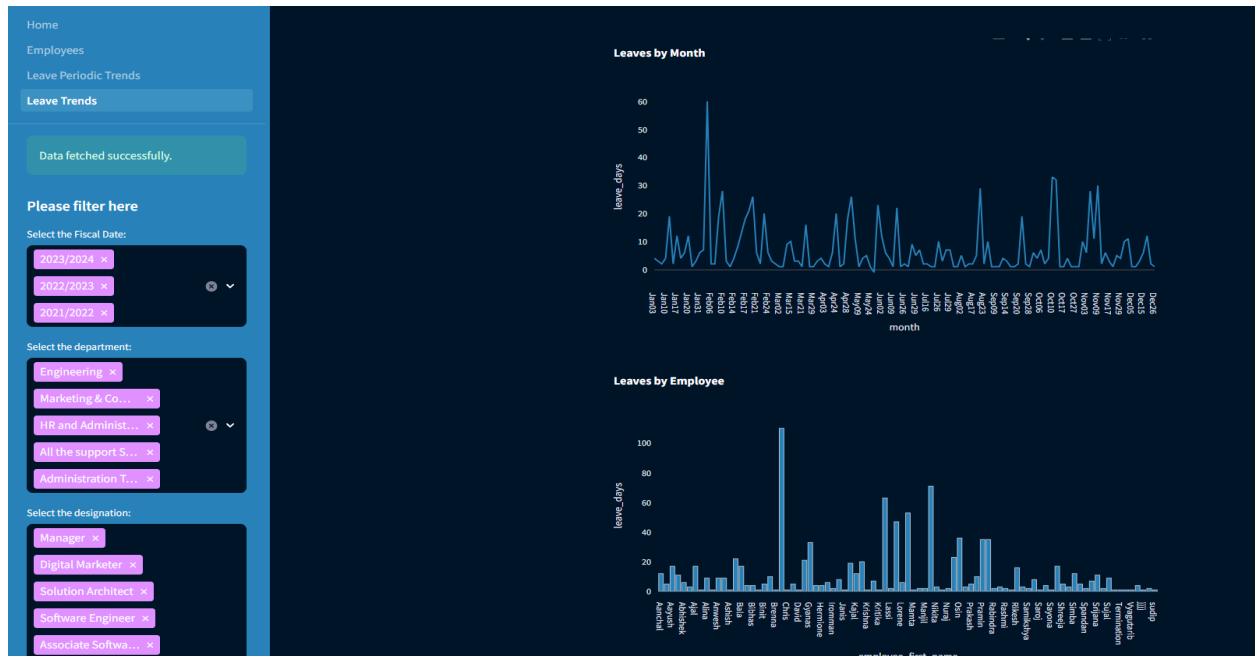
3. Leave Balance Visualization

- The code allows the user to select a fiscal date using a selectbox (``st.selectbox``). This date determines which leave balances to display.
- For each leave type (e.g., vacation, sick leave), a gauge chart is generated using Plotly (``go.Figure``) to visualize the used leave versus balance.
- The gauge chart indicates the used leave compared to the default number of days allotted for that leave type.
- Multiple gauge charts are displayed in a grid layout using Streamlit columns (``st.columns``) to organize the visualization.

Overall, the system provides a user-friendly interface for viewing employee details, allocation details, and leave balances, with interactive features such as dropdown menus and gauge charts for effective data visualization.

Comprehensive Leave Data Visualization

To visualize employee leave data for analyzing trends, balances, and distribution, the provided code utilizes Streamlit along with Plotly for generating interactive plots.



1. Fetching Data:

- The code fetches leave data from a SQL database using the `fetch_data` function. This data includes information about leave types, employee departments, designations, leave days, and fiscal dates.

2. Sidebar Filters:

- Streamlit's sidebar allows users to filter the data based on fiscal dates, employee departments, designations, and leave types. Users can select multiple options for each filter.

3. Monthly Leave Trend (Bar graph):

- Monthly leave trends are visualized using a line plot (bar chart) created with Plotly Express (`px.line`). The x-axis represents months, the y-axis represents the total leave days, and each line represents a different month.
- The chart provides insights into leave patterns over different months, helping to identify seasonal trends or variations in leave usage.

4. Employee-wise Leave Trend:

- Employee-wise leave trends are visualized using a bar chart created with Plotly Express (`px.bar`). The x-axis represents employee names, the y-axis represents the total leave days, and each bar represents an employee.
- This chart helps to analyze individual employees' leave behavior, identifying employees with high or low leave usage, and comparing leave patterns across the workforce.

5. Monthly Leave Trend (Heatmap):

- Monthly leave trends are visualized using a heatmap created with Plotly (`go.Heatmap`). The x-axis represents years, the y-axis represents months, and the color intensity represents the percentage of leave days for the selected leave type.
- The heatmap provides a clear visualization of leave trends over different months and years, helping to identify patterns or variations in leave usage.

6. Weekly Leave Trend

- Weekly leave trends are visualized using another heatmap created with Plotly (`go.Heatmap`). The x-axis represents years, the y-axis represents days of the week, and the color intensity represents the percentage of leave days for the selected leave type.
- This heatmap helps to analyze leave patterns on a weekly basis, identifying trends or variations in leave usage throughout the week.

7. Styling and Layout:

- The plots are styled using various Plotly parameters to enhance readability and aesthetics. For example, the color scheme is set using `color_discrete_sequence`, and the background color is adjusted using `plot_bgcolor`.
- Streamlit's `use_container_width=True` parameter ensures that the plots utilize the available width of the application's container, making them responsive to screen size changes.
- Plotly parameters are used to style the heatmaps and improve visibility. For example, the `hovertemplate` provides additional information when hovering over data points, and the `colorscale` enhances the readability of the heatmaps.
- Layout titles and axis labels are set to provide context and improve understanding of the visualizations.

Real-time Data Updates

1. Sources of Real-Time Data:

- External API: The primary source of data in the codebase is the external API data from Vyaguta. The backend API continuously fetches data from this API every 30 seconds and stores it in the raw schema of the postgres database.

- PostgreSQL: The data for the visualization is stored in this database. The ETL process is triggered automatically once the raw data is ingested.

2. Update Mechanism:

- Merge and Insert: The postgresql standard table applies **Merge Insert** mechanism due to which only new data are merged from the source into the target database.
- Asyncio: The codebase uses asyncio, employing asynchronous programming techniques to handle concurrent operations efficiently. This facilitates the implementation of asynchronous mechanisms for fetching real-time data.

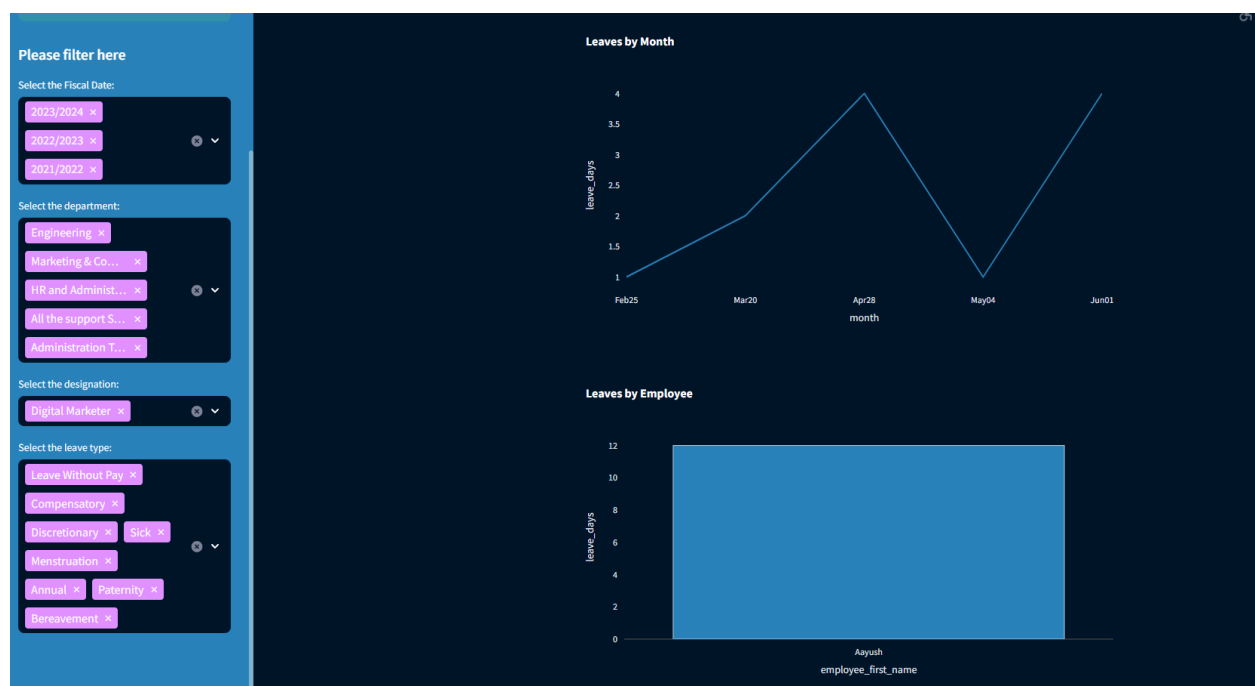
3. Ensuring Data Consistency:

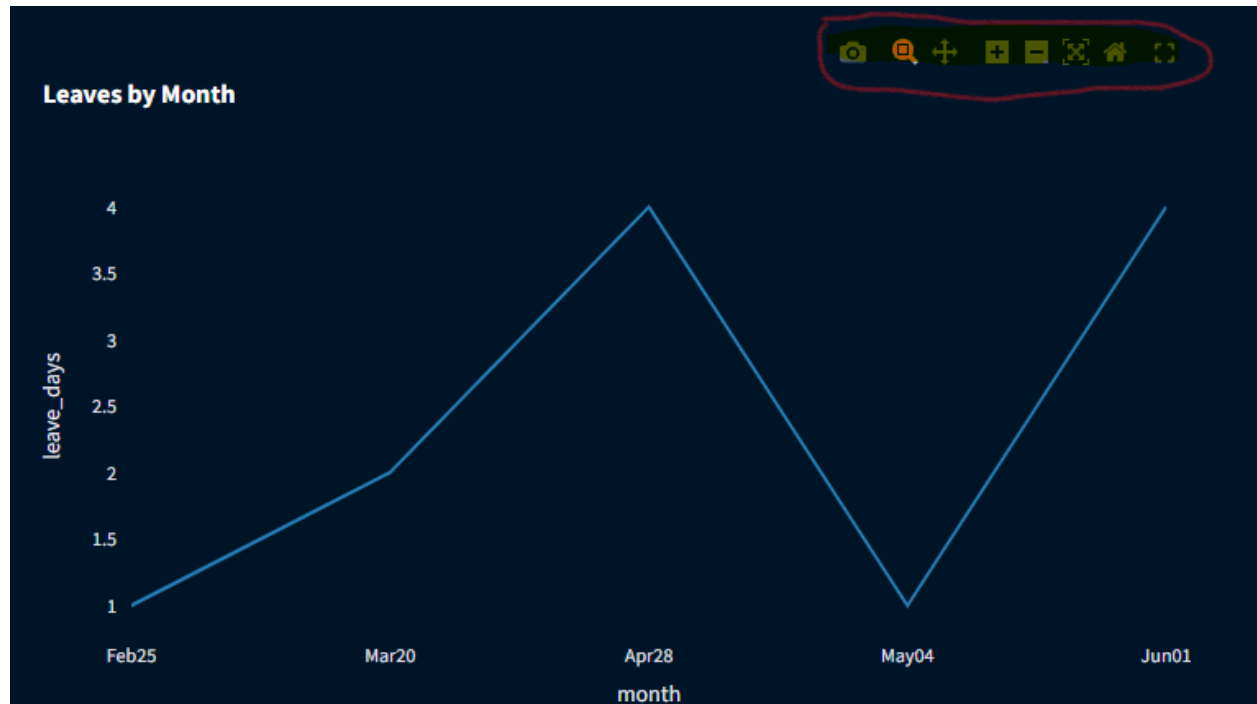
- Transactional Updates: When updating data in the SQL database, the codebase appears to use transactions (`async with conn.transaction()`) to ensure that updates are atomic and consistent. This helps maintain data integrity by either committing all changes or rolling back the entire transaction in case of failure.
- Error Handling: The codebase implements error handling mechanisms to handle exceptions gracefully. In case of any errors during data retrieval or insertion, appropriate error messages are logged (`logger.error`) to facilitate debugging and resolution.
- Data Validation: Before inserting new data into the database, the codebase performs data validation checks to ensure the quality and consistency of the data being integrated. This includes converting allocations to JSON strings and inserting each row using predefined SQL queries.

- **Logging:** Logging is extensively used throughout the codebase (`logger.info`, `logger.error`) to record information about data retrieval, insertion, and any errors encountered during execution. This aids in monitoring the system's behavior and diagnosing issues related to data consistency.

Overall, the codebase demonstrates a structured approach to handling real-time data updates, ensuring data consistency, and maintaining the integrity of the information stored and displayed by the system.

Customizable Visualization Options








Customization Features: The user can filter the chart on the basis of fiscal date, department, designation and leave type. There are different charts like line graph, bar graph, heatmap, gauge chart which are built using plotly providing better customization options like zoom in, zoom out, capturing photo, move image, autoscale, full screen layout etc.

Reporting and Insights

Employee Details

Download as CSV

	first_name	last_name	email	department_name	designation_name
32	Aakriti	Subedi	sizaadhikari@lfttechnology.com	Engineering	Lead Engineer

Allocation Details

	allocation_id	name	type
137	65783	Bench	None
228	1706	Development	area
231	1557	A Logo Project 002	project
251	1117	Business Development	area
468	1556	Daft Punk	project
485	1641	Adidas	project
780	65784	Product Design	area

The insights are provided by the different charts with ability to visualize them on the basis of available filters. User can download such visualizations for future reference. They can also download employee detailed information in csv format for future reference.

Fiscal Year Configuration



Please filter here

Select the Fiscal Date:

2023/2024 ×

2022/2023 ×

2021/2022 ×

Users can filter the chart on the basis of the fiscal year. This filter button retrieves the required data stored in a pandas dataframe on the basis of provided fiscal year.

Technical Requirements

Backend Technologies

1. Python

- Python is the primary programming language used to develop the application.
- Various Python modules and libraries are imported to facilitate different functionalities within the code.
- Asynchronous programming with ``asyncio`` is utilized to handle concurrent operations efficiently, particularly when making HTTP requests and interacting with the database asynchronously.

2. APIs

- HTTP APIs (RESTful):** The code interacts with HTTP APIs to fetch and insert data. Specifically, it uses the ``httpx.AsyncClient`` from the ``httpx`` library to make asynchronous HTTP GET requests to retrieve leave information.
- The application exposes its own API endpoints using the FastAPI framework (``FastAPI`` instance). These endpoints define the interface through which clients can interact with the application.
- FastAPI handles incoming HTTP requests and routes them to the appropriate functions for processing. For example, the ``/leave_info`` endpoint handles requests to fetch leave information, while the ``/insert_leave_info`` endpoint handles requests to insert leave data into the database.

3. Database Interaction

- The code interacts with a database using asynchronous PostgreSQL (``asyncpg``) to insert employee leave data into the database.
- Database connections are managed asynchronously using functions like ``connect_db`` and ``close_db`` from the ``utils.database`` module. These functions establish and close connections to the database, respectively.
- SQL queries are executed asynchronously using ``await conn.execute()`` within a transaction to ensure atomicity and data consistency during insertion.

Visualization Tool

1. Plotly Express (px)

- Plotly Express is a high-level interface for creating complex, interactive plots with Plotly. It offers a concise syntax for creating various types of visualizations.
- The `px.line`, `px.bar`, and other `px` functions are used to create line charts and bar charts with Plotly Express.
- These functions allow customization of the visualizations, including setting titles, colors, templates, etc.

2. Plotly Figures

- Plotly figures (`fig_monthly_sales`, `fig_dept_wise_leave`, `fig_leave_type_wise_leave`, `fig_designation_wise_leave`) are created using Plotly Express functions.
- These figures represent different aspects of leave data, such as monthly leave trends, total leaves by department, total leaves by leave type, and designation-wise total leave.

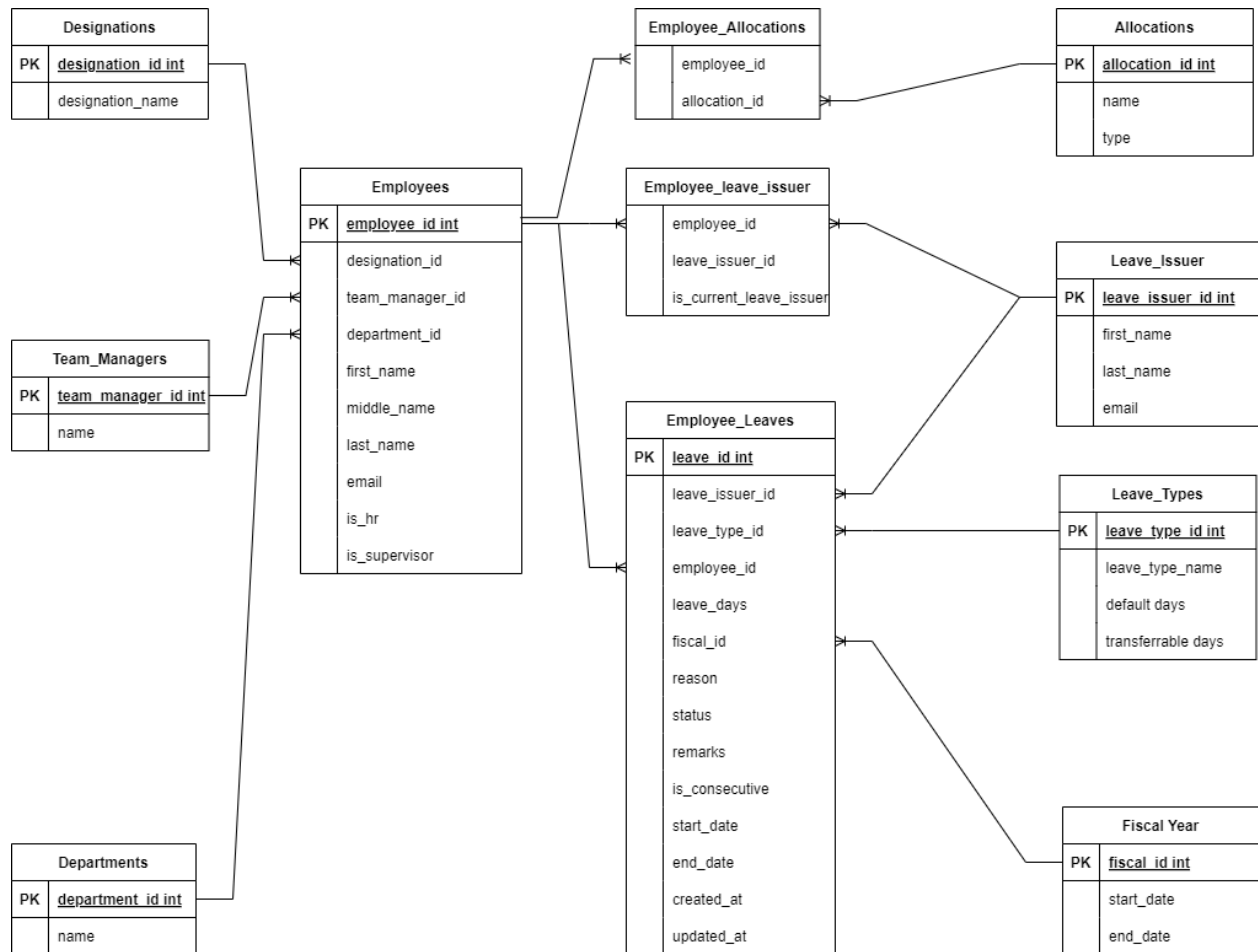
3. Displaying Visualizations

- The visualizations are displayed using Streamlit's `st.plotly_chart` function.
- The generated Plotly figures are passed as arguments to `st.plotly_chart` to render them within the Streamlit app.
- The visualizations are arranged in a grid layout using Streamlit's `st.columns` function, allowing multiple charts to be displayed side by side.

4. Interactive and Customizable Visualizations

- Plotly visualizations are interactive by default, allowing users to hover over data points, zoom in/out, pan, and more.
- Visualizations can be customized further by modifying parameters such as titles, colors, templates, etc., passed to Plotly Express functions.

Database



Here's the description of the database schema:

1. Tables

- allocations
 - Fields: allocation_id (Primary Key), name (varchar), type (varchar)
- designations
 - Fields: designation_id (Primary Key), designation_name (varchar)
- Team_managers
 - Fields: team_manager_id (Primary Key), name (varchar)

- departments
 - Fields: department_id (Primary Key), department_name (varchar)

- leave_issuer
 - Fields: leave_issuer_id (Primary Key), first_name (varchar), last_name (varchar), email (varchar)

- leave_types
 - Fields: leave_type_id (Primary Key), leave_type (varchar), default_days (int), transferrable_days (int)

- fiscal_year
 - Fields: fiscal_id (Primary Key), start_date (date), end_date (date)

- employees
 - Fields: employee_id (Primary Key), first_name (varchar), middle_name (varchar), last_name (varchar), email (varchar), is_hr (boolean), is_supervisor (boolean), designation_id (int, Foreign Key), team_manager_id (int, Foreign Key), department_id (int, Foreign Key)

- employee_allocations
 - Fields: employee_id (int, Foreign Key), allocation_id (int, Foreign Key)

- employee_leave_issuer
 - Fields: employee_id (int, Foreign Key), leave_issuer_id (int, Foreign Key), is_current_leave_issuer (boolean)

- employee_leaves
 - Fields: leave_id (Primary Key), leave_issuer_id (int, Foreign Key), leave_type_id (int, Foreign Key), employee_id (int, Foreign Key), fiscal_id (int, Foreign Key), leave_days (int), reason (text), status (varchar), remarks (text), is_consecutive (bit), start_date (date), end_date (date), created_at (date), updated_at (date)

2. Relationships

- One-to-Many Relationships
 - One designation can be associated with many employees. (employees.designation_id → designations.designation_id)
 - One team manager can manage multiple employees. (employees.team_manager_id → team_managers.team_manager_id)
 - One department can have multiple employees. (employees.department_id → departments.department_id)
 - One leave issuer can issue multiple leaves to employees. (employee_leave_issuer.leave_issuer_id → leave_issuer.leave_issuer_id)
 - One leave type can be associated with multiple leaves. (employee_leaves.leave_type_id → leave_types.leave_type_id)
 - One fiscal year can have multiple leaves associated with it. (employee_leaves.fiscal_id → fiscal_year.fiscal_id)
- Many-to-Many Relationships
 - Many employees can have multiple allocations. (employee_allocations.employee_id ↔ employees.employee_id, employee_allocations.allocation_id ↔ allocations.allocation_id)

3. Data Types

- Primary Key Fields: int
- Varchar Fields: varchar
- Text Fields: text
- Boolean Fields: boolean
- Integer Fields: int
- Date Fields: date

The database schema is normalized to some extent, aiming to reduce data redundancy and improve query efficiency. Here's how normalization has been applied:

1. First Normal Form (1NF):

- Each table has a primary key that uniquely identifies each record.
- There are no repeating groups or arrays within any fields.
- Each field contains atomic values.

2. Second Normal Form (2NF):

- All non-key attributes are fully functional dependent on the primary key.
- There are no partial dependencies.

3. Third Normal Form (3NF):

- There is no transitive dependency between non-key attributes.
- All attributes are functionally dependent on the primary key.

4. Further Normalization:

- Additional tables like `employee_allocations`, `employee_leave_issuer`, and `employee_leaves` are used to handle many-to-many relationships, ensuring that no data redundancy occurs when associating employees with their allocations, leave issuers, and leaves.

- The `fiscal_year` table separates fiscal year data, avoiding redundancy when multiple leaves are associated with the same fiscal year.

- The use of foreign key constraints ensures referential integrity, preventing inconsistencies and data anomalies.

By adhering to these normalization principles, redundant data is minimized, and the database structure is optimized for efficient querying and data maintenance. This normalization approach helps in avoiding anomalies such as insertion, update, and deletion anomalies, ensuring data integrity and consistency across the database.

API Documentation

Overview

This API provides endpoints for fetching leave information and inserting leave data into the database. It includes a background task that periodically updates the leave data.

Authentication

All endpoints require a Bearer token for authorization. The token should be included in the `Authorization` header of the requests.

Endpoints

1. Get Leave Information

Endpoint: `GET /leave_info`

Description: Fetches leave information from an external service.

Request:

- Headers:

- **Authorization:** `Bearer <token>` (string, required)

Response:

- **200 OK:** JSON object containing leave information.
- **401 Unauthorized:** If the provided token is invalid.

Example Request:

```
curl -X GET "http://127.0.0.1:8000/leave_info" -H "Authorization: Bearer YOUR_TOKEN"
```

Example Response:

```
{
  "data": [
    {
      "leave_id": 1,
      "employee_id": 123,
      "leave_days": 5,
      "leave_type": "Sick",
      ...
    },
    ...
  ]
}
```

2. Insert Leave Information

Endpoint: `GET /insert_leave_info`

Description: Fetches leave information and inserts it into the database.

Request:

- **Headers:** None

Response:

- **200 OK:** JSON object indicating success.
- **500 Internal Server Error:** If there is an issue with inserting the data.

Example Request:

```
curl -X GET "http://127.0.0.1:8000/insert_leave_info"
```

Example Response:

```
{"success": "Leave Data Inserted Successfully!"}
```

3. Health Check

Endpoint: `GET /`

Description: Simple health check endpoint to ensure the API is running.

Request:

- **Headers:** None

Response:

- **200 OK:** JSON object indicating success.

Example Request:

```
curl -X GET "http://127.0.0.1:8000/"
```

Example Response:

```
{  
  "success": true  
}
```

Background Task

A background task runs periodically to update the leave data.

Description: The task fetches leave information and inserts it into the database every 10 seconds.

Helper Functions

1. `get_leave_info(bearer_token)`

Fetches leave information from an external service using the provided Bearer token.

- **Parameters:** `bearer_token` (string, required)
- **Returns:** JSON object containing leave information
- **Raises:** `HTTPException` with status code 401 if unauthorized

2. insert_leave_data(data, conn)

Inserts the provided leave data into the database.

- **Parameters:**
 - **data** (list of dicts, required): Leave data to be inserted
 - **conn** (asyncpg connection, required): Database connection
- **Returns:** None
- **Raises:** Exception if there is an issue with insertion

3. run_insert_leave_info()

Combines fetching leave information and inserting it into the database.

- **Parameters:** None
- **Returns:** JSON object indicating success
- ***Raises:** `HTTPException` with status code 500 if there is an issue

Error Handling

The API uses `HTTPException` to handle errors. Relevant status codes and error messages are returned to the client for unauthorized access and server errors.

Logging

The API uses a logger to log information and errors throughout the code. Logs are generated for successful operations and error scenarios to assist in debugging and monitoring.

Usage

To use the API, start the FastAPI server and make requests to the endpoints as documented above. Ensure to include the necessary headers for authentication where required.

Running the Server

To run the server, execute the following command:

```
uvicorn main:app --host 127.0.0.1 --port 8000 --reload
```

Replace `main` with the name of your Python file if it is different.

Error Mechanism

Error Handling in the Backend

1. Exception Handling with HTTPException:

- The code uses HTTPException from FastAPI to handle errors that occur during API requests.
- When an error occurs, the code raises an HTTPException with a specific status code and detail message.

2. Error Codes and Messages:

- **401 Unauthorized:** This status code is used when authentication fails or when the request does not have proper authorization headers.
- **500 Internal Server Error:** This status code is used for unexpected errors that occur during the execution of the code.

3. Logging Mechanism:

- The code uses a logger, configured through get_logger() from utils.logging, to log information, warnings, and errors.
- **logger.info()** is used for logging informational messages.
- **logger.error()** is used for logging error messages.

,

Detailed Explanation

1. Retrieving Leave Information

```
async def get_leave_info(bearer_token=DEFAULT_BEARER_TOKEN):
    async with httpx.AsyncClient() as client:
        headers = {"Authorization": f"Bearer {bearer_token}"}
        response = await client.get(URL, headers=headers)
        if response.status_code == 200:
            logger.info("Leave information successfully retrieved")
            return response.json()
        else:
            logger.error(
                "Failed to retrieve leave information. Status code: %d",
                response.status_code,
            )
            raise HTTPException(status_code=401, detail="Unauthorized")
```

- **Logging Success:** If the request to retrieve leave information is successful (status_code == 200), it logs an informational message.

- **Logging Failure:** If the request fails (any status code other than 200), it logs an error message with the status code and raises an HTTPException with a 401 status code and "Unauthorized" detail message.

2. Inserting Leave Data

```
async def insert_leave_data(data, conn):
    async with conn.transaction():
        await conn.execute("TRUNCATE TABLE
raw.imported_leave_information;")
        for row in data:
            if row["allocations"] is not None:
                row["allocations"] = json.dumps(row["allocations"])
            await conn.execute(insert_query, *row.values())
```

- **Error Handling:** This function doesn't have explicit error handling within it, but any error during the transaction would be caught by the run_insert_leave_info() function.

3. Running Insert Leave Info

```
async def run_insert_leave_info():
    try:
        conn = await connect_db()
        json_data = await get_leave_info()
        data = json_data["data"]
        await insert_leave_data(data, conn)
        with open("../db/procedures.json") as f:
            proc_steps = json.load(f)
        async with conn.transaction():
            for step in proc_steps["steps"]:
                await conn.execute(f'CALL {step["proc"]}();')
        await conn.close()
        logger.info("Leave Data Inserted Successfully!")
        return {"success": "Leave Data Inserted Successfully!"}
    except Exception as e:
        logger.error(f"Couldn't insert the leave data! {e}")
        raise HTTPException(
            status_code=500, detail=f"Couldn't insert the leave data! {e}"
        )
```

- **Try Block:** The main operations are wrapped in a try block to catch any exceptions that occur during the process.
- **Logging Success:** If the operation completes successfully, it logs an informational message.
- **Handling Exceptions:** If an error occurs:
 - It logs the error message.
 - Raises an HTTPException with a 500 status code and the error message as the detail.

4. Background Task

```
async def background_task():
    while True:
        await run_insert_leave_info()
        await asyncio.sleep(10)    Sleep for 60 seconds
```

- **Continuous Execution:** This function continuously runs run_insert_leave_info() in a loop with a sleep interval.
- **Error Handling:** Errors within run_insert_leave_info() are handled as described above.

5. FastAPI Routes

```
@app.get("/leave_info")
async def leave_info(authorization: str = DEFAULT_BEARER_TOKEN):
    try:
        bearer_token = authorization.replace("Bearer ", "", 1)
        return await get_leave_info(bearer_token)
    except Exception as e:
        raise HTTPException(status_code=401, detail="Unauthorized")
```

- **Authorization Handling:** If the authorization token is invalid or an error occurs in `get_leave_info`, it raises an `HTTPException` with a 401 status code and "Unauthorized" detail message.

```
@app.get("/insert_leave_info")
async def insert_data():
    try:
        return await run_insert_leave_info()
    except HTTPException as e:
        return {"error": e.detail}
    except Exception as e:
        return {"error": str(e)}
```

- **Error Handling:** If an error occurs in `run_insert_leave_info()`, it catches both `HTTPException` and general `Exception` to return appropriate error messages.

Summary of Error Handling

- **HTTPException:** Used to handle and respond to client errors with appropriate status codes and messages.
- **Logging:** Uses `logger.info()` for success messages and `logger.error()` for error messages, providing insights into the application's behavior and issues.
- **Try-Except Blocks:** Wraps critical sections of code to catch and handle exceptions, ensuring the application can respond gracefully to errors.

This structured approach to error handling ensures that errors are logged, appropriate HTTP status codes are returned, and detailed error messages are provided, making the system robust and easier to debug.

Error Handling in the Dashboard

In the Streamlit and Plotly code for generating and displaying visualizations, error handling is implemented primarily using try-except blocks along with logging mechanisms. Here's a detailed explanation of how errors and exceptions are handled in this system:

Error Handling Mechanisms

1. Try-Except Blocks:

The `generate_visualizations` function contains a try-except block to catch any exceptions that occur during the data processing and visualization generation process.

If an exception is caught, the error is logged, and the function returns a tuple of `None` values for the visualizations, signaling that an error occurred.

2. Logging:

The code utilizes a logging mechanism to log informational messages and errors.

The logger is initialized using `get_logger("Dashboard-Home")`, and it logs messages with different levels (info and error) to help with debugging and monitoring the application's behavior.

3. Conditional Checks:

After generating the visualizations, there is a conditional check to ensure that all visualizations were generated successfully (i.e., none of them are `None`).

If any visualization is `None`, it logs an error message indicating that there was an issue generating the visualizations.

Error Codes and Messages

- **Error Codes:**

The code does not directly use HTTP status codes since it is a Streamlit application rather than a traditional web API. Instead, it uses logging and conditional checks to handle errors and provide feedback.

- **Error Messages:**

Descriptive error messages are logged when exceptions occur. These messages include information about the nature of the error and are useful for debugging purposes.

Logging Mechanisms

- **Initialization:**

The logger is initialized with a specific name ("Dashboard-Home") to distinguish logs from different parts of the application.

- **Info Logging:**

Informational messages are logged using `logger.info()`. For example, when visualizations are generated successfully or displayed successfully, an info message is logged.

- **Error Logging:**

Error messages are logged using `logger.error()`. These messages include the error details and provide insights into what went wrong.

Security Considerations

The system ensures that errors are handled gracefully, logged appropriately, and the system can provide meaningful feedback to users and developers.

Several measures to protect sensitive data and ensure secure handling of operations have been included, particularly around the retrieval and insertion of leave information data. Here are the key security measures implemented:

1. Use of Bearer Tokens for Authorization:

- The ``get_leave_info`` function uses a Bearer token for authorization. The token is included in the request headers to authenticate the request.
- The ``leave_info`` endpoint extracts and uses the Bearer token provided in the request header for secure access.

```
headers = {"Authorization": f"Bearer {bearer_token}"}
```

2. Default Bearer Token:

- A default Bearer token (`DEFAULT_BEARER_TOKEN`) is used when none is provided, ensuring that a token is always used for requests to the external service.

3. Secure HTTP Requests:

- The `httpx.AsyncClient` is used for making HTTP requests, which supports asynchronous operations and provides secure methods to handle HTTP interactions.

4. Exception Handling and Logging:

- The code includes detailed exception handling and logging to track errors and unauthorized access attempts.

```
if response.status_code == 200:
    logger.info("Leave information successfully retrieved")
else:
    logger.error("Failed to retrieve leave information. Status code: %d", response.status_code)
    raise HTTPException(status_code=401, detail="Unauthorized")
```

5. Database Transactions:

- The use of database transactions ensures that operations are atomic. If an error occurs, the transaction can be rolled back to maintain data integrity.

```
async with conn.transaction():
```

6. Truncating Sensitive Data Before Insert:

- Before inserting new data, the `TRUNCATE TABLE` command is used to clear the existing data, which ensures that no stale or redundant data remains.

```
await conn.execute("TRUNCATE TABLE raw.imported_leave_information;")
```

9. Endpoint Security:

- The `leave_info` and `insert_data` endpoints include error handling to manage unauthorized access and other exceptions securely.

```
except Exception as e:
    raise HTTPException(status_code=401, detail="Unauthorized")
```

Performance Optimization

The system includes several performance optimization techniques to ensure efficient execution of tasks. Here are the key performance optimization measures implemented:

Asynchronous Programming:

The use of asynchronous programming (with `async` and `await`) in FastAPI and database operations allows for non-blocking execution. This helps in handling multiple I/O-bound operations concurrently without waiting for each to complete sequentially, thus improving overall performance.

Connection Pooling with `asyncpg`:

The use of `asyncpg`, an asynchronous PostgreSQL client library, is known for its performance benefits, including efficient connection pooling. This reduces the overhead of establishing new database connections repeatedly.

Batch Data Insertion:

In the `insert_leave_data` function, the code truncates the table and then iterates through the data to insert each row. While this isn't explicitly using batch insertion, the transaction block (`async with conn.transaction()`) ensures that multiple insert operations are treated as a single unit of work, reducing the overhead associated with committing each insert operation separately.

Efficient Use of Transactions:

Using transactions for both the insertion of data and the execution of stored procedures ensures that all operations are done in a batch manner, which can be more efficient than executing each operation independently.

Scheduled Background Task:

The `background_task` function runs the `run_insert_leave_info` function in a loop with a delay (`await asyncio.sleep(10)`). This periodic execution ensures that data retrieval and insertion tasks are performed regularly without blocking other operations. The sleep interval can be adjusted based on the system's performance needs.

Asynchronous HTTP Requests:

The use of `httpx.AsyncClient` for making HTTP requests allows multiple HTTP requests to be made concurrently, improving the performance of I/O-bound tasks like fetching leave information from an external service.

Lazy Loading of Configuration:

Configuration data, such as the procedures to run, are loaded only when needed. For instance, the procedures are loaded from a JSON file right before they are executed. This avoids unnecessary loading of configuration data, thereby saving memory and initialization time.

Efficient Error Handling:

Exception handling is implemented to catch and log errors promptly. This not only helps in debugging but also ensures that the application can recover quickly from errors without crashing, thus maintaining performance stability.

By combining these techniques, the code ensures that tasks are executed efficiently, resources are utilized optimally, and the application remains responsive even under load.

Load Testing With Locust

Locust is an open-source load testing tool used to simulate user traffic on web applications and APIs. It allows developers to test the performance and scalability of their systems by creating scenarios where multiple users interact with the application simultaneously. Locust is highly customizable, easy to use, and written in Python, making it a popular choice for performance testing.

Running the Test

To run the Locust test, follow these steps:

Start Your Flask App:

1. Ensure your Flask app is running. You might do this from the command line with:

```
uvicorn main:app --reload
```

Or if it's part of the script, ensure the app is started appropriately before running Locust.

2. Start Locust:

Use the command line to start Locust with your test script:

```
locust -f locust_test.py
```

3. Locust Web Interface:

Open a web browser and navigate to <http://localhost:8089> (default Locust web interface URL).

4. Configure and Start the Test:

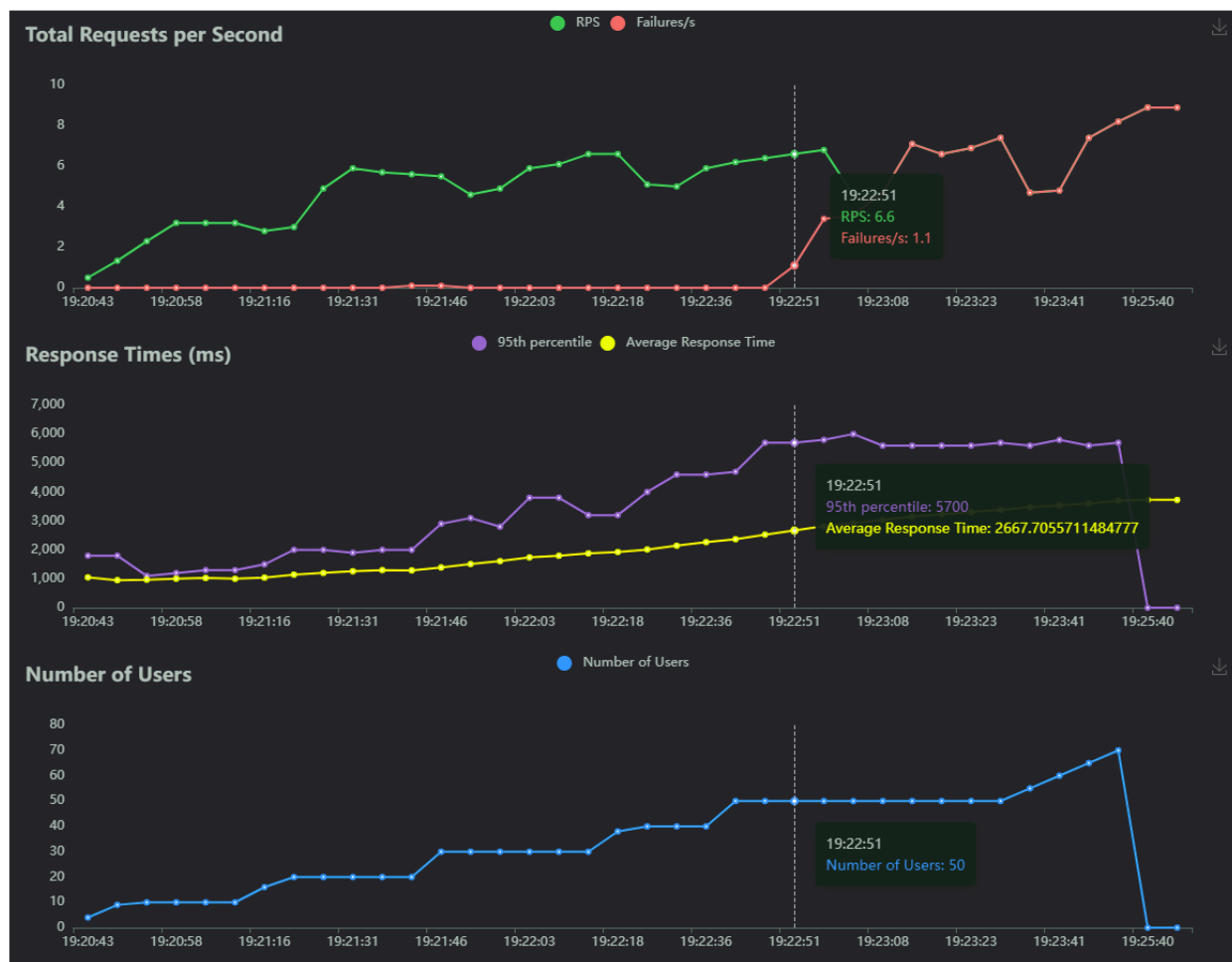
- Number of Users to Simulate: Enter the total number of concurrent users you want to simulate.
- Spawn Rate: Enter how many new users to start per second.
- Click the Start Swarming button.

Locust will gradually start 10 users per second until it reaches 100 concurrent users. Each user will perform the `get_leave_info` task, sending GET requests to `/leave_info` and waiting between 1 to 3 seconds between each request.

Monitoring Results

- RPS (Requests Per Second): View how many requests per second your application is handling.
- Response Times: Check the response times for the requests.
- Failures: Monitor any request failures to identify potential issues.
- Charts: Visualize the performance metrics over time.

This setup helps you identify performance bottlenecks, test your application's scalability, and understand how it behaves under load. Our system can handle 50 users with 15 users spawned at a time for fetching the leave data.



Database Migration

A migration procedure for a database, with the capability to either apply migrations (set up the database) or roll them back (clean up the database) is in place. This is facilitated through the command-line arguments **--up** and **--down**.

Migration Down (``migration_down`` function)

The ``migration_down`` function is designed to clean up the database by removing specific schemas.

- 1. Schemas to Drop:** The script specifies two schemas: ``"raw"`` and ``"dbo"``.
- 2. Database Connection:** It establishes a connection to the database using ``connect_db()``.
- 3. Schema Deletion:** For each schema, the function executes a SQL command to drop the schema if it exists, along with all its contained objects (using ``CASCADE``).
- 4. Logging and Output:** The function logs a message indicating the database has been cleaned and prints the same message to the console.
- 5. Commit and Close:** It commits the transaction to ensure the changes are saved and then closes the database connection using ``close_db``.

Migration Up (``migration_up`` function)

The ``migration_up`` function is designed to apply database migrations by executing SQL scripts located in specified directories.

- 1. Directories to Scan:** The script looks for SQL files in two directories: ``./sql/migrations`` and ``./sql/procedures``.
- 2. Database Connection:** It establishes a connection to the database using ``connect_db()``.
- 3. SQL Execution:** For each SQL file in the specified directories:
 - It reads the content of the file.
 - Executes the SQL command.
 - If successful, it logs a success message and prints it to the console.
 - If there's an error during execution, it rolls back the transaction, logs the error, and continues with the next file.

4. Commit and Close: It commits the transaction after each successful execution of an SQL file and then closes the database connection using **close_db**.

Command-Line Interface

The script uses **argparse** to provide a command-line interface with two options:

- **--up**: Triggers the **migration_up** function to apply the migrations.
- **--down**: Triggers the **migration_down** function to clean the database.

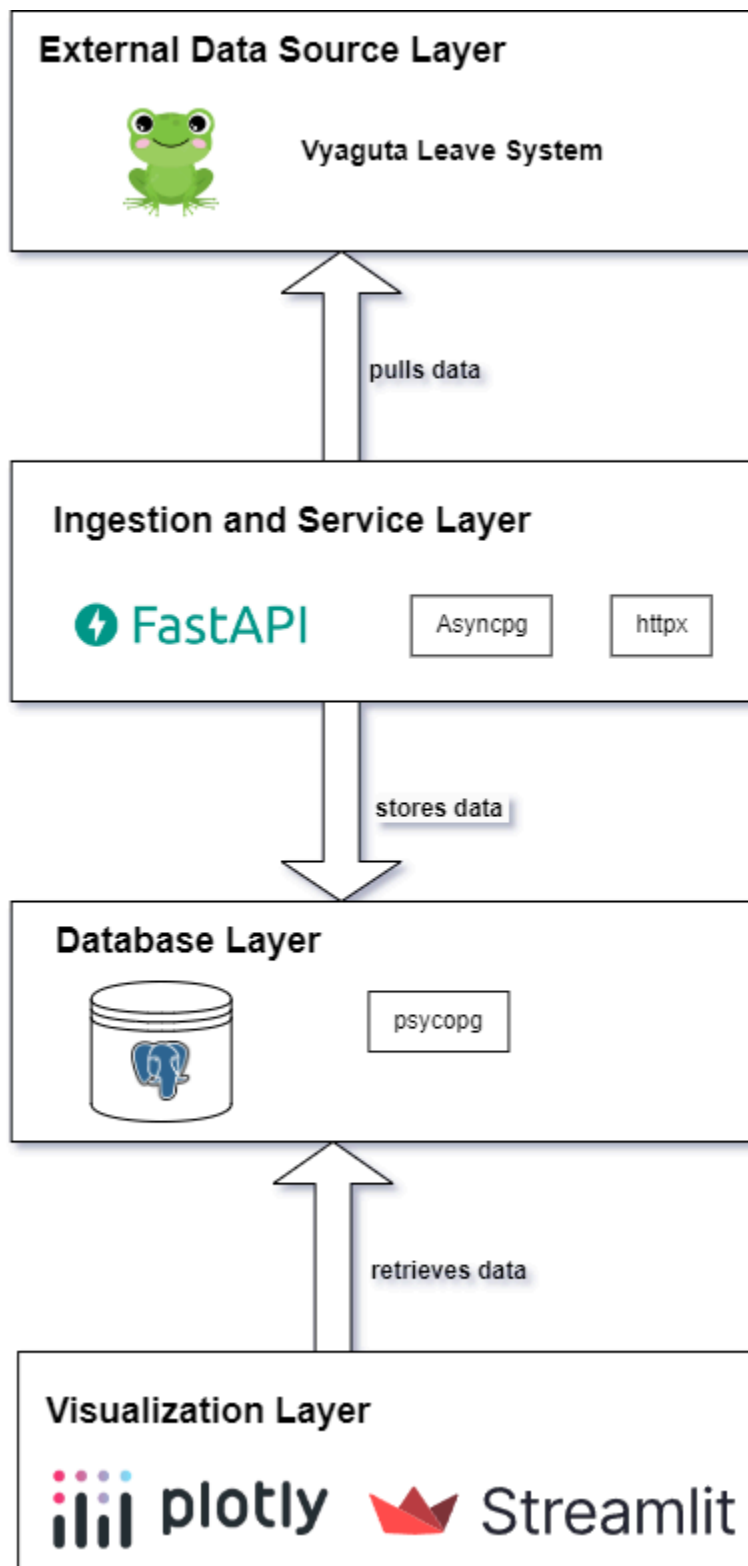
Depending on the command-line argument provided when running the script, it will either apply the migrations or roll them back.

Summary

- **Migration Down**: Drops the specified schemas (**raw** and **dbo**) and all their contents from the database.
- **Migration Up**: Executes all SQL scripts in the **./sql/migrations** and **./sql/procedures** directories to set up the database.
- **Command-Line Options**: **--up** to apply migrations, **--down** to roll them back.

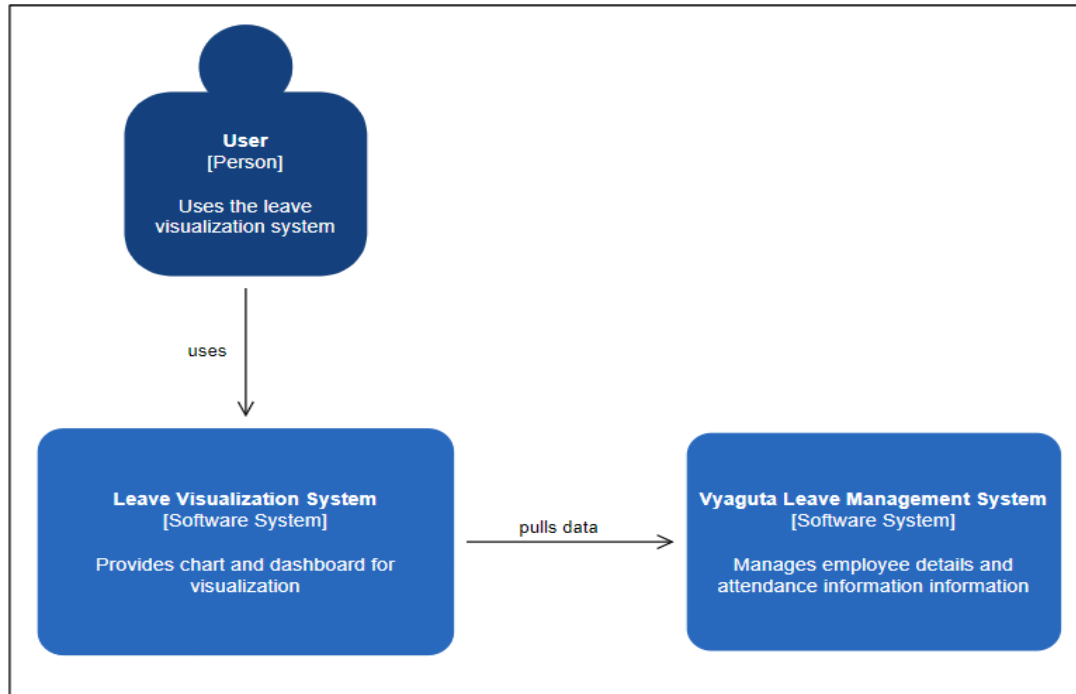
This approach allows for easily managing database schema changes and ensures that the database can be reset or updated with new changes as needed.

Architecture Diagram

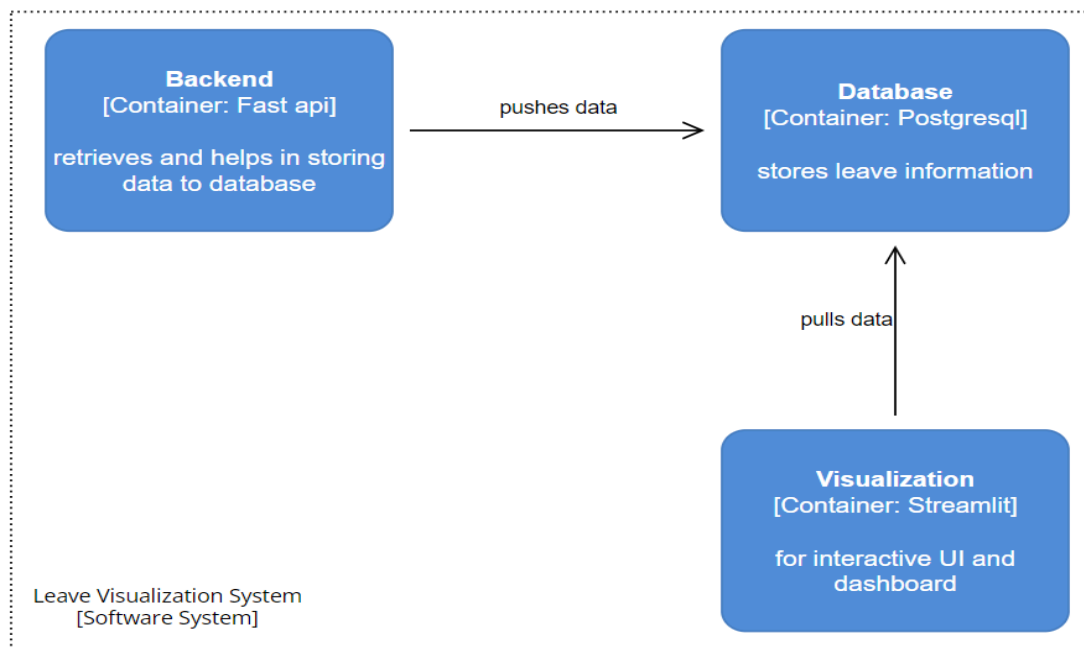


C4 Diagram

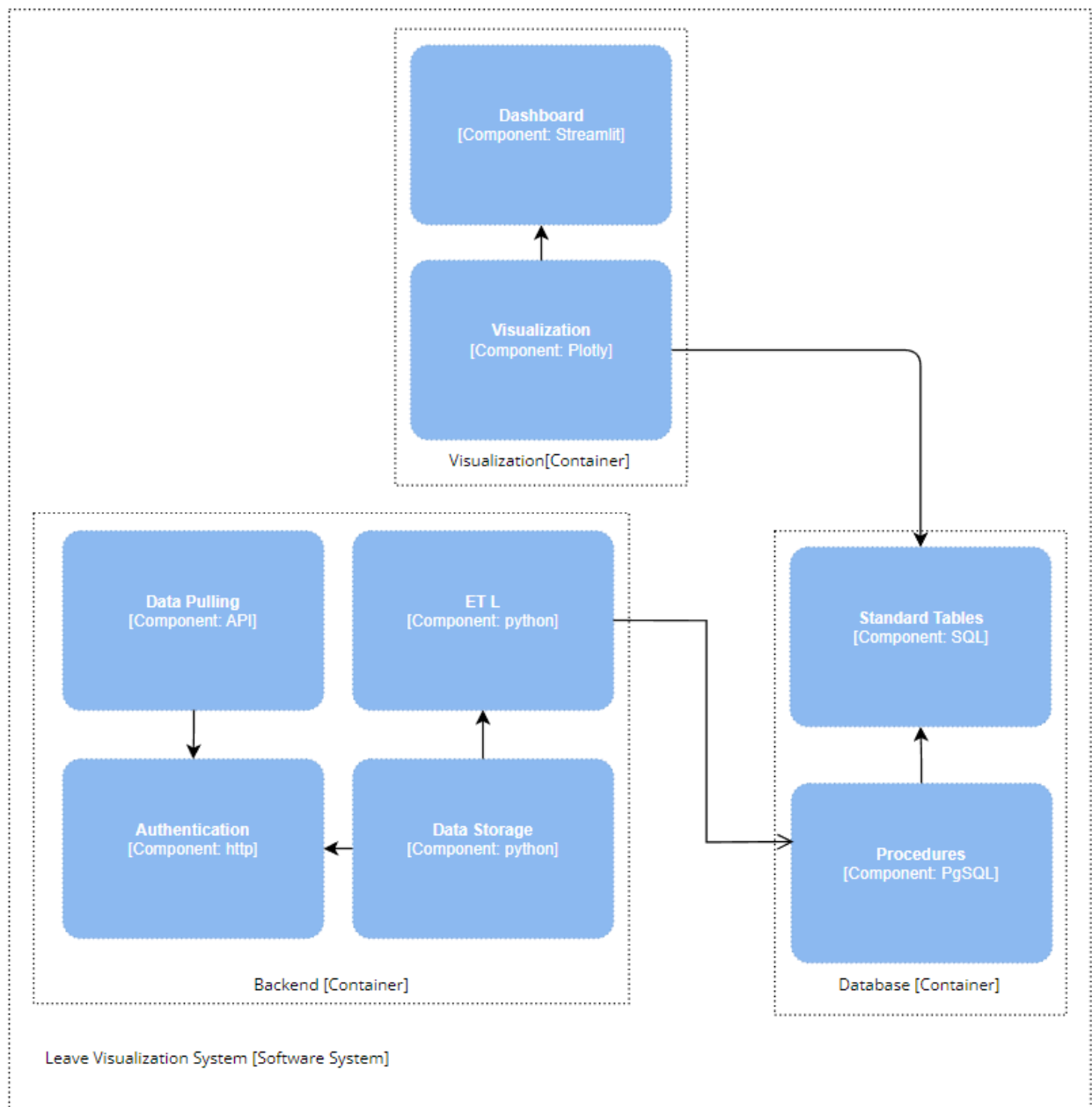
Context Diagram



Container Diagram



Component Diagram



Testing and Quality Assurance

I have used pytest for testing the database migration and backend endpoints.

Pytest DB Migration

```
ubuntu@LF-00002371:~/leapfrog_leave_assessment/src/db$ pytest test_migration.py -v
===== test session starts =====
platform linux -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/ubuntu/leapfrog_leave_assessment/src/db
plugins: asyncio-0.23.7, Faker-25.2.0, anyio-4.4.0
asyncio: mode=Mode.STRICT
collected 2 items

test_migration.py::test_migration_up PASSED
test_migration.py::test_migration_down PASSED

===== 2 passed in 0.19s =====
```

Pytest Fast API

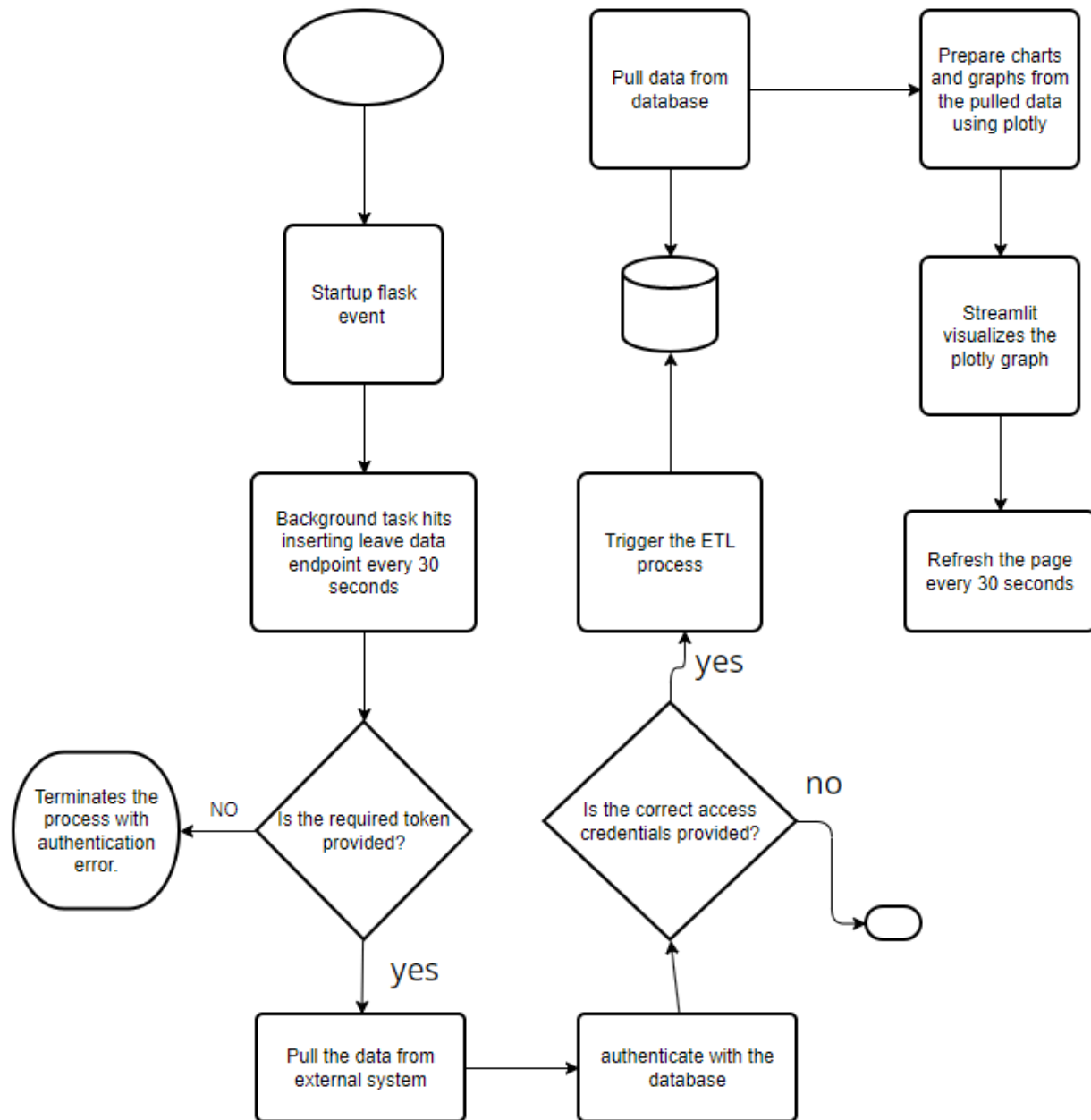
```
===== 2 passed, 1 warning in 0.61s =====
ubuntu@LF-00002371:~/leapfrog_leave_assessment/src/backend$ pytest test_flask.py -v
===== test session starts =====
platform linux -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/ubuntu/leapfrog_leave_assessment/src/backend
plugins: asyncio-0.23.7, Faker-25.2.0, anyio-4.4.0
asyncio: mode=Mode.STRICT
collected 2 items

test_flask.py::test_index PASSED
test_flask.py::test_leave_info PASSED

===== warnings summary =====
test_flask.py::test_leave_info
  /home/ubuntu/.local/lib/python3.12/site-packages/httpx/_client.py:1426: DeprecationWarning: The 'app' shortcut
instead.
    warnings.warn(message, DeprecationWarning)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
===== 2 passed, 1 warning in 0.59s =====
ubuntu@LF-00002371:~/leapfrog_leave_assessment/src/backend$
```

Flow Chart Diagram



Appendix

Glossary of Terms and Abbreviations

- API: Application Programming Interface, a set of functions and procedures allowing the creation of applications that access features or data of an operating system, application, or other service.
- ETL: Extract, Transform, Load; a process in database usage and data warehousing.
- FastAPI: A modern, fast (high-performance) web framework for building APIs with Python 3.6+ based on standard Python type hints.
- PostgreSQL: An open-source relational database management system emphasizing extensibility and SQL compliance.
- Streamlit: An open-source app framework for Machine Learning and Data Science teams.
- Plotly: A graphing library that makes interactive, publication-quality graphs online.
- Docker: A set of platform-as-a-service products that use OS-level virtualization to deliver software in packages called containers.
- Asyncio: A library to write concurrent code using the `async/await` syntax in Python.
- HTTPException: An error response in HTTP, which stands for HyperText Transfer Protocol.
- SQL: Structured Query Language, used in managing and manipulating relational databases.

References to External Documentation, APIs, Libraries, or Frameworks Used

- FastAPI Documentation: [FastAPI](#)
- PostgreSQL Documentation: [PostgreSQL](#)
- Streamlit Documentation: [Streamlit](#)
- Plotly Documentation: [Plotly](#)
- Docker Documentation: [Docker](#)
- Asyncio Documentation: [Asyncio](#)
- Locust Documentation: [Locust](#)

Additional Resources or Supporting Materials

- Environment Configuration: Instructions for setting up environment variables can be found in `.env.example` files provided in both backend and database modules.
- Error Handling Mechanisms: The project implements error handling through `HTTPException`, generic `Exception` handling, and logging error messages using `logger.error()`.
- Real-time Data Updates: Data fetching from external API (Vyaguta) every 30 seconds, storing in PostgreSQL, and using `asyncio` for asynchronous operations .

Coding Standards Followed in the Project

- PEP 8: The project follows PEP 8 - the Style Guide for Python Code.
- Modular Code Structure: Organized using directories like `src`, `backend`, `db`, `utils`, and `visualization` to maintain separation of concerns.
- Asynchronous Programming: Utilized `asyncio` for efficient concurrent operations.
- Error Handling: Implemented through `HTTPException` and generic exception handling with proper logging.