Urban-MobilityApp — Reflective Technical Documentation

1. Introduction

Urban-MobilityApp is a Flask web application that communicates with a MySQL database for processing, storing, and serving urban mobility data. It provides an API endpoint (/api/trips) that returns trip records in JSON for visualization or further analysis.

Our team created this system with the aim to gain more experience in managing urban trip data effectively using Python, Flask, and relational databases with considerations for backend organization, data cleaning, and deployment procedures.

This document is intended to provide a reflective, detailed, and justified description of our decisions, architecture, results, and issues found through the project.

2. Project Objectives

The primary goal was to develop a lean yet functional data handling system capable of:

Cleaning raw trip data (train.csv),

Pleasantly store it in a MySQL database with proper structure,
Expose it via a RESTful Flask API for easy access,
Make it reproducible and sustainable with open setup processes.

Emphasis was laid upon readability, good collaboration with peers, and technical justification behind every major design decision.

3. System Architecture and Rationale for Design

3.1 High-Level Overview of Architecture

The system architecture is comprised of five main components:

Data Cleaning Layer – cleaning_train_data.py
Processes and cleans the raw data (train.csv) to a formatted file cleaned_trips_data.csv.

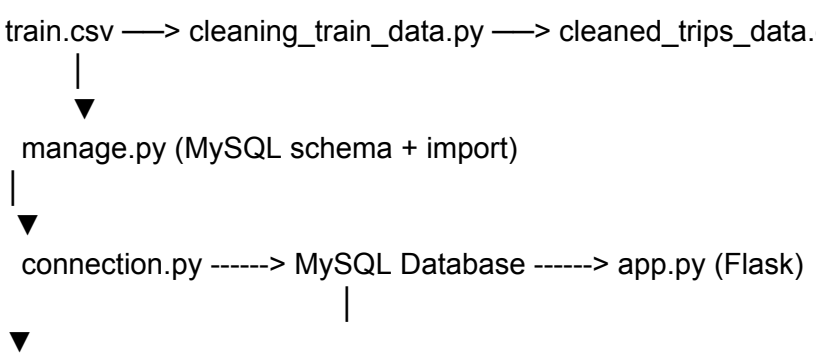
Database Management Layer – manage.py
Is responsible for creating the database schema, establishing tables, and inserting the cleaned data into MySQL.

Database Connection Configuration – connection.py
Stores MySQL connection details (to be replaced later by environment variables for security reasons).

Application Layer – app.py & views.py
The Flask web server delivering up routes and API endpoints.
The /api/trips endpoint delivers up trip information in JSON.

Frontend/Consumer Layer –
Though simple now, the exposed API can be consumed by dashboards or visualization software in the future.

Flow Diagram (Conceptual):



3.2 Architecture Motivation

We deliberately selected this modular design to:

Separate things: Each module has only one, clearly defined function.

Make testing simpler: Independent scripts are simpler to debug.

Improve scalability: Further work (e.g., dashboards or analysis) can be added easily to the /api/trips endpoint.

Make reproducible: The data cleansing script creates a permanent cleaned_trips_data.csv version, so environments are identical.

4. Installation and Setup (Reproducing Our Workflow)
1.
Clone the Repository

```
git clone <repository-url>
cd Urban-MobilityApp
```

Produce and Provoke Virtual Environment.

```
python -m venv venv
source venv/bin/activate       # Linux/MacOS.
venvScriptsactivate  #Windows.
```

Install Dependencies
by Requirement.

pip install requirements.txt.

Clean the Raw Data

Make sure that train.csv is in the same directory as cleaning train data.py.

python cleaning_train_data.py

This gives an output of a clean version: tripped cleaned.csv.

Set Up Database Connection

Enter your MySQL passwords in:

connection.py

manage.py (lines 10–20)

Run the database setup:

python manage.py

Run the Flask Application

python app.py

App runs on port 3000 by default.

Main endpoint: http://127.0.0.1:3000/api/trips

Virtual Environment must be Active.

source venv/bin/activate

5. Technical Decisions and justifications.

Decision	Justification
With Flask	Lightweight, simple to interface with MySQL, ideal with REST APIs.
MySQL Database	Structured, relational and good at supporting tabular trip data with distinct relationships.
csv-based Batch Processing	Enhances reproducibility of data and easy debugging; suitable with small to mid-size datasets.
Separate Cleaning and Import Scripts	Encourages modularity; all data cleaning mistakes will not impact imports in the database.

Flask port 3000 The port avoids any conflict with usual ports (i.e., 5000 or 8080).
JSON Endpoint (/api/trips) Can be read universally and can be simply combined with a web frontend or data visualization tool.

6. Insights and Reflections

6.1 Team thinking and Decision Process.

We wanted to develop an architecture that was clean and comprehensible as opposed to a product that was feature packed.
Every decision was arrived at after consideration of:

Simplicity vs. Scalability

Short-term vs. Long-term.

Quick onboarding of new developers.

6.2 Key Insights Gained

The system is based on modular code, allowing collaboration through independent testing.

Silent data corruption is avoided by data validation during the cleaning process.

A version control is improved when the cleaned data is stored as CSV prior to importing them into the database.

Onboarding of the developer is made easy by explicit documentation (README + scripts).

7. Difficulties and The way we worked out.

Challenge Solution
Connection problems with the database (e.g. port or credential mismatch) Clear instructions that are documented to make changes to connection.py and manage.py and verified MySQL was running on localhost.
Slow import time in CSV Large CSVs Imported in a batch Large CSVs Imported in bulk with MySQL bulk import (LOAD DATA LOcal INFILE) Large CSVs Imported in bulk with MySQL bulk import (LOAD DATA Local INFILE) Large CSVs Imported in bulk with MySQL bulk import (LOAD DATA Local INFILE) Large CSVs Imported in bulk with MySQL bulk import (LOAD DATA Local INFILE) Large CSVs Imported in bulk with MySQL bulk import (LOAD DATA Local INFILE) Large CSVs Imported in bulk with MySQL
Credential exposure risk As was observed, security issues need attention; the next version will use environment variables to represent the hardcode credentials.
Inconsistencies in data cleaning Added validation checks and skipped or malformed row log files.
Mismatch in team environment The use of virtual environments (venv) and requirements file was encouraged.

8. Lessons Learned

Teamwork can be enhanced by proper organization: The module boundaries were clear thus preventing any mix-ups between the cleaning, importing and running phases.

Repeatability: A snapshot of the data (cleaned trips data.csv) was cleaned, which provided everyone with consistent data.

Error logs are essential: It should not simply drop bad rows, but log them, which will give it transparency and debugging ability.

The code is not all there is: Detailed README and reflective reporting improved project maintainability.

9. Future Improvements

Configure Hardcoded credentials with environment variables with .env.

Set up the project using Docker Compose to provide a consistent setup.

Install API authentication (JWT tokens or API keys).

Add API to allow filtered queries (e.g., start date).

Add a basic data visualization dashboard.

Test using GitHub Actions.

10. Conclusion

The Urban-MobilityApp signifies how our team can bridge many levels of a data-driven system: data cleaning, database management and RESTful service exposure.
The choices that have been undertaken during the project are very balanced as the simplicity of the project is practical yet can be scaled to the future.

The process of reflection exposed us to not only the process of how to create such systems, but also why some of these architectural patterns (such as separation of concerns, and modular scripts) are so critical in the real world.

Appendix: Key Files Overview

File	Purpose
app.py	Launches the Flask application and opens routes.
views.py	The API endpoints such as /api/trips are there.
manage.py	Deals with database creation and CSV importation.
connection.py	Connection configuration to MySQL.
cleaning_train.py	Fills in and cleans raw CSVs and produces a clean dataset.
requirements.txt	Shows all the necessary Python packages.