LiftDrop

Mobile App for Deliveries

Gonçalo Morais, n.º 49502, e-mail: a49502@alunos.isel.pt, tel.: 927468061
João Ramos, n.º 49424, e-mail: a49424@alunos.isel.pt, tel.: 919222551

Advisor: Miguel Gamboa, e-mail: miguel.gamboa@isel.pt
Co-Advisor: Diogo Silva, e-mail: diogo.silva@lyzer.tech

June 2025

LiftDrop

LiftDrop Application

49502 Gonçalo Morais

49424 João Ramos

Advisor: Professor Miguel Gamboa
Co-Advisor: Engineer Diogo Silva

June 2025

Instituto Superior de Engenharia de Lisboa

Abstract

This project aims to develop an Android application for managing food deliveries. The focus is on the courier experience, so the client delivery requests are mocked using predefined API calls, and only a mobile interface for the courier was created.

The project is divided into two main modules: the Web API and the client application. The Web API is responsible for handling all requests, whether they are made by the mobile app through the Courier API or by those simulating client behavior through the Client API.

The platform supports order assignment based on courier performance and proximity, seamless delivery cancellation, and smart order placement that defaults to the nearest restaurant branch if no location is specified.

The backend is built with Kotlin and Spring Boot, using PostgreSQL for relational data storage. The frontend is implemented in Android using Jetpack Compose. The Google Maps API was also used, both to display the courier's current position in the mobile app and to calculate proximity during the backend's courier assignment logic.

Acknowledgements

We would like to express our sincere gratitude to our project supervisor, Prof. Miguel Gamboa and Engineer Diogo Silva, for their invaluable guidance, feedback, and support throughout the duration of this project. Their insights and encouragement greatly contributed to the development and completion of this work.

Finally, we would like to thank our peers, friends, and family for their encouragement and support during this journey.

Contents

List of Figures

Listings

List of Acronyms and Abbreviations

Acronym	Description
API	Application Programming Interface
DAW	Desenvolvimento de Aplicações Web
DB	Database
HTTP	HyperText Transfer Protocol
HTTPS	HTTP Secure
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
PDM	Programação em Dispositivos Móveis
RBAC	Role-Based Access Control
REST	Representational State Transfer
SQL	Structured Query Language
SSE	Server-Sent Events
UI	User Interface

Table 1: List of Acronyms and Abbreviations

1 Introduction

In recent years, urban delivery services have become increasingly dependent on mobile-first platforms that enable real-time logistics. From food to retail, customers now expect near-instant coordination, while couriers require intuitive tools to manage pickups, drop-offs, and sudden changes.

This project presents **LiftDrop**, a mobile application that simulates a real-time courier service for food deliveries. It is designed with an emphasis on responsive UI behavior, real-time backend communication, and order assignment based on various factors.

1.1 Motivation

Our motivation stems from the rising demand for flexible, real-time logistics platforms that balance performance with usability. While several commercial apps have set the standard for courier-based delivery, we wanted to explore the underlying technical mechanics of such systems.

1.2 Organization

The remainder of this report is organized as follows:

- **Background and Project Scope**: Overview of urban delivery platforms and relevant technologies.
- **System Design**: A comprehensive view of the platform's design, including the overall system architecture, user interface structure, and client/courier journey flows.
- **Implementation Overview**: A technical breakdown of the delivery system's architecture along with the development tools used for building and testing it.
- **Backend**: Details about backend deployment, authentication and authorization mechanisms, communication protocols, order assignment logic, database access, and error-handling strategy.
- **Frontend**: Explanation of state management with Jetpack Compose, communication flow with WebSockets, UI behavior during delivery journey, and use of DataStore for persistent storage.
- **Testing and Validation**: Description of the testing strategies applied, including integration tests, unit tests, and simulated user flows.
- **Final Remarks**: Final thoughts on the system's development and suggestions for future improvements.

2 Background and Project Scope

2.1 Background

Modern delivery platforms such as Uber Eats, Glovo, and Bolt Food exemplify the use of geolocation tracking, background services, and persistent user sessions to maintain reliable, low-latency interactions between clients and couriers. These systems are built atop robust infrastructures that balance real-time performance with fault-tolerance, often requiring complex state machines and backend coordination logic.

This project builds on these foundations by focusing on three core areas: real-time courier assignment, state-driven user experience, and backend message handling. While not meant to compete with commercial systems in scale, it implements many of the same patterns to explore the architectural and interactional demands of such services. It achieves this through the use of real-time geolocation and distance-based filtering powered by the Google Maps API, bidirectional communication via WebSockets for live updates between the courier and backend, and a modular backend architecture that enforces role-based access for both clients and couriers.

2.2 Project Scope

This section defines the functional and technical boundaries of the LiftDrop platform. It outlines the key features available to clients and couriers, the strategies for real-time data communication and order assignment. It also clarifies what the system is designed to accomplish and what lies outside the current scope of the project.

2.3 Platform Features

2.3.1 Client Features

- Register with email, password, and address;
- Login with email and password;
- Add a drop-off point when placing an order;
- Place orders by selecting restaurants, items, and a drop-off point;

2.3.2 Courier Features

- Register as a courier with name, email, and password;
- Login as a courier with email and password;
- Accept or decline delivery requests;
- Set availability by entering or exiting ”listening” status;
- Cancel an accepted order;

- Confirm pickup with a code;
- Confirm delivery with a code upon completion;

2.4 Real-Time Communication Strategy

Real-time data communication is crucial for dynamic interactions and courier management. Three communication models were considered:

- **Polling (Periodic Requests):** Simple but inefficient, causing excessive network overhead.
- **Server-Sent Events (SSE):** Effective for unidirectional updates, but limited in interaction complexity.
- **WebSockets:** Full-duplex communication that minimizes latency, ideal for bidirectional updates like order assignments or courier status changes.

WebSockets were chosen given the need for a communication method that supports bidirectional data exchange between the couriers and the system.

2.5 Order Assignment Strategy

Order assignments consider the following criteria:

- **Proximity:** Couriers closest to the pickup location are prioritized.
- **Travel Time:** The estimated time it takes for the courier to reach the pickup location is also considered. This estimate is obtained using the Google Maps API.
- **Courier Rating:** Courier Ratings are taken in consideration when choosing between two couriers who are equally close to the pickup spot.

2.6 User Roles

The platform supports two user roles:

- **Clients:** Individuals who place orders for delivery. They can track their orders by checking out the order's current ETA and status
- **Couriers:** Individuals who fulfill delivery orders. They can accept or reject delivery assignments, set availability, cancel ongoing deliveries and confirm pickups and deliveries.

2.7 Scope Limitations

To keep the scope manageable, the following features were intentionally excluded:

- **No full-featured client app:** The project simulates client interactions through API calls but does not implement a native client-side mobile interface.
- **No payment or invoicing systems:** Handling of payments is not included in the scope, but estimated earnings are still calculated and displayed for each order.
- **No administrative panel or analytics dashboards:** The system focuses on the courier workflow and backend coordination logic only.
- **Limited user types:** Only two roles are modeled, with no support for other user types such as admin or others.

These decisions were guided by time constraints.

3 System Design

3.1 User Journeys

3.1.1 Client Journey

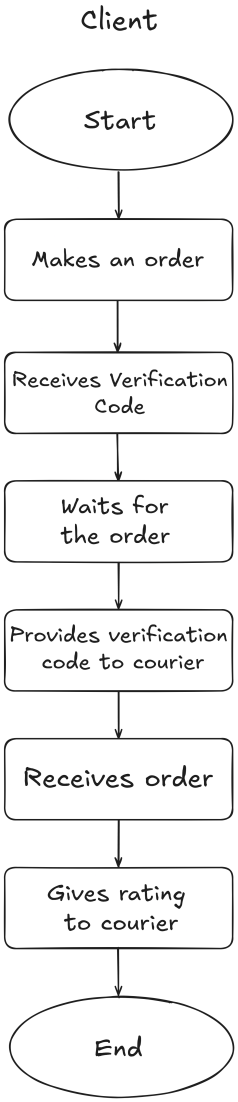


Figure 1: Client journey flowchart showing from the beginning

This flowchart (1) depicts the client's journey during the delivery phase of the LiftDrop platform. While the project scope does not include order management (as LiftDrop operates under the assumption that orders originate from external e-commerce systems like Shopify in a real-world scenario), the process begins when a client's Order triggers a DeliveryRequest within LiftDrop. The client then awaits courier assignment, receives a verification code, and provides it to the courier upon delivery. The journey concludes with the client rating the service.

3.1.2 Courier Journey

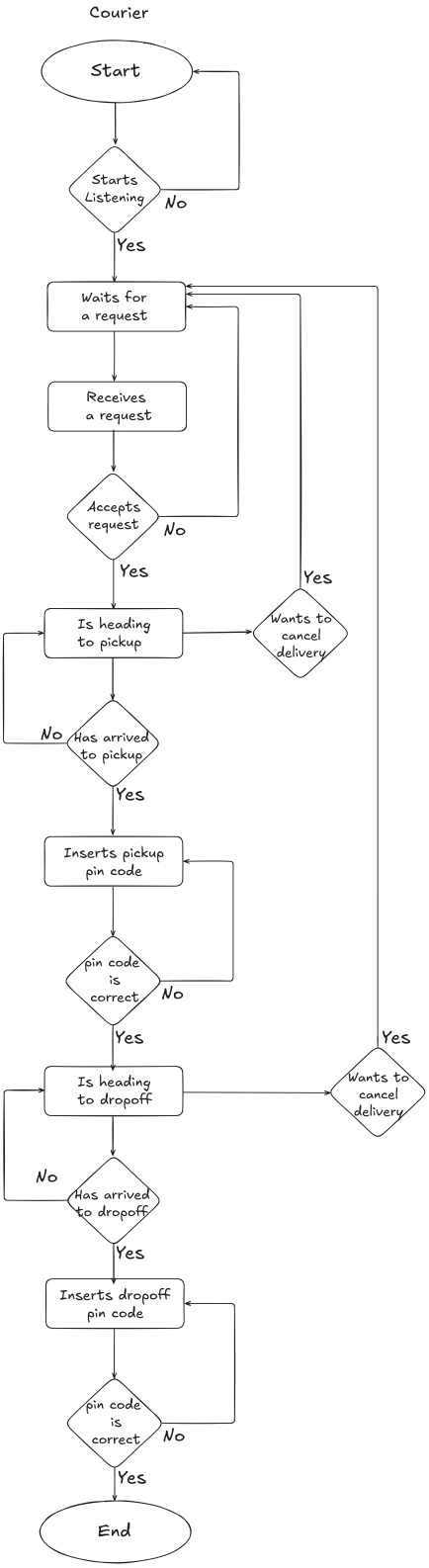


Figure 2: Courier journey flowchart showing from the beginning

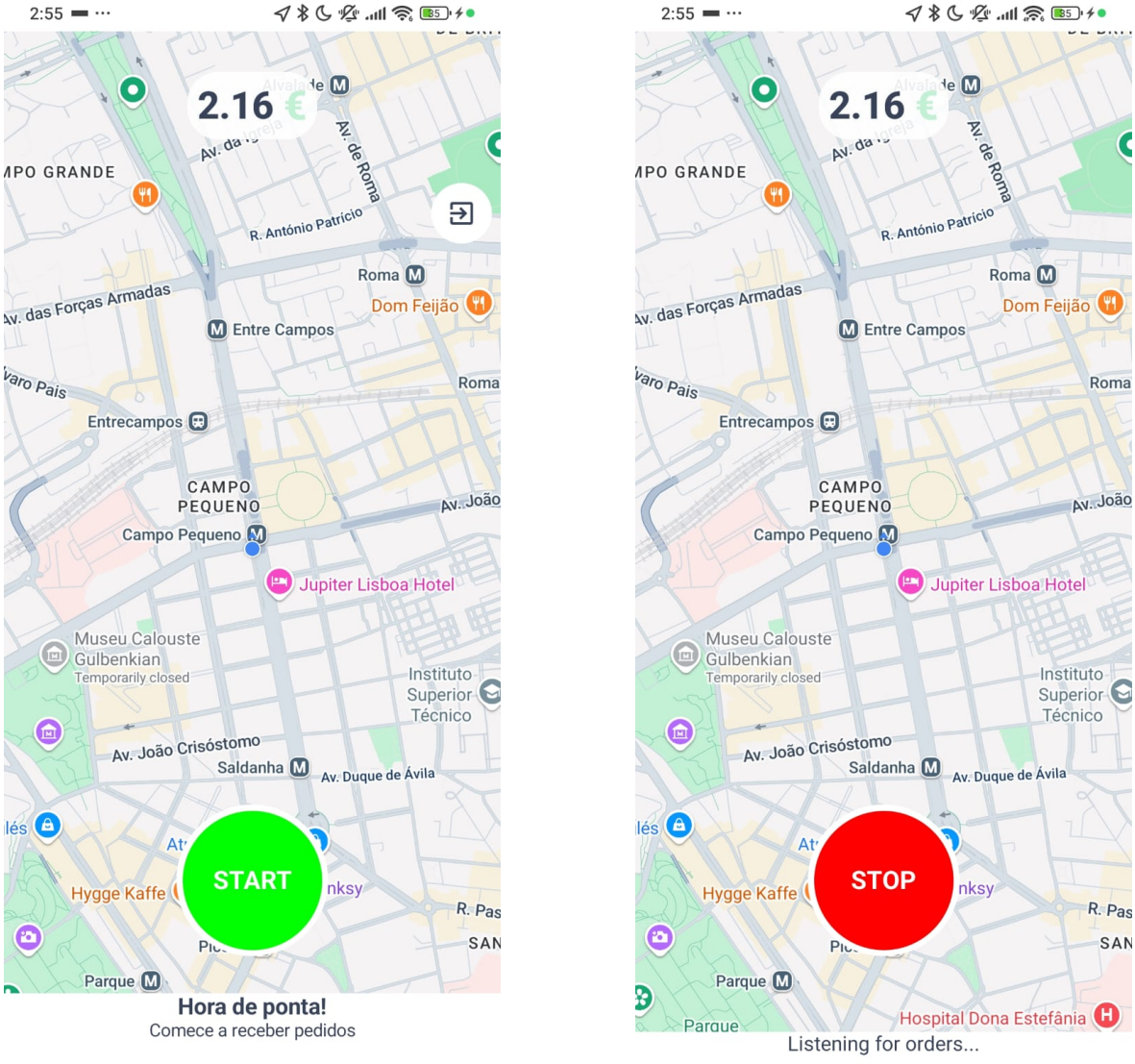
This flowchart (2) outlines the step-by-step decision-making process a courier follows while managing delivery requests on the LiftDrop platform. It begins with the courier awaiting an incoming order, which can either be accepted or rejected. Upon accepting a request, the courier proceeds to the pickup location, with the option to cancel if necessary. After pickup, the courier advances to the delivery phase. If the courier successfully reaches the destination, the delivery is confirmed. If the destination is not reached, the system prompts reevaluation or cancellation.

3.2 Interface Design

The LiftDrop application includes various UI flows that guide couriers through each stage of the delivery lifecycle. This section presents key interface screens, organized by functionality and user context.

3.2.1 Courier Flow

The following screens illustrate the courier experience, from going online to completing a delivery.

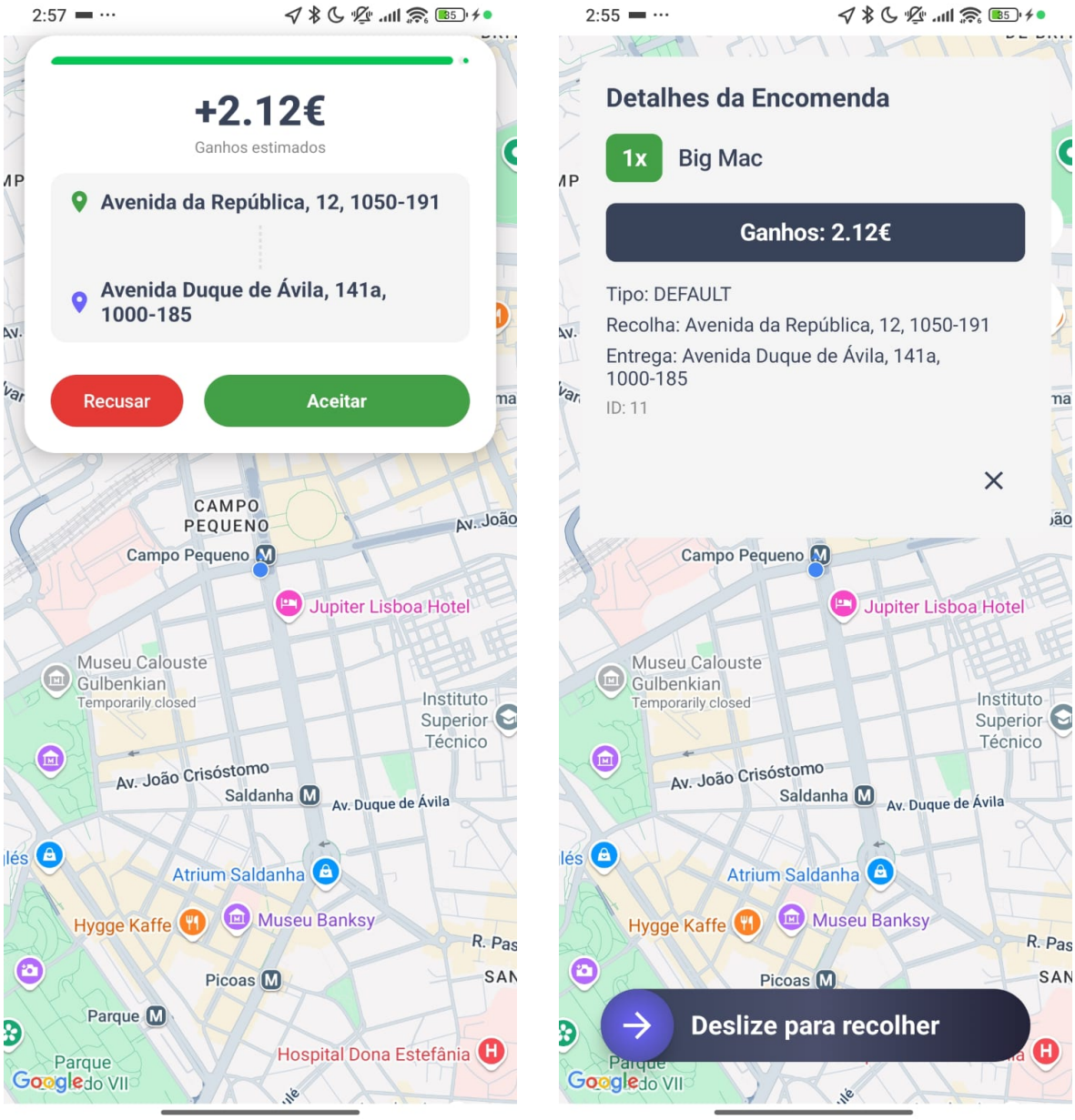


(a) Idle status screen

(b) Waiting for order screen

Figure 3: Courier status screens during online session

Idle Screen (3a): Allows couriers to toggle availability and begin accepting orders.
Waiting Screen (3b): Indicates that the system is searching for nearby orders in real time.



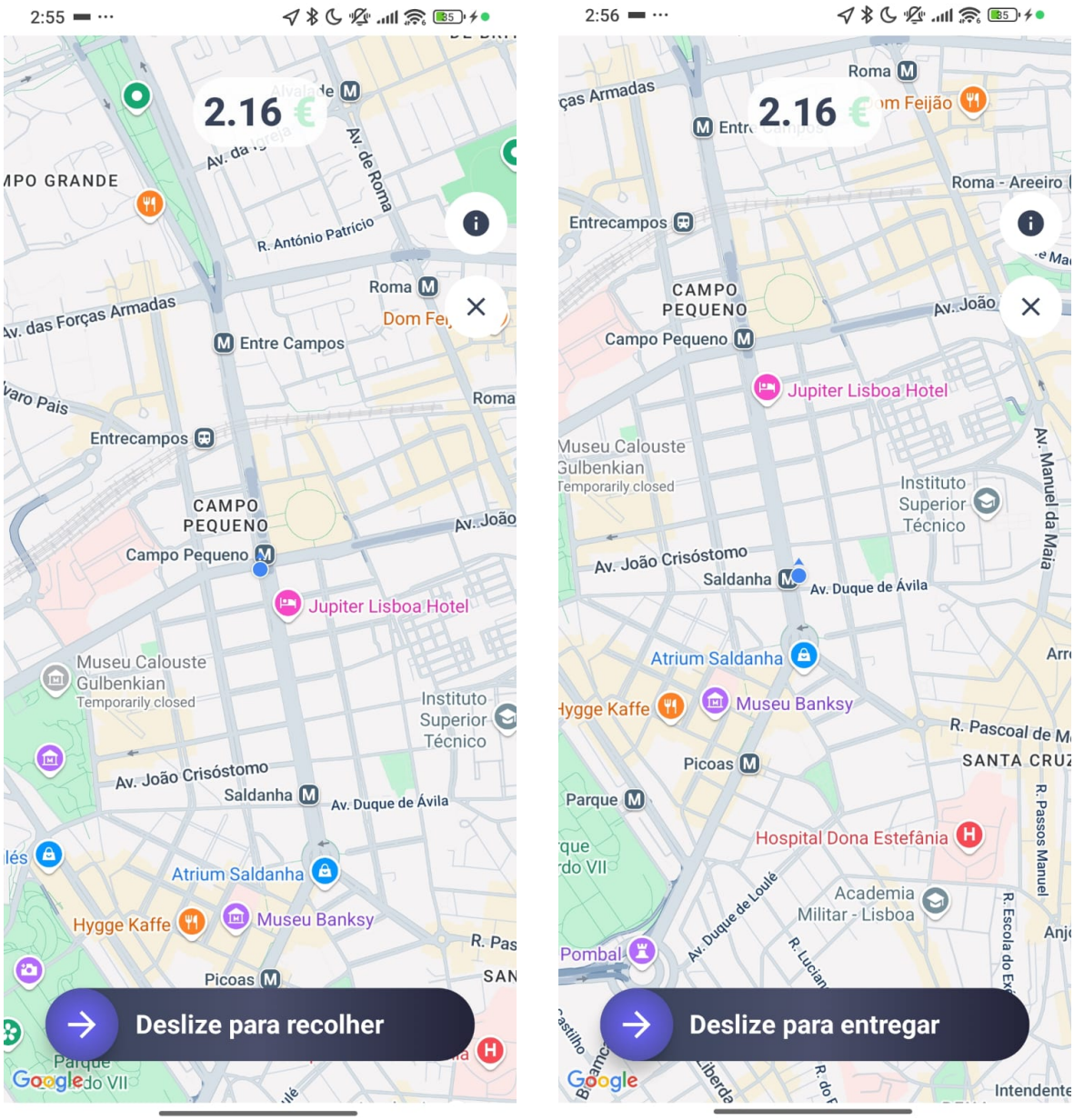
(a) Incoming delivery request

(b) Delivery confirmation screen

Figure 4: Screens for incoming request and order info

Delivery Request (4a): Displays order metadata such as pickup and drop-off addresses, delivery distance, and quick actions to accept or decline.

Order Info (4b): Presents a detailed breakdown of the assigned delivery, including the item designation, quantity, estimated earnings, delivery type, and both pickup and drop-off information.



(a) Pickup confirmation screen

(b) Delivery info screen

Figure 5: Screens for confirming pickup and delivery

Pickup Screen (5a): Guides the courier to confirm pickup, with cancelation support and verification of PIN.

Delivery Screen (5b): Guides the courier to confirm delivery, with cancelation support and verification of PIN.

3.3 System Architecture and Data Model

This section presents multiple architectural views of the LiftDrop platform, highlighting how its components interact and how its data model is structured. Each diagram focuses on a different conceptual layer of the system, from user roles and sessions to request fulfillment and geolocation modeling.

3.3.1 High-Level System Architecture

Figure 6 shows the overall system architecture, illustrating the interaction between the LiftDrop Mobile Application running on the courier's Android device and the LiftDrop Backend deployed on a server.

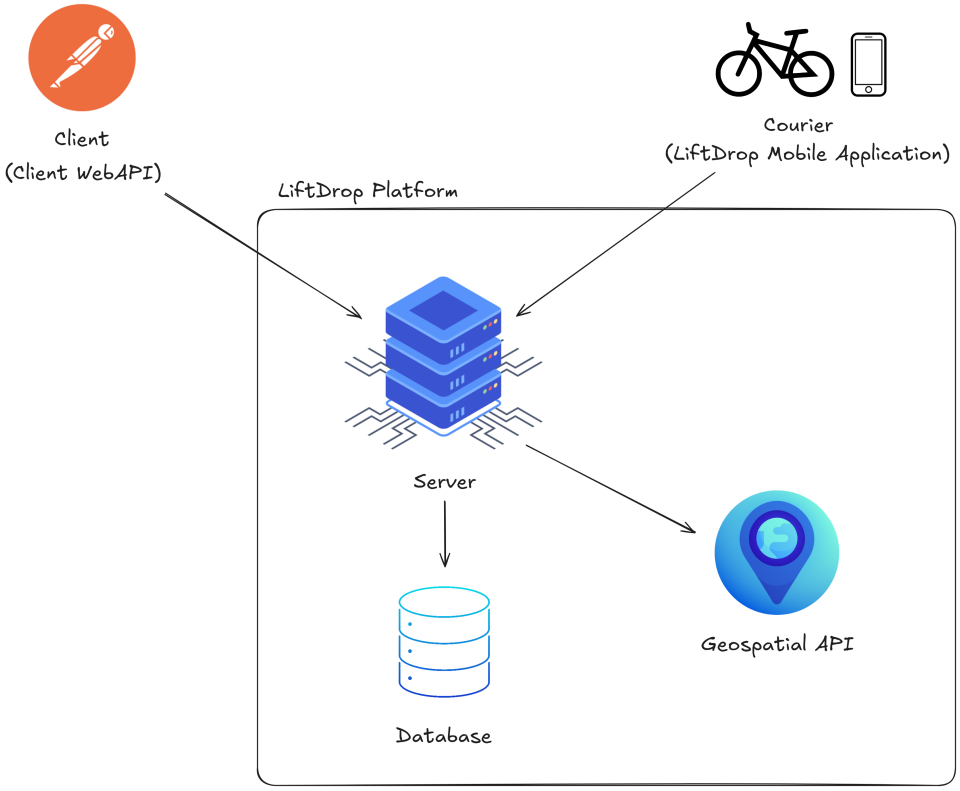


Figure 6: High-level overview of the LiftDrop system architecture

3.3.2 User and Role Hierarchy

This diagram illustrates how the abstract `User` entity is specialized into the `Client` and `Courier` roles. It also shows how sessions are managed on a per-user basis for secure authentication.

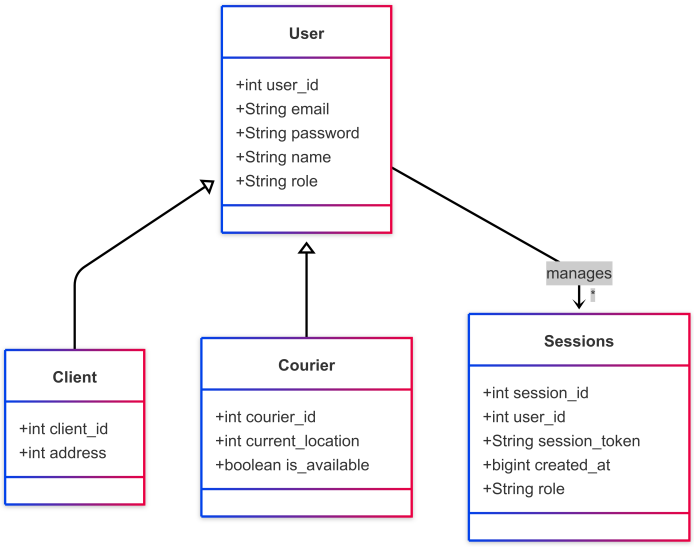


Figure 7: User roles and session relationships

3.3.3 Request and Delivery Lifecycle

This model depicts the lifecycle of a delivery—from creation of a request by a client, through courier assignment, to delivery completion and rating. It reflects how entities such as `Request`, `Delivery`, and `CourierRating` are interlinked throughout the delivery flow.

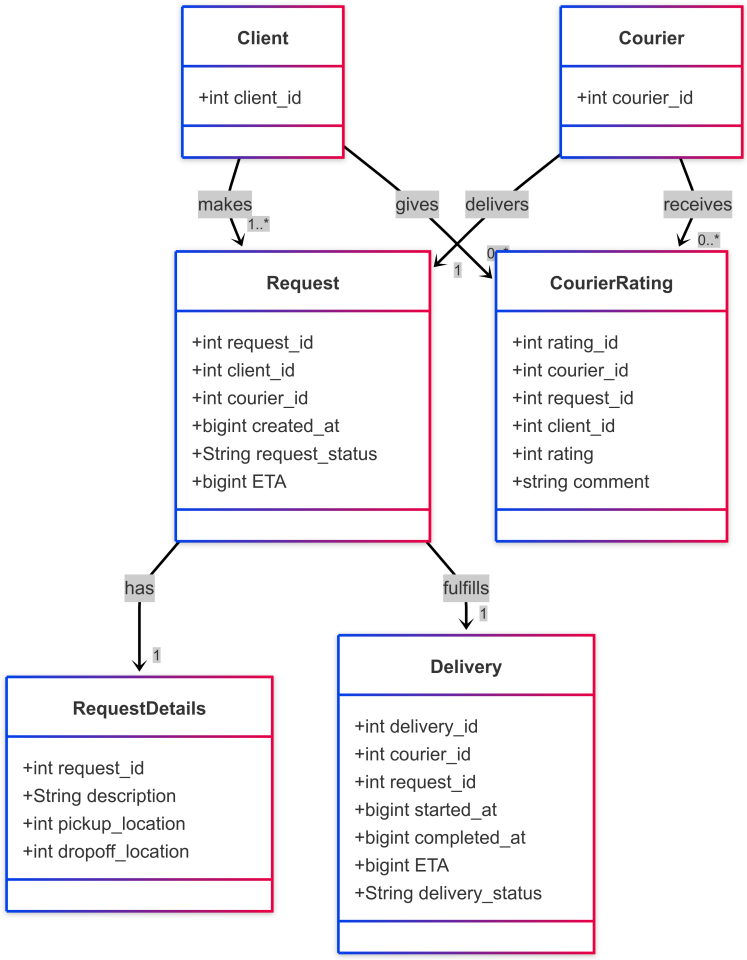


Figure 8: Request and Delivery lifecycle

3.3.4 Location and Address Modeling

This view focuses on how real-world geographic data is modeled within the platform. It shows the relationship between generalized `Location` entities and their specializations—`PickupSpot` and `DropOffSpot`—as well as how these relate to physical addresses.

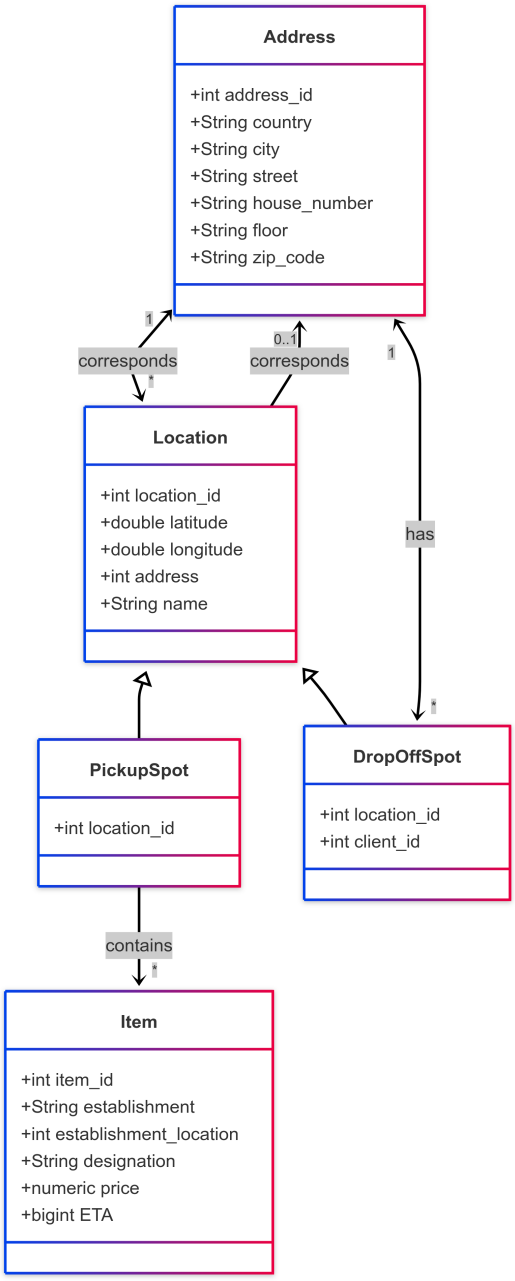


Figure 9: Location and Address structure

For a complete view of the system's data model, including all class relationships and fields, refer to the full class diagram provided in Appendix B.

4 Implementation Overview

4.1 Technology Stack

- **Mobile Application**: Developed using Android Jetpack Compose, the app enables couriers to manage their current delivery and respond to incoming delivery requests.
- **Backend Server**: Built with Spring Web MVC, the server handles business logic, order assignment, session management, and serves RESTful and WebSocket endpoints.
- **Database**: PostgreSQL is used for structured data storage, including user info, order metadata and courier location logs.
- **Geospatial Services**: The Google Maps API is used to estimate travel times and display the map interface.

4.2 Development Tools

- **Postman**: Used to both mock client actions and validate REST endpoints during backend development.
- **Git and GitHub**: Used for version control, issue tracking, and collaborative development.
- **IntelliJ IDEA**: Used for backend development and debugging.
- **Android Studio**: Used to build and test the Android application.
- **Ngrok**: Used to expose the local backend for mobile app testing.
- **Fake GPS**: Mobile application used to emulate current location in order to test location-based features.

4.3 Google Maps API Usage

The Google Maps API is used both in the backend and frontend:

- **Frontend:** Used to display the built-in map interface by using Google's *Maps SDK for Android*.
- **Backend:** Used for proximity calculations and travel time estimation via the Distance Matrix API. The Google Maps API is also used to calculate the geographic coordinates from the given address when the client registers, via its Geocoding API.

5 Backend

5.1 Deployment Details

Backend Deployment

The backend is hosted on Render and it was chosen for its adequate free plan and our prior familiarity with the platform.

Database Hosting

PostgreSQL is also managed by Render, where the data is persisted in a cloud PostgreSQL instance, with automated backups enabled and access limited to the backend service.

Environment Configuration

Environment variables (e.g., API keys, DB connection URLs) are configured via the Render dashboard and injected during deployment.

Continuous Deployment

Each push to the GitHub repository triggers an automatic build and deployment on Render, ensuring that the most recent changes are reflected without manual intervention.

5.2 Authentication and Authorization

Authentication

To secure user sessions and API access, LiftDrop uses session-based authentication:

- **HTTP-only Session Cookies**: Upon successful login, the backend generates a session cookie containing a signed token. The cookie is marked as HTTP-only, preventing JavaScript access and mitigating cross-site scripting (XSS) risks.

Cookie Generation and Deletion

The following Kotlin snippets show how the session cookie is generated and invalidated.

```
ResponseCookie
    .from("auth_token", token)
    .path("/")
    .maxAge(Duration.ofDays(1))
    .httpOnly(true)
    .build()
```

Listing 1: Generating the Session Cookie

```
ResponseCookie
    .from("auth_token", "")     // Clear the value
    .path("/")                  // Match the original path
    .maxAge(0)                  // Immediate expiry
    .httpOnly(true)
    .build()
```

Listing 2: Deleting the Session Cookie

Authorization

LiftDrop uses role-based access control (RBAC) to enforce feature access policies. Each API endpoint inspects the session token to determine whether the requester is a `Client` or a `Courier`. This is handled at the controller layer using a Spring `HandlerInterceptor`:

```
@Component
class AuthenticationInterceptor(...) : HandlerInterceptor {
    override fun preHandle(request: HttpServletRequest, response:
        HttpServletResponse, handler: Any): Boolean {
        if (handler !is HandlerMethod) return true
        val authCookie = ... // extract auth cookie
        // Client auth (the same is done for the courier but with the
            AuthenticatedCourier class)
    if(handler.hasParameterType(AuthenticatedClient::class.java)){
            val client = authorizationHeaderProcessor.
                processClientAuthorizationHeaderValue(authCookie?.value)
            return client?.let {
    AuthenticatedClientArgumentResolver.addClientTo(it, request)
                true
            } ?: unauthorized(response) // returns false with a 401 status
                code
        } return true } }
```

Listing 3: Condensed Kotlin-style pseudocode for AuthenticationInterceptor

This approach was made based on what was learned in the DAW course as it ensures that access is only granted to authenticated and properly authorized users based on their declared role. Unauthorized requests result in an HTTP `401 Unauthorized` response.

5.3 Address Geocoding

Whenever a new client registers in the application, it is necessary to associate their physical address with geographic coordinates (latitude and longitude) to support delivery-related operations.

This API allows the backend to convert a textual address (e.g., `Rua do Instituto Superior de Engenharia de Lisboa`) into precise coordinates that can be stored in the database and used in geospatial computations, such as courier assignment.

Usage Flow:

1. The client fills in their address during registration.
2. The backend sends the address to the Google Geocoding API.
3. The API returns the corresponding coordinates.
4. These coordinates are associated with the client and stored in the database as part of the `Address` entity.

The Kotlin snippet below illustrates the geocoding logic on the backend (simplified):

```kotlin
fun getLatLngFromAddress(address: String): Pair<Double, Double>? {
    val client = OkHttpClient()
    val url = baseUrl + "?address=${address.replace(" ", "+")}&key=$apiKey"
    val request = Request.Builder().url(url).build()

    client.newCall(request).execute().use { response ->
        if (!response.isSuccessful) throw Exception("Unexpected code
            $response")

        val body = response.body?.string() ?: throw Exception("Empty response
             from geocoding API")
        val json = JsonParser.parseString(body).asJsonObject

        if (json["status"].asString == "ZERO_RESULTS") return null

        if (json["status"].asString == "OK") {
            val location = json["results"].asJsonArray[0]
                .asJsonObject["geometry"].asJsonObject["location"].
                    asJsonObject
            val lat = location["lat"].asDouble
            val lng = location["lng"].asDouble
            return Pair(lat, lng)
        } else {
            throw Exception("Error from geocoding API: ${json["status"].
                asString}")
        }
    }
}
```

Listing 4: Using Google Geocoding API during client registration

5.4 Communication Protocol

LiftDrop uses Spring's WebSocket support to manage courier communication. The WebSocket endpoint at `/ws/courier` is configured through the `CourierWebSocketConfig` class, which implements `WebSocketConfigurer` and registers the `CourierWebSocketHandler`, a custom handler that extends Spring's `TextWebSocketHandler`. This can be seen in the following snippets:

```
    @Configuration
    @EnableWebSocket
    class CourierWebSocketConfig(
        private val courierWebSocketHandler: CourierWebSocketHandler,
    ) : WebSocketConfigurer {
        override fun registerWebSocketHandlers(registry:
            WebSocketHandlerRegistry) {
            registry
            .addHandler(courierWebSocketHandler, "/ws/courier")
                .setAllowedOrigins("*")
        } }
```

Listing 5: WebSocket configuration

```
class CourierWebSocketHandler(...) : TextWebSocketHandler() {
    // Active sessions: courierId -> WebSocketSession
    private val sessions = ConcurrentHashMap<Int, WebSocketSession>()
    // Handles new connections (auth required)
    override fun afterConnectionEstablished(session: WebSocketSession) { ...
        }
    // Cleanup on disconnect
    override fun afterConnectionClosed(...) { ... }
    // Handles Couriers decisions regarding incoming requests
    override fun handleTextMessage(
        session: WebSocketSession,
        message: TextMessage,
    ) {
        val json = jacksonObjectMapper().readTree(message.payload)
        when (json.get("type").asText()) {
            "DECISION" -> handleCourierDecision(session, json)
            else -> println("Unknown message type")
        }
    }
    // Sends JSON message to specific courier
    fun <T : Any> sendMessageToCourier(courierId: Int, message: T) {
        sessions[courierId]?.takeIf { it.isOpen }?.let { session ->
            session.sendMessage(TextMessage(jacksonObjectMapper().
                writeValueAsString(message)))
        }
    }
}
```

Listing 6: CourierWebSocketHandler and its different methods

The `CourierWebSocketHandler` acts as the messenger between the system and couriers by managing active courier sessions, handling couriers' yes or no responses to delivery requests, and sends them updates about new delivery requests or changes regarding an ongoing order cancellation. Here's how each part works:

afterConnectionEstablished/Closed

These methods are invoked whenever the courier starts or stops listening to new requests, respectively.

handleTextMessage

This method is invoked whenever a courier accepts or declines an incoming delivery request.

sendMessageToCourier

This method is invoked in three different scenarios:

- **Initial assignment**: When dispatching a new delivery request during courier assignment (see Chapter 5.5).
- **Reassignment**: When an order needs a new courier after cancellation (see Chapter 5.6).
- **Cancellation updates**: When resolving a post-pickup cancellation (see Chapter 5.6).

This entire flow works because the backend actively maintains courier connections, routing messages only to their intended recipients. All delivery-related communication uses structured JSON messages.

The following function demonstrates how a courier's acceptance of a delivery request is sent to the backend via WebSocket:

```
override suspend fun acceptRequest(requestId: String, token: String): Boolean
    {
    val messageJson = """
        {
            "type": "DECISION",
            "requestId": "$requestId",
            "decision": "ACCEPT"
        }
    """.trimIndent()

    return webSocket?.send(messageJson) == true
}
```

Listing 7: Sending an acceptance message via WebSocket

On the server side, the `handleTextMessage` function will then receive this WebSocket message, parse the message payload and call the `handleCourierDecision` function with the parsed message and the current WebSocket session. This handler then processes the courier's response (either `ACCEPT` or `DECLINE`) and triggers the corresponding order state updates.

5.5 Order Assignment Logic

Order assignment is based on:

- **Proximity to Pickup**: Calculated using Google Maps Distance Matrix API to ensure routing reflects real travel time, not just straight-line distance.
- **Courier Rating**: Couriers with higher ratings receive slight prioritization when travel times are similar.

To rank couriers based on proximity to the pickup location, we apply a two-phase sorting approach.

5.5.1 First Phase: Distance Calculation

In the first phase, we sort five couriers by their direct (as-the-crow-flies) distance to the pickup spot using PostgreSQL's PostGIS extension. The following snippet illustrates how this is implemented:

```
override fun getClosestCouriersAvailable(
        pickupLat: Double, // Pickup spot's latititude
        pickupLng: Double, // Pickup spot's longitude
        requestId: Int, // Request id
        maxDistance: Double, // Max distance to be considered
    ): List<CourierWithLocation> =
        handle.createQuery("""
            SELECT ..., -- Other courier attributes
                ST_Distance(  -- Direct distance using PostGIS
                    ST_SetSRID(ST_MakePoint(l.longitude, l.latitude), 4326)::
                        geography,
                    ST_SetSRID(ST_MakePoint(:pickupLon, :pickupLat), 4326)::
                        geography
                ) AS distance_meters
            FROM ... -- Joins Courier table with Location table where courier
                 is available
            AND ... -- Checks if courier has already declined this specific
                request
            AND ST_Distance( -- Gets 5 closest couriers
                    ST_SetSRID(ST_MakePoint(l.longitude, l.latitude), 4326)::
                        geography,
                    ST_SetSRID(ST_MakePoint(:pickupLon, :pickupLat), 4326)::
                        geography
                ) < :maxDistance
            ORDER BY distance_meters
            LIMIT 5;
            """,
            ).bind(...).list()
```

Listing 8: Ranking couriers by direct distance using PostgreSQL PostGIS extension

5.5.2 Second Phase: Travel Time Calculation

Although proximity is important, actual travel times can vary significantly. After getting the five closest couriers, each one is sorted using the Google Maps Distance Matrix API, which calculates real driving times based on road routes and current traffic.

```
private suspend fun rankCouriersByTravelTime(
    couriers: List<CourierWithLocation>,
    destinationLat: Double,
    destinationLon: Double,
): List<CourierWithLocation> {
    val origins = couriers.joinToString("|") { "${it.latitude},${it.longitude}" }
    val destination = "$destinationLat,$destinationLon"


    // Sends a request to the Google Maps Distance Matrix API,
    // using courier locations as origins and the destination as the drop-off point.
    val response = ...

    // Parses the API response to extract a list of travel times (in seconds), where each value corresponds to a courier in the original list.
    val travelTimes = ...

    // Combines each courier with their corresponding travel time,
    // sort them in ascending order of travel time,
    // and return a list of couriers with the estimated travel time included.
    return couriers.zip(travelTimes)
        .sortedBy { it.second }
        .map { (courier, time) -> courier.copy(estimatedTravelTime = time) }
}
```

Listing 9: Ranking couriers by travel time using Google Maps API

After obtaining the estimated travel times, each courier is scored using a weighted formula that combines normalized travel time and courier rating. Couriers with lower scores are ranked higher and prioritized for order assignment.

```
fun rankCouriersByScore(couriers: List<CourierWithLocation>): List<CourierWithLocation> {
    val minTime = couriers.minOf { it.estimatedTravelTime ?: 0L }
    val maxTime = couriers.maxOf { it.estimatedTravelTime ?: 1L }
    val minRating = couriers.minOf { it.rating ?: 0.0 }
    val maxRating = couriers.maxOf { it.rating ?: 5.0 }

    val alpha = 0.7 // weight for travel time
    val beta = 0.3  // weight for rating

//Faster couriers and those with higher ratings get a lower score.
    return couriers.sortedBy { courier ->
        val normTime = if (maxTime > minTime) {
```

```
            ((courier.estimatedTravelTime ?: 0L) - minTime).toDouble() / (
                maxTime - minTime)
        } else 0.0

        val normRating = if (maxRating > minRating) {
            ((courier.rating ?: 0.0) - minRating) / (maxRating - minRating)
        } else 0.0

        alpha * normTime + beta * (1 - normRating)
    }
}
```

Listing 10: Scoring couriers by travel time and rating

Lastly, the centerpiece of LiftDrop's dynamic dispatching logic is the courier assignment handler, which orchestrates several previously discussed components into a single cohesive workflow. This function initiates the process of locating available couriers within an incrementally expanding search radius, sending them real-time assignment requests, and awaiting their responses. If a courier accepts, the assignment is confirmed; if not, the system proceeds to the next available candidate. The snippet below illustrates how these interconnected elements are brought together:

```
suspend fun handleCourierAssignment(
        // request id, pickup location, distance increments...
    ): Boolean {
    // Fetches ranked list of available couriers within current //search
        radius, ordered by proximity, travel time and rating
    val rankedCouriers = fetchRankedCouriersByTravelTime(...)
    // If couriers are available for assignment
    if (rankedCouriers is Either.Right) {
        // Attempt assignment with each courier in order of proximity
        for (courier in rankedCouriers.value) {
            // Registers this request with the assignment coordinator
            // to track courier responses
            val response = ...
            // Retrieves full request details from database
            val requestDetails = getRequestForCourierById(requestId)
            // Calculates estimated earnings for this delivery
            val estimatedEarnings = ...
            // Sends assignment offer to courier via WebSocket
            courierWebSocketHandler.sendMessageToCourier( courier.courierId,
                requestDetails)
            // Waits for courier response with 20 second timeout
            try { withTimeout(20_000) { response.await() }}
            catch (e: TimeoutCancellationException) { false }
            } }
    // If no couriers accepted, retry with expanded search radius
    return handleCourierAssignment(...)
    }
    //If no couriers are found, wait 10 seconds and then retry
    // with expanded search radius
```

```
28      delay(10_000)
29      return handleCourierAssignment(...)
30  }
```

Listing 11: Handling Courier Assignments (Simplified Pseudocode)

To facilitate asynchronous communication between the backend and couriers during the assignment process, a dedicated coordination component named `AssignmentCoordinator` was implemented. This object maintains an in-memory registry that maps each delivery request ID to a corresponding `CompletableDeferred<Boolean>` instance. When a courier assignment is initiated, the request is registered and a deferred response is created. This deferred serves as a synchronization point, allowing the backend to suspend execution while awaiting the courier's response. Once the courier accepts or declines the assignment, the complete function is invoked to resolve the deferred and resume the assignment logic accordingly.

```
1  object AssignmentCoordinator {
2      // requestId -> pending decision
3      private val pendingResponses = ConcurrentHashMap<Int, CompletableDeferred
           <Boolean>>()
4
5      // Creates response promise for a request assignment
6      fun register(requestId: Int) = CompletableDeferred<Boolean>().also {
           pendingResponses[requestId] = it }
7
8      // Handles courier's response to a request assignment
9      fun complete(requestId: Int, accepted: Boolean) =
10         pendingResponses.remove(requestId)?.complete(accepted) ?: false
11 }
```

Listing 12: Assignment Coordination

5.6 Canceling Orders

Order cancellations were designed to be seamless, so if a courier cancels before pickup, the system treats the order as unassigned and assigns it to nearby couriers. However if the cancellation occurs after pickup, the order is re-assigned using the courier's last known location as the new pickup point. The following sketch illustrates the two possible user journeys when a courier cancels an order.

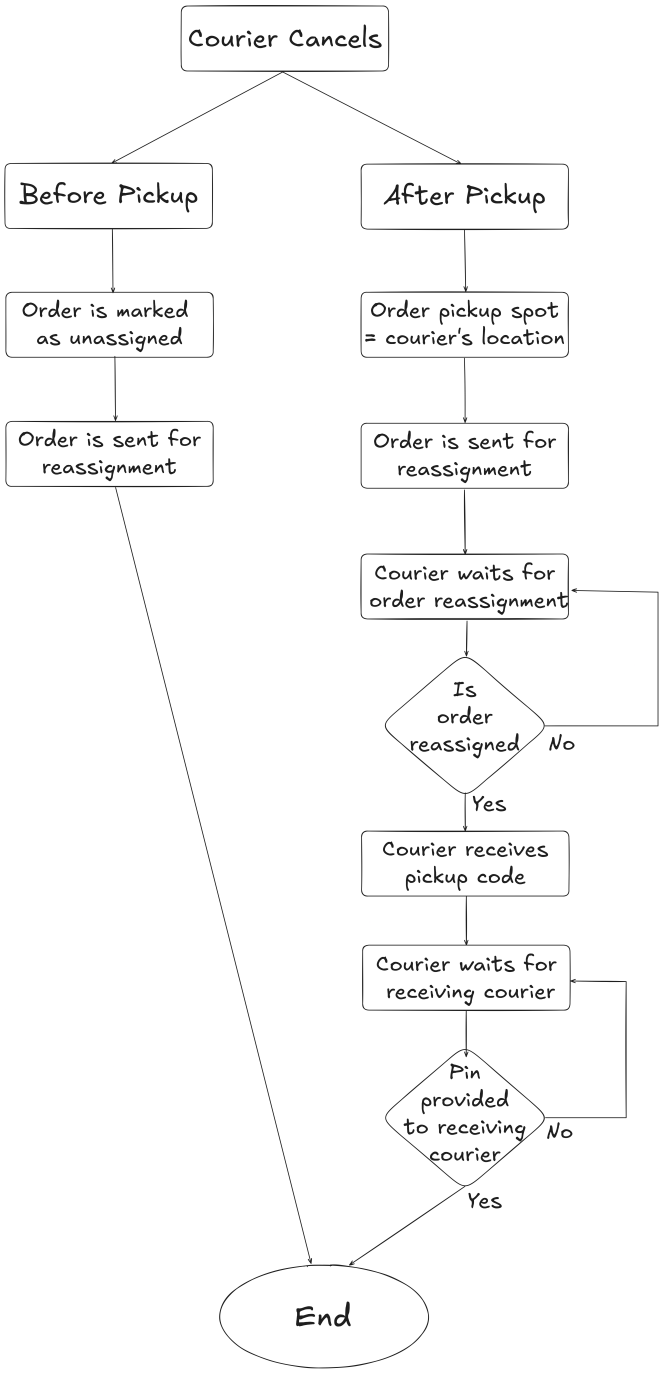


Figure 10: Courier Order Canceling Journey

Before reassigning a canceled request, the system determines the appropriate pickup coordinates and pin code based on the current delivery status. If the request was canceled after pickup and is successfully reassigned to a new courier, the backend sends a WebSocket message to the original canceling courier containing the pin code. This code is then passed along to the newly assigned courier to ensure the delivery can still be completed securely.

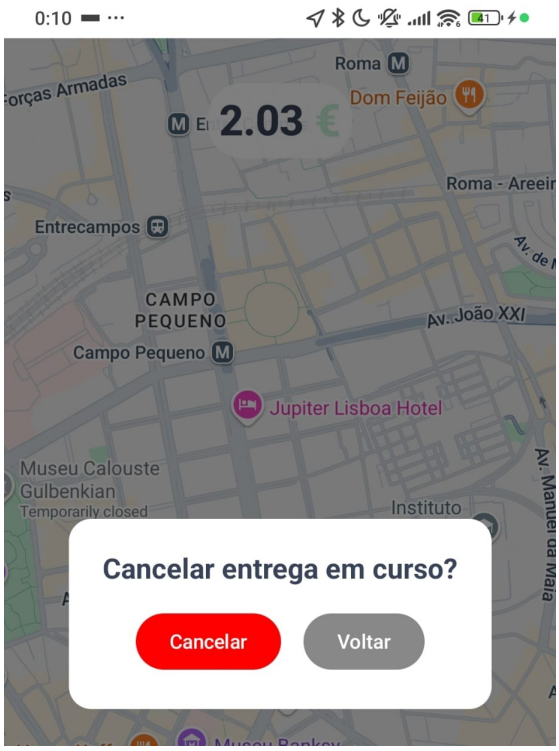
```
fun handleRequestReassignment(
        requestId: Int,
        courierId: Int,
        deliveryStatus: DeliveryStatus,
        pickupLocationDTO: LocationDTO?,
    ): Boolean {
    return transactionManager.run {
            ... // pickupLat, pickupLon and pickupPin are determined
                according to the delivery status
        CoroutineScope(Dispatchers.IO).launch {
            if (handleCourierAssignment(
                        pickupLat = pickupLat,
                        pickupLon = pickupLon,
                        requestId = requestId,
                        deliveryKind = deliveryStatus.toDeliveryKind().name,
                    )
                ) {
            if (deliveryStatus == DeliveryStatus.HEADING_TO_DROPOFF) {
                courierWebSocketHandler.sendMessageToCourier(
                            courierId = courierId,
                            message = DeliveryUpdateMessage(
                                    hasBeenAccepted = true,
                                    pinCode = pickupPin
                                ),
                        )
                    } } }
            return@run true
        } }
```

Listing 13: Reassignment of a cancelled request

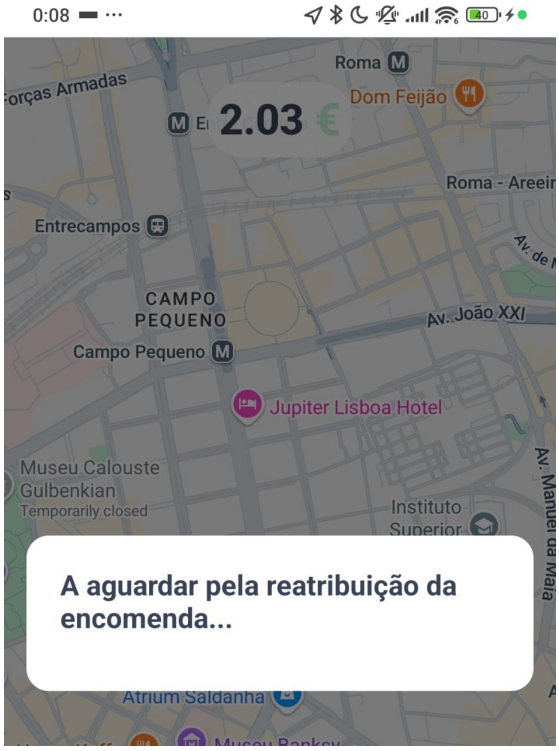
```
fun completeReassignment(requestId: Int) {
    // Gets courierId of cancelling courier
    val courierId = ...
    // Notifies the courier that the pickup cancellation was processed
        successfully
    courierWebSocketHandler.sendMessageToCourier(
            courierId = courierId,
            message = DeliveryUpdateMessage(
                hasBeenAccepted = true,
                hasBeenPickedUp = true
            ) )
    } }
```

Listing 14: Completion of a post-pickup cancelled request

The following can be observed when that happens in the following figures:

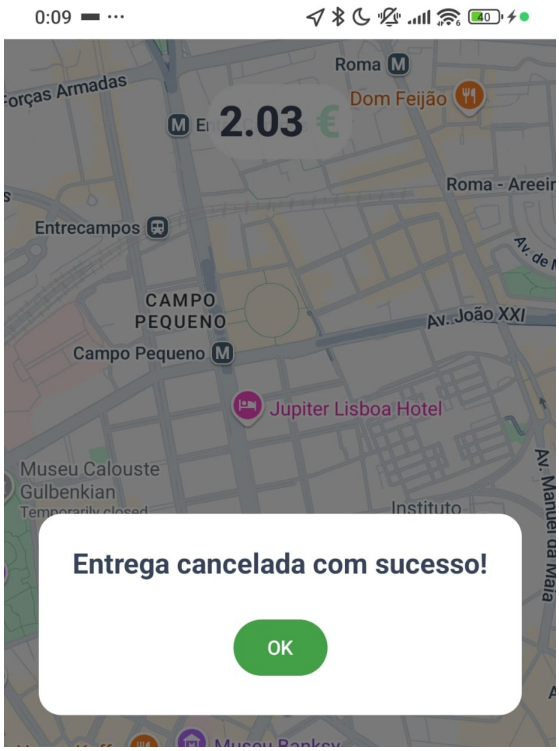


(a) Cancel confirmation screen



(b) Reassignment in progress screen

(c) Pickup code for reassigned courier



(d) Successful cancellation confirmation

Figure 11: UI states for order cancellation and reassignment flow

5.7 Courier Earnings

LiftDrop calculates courier earnings using a mix of fixed and dynamic components:

- **Base Fee**: A flat fee for completing any order.
- **Distance Fee**: A per-kilometer rate based on travel distance from courier to pickup.
- **Item Value Multiplier**: A bonus based on order value to reflect risk or effort for higher-priced deliveries.
- **Quantity**: All components are scaled by the number of items in the delivery.

The Kotlin function below shows the earnings calculation:

```
fun estimateCourierEarnings(
        distanceKm: Double,
        itemValue: Double,
        quantity: Int,
        baseFee: Double = 2.0,
        perKmRate: Double = 0.5,
        valueRate: Double = 0.005,
    ): Double = quantity * (baseFee + (distanceKm * perKmRate) + (itemValue *
         valueRate))
```

Listing 15: Calculation of Courier Earnings

5.8 Database Access

The LiftDrop backend interacts with the PostgreSQL database through a transactional abstraction based on JDBI. The design separates transaction management from repository implementation, ensuring consistency and maintainability across all database operations.

The following Kotlin class handles transactional execution using JDBI:

```
@Component
class JdbiTransactionManager(
    private val jdbi: Jdbi,
) : TransactionManager {
    override fun <R> run(block: (Transaction) -> R): R =
        jdbi.inTransaction<R, Exception> { handle ->
            val transaction = JdbiTransaction(handle)
            block(transaction)
        }
}
```

Listing 16: JDBI Transaction Manager

The ‘JdbiTransaction‘ class exposes repository instances tied to the same underlying handle, allowing atomic operations across multiple tables:

```
class JdbiTransaction(
    private val handle: Handle,
) : Transaction {
    override val usersRepository = JdbiUserRepository(handle)
    override val clientRepository = JdbiClientRepository(handle)
    override val requestRepository = JdbiRequestRepository(handle)
    override val courierRepository = JdbiCourierRepository(handle)
    override val deliveryRepository = JdbiDeliveryRepository(handle)
    override val locationRepository = JdbiLocationRepository(handle)
}
```

Listing 17: JDBI Transaction with Repository Bindings

5.9 Error Handling

To model operation results consistently, the backend uses a custom Either type to represent success or failure, following concepts learned in the DAW course.

5.9.1 Result Modeling with `Either`

```
sealed class Either<out L, out R> {
    data class Left<out L>(val value: L) : Either<L, Nothing>()
    data class Right<out R>(val value: R) : Either<Nothing, R>()
}

fun <R> success(value: R) = Either.Right(value)
fun <L> failure(error: L) = Either.Left(error)
```

Listing 18: Functional Result Modeling with Either

5.9.2 Standardized API Errors with Problem Details

For all HTTP interactions, the backend follows the **RFC 7807 Problem Details for HTTP APIs** specification. This standard defines a JSON structure for expressing errors.

```
data class Problem(
    val type: String?,
    val title: String?,
    val status: Int?,
    val detail: String?,
    val instance: String? = null
)
```

Listing 19: Problem Details Model

Responses use the `application/problem+json` media type and include fields such as `status`, `title`, and `detail` to clearly describe what went wrong.

A centralized factory exposes named constructors like:

- `courierNotNearPickup()`
- `ratingAlreadyDone()`

Each problem type links to documentation hosted at:
`https://github.com/isel-sw-projects/2025-lift-drop/tree/main/docs/problems`

5.9.3 Global Exception Handling

All errors not handled by business logic are intercepted by a custom Spring `@ControllerAdvice`, shown below:

```
@ControllerAdvice
class CustomExceptionHandler : ResponseEntityExceptionHandler() {

    override fun handleMethodArgumentNotValid(...) =
        Problem.invalidRequestContent("Argument received is not valid")
               .response(HttpStatus.BAD_REQUEST)

    override fun handleTypeMismatch(...) =
        Problem.invalidRequestContent("There is a type mismatch")
               .response(HttpStatus.BAD_REQUEST)



    override fun handleHttpMessageNotReadable(...) =
        Problem.invalidRequestContent("Http message is not readable")
               .response(HttpStatus.BAD_REQUEST)

    @ExceptionHandler(Exception::class)
    fun handleAllExceptions(ex: Exception, request: WebRequest):
        ResponseEntity<String> {
        GlobalLogger.log("Http.CustomExceptionHandler AllExceptions: $ex")
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body("Internal Server Error")
    }
}
```

Listing 20: Global Exception Handler

This handler catches unexpected errors, logs them, and sends these error messages back to clients.

6 Frontend

6.1 Frontend State Management

6.1.1 Message Handling and State Updates

The LiftDrop mobile application adopts a reactive UI architecture using Jetpack Compose, where screen behavior is driven by immutable state objects. Each screen is modeled using a dedicated sealed class that encapsulates its possible UI states. The use of state-driven design and reactive patterns in the development of the application was informed by concepts learned in the PDM course.

For example, the login and registration processes are controlled by the `LoginScreenState` and `RegisterScreenState` classes, respectively. These include the following variants:

LoginScreenState: `Idle`, `Login`, `Error`

RegisterScreenState: `Idle`, `Register`, `Error`

These states allow the UI to reflect user progress during authentication, including success and failure outcomes.

The Home screen follows a more complex state model, represented by the sealed class `HomeScreenState`. The simplified listing below outlines all defined UI states used throughout the delivery lifecycle.

```
sealed class HomeScreenState {
    data class Listening(...) : HomeScreenState()
    data class RequestAccepted(...) : HomeScreenState()
    data class RequestDeclined(...) : HomeScreenState()
    data class HeadingToPickUp(...) : HomeScreenState()
    data class PickedUp(...) : HomeScreenState()
    data class HeadingToDropOff(...) : HomeScreenState()
    data class Delivered(...) : HomeScreenState()
    data class Idle(...) : HomeScreenState()
    data class Error(...) : HomeScreenState()
    data class Logout(...) : HomeScreenState()
    data class CancellingOrder(...) : HomeScreenState()
    data class CancellingPickup(...) : HomeScreenState()
    data class CancellingDropOff(...) : HomeScreenState()
}
```

Listing 21: Simplified HomeScreenState sealed class

All screen states are managed using a `StateFlow` in the ViewModel layer. When updates are received, either from real-time WebSocket messages or local UI interactions, the corresponding ViewModel processes the event and updates the screen state accordingly.

Error handling is also integrated into the state model. Whenever an unexpected event occurs, the ViewModel updates the state to `HomeScreenState.Error`. The UI reacts accordingly, displaying contextual information to the user, such as the problem title and message, to clearly indicate what went wrong. To better illustrate how the error state appears in practice, Figure 11 shows the application's response when a delivery fails mid-process.

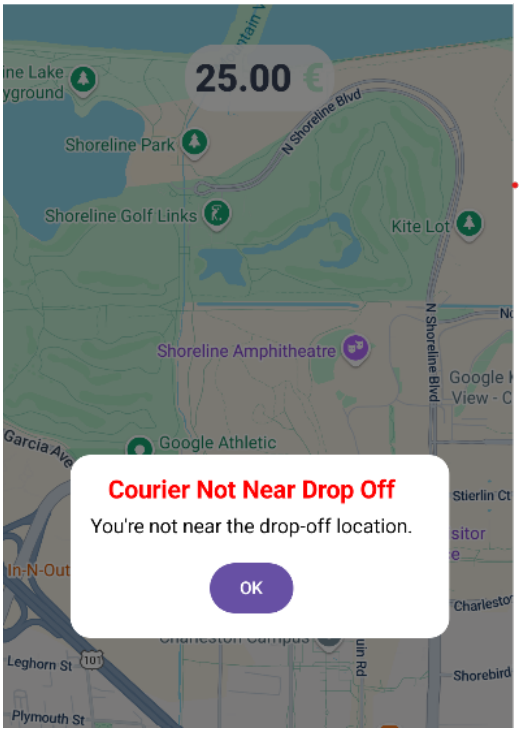


Figure 12: Error screen state when delivering order

The following example illustrates how the Home screen begins listening for messages and updates its state based on the received input:

```
fun startListening() {
    viewModelScope.launch {
        val token = ... // Retrieved from DataStore
        homeService.startListening(
            token = token,
            onMessage = { msg ->
                when (val req = parseDynamicMessage(msg)) {
                    is IncomingRequestDetails -> _stateFlow.update {
                        processIncomingRequest(it, req) }
                    is DeliveryUpdate         -> _stateFlow.update {
                        processDeliveryUpdate(it, req) }
                    is ResultMessage          -> _stateFlow.update {
                        processResultMessage(it, req) }
                    else                      -> ...
                }
            },
            onFailure = { _stateFlow.update { ... } }
        )
        _stateFlow.update { ... } // Initialize to Listening state
    }
}
```

Listing 22: Simplified WebSocket message handling in startListening

Incoming WebSocket messages are sent in a unified JSON format and differentiated by a `type` field. The `parseDynamicMessage` function reads this field and dynamically maps the message to its corresponding class using Jackson:

```
fun parseDynamicMessage(message: String): HomeMessage {
    val mapper = jacksonObjectMapper()
    val root: JsonNode = mapper.readTree(message)
    val typeValue = root.get("type")?.asText()
        ?: throw IllegalArgumentException("Missing type field")

    val type = MessageType.fromValue(typeValue)
        ?: throw IllegalArgumentException("Unknown message type: $typeValue")

    return when (type) {
        MessageType.DELIVERY_REQUEST ->
            mapper.treeToValue(root, IncomingRequestDetails::class.java)
        MessageType.DELIVERY_UPDATE ->
            mapper.treeToValue(root, DeliveryUpdate::class.java)
        MessageType.SUCCESS, MessageType.ERROR ->
            mapper.treeToValue(root, ResultMessage::class.java)
    }
}
```

Listing 23: Dynamically parsing incoming WebSocket messages

The following diagram provides a high-level overview of a complete delivery request assignment cycle, illustrating how messages are exchanged between the backend and the courier application via WebSocket. It captures the progression from the initial assignment message to the courier's decision and the backend's final acknowledgment, all of which contribute to state transitions within the application.

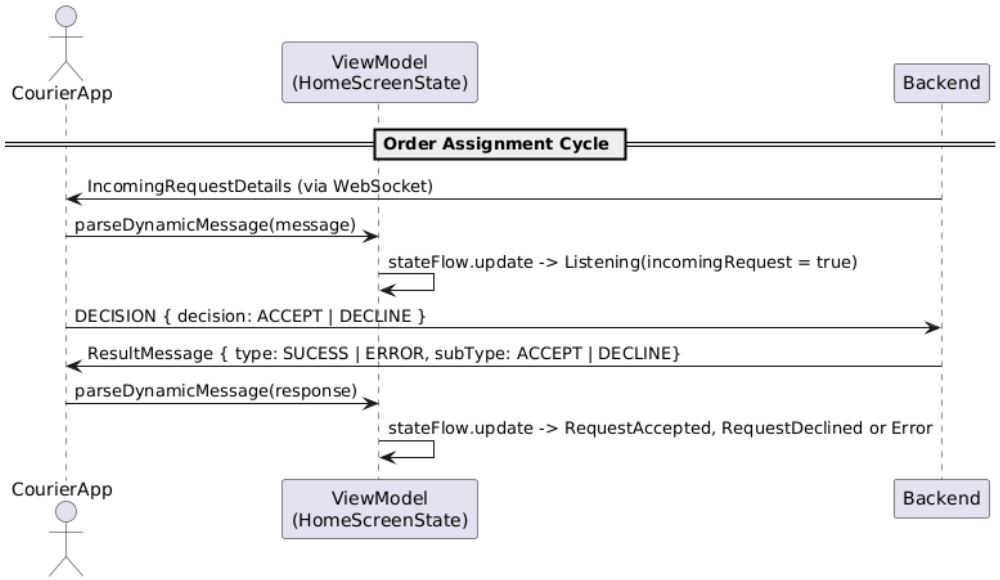
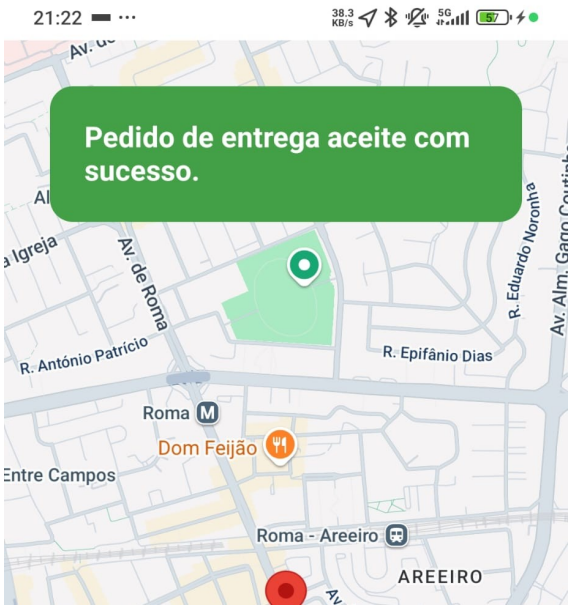


Figure 13: Message exchange between backend and frontend during a delivery request cycle

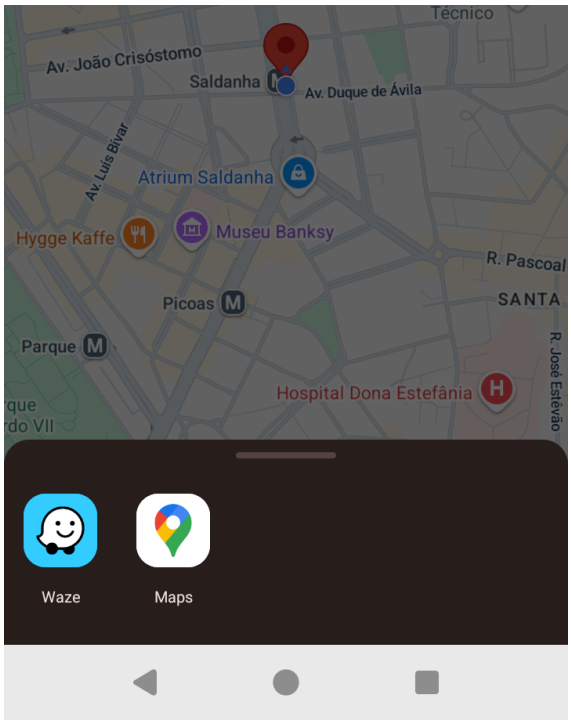
Before heading to either the pickup or drop-off location, couriers are prompted to choose their preferred navigation application. Instead of enforcing a default, LiftDrop uses Android's intent system to display a list of compatible apps installed on the device, such as Google Maps, Waze, or other mapping tools. Once the courier selects an app, it is launched with the appropriate destination coordinates already filled in.

The user interface mirrors this flow as illustrated below:

(a) Pickup confirmation screen



(b) Delivery confirmation screen



(c) Choose Navigation Tool

Figure 14: Screens for confirming pickup, delivery and preferred navigation application choice

6.1.2 State Recovery and Rollback Mechanism

To improve user experience and reduce friction after transient failures, the application implements a rollback strategy using a dedicated `previousState` flow. This auxiliary state is used to restore the last known valid screen state following an error, allowing the user to seamlessly resume their previous workflow.

Rather than manually capturing previous states through function calls, the ViewModel initializes a coroutine that continuously tracks screen state transitions. By using the `scan` operator on the `stateFlow`, the ViewModel records the most recent prior value on each emission:

```
init {
        viewModelScope.launch {
            _stateFlow
                .scan(_previousState.value to _stateFlow.value) { acc, new ->
                    acc.second to new
                }
                .collectLatest { (prev, _) ->
                    _previousState.value = prev
                }
        }
    }
```

Listing 24: Tracking previous screen state on ViewModel initialization

Screen state transitions throughout the app follow three distinct patterns, each chosen based on the nature of the state change:

- `transitionTo(state)` is used in rare cases where the next state does not depend on the current one—such as logging out, where previous context is irrelevant.
- `raiseError(problem)` is used exclusively to transition the screen into an `Error` state when a backend service call fails and returns a `Problem` object. This keeps error handling consistent across service-related failures.
- `_stateFlow.update { current -> ... }` is the primary mechanism for all other transitions where the next state depends on the current one.

6.1.3 Frontend HTTP Service Abstraction

While WebSocket communication powers interactions between the courier and backend, the LiftDrop mobile application also relies on traditional HTTP requests for several operations, including authentication, data retrieval and delivery related operations.

To manage these requests consistently, this service is using the `OkHttpClient` library, which provides a robust and efficient way to perform network operations on Android.

The `HttpService` encapsulates the logic for performing `GET`, `POST`, and `DELETE` requests, including header configuration, exception handling, and response parsing using Gson.

The following snippet shows the main structure of the `HttpService` class:

```
class HttpService(
    val baseUrl: String,
    val client: OkHttpClient,
    val gson: Gson
) {
    suspend inline fun <reified T> get(url: String, token: String): Result<T>
        { ... }

    suspend inline fun <reified T, reified R> post(url: String, data: T,
        token: String): Result<R> { ... }

    suspend inline fun <reified T> delete(url: String, token: String): Result
        <T> { ... }
}
```

Listing 25: Frontend HttpService class for coroutine-based requests

Each HTTP method in `HttpService` returns a sealed `Result` type.

The definition of the `Result` class is shown below:

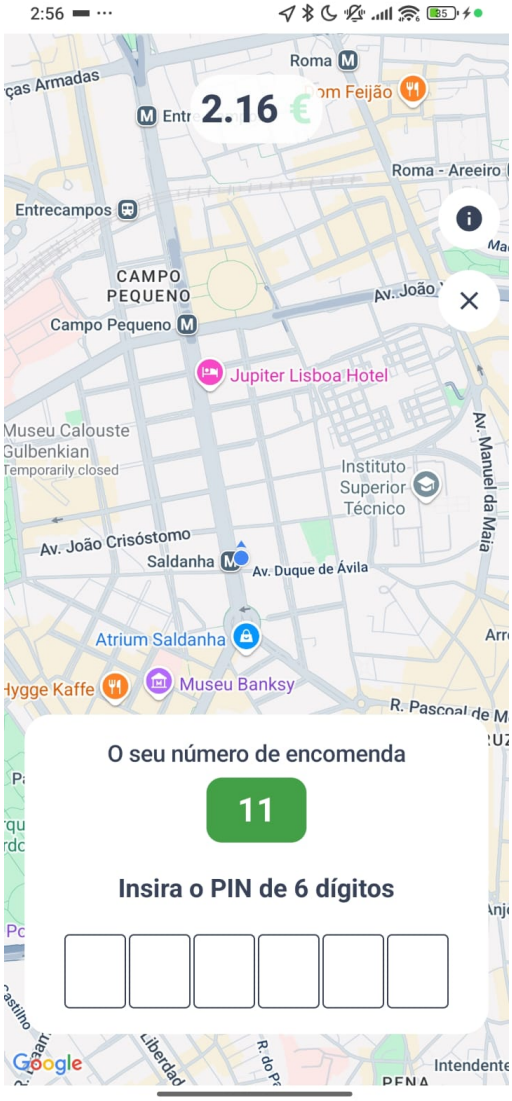
```
sealed class Result<out T> {
    data class Success<out T>(val data: T) : Result<T>()
    data class Error(val problem: Problem) : Result<Nothing>()
}
```

Listing 26: Sealed Result class used for HTTP responses

6.2 Pickup and Drop-off Code Validation

As part of the delivery confirmation flow, LiftDrop includes a code-based verification step to ensure that orders are only picked up or delivered by the correct courier. When a courier reaches the pickup or drop-off location, they can slide to pickup or drop off the order.

This verification screen is shown only when the courier is sufficiently near the designated location (within 100 meters). If the provided code matches the one stored in the backend for that delivery, the delivery status proceeds to the next phase. Otherwise, an error state is triggered, and the user is prompted to retry.



(a) Code entry screen for validating pickup or drop-off

(b) Delivery confirmation screen

6.3 Persistent Local Storage with DataStore

To persist authentication and user-specific data across sessions, LiftDrop uses Jetpack's `PreferencesDataStore`, a modern, coroutine-based alternative to `SharedPreferences`. It provides a lightweight and lifecycle-aware solution for storing key-value data such as the courier's ID, username, email, and bearer token.

These values are required to initiate authenticated requests and resume background services like location tracking. By storing them locally, the application can restore the session on cold start, reducing user friction and enabling seamless continuity. The snippet below illustrates how the stored data is retrieved and reconstructed into a `UserInfo` object:

```
class PreferencesDataStore(
    private val store: DataStore<Preferences>
) : PreferencesRepository {
override suspend fun getUserInfo(): UserInfo? {

    val preferences = store.data.first()
    val username = preferences[nameKey]
    val courierId = preferences[idKey]
    val token = preferences[tokenKey]
    val email = preferences[emailKey]

    return if (username != null && courierId != null && token != null &&
        email != null) {
        UserInfo(
            id = courierId,
            username = username,
            email = email,
            bearer = token,
        )
    } else null
} }
```

Listing 27: Retrieving stored user data from PreferencesDataStore

The `PreferencesRepository` interface is injected into the application through a central `DependenciesContainer`, as shown below. This allows the ViewModel and service layers to access user preferences with ease.

```
interface DependenciesContainer {
    val preferencesRepository: PreferencesRepository
    // Other services...
}

override val preferencesRepository: PreferencesRepository
    get() = PreferencesDataStore(dataStore)
```

Listing 28: Injecting PreferencesDataStore via DependenciesContainer

6.4 Courier Location Updates

Courier location tracking is implemented using Android **Foreground Services**, which allow the app to run and transmit location data even while running in the background.

To ensure efficient backend communication, the app retrieves the courier's location every **30 seconds** but only transmits it under two conditions:

- The courier is **logged in** and authenticated;
- The courier has set their status to **waiting for orders**.

Additionally, the system only sends a location update if the courier has moved more than **50 meters** since the last recorded position. This strategy minimizes unnecessary network traffic while maintaining sufficient tracking accuracy.

The 50-meter threshold was chosen based on common urban delivery speeds (e.g., 30 km/h for scooters or e-bikes), allowing meaningful movement to be captured without overwhelming the server. Furthermore, a delivery can only be confirmed if the courier is within 100 meters of the drop-off point—making the location filter both efficient and functionally sound.

The simplified Kotlin pseudocode below outlines the core logic used for checking movement and conditionally sending location updates:

```
override fun startUpdating(authToken: String, courierId: String) {
    // Defines a high-accuracy location request every 30 seconds
    val request = LocationRequest.Builder(Priority.PRIORITY_HIGH_ACCURACY, 30
        _000L)
        .setMinUpdateIntervalMillis(30_000L).build()

    // Defines a callback to handle incoming location updates
    locationCallback = object : LocationCallback() {
        override fun onLocationResult(result: LocationResult) {
            val location = result.lastLocation ?: return

            // Launches coroutine to evaluate whether the update should be
                sent or not
            coroutineScope.launch {
                val movedEnough = lastSentLocation?.distanceTo(location)?.let
                    { it > MIN_DISTANCE_METERS } != false
                if (movedEnough) {
                // Sends updated location to backend
                sendLocationToBackend(location.latitude, location.longitude,
                    courierId, authToken)
                    lastSentLocation = location
                }
            }
        }
    }
    // Start location updates using the defined request and callback
    fusedLocationClient.requestLocationUpdates(request, locationCallback!!,
        Looper.getMainLooper())
}
```

Listing 29: Condensed Kotlin-style pseudocode for Courier location updates mechanism

7 Tests

We tested LiftDrop using a mix of automated and manual tests. The backend used automated testing exclusively, while the frontend relied solely on manual testing.

7.1 Manual Tests

Manual tests were used primarily to validate frontend behavior by observing real-time UI updates in response to API calls and courier-client interactions. Using Postman, we simulated HTTP requests to trigger and monitor frontend changes, while the use of Fake GPS allowed us to test location-based UI updates without physical movement. This approach ensured the frontend reflected all expected state changes during end-to-end scenarios.

7.2 Automated Tests

Automatic tests were written using the **JUnit** testing framework to validate the backend's core business logic. These tests were executed during development to verify key modules in isolation.

Test coverage included:

- Unit tests for repositories and services
- Validation of order assignment logic
- Authentication and authorization checks

In future iterations, these tests can be extended with integration and UI testing using tools like Espresso or Retrofit mock services.

8 Final Remarks

8.1 Limitations and Future Work

While all features outlined in the project scope were implemented, some features were intentionally left out due to time constraints. This section outlines key areas where the system could be expanded or improved in future iterations.

8.1.1 Future Improvements

- **Client Application Interface**: The current system includes a simulated client API for testing, but no dedicated client-facing mobile application was developed.
- **WebSocket Reconnection Support**: The WebSocket implementation currently lacks fault-tolerance mechanisms for automatic reconnection in case of network disruptions. This may affect delivery request communication stability on unreliable connections.
- **Enhanced Earnings Model**: Extend the current earnings calculation to include compensation for time spent waiting at pickup locations, in addition to distance and item value.
- **Kotlin Multiplatform (KMP) Integration**: By using Kotlin Multiplatform, it would be possible to develop an application for iOS devices, expanding the app's reach to more mobile users.
- **Internationalization (i18n)**: Support for multiple languages to improve accessibility and usability for international users.

8.2 Conclusion

In conclusion, this project offered a focused exploration of the courier-facing side of food delivery platforms. Initially, we planned a full-stack delivery platform but pivoted to a courier-facing system with mocked clients due to time constraints. Even with this narrower scope, the project proved challenging as we faced dilemmas about optimal request assignment. After discussions with our project advisors and our reviewer, we managed to integrate order distribution, real-time communication, location tracking, and realistic delivery service simulations. This project's development process combined knowledge acquired throughout the degree with the adoption of new technologies such as the Google Maps API and WebSockets.

References

[1] Spring Framework. *Spring Web MVC Documentation*. Available at: `https://spring.io/projects/spring-framework`

[2] Google Developers. *Android Jetpack Compose*. Available at: `https://developer.android.com/jetpack/compose`

[3] Google Cloud. *Google Maps Platform Documentation*. Available at: `https://developers.google.com/maps/documentation`

[4] PostgreSQL Global Development Group. *PostgreSQL Documentation*. Available at: `https://www.postgresql.org/docs/`

[5] MDN Web Docs. *WebSockets*. Available at: `https://developer.mozilla.org/en-US/docs/Web/API/WebSocket`

[6] JDBI Project. *JDBI Documentation*. Available at: `https://jdbi.org/`

[7] Glovo. *How It Works for Couriers*. Available at: `https://glovoapp.com`

[8] GitHub Docs. *Understanding the GitHub Flow*. Available at: `https://docs.github.com/en/get-started/quickstart/github-flow`

[9] JetBrains. *Kotlin Language Documentation*. Available at: `https://kotlinlang.org/docs/home.html`.

[10] Google. *Jetpack DataStore*. Available at: `https://developer.android.com/topic/libraries/architecture/datastore`.

[11] Google. *State in Jetpack Compose*. Available at: `https://developer.android.com/jetpack/compose/state`.

A EA Model

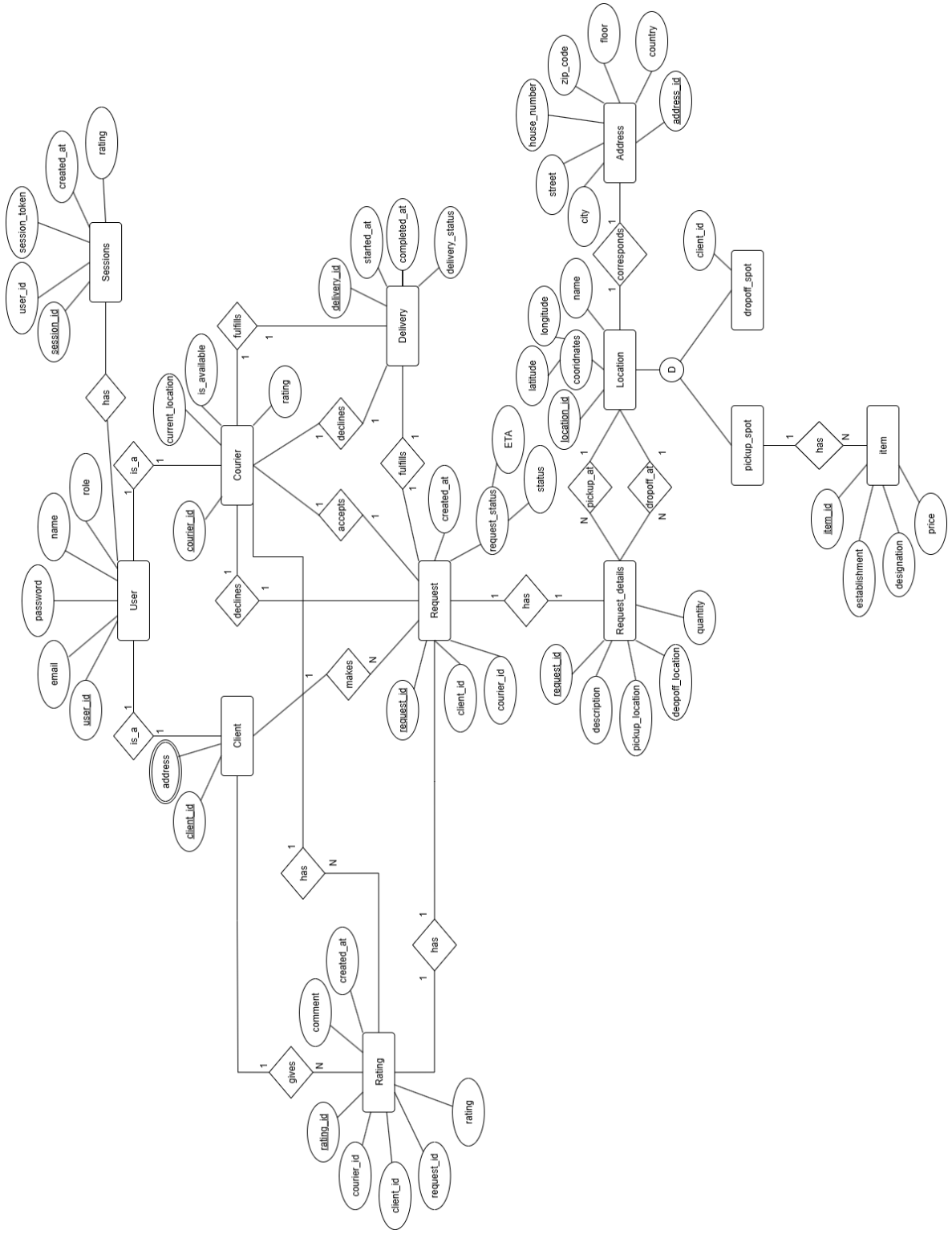


Figure 16: EA model

B Entity Class Diagram

This appendix provides the complete class diagram of the LiftDrop system. The diagram illustrates the main entities, their relationships, and the roles they play in the overall architecture.

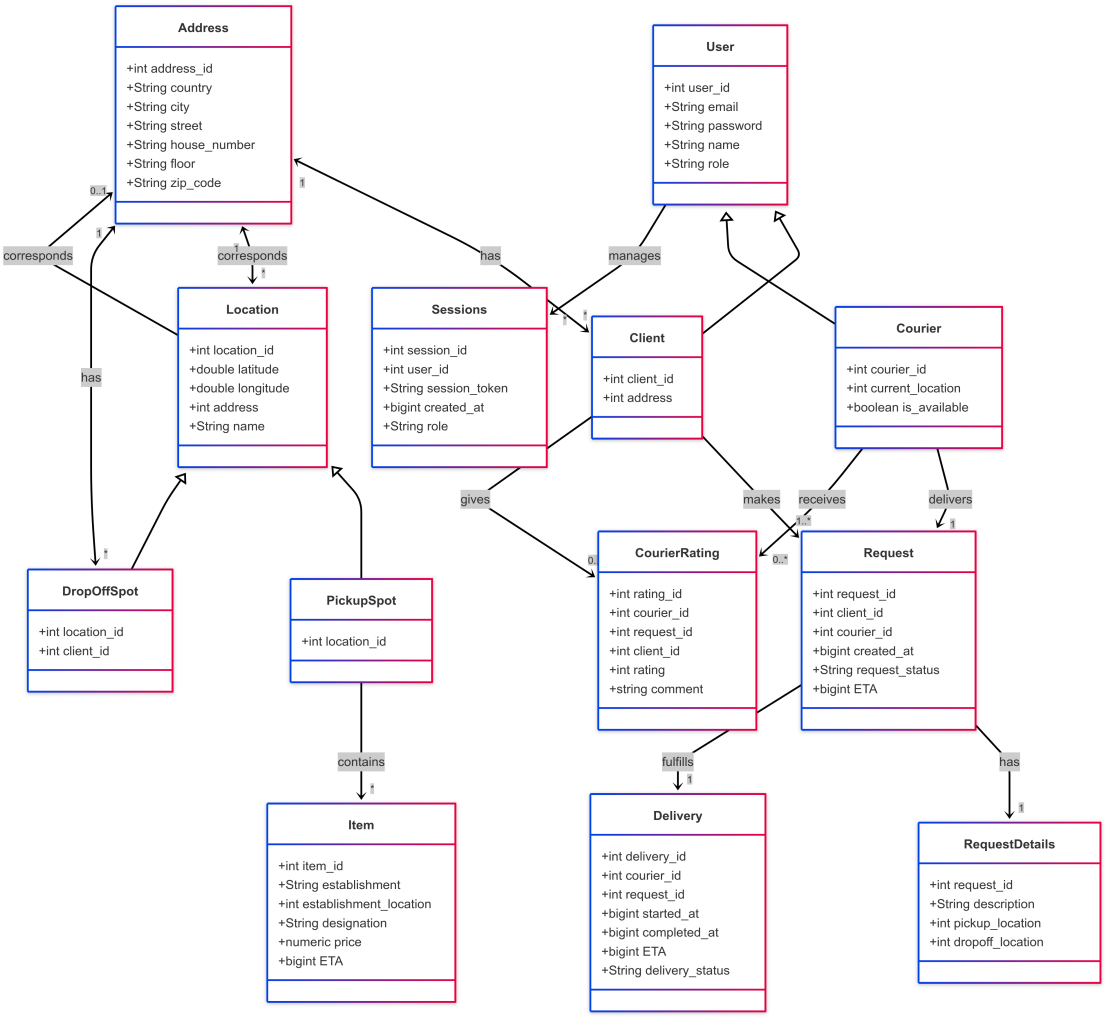


Figure 17: Full Class Diagram of the LiftDrop System