UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Progetto P.A.C.

A.A. 2024/2025

Ingegneria
Informatica

Students

Bolis Filippo Antonio (matr 1079493)
Gotti Daniele (matr 1079011)
Mazzoleni Gabriele (matr 1079514)

Introduzione

SmarTrip è una soluzione software progettata per ottimizzare l'organizzazione dei viaggi, offrendo agli utenti un'esperienza semplice ed efficace nella pianificazione degli itinerari.

A partire dalle città e dai luoghi selezionati, l'app è in grado di generare automaticamente il percorso migliore, tenendo conto di vincoli temporali e preferenze personali.

Toolchain

Eclipse	CodeMR
Visual Studio Code	JUnit 5
GitHub	Postman
Java	Flutter
Maven	Dart
Spring Boot	Draw.io
jOOQ	Microsoft Word
JGraphT	Microsoft PowerPoint
DbBrowser for SQLite	

ITERAZIONE 0

Analisi requisiti

Requisiti utente:

- L’utente può **scegliere una città** tra quelle disponibili per organizzare un viaggio.
- L’utente può **visualizzare la lista dei luoghi** della città selezionata in modo da poter scegliere quali visitare.
- L’utente deve poter **decidere i “ritmi”** (giorni, orari partenza, ritorno, pausa, pranzo, ecc.) della vacanza.
- L’utente deve ricevere in maniera **automatica l’itinerario di viaggio**. L’itinerario calcolato **massimizza il numero di luoghi** che l’utente vuole visitare tra quelli selezionati.
- L’utente deve poter **scegliere tra una lista di ristoranti** disponibili più vicini al punto in cui effettuerà pause pranzo.

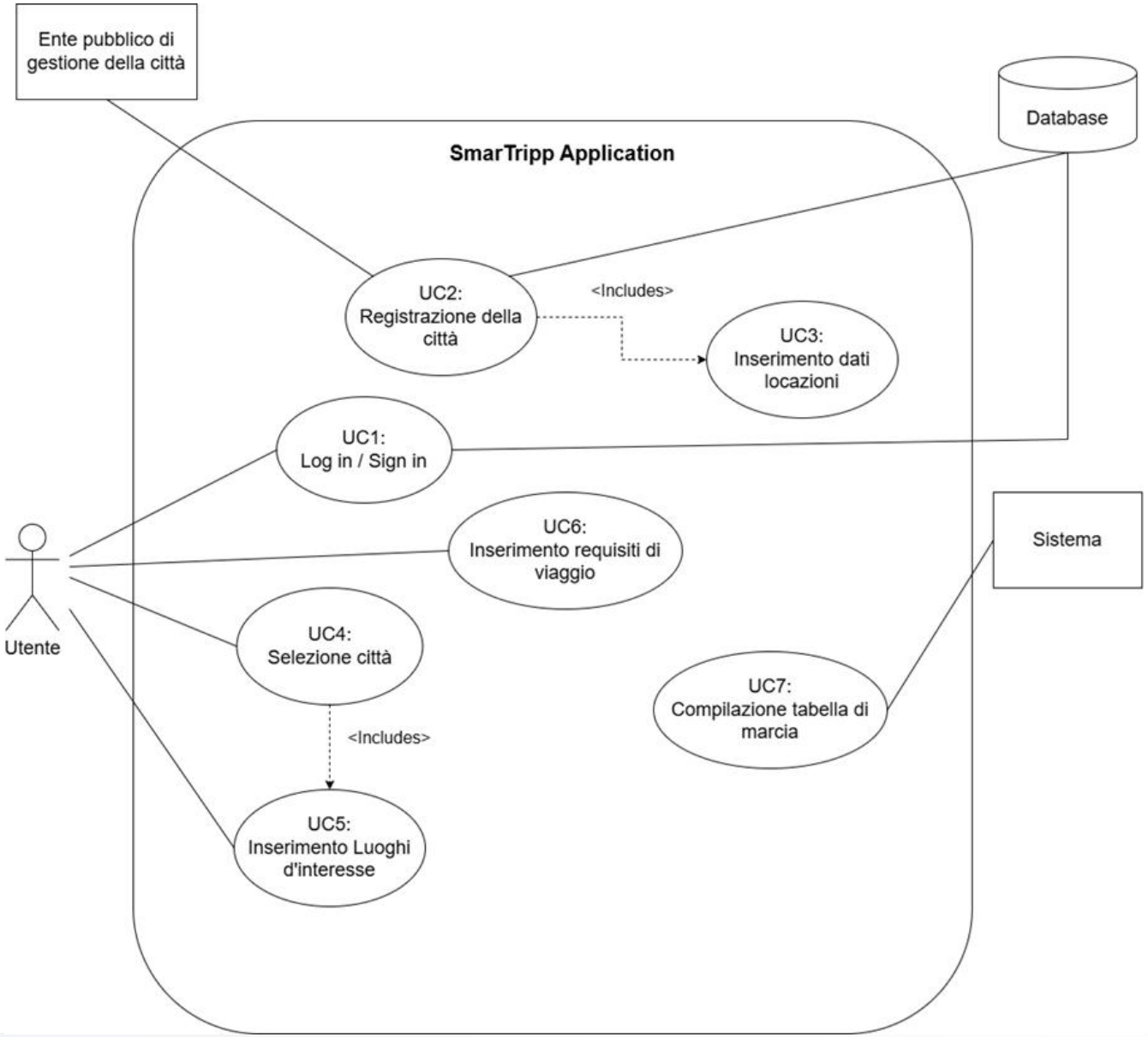
Analisi requisiti

Requisiti dell'addetto dell'amministrazione cittadina

- L'addetto deve essere in grado di **inserire** con facilità la sua **città** e i **luoghi** visitabili.
- L'addetto deve essere in grado di **inserire** per ogni luogo tutte le **caratteristiche utili** alla costruzione di un itinerario di viaggio.

NOTA: per questo progetto abbiamo deciso di concentrarci sulla soddisfazione dei requisiti dell'utente, cioè si realizzerà database, server e app mobile per utente funzionanti.

Diagramma dei casi d'uso

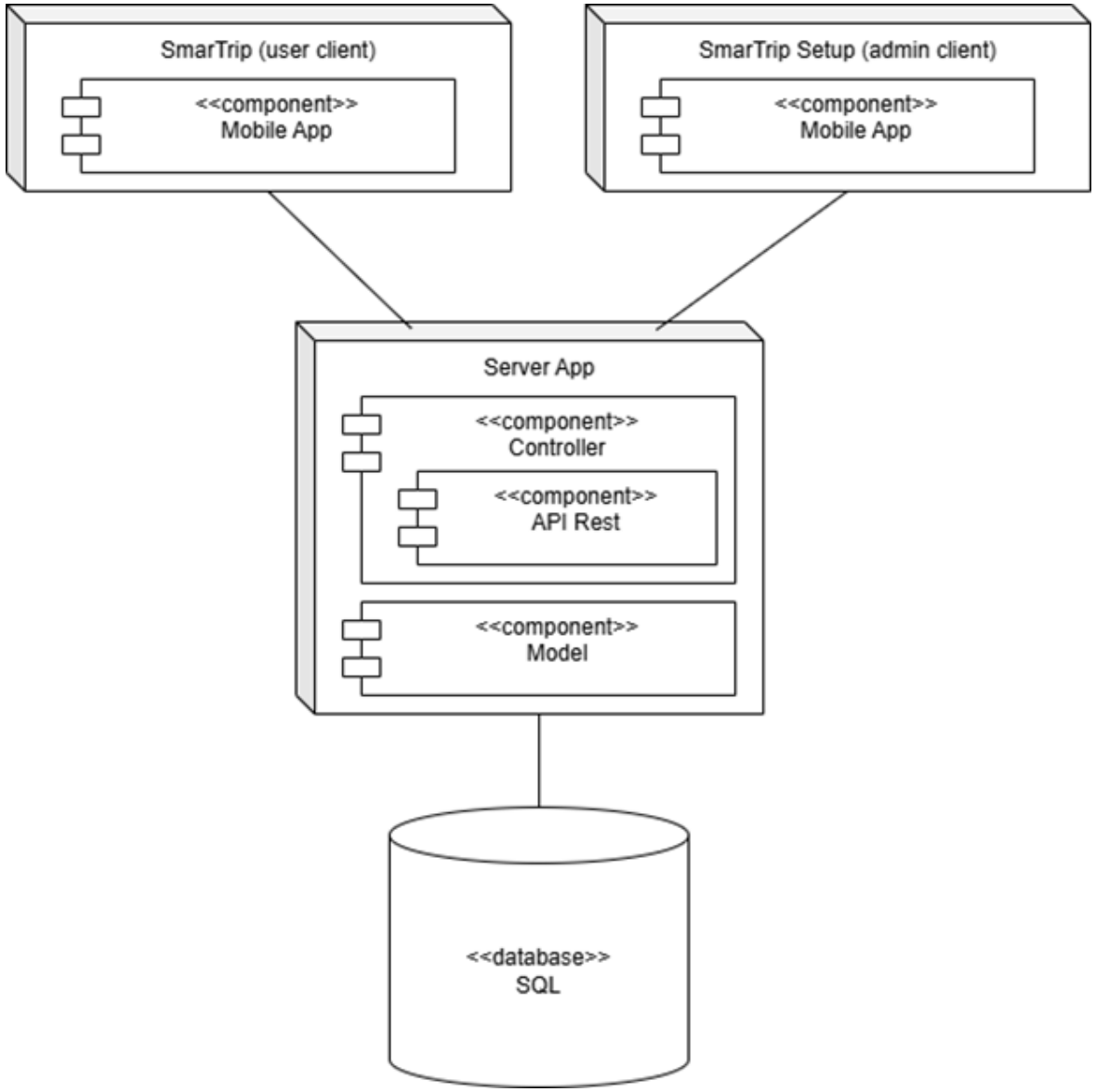


Analisi delle specifiche

Specifica	Priorità	Implementato
Sign in	Alta	Sì
Log in	Alta	Sì
Visualizzazione città disponibili	Alta	Sì
Visualizzazione luoghi disponibili della città selezionata	Alta	Sì
Inserimento dati del viaggio	Alta	Sì
Algoritmo ottimizzazione viaggio	Alta	Sì
Visualizzazione mappe precedenti	Bassa	Sì
Inserimento città (app gestore)	Bassa	No
Inserimento luoghi (app gestore)	Bassa	No

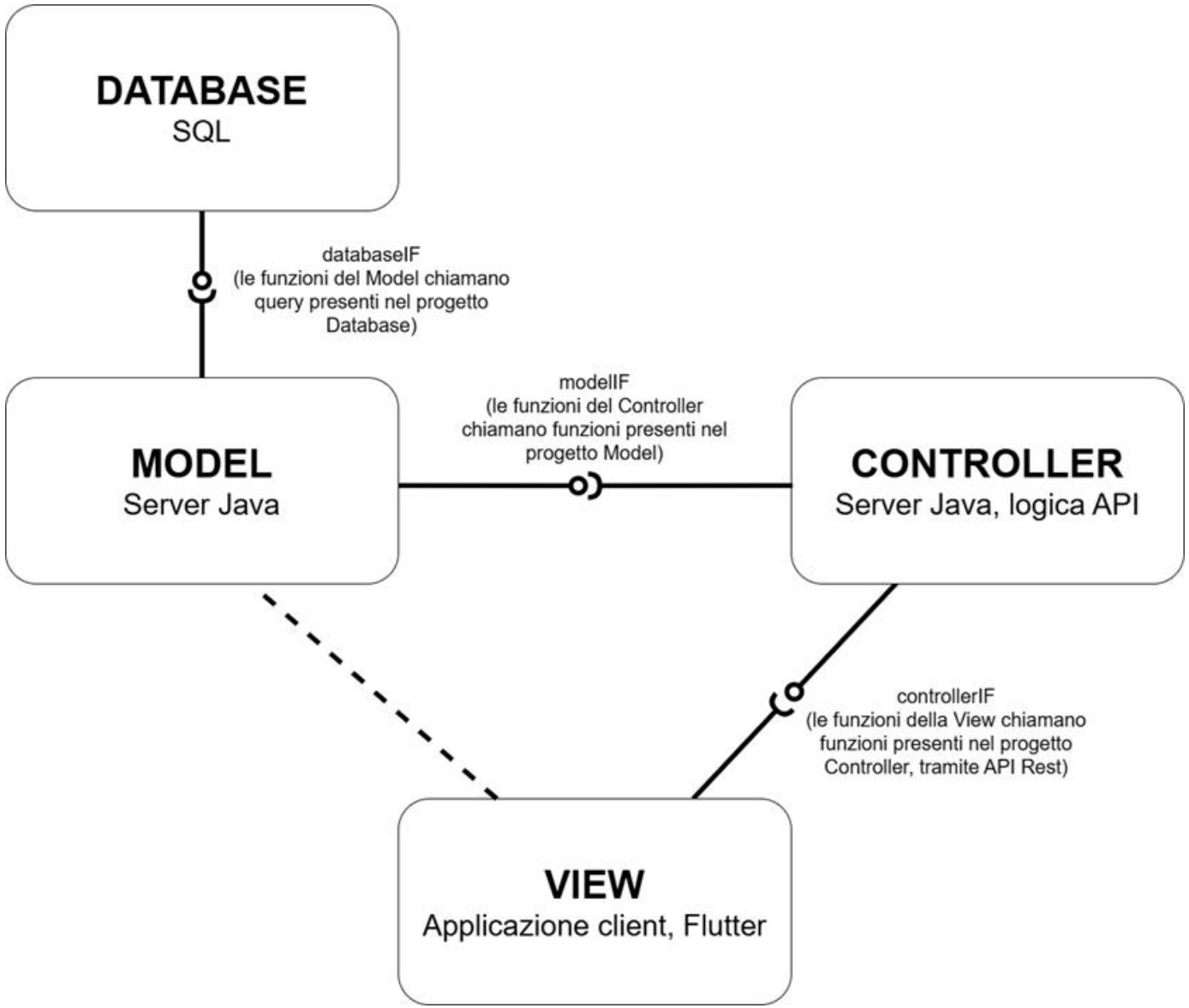
Analisi dell'architettura

Diagramma di deployment



Analisi dell'architettura

Diagramma MVC



ITERAZIONE 1

Aggiornamento dei casi d’uso

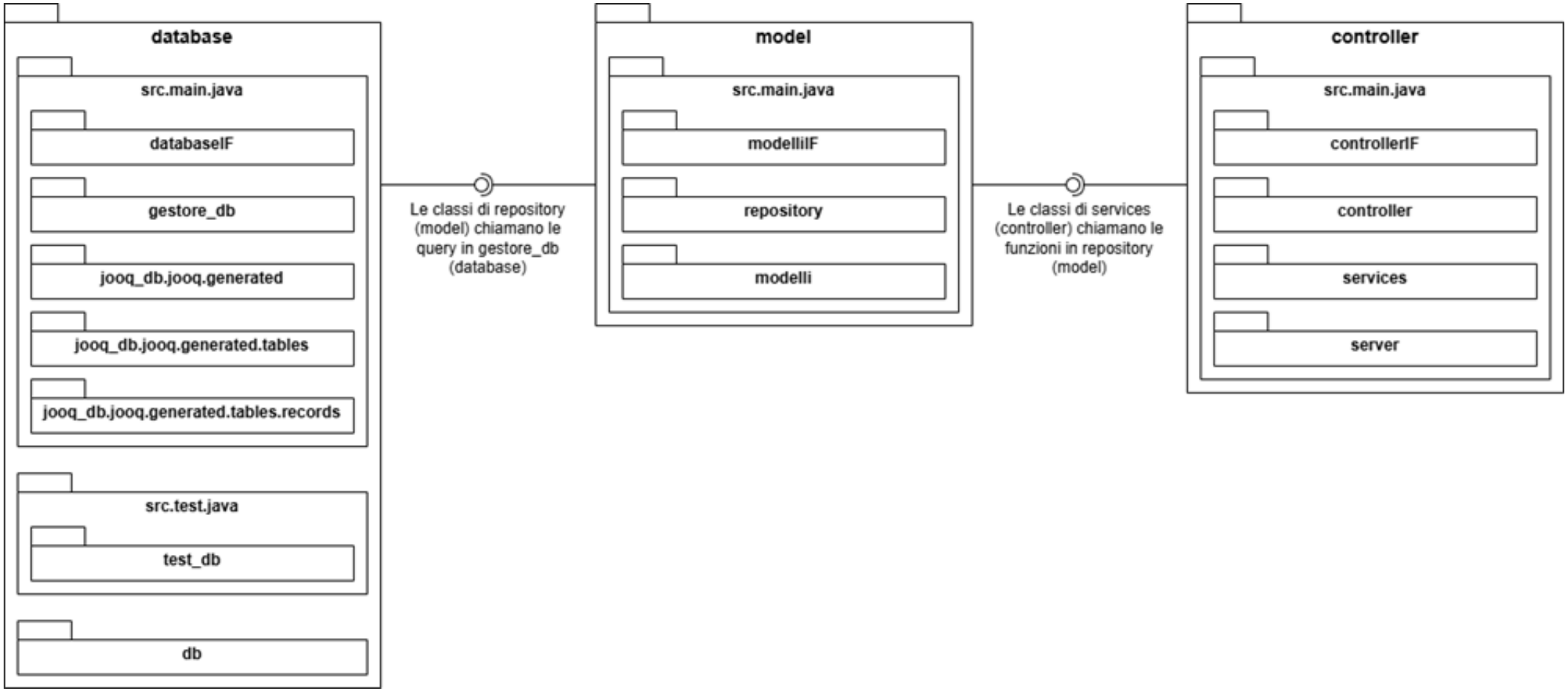
- **UC1: LOG IN / SIGN IN** - Eliminazione variabile email. Non è necessario l’utilizzo di una mail per il login, in quanto non viene utilizzata in alcun modo.

  POST CONDIZIONI: l’utente può eseguire nuovamente l’accesso all’app inserendo username e password.

- **UC2 e UC3** - I presenti casi d’uso restano invariati, è stata presa la decisione di non implementarli nella presente versione dell’app; il ruolo di “Ente” verrà ricoperto dal team di sviluppo, che si occuperà di popolare il database dei luoghi.

- **UC4 a UC7** - I presenti casi d’uso restano invariati.

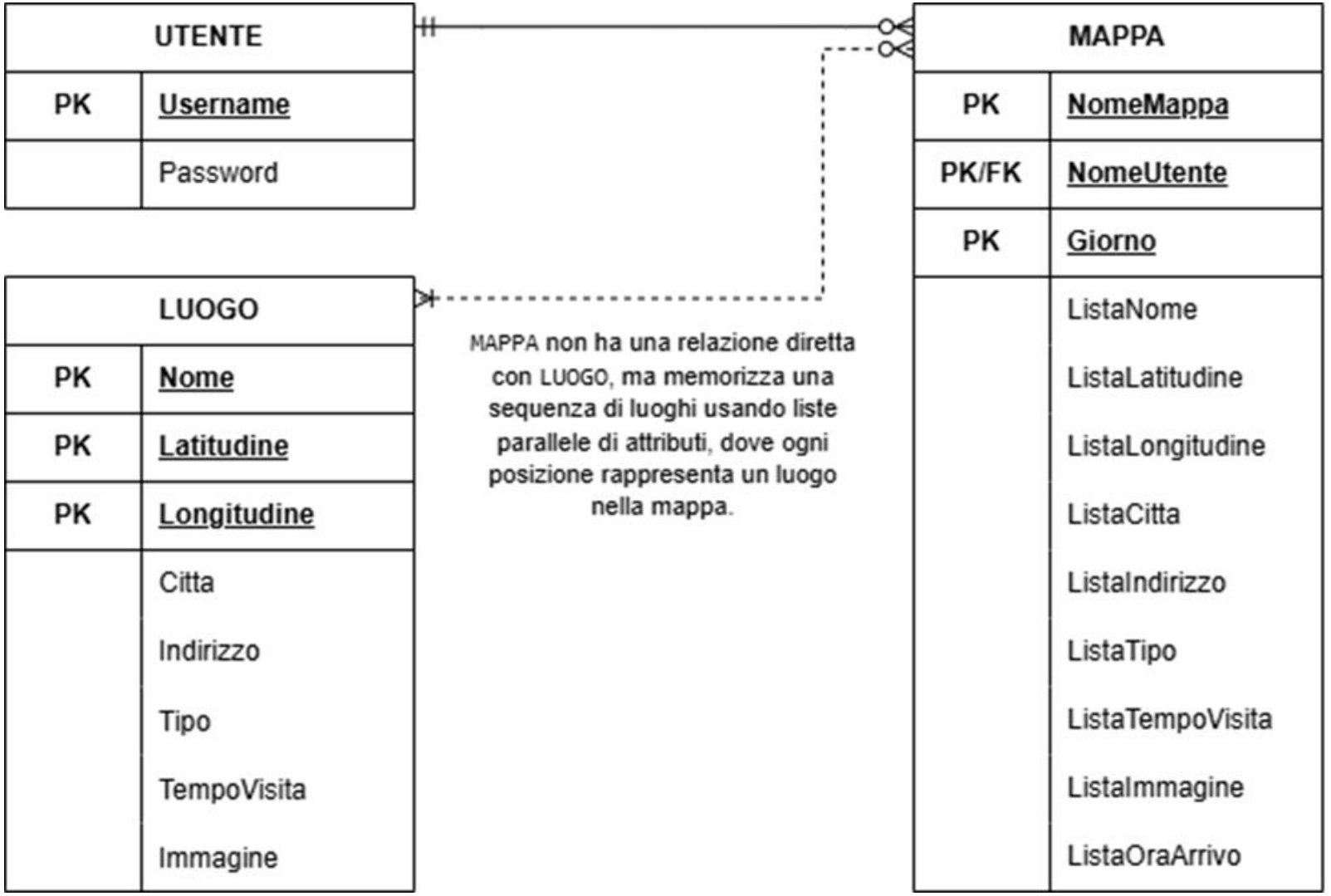
Architettura server - Diagramma dei package



Database

A livello di codice, l'accesso al database è gestito dalla classe **DatabaseManager**, che implementa il *Singleton pattern*.

Le operazioni di creazione, popolamento e interrogazione del database sono delegate a classi separate, anch'esse incapsulate nel **DatabaseManager**.



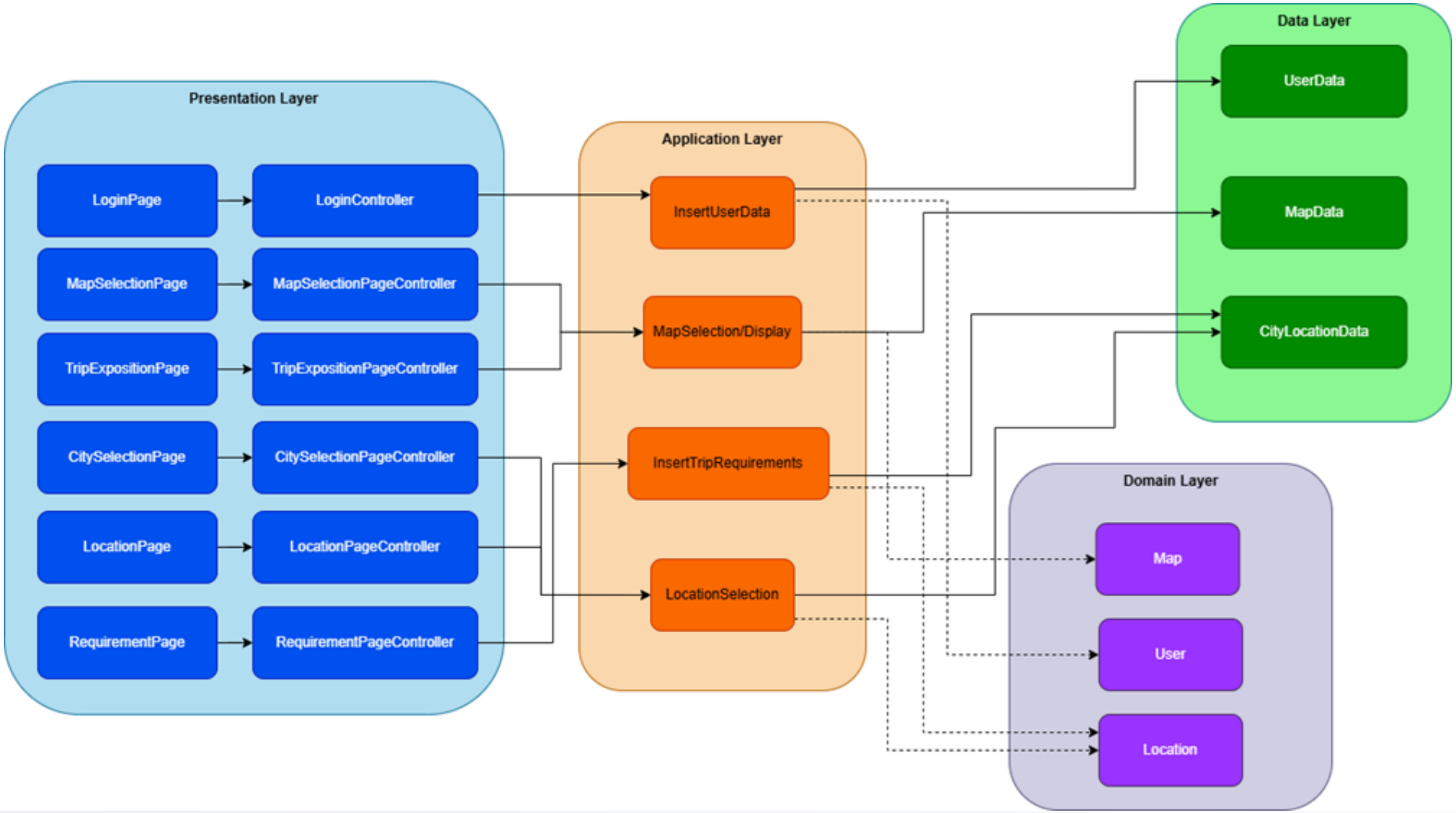
Testing sul database

Per le attività di testing è prevista una copia separata del database, denominata *test_db.db3*, utilizzata esclusivamente per i test automatizzati, in modo da non alterare né compromettere i dati presenti nel database principale *db.db3*.

I test sono sviluppati utilizzando **JUnit 5**, il framework di riferimento per il testing in Java.

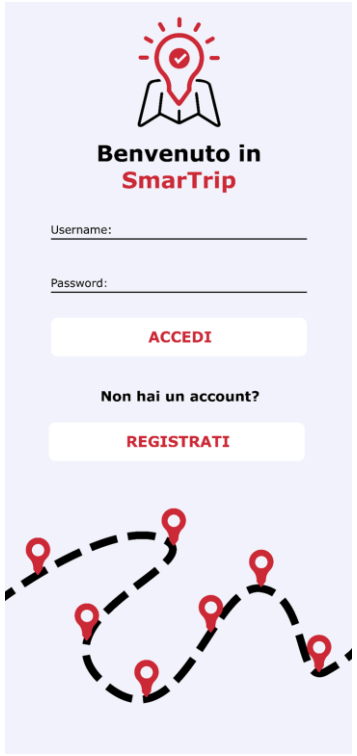
È inoltre prevista una classe *DaTestare*, progettata per eseguire in cascata tutti i test definiti, in modo da fornire un riscontro complessivo e immediato sull'esito delle verifiche.

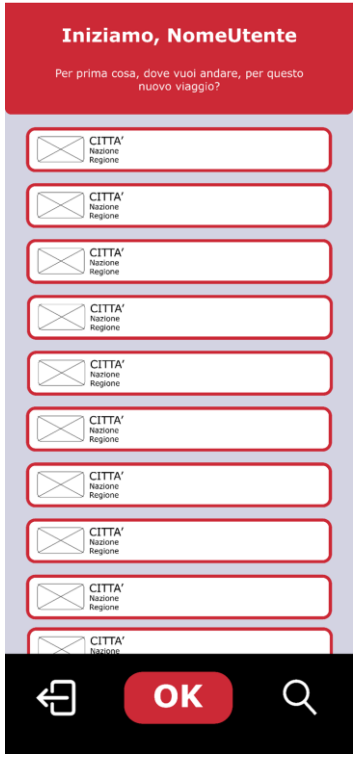
Progettazione dell'architettura client

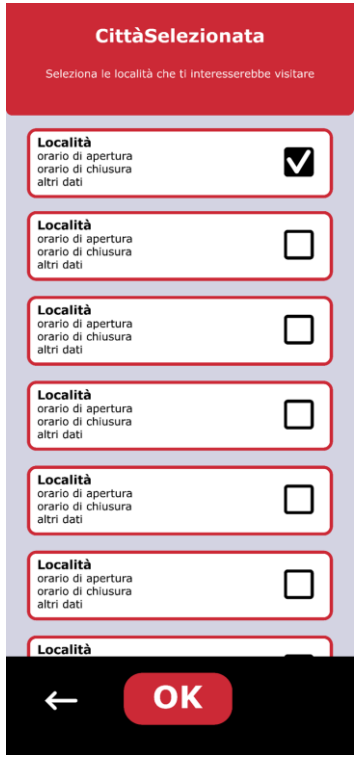


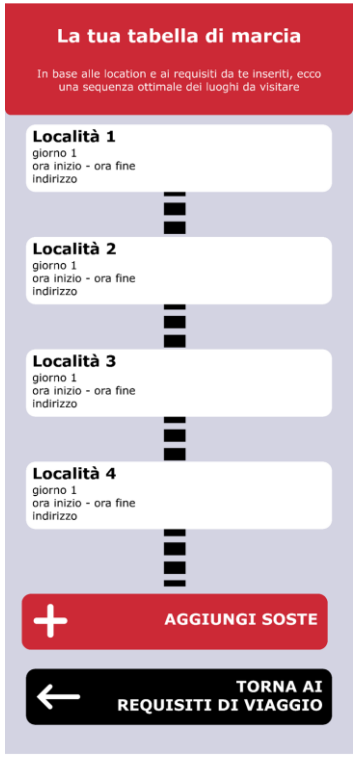
Progettazione UI

Progettazione preliminare realizzata con *Adobe Illustrator*

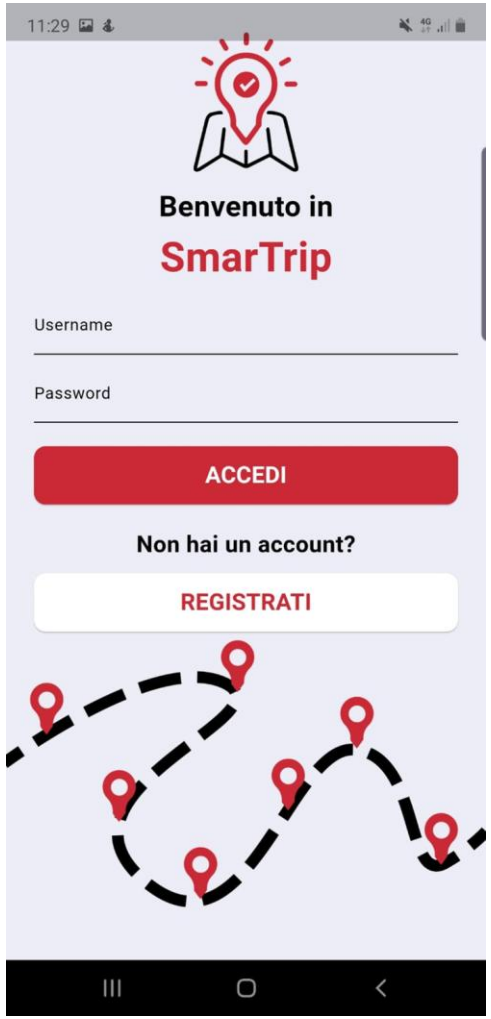


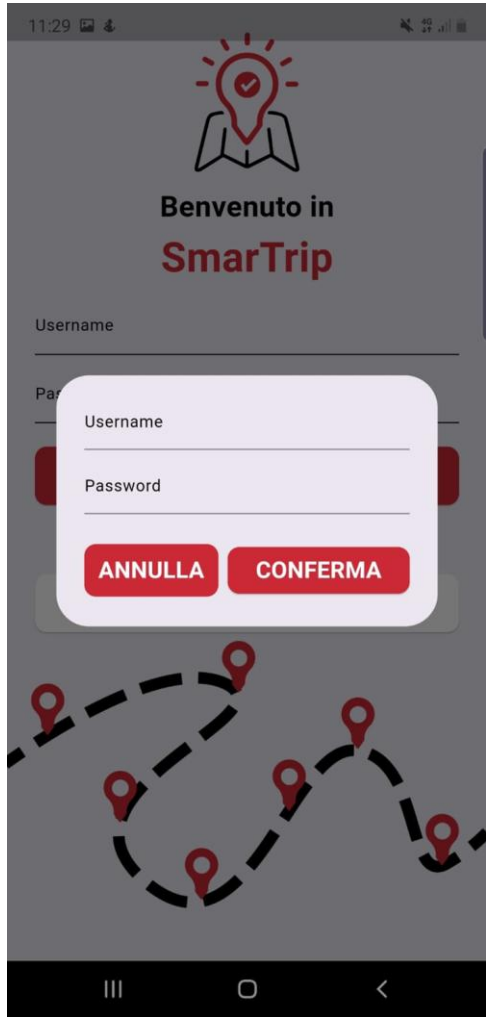






Implementazione front-end - log-in / sign-in page





API per la gestione account utente

Funzionalità realizzate in Java tramite il framework *Spring Boot*, che permette di esporre in modo semplice e standard le API REST necessarie per la comunicazione tra client e server.

Le operazioni principali gestite da queste API sono due:

- **SignIn**: registrazione di un nuovo utente nel database tramite username e password.
- **LogIn**: verifica delle credenziali di un utente esistente per consentire l'accesso all'applicazione.

API SignIn

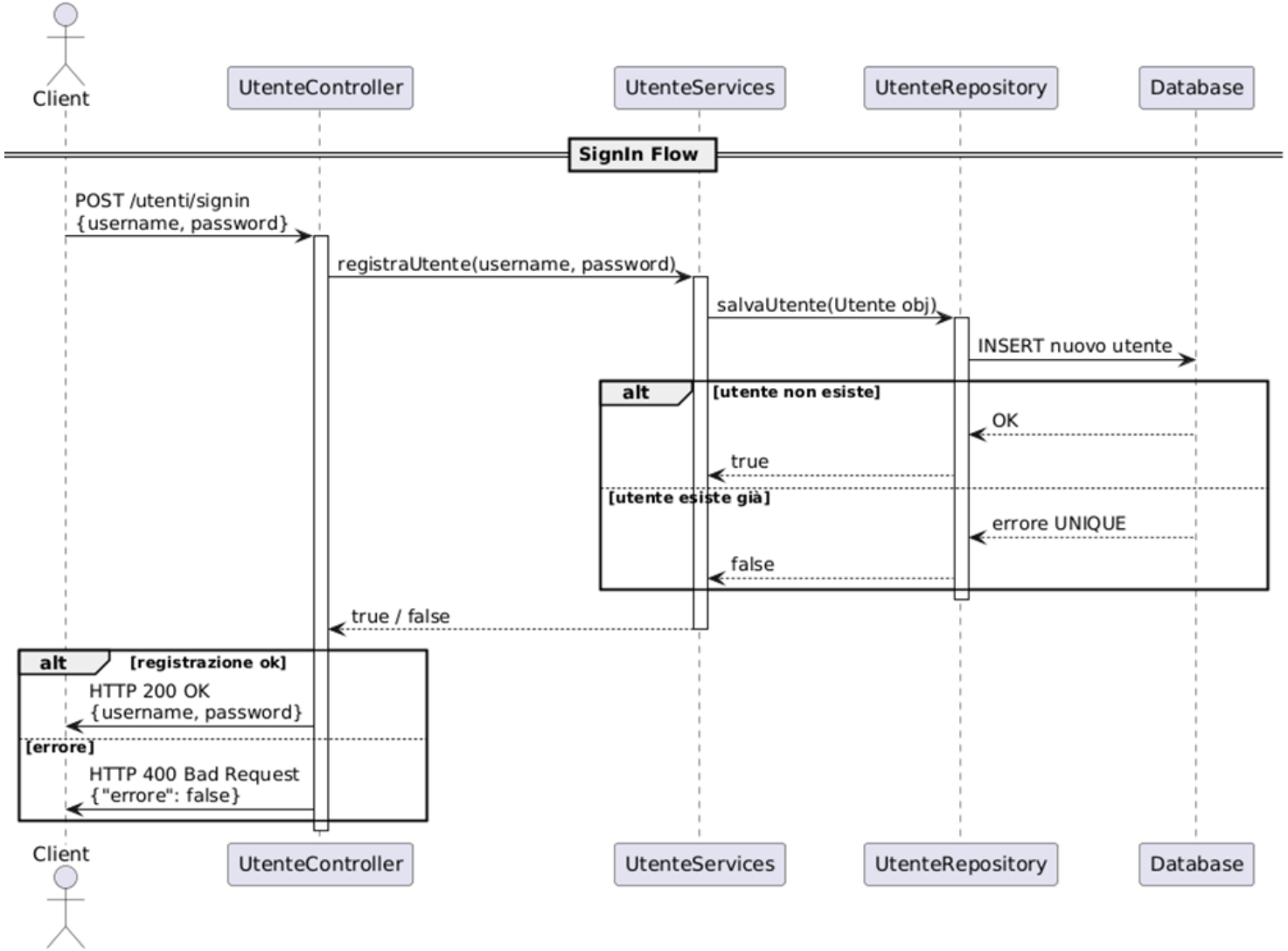
Questa operazione consente ad un utente di registrarsi al sistema.

Il metodo signInUtente riceve in ingresso un oggetto Utente contenente username e password, e chiama il metodo registraUtente del service.

Se l'utente non esiste già, viene salvato nel database (tramite il repository) e il sistema restituisce l'oggetto utente in risposta.

In caso contrario, viene restituito un messaggio d'errore (HTTP 400) che indica il fallimento dell'operazione.

SignIn - Sequence diagram



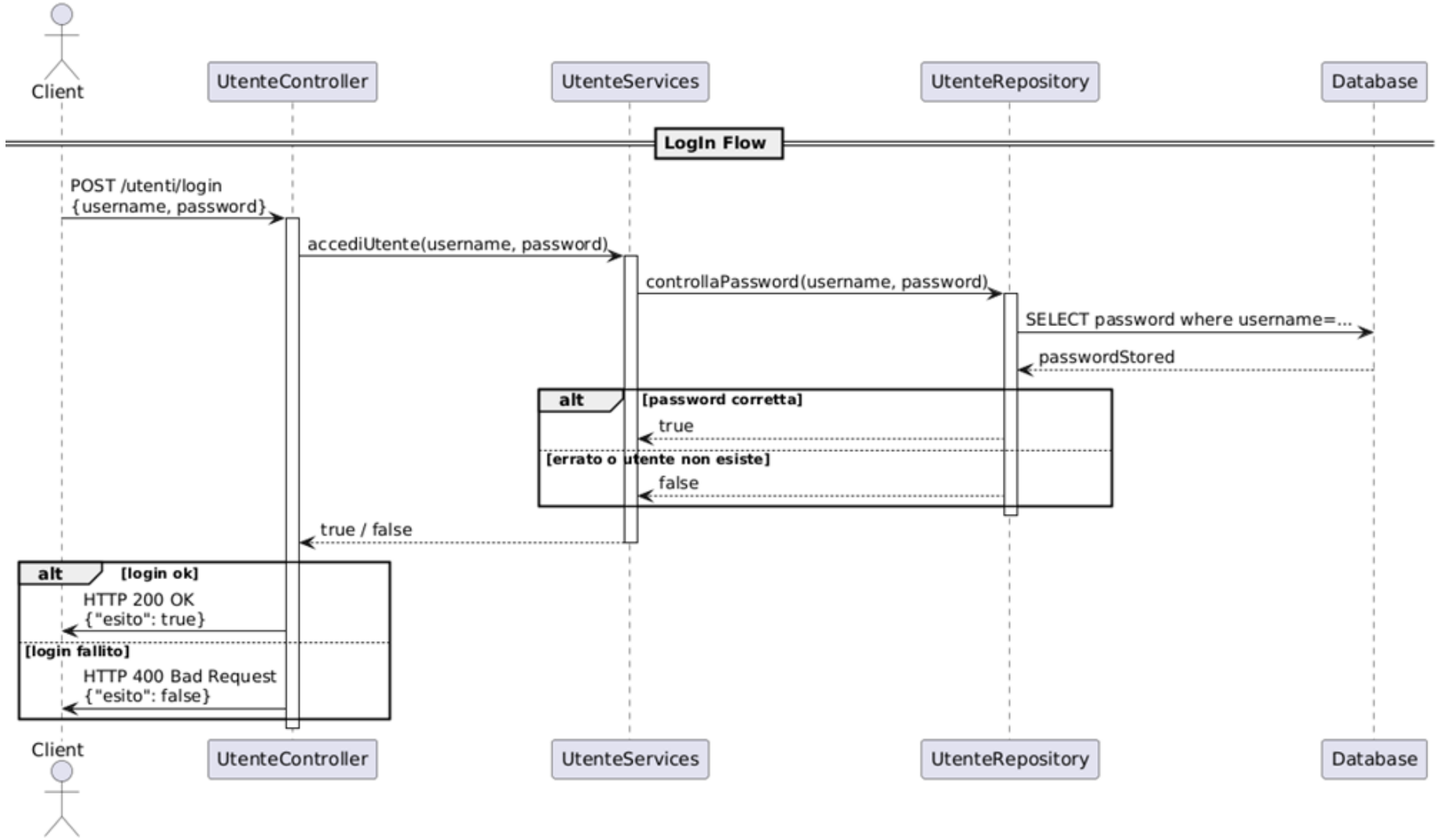
API LogIn

Permette ad un utente già registrato di effettuare il login nell’app.

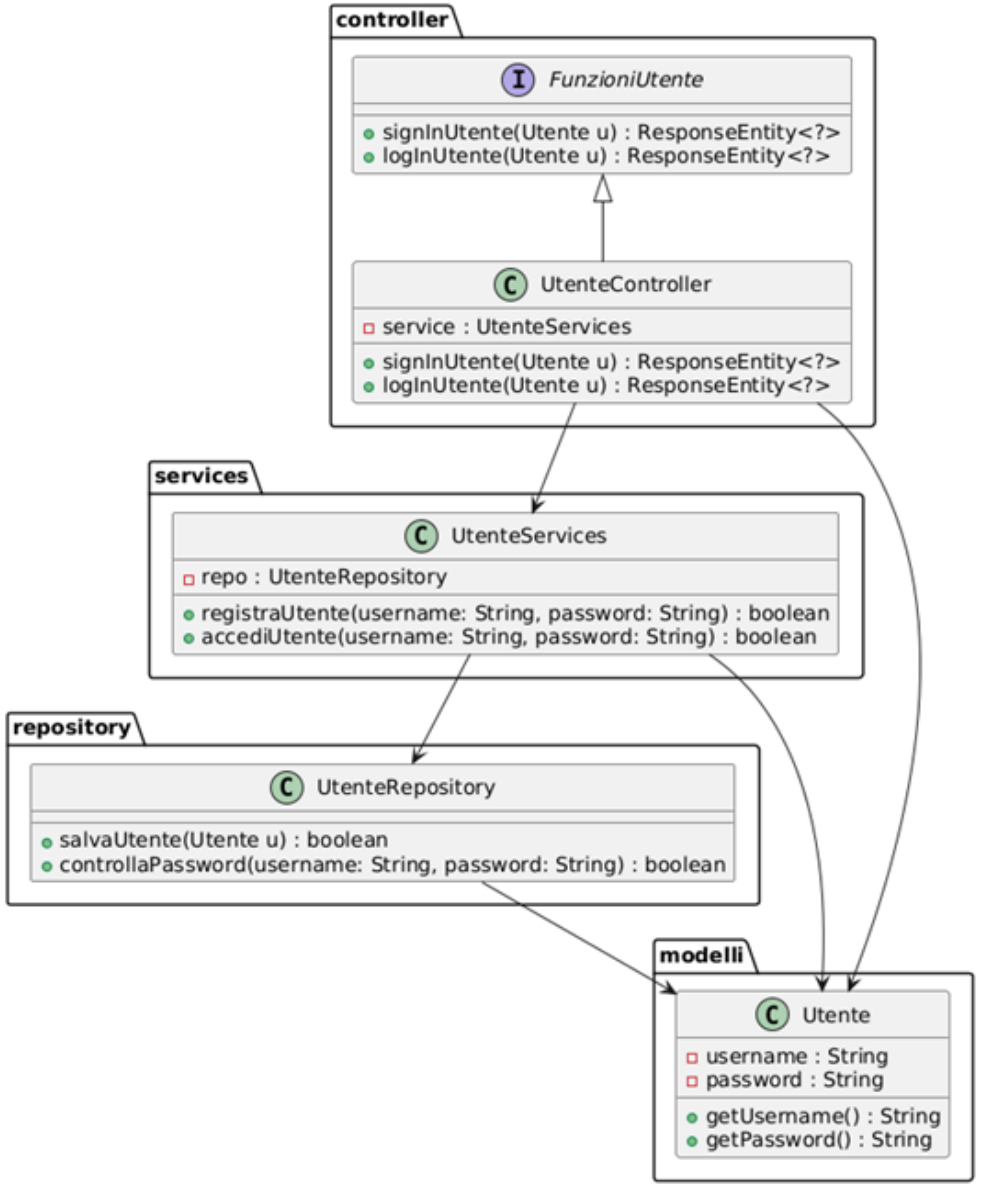
Il metodo logInUtente riceve l’oggetto Utente e chiama il metodo accediUtente del service per verificare che le credenziali siano corrette.

Se la password corrisponde a quella presente nel database, viene restituito l’esito positivo ("esito": true); in caso contrario, un errore HTTP 400 con "esito": false.

LogIn - Sequence diagram



API-Utente Class diagram



API per la gestione degli itinerari

AddItinerario

Questa è l'API cuore del progetto, l'idea è di permettere all'utente di costruire una tabella di marcia coerente con i suoi interessi e i suoi bisogni.

Durante l'iterazione 1 ci preoccupiamo solo di scrivere lo pseudocodice e la complessità della stessa.

Lo pseudocodice si trova nella doc in quanto molto corposo; in questa presentazione verrà mostrata l'analisi della complessità (per osservare al meglio il suo funzionamento è stato fatto anche un diagramma apposito).

AddItinerario - Complessità

La complessità algoritmica è data da 2 parti in cui l'Input è dato da un itinerario con N luoghi e G giorni.

1. Costruzione del grafo
2. Costruzione della tabella di marcia

AddItinerario - Complessità

Costruzione del grafo:

Si crea un grafo pesato e si aggiunge il nodo dell'alloggio + tutti gli N luoghi (in totale N+1 nodi)

La costruzione degli archi è data da due cicli annidati

```
for i da 0 a N-1:
    for j da i+1 a N:
```

Con complessità Temporale O(N^2)

Inoltre, essendo un grafo denso e completo tra luoghi, ogni coppia ha un arco.

Quindi ha complessità Spaziale O(N^2)

AddItinerario - Complessità

Creazione della tabella di marcia:

Per ogni giorno, si costruisce un percorso.

All'inizio del giorno si inizializza tempo, flags e percorso (lista vuota con il nodoAlloggio iniziale).
Finché ci sono luoghi non visitati nel grafo, sceglie il prossimo luogo da visitare.

for nGiorno da 0 a G-1

Quindi, ogni volta si visita e rimuove un luogo → max N iterazioni.

Ad ogni iterazione per trovare il prossimo luogo più vicino si deve guardare i nodi rimanenti → peggio ≈ O(N).

Questo ci conduce a una complessità Temporale di O(N) * O(N) = O(N^2)

AddItinerario - Complessità Temporale

La complessità Temporale totale dell'algoritmo è data allora da:

- Costruzione grafo: O(N^2)
- Per ogni giorno: O(N^2)
- Numero di giorni: G

O(N^2 + G·N^2) = **O(G·N^2)** (termine G·N^2 domina se G cresce)

AddItinerario - Complessità Spaziale

La complessità Spaziale totale dell'algoritmo è data da:

- Grafo: archi O(N^2)
- Tabella di marcia di return: O(G·N) (Ha G giorni come chiavi e per ogni giorno, la lista di luoghi è al massimo O(N) luoghi)

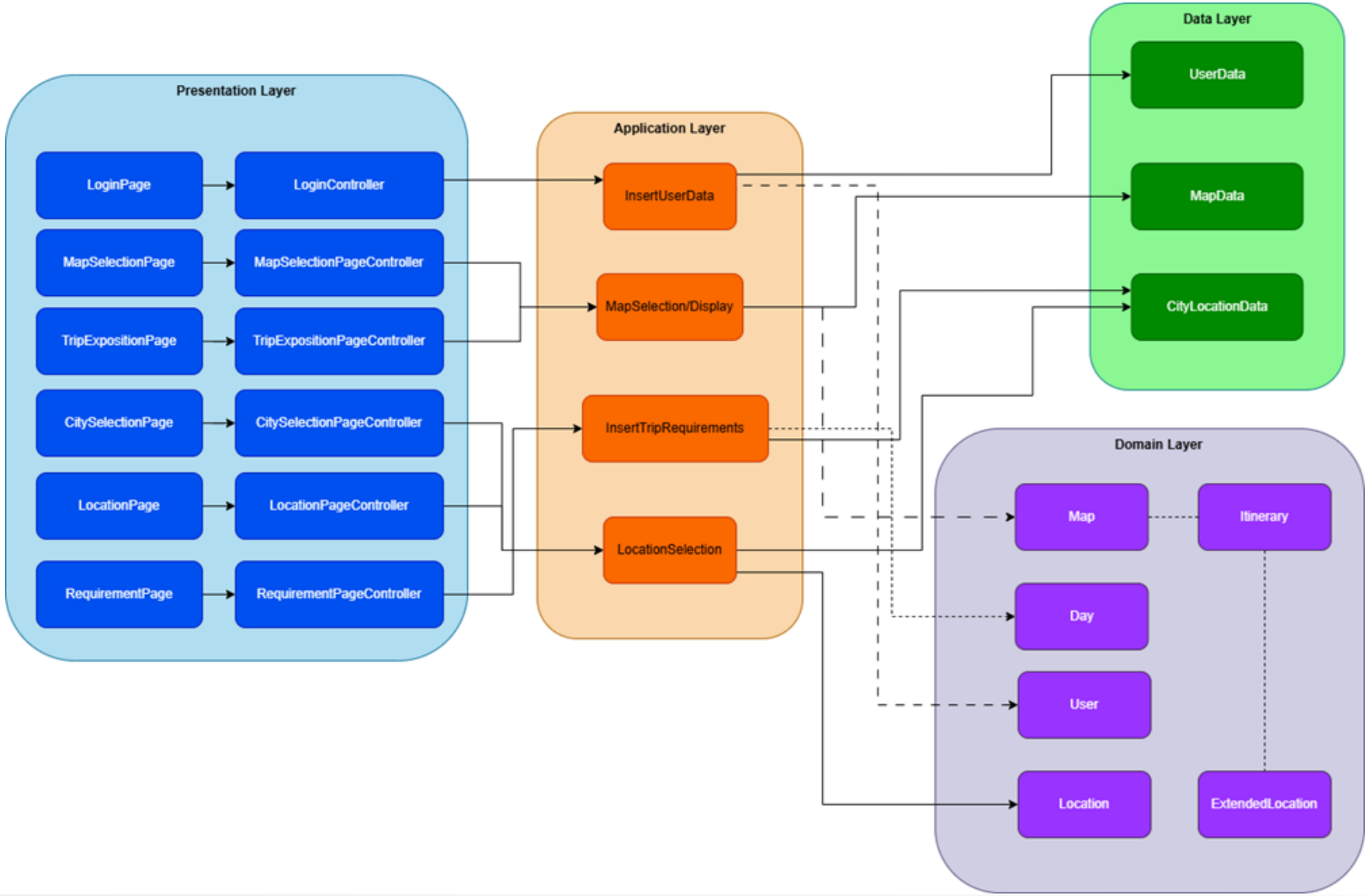
O(N^2 + G·N) = **O(N^2)** (il termine N^2 domina per N grande)

ITERAZIONE 2

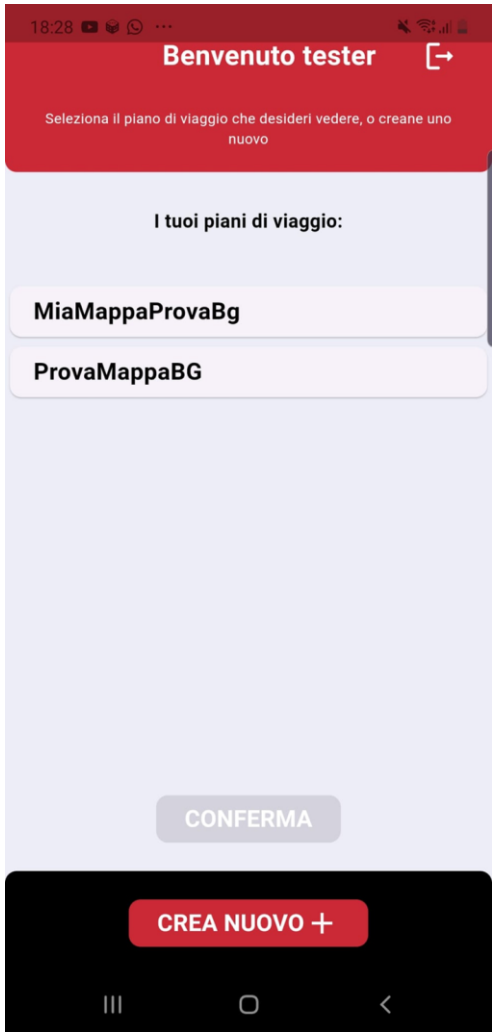
Aggiornamento dei casi d’uso

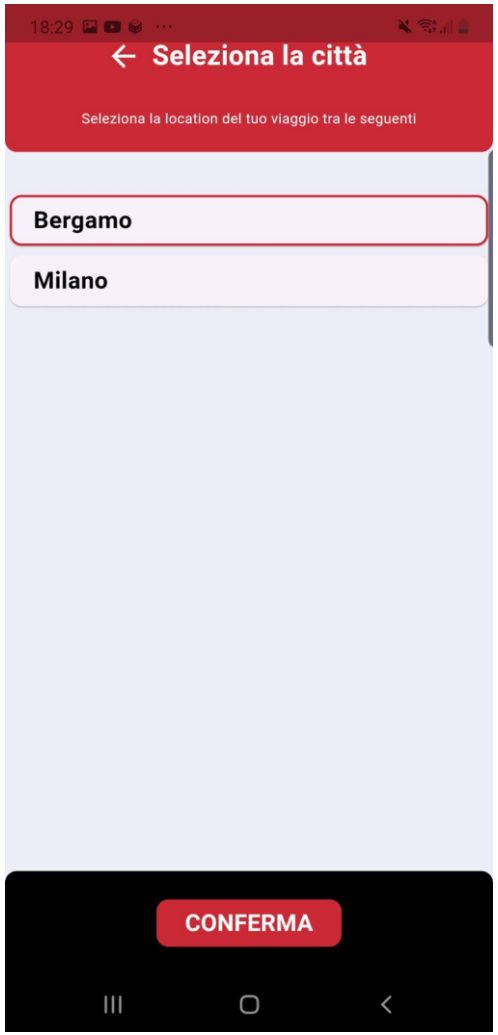
- **UC8 (NUOVO): SELEZIONE MAPPA** - L’utente visualizza i viaggi che ha pianificato in precedenza, scegliendo se visualizzarne nuovamente uno o se crearne uno nuovo. (Per comodità i piani di viaggio sono identificati come “mappe” o “itinerari”).
  PRE CONDIZIONE: l’utente deve avere eseguito il log-in all’applicazione.
  POST CONDIZIONE: se l’utente decide di creare una nuova mappa, vengono avviati i casi d’uso da 4 a 7.

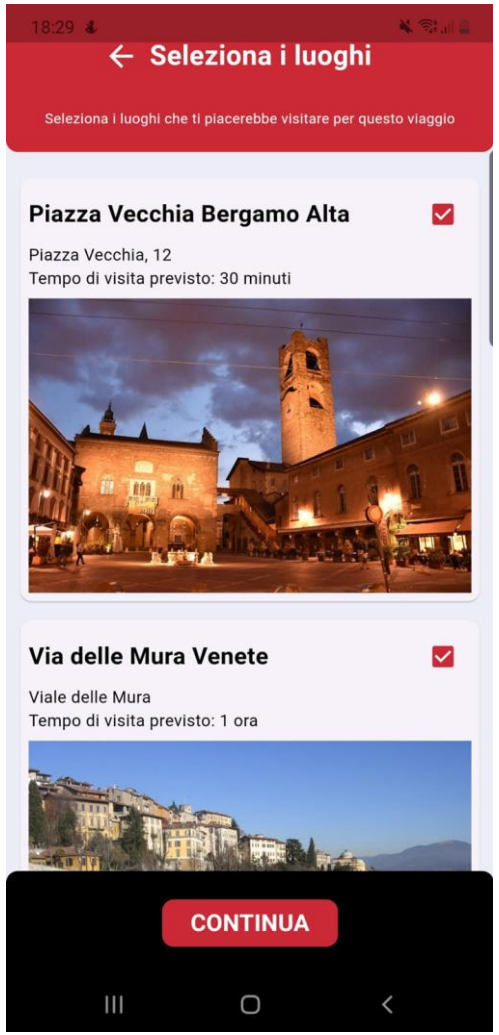
Aggiornamento dell'architettura client



Implementazione front-end - Map-City-Location page







API per la selezione di mappe/città/luoghi

Le API di Luoghi nascono per consentire alla nostra applicazione di recuperare in modo semplice e veloce informazioni turistiche e pratiche su città e luoghi.

L’architettura segue il classico modello a tre livelli:

- Controller: riceve e gestisce le richieste HTTP.
- Service: contiene la logica applicativa.
- Repository: si occupa dell’accesso diretto ai dati.

Tutte le richieste arrivano al controller, che chiama il service, il quale a sua volta interroga il repository.

Il repository comunica con il database e restituisce le informazioni richieste che, passando di nuovo dal service e dal controller, tornano infine al client.

API GetAllCitta

La prima API offerta è getAllCitta.

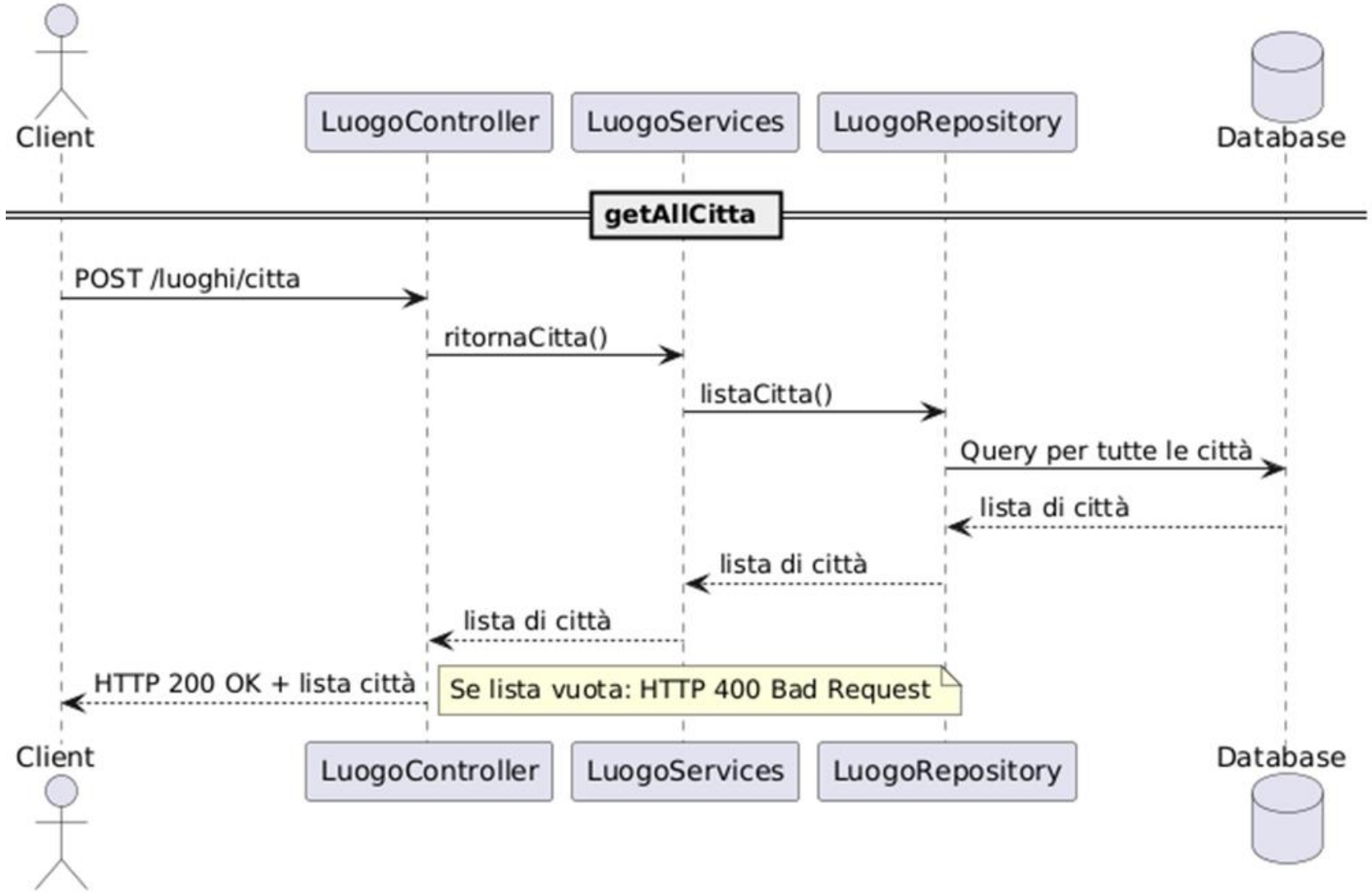
Questa serve per mostrare all'utente quali città sono supportate dal sistema.

Quando il client invia una richiesta, il controller (*getAllCitta*) chiama il metodo ritornaCitta del service.

Questo, a sua volta, contatta il repository (*listaCitta*), che interroga il database per ottenere tutte le città memorizzate.

Se la lista di città non è vuota, il controller restituisce al client un messaggio di successo (HTTP 200 OK) con l'elenco; altrimenti risponde con un errore (HTTP 400 Bad Request) e un messaggio che indica che non ci sono città registrate.

GetAllCitta - Sequence diagram



API getLuoghiByCitta

Il secondo servizio è getLuoghiByCitta, un'API che permette di ottenere tutti i luoghi di interesse presenti in una determinata città.

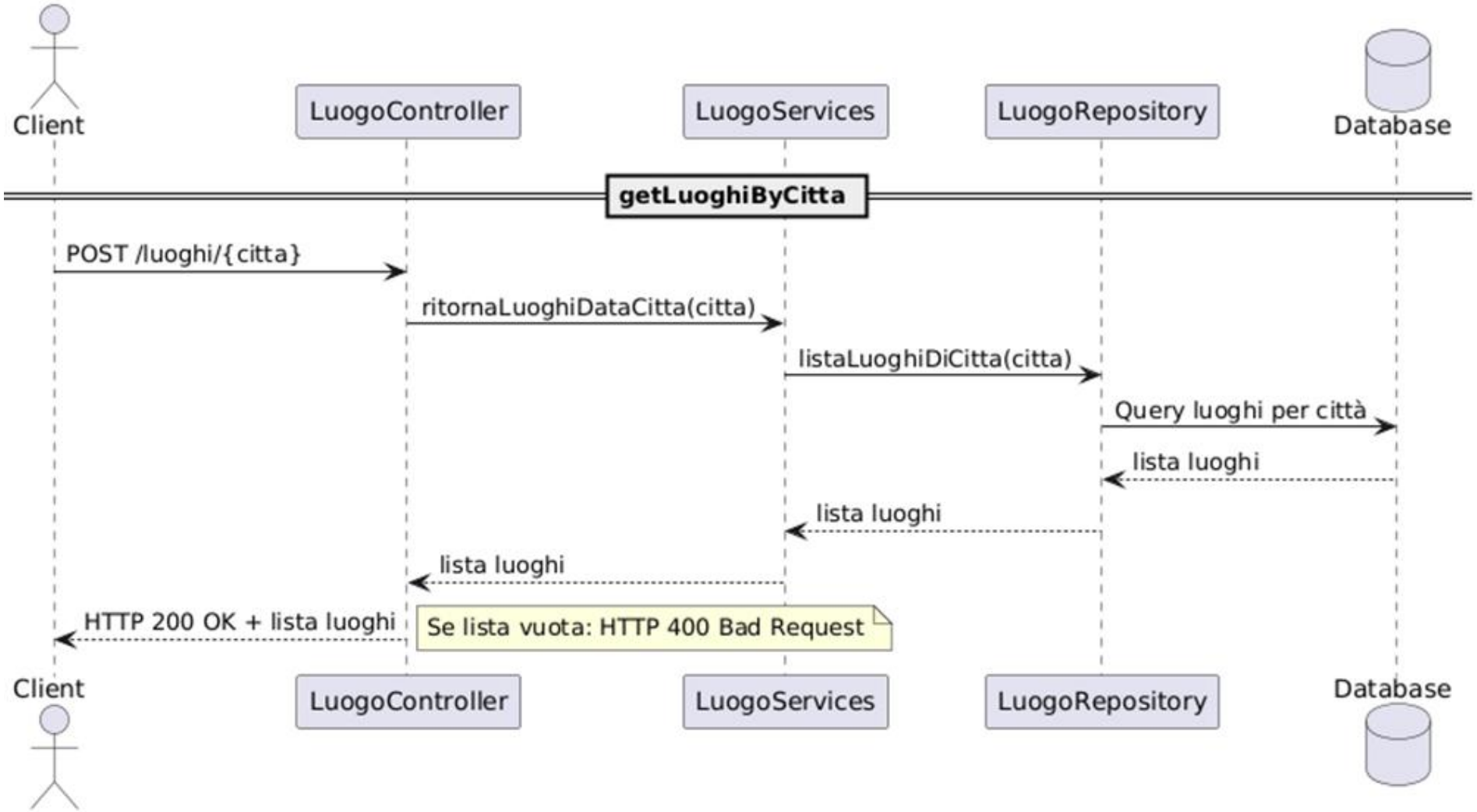
Il controller riceve il nome della città come path variable e chiama *ritornaLuoghiDataCitta* nel service.

Il service a sua volta contatta il repository (l*istaLuoghiDiCitta*), che esegue una query per recuperare tutti i luoghi associati a quella città nel database.

Se il repository trova dei luoghi, il controller risponde al client con HTTP 200 OK e la lista; altrimenti HTTP 400 Bad Request con un messaggio d'errore.

Questo metodo consente all'applicazione di mostrare musei, monumenti, attrazioni o punti d'interesse nella città scelta dall'utente.

getLuoghiByCitta - Sequence diagram



API per la gestione degli itinerari

getNomiItinerarioByUtente

Questa API serve a restituire la lista dei nomi delle mappe (cioè degli itinerari) associati a uno specifico utente.

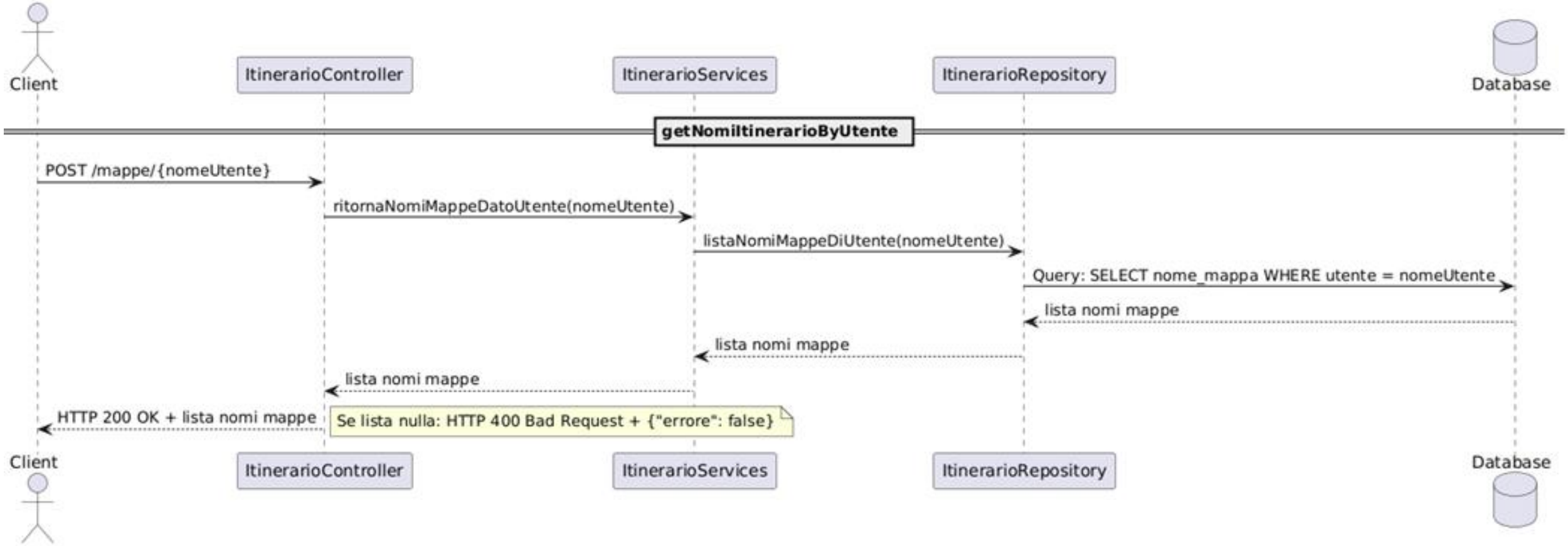
È pensata per permettere all'applicazione client di mostrare la lista di tutti gli itinerari che quell'utente ha già salvato.

Il controller riceve il nome dell'utente come path variable e chiama il metodo *ritornaNomiMappeDatoUtente* nel service.

Il service, a sua volta, si appoggia al repository (*listaNomiMappeDiUtente*), che interroga il database per recuperare tutti gli itinerari associati a quell'utente.

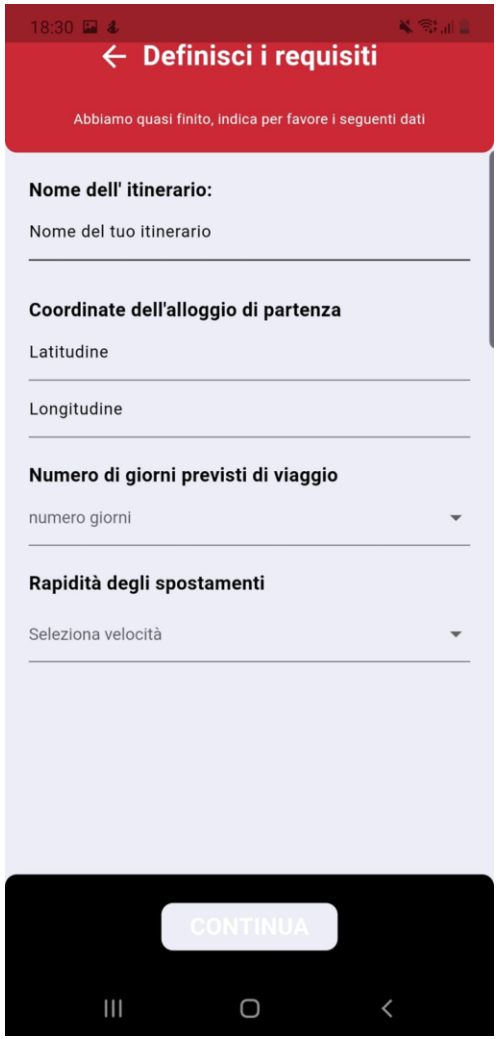
Se il repository trova dei risultati, il controller risponde al client con HTTP 200 OK e restituisce la lista dei nomi; altrimenti, risponde con HTTP 400 Bad Request e un messaggio d'errore.

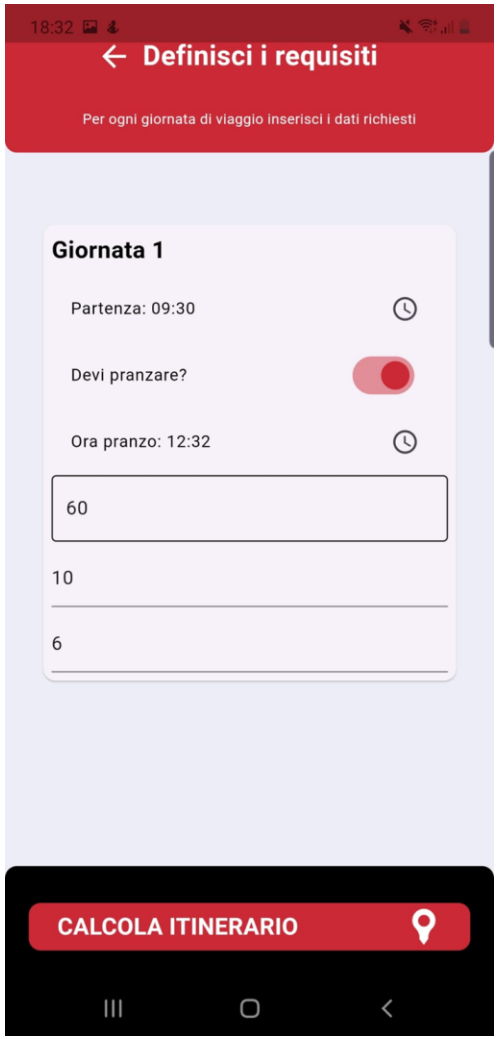
getNomiByUtente - Sequence diagram



ITERAZIONE 3

Implementazione front-end - Requirements-Trip page





API per la selezione di mappe/città/luoghi

Sono state inoltre aggiunte le seguenti API per la selezione dei luoghi:

- **getRistorantiByCoordinate,** che date in input una coppia di coordinate restituisce una lista di ristoranti, da quello più vicino a quello più lontano;
- **getItinerarioByNomeAndUtente,** che dati in input il nome di una mappa già esistente e il nome dell'utente che la ha creata restituisce l'intero itinerario, se questo è presente sul database.

API getRistorantiByCoordinate

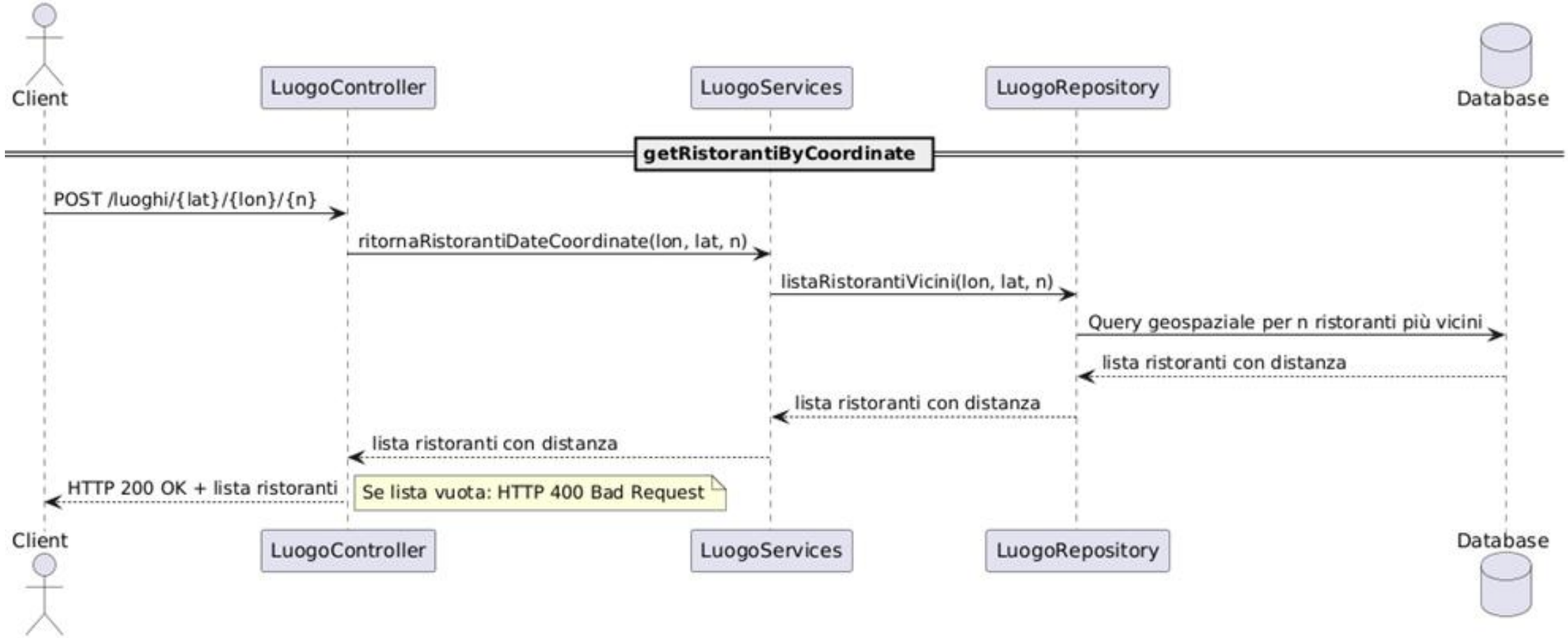
Questo terzo servizio permette all'utente di vedere i ristoranti più vicini a dove si trova o a un punto scelto sulla mappa.

Il controller (*getRistorantiByCoordinate*) passa i parametri necessari al service (*ritornaRistorantiDateCoordinate*). Il service chiama il repository (*listaRistorantiVicini*), che grazie a una query geospaziale calcola quali sono i n ristoranti più vicini.

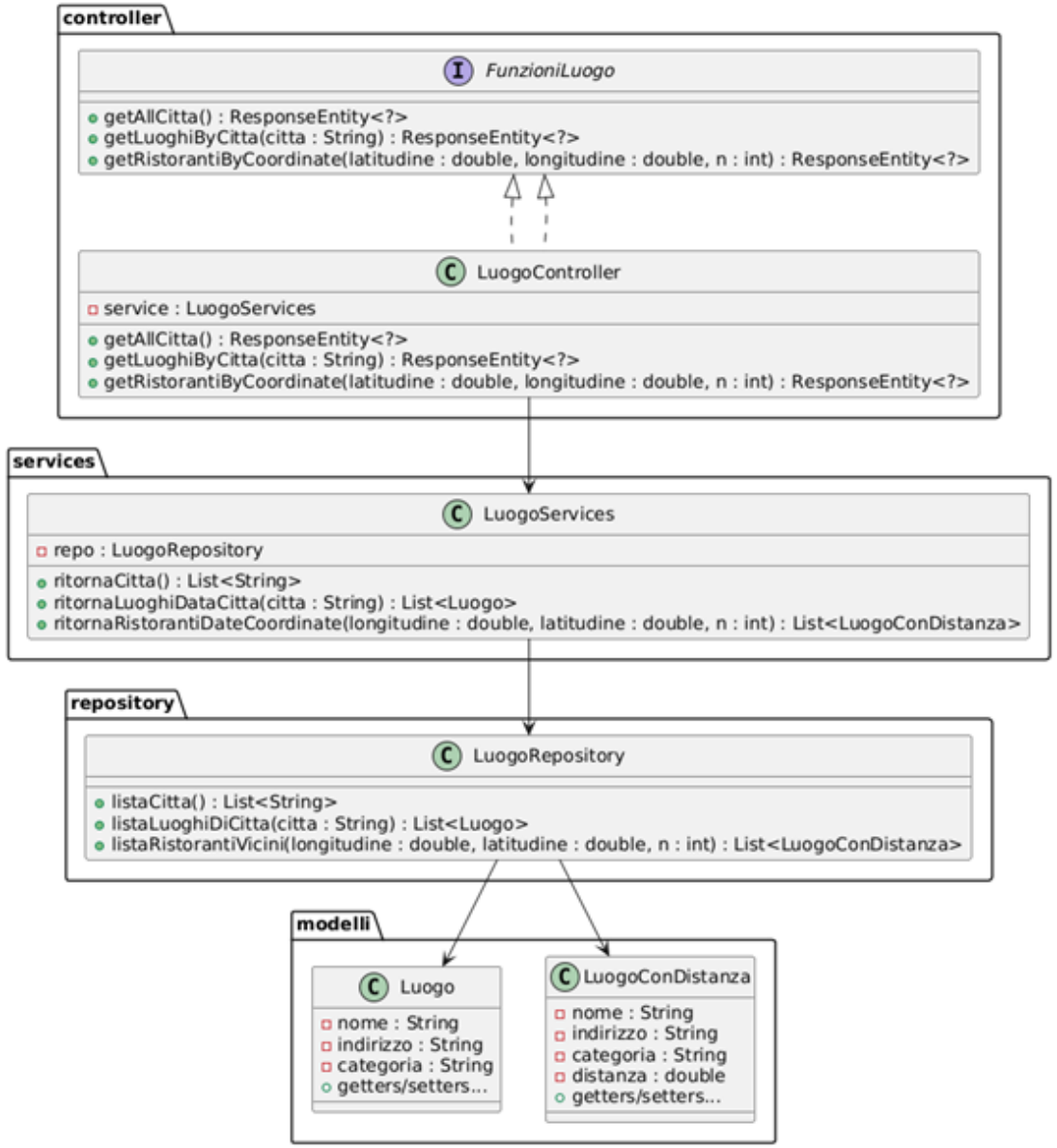
Se ci sono risultati, il controller invia al client HTTP 200 OK con la lista; altrimenti HTTP 400 Bad Request con un messaggio d'errore.

Questo servizio è pensato per chi sta visitando una città e vuole trovare subito dove mangiare nelle vicinanze.

getRistorantiByCoordinate - sequence diagram



API-Luoghi Class diagram



API per la gestione degli itinerari

getItinerarioByNomeAndUtente

Il servizio getItinerarioByNomeAndUtente è un'API che permette di recuperare la mappa dettagliata di un itinerario (composta da tappe ordinate per giorno) dato il nome della mappa e l'username dell'utente proprietario.

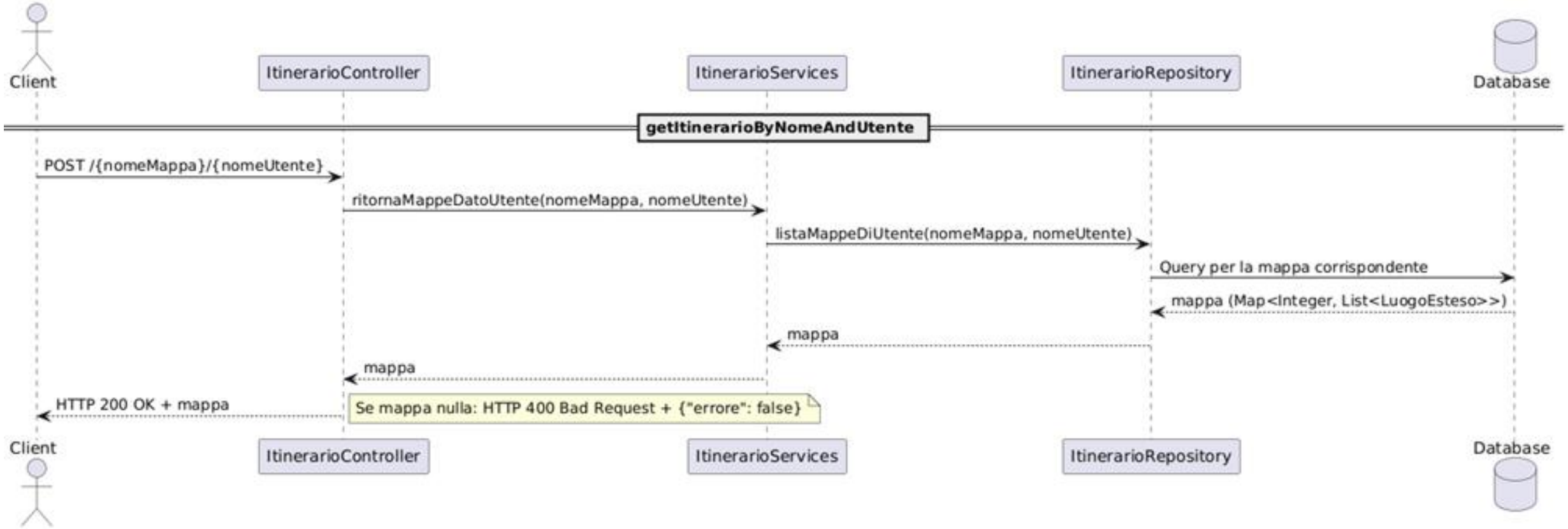
Il controller riceve due parametri come path variable e chiama il metodo *ritornaMappeDatoUtente* nel service.

Il service si occupa di contattare il repository (*listaMappeDiUtente*), che esegue una query al database per trovare la mappa corrispondente all'utente e al nome indicato.

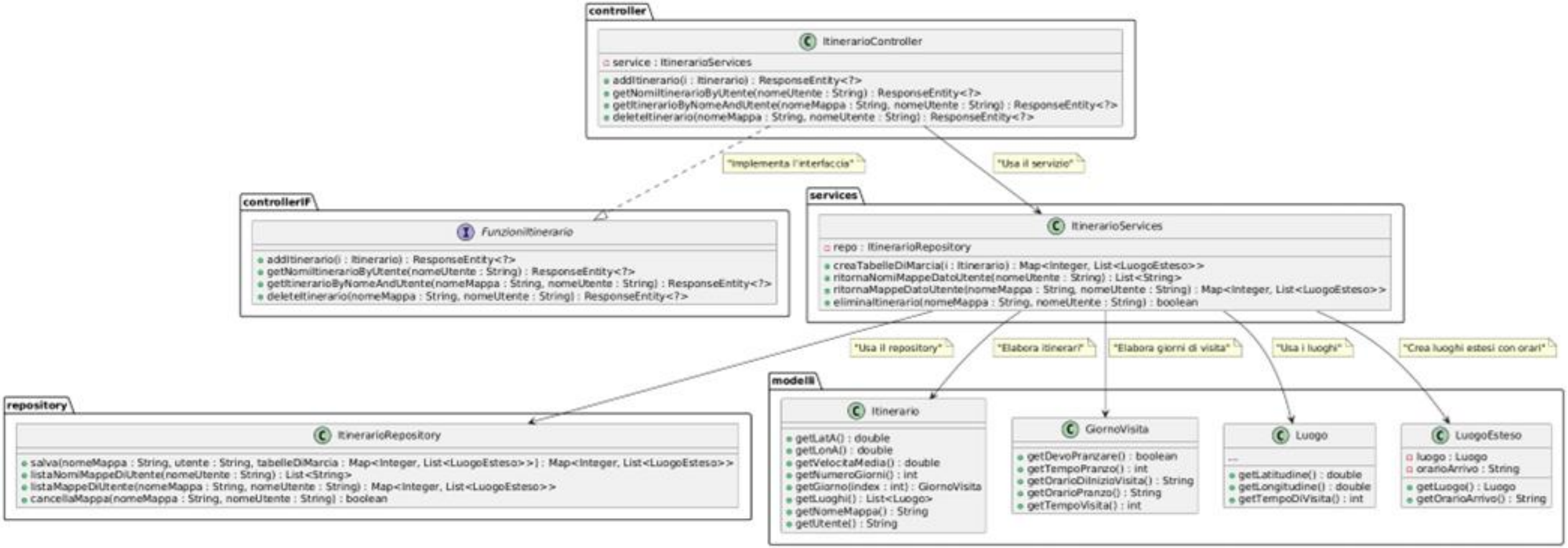
Se la mappa viene trovata, il controller risponde al client con HTTP 200 OK e la mappa stessa.

Se invece non viene trovata, il controller restituisce HTTP 400 Bad Request con un messaggio d'errore.

getItinerarioByNomeAndUtente - sequence diagram



API-Itinerari Class diagram

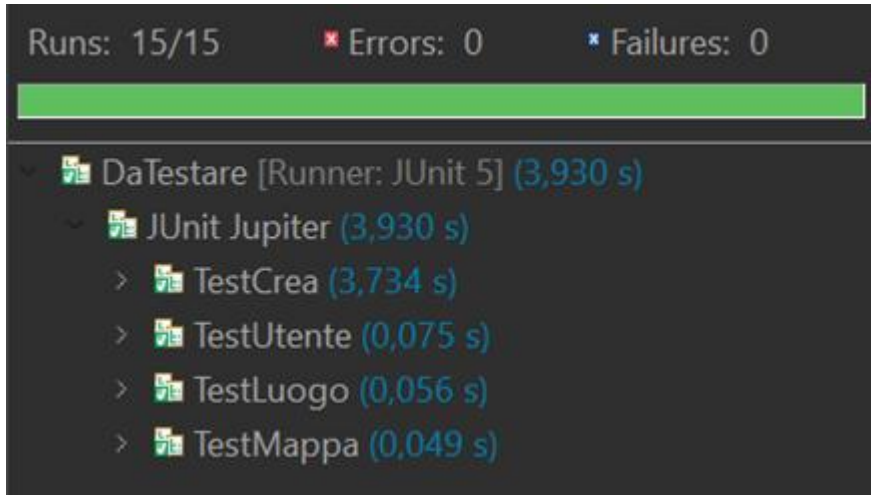


Testing del software

Ogni API è stata testata seguendo tre modalità complementari:

1. verificare "**a occhio**" che l'API svolga correttamente la funzione prevista, osservandone il comportamento durante l'esecuzione;
2. verifica attraverso l'uso di **Postman**, che consente di inviare richieste controllate e osservare le risposte, verificando così anche la gestione dei diversi casi di errore e la correttezza dei JSON inviati e ricevuti.
3. Utilizzo di **JUnit**, dove vengono scritti metodi di test per verificare puntualmente le operazioni di lettura e scrittura che le API eseguono sul database.

Riportiamo i risultati dell'esecuzione in cascata dei test di tutte le query delle API.



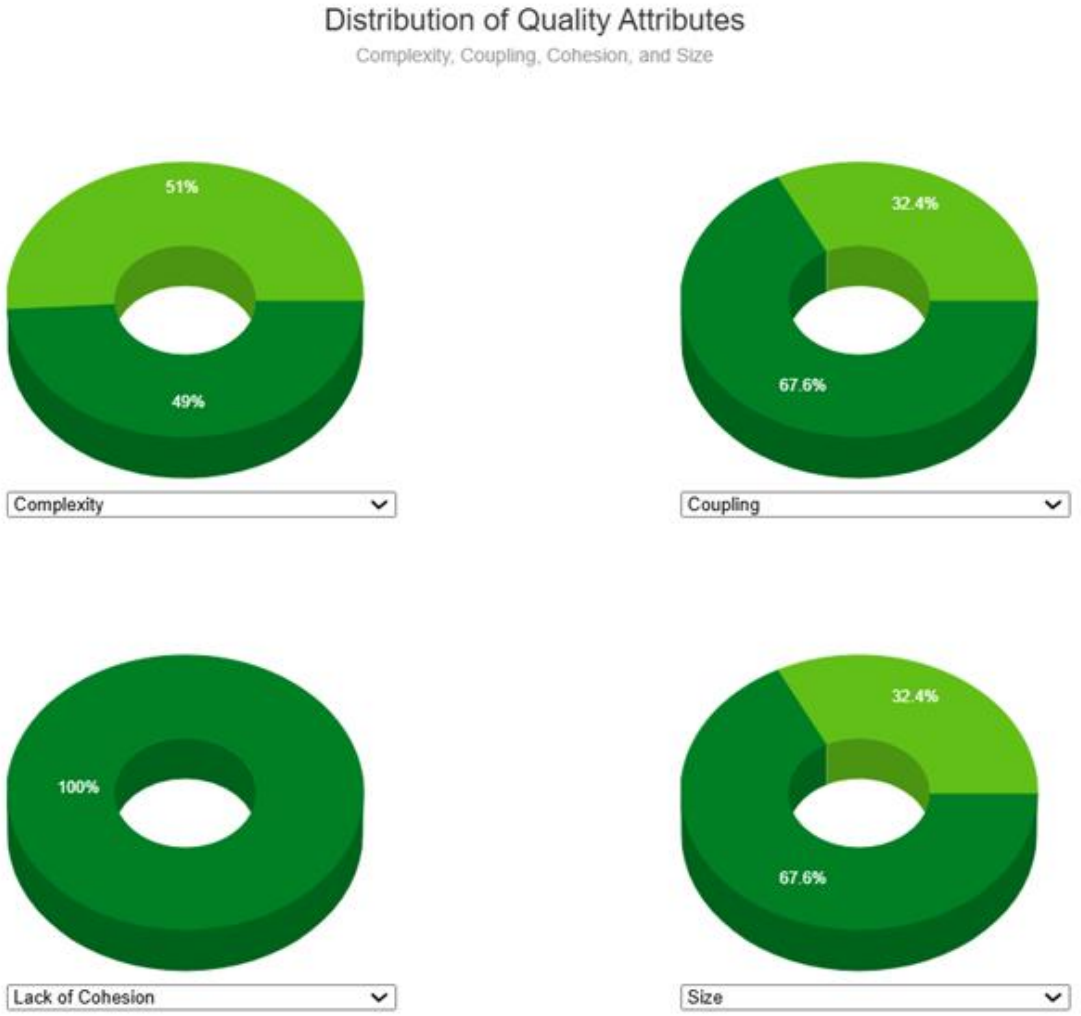
Analisi di qualità e metriche del software

Per la valutazione della qualità del software e l'analisi delle metriche è stato utilizzato lo strumento **CodeMR**

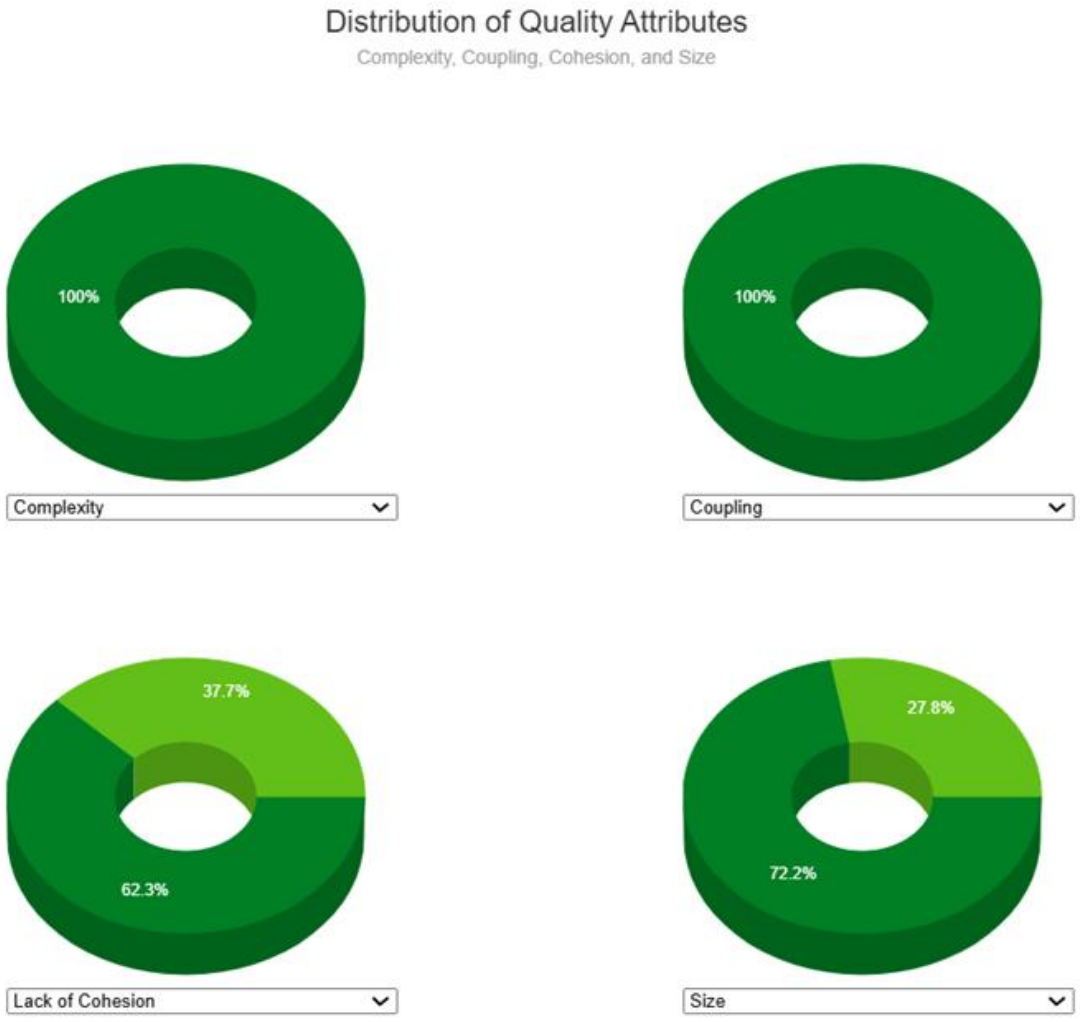
Tra i vari grafici generati riportiamo solo quelli relativi a:

- **complessità,** che misura quanto è difficile comprendere e mantenere il codice;
- **coupling,** che indica il grado di dipendenza tra classi o moduli;
- **l'assenza di coesione,** che evidenzia quanto le responsabilità di una classe siano disperse e poco correlate;
- **dimensione del codice,** che è il numero di righe o metodi.

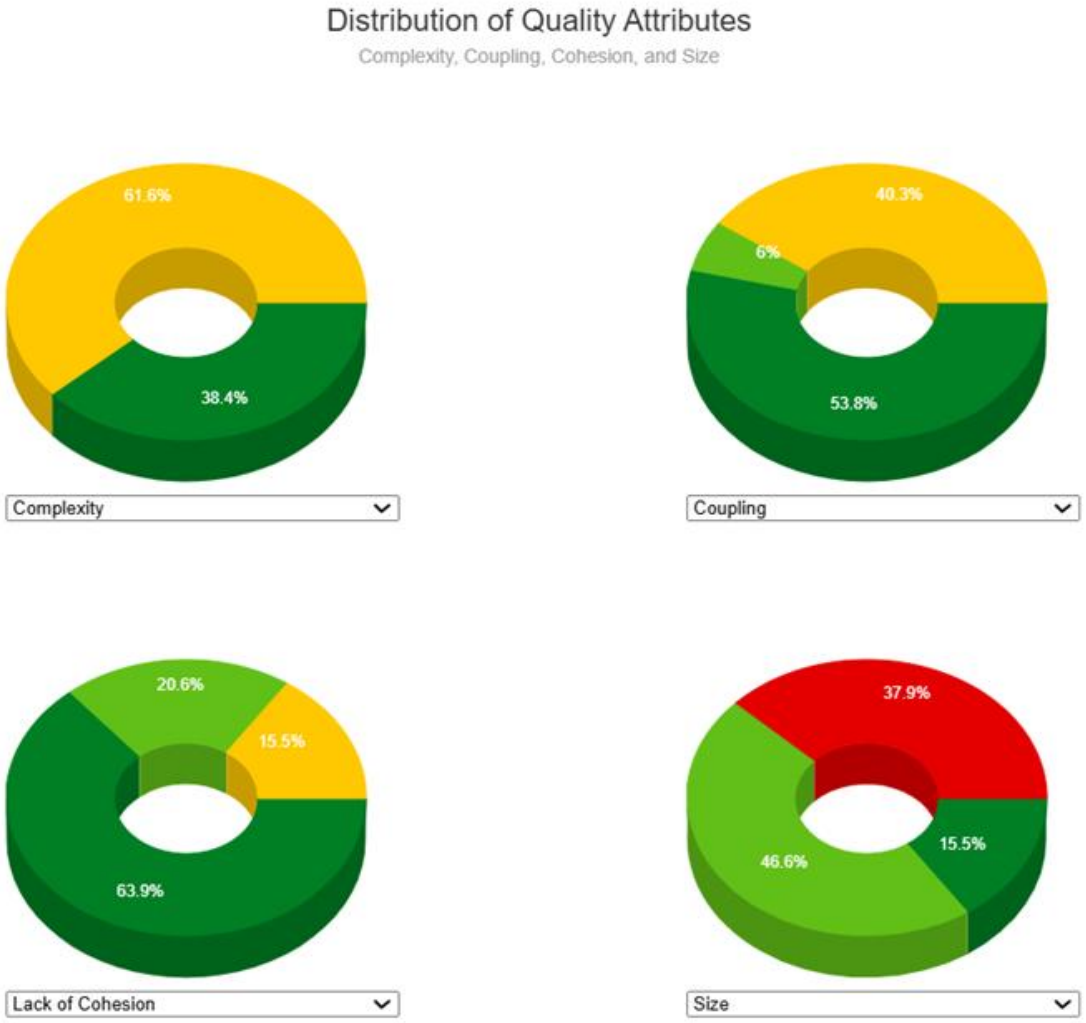
Analisi di qualità e metriche - controller



Analisi di qualità e metriche - model



Analisi di qualità e metriche - database



Conclusioni

L’applicazione è in grado di soddisfare tutti i requisiti funzionali dell’utente finale, generando e memorizzando itinerari personalizzati secondo preferenze e necessità di ognuno.

Unico elemento che nella versione corrente del software presenta una bassa usabilità per l’utente è l’inserimento dei dati riguardanti la posizione di partenza per gli itinerari: inserire manualmente le coordinate non è intuitivo.

Per rimediare a questa problematica sarebbe possibile acquistare ed utilizzare API di Google che permettano di identificare le coordinate di hotel associandole a un nominativo inserito dall’utente.

Negli update futuri, inoltre, si potrebbe comprare l’API di Google-Maps per compiere stime più corrette sulle distanze, fare accordi con i proprietari delle zone di attrazione incentivando l’utilizzo dell’app fornendo un’interfaccia ad hoc per l’inserimento e l’aggiornamento delle zone interessate.