OptimaLDN

SDCC Project A.Y. 24/25

Matteo Avallone
dept. Software Engineering
University of Rome, Tor Vergata
Rome, Italy
matteo.avallone2000@gmail.com

***Abstract*—Like many areas of interest, the public transport sector has undergone a massive transformation by shifting towards mobile app technology. Apps like these can really make commuting easier and enhance convenience in a hectic schedule. They represent a necessity to get real-time information about buses, undergrounds and other means of transportation. In this work, we explore a way to do so by accessing and manipulating the huge amount of data provided by the TfL (Transport for London) API, leveraging the power of microservice-based apps.**

I. Introduction

OptimaLDN stands for optima London: "optima" for optimal routes and "LDN" for London, emphasizing efficient travel across the city. This work puts the spotlight on the everyday commuter journey from home to work or any other kind of exploitation of public transportation. It is a microservice-based application that focuses on providing users with the most convenient and reliable ways of reaching point B from point A. In a city like London, where loads of different transportation options are available, commuters often struggle to make a call about what is the best route based on one's needs; it is usually paramount for a person to reach a certain destination the fastest way possible, but it is also agreed that, sometimes, embark on a more comfortable and less crowded route would benefit to a commuter's mood, at the expense of a slightly slower ride. This app comprises of six overall functionalities: users can login to the system, thus they can save past favourite routes, in order to take a look on them in the future; they also have the opportunity to embark on one of these saved routes or start a new journey, letting the system return the most favourable route based on two criteria: fetching different route options from the vast amount of information made available by the TfL API lets the app calculate the best option by meticulously taking into account two factors, them being overall speed and general crowding at each stop along the way. As a result, a user always gets the best alternative, thus accomodating its different needs, and gets a better overall experience that could positively affect its mood. Users can of course also terminate a route if they don't feel like pursuing it or for any other possible reason; two out of the five microservices also run in the background to spot any strange pattern in the status history of buses, dlr and tube: if a critical delay, usually referring to a part suspended, closed or severe delays affected line, is detected, the system readily recalculates a different route from the current user's position and lets the user know as fast as possible; if a sudden service worsening is observed, the system lets the user know and decide if it is best to recalculate the route or just keep the current active one if a minor delay wouldn't impact its needs.

II. TFL API

Transport for London (TfL)) is the local government body responsible for implementing transport strategy and managing transport services across the UK capital. As well as managing London's buses, it manages the Tube, Docklands Light Railway (DLR), Overground, and Tramlink, it runs the city's cycle hire scheme, its River Services, coach station and so on. TfL makes available 62 separate datasets. These are a mix of real-time feeds (such as Tube departure boards, live traffic disruption, live bus arrivals, and TfL's Journey Planner API), fixed datasets (such as timetables, station locations, and station facilities) and transparency-oriented datasets (detailing operational performance, directors' remuneration, etc.). In this work we're making use of the Line Status API in order to fetch crucial information about the status of public transport systems and inject them into InfluxDB, a time series database; the Crowding API jointly with the Journey API to get the available options for a certain user route, as well as filtering these latter by speed and tube/dlr or bus congestion.

III. Overview of the system

Before delving deeper into the details of the application, the following is a general overview of the system.

The microservice-based architecure lets the application break down into smaller and independent services which communicate with one another; this in contrast to a monolith architecure in which all the application's components are packaged together. The application consists of four microservices, plus the API Gateway and a CLI frontend for the user to interact with it. The Route Planner microservice manages route calculation requests, route recalculations and terminations, leveraging dedicated means to convert data into the correct format as requested by the TfL API, manipulate query results to appear more user-friendly and use a specific mathematic expression to get the best route; the User service manages the login procedure and storage of user favourite routes, as well as displaying a list of them; the Traffic delays service queries InfluxDB to get unsual delay spikes or sudden worsening of tube, bus and dlr services, on the occasion of which, alert messages are forwarded to the Notification service; this last manages notifications coming from both the Traffic and the Route planner services: it acts as a "proxy", forwarding notifications to other services if specific conditions meet, and it saves current user routes info in order to do so. The Api Gateway conveys the user requests to the appropriate microservices and returns the responses. Each of these microservices runs inside a container via Docker, so a total of five dockerfiles are provided, and Docker Compose has been used to orchestrate the containers. Speaking of patterns, each one of the microservices makes use of a database (InfluxDB, DynamoDB and PostgreSQL), so 'Database per service' is correctly implemented; 'Circuit Breaker' has also been used in order to manage the underlying consumption process in RabbitMQ (channel issues), the messaging broker used to send notifications between the services.

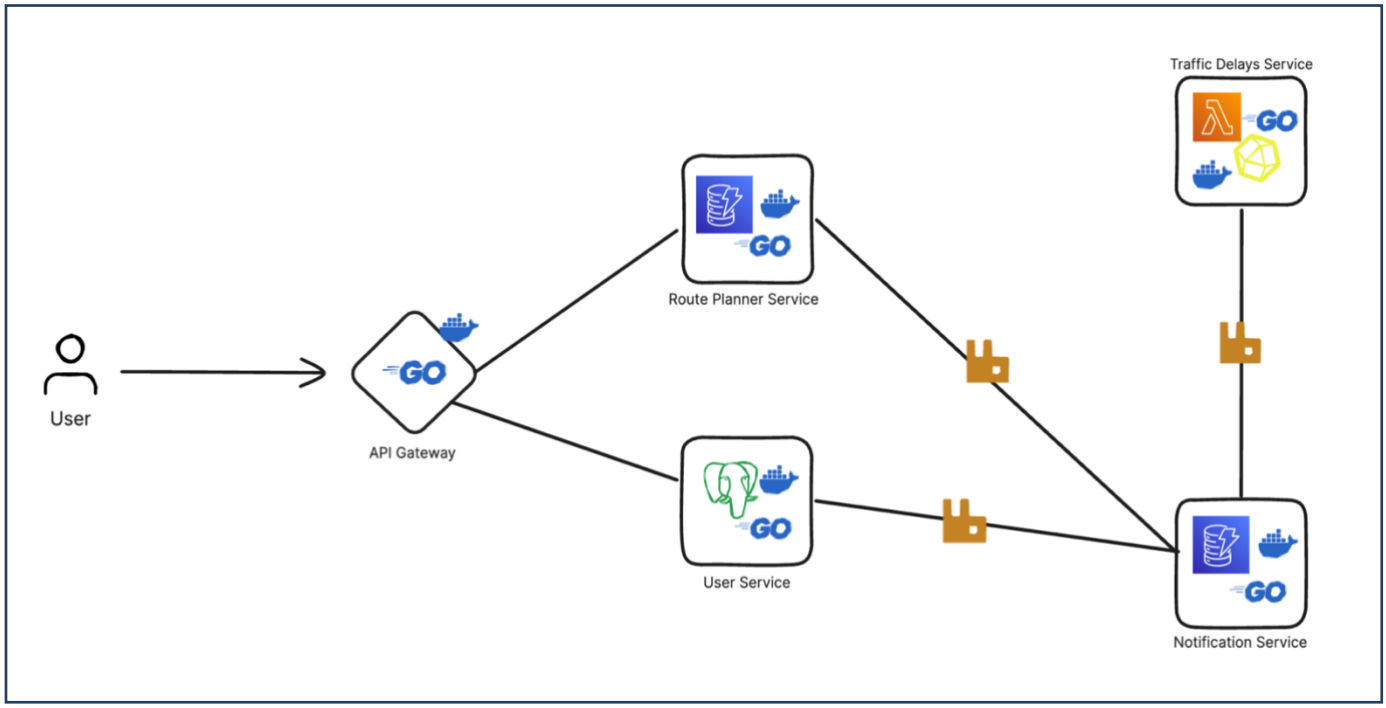

Figure 1 - System architecture

IV. Microservices

A. Route Planner

The Route planner is the core service of the system; it responds to user route requests, route terminations and real-time recalculations when disruptions occur. It exposes its functionalities through an RPC communication style and listens for events via RabbitMQ, thus handling route updates dynamically. Its key responsabilities are:

1) Journey planning: It accepts user requests to find the best route between two points in the London metropolitan area. The first thing it does is looking up for the start point and end point NapTan codes, which are unique identifiers within the National Public Transport in Great Britain; this is to ensure query lookups behave accordingly, without ambiguous results. It then fetches the possible journeys via TfL API and scores routes based on duration and crowding levels; the best journey is returned, and a structured summary of its key points will get its way to the user.

2) Route management: After having successfully determined the best route possible, it saves it onto DynamoDB for persistence; it then proceeds to inform the Notification service about this new active route via RabbitMQ. By exposing an RPC server, the route planner can also accept any pre-saved route from a specific user, it persists it and warns the notification service about it. It also handles termination of a route, by first querying DynamoDB to get the current active route of the user, it then informs the notification service and deletes the corresponding entry from the AWS database. It finally manages possible route recalculations by first estimating the current stop at which the user may find himself, it then gets the corresponding NapTan code and proceeds as done for a regular journey planning.

3) Event Handling: this service subscribes to disruption events via RabbitMQ and publishes updates to make external services aware; on receiving such an event, it proceeds to recalculate a route as specified above.

4) Infrastructure management: the route planner service stores and deletes route from AWS DynamoDB, it provides request/response operations for clients by exposing an RPC server, it loads a csv file called 'stationCodes.csv', containing NapTan codes for loads of London public transport stops and maps these to their corresponding station names in order to ease the lookup process. It also handles graceful shutdown of consumers (RabbitMQ queues) and cleaning of resources.

All of these functionalities are, to some extent, possible thanks to auxiliary go functions: to get a score for a route, based on crowding and duration, the first step requires *TimeStringToTfLTimeband,* which converts a time string into a TfL 15-minute timeband; this is crucial because TfL Crowding API lists crowding levels for a particular station by day of the week and 15-minute timeband, so 4 possibly different results in the span of an hour. To get the final score, the average crowding level is calculated across all stations of a route and then this mathematical expression is applied to each possible journey returned during the journey planner phase:

$$float64(duration) * (1 + avgCrowding)$$

where 'duration' is the total journey time in minutes; the multiplication by 1 + 'avgCrowding' scales the duration proportionally: for example, a 50% crowded route makes the effective duration 50% longer, reflecting user discomfort or potential delays, and suggesting that lower scores indicate faster and less crowded journeys.
In order to estimate the current stop when in need of a route recalculation, *EstimateCurrentStop* estimates progress along a journey leg (in TfL documentation, journeys are divided into legs, which represent segments of the journey, each one characterized by a start and end time, along with a list of stops), by calculating the fraction of the leg completed and multiplying this by the number of stops in the leg.

There's also 3 main functions that help converting data to the correct form needed, and of course the ones that help communicating with dynamodb for route deletion or insertion, as well as the ones that handle TfL queries.

B. *Traffic Delays*

The traffic delays service is a real-time monitoring microservice that detects significant disruptions in London's transport system and publishes alerts to RabbitMQ. It integrates with InfluxDB, where TfL data is stored and continuously scans for Critical delays, like severe disruptions, line closures or partial suspensions, as well as Sudden service worsening, referring to sudden drops in line severity. These alerts are sent to the notification service via RabbitMQ, which will end up potentially forwarding them to the route planner or user services.

It basically connects to an InfluxDB instance created on InfluxDB Cloud and creates an exchange on RabbitMQ to send notifications to the Notification service, so it is a publisher-only service. Its core functionality is the *StartDelayMonitor* that runs every minute using a ticker: it executes flux queries to detect potential delays, processing query results into appopriate alerts in order to publish them to RabbitMQ or logging an "all clear" message if no anomalies are detected. Using a goroutine allows the service to monitor traffic delays in the background, while the main thread waits for a shutdown signal.

The monitoring function performs two different flux queries to detect critical or sudden delays:

```
from(bucket: "%s")
  |> range(start: -15m)
  |> filter(fn: (r) => r._measurement == "tfl_line_status")
  |> filter(fn: (r) =>
    r.status_severity_description == "Severe Delays" or
    r.status_severity_description == "Part Suspended" or
    r.status_severity_description == "Closed"
  )
  |> group(columns: ["line_name", "mode_name"])
  |> last()
  |> keep(columns: ["_time", "line_name", "mode_name",
"status_severity_description", "reason"])
```

This query is designed to identify the tube, bus or dlr lines that are currently experiencing significant disruptions; it filters the data to include only records from the last 15 minutes relative to the current time and narrows down the records to only those belonging to the tfl_line_status measurement.

```
from(bucket: "%s")
  |> range(start: -30m)
  |> filter(fn: (r) => r._measurement == "tfl_line_status") and
r._field=="status_severity")
  |> group(columns: ["line_name", "mode_name"])
  |>sort(columns: ["_value"])
  |>filter: ® => r._value < -3.0)
  |> keep(columns: ["_time", "line_name", "mode_name",
"_value"])
```

This query, on the other hand, detects rapid drops in service quality, not yet labeled as "severe" but that may indicate a worsening trend; it focuses on the change in a numeric severity score over time. It sorts the records in each group by their timestamp in ascending order, calculates the difference between consecutive values in the _value column (status_severity score) and filters the results to only include those where there is a drop of more than 3 points (for example, from "Good Service" (score 10) to "Minor delays" (score 6).

C. *Notification service*

The notification service acts as a bridge between traffic delays events and user notifications. It dispatches the events based on the type: when receiving one via RabbitMQ, it handles it by its type, forwarding the alert to the route planner service if it's a critical delay, or to the user service if it's a sudden service worsening delay.

This service also listens to route planner events; when it receives a new route created event, the notification service records it in its DynamoDB table and deletes it once the route planner service warns it about a route abortion/termination.

The active routes tracking via DynamoDB ensures the service will be able to digest traffic alerts by checking if any active route will be affected by route disruptions. In order to reduce load, a local cache is used, so once the service reads the current active routes from the database, it caches them to avoid multiple successive reads. For each line, there's a list of active users associated to it, for example:

$$"victoria" \rightarrow ["user123", "user456", "user789"]$$

D. *API Gateway*

The API Gateway service conveys clients' requests to backend microservices. At startup it connects to user and route planner services, since the other two act just like "behind the scenes" microservices; persistent RPC calls are reused for all requests.

It exposes multiple REST-style endpoints, each related to a specific functionality, ranging from user management to route planning. It also makes use of WebSocket connections to push real time notifications coming from the backend services to the clients: it looks up connection for a user and sends a message, if any; a goroutine listens for disconnections.

E. *User Service*

The user service is responsible for user authentication, route persistence and receiving notifications about sudden disruptions that may affect a user's route. It ties together: PostgreSQL for persistence, RPC for route operations and RabbitMQ for external notifications.

User authentication happens via hashed passwords: the system checks if the password entered by the user matches the one corresponding to its username record in PostgreSQL.

This service also serves route requests: it gets a list of user saved routes from Postgres, it saves one to favourites by calling the route planner service to get the current active route and converting it into the right format in order to persist it in the database, as well as accepting a saved route, by forwarding the started route to the route planner service that can, in turn, warn the notification service and save it in DynamoDB.

It also supports RabbitMQ integration: it subscribes to notifications coming from the Notification service; if a sudden delay is received, it interacts with the api gateway to get the user to decide if he wants to recalculate the route or simply keep on the current one.

V. Patterns

A. Database per service

Each of the previous microservices makes use of a database, to ensure loose coupling, data autonomy and bounded contexts.
The user service owns a PostgreSQL database called 'optimaldn', with two main tables: the table "user", where user information is stored, and the table "user_saved_routes", where user favourite routes are stored, with fields like the start and end points, the estimated time, transport mode and traversed transport lines.
The route planner service uses DynamoDB, specifically the 'ChosenRoute' table in which info about a user's current active route is stored, and the user id is the main key used for look up queries.
The notification service uses DynamoDB as well, specifically the 'ActiveRoute' table, populated by users' active routes, queried every now and then to check if a delay alert coming from the traffic service matches one of these.
Lastly, the traffic delays service uses InfluxDB to store real time, temporal data about the transport network. It was chosen as it can ingest and handle high-frequency status updates from TfL API; alerts generated from the InfluxDB analysis are then sent into the event-driven system (RabbitMQ) for downstream processing and user notifications.

B. Circuit breaker

This pattern has been implemented for both the Publisher and Consumer components that interact with RabbitMQ. This is to prevent cascading failures: if RabbitMQ becomes unavailable, the system avoids continuously retrying and overwhelming the broker; when the circuit is open, publishing and consuming processes pause instead of flooding the system with failing calls, and the breaker automatically retries after a timeout, allowing recovery when RabbitMQ comes back online.
Both Publisher and Consumer wrap their Publish and Consume operations inside a circuit breaker: this latter trips if more than 3 consecutive publish/consume attempts fail to succeed; while open, it skips publish or consume operations, thus logging the error.
The breaker resumes automatically after the timeout, so the service can continue digesting messages without a full restart.

VI. Other technologies

A. AWS Lambda

As an example of serverless computing, AWS Lambda was used to create a Lambda function, in order to run code on demand without managing servers and in a fully managed, scalable and cost-efficient way. It was written in Go, like the rest of the project, and deployed on AWS; it was configured with environmental variables that enable it to connect to an InfluxDB Cloud bucket and each invocation is independent: an AWS EventBridge trigger runs the function every 15 minutes, so 4 times per hour; in the CloudWatch section on AWS, it can be seen, in the logs section of the function, that it successfully writes approximately 800 lines in InfluxDB.
That is because the main objective of this function is to periodically fetch live status updates from TfL API and store them in InfluxDB Cloud for monitoring, analytics and delay detection purposes.
For each line status returned, it creates an influxdb point with line_id, line_name and mode_name as tags and status_severity, description, reason and is_disrupted fields; basically anything other than a status severity of 10, which means "Good line service" determines a disruption. It's serverless and event-driven, and it automatically retries on API or network failures via Lambda's retry mechanism.

B. InfluxDB Cloud

It is a time-series database used to store, query and analyze real-time transport data collected from an external API; it is optimized for time-series workloads, so data with a timestamp. The 'Cloud' version was chosen instead of running it in Docker because the managed version ensures reliability, high availability and automatic scaling. This also aligns with the AWS Lambda architecture need for minimal latency and especially less management complexity; the Lambda function needs a reachable and always-on endpoint to push data to, so a docker version of InfluxDB would only work out if it's running and exposing it properly. Once the container stops, AWS Lambda will result in not being able to reach it. Using other AWS facilities to run InfluxDB would result in a more costly approach, while its cloud version makes it more cost-effective.

C. RabbitMQ

Other than publishing and consuming messages, there's also a setup of RabbitMQ, designed with fault-tolerance in mind. The connection logic implements a retry mechanism with exponential backoff: rather than failing immediately, the client retries every up to five times with increasing delays, preventing unnecessary load while still ensuring proper recovery.
Services can declare exchanges, queues and bindings directly at startup, simplifying infrastructure management; a centralized resource cleanup function also ensures resources like channels and connections are properly shutdown, reducing the risk of resource leaks. Error handling makes errors during initialization cause immediate termination via the 'failOnError' function.

VII. Libraries and tools

Several libraries have been used in support of the application; those are:

- github.com/jackc/pgx/v5/pgxpool:
  this is used for the PostgreSQL driver and connection pool manager; it automatically handles connection reuse, cleanup and load distribution across queries, improving performance and reliability.
- github.com/rabbitmq/amqp091-go:
  this library provides a native Go implementation of the AMQP protocol, enabling services to publish and consuming messages reliably.

- github.com/sony/gobreaker:
  this is used to implement the circuit breaker pattern, preventing the system from repeatedly calling a failing external dependency by tripping after several failures.
- golang.org/x/crypto/bcrypt:
  this is used by the User Service to secure password hashing and verification: it basically prevents users from storing raw passwords and incorporates a salt and cost mechanism to resist brute-force attacks.
- github.com/google/uuid: used by the User Service to generate universally unique identifiers for user routes, avoiding collisions that could occur with incremental IDs in a microservice architecture and ensuring idempotency in distributed systems.
- github.com/influxdata/influxdb-client-go/v2:
  this is the Go client for InfluxDB, supporting both writing and querying time-series data and it is used by the Traffic Service to push TfL line status data into InfluxDB for real-time monitoring.
- github.com/aws/aws-sdk-go-v2/config:
  this is used to load AWS configuration data; it handles credentials discrovery and region settings.
- github.com/aws/aws-sdk-go-v2/service/dynamodb:
  this provides a client to interact with DynamoDB, allowing the Route Planner/Notification services to store and retrieve structured data.
- github.com/patrickmn/go-cache:
  this provides an in-memory key-value store with expiration support and automatic eviction; used by the Notification Service to reduce repeated calls to DynamoDB, thus improving performance, reducing latency and lowering the pressure on the database by keeping frequently accessed data in memory for fast retrieval.
- github.com/aws/aws-lambda-go/lambda:
  package used to build and deploy the AWS Lambda function that fetches TfL data and stores it in InfluxDB and it enables serverless execution without a managing infrastructure.
- Github.com/gorilla/websocket:
  This package implements the websocket protocol in Go for persistent full-duplex connections between clients and servers; it is used by two microservices as well as the api gateway and the frontend, allowing this to receive real-time notifications pushed by the api gateway, without requiring constant polling; each connected user is stored in a map[userID]*websocker.Conn so the system can send targeted notifications, and background goroutines read messages, detect disconnections and clean up resources safely.

VIII. Deployment

- Docker was used to package each microservice along with its dependencies inside a container.
- Docker Compose was used to define and manage multiple services, including databases and RabbitMQ, as a single stack: this provides a simplified setup (just a single command to start all services) and orchestration.
- AWS EC2: an EC2 instance was launched to host the dockerized services in a cloud environment, thus providing scalability, flexibility by choosing the instance type and size and also integration with the AWS ecosystem.

IX. Testing

Since one of the functionalities of this application, namely the “Recalculate route” one, along with its flow of operations, appeared to be challenging to live-observe, to ensure the notification pipeline and message handling were functioning correctly, a test script was implemented to simulate a critical delay event and publish it directly to RabbitMQ. This allowed the verification of the system’s behaviour (message consumption, user notifications..), without relying on live disruptions on the transport network. The test constructs a sample notification payload with a simulated delay on the Jubilee Line, encodes it into JSON format and publishes it to the ‘traffic_events_exchange’ exchange with routing key ‘traffic.route.update.critical’ and metadata via header to indicate the alert type.
This validates the integration between services.

X. Closing remarks

This application is a practical example of leveraging the extensive Transport for London data. By implementing a scoring strategy based on multiple criteria, such as travel time, disruptions, number of stops, crowding, to evaluate routes, the system provides optimal ways to reach a destination. Moreover, by integrating real-time service updates, historical trends and user preferences, the system can interpret raw TfL feeds into insights for route planning. It shows how large-scale data can be effectively processed and used to deliver personalized and efficient travel recommendations to end users. The system could also be further enhanced by incorporating weather conditions or machine learning to refine route scoring based on user feedback.

References

[1] Transport for London, “Travel in London 2024, annual overview” 2024, Available: https://content.tfl.gov.uk/travel-in-london-2024-annual-overview-acc.pdf

[2] Pavel Despot, “Monolith versus Microservices: weight the pros and cons of both configs” 2023, Available: https://www.akamai.com/blog/cloud/monolith-versus-microservices-weigh-the-difference

[3] Becky Hogge, “Transport for London, get set, go!” 2016, Available: https://odimpact.org/files/case-studies-transport-for-london.pdf

[4] Muralish Clinton, “Microservice Circuit Breaker Pattern With RabbitMQ” 2023, Available: https://levelup.gitconnected.com/microservice-circuit-breaker-pattern-with-rabbitmq-a8cc1ebd3c7c