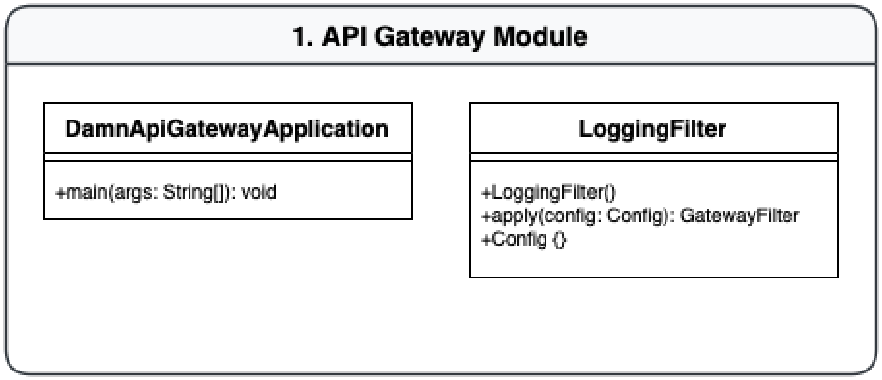
components of our system and the dynamic interactions between them, offering insight into how user-centric sustainable routes are generated and delivered.

Our system comprises a **Flutter-based front-end** that interfaces with a **Spring Boot-driven microservice backend**, built using modular design principles. The frontend handles user interactions and data presentation, while the backend manages business logic, route computation, and sustainability analysis. In the project, DAMN refers to Dublin Area Multimodal Navigation System.

Backend Modules

1. API Gateway


1. API Gateway Module
DamnApiGatewayApplication
+main(args: String[]): void
LoggingFilter
+LoggingFilter()
+apply(config: Config): GatewayFilter
+Config {}

The **API Gateway Module** serves as the central entry point for all client requests to the system. It handles routing, request filtering, and basic logging functionalities before delegating calls to the appropriate microservices.

- *DamnApiGatewayApplication* - the main Spring Boot application class responsible for bootstrapping the gateway service and managing the routing configuration.
- *LoggingFilter* - a custom filter that intercepts incoming requests, enabling logging of request metadata such as headers, URIs, and method types—essential for monitoring and debugging.

This module plays a critical role in cross-cutting concerns like logging, request validation, and future scalability enhancements such as rate limiting or authentication

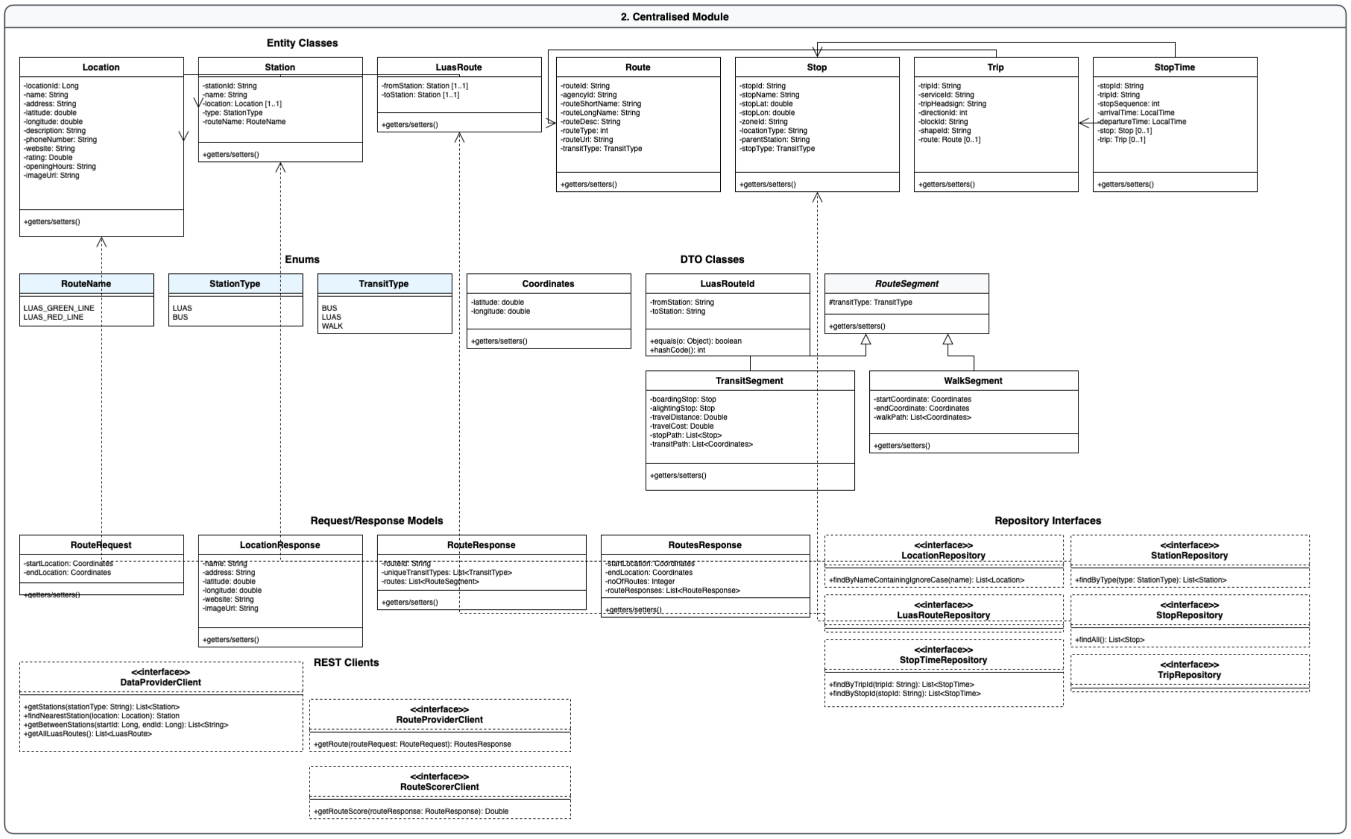
2. DAMN Common Module

The Common Core Module forms the heart of our software architecture, acting as a shared foundation for data modeling, service interaction, and domain consistency across all backend services. By centralizing key components such as entities, data transfer objects (DTOs), enums, and REST client interfaces, this module promotes reusability, reduces duplication, and ensures consistent behavior across the distributed microservices. At its core, the module defines several *entity classes* that represent the real-world structure of transit and geographic data.

To standardize behavior and enable clear classification, the module includes custom *enumerations*, such as:
- *Route-Name* (e.g., LUAS_GREEN_LINE),
- *Station-Type* (e.g., LUAS, BUS),
- *Transit-Type* (e.g., WALK, LUAS, BUS).

The *DTO classes and route segment models* encapsulate how complex journeys are composed and transferred across services. Classes like TransitSegment and WalkSegment abstract the components of a journey, while structures such as RouteSegment and LuasRouteId help define transitions and specific identifiers within multimodal travel routes.

For inter-service communication, the module also defines a set of request and response models such as RouteRequest, RouteResponse, and RoutesResponse. These models act as a contract between front-end clients and back-end microservices, ensuring reliable and predictable exchanges of data.



To support modular service integration, a collection of **REST client interfaces** is provided:

- DataProviderClient, RouteProviderClient, and RouteScorerClient describe how services exchange data like station details, route computations, and sustainability scores.

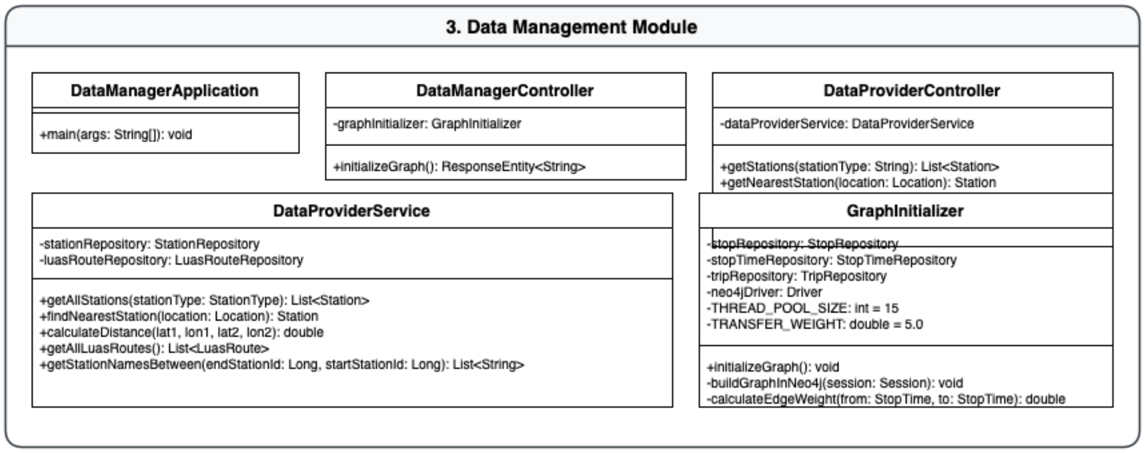
Finally, the module also defines a set of repositories which provide access to core datasets and include specialized queries offering flexible access patterns for different service layers. Altogether, the Common Core Module provides a unified, extensible, and maintainable base that supports the system's broader goals of dynamic, sustainable wayfinding. It abstracts complexity while preserving rich, real-world semantics—making it a vital part of architecture.

3. Data Management Module

This module handles the preparation, organization, and delivery of core transit data used across the system. It plays a dual role as both a data provider and a graph initializer for route computation. The *DataManagerApplication* boots the service, coordinating two key controllers:

- *DataManagerController* initializes the transit graph structure via GraphInitializer.
- *DataProviderController* exposes endpoints to fetch stations and nearest locations. This service accesses transit repositories to:
    - Retrieve all stations by type,
    - Find nearest stations to a given location,
    - Calculate distances,
    - Fetch LUAS route paths between stations and other paths for walking or biking.
- *GraphInitializer* builds a weighted transit graph using data from stop, trip, and stop-time repositories. It applies configurable parameters like thread pool size and transfer weight to support optimized pathfinding.

This module ensures that route generation services receive structured, preprocessed data for accurate and efficient decision-making.

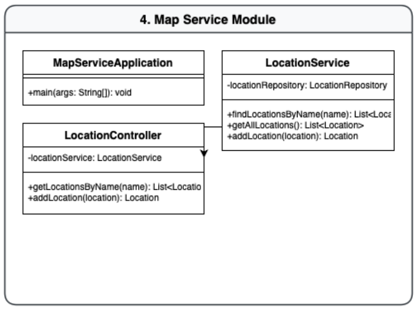

3. Data Management Module
DataManagerApplication
+main(args: String[]): void
DataManagerController
-graphInitializer: GraphInitializer
+initializeGraph(): ResponseEntity<String>
DataProviderController
-dataProviderService: DataProviderService
+getStations(stationType: String): List<Station>
+getNearestStation(location: Location): Station
DataProviderService
-stationRepository: StationRepository
-luasRouteRepository: LuasRouteRepository
+getAllStations(stationType: StationType): List<Station>
+findNearestStation(location: Location): Station
+calculateDistance(lat1, lon1, lat2, lon2): double
+getAllLuasRoutes(): List<LuasRoute>
+getStationNamesBetween(endStationId: Long, startStationId: Long): List<String>
GraphInitializer
-stopRepository: StopRepository
-stopTimeRepository: StopTimeRepository
-tripRepository: TripRepository
-neo4jDriver: Driver
-THREAD_POOL_SIZE: int = 15
-TRANSFER_WEIGHT: double = 5.0
+initializeGraph(): void
-buildGraphInNeo4j(session: Session): void
-calculateEdgeWeight(from: StopTime, to: StopTime): double

4. Map Service Module

The Map Service Module manages location data, acting as a dedicated service for handling operations related to places on the map—like stops, stations, or points of interest. The module is bootstrapped by the *MapServiceApplication*. At its core is the *LocationService*, which interfaces with the *LocationRepository* to:

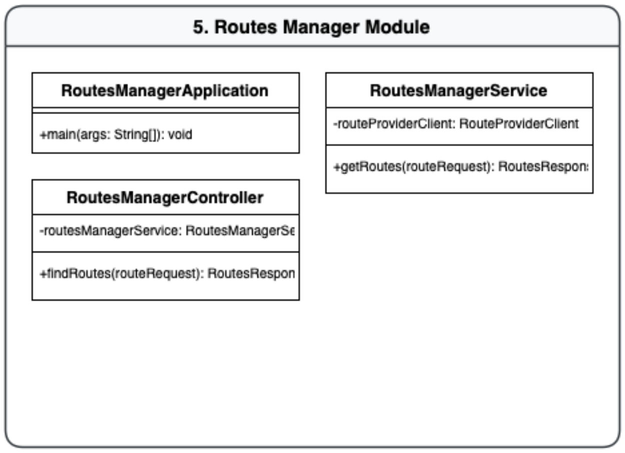
- Retrieve all available locations (getAllLocations()),
- Perform location name-based searches (findLocationsByName()),
- Add new location entries into the database (addLocation()).

These operations are exposed through the *LocationController*, which acts as the API layer, making the service easily consumable by other modules and the frontend.


4. Map Service Module
MapServiceApplication
+main(args: String[]): void
LocationService
-locationRepository: LocationRepository
+findLocationsByName(name): List<Loca
+getAllLocations(): List<Location>
+addLocation(location): Location
LocationController
-locationService: LocationService
+getLocationsByName(name): List<Locatio
+addLocation(location): Location

This module ensures decoupled, centralized management of geospatial data, making it easier to extend or integrate with mapping and visualization components in the future.

5. Routes Manager Module


5. Routes Manager Module
RoutesManagerApplication
+main(args: String[]): void
RoutesManagerService
-routeProviderClient: RouteProviderClient
+getRoutes(routeRequest): RoutesRespon:
RoutesManagerController
-routesManagerService: RoutesManagerSe
+findRoutes(routeRequest): RoutesRespon

The *Routes Manager Module* acts as a coordination layer between the frontend and the backend route computation services. It's responsible for handling user route requests and forwarding them to the core processing services behind the scenes.

- The module starts with *RoutesManagerApplication*, which initializes the service.
- Incoming requests are captured by *RoutesManagerController*, particularly through the *findRoutes*() method.
- The controller delegates these requests to *RoutesManagerService*, which in turn uses the *RouteProviderClient* to fetch computed routes from the Route Provider module.

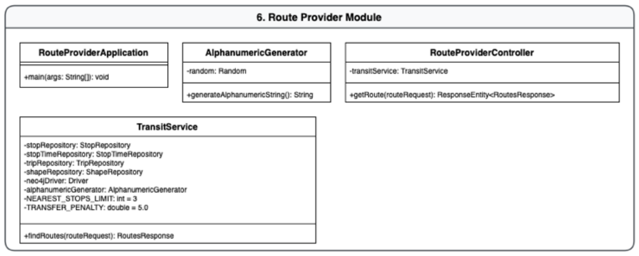
This setup promotes a clean separation of responsibilities, enabling the system to scale and evolve routing logic independently from the interface layer. It also simplifies request routing and service chaining in a microservice architecture.

6. Route Provider Module

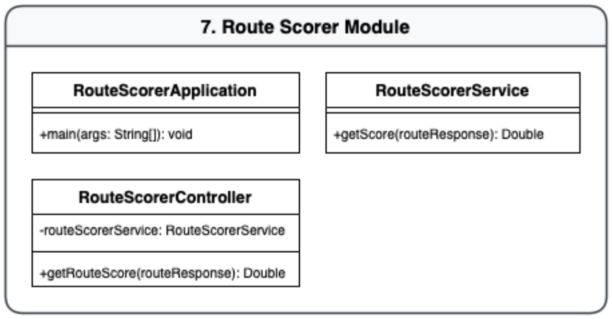
The *Route Provider Module* is the computational engine of the system—responsible for generating optimized, multimodal transit routes based on various factors such as distance, time, and transfer penalties. It is initialized by the RouteProviderApplication.

- *TransitService* is the heart of the module. It interfaces with multiple repositories (*StopRepository, StopTimeRepository, TripRepository, ShapeRepository*) to gather transit data and uses a neo4jDriver to perform graph-based route calculations.
- The service considers constraints like:
    - NEAREST_STOPS_LIMIT – how many nearby stops to evaluate,
    - TRANSFER_PENALTY – the weight added for switching modes of transport.
- *RouteProviderController* exposes a REST endpoint to receive RouteRequest objects and return a structured RoutesResponse.

This module ensures routes are generated with speed, flexibility, and sustainability in mind, leveraging graph traversal techniques and data-driven constraints to deliver practical results to the user.


6. Route Provider Module
RouteProviderApplication
+main(args: String[]): void
AlphanumericGenerator
-random: Random
+generateAlphanumericString(): String
RouteProviderController
-transitService: TransitService
+getRoute(routeRequest): ResponseEntity<RoutesResponse>
TransitService
-stopRepository: StopRepository
-stopTimeRepository: StopTimeRepository
-tripRepository: TripRepository
-shapeRepository: ShapeRepository
-neo4jDriver: Driver
-alphanumericGenerator: AlphanumericGenerator
-NEAREST_STOPS_LIMIT: int = 3
-TRANSFER_PENALTY: double = 5.0
+findRoutes(routeRequest): RoutesResponse

7. Route Scorer Module


7. Route Scorer Module
RouteScorerApplication
+main(args: String[]): void
RouteScorerService
+getScore(routeResponse): Double
RouteScorerController
-routeScorerService: RouteScorerService
+getRouteScore(routeResponse): Double

The Route Scorer Module is dedicated to evaluating the sustainability and efficiency of generated routes. It calculates a numerical score based on various parameters in the RouteResponse object, helping the system rank and present the most suitable paths to users.

- The main logic resides in the RouteScorerService, which processes a RouteResponse and returns a double value as a score via the getScore() method.
- The RouteScorerController exposes this functionality through the getRouteScore() endpoint, making it accessible to other backend services like the Route Manager or Recommendation engine.

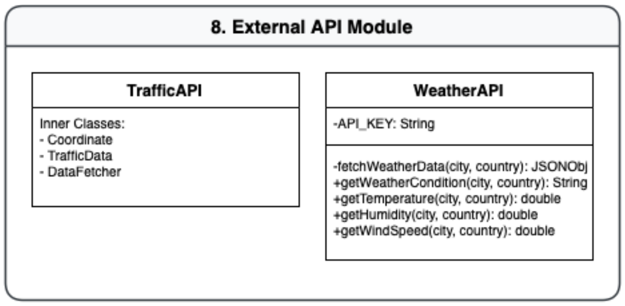
By isolating the scoring logic, this module ensures scalability, maintainability, and future extensibility, such as incorporating carbon footprint data, user preferences, or contextual factors like traffic or weather.

8. External APIs

This is an abstraction layer over third-party data sources, enriching the system with real-time contextual information such as weather and traffic conditions—both of which can influence route suitability and user comfort.

- TrafficAPI offers access to traffic-related data, supported by internal classes like:
  - Coordinate: to represent locations,
  - TrafficData: for congestion or flow information,
  - DataFetcher: a utility to pull traffic data from external sources (implementation assumed).
- WeatherAPI encapsulates weather functionality with methods like:
  - fetchWeatherData(city, country): retrieves full weather info as JSON,
  - getTemperature(), getHumidity(), getWindSpeed(), and getWeatherCondition() return specific metrics.

With a simple, modular design, this API layer allows seamless integration of environmental intelligence into the route planning and scoring process—paving the way for more realistic, adaptive, and user-friendly recommendations.

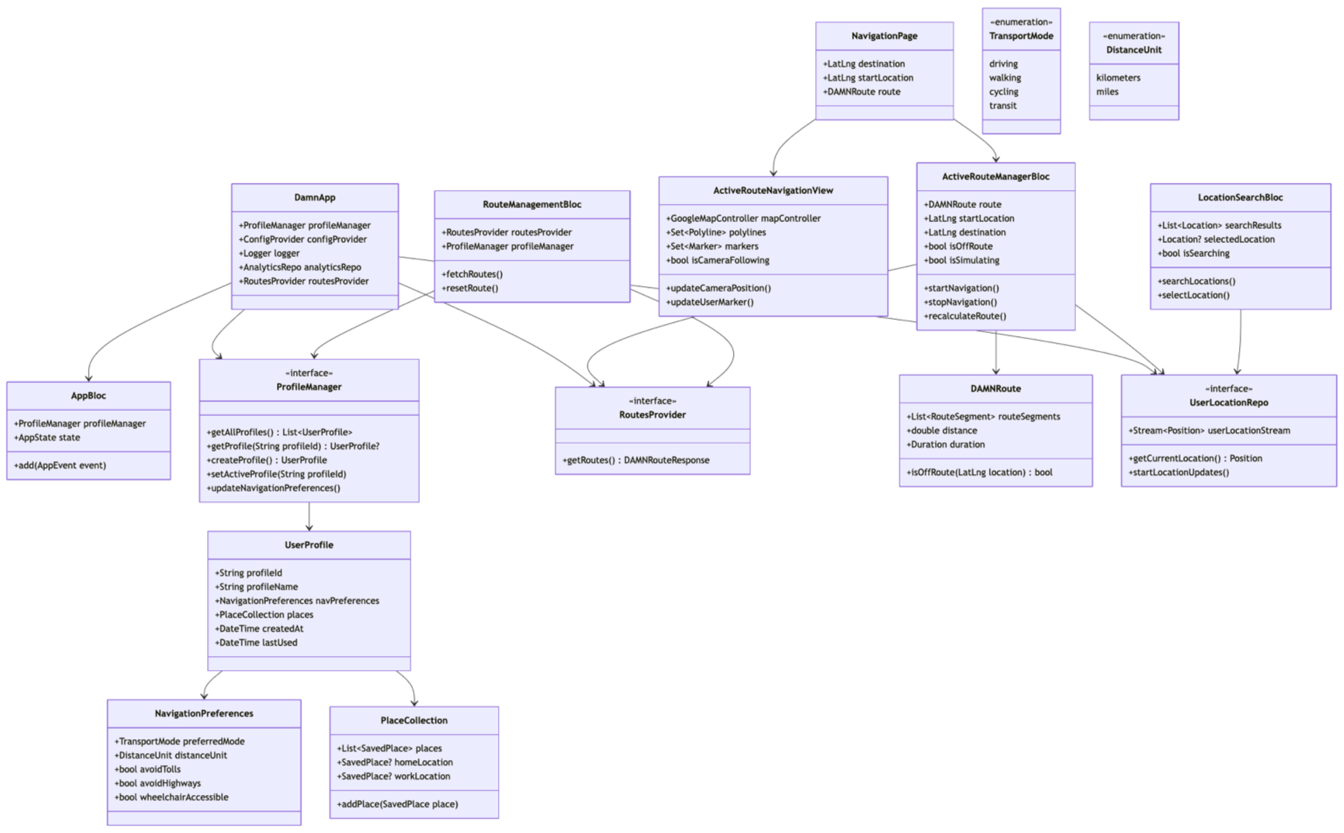

8. External API Module
TrafficAPI
Inner Classes:
- Coordinate
- TrafficData
- DataFetcher
WeatherAPI
-API_KEY: String
-fetchWeatherData(city, country): JSONObj
+getWeatherCondition(city, country): String
+getTemperature(city, country): double
+getHumidity(city, country): double
+getWindSpeed(city, country): double

Frontend Modules

The **Frontend Module**, developed using Flutter, serves as the primary user interface for the system, enabling real-time interaction with sustainable route data. Architected with modularity and responsiveness in mind, it leverages the BLoC (Business Logic Component) pattern to separate business logic from UI, ensuring maintainability and scalability as the app evolves. Please look at the figure given in the next page.

At the heart of the application is the *DamnApp* class, which bootstraps the system and wires up core dependencies such as the *ProfileManager*, *RoutesProvider*, and *AnalyticsRepo*. This centralized management structure ensures that global services like configuration, logging, and routing remain accessible across the app lifecycle.

For route operations, the *RouteManagementBloc* manages state related to route fetching and resetting. When a user initiates a search, this bloc coordinates with the *RoutesProvider* interface to retrieve route data, which is then passed to UI components for rendering. Meanwhile, the *ActiveRouteNavigationView* handles the visual representation of routes, using a *GoogleMapController* to display the map, markers, and polylines. It also supports dynamic camera movement and marker updates based on the user's movement.


NavigationPage
+LatLng destination
+LatLng startLocation
+DAMNRoute route
«enumeration»
TransportMode
driving
walking
cycling
transit
«enumeration»
DistanceUnit
kilometers
miles
DamnApp
+ProfileManager profileManager
+ConfigProvider configProvider
+Logger logger
+AnalyticsRepo analyticsRepo
+RoutesProvider routesProvider
RouteManagementBloc
+RoutesProvider routesProvider
+ProfileManager profileManager
+fetchRoutes()
+resetRoute()
ActiveRouteNavigationView
+GoogleMapController mapController
+Set<Polyline> polylines
+Set<Marker> markers
+bool isCameraFollowing
+updateCameraPosition()
+updateUserMarker()
ActiveRouteManagerBloc
+DAMNRoute route
+LatLng startLocation
+LatLng destination
+bool isOffRoute
+bool isSimulating
+startNavigation()
+stopNavigation()
+recalculateRoute()
LocationSearchBloc
+List<Location> searchResults
+Location? selectedLocation
+bool isSearching
+searchLocations()
+selectLocation()
AppBloc
+ProfileManager profileManager
+AppState state
+add(AppEvent event)
«interface»
ProfileManager
+getAllProfiles() : List<UserProfile>
+getProfile(String profileId) : UserProfile?
+createProfile() : UserProfile
+setActiveProfile(String profileId)
+updateNavigationPreferences()
«interface»
RoutesProvider
+getRoutes() : DAMNRouteResponse
DAMNRoute
+List<RouteSegment> routeSegments
+double distance
+Duration duration
+isOffRoute(LatLng location) : bool
«interface»
UserLocationRepo
+Stream<Position> userLocationStream
+getCurrentLocation() : Position
+startLocationUpdates()
UserProfile
+String profileId
+String profileName
+NavigationPreferences navPreferences
+PlaceCollection places
+DateTime createdAt
+DateTime lastUsed
NavigationPreferences
+TransportMode preferredMode
+DistanceUnit distanceUnit
+bool avoidTolls
+bool avoidHighways
+bool wheelchairAccessible
PlaceCollection
+List<SavedPlace> places
+SavedPlace? homeLocation
+SavedPlace? workLocation
+addPlace(SavedPlace place)

The navigation logic is governed by the *ActiveRouteManagerBloc*, which controls the flow of an active route-handling operations such as starting, stopping, and recalculating navigation. This layer ensures the app responds intelligently as the user moves, enhancing the sense of guidance and direction.

Location-related features are managed by the *LocationSearchBloc*, which supports location lookup, selection, and live tracking. Behind the scenes, real-time location data is streamed via the *UserLocationRepo*, which allows the app to adapt instantly as users travel through their selected routes. A key part of the frontend architecture is its user customization capabilities. The *ProfileManager* allows users to create and manage multiple *UserProfiles*, each storing individual preferences and saved places. Preferences such as transport mode (e.g., walking, cycling, transit), distance units, and accessibility considerations are handled via the *NavigationPreferences* class. These are linked with a *PlaceCollection*, which stores frequently used locations like home and work, further enhancing the user experience.

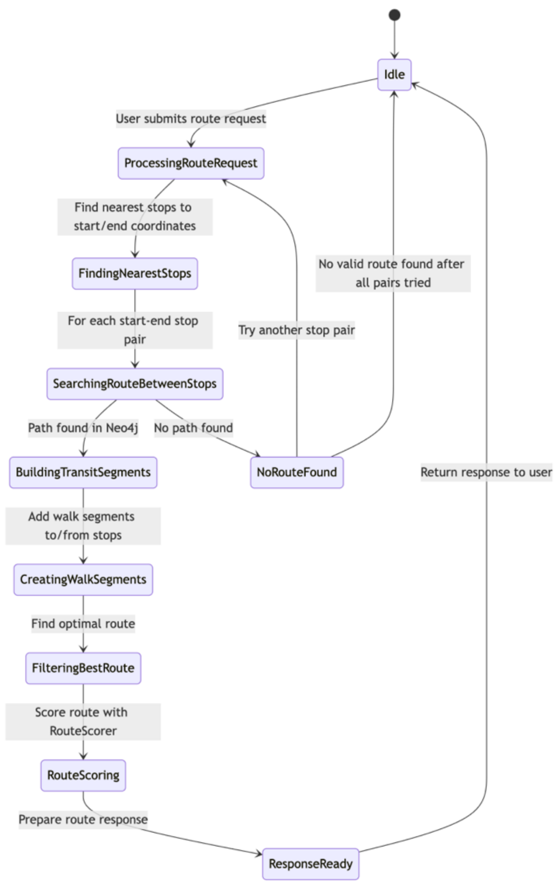
Routes are encapsulated within the *DAMNRoute* class, which contains details like a list of *RouteSegments*, total distance, and estimated duration. It also provides logic for determining if the user has deviated from the current route, allowing for responsive rerouting when needed.

2. Detailed System Behavioral Models

To capture the flow of interactions and lifecycle of key operations, let's focus on state diagrams and sequence diagrams. These behavioral models illustrate how components respond to events over time, how data flows between services, and how the system gracefully handles both success and failure paths. From tracking a user's location to calculating sustainable routes across multimodal networks, these diagrams help visualize the real-time intelligence and coordination at the heart of our application. The figure below shows how the system behaves when a new route request comes in.

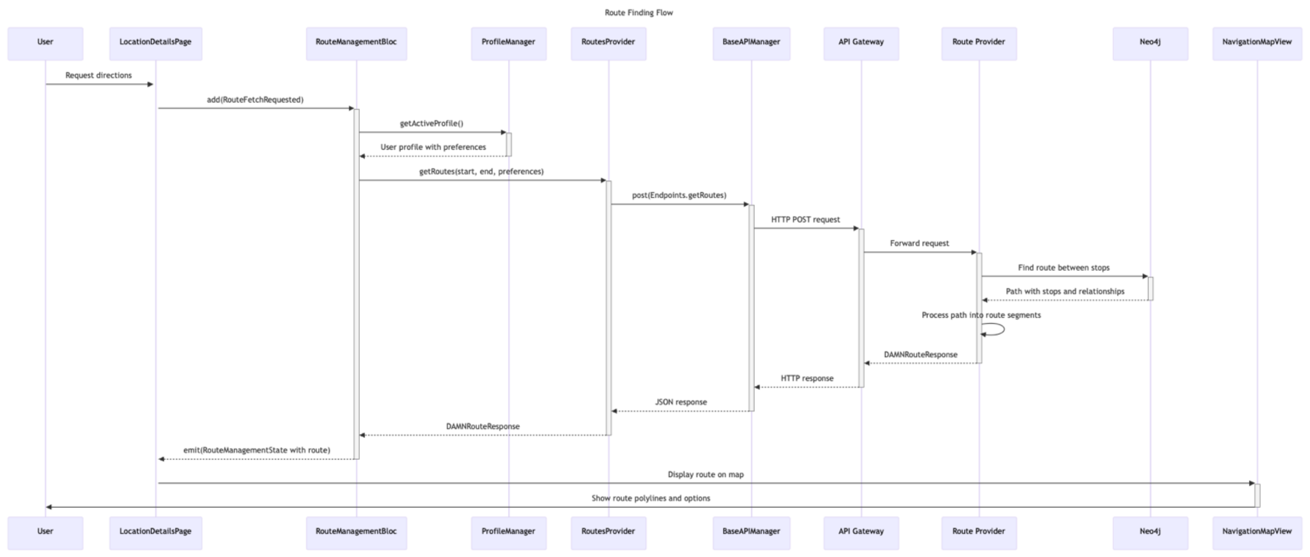
Route Request and Creation Flow – End-to-End Overview

The Route Formation Flow represents the complete journey of a user-initiated request to discover an optimal, sustainable path. It weaves together frontend interactions, backend microservice orchestration, and internal system state transitions to deliver personalized and environmentally conscious route recommendations.

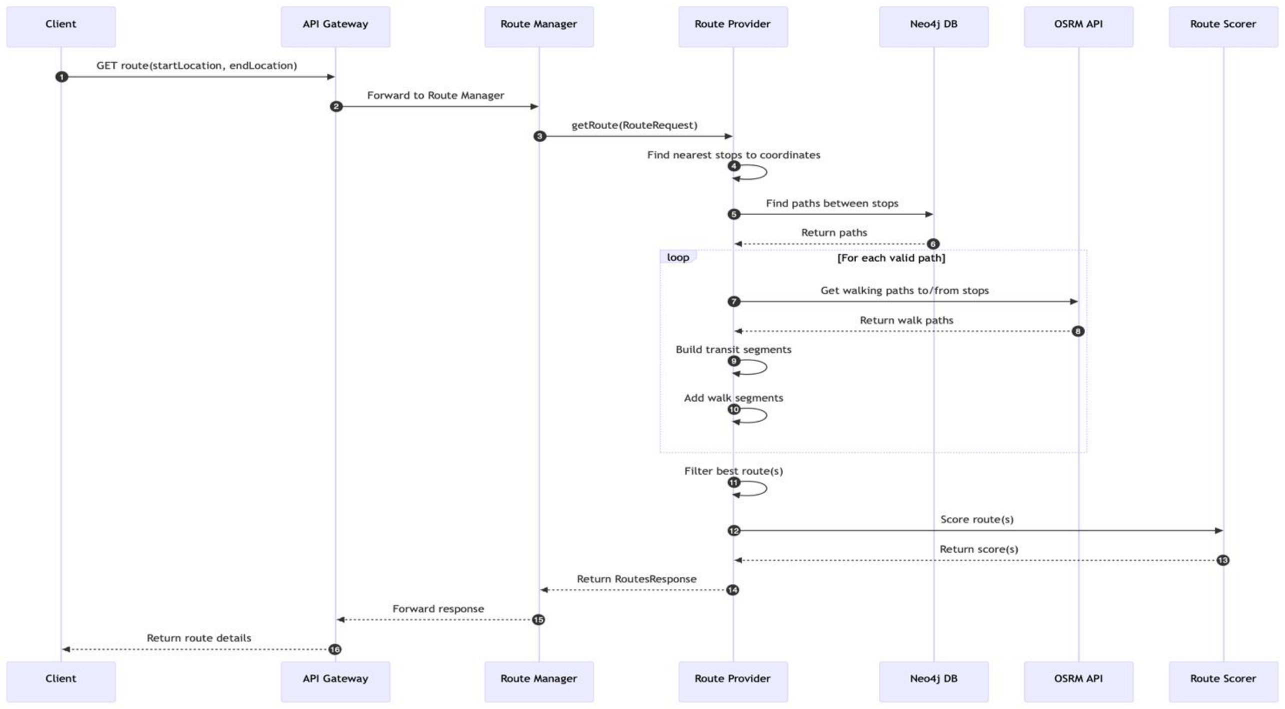


The corresponding sequence diagram for the front end and backend is as follows. The process begins when a user, from the *LocationDetailsPage*, initiates a request for directions. This action triggers the *RouteManagementBloc* through an event

(*RouteFetchRequested*). The bloc queries the *ProfileManager* to retrieve the currently active user profile, which includes the user's saved navigation preferences—like preferred transport modes, distance units, and accessibility settings.

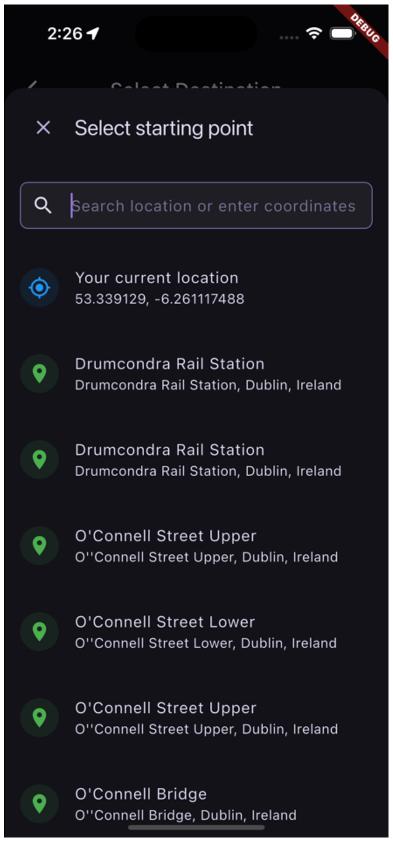


These preferences, alongside the start and end coordinates, are passed to the *RoutesProvider*, which constructs a request payload and sends it through the *BaseAPIManager* and API Gateway to the backend's Route Provider service. This phase ensures that the route computation process begins with contextually rich, user-centered information.



Once the backend receives the route request, the system transitions from the *Idle* state to *ProcessingRouteRequest*, as illustrated in the state diagram. The Route Provider begins by identifying the nearest transit stops to the user's coordinates (*FindingNearestStops*). It then queries the Neo4j graph database to determine potential paths between these stops (*SearchingRouteBetweenStops*). If a valid path is found, the system proceeds to build transit and walking segments (*BuildingTransitSegments*, *CreatingWalkSegments*) and then filters for the most efficient or sustainable route (*FilteringBestRoute*). If no valid path exists, the system retries with alternate stop pairs or transitions to a NoRouteFound state. Before returning, the selected route is scored by the *Route Scorer* service (*RouteScoring*) based on multiple sustainability and efficiency metrics. This completes the backend flow with a finalized, enriched *DAMNRouteResponse*.

The scored route response is then passed back through the API Gateway and Route Manager, where it's returned to the frontend's *RouteManagementBloc*. The bloc emits a new state containing the route data, which is consumed by the UI to render route polylines, markers, and user navigation paths on the map via the *NavigationMapView*. At this point, the user can view the recommended options, select their preferred path, and begin navigation—all rooted in a flow that dynamically blends real-time data, user context, and system intelligence.



Your current location
53.339129, -6.261117488

Drumcondra Rail Station
Drumcondra Rail Station, Dublin, Ireland

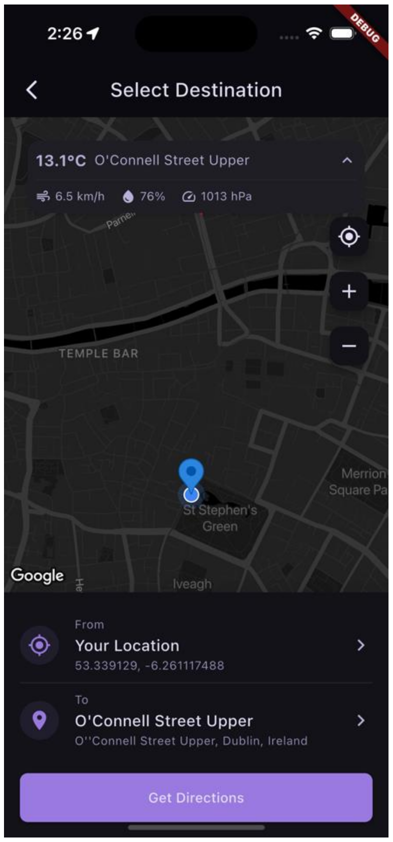
Drumcondra Rail Station
Drumcondra Rail Station, Dublin, Ireland

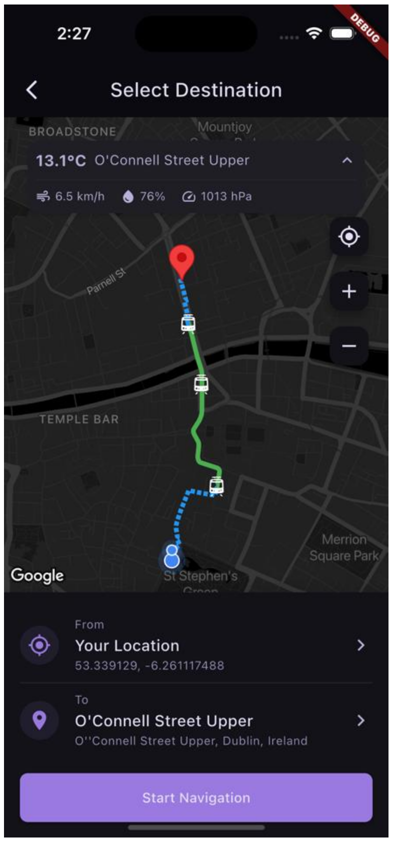
O'Connell Street Upper
O''Connell Street Upper, Dublin, Ireland

O'Connell Street Lower
O''Connell Street Lower, Dublin, Ireland

O'Connell Street Upper
O''Connell Street Upper, Dublin, Ireland

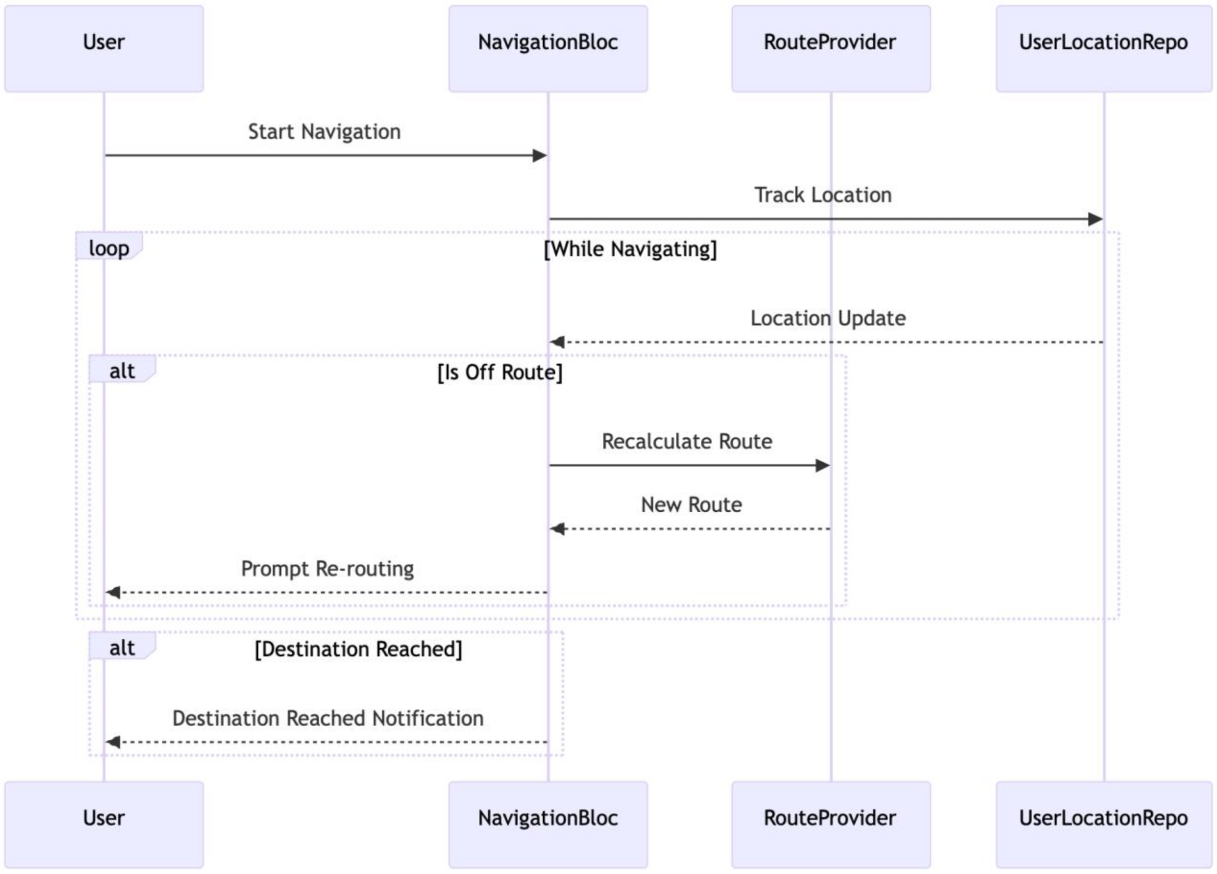
O'Connell Bridge
O''Connell Bridge, Dublin, Ireland





Navigation Flow – Real-Time Guidance, Deviation Detection and Rerouting

After a user selects a route, the system transitions into active navigation. Tapping **"Start Navigation"** from the *LocationDetailsPage* triggers the *NavigationPage*, which initializes the route in the *ActiveRouteManagementBloc (Navigation Bloc)*. This bloc sets up the *ActiveRouteNavigationView*, which configures the map using *GoogleMapController*.



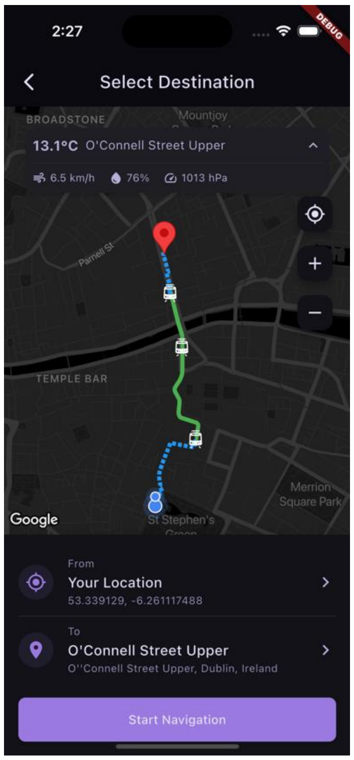
As updates come in:

- The bloc emits navigation states (*UserNavigationInProgress*),
- The view animates the map camera to follow the user,
- Updates continue in a loop until the destination is reached.

Once the user arrives, the system emits a *DestinationReached* state, ends navigation, and displays a completion dialog. This flow ensures smooth, real-time route tracking—whether using actual GPS or simulated movement—with clear, adaptive feedback through the UI.

Also, *a deviation detection check* is performed on each update. If the user's current position is determined to be off-route, the *ActiveRouteManagementBloc* (*NavigationBloc*) triggers a re-routing request to the *RouteProvider*. A new optimal route is calculated and returned, prompting the UI to update and guide the user accordingly. If the destination is reached during this loop, a completion notification is sent, and the navigation session gracefully ends. This flow ensures that the user stays on track—or gets seamlessly redirected—without needing manual intervention, enhancing both navigation reliability and user trust.

Start User Navigation

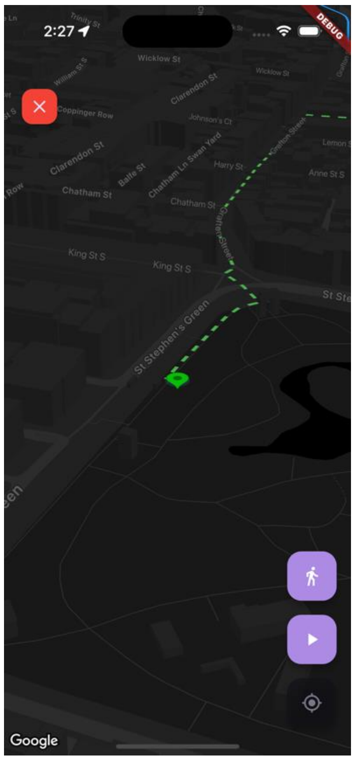


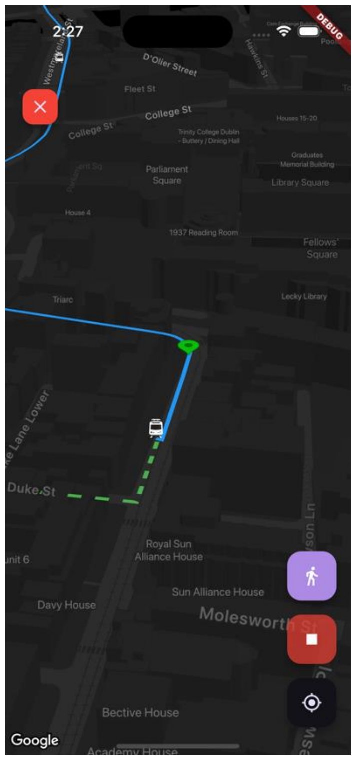
13.1°C O'Connell Street Upper

6.5 km/h 76% 1013 hPa

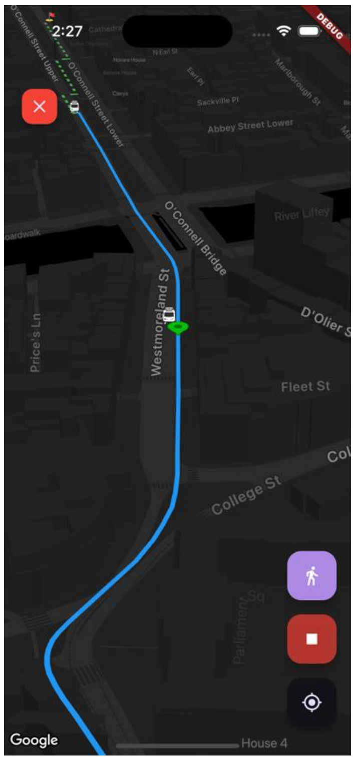
From
Your Location
53.339129, -6.261117488

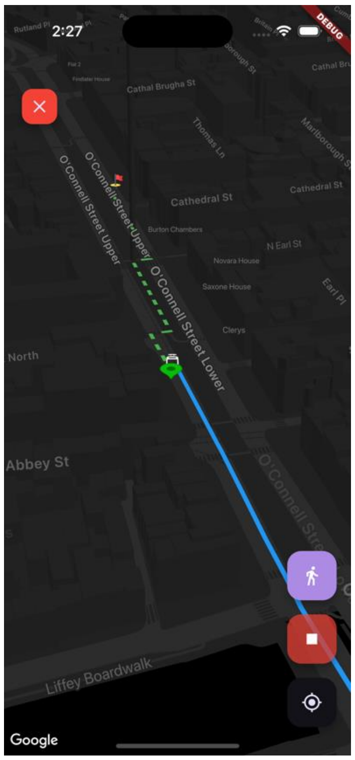
To
O'Connell Street Upper
O''Connell Street Upper, Dublin, Ireland

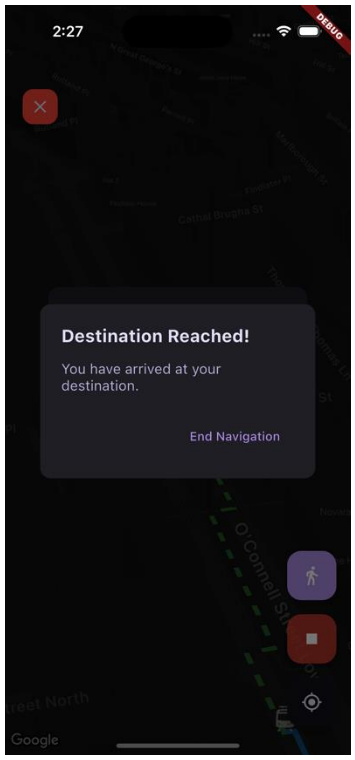




Destination Reached



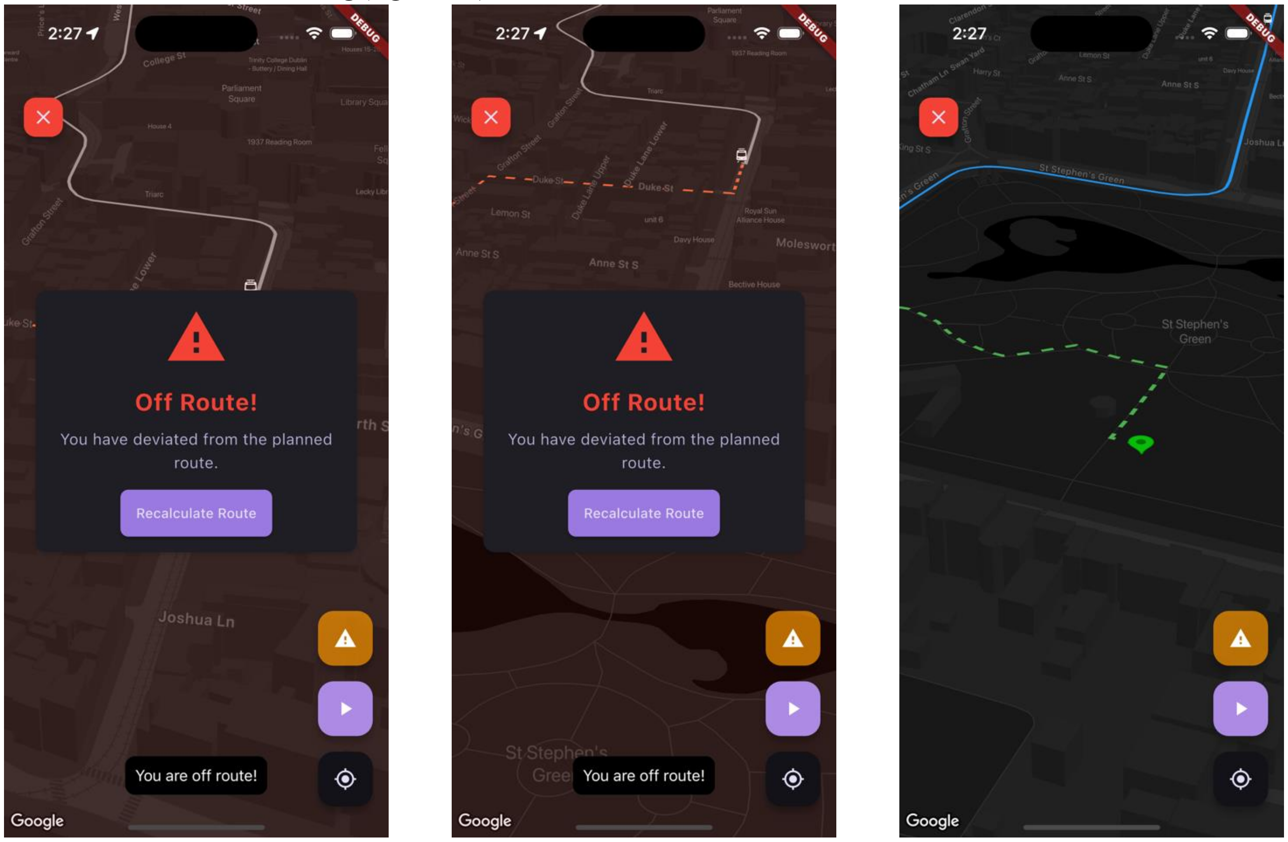




Destination Reached!

You have arrived at your destination.

Deviation Detected and re-routing (right most)


Off Route!

You have deviated from the planned route.

You are off route!

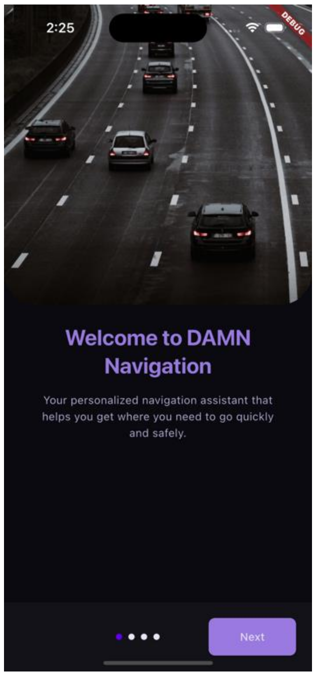
Off Route!

You have deviated from the planned route.

You are off route!

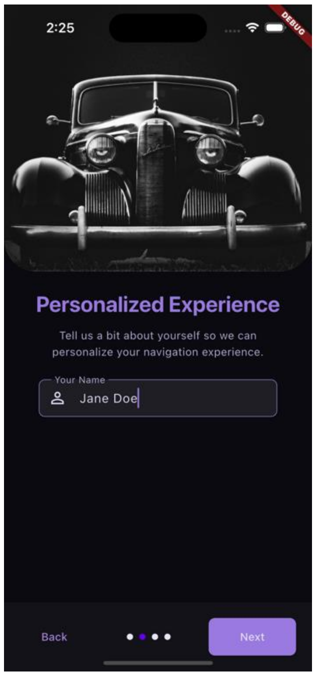
User Onboarding Flow

The onboarding flow captures the initial user journey from the moment the app is launched for the first time until a profile is created and activated. Upon launch, the app dispatches an *AppStarted* event to the *AppBloc*, which queries the *ProfileManager* to check for an active profile. If none exists, the system emits an AppState indicating the need for onboarding. The user is then guided through the onboarding process via the *OnboardingPage*, where they complete their personal information and configure preferences (e.g., transport mode, accessibility, distance units).



Welcome to DAMN Navigation

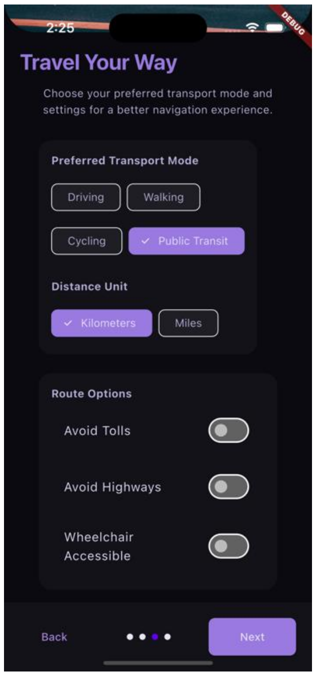
Your personalized navigation assistant that helps you get where you need to go quickly and safely.



Personalized Experience

Tell us a bit about yourself so we can personalize your navigation experience.

Your Name: Jane Doe



Travel Your Way

Choose your preferred transport mode and settings for a better navigation experience.

Preferred Transport Mode: Driving, Walking, Cycling, [✓ Public Transit]

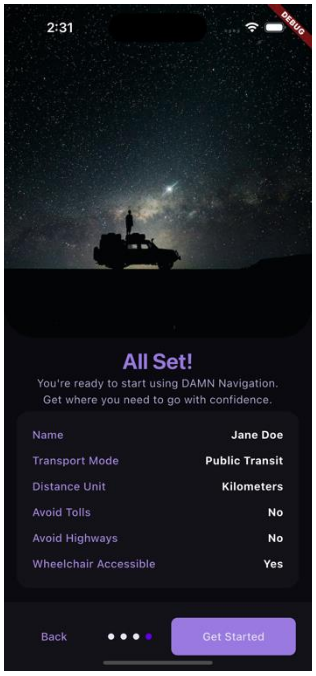
Distance Unit: [✓ Kilometers], Miles

Route Options

Avoid Tolls

Avoid Highways

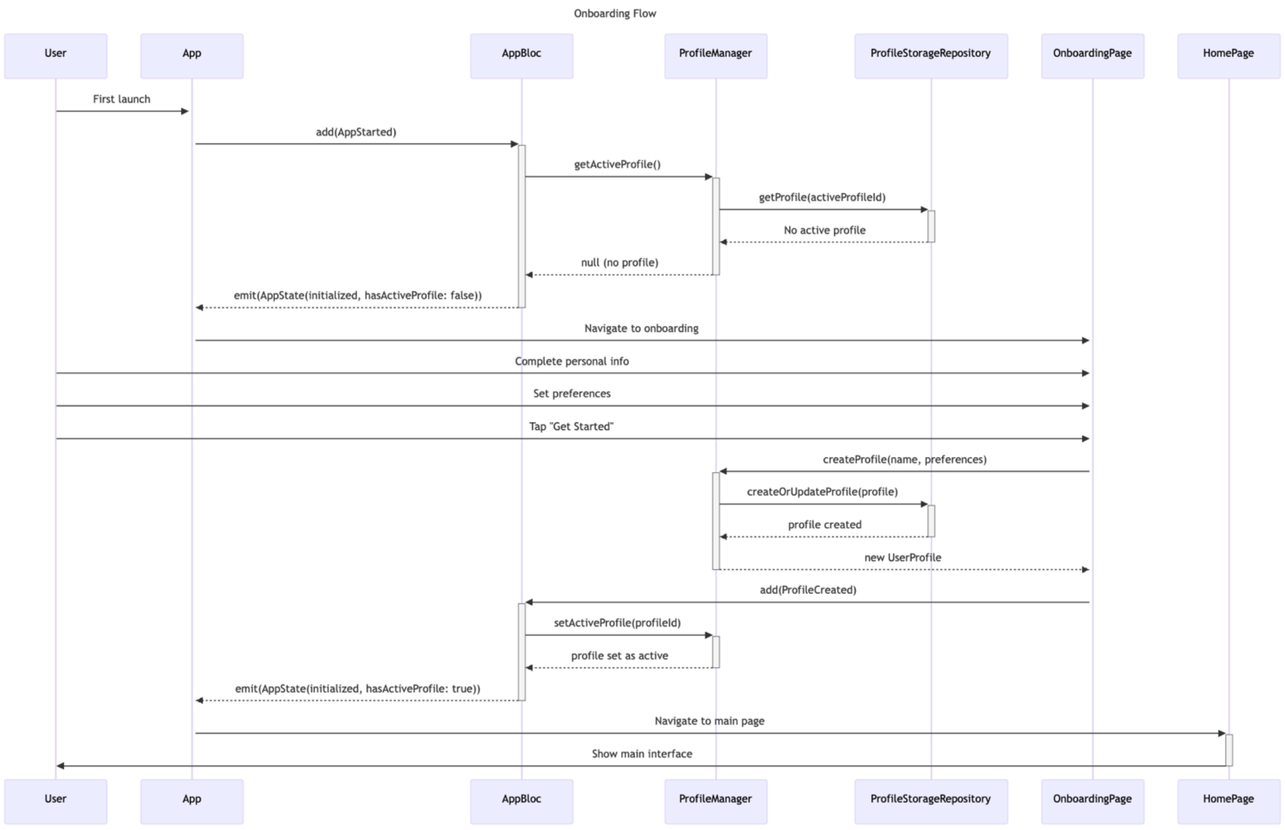
Wheelchair Accessible



All Set!

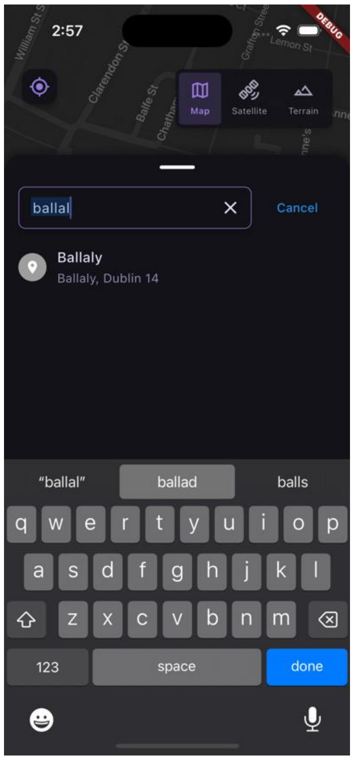
You're ready to start using DAMN Navigation. Get where you need to go with confidence.

Name	Jane Doe
Transport Mode	Public Transit
Distance Unit	Kilometers
Avoid Tolls	No
Avoid Highways	No
Wheelchair Accessible	Yes



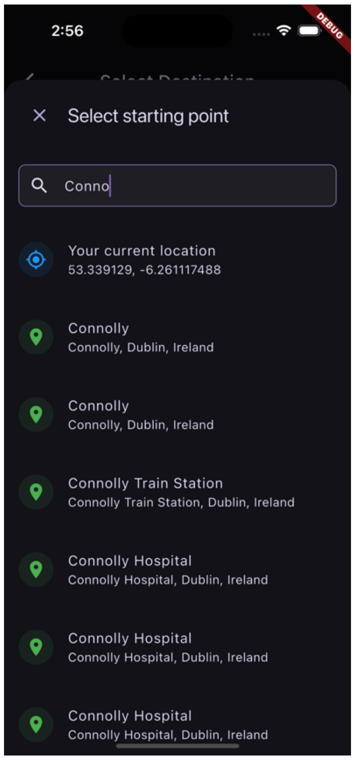
When the user taps **"**Get Started**"**, a profile is created via the *ProfileManager* and persisted by the *ProfileStorageRepository*. Once the new profile is created, it is set as active and the *AppBloc* updates the state (hasActiveProfile: true). The user is then navigated to the *HomePage*, where the main interface is presented—signaling the successful completion of onboarding. This flow ensures a personalized experience from the very first interaction, seamlessly linking profile preferences with route planning and navigation logic down the line.

Location Selection Flow



ballal

Ballaly
Ballaly, Dublin 14



Select starting point

Conno

Your current location
53.339129, -6.261117488

Connolly
Connolly, Dublin, Ireland

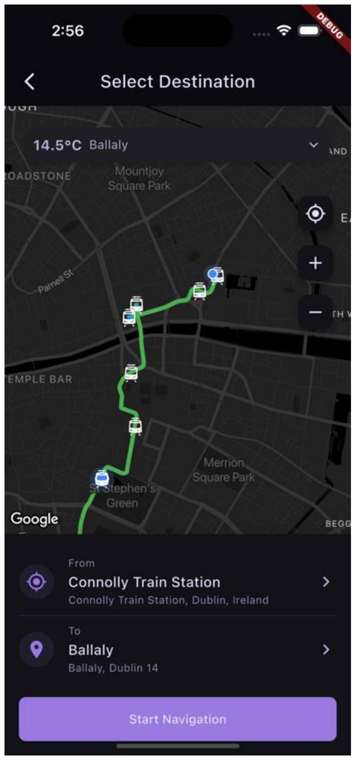
Connolly
Connolly, Dublin, Ireland

Connolly Train Station
Connolly Train Station, Dublin, Ireland

Connolly Hospital
Connolly Hospital, Dublin, Ireland

Connolly Hospital
Connolly Hospital, Dublin, Ireland

Connolly Hospital
Connolly Hospital, Dublin, Ireland



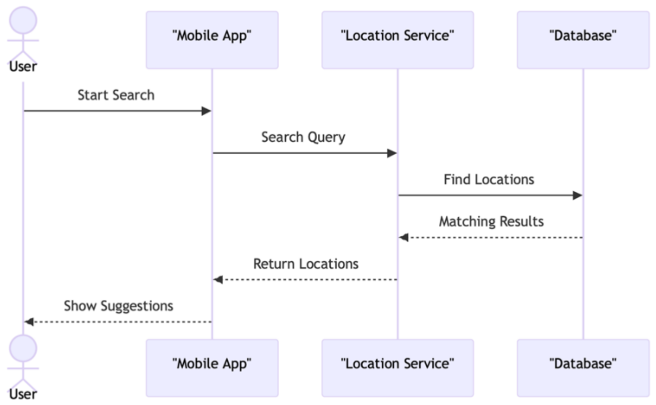
Select Destination

14.5°C Ballaly

From
Connolly Train Station
Connolly Train Station, Dublin, Ireland

To
Ballaly
Ballaly, Dublin 14

Start Navigation



The Location Search Flow outlines how a user interacts with the system to search for a destination or place of interest. When a user begins a search on the mobile app, the input query is forwarded to the Location Service, which in turn queries the Database for matching entries. The database responds with a list of relevant locations, which is passed back through the service to the app. The app then presents these results as suggestions to the user, allowing for seamless selection and improved usability during route planning. This interaction ensures a fast and intuitive search experience, supported by responsive backend querying.