MINISTRY OF COMMUNICATIONS AND INFORMATION TECHNOLOGY (MCIT).

DIGITAL EGYPT PIONEERS INITIATIVE (DEPI).

SOFTWARE DEVELOPMENT TRACK
Mobile Application Track

PROJECT: CAIRO METRO APPLICATION

Using Flutter and Android.

SUPERVISED BY
Eng. Hany Nemr

2024-2025

PROJECT TEAM WORK

1	Ahmed Mohamed Saleh
2	Doaa Abd El-Hafez Gad
3	Radwa Mohamed Shehata
4	Mohamed Al Sayed Nosair

Table of Contents

Abstract

The Cairo Metro App is a smart mobile application designed to improve daily transportation for Cairo's metro users. The app allows users to input their starting and destination stations to receive the shortest and all routes, number of stations, ticket price, estimated travel time, and real-time station alerts. Developed using both Android (Kotlin) and Flutter (Dart), the application highlights how technology can transform public transportation into a more efficient and user-friendly experience.

Acknowledgment

We extend our sincere gratitude to the **Ministry of Communications and Information Technology** for launching the "Digital Egypt Builders Initiative," which provided us with this outstanding opportunity to develop our skills bring this project to life.

We also express our deep appreciation to **CLS Company** for its dedication and for providing us with a highly experienced professional in mobile application development — **Eng. Hany Nemr** — whose guidance played a vital role in enhancing our learning journey.

His broad expertise, practical insights, and valuable feedback greatly contributed to the successful execution of this project with high quality.

At the conclusion of this journey, we offer our heartfelt thanks and respect to **Eng. Hany Nemr** for his continuous support and commitment, and we are truly honored to have learned from him during this inspiring educational experience.

CHAPTER 1:

GENERAL INTRODUCTION

1.1 What is the Cairo Metro App?

The Cairo Metro App is an innovative mobile application developed as part of a graduation project to address a real-world challenge faced by millions of daily metro users in Cairo. The application is designed to simplify commuting within Egypt's largest and busiest metro system by offering a smart and interactive platform. Through this app, users can input their current metro station and their intended destination, and the app will instantly calculate all possible travel routes between the two points. It then selects the most optimal route, usually the shortest one in terms of the number of stations. In addition, the application displays valuable travel information such as the estimated time of arrival, total distance to be covered, ticket pricing, and even provides alerts as the user approaches each station.

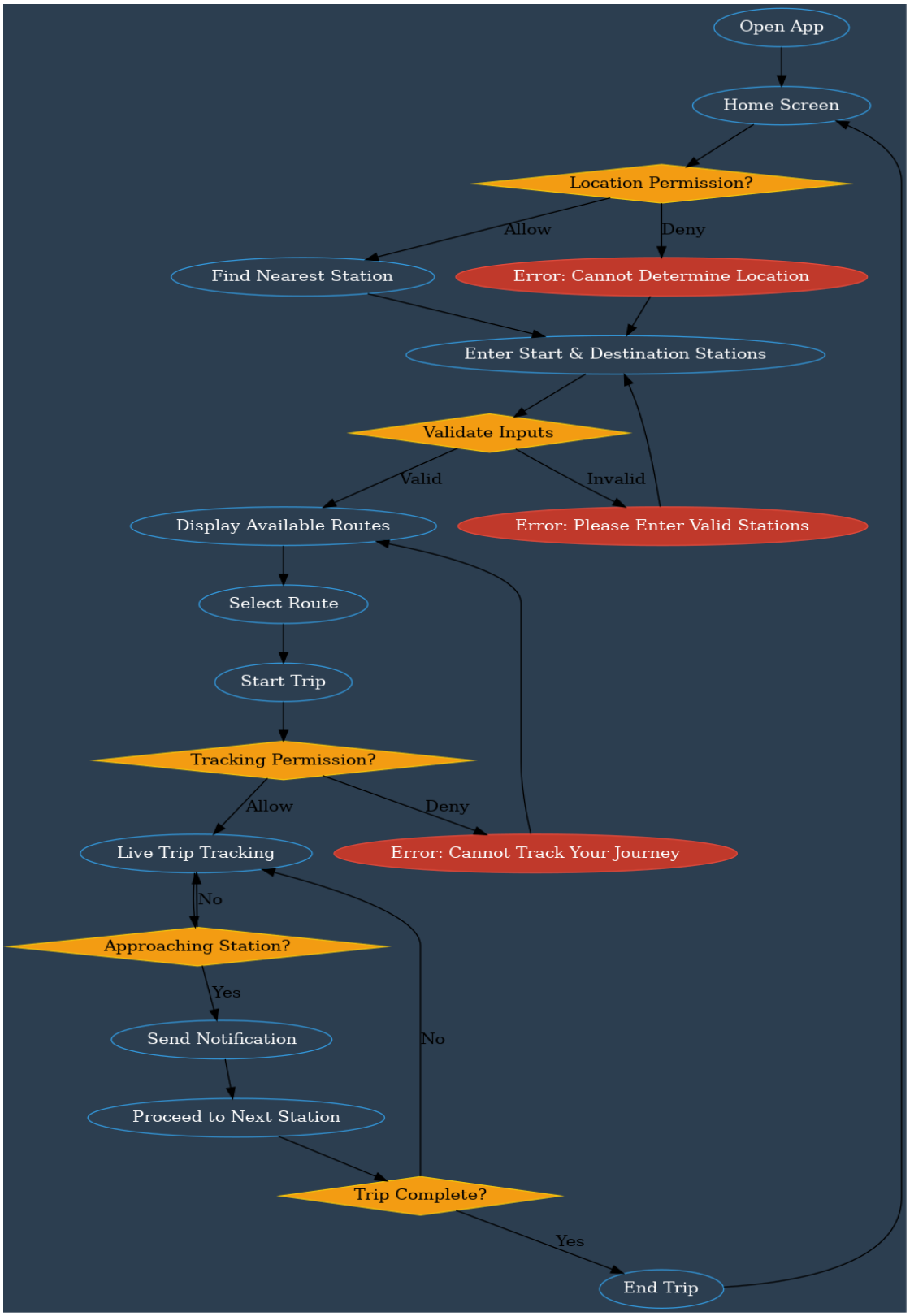
This tool enhances not only navigation but also planning, especially for newcomers, tourists, or anyone unfamiliar with the intricacies of Cairo's metro lines. The app serves as a daily companion to help users avoid confusion and delays, making urban transportation smoother and more accessible.

1.2 Importance of Using Technology in the Cairo Metro

The use of digital technologies in public transportation is no longer a luxury—it has become a necessity in major cities around the world. Cairo, being one of the most populous cities globally, experiences intense pressure on its public transportation systems. By integrating technology into the Cairo Metro system, we can enhance efficiency, reduce congestion, and offer passengers a more seamless and reliable commuting experience.

Digital applications help solve daily commuting problems such as lack of information, missed stops, inefficient route planning, and long travel times. Our mobile application directly addresses these issues by offering real-time routing, optimized path selection, and clear travel data. Furthermore, the app promotes sustainability by encouraging the use of public transportation through convenience, which may help reduce traffic congestion and pollution in the city.

Flow chart:


Open App
Home Screen
Location Permission?
Allow
Deny
Find Nearest Station
Error: Cannot Determine Location
Enter Start & Destination Stations
Validate Inputs
Valid
Invalid
Display Available Routes
Error: Please Enter Valid Stations
Select Route
Start Trip
Tracking Permission?
Allow
Deny
Live Trip Tracking
Error: Cannot Track Your Journey
No
Approaching Station?
Yes
Send Notification
No
Proceed to Next Station
Trip Complete?
Yes
End Trip

1.5 Technologies We Used

We will explain each of them in next chapters

The link include all codes about our project:

https://github.com/RAMD-hub/CairoMetroApp.git

CHAPTER 2:

NATIVE ANDROID APPLICATION

2.1 Introduction

The Android version of the Cairo Metro application was developed using **Kotlin** and the **XML** layout language. The app follows modern Android development practices and aims to deliver a smooth and responsive experience for Android users. The project leverages Android's native capabilities to interact with system services like GPS, notifications, and background tasks.

2.2 Packages We Used in This Application

✓*Air Location Library (Third-Party Library)*

The Air Location Library is a third-party Android library used to simplify and enhance the process of retrieving a user's geographical location. It combines both GPS and network providers to deliver accurate and efficient location data.
Key Benefits:

- Simplifies location permissions and handling.
- Supports real-time tracking or single-time location fetch.
- Reduces code complexity and battery consumption.

✓*Fused Location Provider (Google Play Services)*

The Fused Location Provider API is the official Google service for location tracking. It intelligently combines GPS, Wi-Fi, and mobile networks to provide the best possible location data with minimal power usage.
Key Benefits:

- High accuracy with optimized battery usage.
- Background and foreground tracking supported.
- Smooth integration with Android services.

✓*Foreground Service*

A Foreground Service runs in the foreground of the system and is less likely to be killed by the system under memory pressure. It's ideal for long-running operations like location tracking during a user's metro trip.

Key Benefits:

- Maintains active tracking even when the app is in the background.
- Required to display a persistent notification.
- Ensures consistent performance in critical tasks like navigation.

✓*Notifications*

Notifications are used to communicate important updates to the user throughout the metro journey. In this app, notifications are triggered during:

- Departure
- Arrival
- Station transfers
  this improves user awareness and enhances the overall travel experience.

2.3 Implementation and UI Design

The user interface was carefully crafted using XML layouts, following Material Design principles to ensure a smooth and intuitive user experience. Icons, color schemes, and navigation were all tailored to reflect metro travel themes and ensure ease of use.

In android version the app features will be divided depending on the functions related to it to:

1- Paths algorithms.
2- Paths calculations.
3- Location calculations
4- Notifications
5- Language

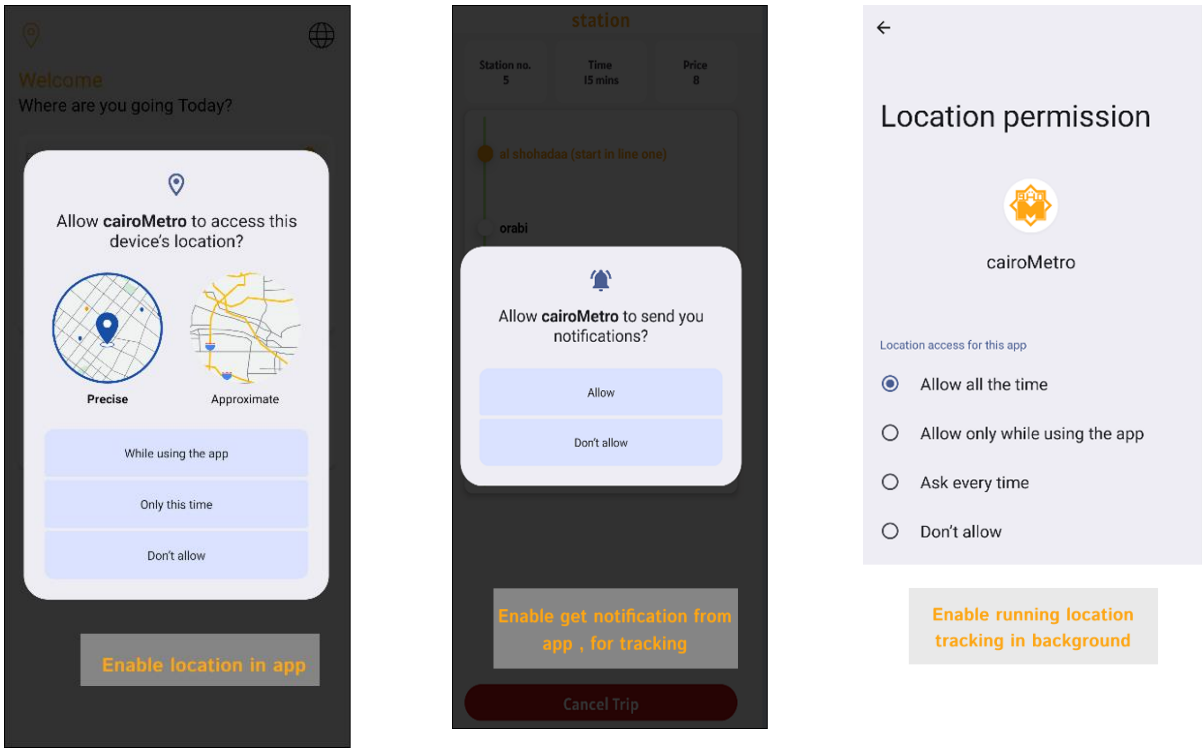
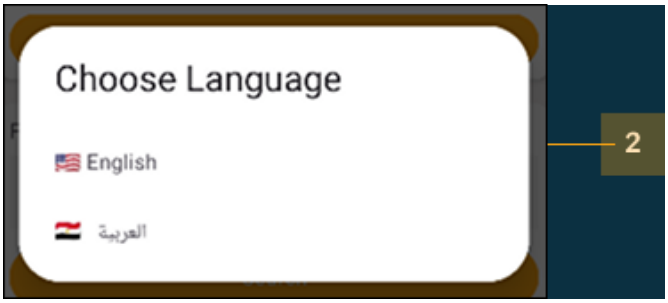


Figure 1 permissions required

Figure 2 home screen



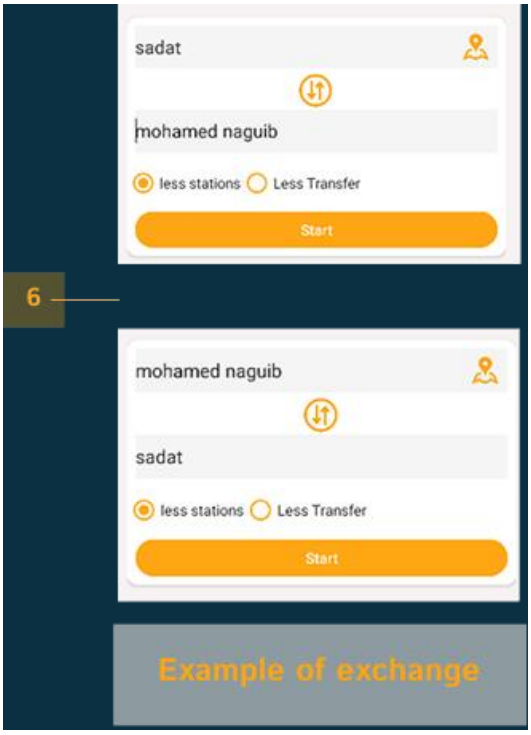


Figure 4 home buttons

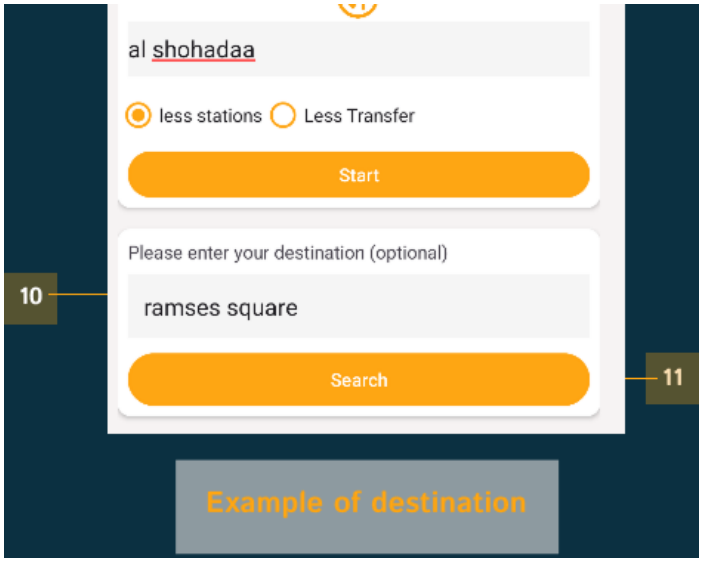
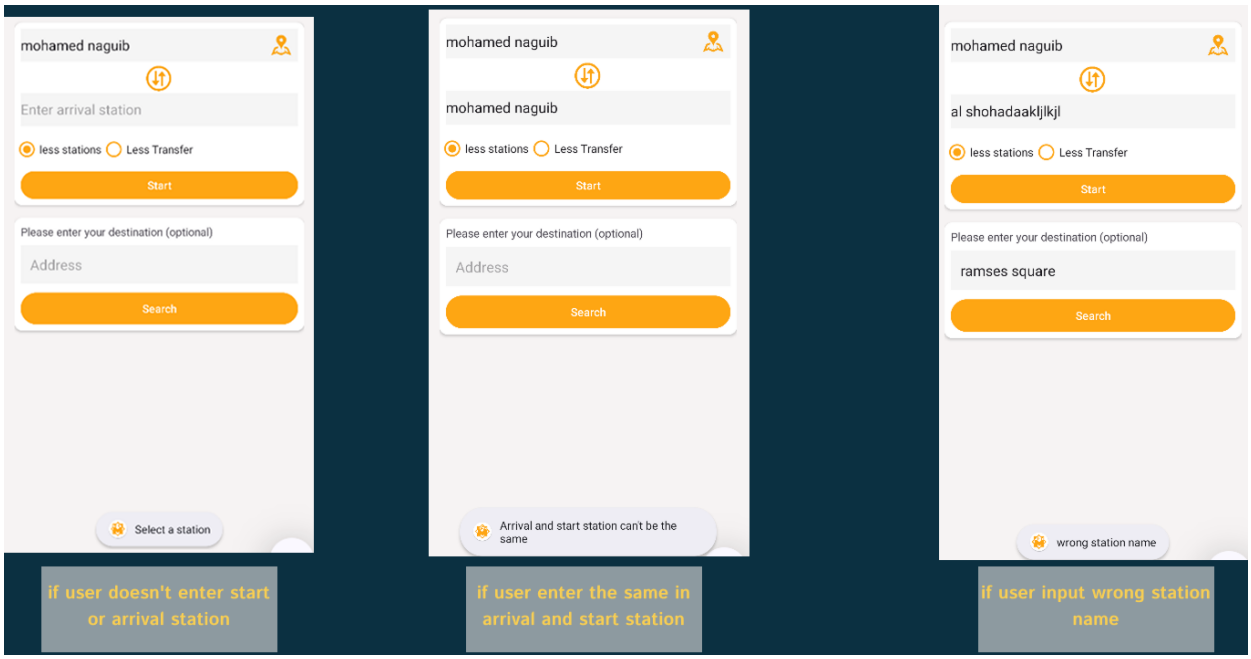


Figure 3 example of using app1



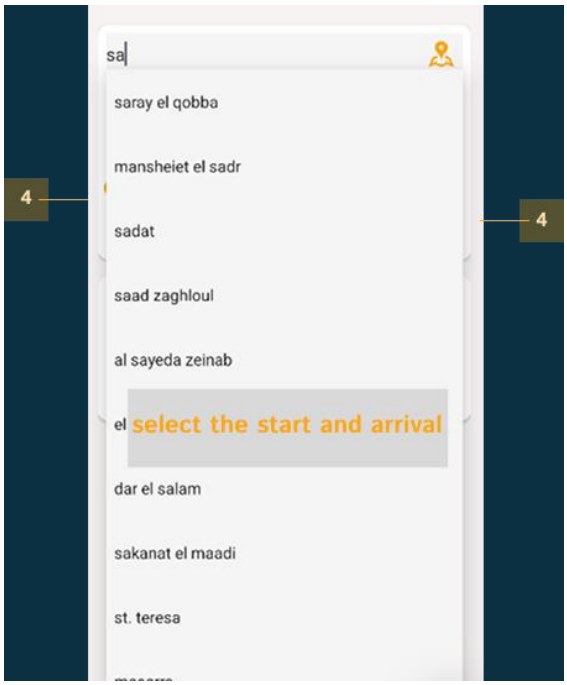


Figure 5 example of using app 2

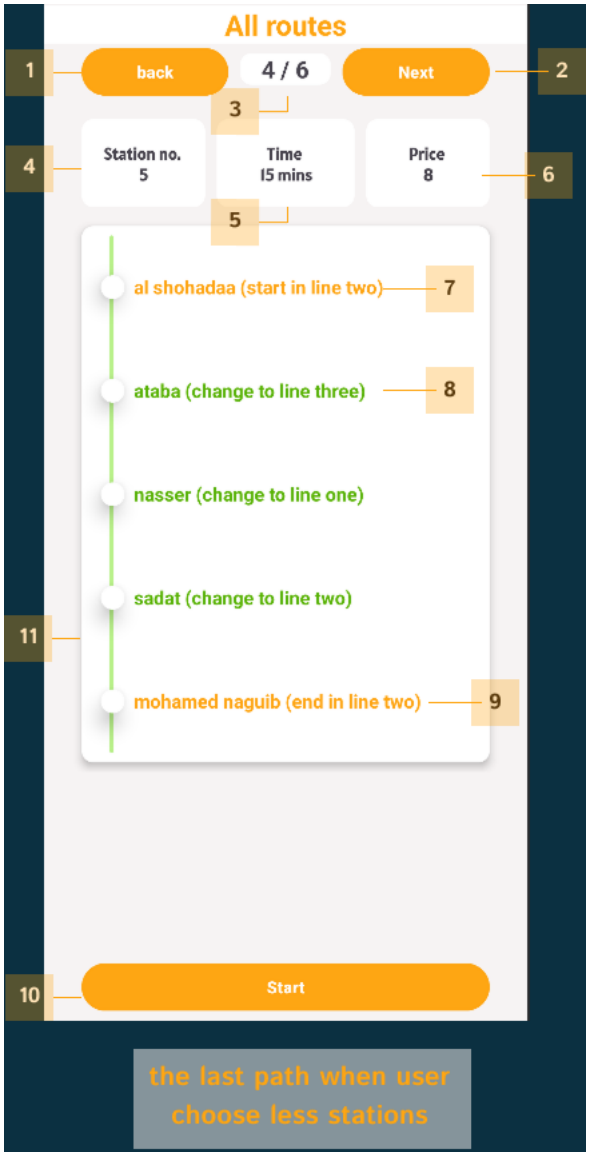
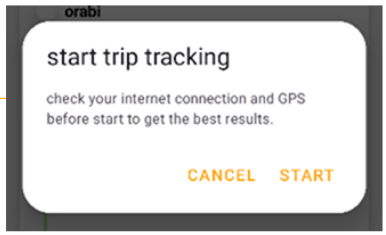
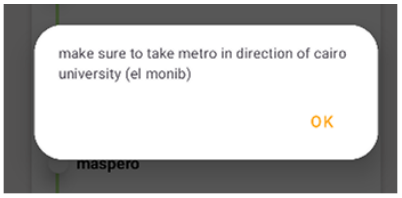
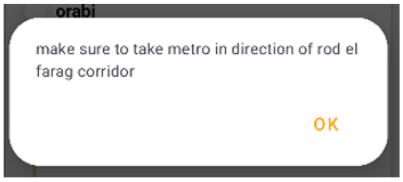


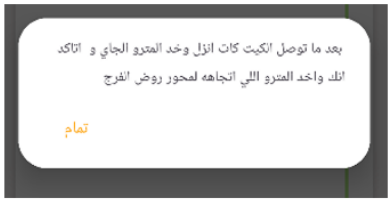
Figure 6 all routes screen

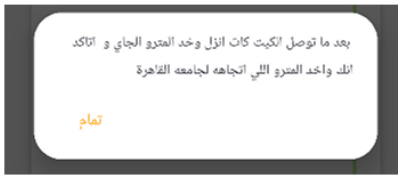


Appear when user click to start









Appear when the line has the branch of kitkat , and the user will go in direction of rod elfarag corridor or cairo university

the last path when user choose less transfer

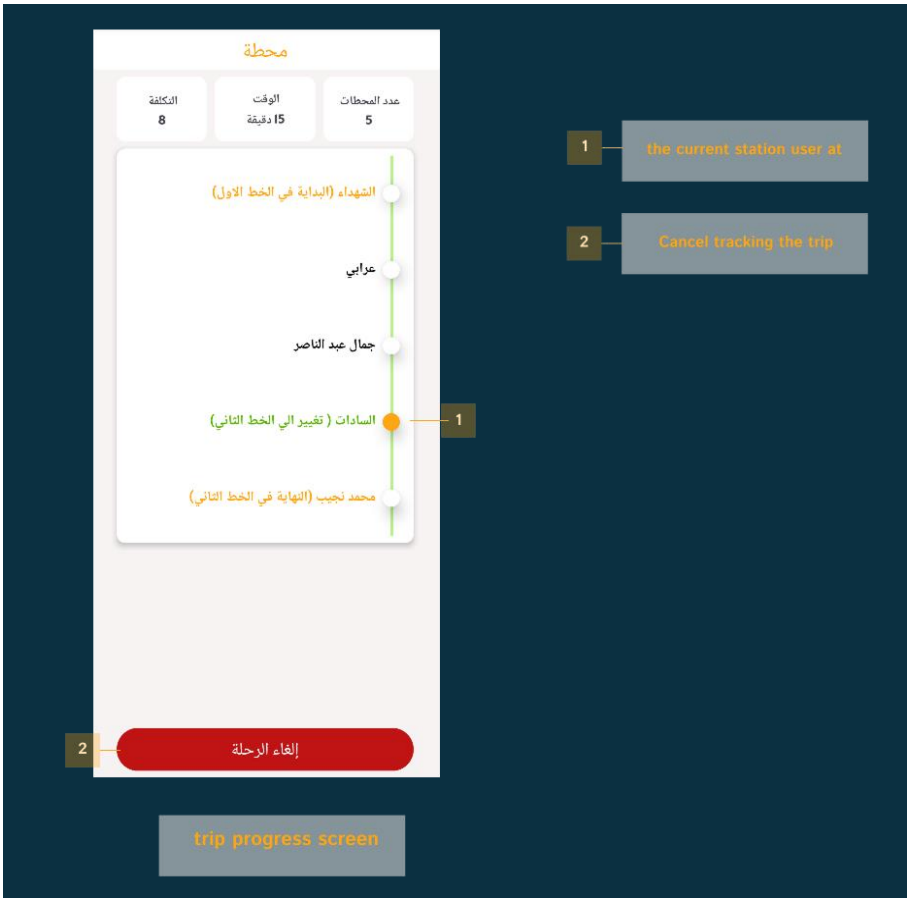


Figure 8 trip progress screen

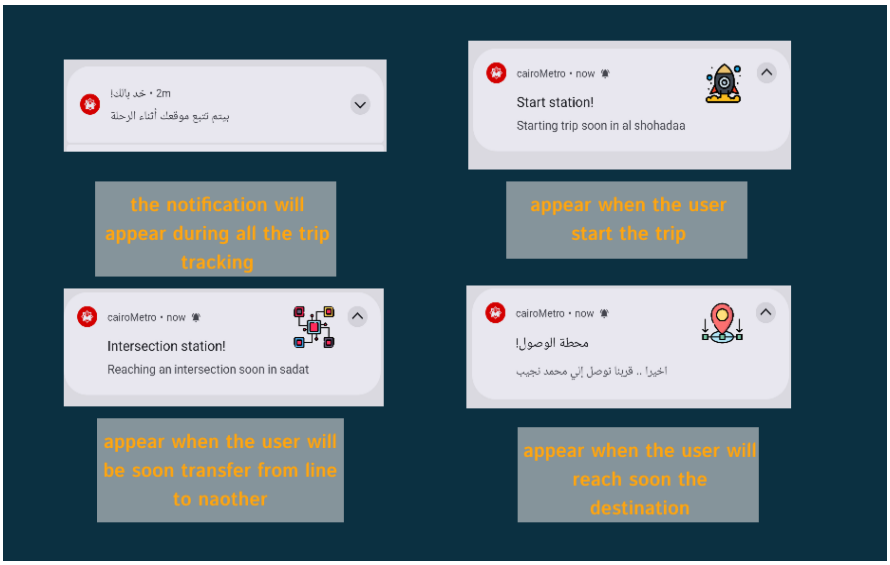


Figure 7 notifications

Paths algorithms

2.3.1 DFS **Algorithm** (All paths)

Depth-First Search (DFS) algorithm used to find all possible paths between two stations in Cairo metro. It explores every possible route from the starting station to the destination station while avoiding cycles.

2.3.1.1 Implementation function of DFS

```
private fun dfs(
    current: String,
    end: String,
    visited: MutableSet<String>,
    path: MutableList<String>,
    result: MutableList<List<String>>
){

    visited.add(current)
    path.add(current)

    if (current == end) {
        if(path !in result) {
            result.add(ArrayList(path))
        }
    } else {

        val currentClasses = data.filter { it.name == current }
        for (currentClass in currentClasses) {
            currentClass.neighbourStations.forEach { neighbor ->
                if (!visited.contains(neighbor)) {
                    dfs(neighbor, end, visited, path, result)
                }
            }
        }

    }
    visited.remove(current)
    path.removeAt( index: path.size - 1)

}
```

Figure 9 DFS algorithm

❖ **Function Parameters**

- current: The current station being visited (String)
- end: The destination station (String)
- visited: A mutable set of already visited stations to prevent cycles
- path: The current path being explored (list of stations)
- result: A list that accumulates all valid paths found

❖ **Algorithm Behavior**

- Marking Visited: The current station is added to the visited set and current path
- Base Case: If current station matches the end station, the current path is added to results (if not already present)
- Recursive Exploration: For each neighbor of the current station that hasn't been visited:
- Recursively calls DFS to continue exploring paths
- Backtracking: After exploring a path, removes the current station from visited and path to allow other paths to be explored

2.3.2 Sorting paths

2.3.2.1 by number of stations

This function sorts a list of paths based on the number of stations in each path.

2.3.2.1.1 Implementation function of sorting by stations

```
fun sortingByStations(paths:List<List<String>>): List<List<String>>{

        val sortedPath = paths.sortedBy { it.size }
    return sortedPath

}
```

Figure 10 sorting by number of stations

❖ **Function Parameters**

- paths: A list of paths, where each path is represented as a List<String> containing station names.

❖ Function Behavior
- ➢ The function uses Kotlin's sortedBy extension function.
- ➢ It sorts by the size (.size) of each inner list.
- ➢ Paths with fewer stations will appear before paths with more stations.

2.3.2.2 by number of intersections

This function sorts a list of paths based on the number of line changes (intersections) in each path.

2.3.2.2.1 Implementation function of sorting by intersections

Consist of find intersection then sorting them.

❖ Find intersections in path :

```
fun findIntersections(path:List<String>):MutableList<String>{

    val intersections = mutableListOf<String>()
    var lineName = findLine(path[0],path[1])
    val intersectionStations = data.filter { it.intersection }.map { it.name }.toSet()
    for (index in path.indices) {
        if((index + 1 )< path.size) {
            val nextStation = data.filter { it.name == path[index + 1] }.map { it.line }
            val station = path[index]
            if (station in intersectionStations && !nextStation.contains(lineName)) {
                intersections.add(station)
                lineName = findLine(station, path[index + 1])
            }
        }
    }
    return intersections

}
```

Figure 11 find intersections in path

- ➢ **Function Parameters**
  - Path: List<String>, An ordered list of station names representing a route through the metro system.
- ➢ **Function return value**
  - MutableList<String>List of station names where line changes occur (transfer points) along the given path.
- ➢ **Function behavior**
  - Determines the initial metro line from the first two stations in the path
  - Loads all known intersection stations from the system data
  - Iterates through each station in the path sequentially

- For each station (except the last one):
  - Checks if it's a designated intersection station
  - Verifies if the next station isn't on the current line
  - If both conditions are met, identifies it as a transfer point
- When a transfer point is found:
- Records the station name
- Updates the current line reference to the line connecting to the next station
- Returns the complete list of transfer points after processing the entire path

❖ Sorting:

```
fun sortingByIntersections(paths:List<List<String>>):List<List<String>>{

    val sortedPaths = paths.sortedBy { findIntersections(it).size }
    return sortedPaths
}
```

Figure 12 Sorting by number of intersections (line changes)

❖ **Function Parameters**

➢ paths: A list of paths, where each path is represented as a List<String> containing station names.

❖ **Function Behavior**

➢ The function uses Kotlin's sortedBy extension function.

➢ It sorts by the size (.size) of each path’s intersection number list.

➢ Paths with fewer intersections will appear before paths with more intersections.

2.3.3 Direction of path

Getting the line whether it one, two, three

2.3.3.1 function implementation of getting line name

```
fun findLine (first:String,second:String ):String{
    var lineName =""
    val firstStation = data.filter { it.name == first }
    val secondStation = data.filter { it.name == second }

    for (dataItem1 in firstStation) {
        for (dataItem2 in secondStation) {
            if(dataItem2.line == dataItem1.line)
            {
                lineName = dataItem2.line

            }
        }
    }
    return lineName
}
```

Figure 13 find line function

- **Function Parameters**
  - first: String - The name of the first station
  - second: String - The name of the second station

- **Function return value**
  String of the line name

- **Function behavior**
  - First, it data to find all stations matching the first parameter name

- Then it filters the same data collection to find all stations matching the second parameter name
- It then compares each station from the first list with each station from the second list
- For each pair, it checks if they belong to the same subway line (dataItem2.line == dataItem1.line)
- When it finds a matching line between any pair of stations, it returns that line name
- If no match is found after all comparisons, it returns an empty string

Path calculations

3.3.1 **price**

This function calculates a price based on the number of stations provided. It implements a tiered pricing structure where the price increases as the number of stations grows, following specific thresholds.

3.3.1.1 implementation function

```
fun calculatePrice(stationCount:Int): Int {

    return when {
        stationCount <= 9 -> 8
        stationCount <= 16 -> 10
        stationCount <= 23 -> 15
        else -> 20
    }

}
```

Figure 14 price function

- **Function Parameters**
  - stationCount (Int): The number of stations to use for price calculation. This should be a non-negative integer.

- **Function return value**

  Returns an Int representing the calculated price based on the station count

- **Function behavior**

The function uses a tiered pricing model with the following rules:

- 8 units for 9 or fewer stations
- 10 units for 10-16 stations
- 15 units for 17-23 stations
- 20 units for 24 or more stations

The function evaluates these conditions in order and returns the first matching price tier.

3.3.2 **Time**

This function calculates and formats a time duration based on the number of paths (pathCount). It converts the total time (in minutes) into a human-readable string, handling different cases for hours, minutes, and singular/plural formatting.

3.3.2.1 **implementation function**

```
fun time(context: Context, pathCount: Int): String {
    return if (pathCount * 3 / 60 >= 1) {
        context.getString(R.string.time_hrs_mins, ...formatArgs: (pathCount * 3) / 60, (pathCount * 3) % 60)
    } else {
        if (pathCount * 3 % 60 in 3 ≤ .. ≤ 10) {
            context.getString(R.string.time_mins, ...formatArgs: (pathCount * 3) % 60)
        } else {
            context.getString(R.string.time_min, ((pathCount * 3) % 60).toString())
        }
    }
}
```

Figure 15 time function

- **Function Parameters**
  - context (Context): Android Context used to access string resources for localization.
  - pathCount (Int): The number of paths, which determines the total time (each path takes 3 minutes).

- **Function return value**

  Returns a String representing the formatted time duration.

- **Function behavior**

  The function calculates the total time as pathCount * 3 minutes and formats it into a string with the following logic:

  1. If total time ≥ 60 minutes (1 hour):
     - Formats as "X hrs Y mins" (e.g., "1 hr 30 mins").
     - Uses R.string.time_hrs_mins for the string template.
  2. If total time < 60 minutes:
     - If remaining minutes (after removing full hours) is between 3 and 10:
       - Formats as "X mins" (e.g., "5 mins").
       - Uses R.string.time_mins.

3.3.3.stations count

This function generates a localized string that displays the number of stations in a given path.

3.3.3.1 implementation function

```
fun countStations(context: Context, path: List<String>):String = context.getString(R.string.station_no, path.size)
```

Figure 16 count stations function

- **Function Parameters**
  - context (Context): Android Context used to access string resources.
  - path (List<String>): A list representing the stations in a path. The size of this list determines the station count.
- **Function return value**

  Returns a String representing the formatted station count (e.g., "5 stations").

- **Function behavior**
  - The function retrieves the size of the path list (path.size).
  - It then formats this number into a string using the Android string resource R.string.station_no.
  - The string resource should handle proper pluralization (e.g., "1 station" vs. "5 stations").

Location calculations

4.1 libraries used to fetch location

4.1.1 Air location library inside activities.

- ❖ Implementation function

```
private fun startLocation() {
    airLocation = AirLocation( activity: this,  callback: this,  isLocationRequiredOnlyOneTime: false,  locationInterval: 5000)
    airLocation.start()
}
```

Figure 17 air location library function

- ❖ **Function behavior**

Initialization:

- Creates an instance of AirLocation with the following parameters:
- this (Activity/Fragment context)
- this (LocationListener callback)
- false (Indicates if single location update is needed; false means continuous updates)
- 5000 (Time interval between updates in milliseconds, i.e., 5 seconds)

Start Location Tracking:

- Calls airLocation.start() to begin receiving location updates.

4.1. 2 fused location provider inside service class.

- ❖ Implementation function

```
fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
createLocationRequest()

locationCallback = object : LocationCallback() {
    override fun onLocationResult(locationResult: LocationResult) {
        for (location in locationResult.locations) {

            getStationAndNotification(readAndWriteData, application, location)
        }
    }


}
```

Figure 18 fused location provider

- ❖ **Code behavior**

- nitialize FusedLocationProviderClient.

- Configure location request parameters (accuracy, frequency).
- Register the LocationCallback to receive updates.
- Process each location update (e.g., check nearby stations, send notifications).

4.2 Get nearest station

This function finds and displays the **nearest station** to the user's current location.

4.2.1 Implementation function

```
fun showNearest(view: View) {

    if(currentLocation.isNotEmpty()) {
        val station =
            location.nearestLocation(stationData,  shortest: 100F, currentLocation[0], currentLocation[1])
        if (station.isEmpty())
            showToast("no near station from your location")
        else binding.start.setText(station,  filter: false)
    }

}
```

Figure 19 nearest Station

❖ **Function behavior**

Checks if currentLocation is available

- If currentLocation is empty, no action is taken.
- If available, it proceeds to find the nearest station.

Finds the nearest station

- Calls location.nearestLocation() with:
- stationData (List of stations)
- 100F (Max search radius in meters)
- currentLocation[0] (Latitude)
- currentLocation[1] (Longitude)

Handles the result

- If no station is found (station.isEmpty()), shows a toast:
- "No nearby station from your location" (from R.string.no_near_station_from_your_location).

- If a station is found, updates the binding.start UI field.

4.3 Show nearest station on map

This function opens Google Maps with navigation directions from the user's current location to a specified destination.

4.3.1 **Implementation function**

```
fun directionFromCurrentMap(destinationLatitude: String, destinationLongitude: String ,context: Context ) {
    val mapUri = Uri.parse( uriString: "https://maps.google.com/maps?daddr=$destinationLatitude,$destinationLongitude")
    val a = Intent(Intent.ACTION_VIEW, mapUri)
    context.startActivity(a)
}
```

Figure 20 open location on map

❖ **Function Parameters**

- destinationLatitude String Latitude of the destination.
- destinationLongitude String Longitude of the destination.
- context Context Android Context (Activity/Fragment) to launch the map intent.

❖ **Function behavior**

1. Constructs a Google Maps URL
    - Uses Uri.parse() to create a deep link in the format:https://maps.google.com/maps?daddr=<LAT>,<LNG>
    - daddr stands for "destination address."
2. Launches the Maps App
    - Creates an Intent(ACTION_VIEW) to open the URL in the default maps app (Google Maps, Waze, etc.).
    - If multiple apps support this intent, the user sees a chooser dialog.
3. Starts Navigation
    - Google Maps automatically detects the user's current location and provides turn-by-turn directions to the destination.

4.4 Search for destination

This function handles address searches to find the nearest station to a given location.

4.4.1 **Implementation function**

```
fun search(view: View) {
    val address = binding.address.text.toString()
    if (address == "")
    {
        showToast("please input an address")
    }
    else {
        val startDetails = location.getLatAndLong( context: this, address)
        if (startDetails == Pair(0.0, 0.0))
            showToast("the address is not valid")
        else if (startDetails == Pair(-1.0, -1.0))
            showToast("error while loading the data, try again")
        else {
            val station = location.nearestLocation(
                stationData,
                shortest: 50F,
                startDetails.first,
                startDetails.second
            )

            if (station.isEmpty())
                showToast("no near station to your destination")
            else binding.arrival.setText(station,  filter: false)
        }
    }
}
```

Figure 21 get destination station

❖ **Function behavior**

- Validating user input
- Converting an address to coordinates (geocoding)
- Finding the nearest station within 50 kilometers
- Updating the UI or showing appropriate error messages

4.5 station dialog

This function manages station-based tracking logic

❖ 4.5.1 Implementation function

```
private fun stationDialog () {
    if(  indicator && path.isNotEmpty() ) {

        currentStation = location.nearestStationPath(stationData, shortest: 300F,path,currentLocation[0],currentLocation[1])

        if (currentStation.isNotEmpty()) {
            binding.status.isVisible = true

            if(previousStation =="" )
            { previousStation =currentStation}

            if (path.indexOf(previousStation) <= path.indexOf(currentStation)) {
                previousStation = currentStation
                uiForLocationUpdate()
                if (previousStation == path.last()) {
                    indicator = false
                    readAndWriteData.saveSimpleData( context: this,  data: false,  key: "indicator")
                    binding.status.visibility = View.GONE;
                    readAndWriteData.saveID( context: this,  data: 0, key: "previousHome")
                }
            }
        }
        else
        {
            if(previousStation == ""){
                previousStation = path[0]}
            binding.status.isVisible = true
            uiForLocationUpdate()
        }
    }
    val id = stationData.firstOrNull{it.name== previousStation }?.id
    if(id != null) {
        readAndWriteData.saveID( context: this, id, key: "previousHome")
    }
```

Figure 22 station dialog

❖ Function behavior

1.Activation Check

- Only runs if:
- indicator = true (tracking enabled)
- path is not empty (route exists)

2. Station Detection

- Finds the nearest station in path within 300m of currentLocation
- Updates currentStation if found

3. Progress Tracking

- First Detection:
- Sets previousStation if empty

- Forward Movement Check:
- Updates previousStation only if moving toward destination (path.indexOf() validation)
- Calls uiForLocationUpdate() to refresh UI

  4. Trip Completion
- If reaching the last station:
- Disables tracking (indicator = false)
- Hides status UI
- Resets saved station ID

  5. Fallback Handling
- If no station detected:
- Defaults to first station in path
- Still shows status UI
- 6. Data Persistence
- Saves the previousStation's ID for future reference

Notifications

5.1 LocationService – Tracking User Location and Sending Notifications During a Trip

5.1.1 Main Purpose:

LocationService is a **background service** that tracks the user's location during a trip inside the Cairo Metro app, and sends notifications when approaching important stations (start, end, or intersection).

5.1.2 Main Components:

```
private val NOTIFICATION_ID = 1
private val CHANNEL_ID = "location_service_channel"
private val PUSH_CHANNEL_ID = "push_notification_channel"
private val UPDATE_INTERVAL = 10000L // 10 seconds
private val FASTEST_INTERVAL = 5000L // 5 seconds
private val SMALLEST_DISPLACEMENT = 5f // 10 meters


private lateinit var notificationManager: NotificationManager
private lateinit var fusedLocationClient: FusedLocationProviderClient
private lateinit var locationCallback: LocationCallback
private lateinit var locationRequest: LocationRequest
```

Figure 15 – Components

- NotificationManager: Manages notifications.
- FusedLocationProviderClient: Provides location updates.
- LocationRequest: Defines location update conditions (timing, distance).
- LocationCallback: Receives updated location results.
- DataHandling and Application: Handle app data like path and stations.
- NotificationChannel: Creates proper notification channels for Android 8+.

5.1.3 Detailed Breakdown of Functions:

1. onCreate()

- **Initializes everything:**

- Gets the LocationManager.
- Initializes the NotificationManager.
- Creates notification channels (createNotificationChannels()).
- Prepares the LocationRequest (location settings).
- Prepares the LocationCallback (to handle new location updates and send notifications).

```
override fun onCreate() {
    super.onCreate()

    val application = application as Application
    val readAndWriteData = application.readAndWriteData

    locationManager = getSystemService(Context.LOCATION_SERVICE) as LocationManager
    notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
    createNotificationChannels()

    fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
    createLocationRequest()

    locationCallback = object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult) {
            for (location in locationResult.locations) {

                getStationAndNotification(readAndWriteData, application, location)
            }
        }


    }
}
```

Figure 16 – onCreate function

2. onStartCommand()

```
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {

    Application().language
    loadLocale()
    startForeground(NOTIFICATION_ID, createNotification())
    startLocationUpdates()

    return START_STICKY
}
```

Figure 17 – onStartCommand function

- Starts the service as a **Foreground Service**:
  - Loads the current language (loadLocale).
  - Displays a running notification (createNotification()).
  - Starts requesting location updates (startLocationUpdates()).

3. startLocationUpdates()

```
private fun startLocationUpdates() {
    if (ContextCompat.checkSelfPermission(
            context: this,
            android.Manifest.permission.ACCESS_FINE_LOCATION
        ) == PackageManager.PERMISSION_GRANTED
    ) {
        fusedLocationClient.requestLocationUpdates(
            locationRequest,
            locationCallback,
            Looper.getMainLooper()
        )
    }
}
```

Figure 18 – startLocationUpdates function

- Checks for location permissions.
- Starts receiving location updates from Google Location API.

4. getStationAndNotification()

- Finds the **nearest station** based on the user's current location.
- Compares it with the **previous station**.
- Sends a new notification if the user moved forward.
- Saves the current station ID.
- Stops the service if the user reaches the final destination.

```
private fun getStationAndNotification(readAndWriteData: DataHandling, application: Application, location: Location) {
    stationData = application.stationData
    path = application.path.toList()
    language = application.language
    if (language != currentLanguage) {
        loadLocale()
        notificationManager.notify(NOTIFICATION_ID, createNotification())
    }

    nearestStation = LocationCalculations().nearestStationPath(
        stationData, shortest: 500F, path,
        location.latitude, location.longitude)

    val getID = application.previousStationID

    if (getID != 0 && startTrip) {
        previousStation = stationData.firstOrNull { it.id == getID }?.name ?: ""
    }

    if (nearestStation.isNotEmpty() && previousStation!= nearestStation) {
        if (previousStation == "") {
            startTrip =true
            previousStation = nearestStation
        }

        if (previousStation in path && path.indexOf(previousStation) <= path.indexOf(nearestStation)) {
            notifyUsingDistance(nearestStation)
            previousStation = nearestStation

            val id = stationData.firstOrNull { it.name == previousStation }?.id
            if (id != null) {
                readAndWriteData.saveID( context: this@LocationService, id, key: "previousService")

                if (nearestStation == path[path.size - 1]) {
                    readAndWriteData.saveID( context: this@LocationService, data: 0, key: "previousService")
                    stopSelf()
                }
            }
        }
    }
}
```

Figure 19 – getStationAndNotification function

5. **notifyUsingDistance()**

```
private fun notifyUsingDistance(station: String) {
    val intersections = Direction(stationData).findIntersections(path)
    when (station) {
        path[0] -> sendPushNotification(
            getString(R.string.start_station),
            getString(R.string.starting_trip_soon_in_have_a_nice_trip, station),
            stage: "start"
        )
        path[path.size - 1] -> sendPushNotification(
            getString(R.string.arrival_station),
            getString(R.string.reaching_destination_soon_in, station),
            stage: "end"
        )
        in intersections -> sendPushNotification(
            getString(R.string.intersection_station),
            getString(R.string.reaching_an_intersection_soon_in, station),
            stage: "change"
        )
    }
}
```

Figure 20 – notifyUsingDistance function

- Sends different types of notifications depending on the station:
  - **Start station** (trip is beginning).
  - **End station** (trip is ending).
  - **Intersection station** (where the user needs to switch lines).

6. sendPushNotification()

```
private fun sendPushNotification(alertTitle: String, alertMessage: String, stage: String) {
    val a = Intent(applicationContext, TripProgress::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_SINGLE_TOP
    }

    val pendingIntent = if (path.isNotEmpty() && nearestStation != path[path.size - 1]) {
        PendingIntent.getActivity(
            applicationContext,
            requestCode: 0,
            a,
            flags: PendingIntent.FLAG_IMMUTABLE or PendingIntent.FLAG_UPDATE_CURRENT
        )
    } else {
        null
    }

    val bigPicture = BitmapFactory.decodeResource(resources,
        when(stage){
            "start"->R.drawable.start
            "end"->R.drawable.end
            "change"->R.drawable.change
            else -> R.drawable.ic_stat_notificon
        }
    )

    val notification = NotificationCompat.Builder( context: this, PUSH_CHANNEL_ID)
        .setSmallIcon(R.drawable.ic_stat_notificon)
        .setColor(Color.RED)
        .setContentTitle(alertTitle)
        .setContentText(alertMessage)
        .setLargeIcon(bigPicture)
        .setContentIntent(pendingIntent)
        .setPriority(NotificationCompat.PRIORITY_HIGH)
        .setAutoCancel(true)
        .setVibrate(longArrayOf(0, 500, 200, 500))
        .setSound(RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION))
        .build()

    notificationManager.notify( id: 2001, notification)
}
```

Figure 21 – sendNotification function

- Builds a custom push notification with:
  - Title and message.
  - Special icon based on the trip stage (start, end, or change).
  - Vibration and custom notification sound.

7. `createNotificationChannels()`

```
private fun createNotificationChannels() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val serviceChannel = NotificationChannel(
            CHANNEL_ID,
            name: "Location Service Channel",
            NotificationManager.IMPORTANCE_LOW
        )

        val pushChannel = NotificationChannel(
            PUSH_CHANNEL_ID,
            name: "Station Alerts",
            NotificationManager.IMPORTANCE_HIGH
        ).apply {
            description = "Alerts when you arrive at important stations"
            enableLights( lights: true)
            lightColor = Color.RED
            enableVibration( vibration: true)
        }

        notificationManager.createNotificationChannel(serviceChannel)
        notificationManager.createNotificationChannel(pushChannel)
    }
}
```

Figure 22 – createNotificationChannels function

- Creates two notification channels:
  - One for the running location service (LOW importance).
  - One for important station alerts (HIGH importance).

8. `onDestroy()`

- When the service stops:
  - Resets the saved previous station ID.
  - Stops receiving location updates.

```
override fun onDestroy() {

    super.onDestroy()
    DataHandling().saveID( context: this@LocationService,  data: 0,  key: "previousService")
    stopLocationUpdates()
}
```

Figure 23 – onDestroy function

Important Notes:

- **Foreground Service** is critical to prevent Android from killing the service in the background.
- Handles language change live during the trip.
- Supports Android 8.0+ by properly using NotificationChannel.

Languages (Arabic - English)

6.1 Multi-Language Support (Arabic – English)

6.1.1 Language Selection Dialog

Implementation function

```
private fun showLanguageDialog() {
    val languages = arrayOf("\uD83C\uDDFA\uD83C\uDDF8 English", "\uD83C\uDDEA\uD83C\uDDEC العربية   ")
    val languageCodes = arrayOf("en", "ar")

    val builder = MaterialAlertDialogBuilder( context: this, R.style.CustomAlertDialogTheme)
    builder.setTitle("Choose Language")

    builder.setItems(languages) { _, which ->
        val selectedLanguage = languageCodes[which]
        if(selectedLanguage != language) {
            saveStations()
            switchLanguage(selectedLanguage)
        }
        if (selectedLanguage != language && indicator)
        {
            readAndWriteData.extractPathId( context: this,path,stationData)
            readAndWriteData.saveSimpleData( context: this, data: true, key: "languageChange")

        }

    }

    builder.show()
}
```

Figure 24 – Language selection dialog function

Function behavior

- **Dialog Initialization**
  - Displays a language selection dialog with:
    - US English
    - EG العربية
- **Selection Handling**
  - Gets selected language code (en or ar)
  - If different from current language, performs:
    - Saves station data using saveStations()
    - Switches language via switchLanguage()
    - If tracking is active (indicator = true):
      - Reloads path and stations from storage
      - Saves language change flag to shared preferences

6.1.2 Switching Language

Implementation function

```
private fun switchLanguage(lang: String) {
    val prefs: SharedPreferences = getSharedPreferences( name: "savedData", MODE_PRIVATE)
    prefs.edit() {
        putString("My_Lang", lang)
    }

    setLocale(lang)
}
```

Figure 25 – switchLanguage and setLocale functions

Function behavior

- **Saving Preference**
  - Stores selected language in shared preferences under "My_Lang"
- **Applying Locale**
  - Sets the system's Locale to the selected language
  - Updates Configuration with new locale and direction (LTR/RTL)
  - Restarts the Home activity to apply changes

6.1.3 Loading Saved Language Preference

Implementation function

```
private fun loadLocale() {

    val locale = Locale(language)
    Locale.setDefault(locale)

    val config = Configuration(resources.configuration)
    config.setLocale(locale)
    config.setLayoutDirection(locale)

    resources.updateConfiguration(config, resources.displayMetrics)
    baseContext.resources.updateConfiguration(config, baseContext.resources.displayMetrics)
}
```

Figure 26 – loadLocale function

Function behavior

- **Applies Saved Language**
  - Uses stored language variable
  - Sets default locale and applies direction (RTL for Arabic)
  - Updates app resources to reflect selected language

CHAPTER 3:

FLUTTER APPLICATION

3.1 What is Flutter?

Flutter is an open-source UI software development kit created by Google. It allows developers to build natively compiled applications for mobile, web, and desktop using a single codebase. Flutter's layered architecture helps developers control every pixel on the screen, enabling beautiful and customized designs.

3.2 What is Dart?

Dart is the programming language used by Flutter. It is optimized for UI development and offers features like asynchronous programming, strong typing, and object-oriented capabilities. Dart compiles to native code for optimal performance on mobile platforms.

3.3 Packages We Used in This Application

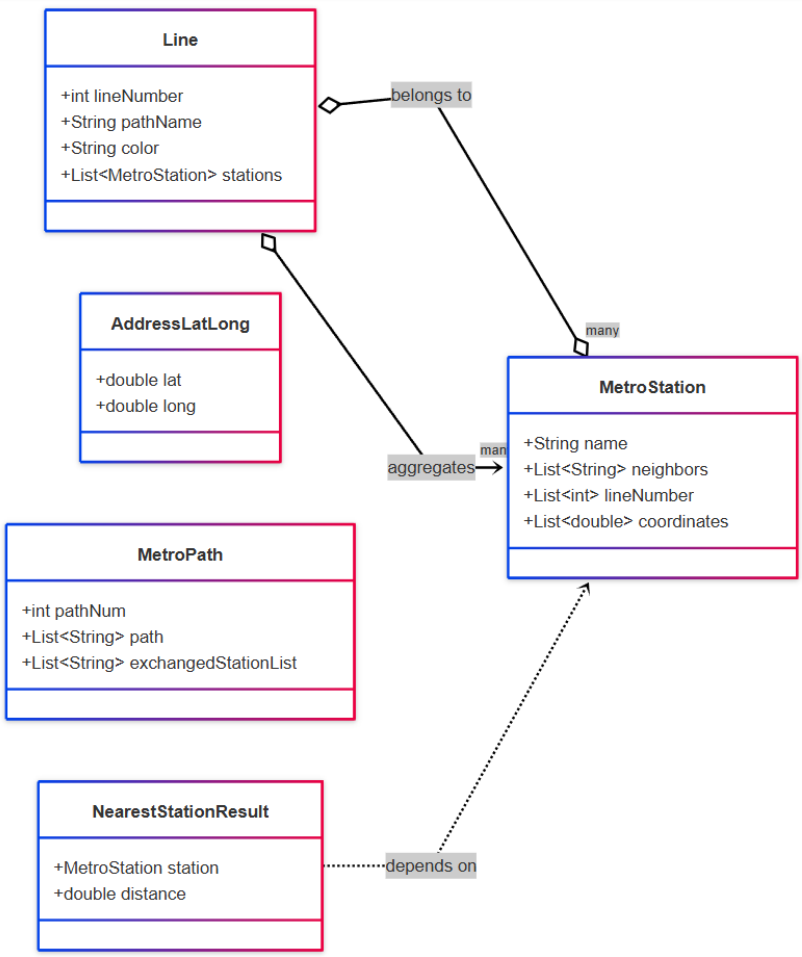
To enhance functionality and simplify development, we used the following packages in the Flutter application:

- `get: ^4.7.2` – State management and routing
- `flutter_launcher_icons: ^0.14.3` – To customize the app icon
- `dartx: ^1.2.0` – Helpful Dart extensions
- `geocoding: ^3.0.0` – Converting coordinates to addresses
- `geolocator: ^13.0.4` – Accessing location information
- `url_launcher: ^6.3.1` – Launching URLs from within the app
- `get_storage: ^2.1.1` – Local storage handling
- `flutter_localizations` & `intl: ^0.19.0` – For Arabic/English translation support
- `awesome_notifications` – Handling rich local notifications
- `WorkManager` – Scheduling background tasks

3.4 What is the Importance of Responsive Design in Flutter?

Responsive design ensures that the app adapts to various screen sizes and orientations, providing a consistent experience across devices. It's crucial in Flutter development to avoid layout issues and offer a user-friendly interface regardless of the device used.

Class module



3.5 Tools of Responsive Design in Flutter

LayoutBuilder

- Builds widgets based on the available space.
- Helps create different UI layouts depending on screen width.
- Ideal for adapting layout to phones vs tablets.

Flexible

- Allows a widget to take up **only the needed space** within a Row or Column.
- Prevents overflow while enabling flexibility in layout.

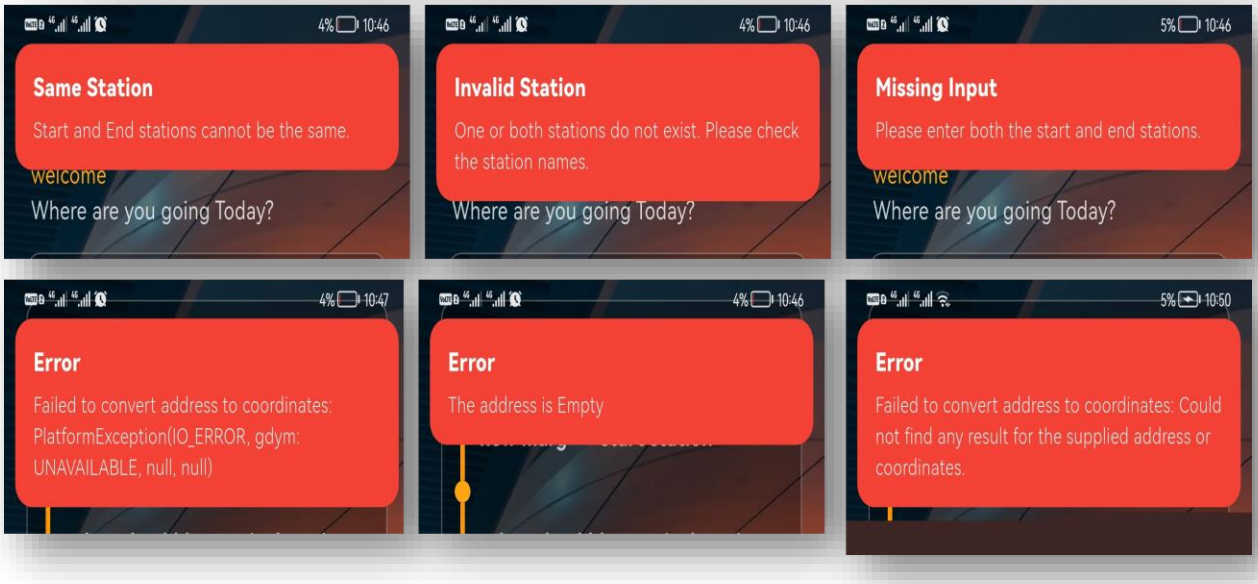
Expanded

- Forces a widget to **take all remaining space** in a Row or Column.
- Useful when you want to evenly distribute space among children.

3.6 About our Application

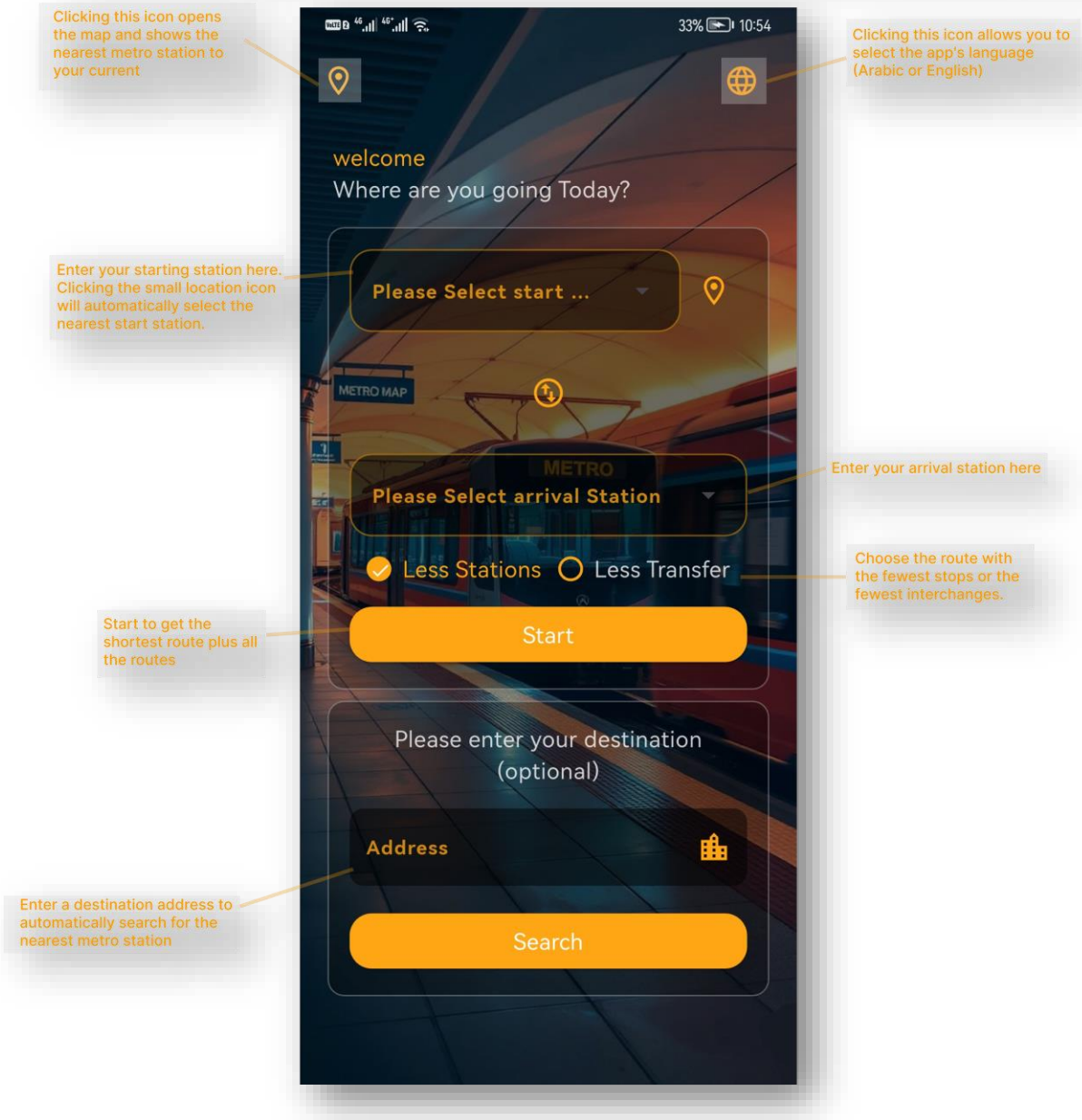
Screens Implementation:

some clear error messages that help users to correct an invalid or missing information.

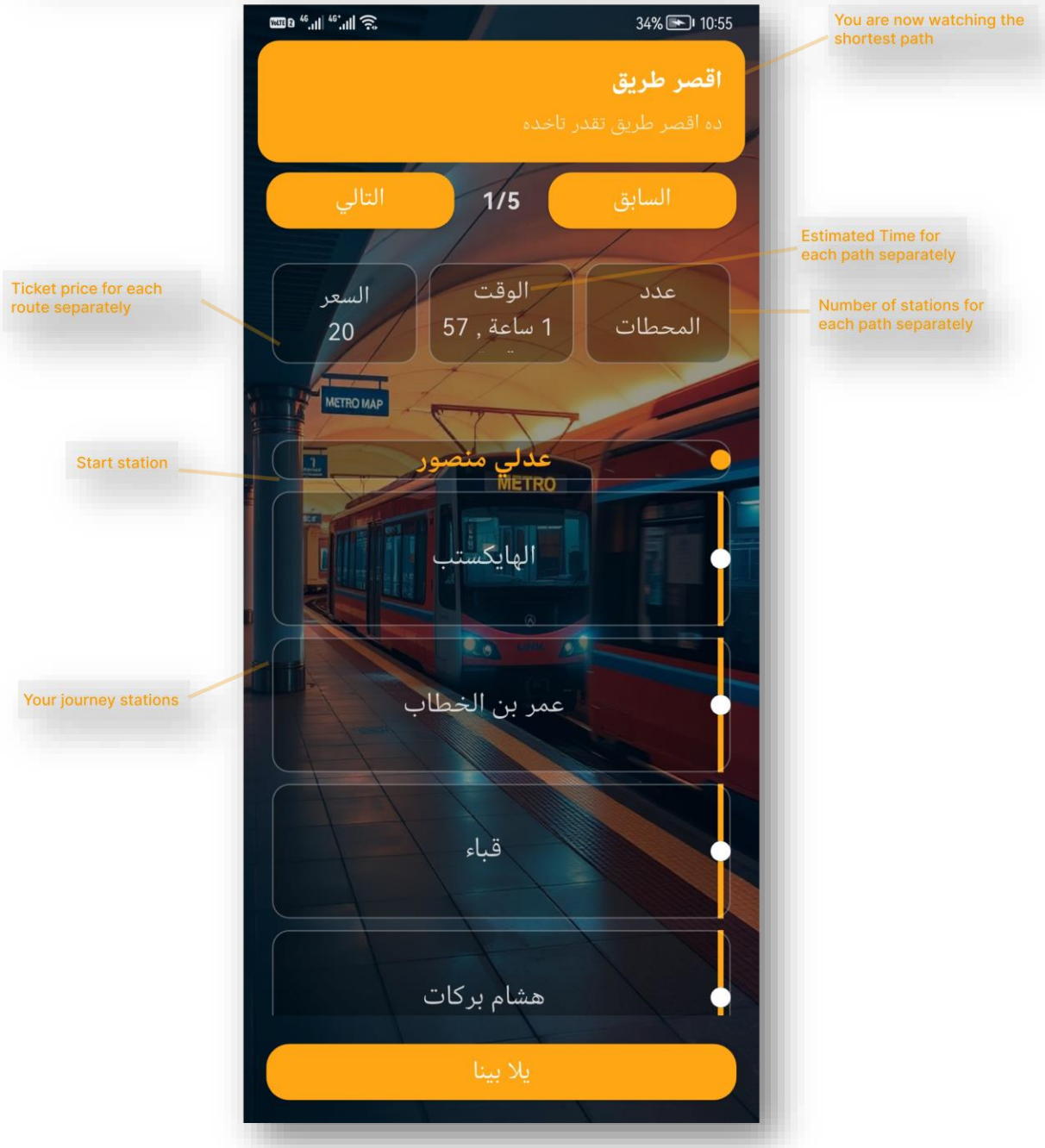


Validation

Implemented to allow the user to: view nearby metro stations, access quick navigation options, and get an overview of all metro lines and available features.



Implemented to allow the user to: view all possible routes between the selected stations, including estimated travel time, total cost, and number of stations for each route option.



Implemented to allow the user to: track their current journey in real time, receive location-based alerts when approaching a station, and get notified at transfer or arrival points.

The page displays updated information including the current station, next station, and the number of remaining stations.

The current station color changes to orange so that the user clearly knows that he has reached it.

Cancel button, if the user presses it, the live tracking will be stopped immediately and the page will be exited

Implemented to allow the user to: receive a dialog pop-up showing the current station if live tracking is activated and the user navigates back to the Home screen, ensuring they stay informed about their journey's progress.

If the journey is active and the user presses the Back button, the app automatically redirects them to the Home page.

When the user presses the view all goes the live track page again.

A dialog will pop up showing the name of the current station.

Code Explanation:

main.dart:

1. Importing Necessary Packages

- The code begins by importing essential packages for the app:
  - flutter/material.dart: Provides core Flutter widgets like buttons, text, and others.
  - flutter_localizations/flutter_localizations.dart: Enables localization support for different languages.
  - get/get.dart: A package for state management and routing in Flutter apps.
  - get_storage/get_storage.dart: A simple storage solution for storing local data.
  - core/bindings/metro_binding.dart: Contains binding configurations for the app.
  - core/routes/app_routes.dart: Defines the routing structure of the app.
  - flutter_gen/gen_l10n/app_localizations.dart: Handles localization of app text based on language settings.

2. The main() Function

- **main()** is the entry point of the app:
  - Initializes **GetStorage** for local storage operations.
  - Calls **runApp()** to start the app with **MyApp**`()` as the root widget.

3. The MyApp Widget

- **MyApp** is the main widget that sets up the app's structure:
  - **GetMaterialApp**: This widget wraps the entire app to enable **GetX** functionalities like state management and routing.
  - **debugShowCheckedModeBanner: false**: Disables the "debug" banner shown during development.
  - **initialBinding: MetroBinding()**: Binds the app to the necessary services and configurations set in **MetroBinding**.
  - **locale**: Sets the app's locale (language) based on the value stored in **GetStorage()**.

- **localizationsDelegates**: Specifies the delegates needed for supporting multiple languages (English and Arabic in this case).
- **supportedLocales**: Defines the supported languages for the app, here it's English and Arabic.
- **initialRoute: '/MetroHome'**: Specifies the initial route to navigate to the **/MetroHome** page.
- **getPages: AppRoutes.routes**: Maps the routes defined in **AppRoutes.routes** to manage navigation.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_localizations/flutter_localizations.dart';
import 'package:get/get.dart';
import 'package:get_storage/get_storage.dart';
import 'core/bindings/metro_binding.dart';
import 'core/routes/app_routes.dart';
import 'package:flutter_gen/gen_l10n/app_localizations.dart';

Future<void> main() async {
  await GetStorage.init();
  runApp(
    const MyApp(),
  );
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(
      debugShowCheckedModeBanner: false,
      initialBinding: MetroBinding(),
      locale: Locale('${GetStorage().read('language')}', ''),
      localizationsDelegates: const [
        AppLocalizations.delegate,
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
        GlobalCupertinoLocalizations.delegate,
      ],
      supportedLocales: const [
        Locale('en', ''), // English
        Locale('ar', ''), // Arabic
      ],
      initialRoute: '/MetroHome',
      getPages: AppRoutes.routes,
    );
  }
}
```

MetroController Class:

1. Controller Initialization (onInit method)

- **MetroController Constructor**: The constructor takes several dependencies like LocationService, PathFinder, TicketService, SortedPaths, ExchangeStation, and MetroRepository, which are injected and used throughout the controller.
- **State Initialization**:
  - **startStation** and **endStation** are initialized to empty strings.
  - **stationsNames**, **allPaths**, **allPathsByExchangedNum**, and **metroPaths** are cleared to reset any previous data.
- **Reactivity**:
  - The ever function is used to listen for changes in metroRepository.stationsNames, updating the list of station names.
  - Another ever listener is set on the getPaths variable, which triggers path-finding when both the start and end stations are defined and getPaths is true.
- **Language Initialization**: The saved language from local storage is retrieved and applied to the app's locale settings.

---

2. Methods in MetroController

- **changeLanguage**: This method updates the language of the app by changing the locale and saving the new language code in local storage. It also updates the language in the repository.
- **findPaths**: This method calculates all possible paths between the start and end stations:
  - **pathFinder.findAllPaths**: Finds all possible paths between the start and end stations.
  - **exchangeStation.getExchangeStations**: Determines stations where exchanges are needed along the path.
  - **sortedPaths.sortMetroPathsByLengthOfStations**: Sorts paths by the number of stations.
  - **sortedPaths.sortMetroPathsByExchangeStations**: Sorts paths based on the number of exchange stations.
- **getTicketPrice**: Uses the TicketService to calculate the ticket price based on the number of stations in a path.
- **updateSelectedTransfer**: Updates the selected transfer option (e.g., "Less Stations").

- **getNearestStation**: This method gets the nearest metro station based on the current location of the user. If isNearStation is true, the station is automatically set as the start station. Otherwise, the method opens a map with the location of the station using a URL launcher.
- **locationFromAddress**: This method finds the nearest station based on a given address by converting the address to latitude and longitude, then comparing the distance to the available stations.
- **metroStationFromStationName**: This method populates the list of stations from the user's selected path, converting each station name into a MetroStation object from the repository.
- **startTracking**: Starts location tracking for the user's selected path by calling locationService.startTracking. It listens for updates on the user's current station and updates currentStation.
- **stopTracking**: Stops location tracking.
- **positionStream**: Checks if the position stream is available. This is useful for handling cases where the position stream might be null or non-null.

---

3. Controller Lifecycle

- **onClose**: This method is called when the controller is disposed of. It resets the startStation and endStation values to empty strings.

```
class MetroController extends GetxController {
  MetroController({
    required this.locationService,
    required this.pathFinder,
    required this.ticketService,
    required this.sortedPaths,
    required this.exchangeStation,
    required this.metroRepository,
  });
final ExchangeStation exchangeStation;
final MetroRepository metroRepository;
final TicketService ticketService;
final PathFinder pathFinder;
final SortedPaths sortedPaths;
final LocationService locationService;
final RxList<String> stationsNames = <String>[].obs;
final startStation = ''.obs;
final endStation = ''.obs;
final selectedTransfers = 'Less Stations'.obs;
final allPaths = <List<String>>[].obs;
final allPathsByExchangedNum = <List<String>>[].obs;
final userSelectedPath = <String>[].obs;
final userSelectedPathToMetroStation = <MetroStation>[].obs;
final metroPaths = <MetroPath>[].obs;
final nearestStation = ''.obs;
```

```
  final currentStation = ''.obs;
  final distance = ''.obs;
  @override
  void onInit() {
    startStation.value = '';
    endStation.value = '';
    stationsNames.clear();
    allPaths.clear();
    allPathsByExchangedNum.clear();
    metroPaths.clear();

    super.onInit();
    ever(metroRepository.stationsNames, (_) {
      stationsNames.assignAll(metroRepository.stationsNames);
    });
    ever(getPaths, (_) {
      if (startStation.value.isNotEmpty &&
          endStation.value.isNotEmpty &&
          getPaths.value != false) {
        findPaths();
      }
    });
    final savedLang = GetStorage().read('language');
    if (savedLang != null) {
      locale.value = Locale(savedLang, '');
    }
  }
  void startTracking() {
    metroStationFromStationName();
    locationService.startTracking(userSelectedPathToMetroStation);
    ever(locationService.currentStation, (String stationName) {
      currentStation.value = stationName;
      print("MetroController currentStation updated to: $stationName");
    });
  }

  void stopTracking() {
    locationService.stopTracking();
  }

  bool positionStream() {
    return locationService.positionStream == null;
    // nullable => return true
    // nonNullable => return false
  }


    @override
    void onClose() {
      startStation.value = '';
      endStation.value = '';
      super.onClose();
    }
  }

```

MetroBinding Class:

MetroBinding is responsible for dependency injection using GetX's Bindings feature.
It ensures that all required classes (dependencies) are properly created and available when the MetroController is used.

Key Points:

- Get.lazyPut is used to lazily initialize each class only when it's needed, improving performance.
- MetroRepository is initialized and its metro station data is loaded via loadJsonData().
- Core algorithm classes like PathFinder, TicketService, SortedPaths, and ExchangeStation are registered here.
- LocationService is also provided to track user location and find nearby stations.
- Finally, MetroController is created with all these dependencies injected.

Importance of Binding:

- Centralizes and organizes dependency management.
- Decouples object creation from business logic, making the code more modular and easier to test.
- Ensures smooth and controlled lifecycle management of your dependencies across the app

```
class MetroBinding extends Bindings {
  @override
  void dependencies() {
    Get.lazyPut(() => PathFinder(metroRepository: Get.find()));
    Get.lazyPut(() => TicketService());
    Get.lazyPut(() => SortedPaths());
    Get.lazyPut(() => ExchangeStation(metroRepository: Get.find()));
    Get.lazyPut(() => MetroRepository());
    Get.lazyPut(() => LocationService());
    Get.find<MetroRepository>().loadJsonData();
    Get.lazyPut(
      () => MetroController(
        pathFinder: Get.find(),
        ticketService: Get.find(),
        sortedPaths: Get.find(),
        exchangeStation: Get.find(),
        metroRepository: Get.find(),
        locationService: Get.find(),
      ),
    );
  }
}
```

AppRoutes Class:

AppRoutes class defines the named routes used in the application. It uses GetX's `GetPage` to set up navigation between screens with optional transitions.

Defined Routes:

- '/MetroHome' → Launches the main metro home screen.
- '/MetroRouteScreen' → Displays available routes between selected stations.
- '/MetroTripProgress' → Shows real-time tracking of the trip, using a zoom transition.

Importance of Routing with GetX:

- Centralizes route management, making navigation easier to maintain and update.
- Allows for named routing which improves readability and decouples navigation logic.
- Supports transitions and bindings, enabling smooth and dynamic screen navigation.

```
class AppRoutes {
  static final routes = [
    GetPage(name: '/MetroHome', page: () => MetroHome()),
    GetPage(name: '/MetroRouteScreen', page: () => MetroRouteScreen()),
    GetPage(
      name: '/MetroTripProgress',
      page: () => MetroTripProgress(),
      transition: Transition.zoom,
    ),
  ];
}
```

PathFinder Class:

This class is responsible for finding all possible paths between two metro stations using Depth-First Search (DFS). It interacts with the MetroRepository to retrieve station data and their neighbors.

Key Method:

- findAllPaths(String start, String end):
  - Returns a list of all valid paths (as lists of station names) between the starting and ending station.
  - Validates that both stations exist before proceeding.

Internal Logic:

- _dfs(...) is a recursive helper function that performs the depth-first traversal. It builds each possible path from the start station to the end station, while avoiding revisiting the same station in one path.

Importance:

- Enables route discovery for user-specified trips.
- Core logic behind displaying multiple navigation options in the app.

```dart
import '../../app/data/repositories/metro_repository.dart';

class PathFinder {
  final MetroRepository metroRepository;
  PathFinder({required this.metroRepository});

  List<List<String>> findAllPaths(String start, String end) {
    if (metroRepository.findStation(start).name.isEmpty ||
        metroRepository.findStation(end).name.isEmpty) {
      return [];
    }
    List<List<String>> allPaths = [];
    _dfs(start, end, [], {}, allPaths);
    return allPaths;
  }

  void _dfs(String current, String end, List<String> path, Set<String> visited,
      List<List<String>> allPaths) {
    path.add(current);
    visited.add(current);

    if (current == end) {
      allPaths.add(List.from(path));
    } else {
      final List<String> currentNeighbors =
          metroRepository.findStation(current).neighbors;
      for (String neighbor in currentNeighbors) {
        if (!visited.contains(neighbor)) {
          _dfs(neighbor, end, path, visited, allPaths);
        }
      }
    }
    path.removeLast();
    visited.remove(current);
  }
}
```

SortedPath Class:

This class provides utility methods for sorting a list of metro paths
based on different criteria to help the user select the most efficient route.

Methods:

- sortMetroPathsByLengthOfStations(List<MetroPath> metroPaths):
  - Sorts the paths in ascending order based on the number of stations.
  - If multiple paths have the same length, the one with fewer exchange stations comes first.

- sortMetroPathsByExchangeStations(List<MetroPath> metroPaths):
  - Sorts the paths in ascending order based on the number of exchange stations.
  - If multiple paths have the same number of exchanges, the shorter one (fewer stations) comes first.

Importance:

- Enhances user experience by presenting sorted and optimized route options.
- Supports prioritization for either fewer stops or fewer line changes depending on user needs.

```
class SortedPaths {
  List<List<String>> sortMetroPathsByLengthOfStations(
      List<MetroPath> metroPaths) {
    return metroPaths
        .sortedBy((e) => e.path.length)
        .thenBy((e) => e.exchangedStationList.length)
        .map((e) => e.path)
        .toList();
  }

  List<List<String>> sortMetroPathsByExchangeStations(
      List<MetroPath> metroPaths) {
    return metroPaths
        .sortedBy((e) => e.exchangedStationList.length)
        .thenBy((e) => e.path.length)
        .map((e) => e.path)
        .toList();
  }
}
```

ExchangeStation Class:

This class is responsible for identifying exchange stations (also known as transfer stations)
within a metro path. An exchange station is where a passenger switches from one metro line to another.

Constructor:
- Takes a reference to the MetroRepository to access station data and line information.

Method:
- getExchangeStations(List<String> path):
- Iterates through the path and checks the line numbers of each consecutive pair of stations.
- If the line changes between two stations, it marks the current station as an exchange station.
- Returns a list of all exchange stations in the given path.

Importance:
- Essential for helping users understand where they need to change lines.
- Used in calculating the complexity of the trip (e.g., how many line changes are required).
- Supports the sorting and optimization of routes based on the number of exchanges.

```
class ExchangeStation {
  final MetroRepository metroRepository;
  ExchangeStation({required this.metroRepository});
  List<String> getExchangeStations(final List<String> path) {
    final List<String> transferStations = [];
    int? currentLine;
    for (int i = 0; i < path.length - 1; i++) {
      final String stationName = path[i];
      final String nextStationName = path[i + 1];
      final List<int> stationLines =
          metroRepository.findStation(stationName).lineNumber;
      final List<int> nextStationLines =
          metroRepository.findStation(nextStationName).lineNumber;
      final int newLine = stationLines.firstWhere(
          (line) => nextStationLines.contains(line),
          orElse: () => -1);

      if (newLine != -1 && newLine != currentLine) {
        if (transferStations.isNotEmpty || i > 0) {
          transferStations.add(stationName);
        }
        currentLine = newLine;
      }
    }
    return transferStations;
  }
}
```

TicketService Class:

This class is responsible for calculating the metro ticket price based on the number of stations traveled.

Method:

- calculateTicketPrice(int stationCount):
  - Takes the total number of stations between the start and end of a trip.
  - Returns the ticket price according to predefined pricing tiers:
    - Up to 9 stations: 8 EGP
    - 10 to 16 stations: 10 EGP
    - 17 to 23 stations: 15 EGP
    - More than 23 stations: 20 EGP

Importance:

- Central to providing fare estimation to users during route planning.
- Simplifies logic by encapsulating ticket pricing rules in a single method.
- Enhances maintainability by allowing easy updates to ticket pricing policy if needed.

```
class TicketService {
  int calculateTicketPrice(int stationCount) {
    if (stationCount <= 9) return 8;
    if (stationCount <= 16) return 10;
    if (stationCount <= 23) return 15;
    return 20;
  }
}
```

LocationService Class:

This class provides all location-related functionalities for the Metro app, including permission handling, user location retrieval, station proximity detection, and real-time tracking of user's movement along their metro route.

Key Responsibilities:

1. permissions():
- Requests and checks for location permissions.
- Notifies the user with appropriate messages if services are disabled or permissions are denied.

2. getCurrentLocation():
- Returns the user's current geographic coordinates after ensuring permissions are granted.

3. distanceBetweenStations(List<MetroStation>, AddressLatLong):
- Compares the user's location with all available metro stations.
- Finds and returns the nearest station along with the calculated distance.

4. getAddressLatLong(String address):
- Converts a written address into geographical coordinates.
- Useful for enabling location search functionality based on user input.

5. startTracking(RxList<MetroStation> routeStations):
- Starts a live tracking service to monitor the user's proximity to each station in their route.
- Automatically detects when the user reaches a new station or the final station.
- Updates the `currentStation` observable with the closest upcoming station.
- Stops tracking upon reaching the destination.

6. stopTracking():
- Cancels the location stream and disables tracking.

Importance:
- This service plays a vital role in the "Metro Trip Progress" feature by enhancing user experience through real-time location updates.
- Improves accessibility by detecting nearest station to user’s current or entered location.

- Encourages safe navigation by alerting users as they approach or reach their destination.
- Integrates tightly with the UI through reactive state management and user notifications.

```
class LocationService {
Future<void> permissions() async {
  bool serviceEnabled;
  LocationPermission permission;

  serviceEnabled = await Geolocator.isLocationServiceEnabled();
  if (!serviceEnabled) {
    Get.snackbar(
      'Metro Cairo ',
      'Location services are disabled.',
      backgroundColor: Colors.red,
      colorText: Colors.white,
    );
  }
  permission = await Geolocator.checkPermission();
  if (permission == LocationPermission.denied) {
    permission = await Geolocator.requestPermission();

    if (permission == LocationPermission.denied) {
      Get.snackbar(
        'Metro Cairo',
        'Location permissions are denied',
        backgroundColor: Colors.red,
        colorText: Colors.white,
      );
      //  return null ;
    }
    if (permission == LocationPermission.deniedForever) {
      // Permissions are denied forever, handle appropriately.
      Get.snackbar(
        'Metro Cairo',
        'Location permissions are permanently denied, we cannot request permissions.',
        backgroundColor: Colors.red,
        colorText: Colors.white,
      );
      // return null  ;
    }
  }
}

Future<AddressLatLong> getCurrentLocation() async {
  await permissions();
  final location = await Geolocator.getCurrentPosition();
  return AddressLatLong(lat: location.latitude, long: location.longitude);
}

NearestStationResult distanceBetweenStations(
    List<MetroStation> stations, AddressLatLong currentLocation) {
  double minDistance = double.infinity;
  MetroStation nearestStation = stations[0];
  for (var station in stations) {
    double distance = Geolocator.distanceBetween(currentLocation.lat,
        currentLocation.long, station.coordinates[0], station.coordinates[1]);
    if (distance < minDistance) {
      minDistance = distance;
```

```
      nearestStation = station;
    }
  }
  return NearestStationResult(nearestStation, minDistance);
}

Future<AddressLatLong> getAddressLatLong(String address) async {
  await permissions();
  try {
    List<Location> locations = await locationFromAddress(address);
    if (locations.isEmpty) {
      Get.snackbar(
        'Error',
        'Invalid or non-existent address',
        backgroundColor: Colors.red,
        colorText: Colors.white,
      );
    }
    return AddressLatLong(
      lat: locations[0].latitude,
      long: locations[0].longitude,
    );
  } catch (e) {
    Get.snackbar(
      'Error',
      'Failed to convert address to coordinates: $e',
      backgroundColor: Colors.red,
      colorText: Colors.white,
    );
    throw Exception("إحداثيات إلى العنوان تحويل في فشل: $e");
  }
}

final currentStation = ''.obs;
bool isTracking = true; // used to enable tracking service
StreamSubscription<Position>? positionStream;
void startTracking(RxList<MetroStation> routeStations) async {
  if (!isTracking) return;
  await permissions();
  const LocationSettings locationSettings = LocationSettings(
    accuracy: LocationAccuracy.high,
    distanceFilter: 10,
  );
  final lastStation = routeStations.last;
  positionStream =
      Geolocator.getPositionStream(locationSettings: locationSettings)
          .listen((Position position) {
    double userLat = position.latitude;
    double userLng = position.longitude;

    for (var station in routeStations) {
      double distance = Geolocator.distanceBetween(
        userLat,
        userLng,
        station.coordinates[0],
        station.coordinates[1],
      );
      print('Tracking On');
      if (distance <= 200) {
        print('لمحطة توصل قربت ${station.name}$distance');
        if (currentStation.value != station.name) {
          currentStation.value = station.name;
        }
```

```
        if (station.name == lastStation.name) {
          print("وصلت للمحطة الأخيرة: ${station.name}");
          stopTracking();
        }
        break;
      }
    }
  });
}

void stopTracking() {
    if (positionStream != null) {
      positionStream!.cancel();
      positionStream = null;
    }
    isTracking = false;
    print("✅ تم إيقاف التتبع");
  }
}
```

LocationServiceBackground:

```
Future<void> initialize() async {
  if (isInitialized) return;

  metroRepository = Get.find<MetroRepository>();
  exchangeStation = Get.find<ExchangeStation>();
  locationService = Get.find<LocationService>();

  await setupNotifications();

  isInitialized = true;
}
```

```
Future<bool> startLocationTracking() async {
  if (!isInitialized) {
    await initialize();
  }

  final FlutterBackgroundAndroidConfig androidConfig = FlutterBackgroundAndroidConfig(
    notificationTitle:
        AppLocalizations.of( context: Get.context!)!.notificationServiceTitle,
    notificationText:
        AppLocalizations.of( context: Get.context!)!.notificationServiceText,
    notificationImportance: AndroidNotificationImportance.high,
    notificationIcon: flutter_background.AndroidResource(
        name: 'notification_icon', defType: 'drawable'),  // flutter_background.AndroidResource
    enableWifiLock: true,
  );  // FlutterBackgroundAndroidConfig

  final bool backgroundInitialized =
      await FlutterBackground.initialize(androidConfig: androidConfig);

  if (backgroundInitialized) {
    if (!FlutterBackground.isBackgroundExecutionEnabled) {
      await FlutterBackground.enableBackgroundExecution();
    }
  }
  startTrip = false;
  checkLocation();
  startLocationUpdates();
  return true;
}
```

```
2 usages
Future<void> stopLocationTracking() async {
  GetStorage().write( key: 'tracking',  value: false);
  GetStorage().write( key: 'path',  value: []);
  if (locationTimer != null) {
    locationTimer!.cancel();
    locationTimer = null;
  }

  startTrip = false;
  previousStation = '';

  if (FlutterBackground.isBackgroundExecutionEnabled) {
    await FlutterBackground.disableBackgroundExecution();
  }
}
```

```
Future<void> checkLocation() async {
  bool isTracking = GetStorage().read( key: 'tracking') ?? false;
  if (!isTracking) return;
  final  AddressLatLong  currentLocation = await locationService.getCurrentLocation();
  print(
      object: 'Current location obtained: ${currentLocation.lat}, ${currentLocation.long}');

  final  dynamic  path = GetStorage().read( key: 'path') ?? [];

  if (path.isEmpty) return;

  final  NearestStationResult  nearestStationResult = locationService.distanceBetweenStations(
      stations: metroRepository.stations, currentLocation);

  final  String  nearestStation = nearestStationResult.station.name;
  final  double  distance = nearestStationResult.distance;

  print( object: 'distance : $distance');
  if (nearestStation == path[0] && !startTrip && distance <= 1000) {
    startTrip = true;
    previousStation = nearestStation;

    await sendStationNotification(
        title: AppLocalizations.of( context: Get.context!)!.notificationStartTitle,
        message: AppLocalizations.of( context: Get.context!)!
            .notificationStartMessage( station: nearestStation),
        type: 'start');
  }

  if (startTrip) {
    await findNearestStationAndNotify(currentLocation, path);
  }
}
```

This class provides background location tracking functionality for metro station proximity detection and notification delivery. It monitors the user's location and sends notifications when approaching metro stations along a predefined path.

Key Features

- **Background location tracking with periodic updates**
- **Metro station proximity detection (within 1000 meters)**
- **Smart notifications for:**
  - **Trip start/end points**
  - **Intersection/exchange stations**
  - **Regular station approaches**
- **Android background execution handling**
- **Notification channel setup and management**

Main Components

- **Initialization: Sets up dependencies, notification channels, and background services**
- **Tracking Control: startLocationTracking() and stopLocationTracking() methods**
- **Location Monitoring: Periodic checks every 5 seconds**
- **Notification Logic: Determines appropriate notification types based on station position in route**

Usage Flow

1. **Initialize the service**
2. **Start tracking with a predefined path**
3. **Service automatically detects station approaches**
4. **Sends context-aware notifications**
5. **Stops automatically when reaching final station or manually**

The class integrates with several packages including AwesomeNotifications for alerts and FlutterBackground for maintaining background execution.

Conclusion

The development of the Metro Cairo application represents a significant step toward enhancing urban mobility through technology. By integrating real-time location tracking, optimized route planning, and user-friendly interfaces, the app delivers a seamless commuting experience for metro users in Cairo. Each service, from station detection to trip progress tracking, was carefully designed with the user in mind, aiming to save time, reduce confusion, and improve safety during travel.

We believe this project lays the groundwork for future innovations in smart public transportation solutions and look forward to expanding its capabilities in future releases.