Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Real Time Public Transport App

Semester Thesis

Yuxin Sun
yuxsun@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Joël Mathys
Prof. Dr. Roger Wattenhofer

October 7, 2022

Acknowledgements

First and foremost, I would like to express my appreciation to my supervisor, Joël Mathys, for his continuous guidance and valuable advice at every stage of the project. My gratitude extends to Prof. Dr. Roger Wattenhofer for giving me the opportunity to conduct this thesis. Finally, I would like to thank the geOps group for answering all of my confusions patiently and supporting this project.

During the total developing procedure, I learned React Native from scratch, looked into the details of different public APIs and spent a lot of time on UI design. Through this project, I learned a lot of new techniques and became more confident in my coding abilities. I am very grateful for this experience. It benefits me a lot and encourages me to write code more happily in the future.

Abstract

In our modern life, mobile phones are becoming more and more important. Different kinds of apps provide people with unprecedented convenience. There exists many apps designed to facilitate people's travel, such as Google Map and SBB Swiss. However, these apps mainly focus on either route planning or online ticket purchasing. A traffic app which is personalized for each individual user based on his own preferences will bring more convenience. But this kind of app is rarely seen in the App Store.

To this end, a personalized real time traffic app is in demand. In this work, we develop a Swiss Real Time App to provide users with the right to specify their own interested traffic routes. Users can save their interested routes and check the departure times and delays in the future. This way, users can focus more on their preferred routes, without having to manually search and check every time. Users can also get real time traffic around them and get notified if their connections are late.

Contents

Chapter 1

Introduction

1.1 Motivation

Real time traffic data plays an important role in our modern daily life. When shopping, working or even traveling, it becomes much more convenient for people to gather some help from a digital timetable. For most people with a regular lifestyle, they often take the same traffic route for working or shopping. What's more, if a possible delay happens, it will be very frustrating for them not being notified and making new plans in time. Therefore, an application that can take users' own route preferences into account, and notify the users on possible delays is in demand.

However, most products do not meet this need at present. Some of the most popular traffic Apps, such as Google Maps, mainly focus on GPS service, real time navigation and route planning. Other applications, including some traffic Apps, mainly focus on booking service. All of them don't have the functionality to provide users with personalized real time traffic data based on their preferences.

According to the needs for the users as well as the lack of relevant Apps in the current market, there is a desire to develop such an application, which can show the interested real time traffic status to users, and furthermore avoid possible delays and potential troubles.

1.2 Assignment

This work shall focus on developing a cross-platform application, which can run on both Android and iOS devices, to help users maintain their own interested routes and detect possible departures and delays based on real time traffic data. In order to achieve this goal, the application should at least include several basic functionalities as below:

- Enable users to select their interested traffic connections based on origin and destination.

- Give users a detailed look into the connection, including transportation transfer and passed stations.
- Enable users to save preferred routes into their local devices.
- Keep track on the departures of users' preferences and notify them on any possible delays.

Except these main functionalities above, it is possible to add some additional auxiliary functionalities for a better usability, such as managing saved items, editing the app settings and adding a map view.

1.3 Overview

First, we describe the main related technologies used in this project in Chapter 2. Then in Chapter 3, we display the functionalities and UI layouts of the app. In Chapter 4, we delve into the implementation of this work. Finally, in Chapter 5, we provide the conclusion and some ideas for future work.

Chapter 2

Technology and APIs

2.1 React Native Framework

React Native [1] is a UI software framework, which can develop applications for Android and iOS by enabling developers to use the React framework along with native platform capabilities. It is available for both Windows and macOS. A single codebase can be shared across platforms. The styling part of React Native has a similar syntax to CSS, while it does not use HTML or CSS. In React Native, messages from Javascript threads are used to manipulate native views. In this way, developers do not have to write native code in the languages for the specific platform. This project uses React Native framework to be available for both major app platforms.

Expo [2] is an open-source npm package and framework for React Native projects, which will make it easier to develop and help developers set up applications quickly. Developers do not need to know native mobile coding as Expo will process all native code in the background. Another convenience provided by Expo is that: by downloading Expo Go from Google Play or Apple Store, it allows the project to be open during the development process without creating it through XCode or Android Studio. This is very useful for testing as developers can view all the code changes without creating an APK or IPA file. Expo also offers many native APIs for iOS and Android, which make it very easy for developers to add features such as location service, map service and notification service. And developers do not need to worry about the integrating procedure, as it is provided as part of the Expo package.

One possible limitation for Expo is that, developers cannot select a specific push notification service, such as Firebase. Expo integrates push notification service with One Signal, so developers must use it out of the box. As this project only needs local notification service rather than push notification service, this potential limitation will not influence our work.

2.2 Transport API

Transport API [3] is a public free API which provides interested developers with public transport data in Switzerland. It builds on REST style resources and responds in JSON. Transport API provides three main API resources: `/locations`, `/connections` and `/stationboard`.

- `/locations`: This method will return a list of locations in JSON, including station name. The required parameters can be either the name of a station/place, or the latitude/longitude coordinates of a place. The resource URL of this method is `http://transport.opendata.ch/v1/locations`.
- `/connections`: This method will return a list of connections based on the given parameters. The response data includes some important fields, such as from, to, stations, departure time, arrival time, delays, passed list, etc. Required parameters include from and to, which represent departure location and arrival location respectively. Optional parameters, such as date and time, can help filter or specify the willing response. The resource URL of this method is `http://transport.opendata.ch/v1/connections`.
- `/stationboard`: This method will return the next connections leaving from a specific location. Required parameter is the name of the station, and the response will include the transportation name, transportation number, arrival time, departure time, etc. The resource URL of this method is `http://transport.opendata.ch/v1/stationboard`.

Based on our required functionalities, `/locations` resource and `/connections` resource are needed in this work. `/locations` can be used to auto complete the input boxes, while `/connecitons` can be used to search for specific routes.

2.3 geOps API

The geOps [4] developer portal provides a toolbox for public transit applications. The main four public APIs provided by it are: Routing API, Maps API, Realtime API and Stops API. Especially, the Realtime Websocket API can return the real time vehicle positions based on scheduled times and real time updates. The required parameters of this Websocket API include a bounding box in epsg:3857 (coordinates of the top, left, bottom and right boundaries of the user's screen) and a zoom level. The response data is in JSON, which includes time intervals, vehicle coordinates, delays, etc.

This work uses geOps Realtime API together with Google Map API [5]. It provides users with a map view of the movements and possible delays of real time vehicles.

Chapter 3

App Design

The following chapter describes the functionalities and screen displays of the final version of the app. First, we give an overall view of the total architecture. Then we dive into four main screens to illustrate their specific functionalities, usages and UI layouts.

Two important concepts will be used in the following part of the report, connections and routes. A connection includes a specific `origin` and a specific `destination`. A route represents a specific transportation. For instance, from `ETH Hönggerberg` to `Zurich Central`, users can either take Bus 69 and then Tram 7, or take Bus 69 and then Tram 14. We call the journey from `origin` to `destination` (such as `ETH Hönggerberg` -> `Zurich Central`) as a 'connection', and the transportation method (such as Bus 69 -> Tram 7) as a 'route'.

3.1 Architecture Overview

The application consists of four main screens: `Home`, `Map`, `Search` and `Settings`. Further more, there are four auxiliary screens: `Details`, `Edit`, `Helper` and `API Support`. The four main screens can be navigated through a tab bar at the bottom of the app, while the four auxiliary screens can be accessed through the four main screens. The page navigation tree is illustrated in Figure 3.1:

The app provides five main functionalities for users: search for interested routes; save, edit and delete routes; display next departure times and delays of saved routes; notifications for possible delays; display nearby moving vehicles in a map.

The `Home Screen` allows users to check the next departure times and possible delays of their saved routes.

The `Map Screen` provides users with a real time view of traffic nearby. The user can see himself as well as the moving vehicles while displaying the current delays.

The `Search Screen` enables users to search for their interested routes based

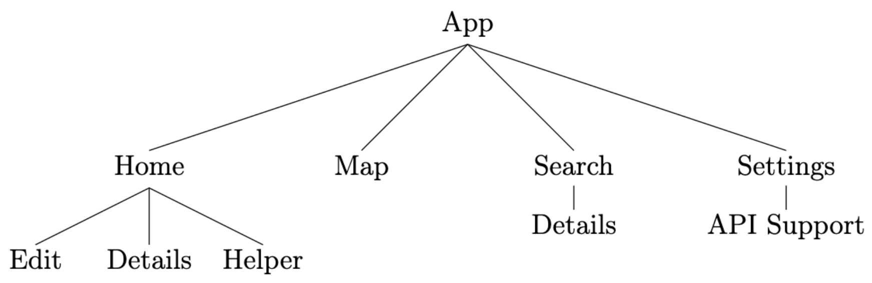


Figure 3.1: This figure shows the overall architecture of the app. Each node represents a screen. A child screen in the tree means that it can be accessed by its parent screen.

on origin and destination.

The `Settings Screen` includes some options for users to personalize the app configurations.

3.2 Home Screen

Upon opening the application, the user starts at the `Home Screen`, which shows all saved preferences. From there, users can access the three auxiliary screens, including `Helper`, `Edit` and `Details`. The `Helper Screen` offers a guidance to users. It illustrates how to use this app. The `Edit Screen` enables users to manage their saved preferences, including deleting a specific notification time and deleting the entire route. The `Details Screen` gives users a detailed view of the specific route, including transportation transfer, passed stations, start time, end time, etc.

All saved connections are shown in the `Home Screen`. Each connection may include several routes. Users can find the departure time and possible delay of each route. If the location permission is granted from the user, all connections will be sorted by the distance from the origin to the user's current location. And the routes inside each connection are sorted by their departure times. Users can refresh their current locations by clicking the refresh button on the top right of the `Home Screen`.

The `Home Screen` is further split into two parts – Nearby Stations and Other Stations. Connections whose origin station is less than 2km away from user's current location are classified into Nearby Stations. The remaining connections are shown in Other Stations.

The screenshot of the `Home Screen` is shown in Figure 3.2. By clicking the `?` button on the top left, the user will navigate to the `Helper Screen`, as shown in

Figure 3.3.

For each route item, users can click the $\cdots$ button to go to the `Edit Screen`, which is displayed in Figure 3.4. As we can see, users can either delete notification times one by one or delete the entire route using the bottom button.

If the user touches the right side of each route item, he will see the `Details Screen`, which includes real time details of the route, as shown in Figure 3.5.

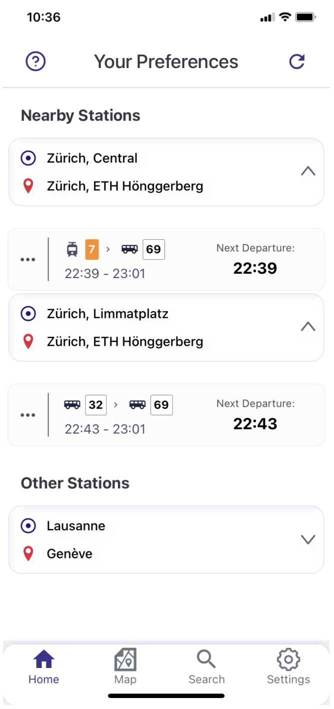
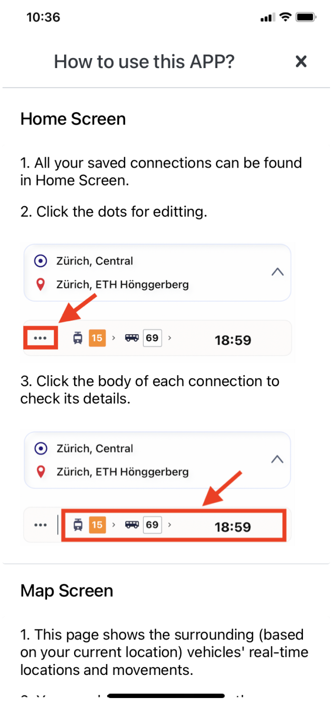


Figure 3.2: This figure shows the Home Screen. Stations that are less than 2km away are included in Nearby Stations part. Others are included in Other Stations part.



10:36

How to use this APP?

Home Screen

1. All your saved connections can be found in Home Screen.

2. Click the dots for editting.

3. Click the body of each connection to check its details.

Map Screen

1. This page shows the surrounding (based on your current location) vehicles' real-time locations and movements.

Figure 3.3: This figure shows the Helper Screen. It illustrates the usage of the app to users.

3.3 Map Screen

The `Map Screen` enables the user to see the surrounding vehicles while displaying their current delays. The map is centered based on the user's current location. Users are able to zoom in/out or change the map region by swiping the screen. By clicking a vehicle marker in the map, users can see the number of that transportation.

The screenshot of the `Map Screen` is shown in Figure 3.6. An on-time vehicle

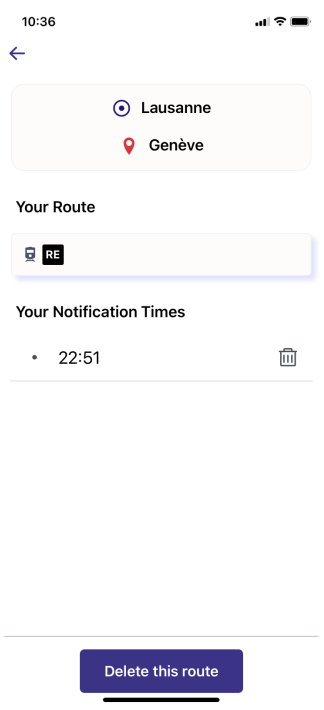
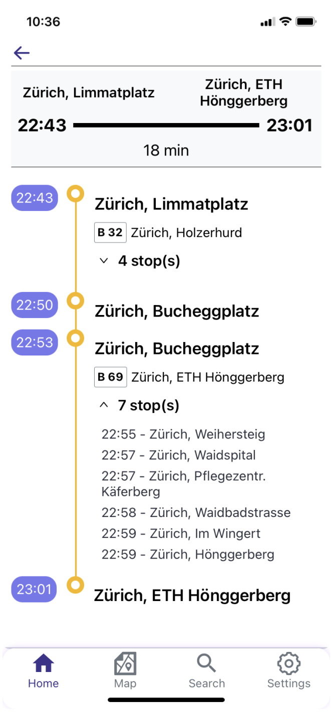


Figure 3.4: This figure shows the Edit Screen. Users can either delete notification times one by one, or delete the entire route directly.



Zürich, Limmatplatz — Zürich, ETH Hönggerberg
22:43 — 23:01
18 min

22:43 Zürich, Limmatplatz
B 32 Zürich, Holzerhurd
4 stop(s)

22:50 Zürich, Bucheggplatz

22:53 Zürich, Bucheggplatz
B 69 Zürich, ETH Hönggerberg
7 stop(s)
22:55 - Zürich, Weihersteig
22:57 - Zürich, Waidspital
22:57 - Zürich, Pflegezentr. Käferberg
22:58 - Zürich, Waidbadstrasse
22:59 - Zürich, Im Wingert
22:59 - Zürich, Hönggerberg

23:01 Zürich, ETH Hönggerberg

Figure 3.5: This figure shows the Details Screen accessed from the Home Screen. Users can find the departure time, arrival time, duration and a detailed passed list.

is represented by a purple circle. A delayed vehicle is represented by a red circle together with delay time. The location marker in the bottom right of the screen is the user's current location. By clicking each moving vehicles in the screen, the user will see the number of that vehicle.

3.4 Search Screen

On the top of the `Search Screen`, two/three input boxes with auto completing functionality are provided. When users input their interested origin, via (optional) and destination, auto completing to suggest possible stations helps select the correct station name and save time.

The searching results are shown under the input part. They are based on the user specified origin, via (optional) and destination as well as the user specified request time. The results will be ordered by the departure time.

By clicking each returned route, users can go to the `Details Screen`. It

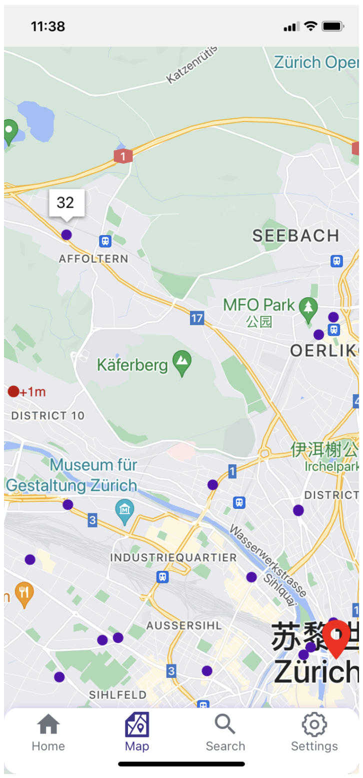


Figure 3.6: This figure shows the Map Screen. The location marker in the bottom right of the screen represents the user's current location. Purple circles are on-time vehicles. Red circles are delayed vehicles.

displays the detailed information of a specific route, including origin, destination, duration, delays and all passed stations.

Different from the `Details Screen` accessed by the `Home Screen`, this screen provides an additional 'SAVE' button for the user to save the route into the local device. Before saving, the app provides two options to the user. One option is to save only the route. This way, users can check the next departure times and delays in the `Home Screen` manually when opening the app. Another option is to save both the route and the time for future notifications. Not only will it enable users to find the departures in the `Home Screen`, but it also detects delays and sends notifications to users automatically.

A screenshot of the `Search Screen` is shown in Figure 3.7. The input part is on the top of the screen. Under the input part, the screen displays the searching results. Figure 3.8 shows the auto completing functionality in the `Search Screen`.

Initially, the `Search Screen` will only include two input boxes, one for origin, another for destination, as shown in Figure 3.9. If the user clicks the ⊕ button, an additional input box for via will be provided, as we can see in Figure 3.10.

With a click on the body of a route, users can navigate to the `Details Screen`, which is shown in Figure 3.11. In order to save an interested route into the local device, the user needs to click the 'SAVE' button. After that, two choices will be given, one is to save only the route, another is to save the departure time for notification as well, as shown in Figure 3.12.

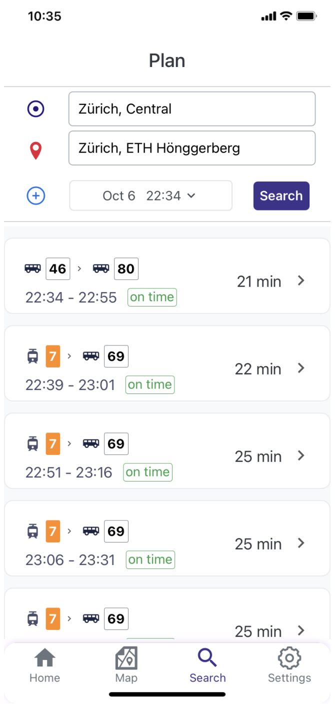


Figure 3.7: This figure shows the Search Screen. On the top of the screen, the user can input his origin and destination. Under that, the screen displays the searching results.

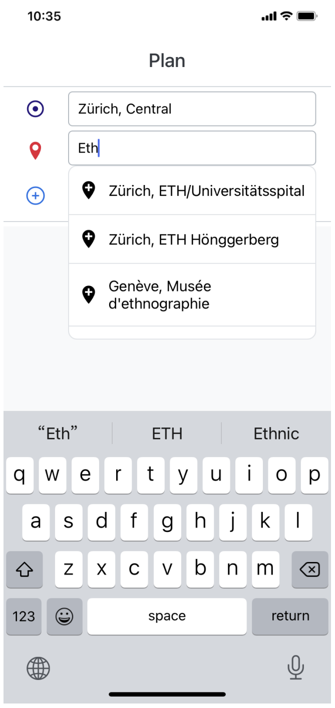


Figure 3.8: This figure shows the auto completing functionality in the Search Screen.

3.5 Settings Screen

The `Settings Screen` enables users to change the app configurations based on their preferences.

- The first option is whether to allow the app to send notifications. This value is set to false by default. Users can enable it if they want. On Android, the app can send notifications even when the user has force-quit the app; on iOS, the app can send notifications only if is running in the background.

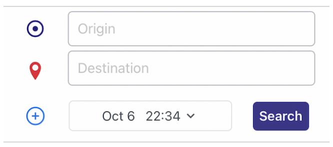


Figure 3.9: This figure shows input boxes in the Search Screen without via.

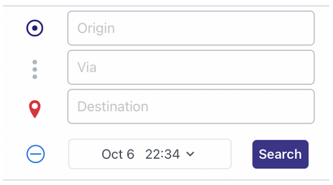


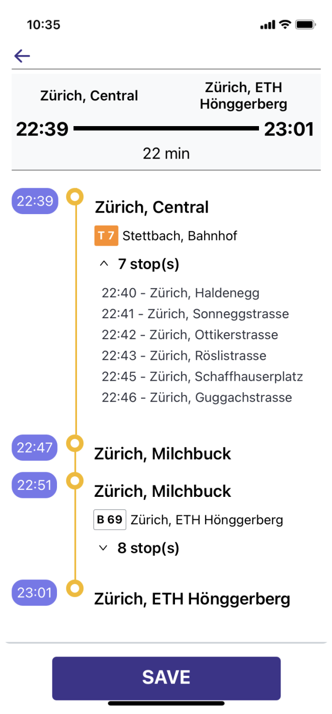
Figure 3.10: This figure shows input boxes in the Search Screen with via after clicking the ‘add’ button.

- The second option is to reset the number of moving vehicles shown in the `Map Screen`. Initially, the app will show 15 vehicles. Users are allowed to reset it to a value between 0 and 30.
- The third option is to clear all saved preferences with a single button. If users would like to reset the contents of the app, they do not have to delete them one by one in the `Home Screen`. Instead, they can delete all of them with a single click of the ‘Clear All’ button.

Notification enables users to be notified on possible delays even when he is not using the app. This way, the user does not have to open the app and check delays in the `Home Screen` every time. Notification times are the ones saved in the `Details Screen`. When the user clicks the notification message and opens the app, the app will directly navigate to the corresponding `Details Screen`. Notification is a useful functionality for users: if the user has a scheduled outdoor task, it will be more convenient for him to be notified for a possible delay, so that he can make a new plan in time.

The `API Support Screen` can be accessed from the `Settings Screen`. It provides further information about tools and APIs used for building the APP.

The screenshot of the `Settings Screen` is shown in Figure 3.13. By clicking the ‘API Support’ button in the bottom of the screen, the user can navigate to the `API Support Screen`, as shown in Figure 3.14.


10:35

Zürich, Central — Zürich, ETH Hönggerberg

22:39 — 23:01

22 min

22:39 Zürich, Central

T 7 Stettbach, Bahnhof

^ 7 stop(s)

22:40 - Zürich, Haldenegg
22:41 - Zürich, Sonneggstrasse
22:42 - Zürich, Ottikerstrasse
22:43 - Zürich, Röslistrasse
22:45 - Zürich, Schaffhauserplatz
22:46 - Zürich, Guggachstrasse

22:47 Zürich, Milchbuck

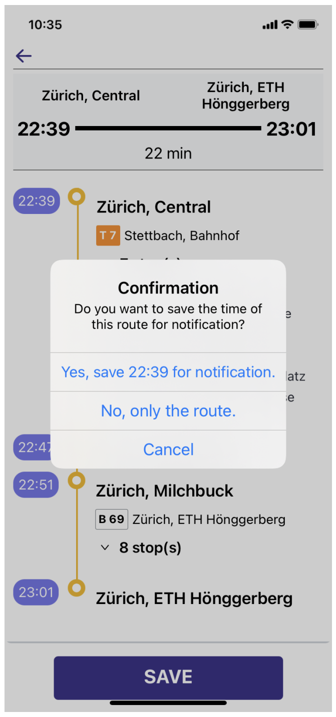
22:51 Zürich, Milchbuck

B 69 Zürich, ETH Hönggerberg

⌄ 8 stop(s)

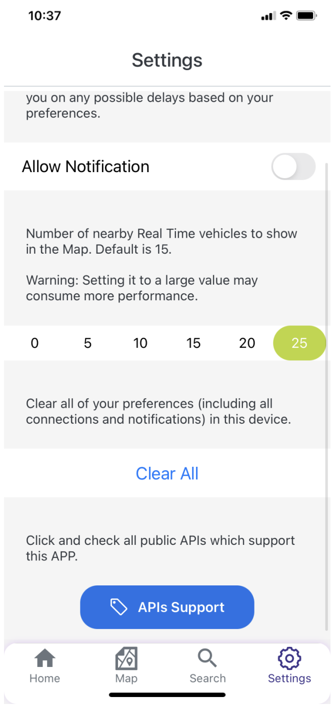
23:01 Zürich, ETH Hönggerberg

Figure 3.11: This figure shows the Details Screen accessed from the Search Screen. Users can save the route by clicking the 'SAVE' button.


Confirmation

Do you want to save the time of this route for notification?

Figure 3.12: This figure shows the saving choices provided for users.


10:37

Settings

you on any possible delays based on your preferences.

Allow Notification

Number of nearby Real Time vehicles to show in the Map. Default is 15.

Warning: Setting it to a large value may consume more performance.

0 5 10 15 20 [✓ 25]

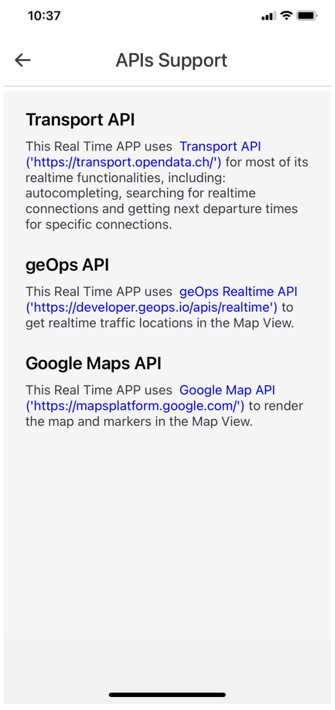
Clear all of your preferences (including all connections and notifications) in this device.

Clear All

Click and check all public APIs which support this APP.

APIs Support

Figure 3.13: This figure shows the Settings Screen. Users can enable notifications, change the number of vehicles displayed in the Map Screen, clear all data, and check the APIs support.


10:37

APIs Support

Transport API

This Real Time APP uses Transport API ('https://transport.opendata.ch/') for most of its realtime functionalities, including: autocompleting, searching for realtime connections and getting next departure times for specific connections.

geOps API

This Real Time APP uses geOps Realtime API ('https://developer.geops.io/apis/realtime') to get realtime traffic locations in the Map View.

Google Maps API

This Real Time APP uses Google Map API ('https://mapsplatform.google.com/') to render the map and markers in the Map View.

Figure 3.14: This figure shows the API Support Screen. Users can check all external APIs used in this app.

Chapter 4

Implementation

The following chapter describes some of the most important implementations in this work. The usage of public API and open-source packages, and some ideas of design will be discussed in this chapter.

In this chapter, first we will discuss the usage of Transport API, which supports most functionalities in this app. Then we will discuss the implementation of local storage. After that we will discuss the implementation of the map view together with geOps API. Finally, we will illustrate the implementation of notification.

4.1 Usage of Transport API

Transport API is the most important API used in this project, which supports most of the functionalities of the app. According to Section 2.2, two resource URLs are useful to this project. URL `/locations` enables our own auto completing functionality, so that the app does not have to use a public API, such as Google Auto Completing. URL `/connections` allows developers to get routes and possible delays based on specified parameters in the request message.

4.1.1 Auto Completing

One optional request parameter of `/locations` URL is `query`, which specifies the location name to search for. For instance, if the parameter is set to 'Bas', all locations whose names include 'Bas' will be returned in a list (usually not longer than 20) to the user in JSON.

When the user is typing in an input box, the app will check its status. If the length of the user's input string is longer than 3, the app will send a request to `/locations` URL, setting the `query` parameter the same as the string in the input box. Then the app will show the `name` field of each location in the returned list to the user.

4.1.2 Search for Routes

When sending request to `/connections` URL, the app will specify at least four parameters: `from`, `to`, `date` and `time`. `from` and `to` represents the `origin` and `destination` in the input boxes, respectively. `date` and `time` enables users to choose their interested departure time, rather than the default one, which is current time. For each response data, we need to parse several fields for further usage:

- `from` field includes information of the origin. Some useful sub-fields of it are:
  - `station.name`: The name of the origin.
  - `station.coordinate`: This field includes the latitude and longitude of the origin. It will be used in the `Home Screen` to calculate distances.
  - `departure`: The time when the transportation will departure from the origin.
  - `delay`: How long the transportation will be delayed at the origin.
- `to` field includes information of the destination. Some useful sub-fields of it are:
  - `station.name`: The name of the destination.
  - `arrival`: The time when the transportation will arrive at the destination.
  - `delay`: How long the transportation will be delayed at the destination.
- `duration` field tells the duration of the total journey.
- `products` field represents a route of the connection.
- `sections` field includes information of all sections of the total route in an array format. This field will be mainly used in the`Details Screen` to show the details of a route. Each item in the array represents a section. Take the route 'Bus 69 -> Tram 7' as an example, a section can represent the journey during Bus 69, or the journey during Tram 7. There are several useful fields in a section item:
  - `journey.category` and `journey.number`: These two fields represent the category of the transportation, such as a bus or tram, and the number of the transportation, respectively. The combination of `category` and `number` represents a specific line, like Bus 69.
  - `departure.departure` and `arrival.arrival`: The departure and arrival time of the journey, respectively.

- `departure.station.name` and `arrival.station.name`: The station name of the origin and destination, respectively.
- `departure.delay` and `arrival.delay`: How long the transportation will be late at the origin and destination, respectively.
- `departure.platform` and `arrival.platform`: The platform number of the origin and destination, respectively. In principle, these fields only work when the transportation is a train.
- `journey.passList`: This field includes information of all passed stations during the journey. Departure time and name of each passed station will be parsed for further usage.

What's more, the app also enables users to choose an additional via station in the `Search Screen`. If so, an additional `via[]` parameter will be specified in the request message based on the user's input.

In the `Search Screen`, parameters `from`, `to`, `date` and `time` can all be chosen by the user. After clicking the 'Search' button, a `/connections` request will be sent.

Except that, `/connections` URL is also used in the `Home Screen` and for notification. Unlike before, `date` and `time` here are set to current time, instead of being specified by the user. `from` and `to` are extracted from the local storage. Every connection needs its own request to be sent.

4.2 Local Storage

This project uses React Native's `AsyncStorage` package [6] to implement its local storage functionality. `AsyncStorage` stores items in a way of key-value pairs. All keys and corresponding values are stored in a `string` format. The key values are not predefined, they can be self-defined by the developer. The package also provides a suite of methods. The main four methods used in this work are: `setItem()`, `getItem()`, `removeItem()` and `getAllKeys()`.

4.2.1 Storage for Connections

One functionality that needs local storage is to save the connections (and routes) selected by the user. The format of the key is: `from` + `to` + `latitude` + `longitude`. `from` and `to` represent the origin and destination specified by the user respectively. `latitude` and `longitude` represent the coordinates of the origin. The reason we need latitude and longitude is that, in the `Home Screen`, the app shows the connections ordered by the distances from the origin to the user's current location. So if saving the value into the key, we are able to decide the connections' order without having to get the items and looking into the details.

According to the fact that in the `Home Screen`, the app will show all saved connections, and inside each connection, it will show all saved available routes of that connection. So, it will be a good choice to include `from` (origin of the connection) and `to` (destination of the connection) into the key. And for each key-value pair's value part, we save all routes of a connection. In this way, by getting all keys in storage, we are able to get all connections; and by getting the item of a specific key, we are able to get all routes of that connection.

More specifically, the app saves the value in format: {time: ‘’, products: ‘’, vias: ‘’, durations: ‘’, sections: ‘’}.

- `products` field includes all routes of a specific connection saved by the user.
- `time` field is the starting time of each corresponding route, and it is also used as the notification time. If the user does not save the notification time for that route, this `time` value of that route will be saved as ‘none’.
- `vias` field represents a station which is passed during the total journey of a specific route.
- `durations` field represents the duration of each route.
- `sections` field includes a list of all passed stations of a route.

All these fields are mainly used in the `Home Screen` and its auxiliary screens. `products` will be used to show routes in the `Home Screen` and `Edit Screen`. `time` will be used to show the notification times of a specific route in the `Edit Screen`. It also enables the functionality which allows users to delete a notification time. `vias` can help filter the response from Transport API, as discussed in Section 4.1.

`durations` and `sections` are used for backup. As discussed in Section 2.2, in some special cases, there is no available real time route for a short time. When it happens, the `Details Screen` will show the backup contents. It includes the details of the route status when the user saved it. `durations` and `sections` will save the corresponding details for this usage.

4.2.2 Storage for Notifications

Another functionality which needs local storage is notification. As illustrated in Section 4.2.1, we already have a storage of notification times in the `time` field. But it is mainly aimed to show notification times in the `Edit Screen`. It is convenient to use in the `Edit Screen`, as we only need to focus on the corresponding route there. But when we need to look for and check all notification times, we need to go over all keys and check their contents one by one, which is a non-trivial work. So it would be much better to create another kind of key-value pair, which aims for notification.

In order to save the notification times in a more explicit way, the format of the key is: `time` + `from` + `to`. In this way, the app is able to check notification times directly through keys, without having to look into the details of each item. The value of each key is the corresponding route (also called `products` before). This `products` field will be used to filter the returned results.

4.2.3 Lookup and Deleting

`setItem(key, value)` is called when saving new connections or notification times into the local storage. `getItem(key)` helps update the saved items. Additionally, in order to access all saved items, the combination of `getItem(key)` and `getAllKeys()` is needed: use `getAllKeys()` to get all keys related to this app, and then traverse all of them and use `getItem(key)` to get their values.

`removeItem(key)` is used to delete a specific connection or notification time by providing the key. When deleting all saved data of the app, the project does not use `clear()`, which clears all items directly. Instead, it uses `getAllKeys()` to get all related keys and then uses `removeItem(key)` to remove them one by one. The main reason to avoid using `clear()` is that: `AsyncStorage` is a non-secure storage package. It does not separate the storage spaces across different apps. So in order to avoid occasionally deleting data of other apps, it would be better to first check the items one by one and then safely delete them.

4.3 Map and geOps API

The `Map Screen` shows a map together with surrounding real time moving vehicles to users. `reat-native-maps` [7] provides a Map component which uses Google Maps on iOS and Android. So this project uses the `react-native-maps` package to render the map and markers. Each marker in the map represents a single moving vehicle.

4.3.1 Set Request Parameters

In order to get the real time position of vehicles, this project uses geOps Realtime Websocket API. The format of an effective request message is:

```
BBOX <left> <bottom> <right> <top> <zoom> [tenant] [gen] [mots]
```

The parameters packed by `<>` are required ones; while the parameters packed by `[]` are optional ones. In this project, we only set the required parameters in a request message.

`left bottom right top` describes the extent of the bounding box in epsg:3857 coordinates. In a Google Map, these values represent the latitude and longitude coordinates of the left, bottom, right and top boundaries of the user's screen.

`zoom` is the zoom level of the base map. Lower zoom value means less details. The zoom parameter in a request message in this app is calculated manually based on the value of `longitudeDelta`.

Another unavoidable effort to make geOps API compatible with Google Map API is that: coordinates in geOps API are based on epsg:3857 standard, while coordinates in Google Map API are based on epsg:4326. So all coordinates sent or received by geOps API need an additional calculation for coordinates transformation.

4.3.2 Parse Response Data

The response of geOps API is in JSON. There are five fields in the response data which are useful to this app: `train_id`, `coordinates`, `time_intervals`, `line.name` and `delay`.

- `train_id` is the ID of a public transport vehicle. It is a unique value among different vehicles. So it can also be used as an identifier for each vehicles rendered in the map. This field is saved into an array: `train_id`.
- `coordinates` is an array of several arrays (like `[[],[],[]...]`). Each item in the array represents a coordinate pair (latitude and longitude in epsg:3857) of a vehicle. The `coordinates` field includes not only one coordinate of that time point. Instead, it represents a list of coordinates inside a time interval. All coordinates of that vehicle during the time interval can received in a single response. This field is saved into an array: `coordinates`.
- `time_intervals` specifies the time interval. During this time interval, the coordinates in `coordinates` field are active. This field is saved into two arrays: `start_times` and `end_times`, which represent the starting time and ending time of the time interval respectively.
- `line.name` is the name of a public transportation. It is a non-unique value. This field is saved into an array: `names`.
- `delay` represents the delay in milliseconds of a vehicle. This field is saved into an array: `delays`.

Except the above arrays, an additional array `readyToReplace` is also defined. It is used to check whether a vehicle is ready to be replaced by others. All these arrays will used for rendering in the future.

4.3.3 Render Map View

The map will be re-rendered in every one second to show the movements of vehicles. As the `coordinates` array represents all coordinates in a time interval, the exact index of the currently rendering coordinates needs to be specified:

$index = ((t_{now} - t_{start})/(t_{end} - t_{start})) * num - 1$

The app first checks the current timestamp t_{now}. Then it calculates the fraction of t_{now} among the t_{start} and t_{end} returned in `time_intervals` field. Finally it multiplies the result with the length of `coordinates`.

Rendering markers is performance-consuming, so the app restricts the number of vehicles in the `Map Screen` with a parameter called `VEHICLE_NUM`. The value of `VEHICLE_NUM` is modifiable by users. In this way, not all vehicles in the response can be shown in the map. The app decides whether to show a particular vehicle in the map based on a few principles:

- Between the last time interval of a trajectory event and the beginning of the new trajectory event, there is a few seconds (can be 6 to 30 seconds). So if the current time is outside the time intervals, the app will make the choice here to display the last (or the first) position of an trajectory event instead of removing it.
- Each value in `readyToReplace` is set to 0 initially. This array represents how long a vehicle has been outdated in seconds. Every time the map is re-rendered, if the current time is later than the ending time in the time interval, this value will be increased by 1. When the value is larger than 30, the corresponding vehicle will be ready to be replaced. It means the app will wait for at least 30 seconds before replacing a particular vehicle.
- When a response comes in, the app will first check whether the `train_id` has been saved before. If so, it will update the `coordinates` array and `end_times` array to include new coordinates and extend the ending time.
- If the vehicle has not been saved before, the app will then check whether the number of current vehicles is less than `VEHICLE_NUM`. If so, the app will directly add the `train_id`, `coordinates`, `time_intervals`, `line.name` and `delays` fields into corresponding arrays, and set the corresponding value in `readyToReplace` to 0.
- If the number of current vehicles is more than `VEHICLE_NUM`, the app will check whether any vehicles in current screen is ready to be replaced. If so, the app will replace that old vehicle with the new one.
- If none of the above scenario is satisfied, the app will just ignore the coming response.

When users are changing the map view with some gestures, the region of the map will also be changed. So a new BBOX request will be sent. Vehicles that are still in the new region will not be replaced. Other vehicles will be replaced directly.

4.4 Notification

This app uses expo-background-fetch [8] package to implement its notification functionality. Initially, the app will define a background task by providing a name and a function that should be executed. When the user enables the notification button in the Settings Screen, the app will register that background task. The minimal interval between subsequent repeats of a background task can be specified. On Android, we set this value to ten minutes. On iOS, it is the smallest fetch interval supported by the system, usually 10 to 15 minutes.

The function executed in a background fetch includes several tasks:

- Get all saved notification times from local device. Check whether there exists any notification in the next 30 minutes. If not, just return and do nothing.
- Otherwise, if a notification time is in the next 30 minutes, the function will send a request to /connections resource of Transport API, with specified from, to, date and time parameters.
- The function will then check all response data to see whether there exists any matched products with a delay. If so, it will send a notification to the user and return; otherwise, it will continue to check the next saved notification time until the last one.

If the user clicks the notification bar, the app will save all useful parameters into a global variable, in order to pass them to the Details Screen for direct navigation.

Chapter 5

Conclusion

5.1 Conclusion

This work provides users with a personalized real time traffic app. Users can save their own interested routes and get notifications for delays. Several techniques are used to support this app, including React Native and some of its important packages, Expo, Google Map API, Transport API and geOps API. The app will be published and can be downloaded from both Google Play and Apple Store, so that more people can use it.

For future work, we can imagine the following functionalities. One possible functionality is to show the user's real time location in the `Details Screen` during a journey, so that the user can check how many stations have been passed. Another possible improvement is to enable notifications from a server side. During the initial design stage of the work, one important feature is to make all functionalities run on the local device, including storage and notification. So we decide not to introduce an external server at this point.

Bibliography

[1] React Native framework, https://reactnative.dev/.

[2] Expo platform, https://docs.expo.dev/.

[3] Transport API, https://transport.opendata.ch/.

[4] geOps Realtime API, https://developer.geops.io/.

[5] Google Map API, https://developers.google.com/maps.

[6] React Native AsyncStorage, https://reactnative.dev/docs/asyncstorage.

[7] React Native Maps, https://docs.expo.dev/versions/latest/sdk/map-view/.

[8] Expo Background Fetch Task, https://docs.expo.dev/versions/latest/sdk/background-fetch/.