POLITECNICO
MILANO 1863

COMPUTER SCIENCE AND ENGINEERING
SOFTWARE ENGINEERING II
2025 - 2026

DD

Design Document

Best Bike Paths

Authors:

Leonardo Guglielmi, Francesco Lo Conte

Version:

1.0
(January 7, 2026)

Contents

1 Introduction

1.1 Scope

1.1.1 Product domain

The scope of the project covers the users interacting with the Best Bike Paths (BBP) system, the data collection processes regarding cycling routes, and the navigation services provided to the community.

For the project, the following users interacting with the system have been identified:

- **Registered Users.**
- **Generic Users.**

Registered Users will be able to record their trips using the mobile application, tracking their performance statistics and path data. During the recording, they contribute to the system by collecting data either automatically (via device sensors) or manually (by reporting obstacles or status). Upon completion of a trip, they can review and confirm the detected anomalies, making them available to the community.

Generic Users (along with Registered Users) can access the platform to search for the best cycling routes between two locations. The system provides them with routes ranked by a "Path Score", calculated based on the aggregated data provided by the community, allowing them to visualize safety information and obstacles on the map.

The system acts as a mediator between the raw data collected from the real world (road conditions) and the end-users, processing this information to ensure safety and reliability.

1.1.2 Main architectural choices

The backbone of the system is to be implemented using a **microservices-oriented architecture**. This choice is driven by the need for a scalable and maintainable system, capable of handling different loads on different components (e.g., the data ingestion service may face high traffic during weekends, while the reporting service might be less stressed).

This architecture allows for:

- **Independent Scaling:** Individual components can be scaled based on their specific resource requirements.
- **Resilience:** If a specific microservice fails (e.g., the weather enrichment service), the core functionality remains available.
- **Technology Agnosticism:** Different teams can develop different services using the most appropriate technologies for each task.

Furthermore, the system adopts a Client-Server model where the mobile application performs significant local processing (Edge Computing) to analyze sensor data before sending it to the backend, optimizing bandwidth and responsiveness.

1.2 Definitions, acronyms, abbreviations

This section contains the definitions for terms that may be technical or specific to the architecture, as well as acronyms and abbreviations used throughout the document.

1.2.1 Definitions

- **Microservice:** A software development technique that arranges an application as a collection of loosely coupled services.
- **API Gateway:** A server that acts as an API front-end, receiving API requests, enforcing throttling and security policies, passing requests to the back-end service and then passing the response back to the requester.
- **Edge Computing:** A distributed computing paradigm that brings computation and data storage closer to the sources of data (in this case, the user's smartphone).

1.2.2 Acronyms

- **BBP:** Best Bike Paths
- **RASD:** Requirement Analysis and Specification Document
- **DD:** Design Document
- **API:** Application Programming Interface
- **REST:** Representational State Transfer
- **DBMS:** DataBase Management System

1.2.3 Abbreviations

- **R*:** Requirement

1.3 Reference documents

This document is based on the following materials:

- The specification of the RASD and DD assignment of the Software Engineering II course a.y. 2025/26.
- Course slides shared on WeBeep.

- The Requirement Analysis and Specification Document (RASD) v1.0 of Best Bike Paths.
- Past Design Documents.

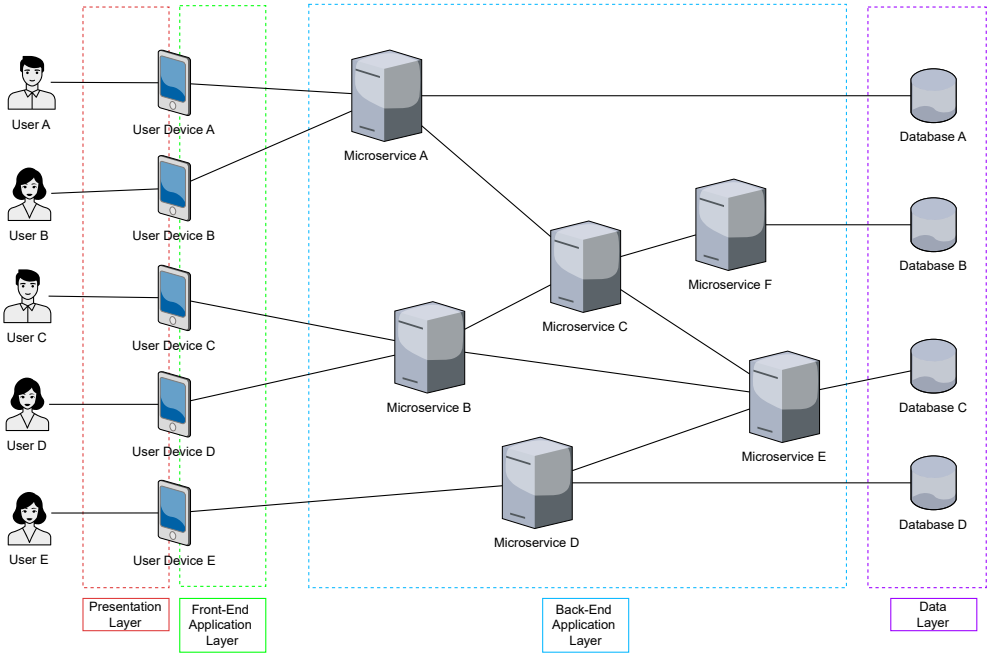
1.4 Overview

1. **Introduction:** this section introduces the project. It contains a high level description of the system, including its architectural style and architectural choices.
2. **Architectural design:** this section is very broad and contains the description of the various interfaces of the system, its deployment and an in-depth description of the components and their interactions (Runtime view).
3. **User interface design:** this section focuses on the user interface design, expanding on what was shown in the RASD.
4. **Requirements traceability:** this section contains a mapping between the requirements defined in the RASD and the design elements defined in the DD.
5. **Implementation, integration and test plan:** this section describes the order in which the various components and subsystems must be developed and tested.
6. **Effort spent:** this section shows the time spent on each section of the document, for each member of the group.
7. **References:** this section contains all the various references used to write this document.

2 Overall description

2.1 Overview

The architecture is defined by two primary structural decisions: the adoption of a Microservices Architectural Style for the backend and a 4-Layer Architectural Approach to organize the logical and physical distribution of components.



For the BBP system we opted to base the architecture on a **4-Layer Approach**. This structural design was chosen to ensure a strict separation of concerns, thereby maximizing both maintainability and operational efficiency. The system is decomposed in the following distinct layers:

1. *Presentation Layer*, which serves as point of contact with the user. It is strictly responsible for managing the User Interface (UI) and orchestrating the direct interaction flow with the user.
2. *Front-End Application Layer*, which encapsulates the logic of the mobile application. The features contained in this layer are those regarding the management of ongoing activities. Another key point of this layer is the execution of an immediate, local analysis of raw data sampled during monitored activities, in order to reduce the volume of data transmission.

3. *Back-End Application Layer*, which acts as the central engine of the system. It houses the majority of the system functionalities, implementing complex functionalities such as trip planning algorithms and comprehensive account management.

4. *Data Layer*, which functions are responsible for the secure storage and retrieval of system's data.

Regarding the *Back-End Application Layer*, the system adopts the **Microservices Architectural Style**. Rather than relying on a monolithic structure, this approach decomposes the system into a suite of small, autonomous services, with each service laser-focused on a specific bounded context within the domain. Adopting this style provides a wide spectrum of strategic advantages. Primarily, it ensures high scalability and fault tolerance, while also allowing the architecture to leverage the benefits of geographical data distribution. Furthermore, it offers the flexibility to employ different programming technologies and languages best suited for specific services. To align with this decentralized principle, the *Data Tier* adopts a **distributed storage strategy**. Data is partitioned across different Database Management Systems (DBMS), where each instance is dedicated to a specific domain area—such as account management or trip handling, ensuring that the storage layer is as modular as the application logic it supports.

2.2 Component View

In this section and the following ones we focus on describing the *Back-end Application Layer* internal structure and its interaction with the *Data layer*.

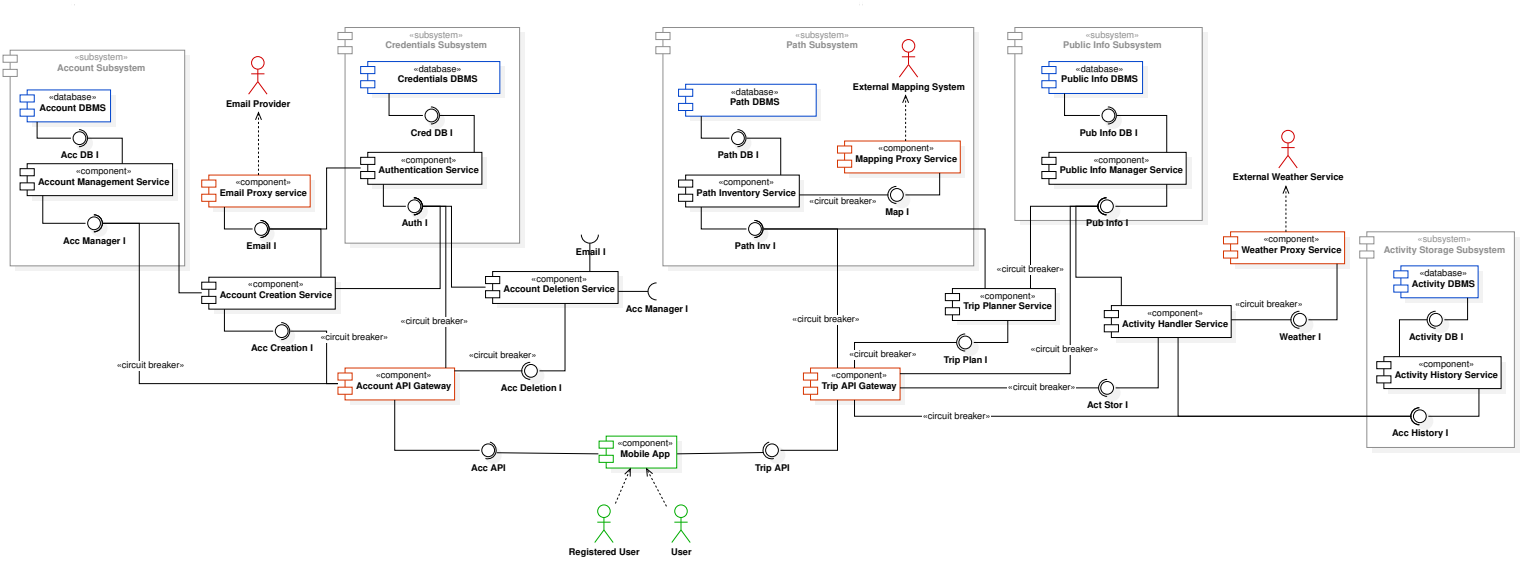


Figure 1: Component Diagram. In red are drown those actors external to our system; in blue the DBMS and those components in the Data tier; in green the mobile app residing on the user's device.

Mobile App

The Mobile Application serves as the primary user interface, enabling users to manage their accounts, browse available paths, and initiate trips with real-time monitoring. It also performs an early analysis of raw data collected during activities.

Account API Gateway

Implements the Gateway Pattern to provide a unified entry point for all account-related services.

Account Creation Service

The Account Creation Service manages the process of creating a new account. It orchestrates the communications with the user and the data storage by interfacing with the *Authentication Service* and the *Account Management Service*.

Account Deletion Service

This service handles the account deletion lifecycle. It interfaces with the *Authentication Service* to execute identity removal and manages outbound communication through the *Email Proxy Service* to verify and confirm the deletion request.

Credentials DBMS

This DBMS stores and manages account credentials.

Authentication Service

This service facilitates the authentication lifecycle. It processes login requests, performs identity verification by querying the *Credentials DBMS*, and acts as the primary interface for any credential-related data retrieval.

Accont DBMS

This DBMS stores and manages general account information, like name, surname or birthdate.

Account Management Service

Manages user account information and profile updates. It handles the access to the data stored in the *Account DBMS*, acting as an interface for all data retrieval operation involving account information.

Email Proxy Server

This component works as a middleware between the user *Email Provider* and those services which need to send an email, avoiding direct contact with something outside the system and allowing a more decoupled approach.

Email Provider

This actor represents the email provider.

Trip API Gateway

Implements the Gateway Pattern to provide a unified entry point for all trip services (both rides and activities) and related aspects, like issues and scores.

Trip Planner Service

This service is responsible for retrieving and merging all the information necessary to the user during trip planning. To achieve this, it interfaces with the *Path Inventory* Service to retrieve the path and with the *Public Info Manager* Service to retrieve scores and issues.

Activity Handler Service

This service scope is to handle the storage of completed activities by acting as a central coordinator. It executes this task by interfacing with the *Activity History Service* to record finished sessions. It also merges weather information into the record by interfacing with the *Weather Proxy Service*. Furthermore, it handles the storage of activity-related information such as path scores by interfacing with the *Public Info Manager Service*.

Path DBMS

This DBMS stores and manages all the informations about the paths.

Path Inventory Service

This service handles the path-related information stored in the *Path DBMS*, acting as management layer for all geographic route data. Upon request, it retrieves the specific information, and in case there isn't a match between the request and the stored data, the service handles the creation of a new path by asking it to the *Mapping Proxy Service*, ensuring that the internal database is updated with new coordinates and route details.

Mapping Proxy Service

This component acts as a security layer for communicating with the External Mapping System, serving as a protected gateway for all geographic requests that cannot be satisfied by the *Path Inventory Service*. Its also ensures that the internal architecture remains independent from interfaces external to our system.

External Mapping System

This actor represent a mapping system external to BBP boundaries.

Public Info DBMS

This DBMS stores and manages all the informations about *Publishable Information* submitted by registered users.

Public Info Manager Service

This service handles the Publishable Information published by the user, managing Issues and Path Scores. It fulfills this role by handling all requests about data stored in the *Public Info DBMS*, and by handling the issue-status updates within the system.

Weather Proxy Service

This component acts as an intermediary between the system and the *External Weather System* to facilitate secure and reliable activity data enrichment with the meteorological conditions present during the time of the trip.

External Weather System

This actor represents a weather system indipendent from the BBP system.

Activity DBMS

This DBMS stores and manages all the informations about activities.

Activity History Service

This service handles the activity-related requests within the system by interfacing with the *Activity DBMS*, serving as the primary management layer for recorded user data. It is directly involved in those operations concerning the activity history of a user.

Circuit Breaker

In order to avoid ripple effect slowing down the entire system, *Circuit Breakers* are positioned on those components which should have high responsivness or that interface with external services. All *Circuit Breakers* depicted in Figure 2.2 are positioned on the calling service side.

2.3 Deployment View

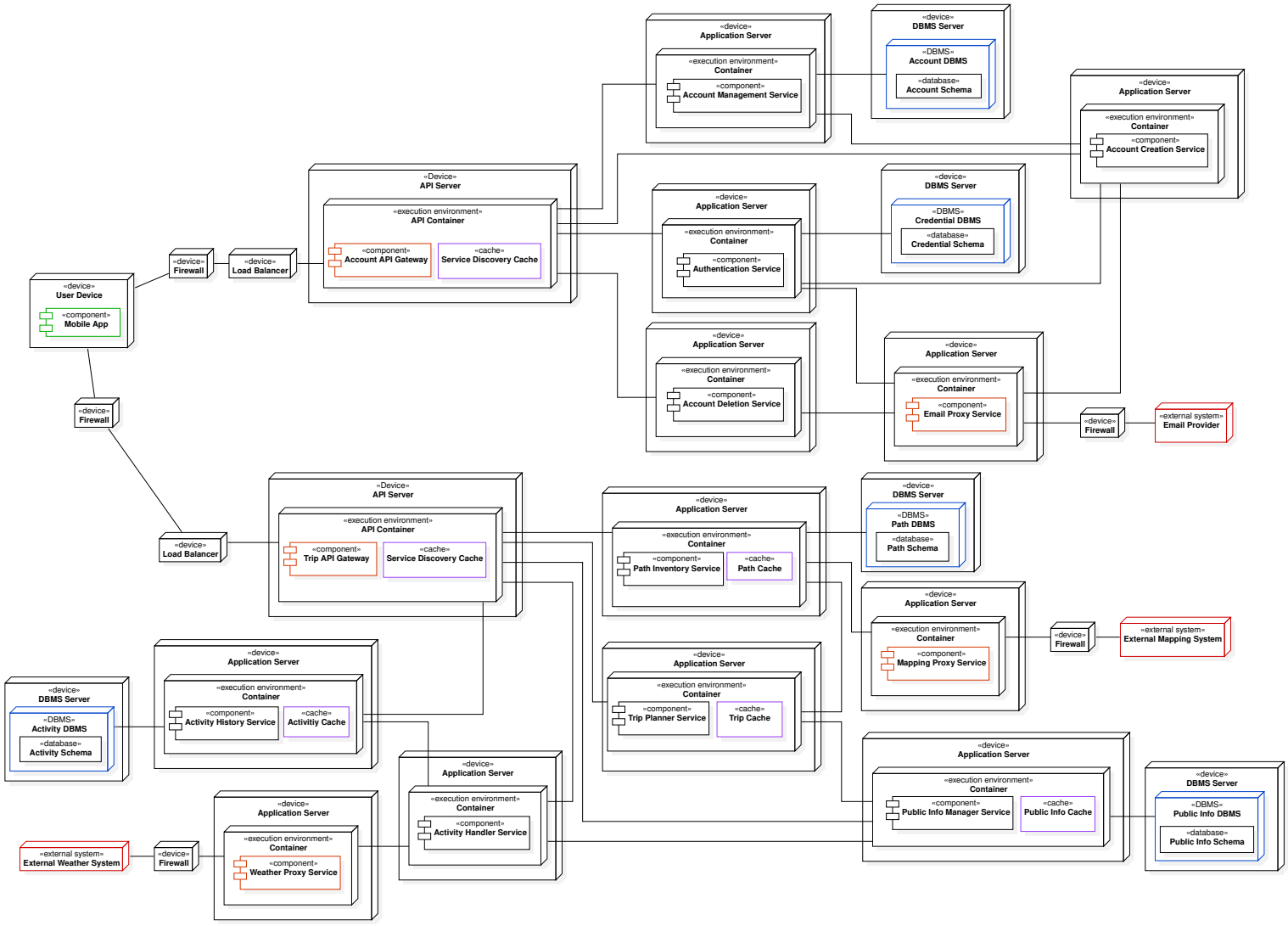


Figure 2: Deployment Diagram.

User Device

This refers to the mobile device used by the user, which serves as the host environment for the execution of the BBP mobile application.

Firewall

Acting as the system's primary gatekeeper, this component monitors network data flow to detect anomalies and security threats. They are placed before the load balancers to secure inter-layer communications. The placement between the proxy services and the external systems is made in order to protect the system from possible threatening message due to external interactions.

Load Balancer

Responsible for traffic distribution, these components serve as the entry point for requests intended for the API gateway layer. In order to facilitating the granular management of traffic on the specific requirements of each service macro-category, a dedicated load balancing instance is utilized for each gateway type.

Container & API container

Every system's service is encapsulated in a container environment to enhance operational elasticity, allowing for the rapid instantiation of additional replicas as traffic volume increases.

2.4 Component Interfaces

Account API Gateway

Name	**Forward Create Account Request**
Signature	create_acc_req(name, surname, gender, birthdate, emailAddr, psw): void
Description	Forwards to the system the account creation request.
Parameters	• name: string, user's first name • surname: string, user's last name • gender: string, user's gender • birthdate: date, user's date of birth • emailAddr: string, user's email address • psw: string, user's chosen password
Returned	
Exceptions	The email is already in use (already_used_email_error)

Name	**Forward Account Creation Confirmation**
Signature	send_create_conf(regToken): void
Description	Forards to the system the confirmation token after the user clicks the email link.
Parameters	• regToken: string, the unique registration token generated during the account request phase
Returned	Returns a success status (acc_creation_succ)
Exceptions	The token has expired(link_expired_error)

Name	**Forward Login Request**
Signature	frw_login(emailAddr, psw): account_info
Description	Forward the account login request to the system.
Parameters	• emailAddr: string, the user's email address • psw: string, the user's password
Returned	Returns the user's account information (account_info)
Exceptions	• Account not found (account_not_found_error) • Invalid password (wrong_psw_error)

Name	**Forward Modify Attribute Request**
Signature	frw_modify_attr(userId, attrId, newVal): void
Description	Forward the modification of an account ttribute modification to the system.
Parameters	• userId: string, the unique identifier of the user • attrId: string, the identifier/name of the attribute to be modified • newVal: string, the new value to be assigned to the attribute
Returned	Returns a completion status (done)
Exceptions	

Name	**Forward Password Reset Request**
Signature	frw_psw_reset_req(emailAddr): void
Description	Forwards the initial request to reset the account password to the system.
Parameters	• emailAddr: string, the email address associated with the account
Returned	
Exceptions	Account not found (no_account_error)

Name	**Forward Reset Password Confirmation**
Signature	frw_reset_psw_conf(resetToken): form
Description	Forwads to the system the token from the email link to validate the reset request.
Parameters	• resetToken: string, the validation token received via email
Returned	Returns the password reset form (reset_psw_form)
Exceptions	The request is expired (link_expired_error)

Name	**Forward New Password**
Signature	frw_new_psw(newPsw, resetToken): void
Description	Forwards to the system the new password.
Parameters	• newPsw: string, the new password to set • resetToken: string, the validation token
Returned	Returns a completion status (done)
Exceptions	

Name	**Send account deletion request**
Signature	send_delete_acc(userId, emailAddr): void
Description	Forwards to the system the request to delete the account
Parameters	• userId: sting, unique identifier of the userId • email address (emailAddr): string, user email address
Returned	
Exceptions	

Name	**Confirm account deletion**
Signature	acc_del_conf(delToken): bool
Description	Forwards to the system the account deletion confirmation
Parameters	• deletion token (delToken): string, token embedded into the confirmation link to validate the account deletion request
Returned	Indicates that the deletion has been executed
Exceptions	

Name	**Forward Account Deletion Request**
Signature	frw_delete_acc_req(userId, emailAddr): void
Description	Forwards to the system the initial request to delete the account.
Parameters	• userId: string, the unique identifier of the user • emailAddr: string, the user's email address
Returned	
Exceptions	

Name	**Forward Account Deletion Confirmation**
Signature	frw_acc_del_conf(delToken): void
Description	Receives the confirmation token from the user (via email link) to finalize account deletion.
Parameters	• delToken: string, the token used to validate the deletion request
Returned	Returns a completion status (done)
Exceptions	The request has expired (link_expired_error)

Account creation Service

Name	**Send Account Creation confirmation**
Signature	send_acc_creat(name, surname, gender, birthdate, emailAddr, psw): void
Description	Sends to the user via email the link to confirm account creation, after executing the appropiate checks.
Parameters	• name: string, user's first name • surname: string, user's last name • gender: string, user's gender • birthdate: date, user's date of birth • emailAddr: string, user's email address • psw: string, user's chosen password
Returned	
Exceptions	The email is already registered for an account (already_used_email_error)

Name	**Account creation**
Signature	create_acc(regToken): void
Description	Creates the new account, orchestrating the intermediate phases.
Parameters	• regToken: string, the registration token to validate
Returned	Returns a success status (acc_creation_succ)
Exceptions	The token validation failed or timed out (link_expired_error)

Account Deletion Service

Name	**Delete Account Request**
Signature	delete_acc_req(userId, emailAddr): void
Description	Sends to the user via email the link to confirm account deletion.
Parameters	• userId: string, the unique identifier of the user • emailAddr: string, the user's email address
Returned	
Exceptions	

Name	**Account Deletion Confirmation**
Signature	acc_del_conf(delToken): void
Description	Validates the token and actually removes the account data.
Parameters	• delToken: string, the validation token
Returned	Returns a completion status (done)
Exceptions	The deletion request is expired (link_expired_error)

Authentication Service

Name	**Check Account Existence**
Signature	check_acc_exist(emailAddr): bool
Description	Checks if the provided email address is already associated with an account.
Parameters	• emailAddr: string, the email address to check
Returned	Returns wheter the porvvided email address is already used or not
Exceptions	

Name	**Create Account Credentials**
Signature	create_acc_cred(newUserId, emailAddr, psw): void
Description	Creates and stores the new account credentials.
Parameters	• newUserId: string, the unique ID assigned to the user • emailAddr: string, the user's email • psw: string, the user's password
Returned	
Exceptions	

Name	**Account Login**
Signature	login_acc(emailAddr, psw): status, userId
Description	Verifies the provided credentials.
Parameters	• emailAddr: string, the user's email address • psw: string, the user's password
Returned	Returns success status (correct_psw) and userId
Exceptions	• Account not found (account_not_found_error) • Invalid password (wrong_psw_error)

Name	**Reset Password Request**
Signature	psw_reset_req(emailAddr): void
Description	Verifies the account exists and in case sends the email with the reset link.
Parameters	• emailAddr: string, the user's email address
Returned	
Exceptions	Account not found (no_account_error)

Name	**Confirm Reset Password Request**
Signature	reset_psw_conf(resetToken): form
Description	Validates the reset token and returns the form to enter a new password.
Parameters	• resetToken: string, the token to validate
Returned	Returns the password reset form (rest_psw_form)
Exceptions	The link has expired (link_expired_error)

Name	**Update Password**
Signature	update_psw(newPsw, resetToken): void
Description	Updates the user's password with the new one.
Parameters	• newPsw: string, the new password • resetToken: string, the token for verification
Returned	Returns a completion status (done)
Exceptions	

Name	**Delete Account Credentials**
Signature	delete_acc_cred(userId): void
Description	Deletes the account credentials for the specified user.
Parameters	• userId: string, the unique identifier of the user to delete
Returned	Returns a completion status (done)
Exceptions	

Credentials DBMS

Name	**Query Account Existence**
Signature	query_acc_exists(emailAddr): bool
Description	Executes a database query to check if the email address exists in some accocunt record.
Parameters	• emailAddr: string, the email address to search for
Returned	Returns a boolean value (match)
Exceptions	

Name	**Query Insert Credentials**
Signature	query_insert_cred(newUserId, emailAddr, psw): void
Description	Store the new account credentials.
Parameters	• newUserId: string, the unique identifier for the record • emailAddr: string, the user's email address • psw: string, the user's password
Returned	Returns a completion status (done)
Exceptions	

Name	**Query Get Credentials**
Signature	query_get_cred(emailAddr): (string, string)
Description	Fetch the account credentials associated with the email address.
Parameters	• emailAddr: string, the email address to search for
Returned	Returns the stored password (psw) and user ID (userId)
Exceptions	No matching account found (no_match)

Name	**Query Update Password**
Signature	query_update_psw(userId, newPsw): void
Description	Updates the password value for the specified user.
Parameters	• userId: string, the unique identifier of the user • newPsw: string, the new password to store
Returned	Returns a completion status (done)
Exceptions	

Name	**Query Delete Account**
Signature	query_delete_acc(userId): void
Description	Delete the account record.
Parameters	• userId: string, the unique identifier of the user
Returned	Returns a completion status (done)
Exceptions	

Account Manager Service

Name	**Generate New UserID**
Signature	get_new_userId(): string
Description	Generates a new unique identifier for account records.
Parameters	
Returned	newUserId (string)
Exceptions	

Name	**Create Account Information**
Signature	create_acc_info(newUserId, name, surname, gender, birthdate): void
Description	Creates and stores the new account informations.
Parameters	• newUserId: string, the unique ID associated with the account • name: string, user's first name • surname: string, user's last name • gender: string, user's gender • birthdate: date, user's date of birth
Returned	
Exceptions	

Name	**Get Account Info**
Signature	get_acc_info(userId): account_info
Description	Retrieves the full profile information for the given account.
Parameters	• userId: string, the unique identifier of the user
Returned	Returns the account details (account_info)
Exceptions	

Name	**Modify Attribute**
Signature	modify_attr(userId, attrId, newVal): void
Description	Updated the attribute value with the new one for the specified user.
Parameters	• userId: string, the unique identifier of the user • attrId: string, the identifier/name of the attribute to be modified • newVal: string, the new value to be assigned
Returned	Returns a completion status (done)
Exceptions	

Name	**Delete Account**
Signature	delete_acc(userId): void
Description	Delete the account information associated to the given userId.
Parameters	• userId: string, the unique identifier of the user
Returned	Returns a completion status (done)
Exceptions	

Account DBMS

Name	**Query New ID**
Signature	query_new_id(): string
Description	Runs a database quesry to create a new account record with a new UserId.
Parameters	None
Returned	newUserId (string)
Exceptions	

Name	**Query Insert Account**
Signature	query_insert_acc(newUserId, name, surname, gender, birthdate): void
Description	Store the new account information.
Parameters	• newUserId: string, the unique identifier for the record • name: string, user's first name • surname: string, user's last name • gender: string, user's gender • birthdate: date, user's date of birth
Returned	Returns a completion status (done)
Exceptions	

Name	**Query Get Account Info**
Signature	query_get_info(userId): account_info
Description	Retrives user's profile data.
Parameters	• userId: string, the unique identifier to search for
Returned	Returns the account details (account_info)
Exceptions	

Name	**Query Update Attribute**
Signature	query_update_attr(userId, attrId, newVal): void
Description	Updates the attribute value for the given user.
Parameters	• userId: string, the unique identifier of the user • attrId: string, the database column/field to update • newVal: string, the value to persist
Returned	Returns a completion status (done)
Exceptions	

Name	**Query Delete Account**
Signature	query_delete_account(userId): status
Description	Delete the account information record for the given user.
Parameters	• userId: string, the unique identifier to search for
Returned	Returns a completion status (done)
Exceptions	

Email Proxy Service

Name	**Send Registration Link**
Signature	send_reg_link(emailAddr, link): void
Description	Send the email containing the account creation confirmation link.
Parameters	• emailAddr: string, the recipient's email • link: string, the confirmation URL containing the token
Returned	
Exceptions	

Name	**Send Reset Password Link**
Signature	send_reset_psw_link(emailAddr): void
Description	Send the email containing the password reset confirmation link.
Parameters	• emailAddr: string, the recipient's email
Returned	
Exceptions	

Name	**Send Deletion Link**
Signature	send_acc_del_link(emailAddr, delLink): void
Description	Send the email containing the account deletion confirmation link
Parameters	• emailAddr: string, the recipient's email • delLink: string, the account deletion URL containing the token
Returned	
Exceptions	

Trip API Gateweay

Name	**Forward Path Request**
Signature	frw_get_paths(sPoint, ePoint, filters): suggested_paths
Description	Forwards to the system the request for path givens the starting and ending point with the specifed filters.
Parameters	• sPoint: location/string, the starting point of the trip • ePoint: location/string, the destination point of the trip • filters: List¡string¿, optional filters to customize the path search (empty if no filter applied)
Returned	Returns the calculated suggested paths (suggested_paths)
Exceptions	

Name	**Forward Save Activity**
Signature	frw_save_act(userId, pathId, date, startTime, endTime, score, List¡detIssue¿:): void
Description	Forwards to the system the completed activity to be saved.
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path taken • date: date, the date the activity occurred • startTime: time, the start time of the activity • endTime: time, the end time of the activity • score: integer, the rating given by the user • List¡detIssue¿: List¡Issue¿(optional), a list of issues automatically detected during the monitored activity
Returned	Returns a completion status (done)
Exceptions	

Name	**Forward Issue Report**
Signature	frw_publish_issue(userId, pathId, timestamp, pos, type, descr): void
Description	Forward to the system the request to publish issue.
Parameters	• userId: string, the unique identifier of the user reporting the issue • pathId: string, the identifier of the path where the issue was found • timestamp: datetime, the time the issue was observed • pos: string/coords, the specific location of the issue • type: string, the category of the issue (e.g., obstacle, damage) • descr: string, a text description provided by the user
Returned	Returns a completion status (done)
Exceptions	

Name	**Forward Fixup Report**
Signature	frw_publish_fixup(issueId): void
Description	Forward to the system the request to publish fixup.
Parameters	• issueId: string, the unique identifier of the issue to be updated
Returned	Returns a completion status (done)
Exceptions	The specified issue was not found (issue_not_found_error)

Name	**Forward History Request**
Signature	frw_get_act_hist(userId, filters): List¡Activity¿
Description	Forward to the system the request to get the activity history.
Parameters	• userId: string, the unique identifier of the user • filters: object/list, optional search criteria (empty if no filter applied)
Returned	Returns the list of historical activities (done)
Exceptions	

Name	**Forward Activity Deletion Request**
Signature	frw_delete_act_req(actId): (conf_form, actDelToken)
Description	Forwards to the system the request to delete the specified activity.
Parameters	• actId: string, the unique identifier of the activity
Returned	Returns the confirmation form (conf_form) and deletion token (actDelToken)
Exceptions	

Name	**Forward Activity Deletion Confirmation**
Signature	frw_send_del_conf(actId, actDelToken): void
Description	Forward to the system the request to delete the selected activity.
Parameters	• actId: string, the unique identifier of the activity • actDelToken: string, the security token received in the previous step
Returned	Returns a completion status (done)
Exceptions	

Trip Planner Service

Name	**Get Paths**
Signature	get_paths(sPoint, ePoint, , filters): suggested_paths
Description	Elaborate the path request, retriving all information about paths and their conditions (issues and score).
Parameters	• sPoint: location/string, the starting point • ePoint: location/string, the destination point • filters: List¡string¿, optional filters to customize the path search (empty if no filter applied)
Returned	Returns a list of suggested paths (suggested_paths)
Exceptions	

PAth Inventory Service

Name	**Retrieve Paths**
Signature	retrive_paths(sPoint, ePoint, filters): List¡pathInfo¿
Description	Returns a list of paths and related information matching the starting and destination point given by the user.
Parameters	• sPoint: location/string, the starting point of the route • ePoint: location/string, the destination point • filters: List¡string¿, optional filters to customize the path search (empty if no filter applied)
Returned	Returns a list of path objects (List¡pathInfo¿)
Exceptions	

Name	**Get Path Location**
Signature	get_path_loc(pathId): pathLocation
Description	Retrieves the location for the given path.
Parameters	• pathId: string, the unique identifier of the path
Returned	Returns the path location (pathLocation)
Exceptions	

Path DBMS

Name	**Query Get Paths**
Signature	query_paths(sPoint, ePoint, filters): List¡pathInfo¿
Description	Fetch the path connecting the specified start and end points applying the specified filters.
Parameters	• sPoint: location/string, the starting point • ePoint: location/string, the destination point • filters: List¡string¿, optional filters to customize the path search (empty if no filter applied)
Returned	Returns a list of paths if found, or a no_match status
Exceptions	

Name	**Query Insert Path**
Signature	query_insert_path(path): void
Description	Store the given path.
Parameters	• path: object/string, the detailed path information to store
Returned	Returns a completion status (done)
Exceptions	

Name	**Query Get Location**
Signature	query_get_loc(pathId): pathLocation
Description	Retrives the location for the given path.
Parameters	• pathId: string, the unique identifier of the path
Returned	Returns the path location (pathLocation)
Exceptions	

Mapping Proxy Service

Name	**Create Path**
Signature	create_path(sPoint, ePoint): path
Description	Creates a path from the specified starting and destination points.
Parameters	• sPoint: location/string, the starting point • ePoint: location/string, the destination point
Returned	Returns the newly calculated path
Exceptions	

Activity Handler Service

Name	**Forward Save Activity**
Signature	frw_save_act(userId, pathId, date, startTime, endTime, score, List¡detIssue¿): void
Description	Orchestrates the process of saving the activity, verifying weather conditions, and updating scores.
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path taken during the activity • date: date, the date the activity occurred • startTime: time, the start time of the activity • endTime: time, the end time of the activity • score: Score, the rating given by the user • • List¡detIssue¿: List¡Issue¿(optional), a list of issues automatically detected during the monitored activity
Returned	Returns a completion status (done)
Exceptions	

Activity DBMS

Name	**Query Insert Activity**
Signature	query_insert_act(userId, pathId, activityInfo): string
Description	Store the given activity.
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path • activityInfo: object, structure containing date, start time, end time, and weather info (if available)
Returned	Returns the generated activity ID (actId)
Exceptions	

Name	**Query Get Activity History**
Signature	query_get_act_hist(userId, filters): list¡activity¿
Description	Fetch the activity history for the given account.
Parameters	• userId: string, the unique identifier of the user • filters: object/list, database query constraints
Returned	Returns the fetched activity records (done)
Exceptions	

Name	**Query Delete Activity**
Signature	query_del_act(actId): void
Description	Delete the activity record.
Parameters	• actId: string, the unique identifier of the activity
Returned	Returns a completion status (done)
Exceptions	

Weather Proxy Service

Name	**Check Weather Availability**
Signature	check_weather_av(): status
Description	Checks the availability status of the external weather service provider in order to avoid useless operation.
Parameters	None
Returned	Returns the availability status (status)
Exceptions	

Name	**Get Weather Information**
Signature	get_weat_info(pathLocation): weatherInfo
Description	Retrieves specific weather data for the activity location from the *External Weather System*.
Parameters	• pathLocation: string, the coordinates or location data of the path
Returned	Returns the weather information set (weatherInfo)
Exceptions	

Actvity History Service

Name	**Create Activity**
Signature	create_act(userId, pathId, activityInfo): string
Description	Cretate and store the new activity.
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path taken • activityInfo: object, structure containing date, start time, end time, and weather info (if available)
Returned	Returns the generated activity ID (actId)
Exceptions	

Name	**Get Activity History**
Signature	get_act_hist(userId, filters): list¡activity¿
Description	Retrives the activity history for the specified user.
Parameters	• userId: string, the unique identifier of the user • filters: object/list, specific criteria to filter the results
Returned	Returns the list of activities (done)
Exceptions	

Name	**Process Deletion Request**
Signature	delete_act_req(actId): (conf_form, actDelToken)
Description	Generates the confirmation form and a temporary deletion token for the request.
Parameters	• actId: string, the unique identifier of the activity
Returned	Returns the confirmation form and token
Exceptions	

Name	**Delete Activtiy**
Signature	delete_act(actId, actDelToken): void
Description	Delete the activity.
Parameters	• actId: string, the unique identifier of the activity • actDelToken: string, the security token for validation
Returned	Returns a completion status (done)
Exceptions	

Public Info Manager Service

Name	**Get Path Scores**
Signature	get_scores(list¡pathId¿): List¡scores¿
Description	Retrieves rating scores for a given list of paths.
Parameters	• list¡pathId¿: List¡string¿, identifiers for the paths to be scored
Returned	Returns a list of scores to the paths
Exceptions	

Name	**Get Path Issues**
Signature	get_issues(List¡topPathId¿): List¡issues¿
Description	Retrieves reported issues for a given list of paths.
Parameters	• List¡topPathId¿: List¡string¿, identifiers for the top paths
Returned	Returns a list of issues found for the paths
Exceptions	

Name	**Create Score**
Signature	create_score(userId, pathId, actId, score): void
Description	Creates and stores the given score witth its information.
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path being scored • actId: string, the identifier activtiy from which the score is derived • score: float, the score to be saved
Returned	Returns a completion status (done)
Exceptions	

Name	**Query Insert Issue**
Signature	query_insert_issue(userId, pathId, issueInfo): void
Description	Store the given issue
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path • issueInfo: object, structure containing timestamp, position (pos), type, and description (desc)
Returned	Returns a completion status (done)
Exceptions	

Name	**Publish Issue**
Signature	publish_issue(userId, pathId, timestamp, pos, type, descr): void
Description	Add to the system issue archive the given issue data.
Parameters	• userId: string, the user identifier • pathId: string, the path identifier • timestamp: datetime, observation time • pos: string/coords, location of the issue • type: string, issue category • descr: string, user description
Returned	Returns a completion status (done)
Exceptions	

Name	**Publish Fixup**
Signature	publish_fixup(issueId): void
Description	Add to the system issue archive the given fixup after running the appropriate checks.
Parameters	• issueId: string, the unique identifier of the issue
Returned	Returns a completion status (done)
Exceptions	The specified issue was not found (issue_not_found_error)

subsubsection*Public Info DBMS

Name	**Query Top Scores**
Signature	query_top_scores(list¡pathId¿): List¡scores¿
Description	Retrive the scores for the given paths.
Parameters	• list¡pathId¿: List¡string¿, identifiers for the paths
Returned	Returns the fetched scores
Exceptions	

Name	**Query Issues**
Signature	query_issues(List¡topPathId¿): List¡issues¿
Description	Retrive issues for the given paths.
Parameters	• List¡topPathId¿: List¡string¿, identifiers for the paths
Returned	Returns the fetched issues
Exceptions	

Name	**Query Insert Score**
Signature	query_insert_score(userId, pathId, actId, score): void
Description	Store the new score.
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path • actId: string, the identifier of the activity • score: float, the score value
Returned	Returns a completion status (done)
Exceptions	

Name	**Create Issue Set**
Signature	create_issue_set(userId, pathId, List¡detIssue¿): void
Description	Create ad store the list of issues given.
Parameters	• userId: string, the unique identifier of the user • pathId: string, the identifier of the path • List¡detIssue¿: List¡Issue¿, a collection of issues containing timestamp, position, type, and description
Returned	Returns a completion status (done)
Exceptions	

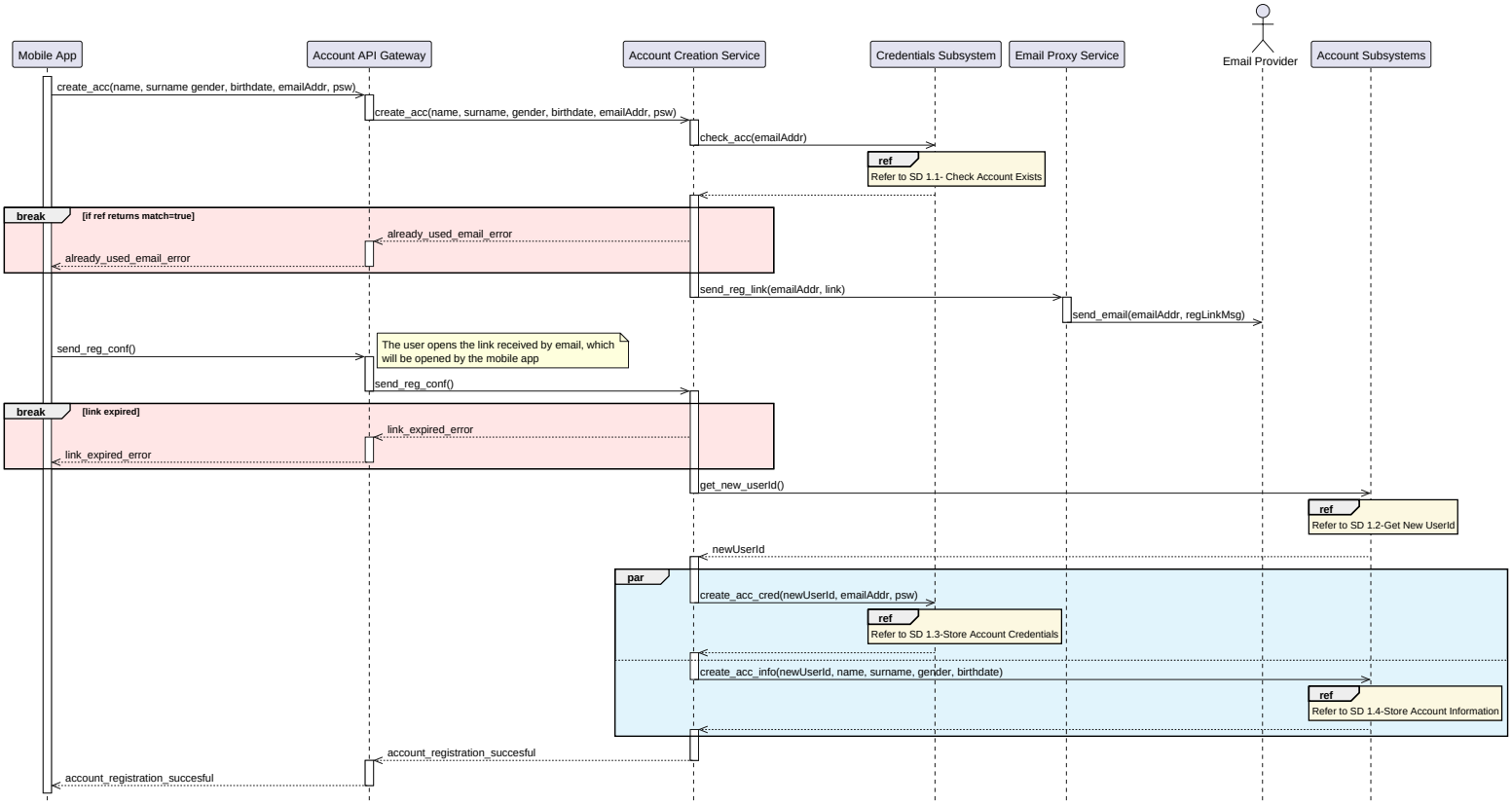
Name	**Query Insert Issue**
Signature	query_insert_issue(userId, pathId, timestamp, pos, type, descr): void
Description	Stores the given issue.
Parameters	• userId: string, the user identifier • pathId: string, the path identifier • timestamp: datetime, observation time • pos: string/coords, location of the issue • type: string, issue category • descr: string, user description
Returned	Returns a completion status (done)
Exceptions	

Name	**Query Check Issue Exists**
Signature	query_check_issue_exists(issueId): status
Description	Check whether the selected issue exists or not.
Parameters	• issueId: string, the unique issue identifier to search for
Returned	Returns the match status (match/noMatch)
Exceptions	No matching issue found (noMatch)

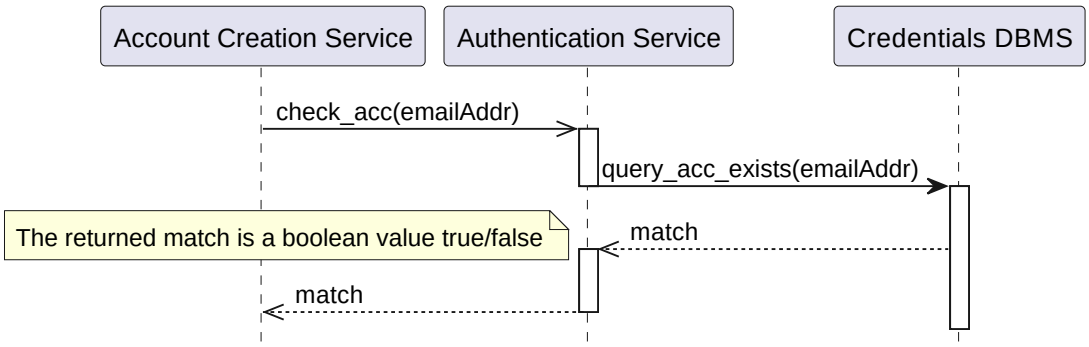
Name	**Query Update Issue**
Signature	query_update_issue(issueId): void
Description	Update the issue status.
Parameters	• issueId: string, the unique identifier of the issue
Returned	Returns a completion status (done)
Exceptions	

2.5 Runtime View

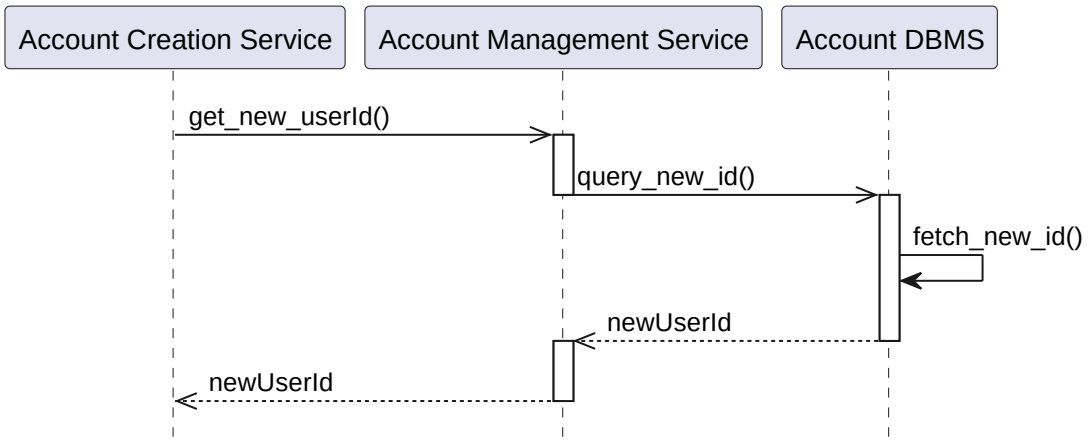
[SD1] Account creation



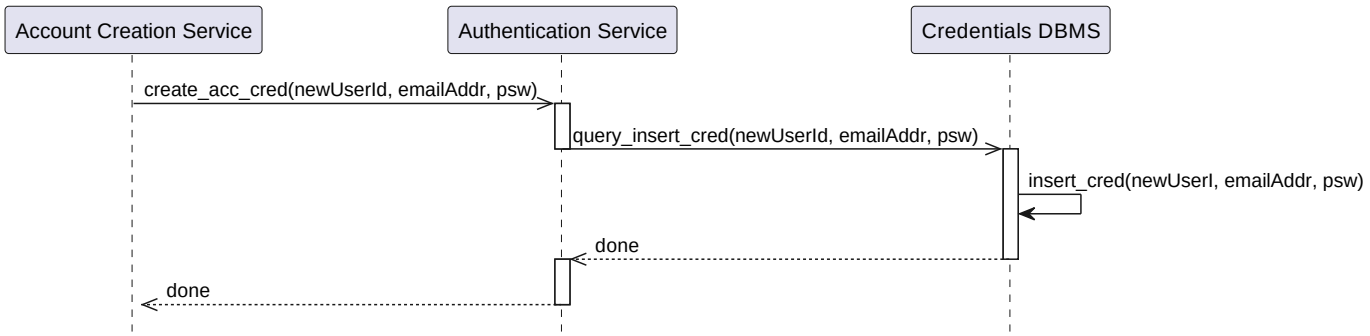
[SD1.1] Check Account Existence



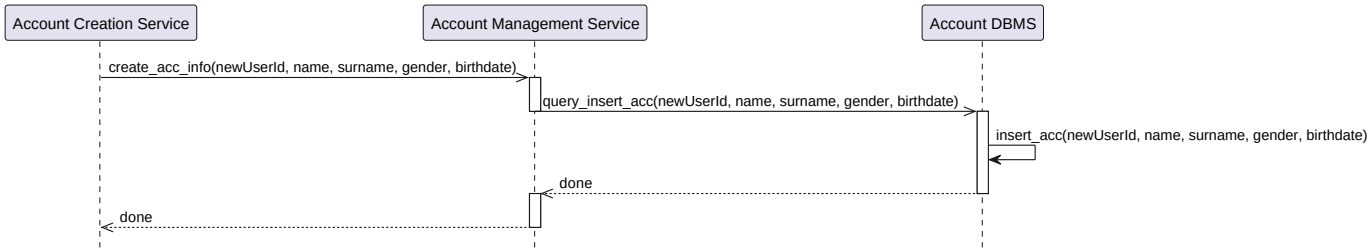
[SD1.2] Get New UserId



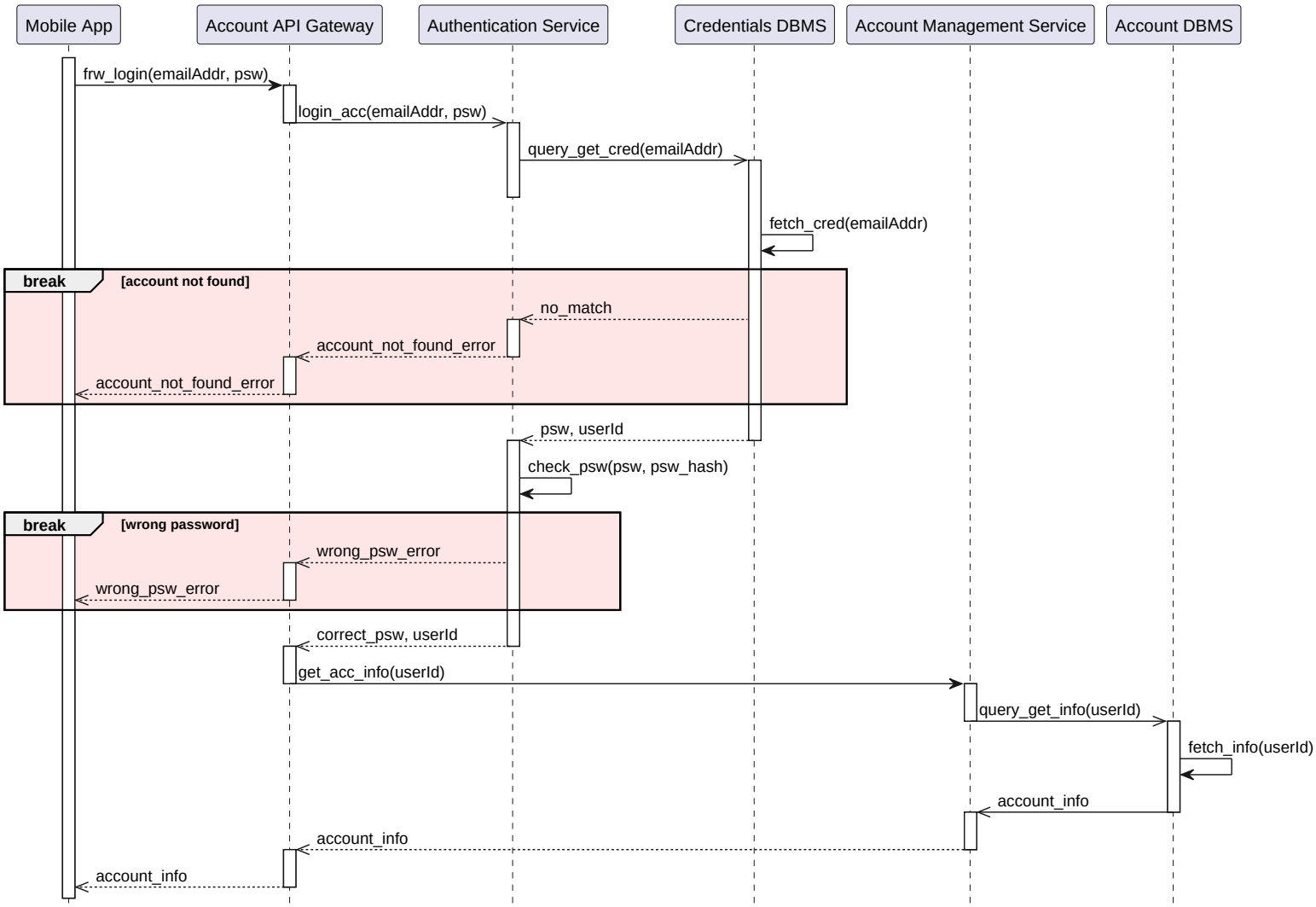
[SD1.3] Store Account Credentials



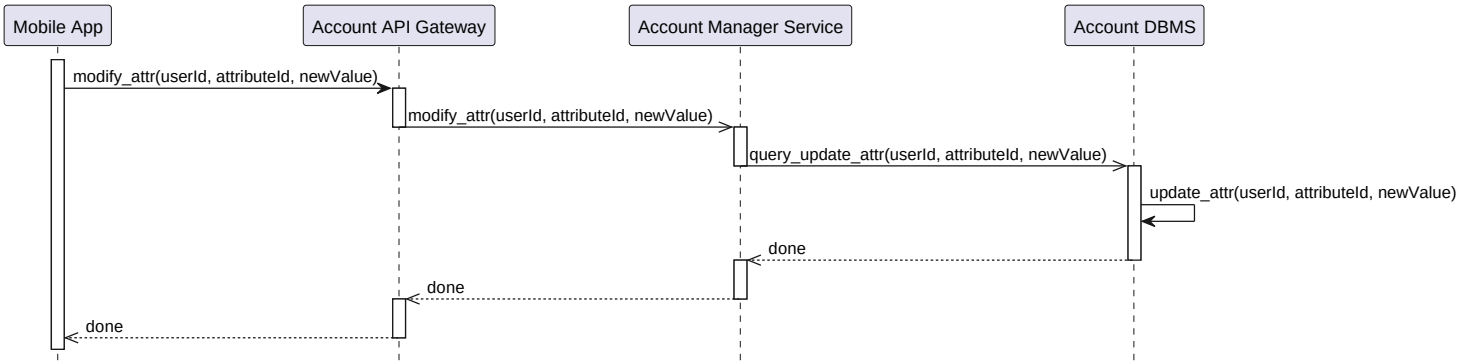
[SD1.4] Store Account Informations



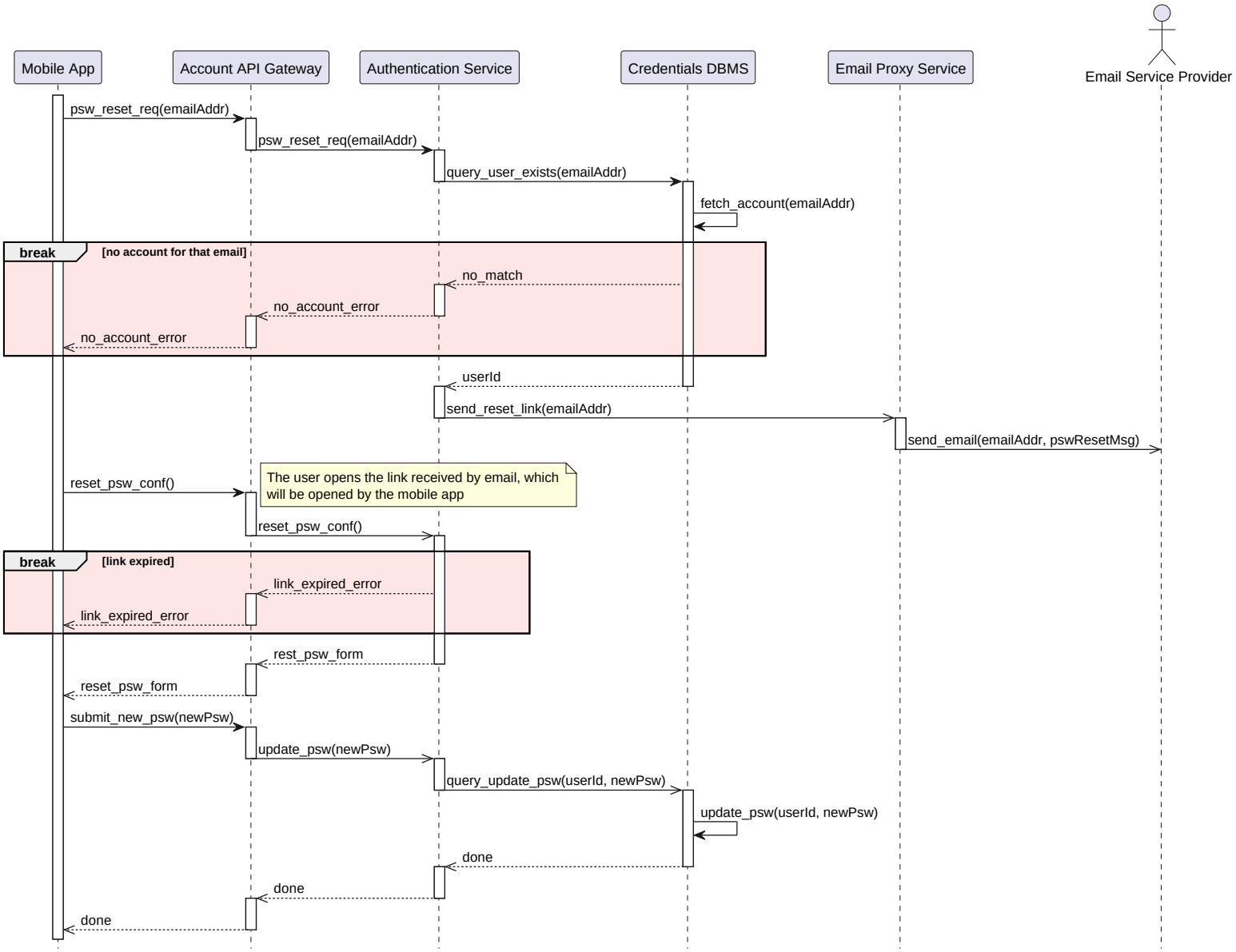
[SD2] Account login



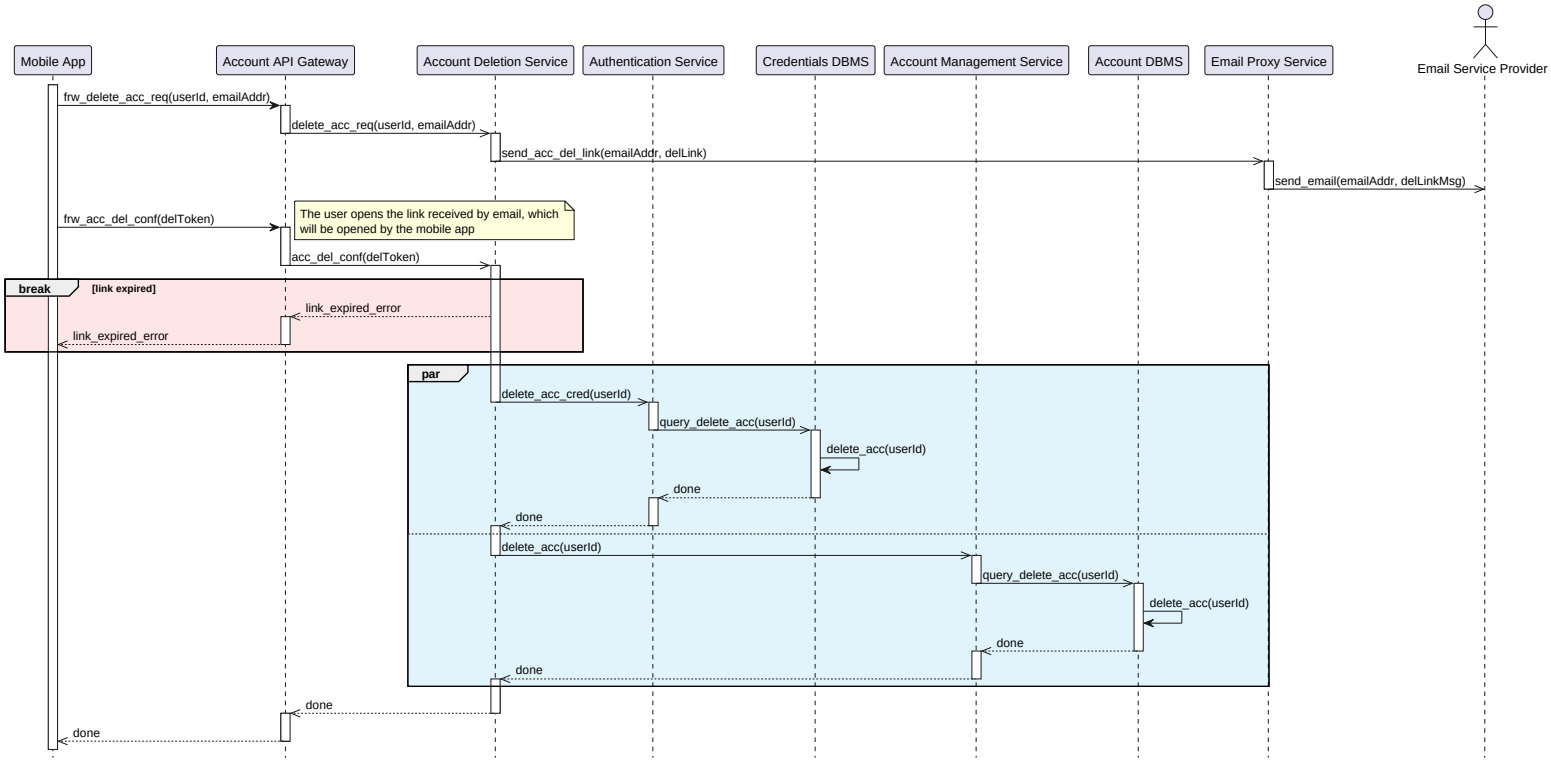
[SD3] Account update



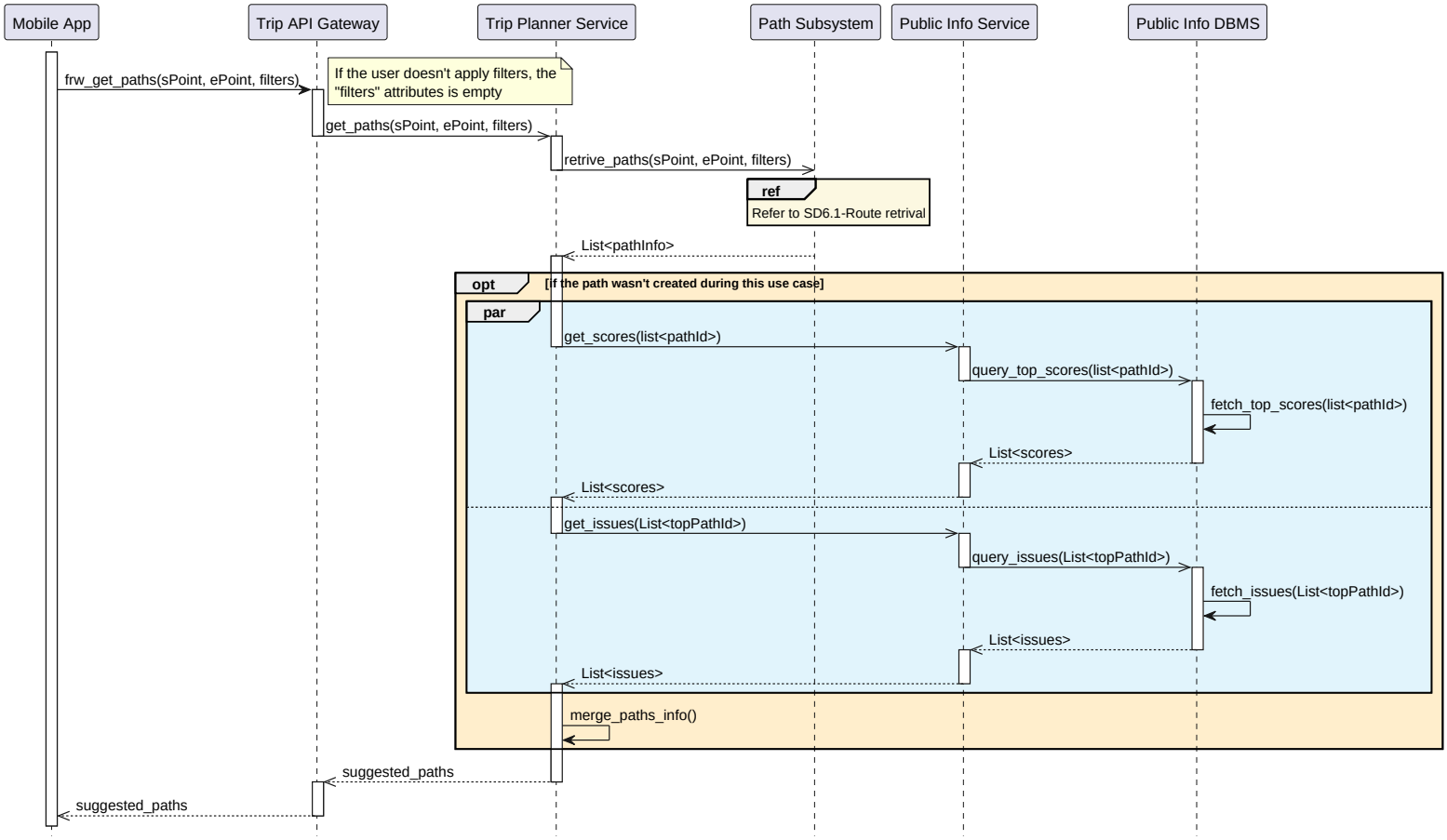
[SD4] Account Password Reset



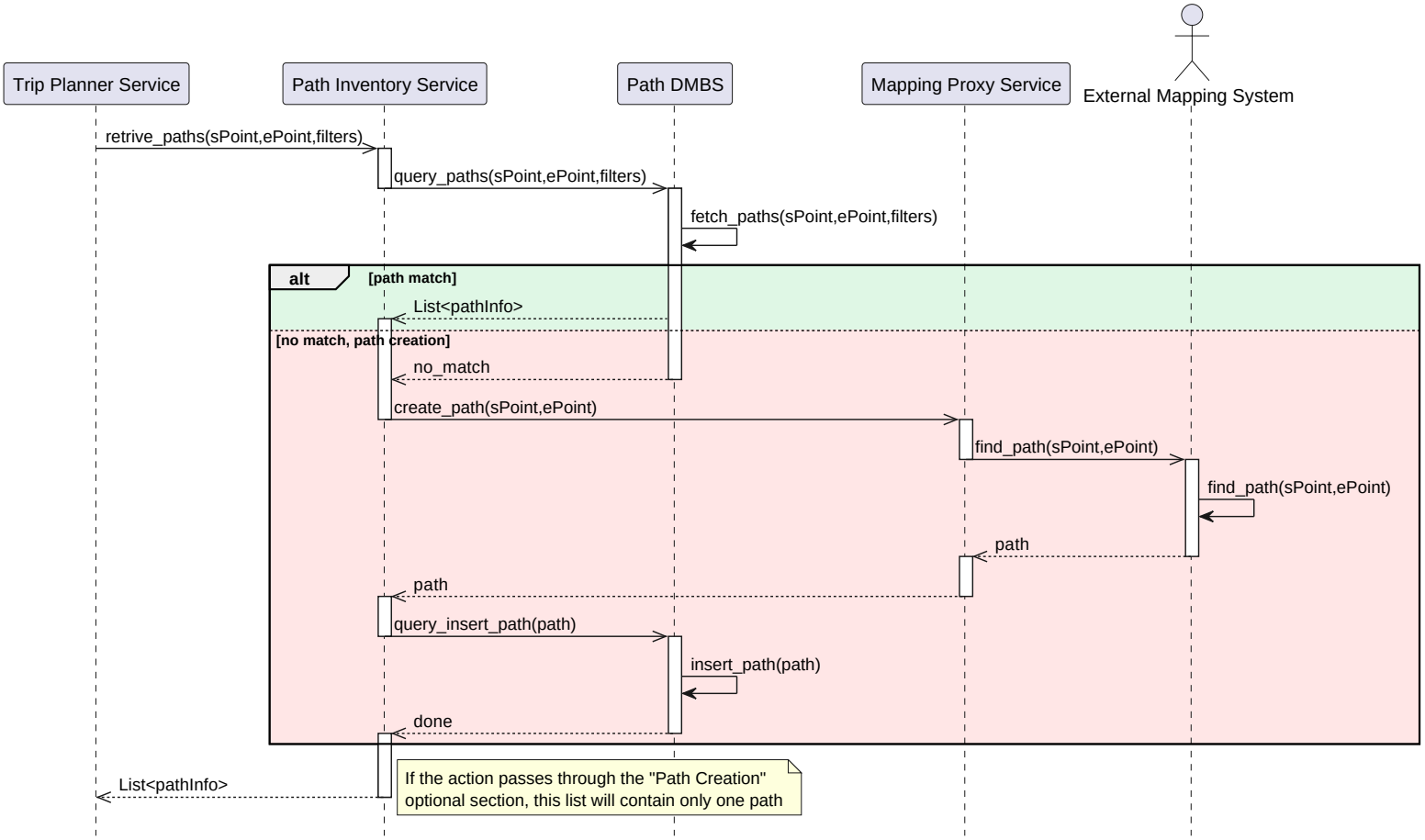
[SD5] Account Deletion



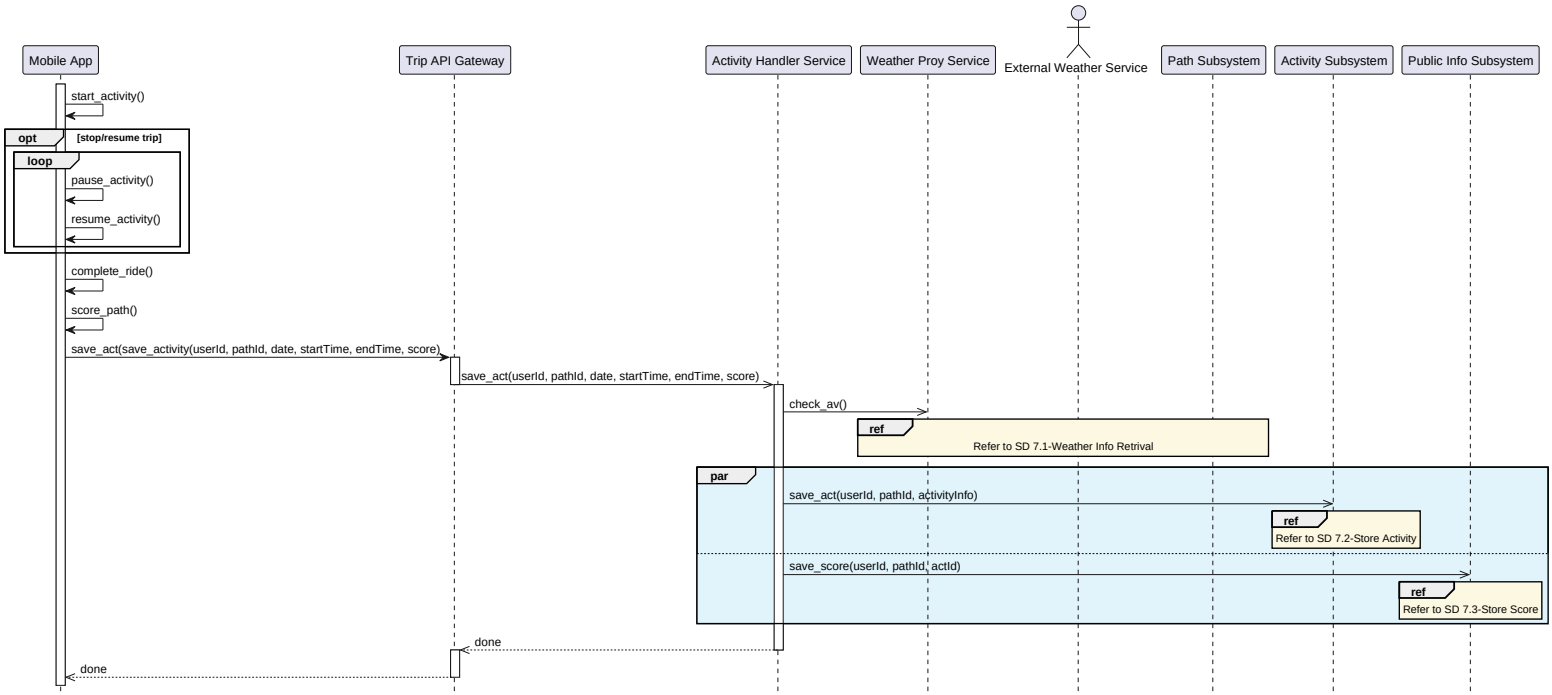
[SD6] Route Planning



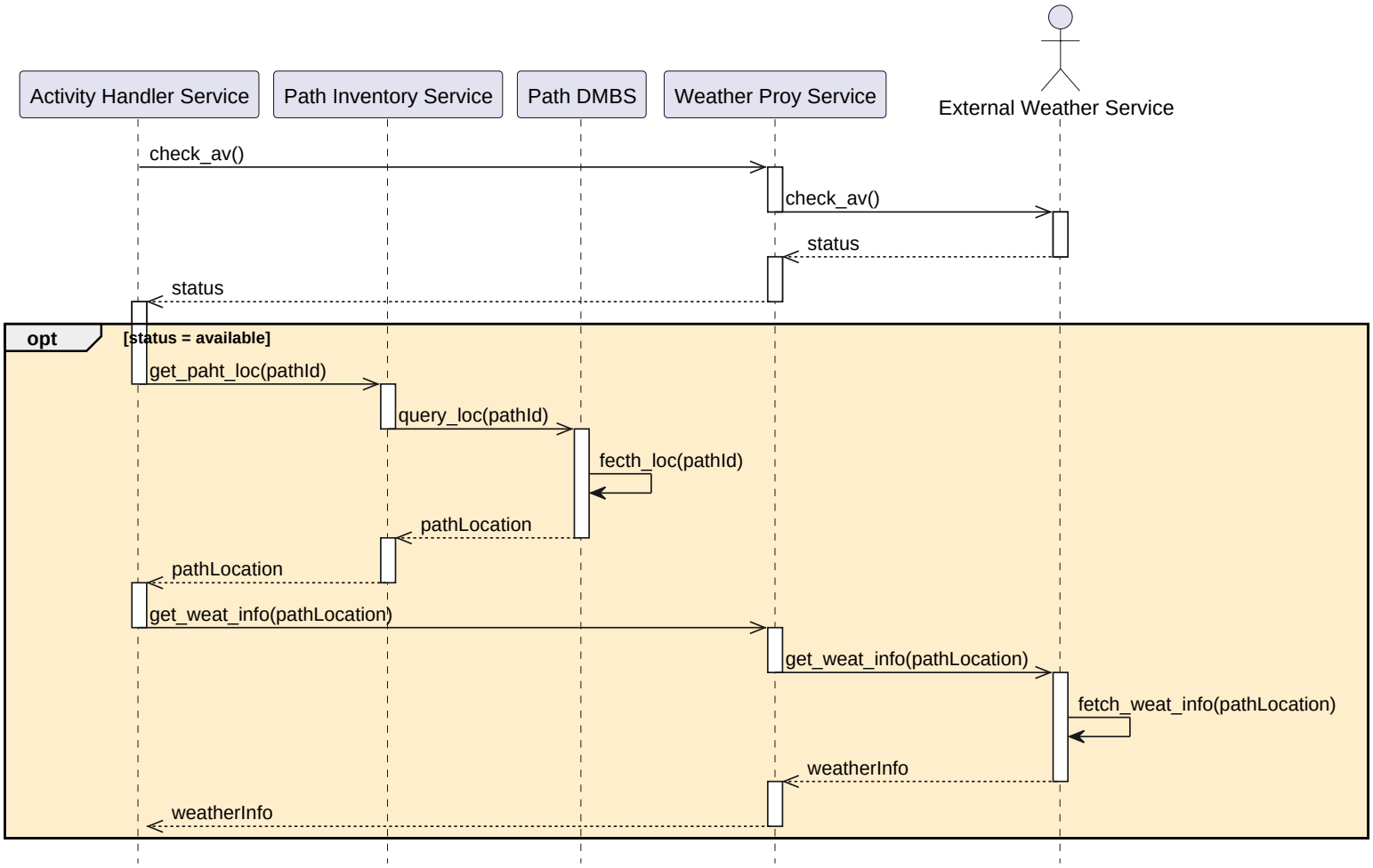
[SD6.1] Route Retrieval



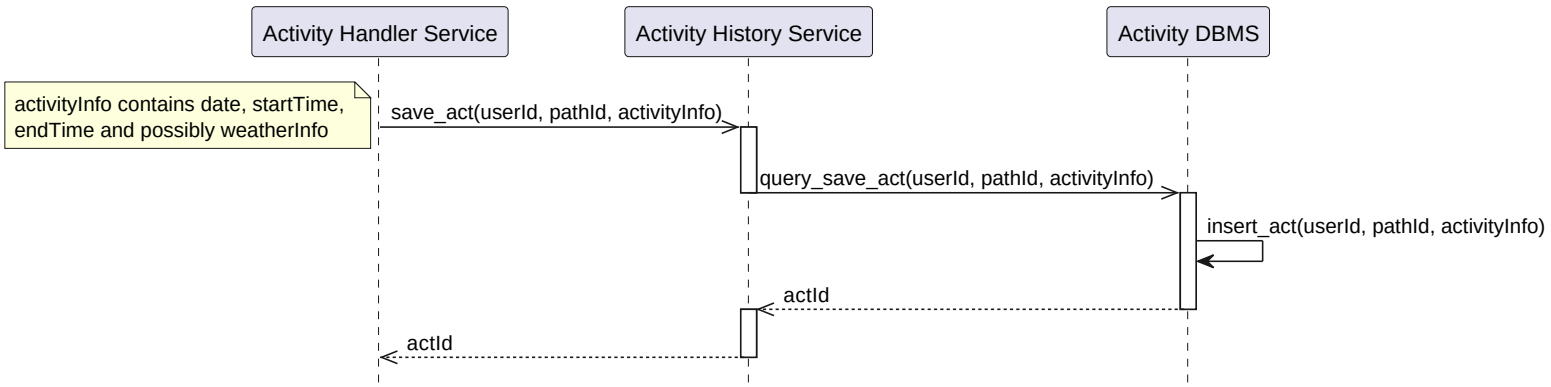
[SD7] Registered User Activity Lifecycle



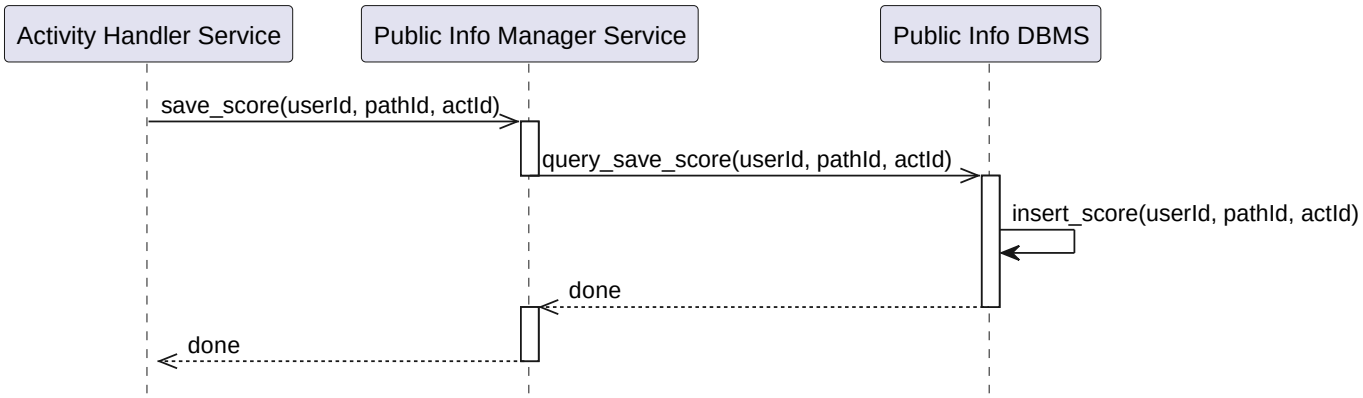
[SD7.1] Weather Information Retrieval



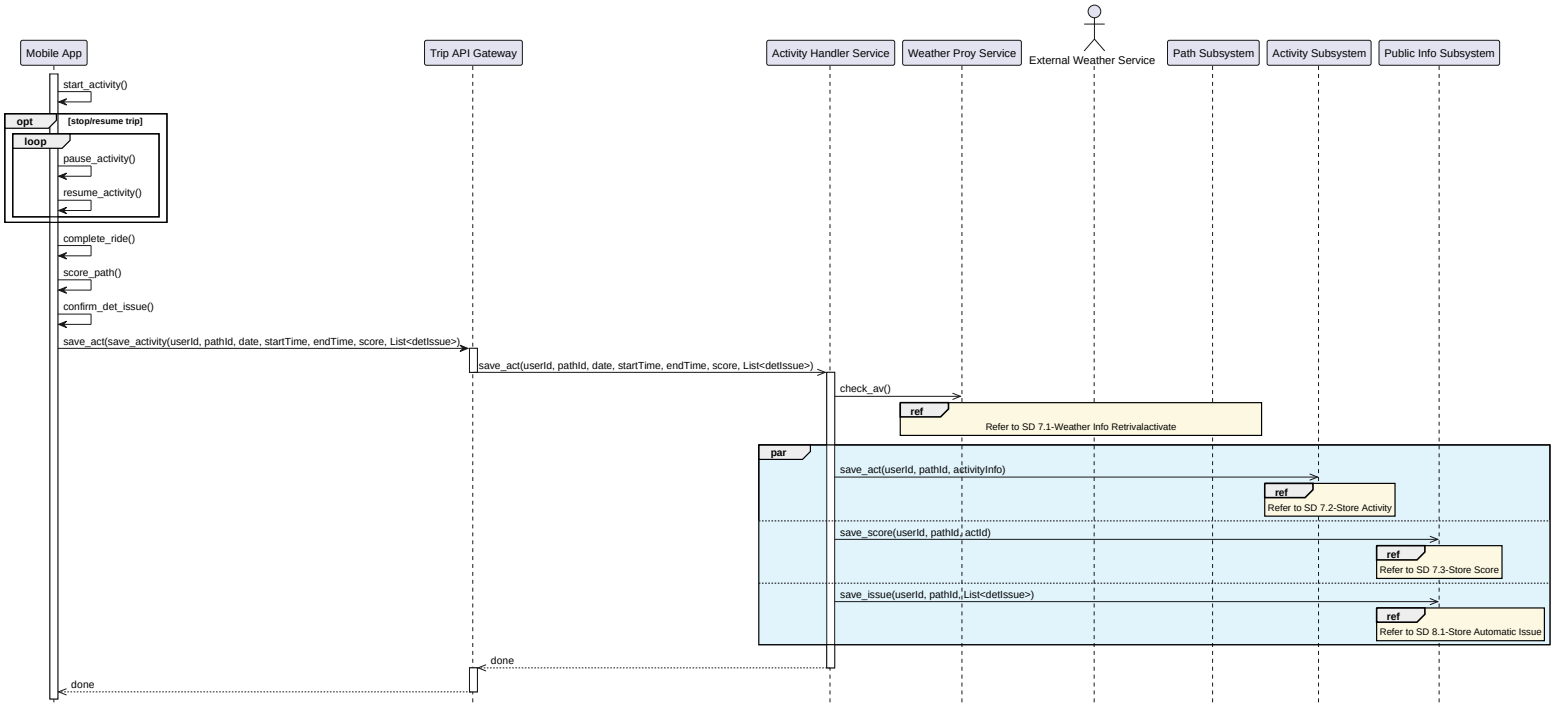
[SD7.2] Store Activity



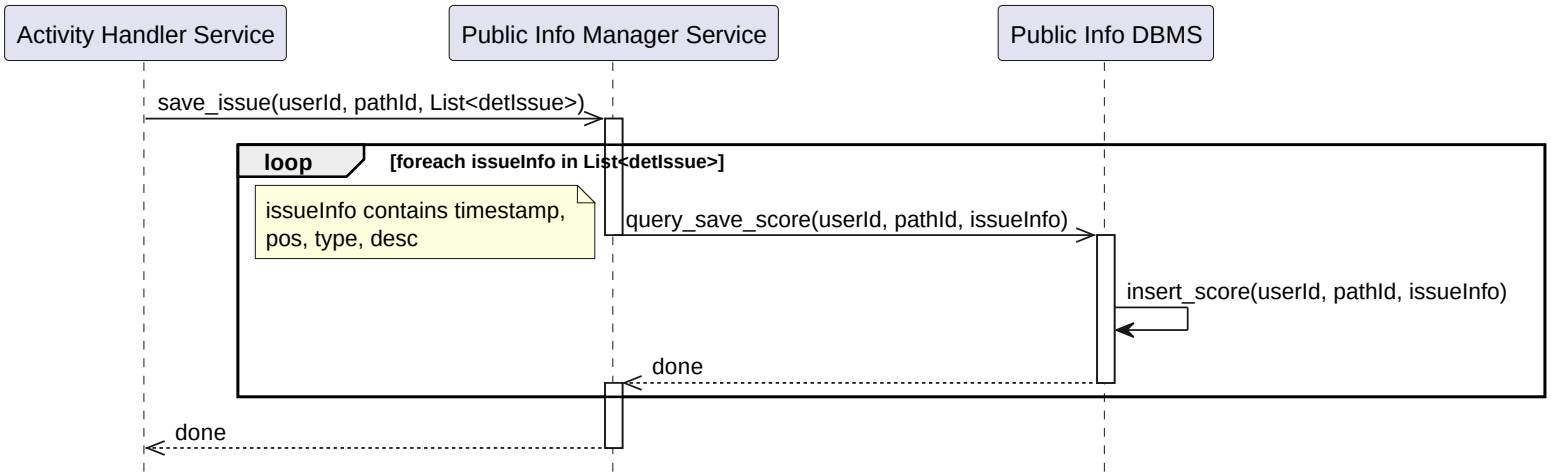
[SD7.3] Store Score



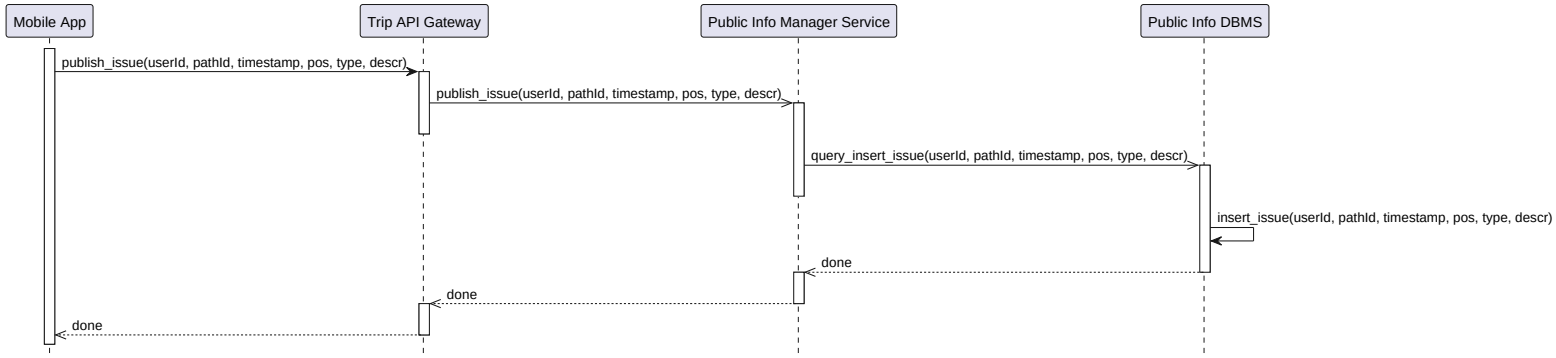
[SD8] Monitored Activity Lifecycle



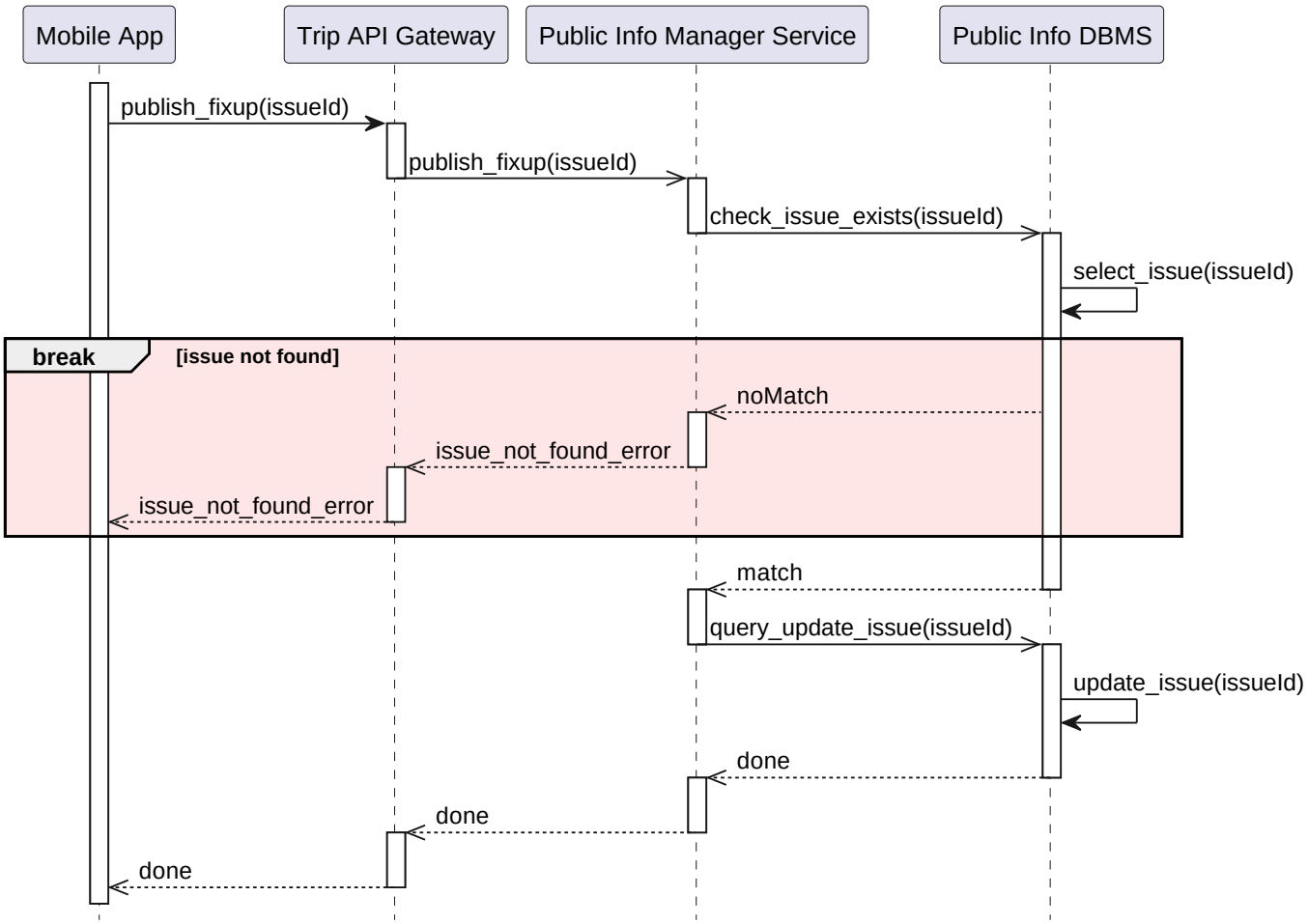
[SD8.1] Store Automatically Detected Activity



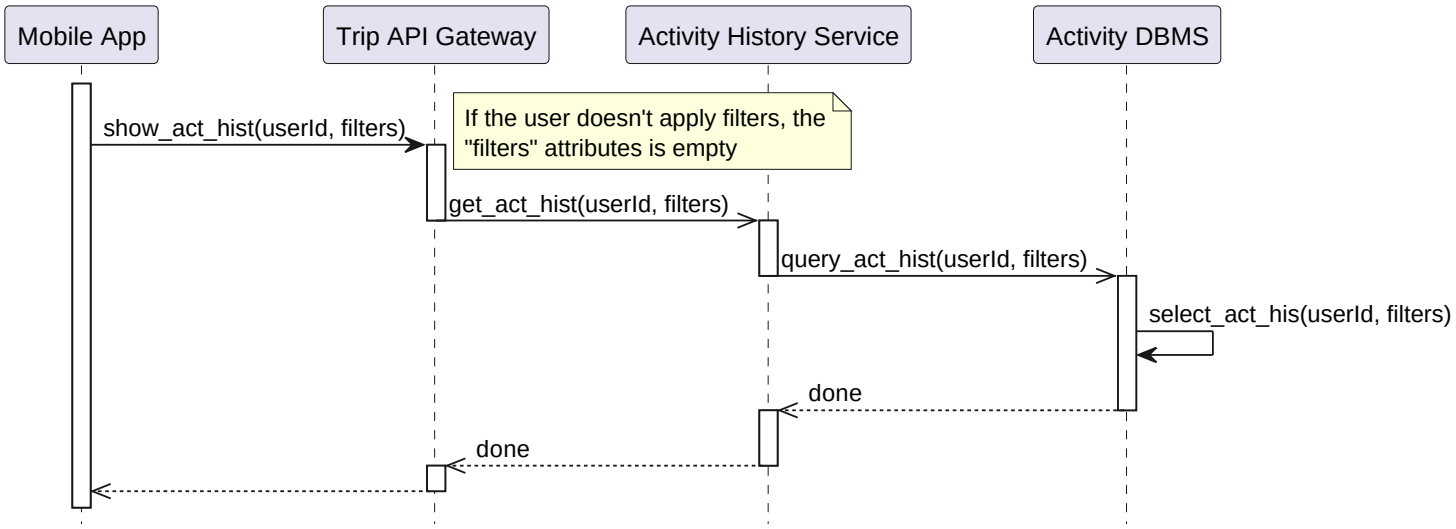
[SD9] Manual Issue Report



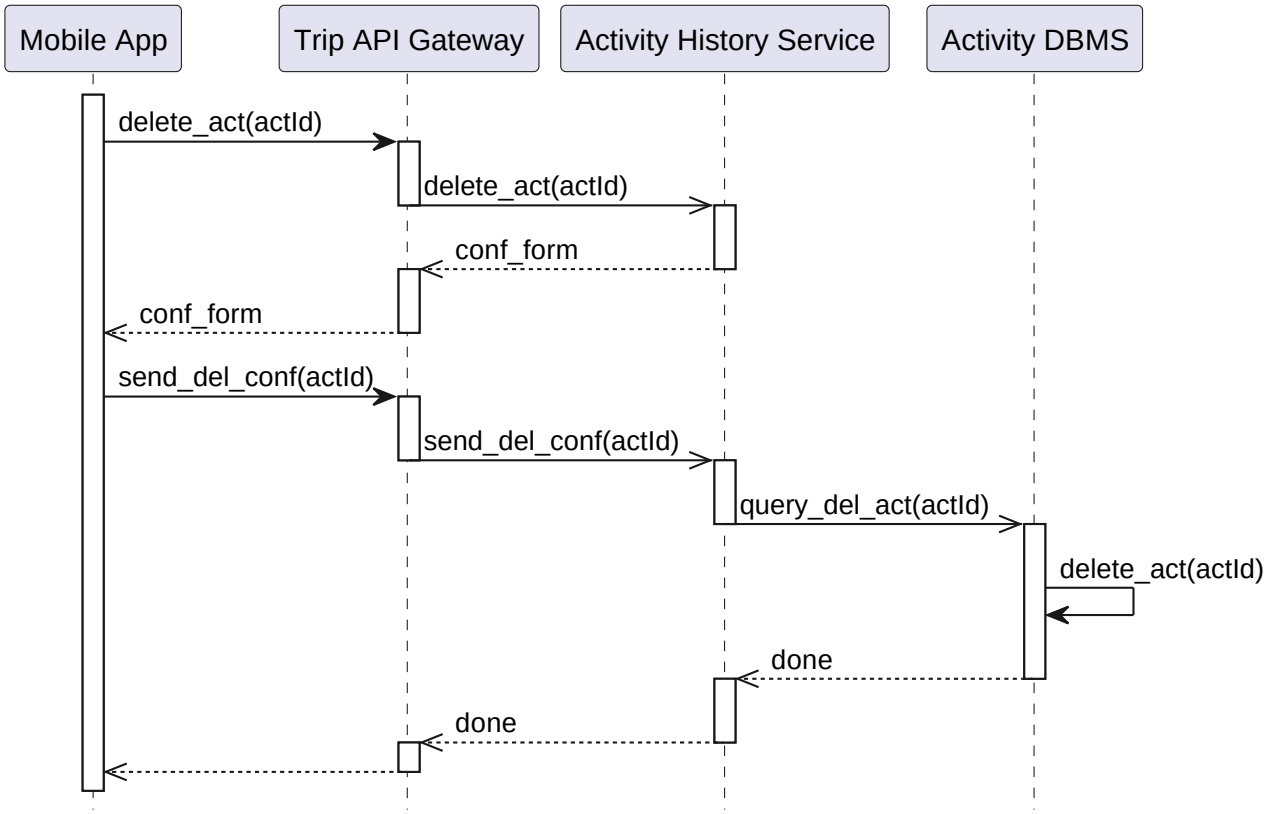
[SD10] Fixup Report



[SD11] Activity History Consultation



[SD12] Delete Activity Record



2.6 Selected architectural styles and patterns

Microservice architectural style

The decision to adopt a Microservices architectural style was primarily driven by the inherent requirement for horizontal scalability. Unlike monolithic structures, this distributed approach focusses on splitting the whole system into small pieces, each one specialized on dealing with a specific task, allowing the system to scale specific components independently, enabling the infrastructure to adapt dynamically to fluctuating workloads in real-time. Furthermore, the fine-grained decomposition of services and the intelligent duplication of critical resources significantly enhance the system's overall resilience.

4-layer architecture

To ensure a modular and scalable environment, we opted for a 4-tier architectural approach. This design allows us to separate system functionalities based on their logical domain while simultaneously defining how these components are distributed across physical hardware and network boundaries. We have identified the following layers:

- **Presentation Layer**: This layer serves as the primary interface between the user and the system. It is responsible for rendering the UI components and facilitating user interaction, allowing the user to interact easily with the system.
- **Front-end Application Layer**: This layer encapsulates the client-side business logic executed within the mobile application. It handles tasks like trip management, high-frequency real-time data sampling and on-device pattern recognition for identifying issues during recorded trips. To handle the issue-recognition task, we've chose to use pre-trained lightweight machine learning models.
- **Back-end Application Layer**: This layer comprises the core server-side logic and the various microservices offered by the platform. It is architected as a set of distributed server nodes, where each service is decoupled to address specific functional domains, ensuring high availability and fault isolation.
- **Data Layer**: Dedicated to the persistence and orchestration of system information, this layer consists of a distributed database cluster. It manages data integrity and retrieval across all nodes, implementing different storage strategies based on the specific type of data is handling.

API Gateway

To fortify system security and streamline the interaction between the client-side and server-side logic, we implemented the API Gateway pattern. By

positioning the gateway as the sole entry point for all client requests, we effectively decouple the front-end from the internal microservices architecture. This abstraction layer enhances the system's security by encapsulating the internal service topology, thereby shielding the underlying services from direct external exposure. To mitigate the risk of a Single Point of Failure, the gateway is deployed across multiple instances, ensuring high availability and continuous service. Beyond acting as a reverse proxy, these gateways also perform the tasks of server-side load balancing and service discovery (see Section 2.7). Furthermore, we adopted for a differentiated gateway strategy by dividing the entry points into two specialized categories: the Account API Gateway and the Trip API Gateway. This granular approach allows each gateway to scale independently according to the specific needs of the underlying services (e.g. an exceptional number of trip requests won't affect the login service)

Proxy

To prevent internal microservices from directly interfacing with external systems, we implemented a Proxy Service pattern. This pattern establishes a secure communication intermediary, creating a strictly controlled boundary between our internal infrastructure and third-party services. By routing all external traffic through these proxies, we implement a robust security perimeter where initial validation and security audits can be performed without exposing the internal services. Beyond security, this approach facilitates a more decoupled architecture. By abstracting the external service's specific API, our internal business logic remains indipendent from the third-party implementation.

Circuit breaker

To safeguard the system against performance degradation caused by a restricted group of services, we implemented the Circuit Breaker pattern. This is particularly useful for services that interact with non-critical dependencies, such as the API Gateways, or services interfaced with third-party systems, where their reliability cannot be guaranteed. By utilizing this pattern, the calling service actively monitors the health and response times of the invoked service. If a particular service begins to fail or exceeds predefined latency thresholds, the circuit "closes," immediately redirecting subsequent calls to a fallback mechanism. This fail-fast approach prevents the calling service from hanging indefinitely, thereby avoiding a ripple effect that could otherwise compromise the entire application's stability.

2.7 Other design decisions

Server-side service discovery

To enable the API Gateways to route traffic effectively, we integrated a Service Discovery mechanism on them. When a new service instance is initialized, it automatically registers itself. To ensure consistency across all nodes in the discovery service, this registration data is propagated throughout the entire discovery system using a Gossiping Protocol, ensuring all gateway nodes are updated eventually. To maintain updated the services' status, each service instance transmits a periodic heartbeat to the discovery service. The API Gateways maintain a local cache of the service mapping. In the event that a gateway attempts to communicate with a service instance that has become unresponsive, the corresponding service-map cache record is invalidated.

Server-side load balancing

To facilitate fluid system scalability and prevent performance bottlenecks, the API Gateway services integrate a server-side load balancing mechanism. This component acts as a traffic orchestrator, distributing incoming requests across multiple instances of a target microservice. By doing so, the gateway ensures that traffic is routed only to available nodes, preventing any single instance from becoming overwhelmed. This approach is helps to maintain high availability and consistent response times. By dynamically balancing the workload, the system can effectively absorb sudden spikes in traffic and maintain a seamless user experience.

Containers

In order to take full advantage of the microservice architecture, we opted to deploy services with containers. Deploying microservices with containers allow to encapsulate each service with it's own environment. This allows to easily scale the system based on the specific needs. This also grants a more reliable system due the fact that containerization ensures that a failure is contained within it, without causing problems to other instances of the same service.

Caches

To enhance system availability, we implemented a robust caching layer for those services that could benefit from it. This strategy is particularly effective for services characterized by read-intensive workloads, like those interfacing directly with the DBMSs. By serving requests from the cache, we significantly decrease response times. Caching approach also allows the system to pre-compute results for some high-demand requests, avoiding to iterate complex and time-consuming operations on demand.

Differentiated Database

To optimize the management of different data types, we adopted a differentiated data storage strategy, differentiating our database and data layer architecture into five distinct functional areas based on the type of data is going to handle. This modular organization allows each domain to utilize the most efficient storage technology for the services that are going to use that data. This division further enables a tailored approach to Data Consistency based on the criticality of the information:

- **Strong Consistency**: For data where coherency among all nodes is non-negotiable (such as user credentials), the system can use Relational Database Management Systems.
- **Eventual Consistency**: For data types with less stringent consistency requirements, such as issue updates or path information, the system can rely on NoSQL technologies. By adopting an eventual consistency model, the system can have higher availability. This trade-off allows the system to remain highly responsive and performant.

3 User Interface Design

3.1 Overview

The BBP user interface is designed according to "Minimal Attention UX" principles to ensure safety during mobile use. While the RASD defined the visual appearance through mockups, this section details the navigation logic and the structure of the transitions between views, organized by functional areas.

3.2 Navigation Logic

3.2.1 Entry Point and Authentication Flow

Access to the system is managed through the *Login* screen, which acts as the main hub to direct the user towards the authenticated or anonymous flow.

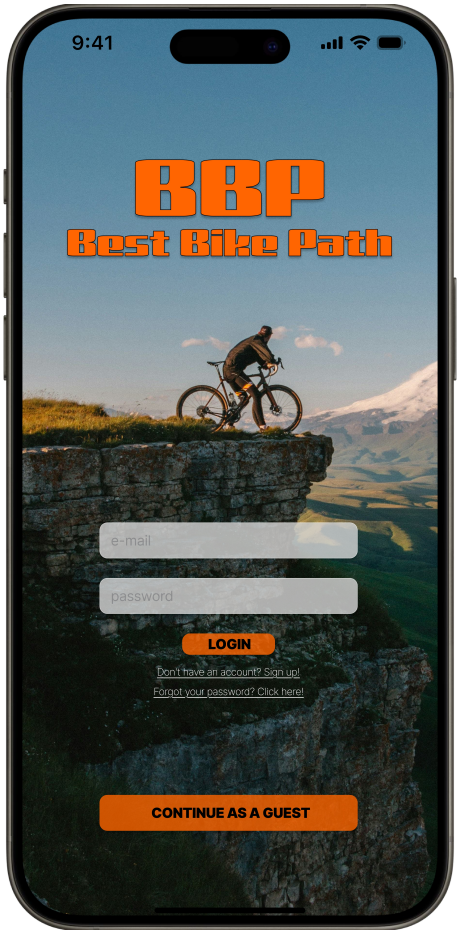


Figure 3: Start Screen and Access Logic

Transitions and Logic:

- **Guest Path:** The ***Continue As A Guest*** button instantiates an anonymous session and redirects the user directly to the *Map View* (see Fig. 4) with limited functionality.
- **Registered Path:** Entering credentials and tapping ***Login*** validates the token; if successful, the user is redirected to the *Map View* with the

Bottom Bar enabled.

- **Registration:** The ***Sign up*** link opens the registration form, which, once completed, returns to this screen for the first login.
- **Password Recovery:** The ***Forgot your password?*** link starts the external credential recovery flow via secure email. Once the process is complete, the user remains on this screen to log in with the new password.

3.2.2 Core Experience: Search and Tracking

The map is the central view of the application. From here, the user plans and takes action.

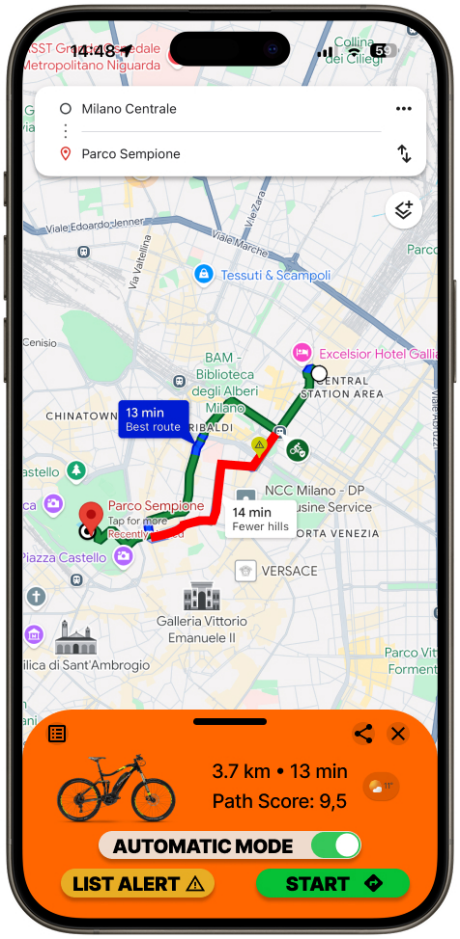


Figure 4: Main Map View and Route Selection

Transitions and Logic:

- **Route Selection:** Interacting with search results or tracks on the map updates the Bottom Sheet shown in the figure.
- **Start Navigation:** The ***Start*** button initializes the Tracking state, placing the interface in "Driving Mode".
- **Alerts:** The ***List Alert*** button opens a modal overlay listing known obstacles on the route.

- **Automatic Mode:** The ***Automatic mode*** toggle switch allows the user to explicitly enable or disable sensor data acquisition for the upcoming trip.
- **Sharing:** The ***Share*** icon (top right) activates the operating system's native share sheet for sharing the route with other users.
- **Text View:** The ***List*** icon (top left of the panel) switches the view from the graphical map to a text list of navigation directions.

3.2.3 Data Governance Flow

At the end of a tracking session, the system prevents an immediate return to the Home page, forcing a critical data validation step.

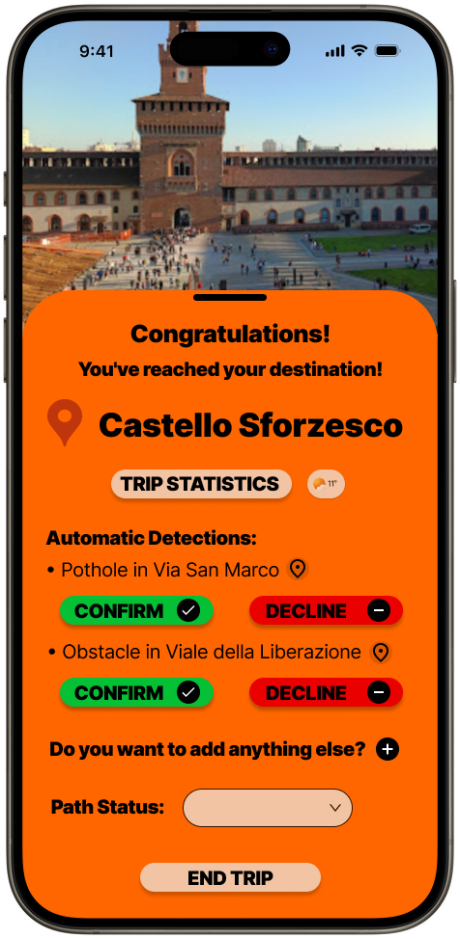


Figure 5: Confirmation and Data Validation Screen

Transitions and Logic:

- **Loop Closure:** This view is accessible only after the `Stop Recording` event and represents the mandatory Data Governance step.
- **Immediate Feedback:** The ***Trip Statistics*** section immediately displays a summary of the performance metrics calculated for the trip just completed, such as distance, duration, average speed, etc.
- **Automatic Validation:** The ***Confirm*** / ***Decline*** toggles allow the

user to validate or reject anomalies detected by the sensors, preventing false positives.

- **Manual Enrichment:** The ***Do you want to add anything else?*** button opens a context menu that allows the user to manually report any issues not detected by the sensors or add specific details.
- **Qualitative Evaluation:** The ***Path Status*** selector allows the user to assign an overall rating on the condition of the path just completed (e.g., Optimal, Average, Poor).
- **Exit and Persistence:** The ***End Trip*** button is the only way out: it finalizes the session, saves the confirmed data to the remote database, and redirects the user to the *History/Profile* view (Fig. 6).

3.2.4 Persistence and History

The personal area allows asynchronous consultation of saved data.

Figure 6: View Trip History

Transitions and Logic:

- **Access:** Accessible via the ***Profile*** tab in the Bottom Navigation Bar.
- **Detail:** Tapping on a card in the list opens the detail view of the individual trip.

4 Requirements Traceability

The following matrix maps each functional requirement to the system components responsible for its realization. An 'X' indicates that the component participates in satisfying the requirement.

ID	Requirement	Mobile App	Acc. API GW	Trip API GW	Acc. Creation Svc	Acc. Deletion Svc	Cred. Subsys.	Acc. Subsys.	Act. Handler Svc	Act. Storage Subsys.	Trip Planner Svc	Path Subsys.	Pub. Info Subsys.	Weather Proxx	Email Proxy
R1	Create an account	X	X		X		X	X							X
R2	Log in	X	X				X	X							
R3	Reset password	X	X				X								X
R4	Update profile info	X	X					X							
R5	Delete account	X	X			X	X								X
R6	Start recording	X													
R7	Pause/Resume	X													
R8	Save activity	X		X					X	X					
R9	Track pos. & stats	X													
R10	Retrieve weather								X			X		X	
R11	Report path status	X		X									X		
R12	Submit feedback	X		X									X		
R13	Report problems	X		X									X		
R14	Enable auto-detect	X													
R15	Analyze anomalies	X													
R16	Review anomalies	X													
R17	Confirm anomaly	X		X									X		
R18	Compute routes	X		X							X	X	X		
R19	Visualize routes	X		X							X	X			
R20	Path Score												X		
R21	Display obstacles	X		X									X		
R22	Filter search	X		X							X	X			
R23	View list	X		X						X					
R24	View trip details	X		X						X		X			
R25	Delete activity	X		X						X					
R26	Search activity	X		X						X					
R27	Filter history	X		X						X					

5 Implementation, Integration and Test plan

5.1 Implementation Plan

The system architecture allows for parallel development. However, adhering to the **Critical Component Strategy**, prioritizing high-risk and foundational modules ensures that the core value proposition is secured early in the lifecycle.

5.2 Component Integration Analysis

The following table classifies system components based on the **Impact of Failure**. The scale ranges from 1 to 5.

- **4 - Critical:** Complete system outage or total loss of the application's core purpose.
- **3 - High:** Major functionality unavailable, but the system may offer degraded service.
- **2 - Moderate:** Secondary features unavailable; core flow remains intact.
- **1 - Low:** Auxiliary information missing or minor workflows blocked.

Component	Rationale for Classification	Impact
Trip API Gateway	Single point of entry for all core features. If down, the App cannot communicate with the backend.	**4**
Path Subsystem	Provides map data and road network topology. Without it, no visualization or routing is possible.	**4**
Trip Planner Service	Core business logic. The primary goal of "Best Bike Paths" is finding routes. Without it, the app loses its purpose.	**4**
Credentials Subsystem	Manages Authentication (Login) and Security. Critical for user access, though Guest Mode bypasses it.	**3**
Account Subsystem	Essential for user persistence, but Guest Mode mitigates total failure.	**3**
Activity Handler Service	Responsible for tracking rides. High value, but users could theoretically still use the map for visual navigation without active recording.	**3**
Account API Gateway	Entry point for Auth/Profile operations. If down, login and registration fail, limiting the app to Guest features.	**3**
Activity Storage Subsystem	Manages history. Failure prevents viewing past trips, but does not block new rides (which can be buffered locally).	**2**
Public Info Subsystem	Crowdsourcing logic (Issue Reporting, Scoring). Important for data enrichment, but the system functions without community reports.	**2**
Account Creation Service	Responsible to create new account. Important for overall experience, but Guest Mode mitigates the inconvenience.	**2**
Email Proxy Service	Used only for password reset or confirmation emails. Failure impacts a tiny fraction of daily user interactions, except for Account Creation.	**2**
Weather Proxy Service	Auxiliary feature. Lack of weather data reduces UX quality but does not stop any core function.	**1**
Account Deletion Service	Used for handle account elimination. The malfunctioning does not compromise the overall experience.	**1**

Table 2: Component Criticality

5.3 Integration Strategy

To balance the complexity of distributed microservices with the need for early risk mitigation, the project adopts a **Hybrid Integration Strategy**. This approach combines two distinct methodologies applied at different architectural levels:

1. **Macro-Level: Risk-Driven Integration.**
   At the system level, components are integrated based on their criticality. The "High Risk" subsystems are integrated first to verify the architectural viability of the solution.
2. **Micro-Level: Bottom-Up Integration.**
   Within each specific subsystem, development follows a linear Bottom-Up path: *DBMS* → *Data Access Layer* → *Service Logic* → *API Interface*. This ensures that each unit is stable before being exposed to the rest of the system.

The integration sequence is dictated by the dependency graph below, structured to support this hybrid flow:

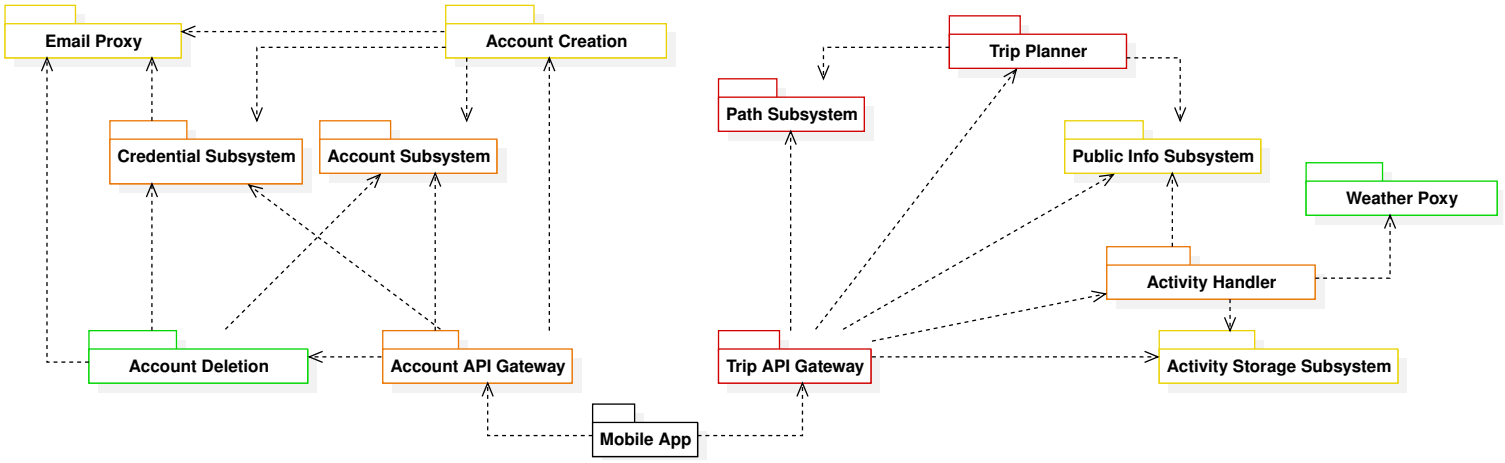


Figure 7: Dependency Graph highlighting the criticality of each module.

The integration roadmap is divided into four phases:

Phase 1: The Critical Core
The primary goal is to secure the application's core value: routing on a map.

- **Action:** Parallel Bottom-Up construction of the *Path Subsystem*, *Trip Planner* and *Trip API Gateway*.
- **Integration Point:** The *Trip Planner* is connected to the *Path Subsystem* and with the *Trip API Gateway* .
- **Validation:** Verification that the routing algorithm correctly returns a valid output, also by verifying the operations that pass by *Trip Api Gateawy*.

Phase 2: Activity Section
Once the core logic is validated, the modules related to activities and publishable info are developed.

- **Action:** Parallel development of *Activity Storage Subsystem* and *Public Info Subsystem*, follows the development of *Activity Handler* .
- **Integration Point:** Connection of *Activity Handler* to *Activity Storage Subsystem* and *Public Info Subsystem*. Then integrate *Trip Planner* with *Public Info Subsystem*.
- **Validation:** Confirmation that a registered user can simulate a ride and that the data is persisted. Verify that manual reports are correcty processed and stored. Verify the remaining funcionalities of *Trip Planner* after the integration.

Phase 3: Account Section
After main application functionalities are tested, core modules related to account management are developed.

- **Action:** Develop simultaneously the *Credential Subsystem* and *Account Subsystem*. After this, develop the secondary modules *Account Creation* and *Email Proxy*.
- **Integration Point:** Connect *Email Proxy* and *Account Creation* modules with their dependecies.
- **Validation:** Verify that all functionalities of the modules developed in this phase are working.

Phase 4: Gateway Aggregation
The system is exposed to the outside world.

- **Action:** Deployment of *Trip API Gateway* and *Account API Gateway*.
- **Integration Point:** Gateways are configured to route traffic to the services deployed in Phases 1 and 2.
- **Validation:** End-to-end API testing to verify authentication tokens and correct routing.

Phase 4: Auxiliary & Client Integration
Final polish and connection of external non-critical dependencies.

- **Action:** Integration of *Weather Proxy*, *Account Deletion*, and the *Mobile App*.
- **Integration Point:** The Mobile App connects to the live Gateways.
- **Validation:** User Acceptance Testing (UAT) and usability verification.

5.4 Test Plan

The testing strategy aligns with the hybrid integration approach, ensuring coverage from granular logic to system-wide workflows.

5.4.1 Unit Testing

Focuses on the correctness of the internal logic of each service, isolated from dependencies using mocks.

- **Tools:** JUnit 5, Mockito.
- **Key Targets:**
  - *Trip Planner:* Routing algorithm correctness on use-case graphs.
  - *Credentials:* Password hashing strength and JWT generation.

5.4.2 Integration Testing

Verifies the interactions defined in the sequence diagrams.

- **Tools:** REST Assured, TestContainers.
- **Key Targets:**
  - *Service-to-Service:* Ensure *Activity Handler* correctly calls *Weather Proxy* upon trip completion.
  - *Gateway-to-Service:* Verify that the API Gateway correctly handles service timeouts and load balancing.

5.4.3 System Testing

Validates the fully integrated system against the functional requirements.

- **Key Scenarios:**
  - *Full Ride Cycle:* Registration → Login → Search Route → Start Ride → Stop & Save.

– *Resilience:* Verify that the *Guest Mode* remains functional even if the *Account Subsystem* is forcefully taken offline (Simulated Failure).

6 Effort Spent

Student	Section 1	Section 2	Section 3	Section 4	Section 5
Guglielmi Leonardo	1h	39h	1h	2h	
Lo Conte Francesco	5h	1h	6h	6h	11h

Table 3: Effort spent by each team member per section

7 References

- **Standards referenced in the document:**
  - *IEEE Std 1016-2009* - IEEE Standard for Information Technology—Systems Design—Software Design Descriptions.
  - *Regulation (EU) 2016/679* (GDPR) - General Data Protection Regulation.
  - *ISO/IEC 27001* - Information Security Management.
- **Tools and Languages:**
  - *PlantUML* - Open-source tool used to generate Sequence Diagram via code.
  - *StarUML* - Software modeling tool used to create UML Diagram, with the exception of SDs.
  - *Draw.io* - Tool used to create Overview Diagram.
  - *Figma* - Tool used for User Interface (UI) mockups.
  - *Visual Studio Code* - Integrated Development Environment used for LaTeX editing.
  - *Git & GitHub* - Version control and collaborative platform.
  - *JUnit 5 & Mockito* - Referenced frameworks for Unit Testing.

8 Declaration of GenAI Usage

The project was supported by the use of generative artificial intelligence tools during the development of this document. The use of these tools was strictly limited to technical support, linguistic refinement, and syntax checking.

Tools Used

- **Model:** Google Gemini
- **Usage Scope:** Linguistic revision and technical typesetting support.

Description of Usage

The authors certify that they are the sole creators of the intellectual content, including all architectural decisions, design patterns selection, and testing strategies. The usage of AI was limited to:

- **Linguistic Checking:** The tool was used to review the English grammar and vocabulary of the technical descriptions to ensure clarity, professional tone, and correctness.
- **Technical Support for LaTeX:** The tool provided assistance in troubleshooting compilation errors and formatting complex elements.
- **Design Choices Discussion:** The tool was used to discuss and challenge architectural decisions, helping the authors to evaluate trade-offs and strengthen the rationale behind the final design.
- **Interface Description Automation:** The tool streamlined the development of Interface Description from Sequence Diagram by analyzing the PlantUML diagram code and generating the LaTeXtable, which was then refined through careful review by ourselves.

Verification

All content processed for linguistic or technical review was manually verified by the authors to ensure it accurately reflects the original intent, the architectural logic, and the project requirements defined in the RASD.