

Technical Implementation

Technology

Front-End

Flutter

Google Maps Flutter Library

Back-End

Python 3.11

FastAPI

Pydantic

Psycopg2

Ruff

Pandas

Docker

Postgres

Front-End

The front-end utilizes Google's Material designs, their Maps Library, and connects to our backend using HTTP Endpoints. The entirety of the front-end is built using Flutter, which allows us to easily develop the application once, and run it across multiple platforms.

A typical Flutter project tree will contain directories for each platform, but all development happens in the 'lib' directory.

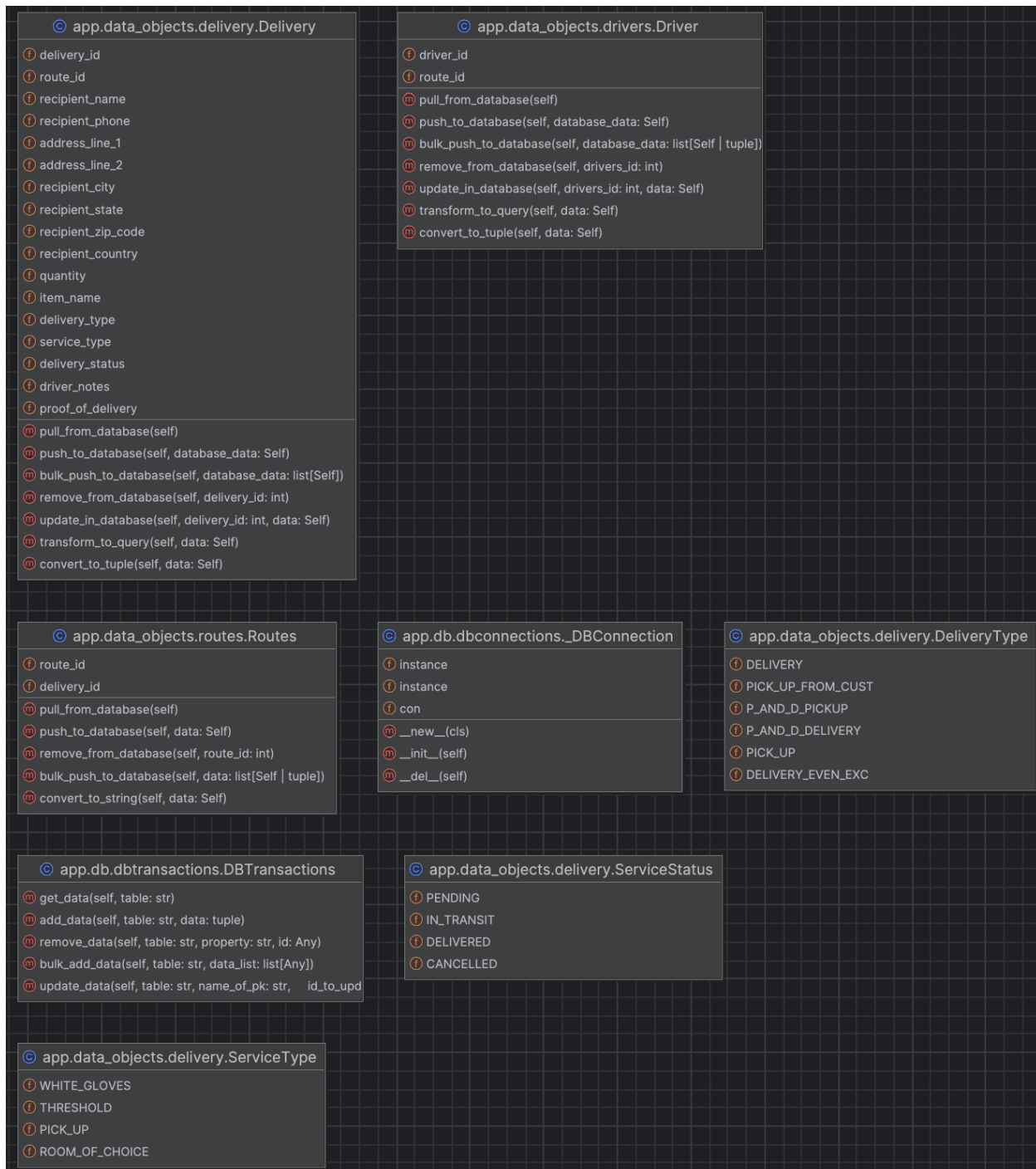
In our case, all "views" are in the pages directory, utility classes and functions are in the "utilities" directory.

Within the utilities directory are the api and dataObjects directories. the api directory contains utility functions to interact with the backend, the dataObjects directory will contain classes for a 1:1 data representation with the back-end.

Back-End

The backend is run independently from the front-end and supports connections to multiple clients through HTTP endpoints. The Python backend holds a temporary data-store that reflects the tables and their columns as stored in Postgres.

The backend works through DAOs as well as utilizing the builder paradigm to build data objects. Data validation is handled automatically through Pydantic. Once data is received and ready to be pushed to either the front-end or the database, the backend will convert it to either a JSON response, or a Postgres query. Data can be manipulated in the backend, but once it is committed, it must be pulled again, and the resultant must be modify then later committed.



The Python files in the data_objects module are 1:1 representations of their Postgres table counterparts. Each data object includes some utility functions that aides in sending, receiving, updating, and deleting data from the database such that no SQL queries are need to be directly written.

The `_DBConnection` class is a singleton that ensures only one Postgres connection is opened and handled by `Psycopg2` at a time. `DBTransactions` holds SQL query strings that are used to execute and commit changes from the data objects.

API

API Calls are defined using `FastAPI`, which will automatically generate an OpenAPI JSON document. Each data object has its own route with corresponding methods to Create, Read, Update, and Delete entries from/to the database. Whenever data is received to be uploaded to the database, it is received as JSON in the request body from the front-end. Once the data is received, it is converted to an object and data validation is performed synchronously using `Pydantic`. The API routes hold no business logic, other to ensure that data movement is correct and valid.

Excel files which hold delivery, route, and driver info are received as byte information, converted in-app as an Excel file, turned into a dataframe using `Pandas`, and finally converted to a data object. Once this process is complete, each row from the Excel file gets uploaded to the database.

The API does not return data objects from the POST or PUT HTTP requests. Once a request is received, and the data is validated, the API will return an HTTP 201 OK. This status code represents that the data was received and accepted. Because synchronous data uploads is not guaranteed, there may be a chance that the data that gets returned to the client is incorrect or incomplete. Thus, it is better to upload the data, then pull the data, versus uploading and returning the data with the same function. This design choice aligns with the Single-Responsibility Principle (SRP) which states “A module should be responsible to one, and only one, actor.”

Database Connections and Transactions

As previously mentioned, database connections are handled through a utility class that follows the Singleton pattern. Doing so ensures that there are no zombie connections which will use necessary resources on the backend, and on the DBMS. Following the singleton pattern will also allow for connections to be auto-closing if there is no additional transactions to be executed. One limitation of this approach is that if multiple connections need to happen synchronously (which may happen if there are over 100 clients requesting or pushing data at a time), transactions between the back-end and front-end may see a small delay. This can be avoided through the use of a connection

pool through psycopg2, but will require additional checks to ensure the pools are closed at the end of their tasks.

Database transactions are largely handled by the data objects using pre-formatted SQL strings. The purpose of the DBTransactions class is to actually execute and commit transactions from each data object. There is no formal data validation in the transactions class.

Database

public
delivery
deliveryid character varying (255)
routeid character varying(255)
receipientname character varying(255)
receipientphone character varying(50)
addressline1 character varying(50)
addressline2 character varying(50)
receipientcity character varying(50)
receipientstate character varying(50)
receipientpostalcode integer
receipientcountry character varying(50)
quantity integer
itemname character varying(255)
deliverytype deliverytype
servicetype servicetype
deliverystatus servicestatus
drivernotes text
proofofdelivery bytea

public
drivers
driverid integer
routeid character varying(255)

public
routes
routeid character varying(255)
deliveryid character varying (255)