Documentazione progetto

SmarTrip

Componenti:

Filippo Bolis - 1079493

Daniele Gotti - 1079011

Gabriele Mazzoleni - 1079514

Università degli Studi di Bergamo

Laurea Magistrale in Ingegneria Informatica

Corso di Progettazione, Algoritmi e Computabilità

Prof. Patrizia Scandurra

Anno Accademico 2024/2025

Indice

Iterazione 0

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.

L'obiettivo principale di SmarTrip è fornire uno strumento innovativo e intuitivo che consenta di risparmiare tempo e migliorare l'esperienza di viaggio, grazie all'impiego di algoritmi di ottimizzazione e tecnologie moderne.

Lo sviluppo del software sarà guidato da una metodologia agile, scelta per garantire flessibilità ed efficienza nell'implementazione del prodotto finale. Questo approccio prevede la suddivisione del lavoro in iterazioni brevi e continue, permettendo un miglior controllo del processo, una rapida risposta ai cambiamenti e una comunicazione costante tra i membri del team. L'utilizzo della metodologia agile contribuisce a ridurre significativamente i costi e i tempi di sviluppo, assicurando al contempo un software più solido, affidabile e in linea con le reali esigenze degli utenti.

Il prodotto finale sarà composto da due elementi principali: un server e un'applicazione mobile. Il server gestirà la logica applicativa, i dati e l'esecuzione degli algoritmi di ottimizzazione, mentre l'app mobile SmarTrip costituirà l'interfaccia utente, progettata per offrire un'esperienza semplice.

Toolchain

Tool	**Utilizzo**
Eclipse	Utilizzato come ambiente di sviluppo integrato (IDE) per scrivere, eseguire e debuggare il codice Java del server.
Visual Studio Code	Usato come editor principale per lo sviluppo dell'applicazione mobile Flutter.
GitHub	Impiegato per il versionamento del codice e la collaborazione tra i membri del team tramite repository remoti.
Java	Linguaggio utilizzato per implementare la logica del server, inclusi i controller e i servizi.
Maven	Strumento di build e gestione delle dipendenze per i progetti Java del back-end.
Spring Boot	Framework scelto per semplificare lo sviluppo del server e la gestione delle API REST.
jOOQ	Utilizzato per interagire con il database in modo typesafe e generare codice SQL a partire dallo schema.
JGraphT	Libreria per rappresentare e manipolare strutture grafiche (nodi, archi, percorsi), fondamentali per il calcolo degli itinerari ottimizzati.

DbBrowser for SQLite	Usato per visualizzare, modificare e gestire manualmente il contenuto del database SQLite.
CodeMR	Strumento utilizzato per l'analisi della qualità del codice e delle metriche di complessità del progetto Java.
JUnit 5	Framework impiegato per scrivere ed eseguire i test automatici delle funzionalità del server.
Postman	Usato per testare le API REST esposte dal server durante le fasi di sviluppo e debugging.
Flutter	Framework scelto per realizzare l'interfaccia grafica dell'app mobile in modo cross-platform.
Dart	Linguaggio utilizzato per implementare la logica dell'app mobile sviluppata con Flutter.
Draw.io	Usato per creare diagrammi UML e schemi architetturali da inserire nella documentazione del progetto.
Microsoft Word	Utilizzato per redigere la documentazione tecnica e descrittiva del progetto.
Microsoft PowerPoint	Usato per realizzare la presentazione finale da esporre in sede di discussione del progetto.

Analisi dei requisiti

L'identificazione dei requisiti ha richiesto un'attenta riflessione sulle reali necessità degli utenti finali, con l'obiettivo di progettare un sistema che rispondesse in modo efficace e mirato ai loro bisogni. Per farlo, è stato necessario considerare diversi contesti d'uso e ipotizzare situazioni concrete in cui il software sarebbe stato impiegato. Questo processo ha permesso di delineare in modo chiaro sia i requisiti funzionali che quelli non funzionali, costituendo una base solida per le fasi successive di progettazione e sviluppo.

Requisiti dell'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, ottimizzando il percorso tra una tappa e l'altra.
- L'utente deve poter scegliere tra una lista di ristoranti disponibili più vicini al punto in cui effettuerà pause pranzo.

Requisiti dell'addetto dell'amministrazione cittadina

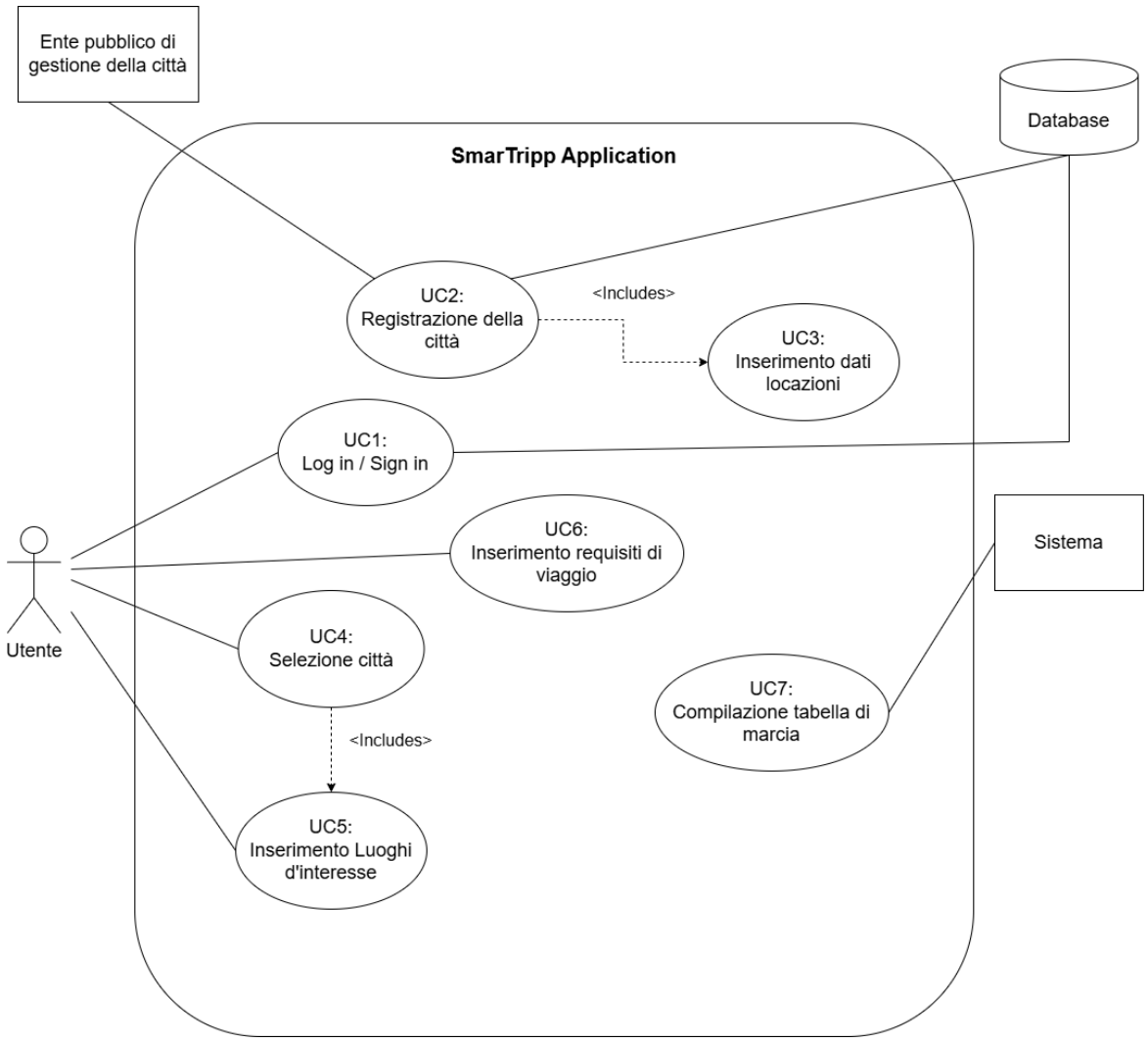
- L'addetto deve essere in grado di inserire con facilità la sua città e i luoghi visitabili.
- L'addetto deve essere in grado di inserire per ogni luogo tutte le caratteristiche utili alla costruzione di un itinerario di viaggio.

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. L'app mobile per l'amministratore non verrà implementata, ma sarà possibile usufruire di luoghi default inseriti nel database nella fase di sviluppo software. Questa scelta è anche dovuta al fatto che esistono già API a pagamento che consentono di prendere caratteristiche dei luoghi, rendendo ridondante la creazione di un'applicazione apposita.

Casi d'uso

Use case diagram

Il diagramma seguente illustra in modo dettagliato tutti i requisiti funzionali che il sistema dovrà implementare, ad eccezione dei casi d'uso del gestore della città che, come già anticipato, non saranno implementati. Ogni funzionalità rappresentata riflette un comportamento atteso dell'applicazione, definendo ciò che l'utente potrà fare e come il sistema dovrà reagire alle diverse interazioni.



Descrizione dei casi d'uso

- UC1: LOG IN / SIGN IN. L'utente accede all'applicazione, inserendo una mail, un username e una password. Qualora la mail sia già in uso, viene restituito all'utente un messaggio d'errore.
  POSTCONDIZIONI: l'utente può eseguire nuovamente l'accesso all'app inserendo solamente username e password.

- UC2: REGISTRAZIONE DELLA CITTA'. L'apposito ente incaricato registra la città all'interno del database dell'applicazione.
  POSTCONDIZIONI: la città diventa disponibile alla selezione da parte degli utenti.

- UC3: INSERIMENTO DATI LOCAZIONI. L'ente inserisce i dati relativi ai punti d'interesse che intende rendere visitabili attraverso l'applicazione, inserendo informazioni riguardo a orari di apertura, prezzi etc.
  POSTCONDIZIONI: le locazioni saranno disponibili per essere selezionate dagli utenti, e potranno essere utilizzate dal sistema per calcolare i percorsi migliori.

- UC4: SELEZIONE CITTA'. L'utente seleziona la città che desidera visitare, tra quelle che hanno precedentemente aderito all'applicazione.
  POSTCONDIZIONI: viene proposta all'utente la scelta sui luoghi d'interesse da visitare.

- UC5: INSERIMENTO LUOGHI D'INTERESSE. L'utente compila una lista di luoghi che gli interessa visitare; questa lista può essere modificata in seguito e deve essere dunque conservata.

- UC6: INSERIMENTO REQUISITI DI VIAGGIO. L'utente inserisce una serie di dati relativi alle specifiche temporali del suo viaggio, quali data di arrivo, data di ritorno e a preferenze su ritmi di visita.

- UC7: COMPILAZIONE TABELLA DI MARCIA. Il sistema, basandosi sulle informazioni inserite dall'utente, calcola il percorso ottimale per permettere la visita del numero maggiore di luoghi d'interesse nei tempi e nei ritmi forniti.
  PRECONDIZIONE: per poter avviare questo caso d'uso, l'utente deve aver inserito almeno un luogo d'interesse e aver compilato i requisiti di viaggio.

Analisi delle specifiche

Le specifiche funzionali sono state suddivise in due livelli di priorità: alta e bassa. Le funzionalità assegnate alla priorità alta comprendono le operazioni essenziali per il corretto funzionamento del sistema. Quelle a priorità media includono caratteristiche utili ma non indispensabili, mentre la priorità bassa raccoglie le funzionalità opzionali, che potranno essere introdotte in versioni successive dell'applicazione.

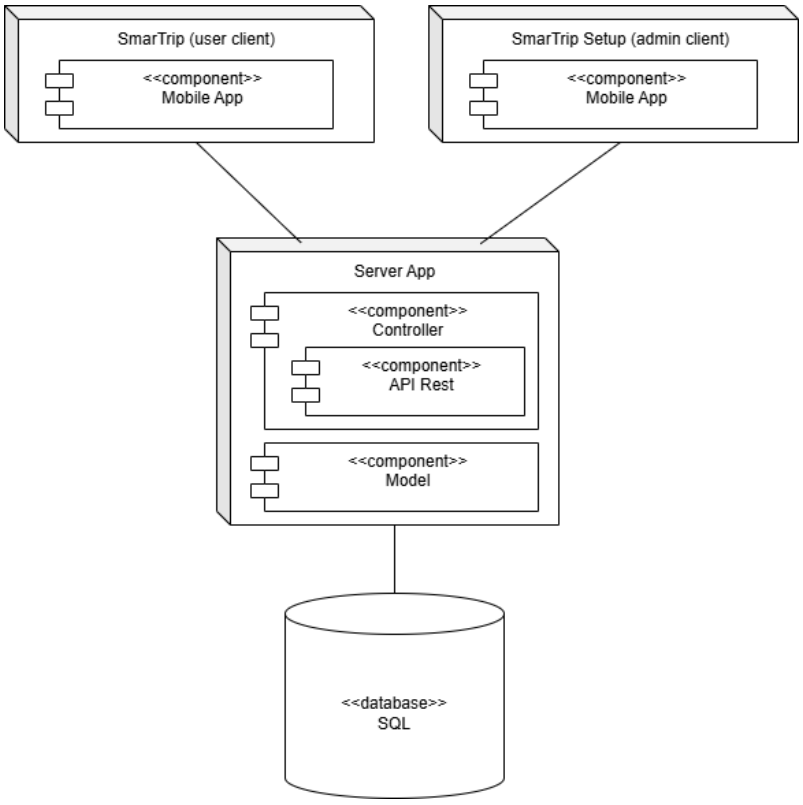
Specifica	**Priorità**	**Implementato**
Sign in	Alta	Sì
Log in	Alta	Sì
Visualizzazione città disponibili	Alta	Sì
Visualizzazione luoghi disponibili della città selezionata	Alta	Sì
Inserimento dati del viaggio	Alta	Sì
Algoritmo ottimizzazione viaggio	Alta	Sì
Visualizzazione mappe precedenti	Bassa	Sì
Inserimento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

Analisi dell'architettura

Per garantire modularità, manutenibilità e scalabilità, il progetto è stato progettato seguendo un'architettura ben definita, in grado di separare logicamente le diverse componenti del sistema. In questa sezione vengono illustrati i principali aspetti architetturali attraverso due diagrammi.

Deployment diagram

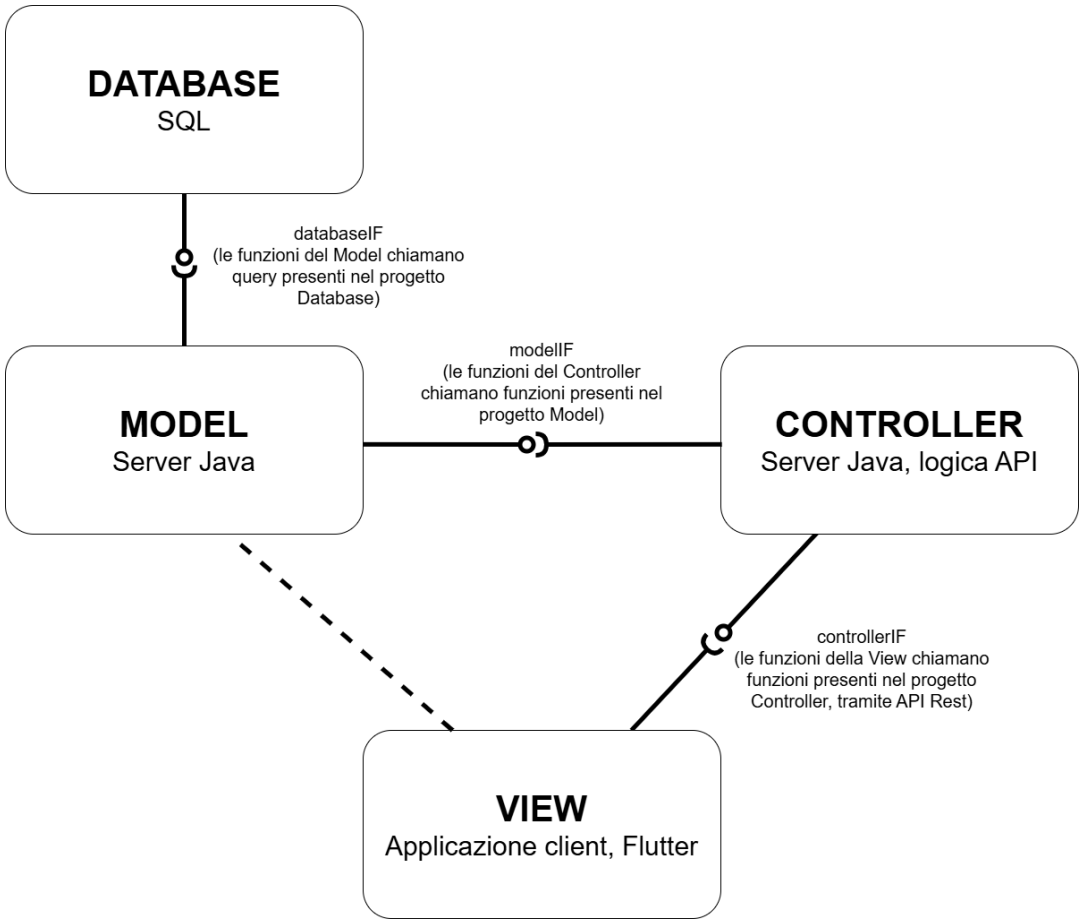
Il diagramma rappresenta la distribuzione fisica delle componenti software e le modalità con cui queste interagiscono all'interno dell'infrastruttura. In particolare, si evidenzia la separazione tra l'applicazione mobile, che funge da interfaccia utente, e il server, responsabile della logica applicativa. Quest'ultimo espone le API necessarie alla comunicazione con il client e gestisce i modelli dei dati, che vengono recuperati e manipolati attraverso un database locale. Tale organizzazione permette una chiara divisione delle responsabilità e facilita l'evoluzione indipendente delle diverse componenti del sistema.



MVC diagram

Il diagramma MVC (Model-View-Controller) illustra la suddivisione logica delle responsabilità all'interno del sistema. Nel nostro progetto è stata adottata una variante semplificata del modello tradizionale, strutturata come segue:

Alla base si trova il Database, che rappresenta la fonte persistente dei dati. Il livello Model, situato sul server, contiene le logiche di accesso ai dati e interagisce con il database tramite funzioni che chiamano le query presenti nel progetto Database. Il Controller, anch'esso parte del server, gestisce la logica applicativa e si occupa di elaborare le richieste ricevute, richiamando le funzioni del Model. Infine, la View è rappresentata dall'app mobile, che comunica con il Controller attraverso chiamate API. Questa separazione consente una gestione ordinata del flusso dati e favorisce la scalabilità e la manutenibilità del sistema.



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.
  POSTCONDIZIONI: l'utente può eseguire nuovamente l'accesso all'app inserendo username e password.

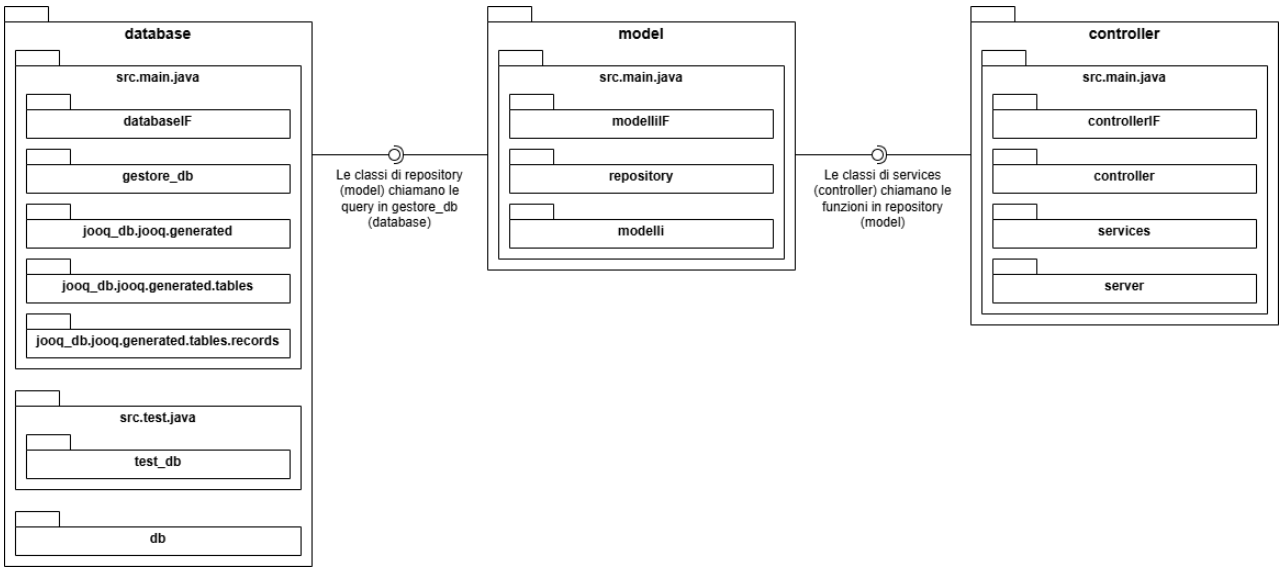
- UC2 e UC3. I presenti casi d'uso restano invariati, è stata presa la decisione di non implementarli nella presente versione dell'app; il ruolo di "Ente" verrà ricoperto dal team di sviluppo, che si occuperà di popolare il database dei luoghi.

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

Architettura del server

In questa iterazione ci si concentra sulla progettazione e sulla scrittura del codice, partendo dal progetto server Java in Eclipse IDE.

Package diagram



Come mostra il diagramma dei pacchetti, l'applicazione è suddivisa in tre progetti Maven separati: `database`, `model` e `controller`, collegati tra loro tramite dipendenze dichiarate nel `pom.xml`.

- `database`: contiene il pacchetto `gestore_db`, che include le classi responsabili della creazione del database e dell'esecuzione delle query (utilizzate dalle API); il pacchetto `databaseIF`, con le interfacce delle classi di `gestore_db`; i pacchetti generati automaticamente da jOOQ; `test_db`, per i test JUnit delle query; e infine `db`, che conterrà i

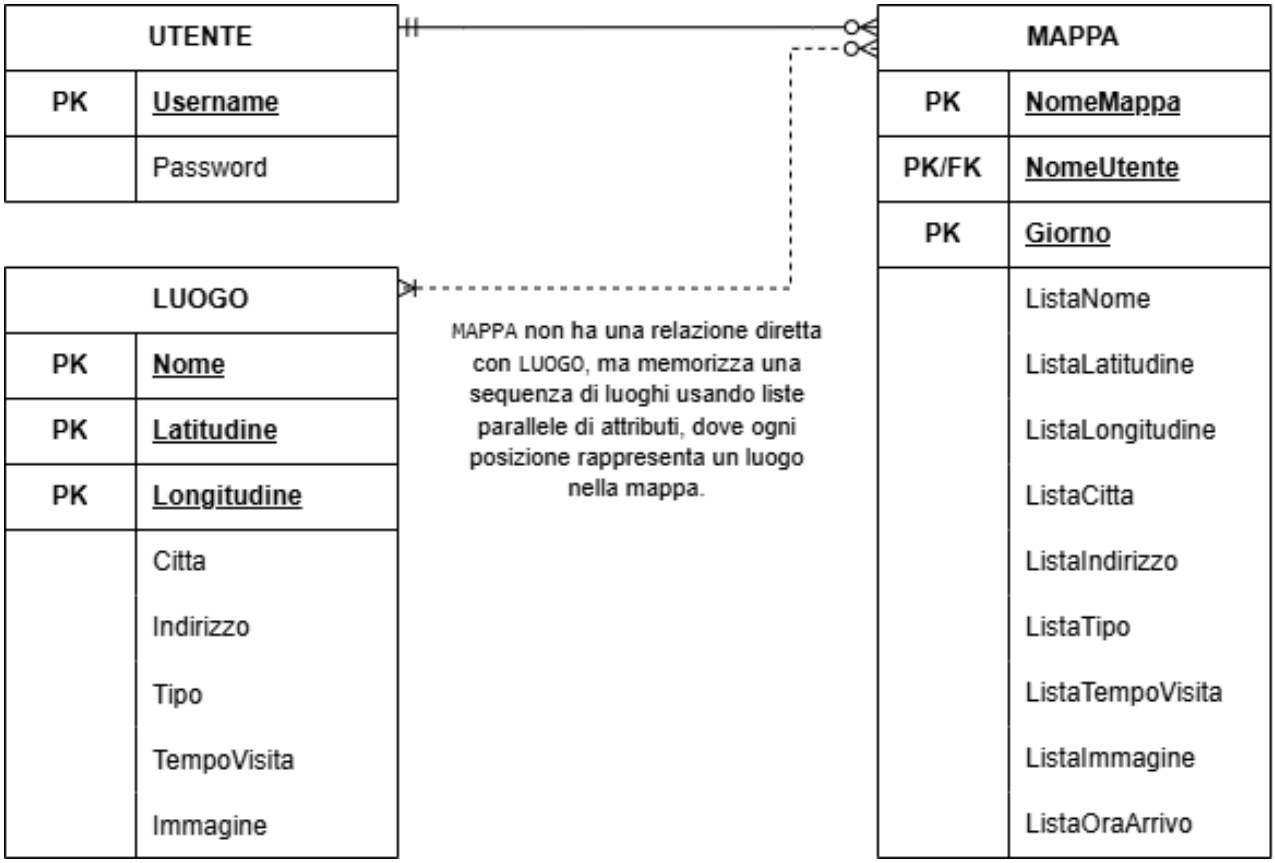
file del database principale e di quello di test, al fine di evitare la modifica involontaria dei dati originali.

- `model`: comprende i pacchetti `modelli`, che contengono le classi che rappresentano gli oggetti del dominio (ad esempio Utente, Luogo, ecc.); `repository`, con le classi annotate con `@Repository` di Spring Boot, le quali si interfacciano con il database invocando le query; e `modelliIF`, che contiene le interfacce dei repository.

- `controller`: include `services`, con le classi annotate `@Service` di Spring Boot, responsabili della logica delle API e dell'interazione con il progetto `model`; `controller`, con le classi annotate `@RestController`, che gestiscono le richieste HTTP e l'invio delle risposte API; `controllerIF`, con le interfacce dei controller; e infine `server`, che conterrà la classe per l'avvio del server.

Database

Il database utilizzato per il progetto è un file locale denominato `db.db3`, realizzato con SQLite. La sua struttura logica è stata progettata seguendo un modello Entità-Relazione (ER), che evidenzia le tre entità principali, insieme alle loro relazioni.



Il database contiene inoltre alcuni dati di prova, tra cui luoghi situati a Bergamo e Milano, inseriti per permettere all'utente di utilizzare immediatamente l'applicazione e testarne le funzionalità.

A livello di codice, l'accesso al database è gestito dalla classe `DatabaseManager`, che implementa il *Singleton pattern.* Il questo design pattern garantisce che esista un'unica istanza della classe per tutta la durata dell'esecuzione dell'applicazione. Ciò è utile per evitare conflitti tra più connessioni concorrenti al database e per centralizzare la configurazione e la gestione delle risorse. Inoltre, la classe prevede anche la possibilità di eseguire operazioni in modalità di test, utilizzando un database alternativo.

Le operazioni di creazione, popolamento e interrogazione del database sono delegate a classi separate, anch'esse incapsulate nel `DatabaseManager`. In particolare, le interrogazioni alle tabelle sono realizzate attraverso le classi `QueryUtente`, `QueryLuogo` e `QueryMappa`, che utilizzano la libreria **jOOQ** per costruire query SQL in modo typesafe. Questa scelta consente di scrivere query robuste e meno soggette a vulnerabilità come le SQL injection.

Testing del 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`.

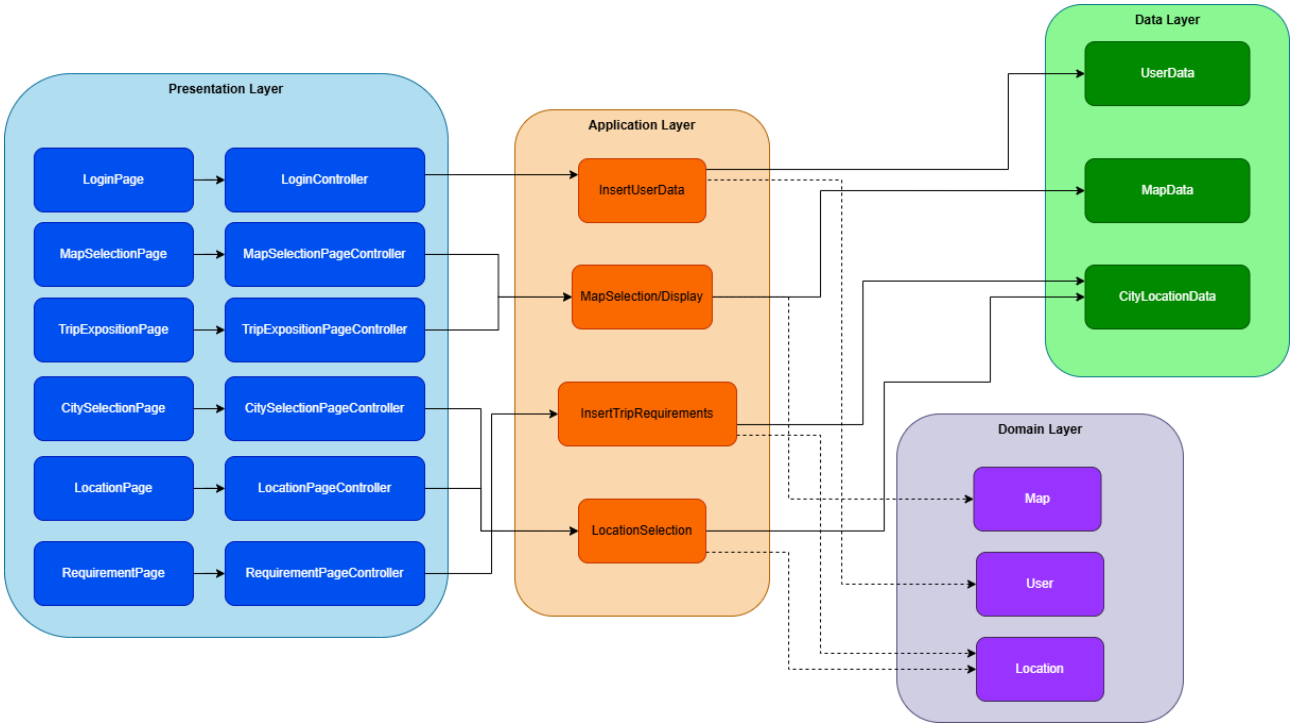
I test sono sviluppati utilizzando JUnit 5, il framework di riferimento per il testing in Java. Ogni API che interagisce con il database, e quindi ciascuna query implementata nelle classi `QueryUtente`, `QueryLuogo` e `QueryMappa`, viene testata mediante apposite funzioni di test contenute nelle rispettive classi.

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. I risultati di questa campagna di test verranno poi riportati e discussi nei capitoli finali, dove saranno analizzati sia gli esiti positivi sia le eventuali problematiche riscontrate.

Progettazione dell'architettura dell'applicazione client

In parallelo alla definizione dell'architettura server è stata definita in via preliminare anche la struttura dell'applicazione utente, da realizzare in Flutter (kit di sviluppo software basato sul linguaggio di programmazione Dart).

La struttura iniziale dei file dell'applicazione è stata definita attraverso il seguente diagramma Riverpod.



Questo grafico suddivide la struttura dell'applicazione in 4 livelli differenti:

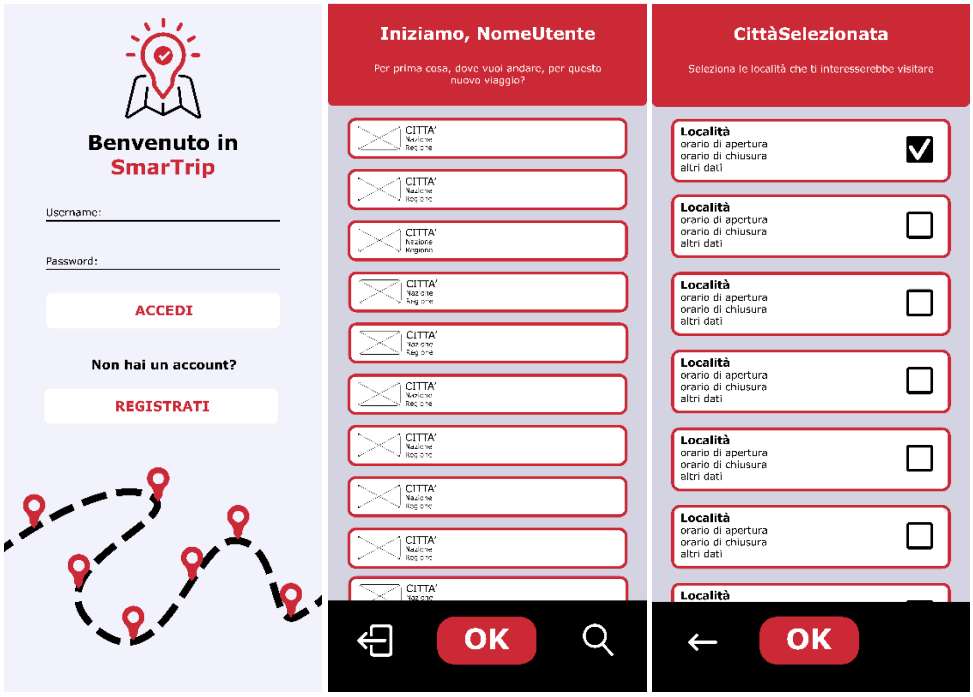
- Presentation Layer: insieme delle classi rappresentanti le pagine dell'app e dei controller logici ad esse associate;
- Application Layer: business logic dell'applicazione, cioè generalizzazione dei casi d'uso;
- Domain Layer: insieme delle strutture dati previste per il funzionamento dell'applicazione;
- Data Layer: Repository/tabelle di database alle quali l'app fa riferimento per il proprio funzionamento.

I contenuti dell'architettura sono spiegati nel dettaglio nella sezione Iterazione 2 in quanto questo schema è stato successivamente ampliato alla definizione di nuovi casi d'uso e nuove necessità.

Progettazione User Interface

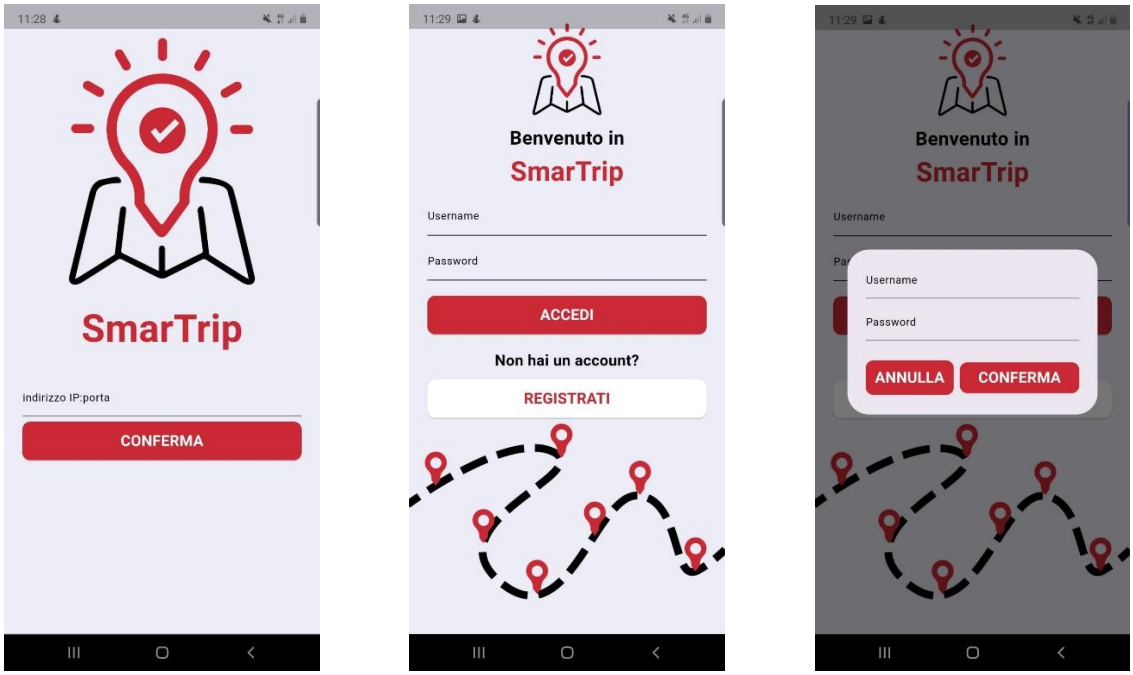
La progettazione grafica dell'interfaccia utente è stata realizzata su Adobe Illustrator; trattasi di una bozza generale per definire lo stile dell'applicazione e la disposizione degli elementi nella pagina.

Queste versioni hanno subito delle modifiche durante la realizzazione in Flutter.



Implementazione front-end

Per la prima iterazione di questo progetto è stato deciso di implementare le funzioni di log-in e sign-in dell'applicazione, con l'obiettivo di verificare il corretto collegamento tra componenti view e controller.
Siccome il server SpringBoot non è al momento disponibile su una macchina fissa (e dunque è possibile che l'indirizzo IP vari tra un avvio e l'altro) è stato necessario introdurre una pagina iniziale in cui inserire l'indirizzo IP e la porta internet del server a cui collegarsi.



Da sinistra a destra: splashPage, loginPage e loginPage (con attivo il modulo per la registrazione all'app).

API-Utente

All'interno dell'architettura di SmarTrip, il modulo di gestione degli utenti rappresenta il punto d'ingresso per tutte le operazioni relative all'autenticazione e registrazione degli utenti dell'app mobile.

Questa funzionalità è realizzata 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.
- **LogIn**: verifica delle credenziali di un utente esistente per consentire l'accesso all'applicazione.

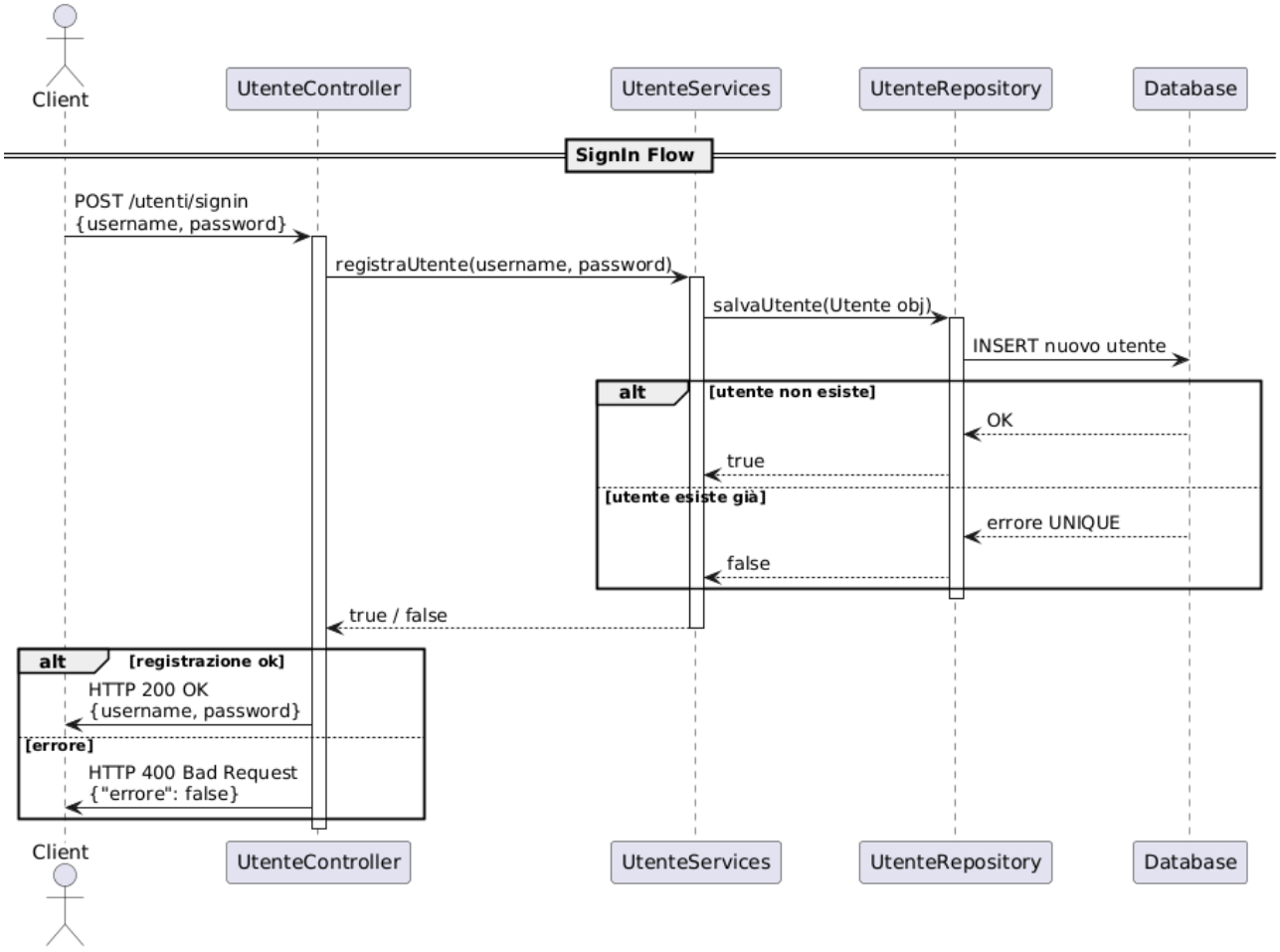
SignIn

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.

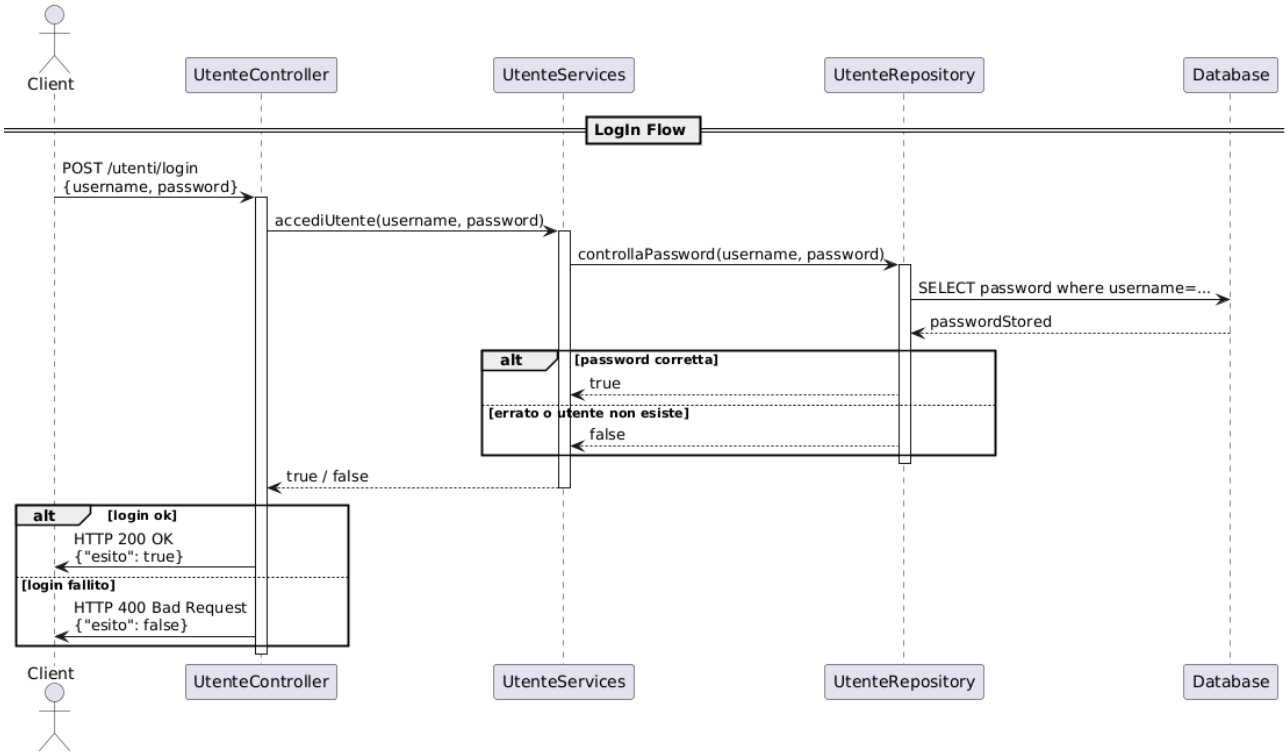


LogIn

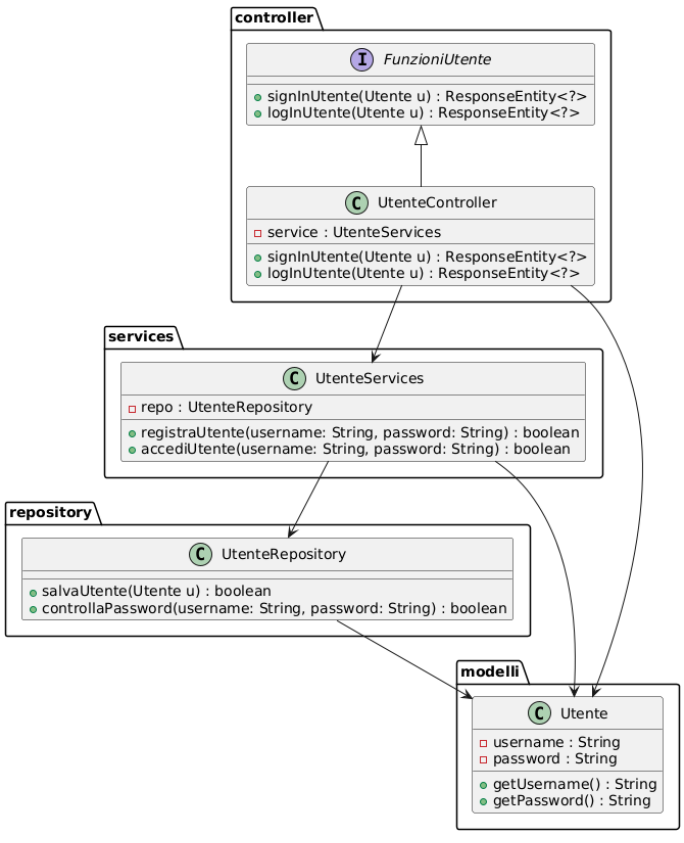
Permette ad un utente già registrato di effettuare il login nell'app.

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.



Entrambe le API poggiano sulla stessa struttura delle classi:



API-Itinerario
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:

```
if (itinerario è nullo oppure itinerario.luoghi è vuoto):
    return (mappa vuota);

grafo = nuovo grafo pesato;

nodoAlloggio = nuovo luogo con coordinate dell'alloggio;
grafo.add(nodoAlloggio);

for each luogo in itinerario.luoghi:
  grafo.add(luogo);

for (i da 0 a numero dei luoghi - 1){
   for j da i+1 a numero dei luoghi:
      luogo1 = itinerario.luoghi[i]
      luogo2 = itinerario.luoghi[j]

      distanza = calcolaDistanza(luogo1, luogo2);
      tempoPercorrenza = calcolaTempoPercorrenza(distanza,velocità);

      if (non esiste ancora un arco tra luogo1 e luogo2):
         grafo.addArco(luogo1,luogo2);

      grafo.setPeso(arco tra luogo1 e luogo2, tempoPercorrenza);
}
tabellaDiMarcia = nuova mappa vuota

for (nGiorno da 0 a itinerario.giorni.size - 1) {
```

```
giornoAttuale = itinerario.getGiorno(nGiorno-1);
tempoInizio = convertiInSecondi(giornoAttuale.Orario_di_inizio_visita);
devePranzare = giornoAttuale.deve_pranzare;
pranzoTroppoLungo = false;
orarioPranzo = convertiInSecondi(giornoAttuale.Orario_di_pranzo);
tempoCorrente = tempoInizio;
percorso = new lista di luoghi;
percorso.add(nodoAlloggio,tempoCorrente);
nodoCorrente = nodoAlloggio;

if(rimasto solo il nodoAlloggio nel grafo):
   break;

while (true){

    prossimoLuogo = nodo più vicino a nodoCorrente non ancora visitato e presente nel grafo che
    minimizza (tempoPercorrenza + tempoDiVisita);
   if (prossimoLuogo.isEmpty()):
      break;

   luogoScelto = prossimoLuogo;
   tempoPercorrenza = grafo.getPeso(nodoCorrente,luogoScelto);

   if (devePranzare && tempoCorrente >= orarioPranzo):
      if (tempoCorrente + durataPranzo < mezzanotte):
         percorso.add(ricercaRistorante,tempoCorrente);
         tempoCorrente += durataPranzo;
         devePranzare = false;
      else:
         pranzoTroppoLungo = true;
    if (tempoCorrente + tempoPercorrenza + luogoScelto.TempoDiVisita >
giornoAttuale.orarioFineVisite || pranzoTroppoLungo):
      break;
```

```
        tempoCorrente += tempoPercorrenza;
        percorso.add(luogoScelto,tempoCorrente);
        tempoCorrente += luogoScelto.TempoPerVisitare;
        nodoCorrente = luogoScelto;
    }
    if (nodoCorrente diverso da nodoAlloggio):
        tempoCorrente += grafo.getPeso(nodoCorrente,nodoAlloggio);
    percorso.add(nodoAlloggio,tempoCorrente);

    tabellaDiMarcia.put(nGiorno,percorso);

    for (luogo in percorso) {
        if (luogo diverso da nodoAlloggio):
            grafo.rimuoviNodo(luogo);
    }
}

return repository.salva(itinerario.NomeMappa, itinerario.Utente, tabellaDiMarcia);
```

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:**
   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)

2. **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) = $O(N^2)$

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

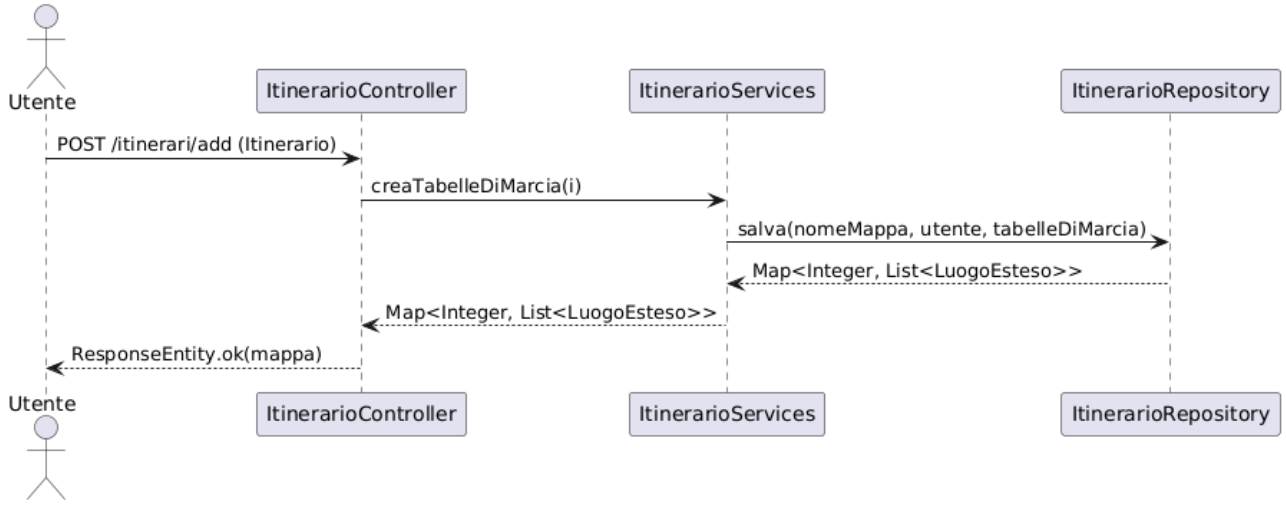
- Costruzione grafo: $O(N^2)$
- Per ogni giorno: $O(N^2)$
- Numero di giorni: G

$O(N^2 + G \cdot N^2) = O(G \cdot N^2)$ (termine $G \cdot N^2$ domina se G cresce)

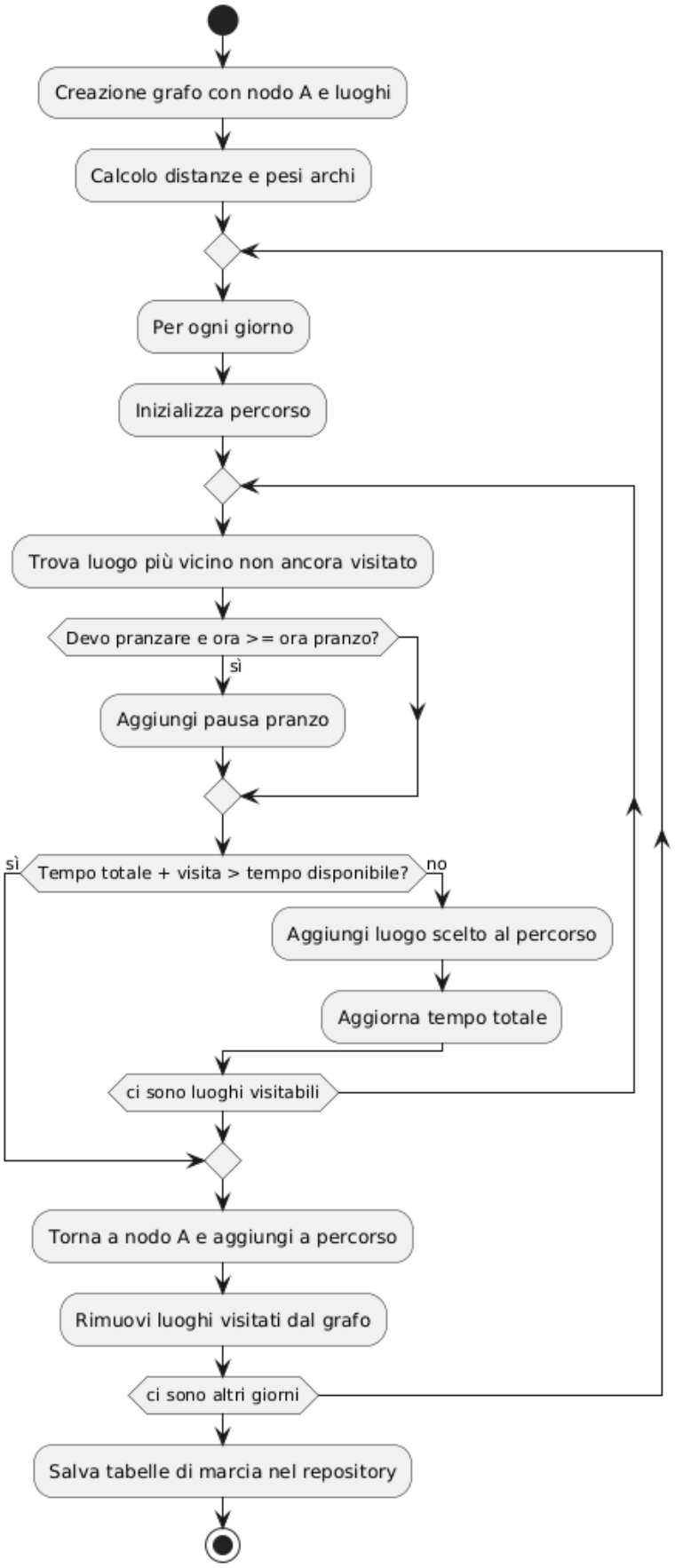
La complessità Spaziale totale dell'algoritmo è data da:

- 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.)

$O(N^2 + G \cdot N) = O(N^2)$ (il termine N^2 domina per N grande)



Per comprendere meglio il funzionamento dell'API è stato costruito un diagramma di attività:



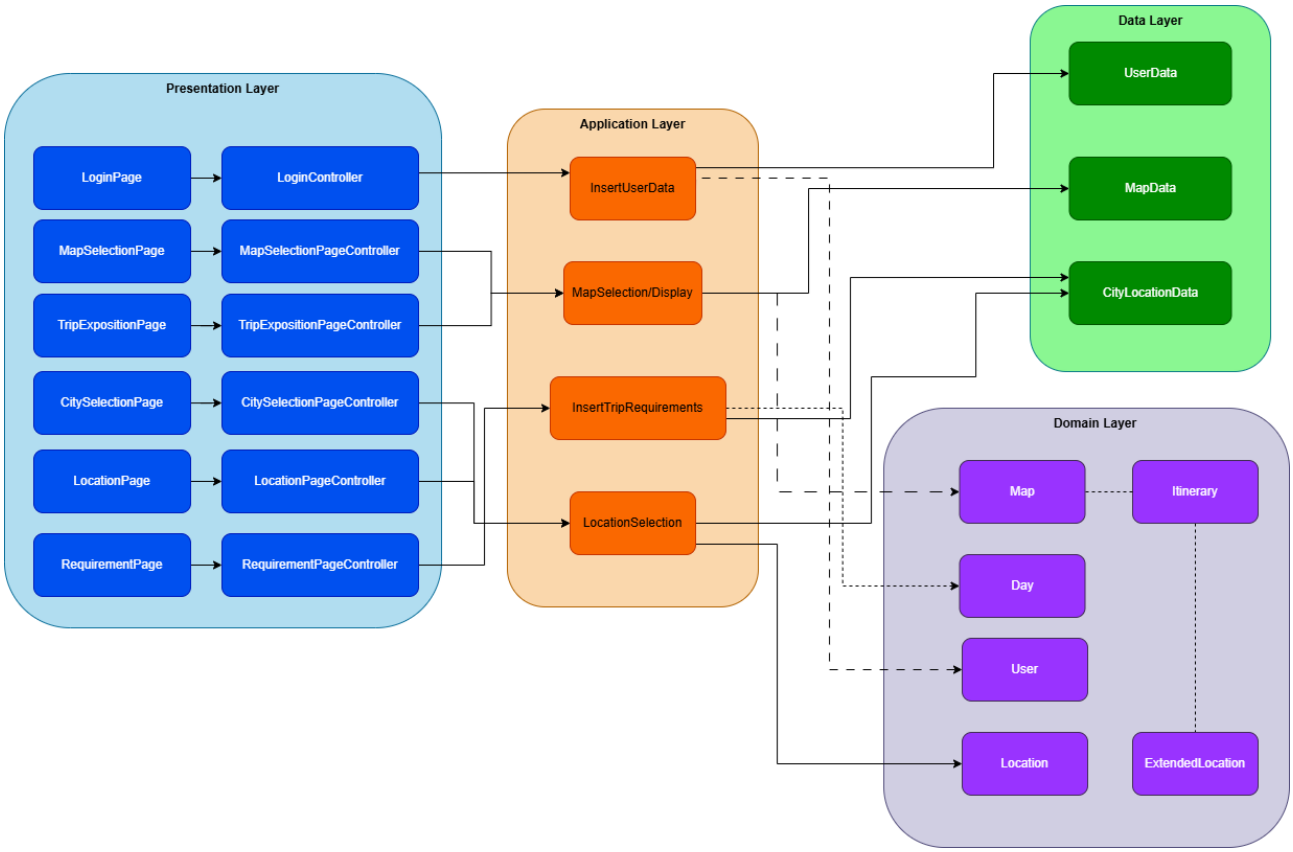
Iterazione 2

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").
  PRECONDIZIONE: l'utente deve avere eseguito il log-in all'applicazione.
  POSTCONDIZIONE: se l'utente decide di creare una nuova mappa, vengono avviati i casi d'uso da 4 a 7.

Aggiornamento dell'architettura dell'applicazione client

Con l'aggiunta di un nuovo caso d'uso si è rivelato necessario espandere anche la struttura dell'applicazione client, introducendo una nuova pagina e varie strutture dati; lo schema Riverpod ampliato è dunque il seguente.



Spiegazione delle componenti dell'architettura Riverpod

Presentation Layer

LoginPage: pagina iniziale all'apertura dell'app (dopo eventualmente una schermata di apertura), permette l'accesso o l'iscrizione dell'utente.

LoginController: riceve i dati di accesso e/o quelli di iscrizione, esegue una funzione di codifica hash sulla password e poi si collega alle funzionalità relative all'accesso (fornite dal server Java); una volta inseriti i dati, l'applicazione si sposta sulla MapSelectionPage.

MapSelectionPage: pagina contenente la lista dei piani di viaggio già creati dall'utente (se ce ne sono, in caso contrario mostra un messaggio che esorta l'utente a crearne di nuovi); è possibile selezionare una delle mappe disponibili toccando la card relativa, spostandosi sulla TripExpositionPage.
In alternativa, in fondo alla pagina è presente un bottone che permette di creare un nuovo piano di viaggio; toccando il bottone ci si sposta alla CitySelectionPage.

MapSelectionPageController: si interfaccia con il server Java per ottenere la lista delle mappe create in precedenza dall'utente.

CitySelectionPage: pagina contenente una listview delle città disponibili per la visita; l'utente ne può selezionare una e dare conferma con un apposito pulsante per passare alla LocationSelectionPage.

CitySelectionPage Controller: si interfaccia con il server Java per ottenere la lista delle città disponibili nel database.

LocationSelectionPage: viene proposta una lista di località disponibili per la visita nella città selezionata; l'utente può sceglierne uno o più e dare conferma con un pulsante per passare alla RequirementsPage.

LocationSelectionPage Controller: si interfaccia con il server Java per ottenere la lista delle località disponibili nel database, filtrando per la città selezionata precedentemente; inoltre si occupa del recupero e del salvataggio delle immagini relative alle località in una cache locale (questo per velocizzare la visualizzazione delle immagini).

RequirementsPage: pagina dove l'utente può inserire una serie di informazioni relative alle proprie necessità per il viaggio: dove è alloggiato (o per lo meno, da dove vuole partire per questa visita), la velocità di marcia (lenta/media/rapida), il numero di giorni che intende trascorrere a visitare la città, e gli orari di partenza e sosta che gradirebbe fare giorno per giorno; una volta terminata la compilazione, l'utente può confermare e passare alla TripExpositionPage.

RequirementsPageController: si occupa del corretto salvataggio dei dati inseriti dall'utente in un'apposita struttura dati, denominata Mappa (Map), in modo che questa possa essere utilizzata per calcolare l'itinerario di viaggio.

TripExpositionPage: (TripPage, alternativamente) Propone all'utente l'itinerario del viaggio, mostrando con chiarezza orari di arrivo previsti per ogni tappa; è possibile scorrere verticalmente nella pagina per visualizzare tutte le località ordinate e, se sono stati impostati più giorni di viaggio, scorrere orizzontalmente per visualizzare l'itinerario giorno per giorno.
Per le pause pranzo, è possibile selezionare il luogo dove si desidera mangiare tra una lista dei ristoranti più vicini nell'orario definito nei requisiti; questi sono visibili in un widget a comparsa che si apre alla pressione del riquadro della tappa relativa alla pausa pranzo.

Da questa pagina è possibile tornare alla RequirementsPage, qualora fosse necessario modificare i requisiti del viaggio, oppure alla MapSelectionPage, qualora si desiderasse creare o selezionare un itinerario differente.

TripExpositionPageController: se si sta creando una nuova mappa, invia al server il dato di tipo Mappa in modo che il server possa generare, salvare e restituire il piano di viaggio, che viene codificato in un'altra struttura dati, Itinerario;
se si sta aprendo una mappa preesistente, invia una richiesta al server per recuperare i dati dell'itinerario.
Altra funzionalità di questo componente è quella di gestire la ricerca dei ricerca dei ristoranti per gli appositi widget di selezione ristorante; anche in questo caso la ricerca è effettuata inviando una richiesta al server.

Application Layer

InsertUserData: l'applicazione deve permettere all'utente di registrarsi, in modo di tener traccia degli itinerari creati da ogni utente.

MapSelection/Display: l'applicazione deve utilizzare le località e le preferenze inserite dall'utente per creare una tabella di marcia scandita temporalmente, modificabile e consultabile in un secondo momento.

LocationSelection: l'utente deve poter selezionare la città che desidera visitare e i luoghi che è interessato a visitare in tale città.

InsertTripRequirements: l'utente deve essere in grado di specificare le proprie preferenze al fine di vivere un'esperienza su misura.

Domain Layer

User: rappresenta i dati di accesso dell'utente all'applicazione

- userName (stringa)
- password (stringa)

Mappa : descrive i dati inseriti dall'utente riguardo al viaggio, prima dell'elaborazione del server

- nomeUtente (string)
- idMappa (string)
- latitudineAlloggio (double)
- longitudineAlloggio (double)
- numeroGiorni (int)
- velocitàMedia (double)
- luoghi (List di oggetti di tipo Luogo)
- giornate (List di oggetti di tipo Giornata)

Itinerario: descrive il piano di viaggio restituito dall'API lato server

- nomeItineratio (string)
- nomeUtente (string)
- giorniViaggio (Dizionario, con chiave la stringa giornoViaggio e valore un oggetto di tipo LuogoEsteso)

Luogo: descrive una località d'interesse in una città tra quelle visitabili dall'utente

- nome (stringa)
- latitudine (double)
- longitudine (double)
- città (string)
- indirizzo (string)
- tipo (string)
- tempoVisita (string)
- immagine (string dell'URL)

LuogoEsteso: si tratta di un oggetto Luogo a cui è assegnato un orario di arrivo

- luogo (Luogo)
- oraArrivo (TimeOfDay – formato di tempo ora:minuti)

Giornata: descrive i requisiti forniti dall'utente per una specifica giornata di viaggio

- oraInizio (TimeOfDay)
- devoPranzare (bool)
- oraPranzo (TimeOfDayx)
- pausa (int)
- tempoPranzo (int)
- tempoVisita (int)

Data Layer

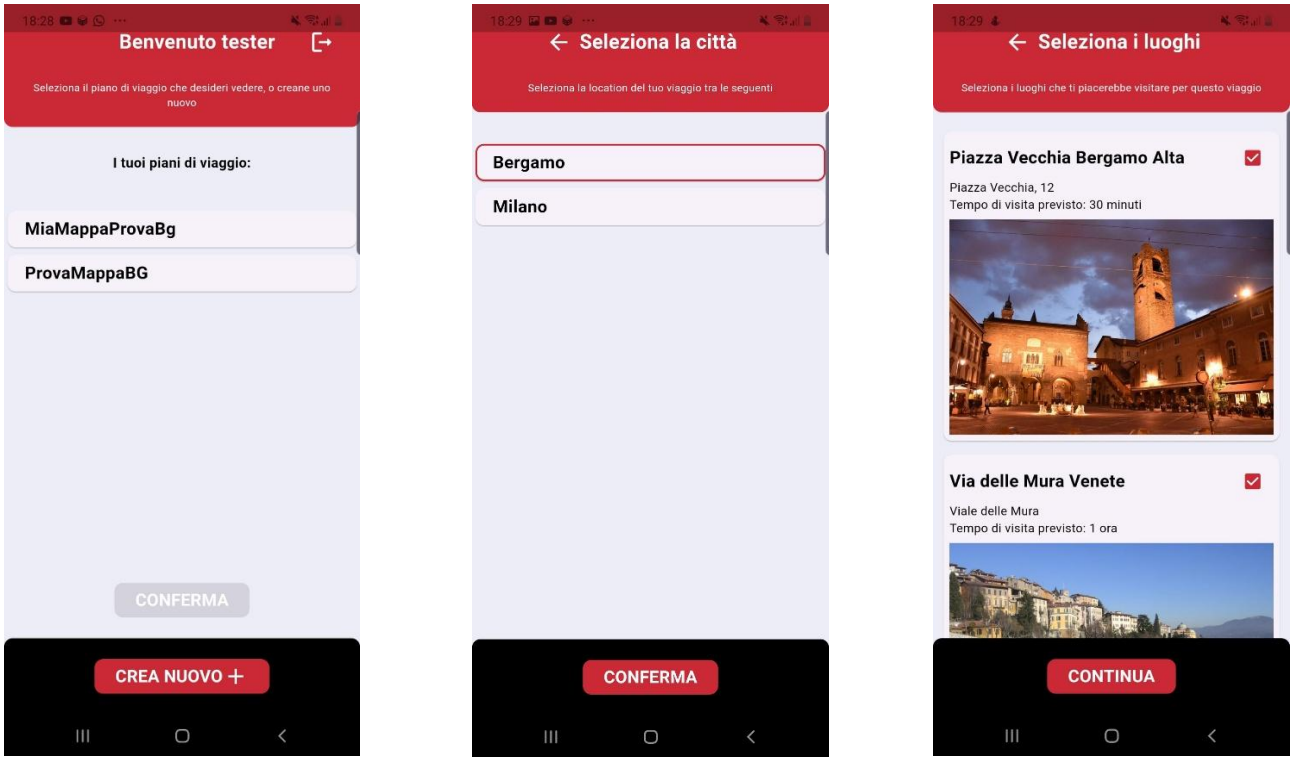
UserData: tabella del database contenente i dati di login di tutti gli utenti, ha la sola funzione di permettere/negare l'accesso all'app.

MapData: tabella del database contenente tutti i dati riguardanti gli itinerari salvati dagli utenti tramite l'utilizzo dell'app.

CityLocationData: tabella del database contenente i dati delle località disponibili per la visita (ogni località ha come attributo la propria città, non c'è una tabella delle città).

Implementazione front-end

Per la seconda iterazione sono state implementate le pagine relative alla selezione delle mappe utente, la cui aggiunta è seguita naturalmente alla definizione del relativo caso d'uso, oltre che alle pagine di selezione della città e delle località da visitare; per la pagina di selezione delle località è stata implementata anche la visualizzazione di una fotografia del luogo, per guidare visivamente le decisioni dell'utente.



Da sinistra a destra: mapSelectionPage, citySelectionPage e locationSelectionPage.

API-Luoghi

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

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.

getAllCitta

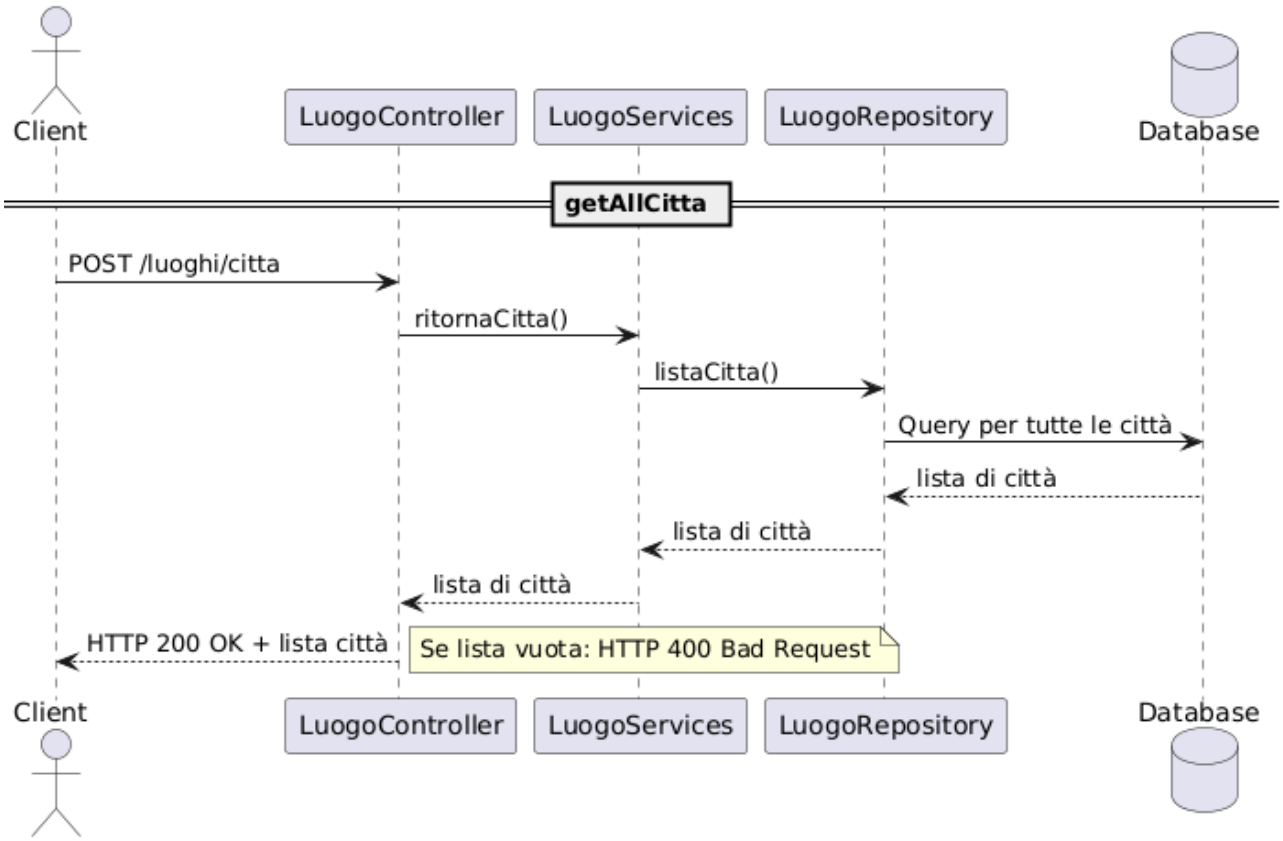
La prima API offerta è getAllCitta.

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.



getLuoghiByCitta

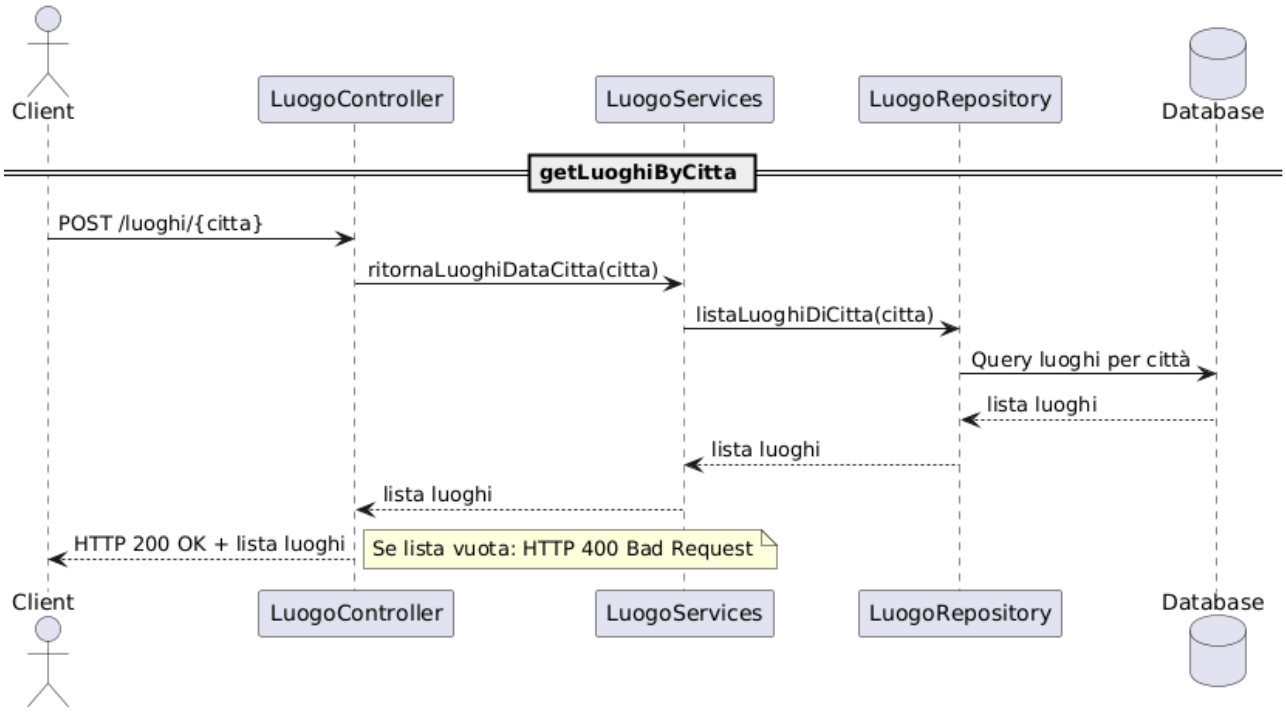
Il secondo servizio è *getLuoghiByCitta*, un API che permette di ottenere tutti i luoghi di interesse presenti in una determinata città.

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

Il service a sua volta contatta il repository *(listaLuoghiDiCitta*), 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.



API-Itinerario

getNomiItinerarioByUtente

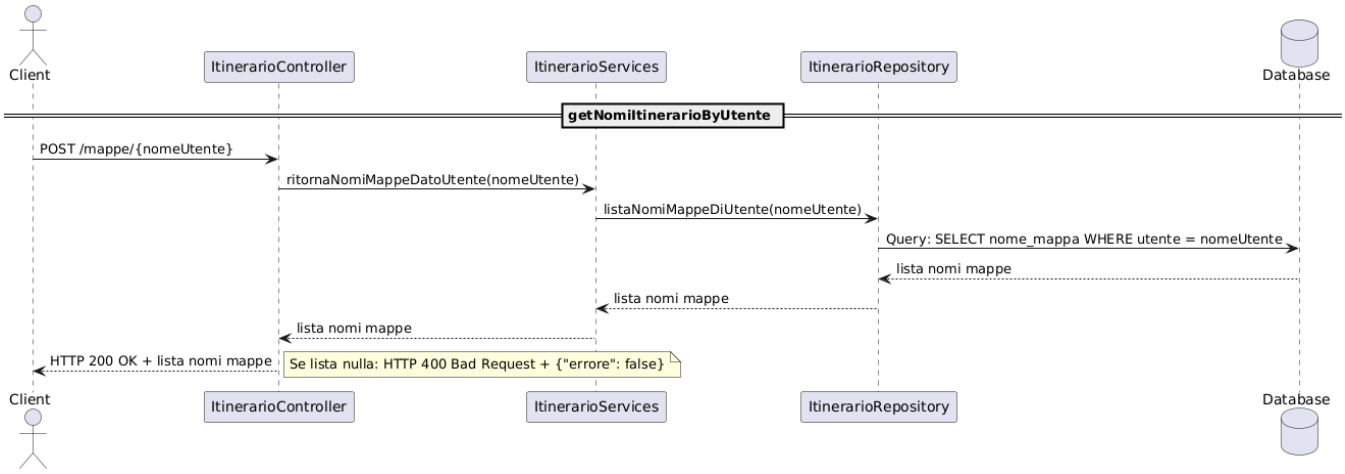
Questa API serve a restituire la lista dei nomi delle mappe (cioè degli itinerari) associati a uno specifico utente.

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.

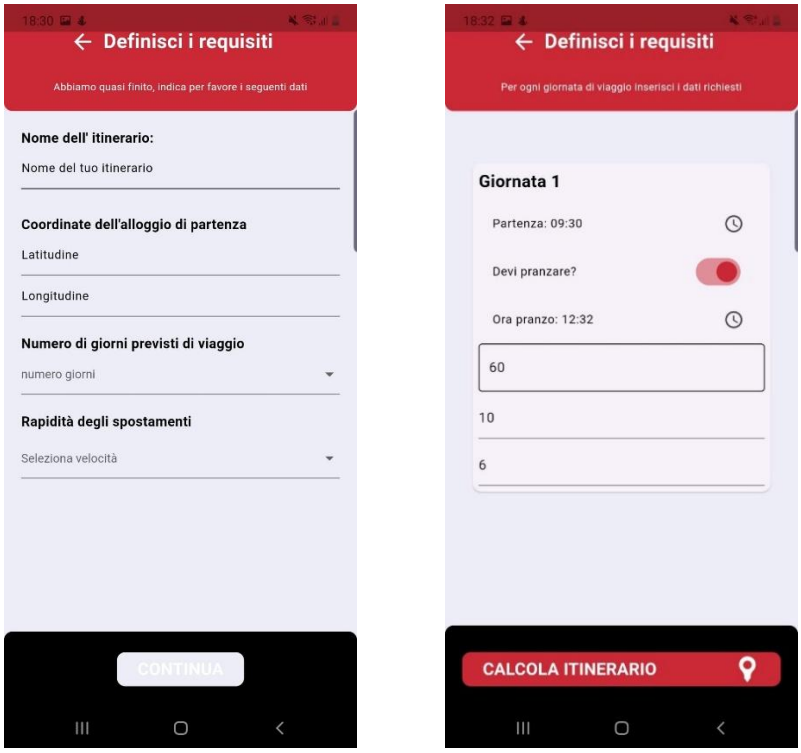


Iterazione 3

Implementazione front-end

Per la terza iterazione sono state implementate le pagine relative all'inserimento dei requisiti e alla visualizzazione dell'itinerario risultante.

Per questioni di chiarezza del codice Flutter, la pagina dei requisiti è stata separata in due file: uno dedicato all'inserimento dei dati generali dell'itinerario (nome dell'itinerario, durata del viaggio in giorni, etc.), l'altro per l'inserimento di dati specifici per il singolo giorno di viaggio (orario di partenza e rientro, orario pausa pranzo etc.).



Da sinistra a destra: requirementsPage e requirementsSubPage.

La pagina dell'itinerario è rimasta pressoché invariata rispetto al layout iniziale, eccetto per la posizione del pulsante di selezione del ristorante, che è stato posto nella posizione corretta rispetto alla timeline della giornata (piuttosto che in fondo alla pagina in maniera statica).

Da sinistra a destra: tripPage e tripPage (con aperto il modal di selezione del ristorante da visitare in pausa pranzo).

API-Luoghi

Abbiamo aggiunto una API a quelle riguardanti i luoghi:

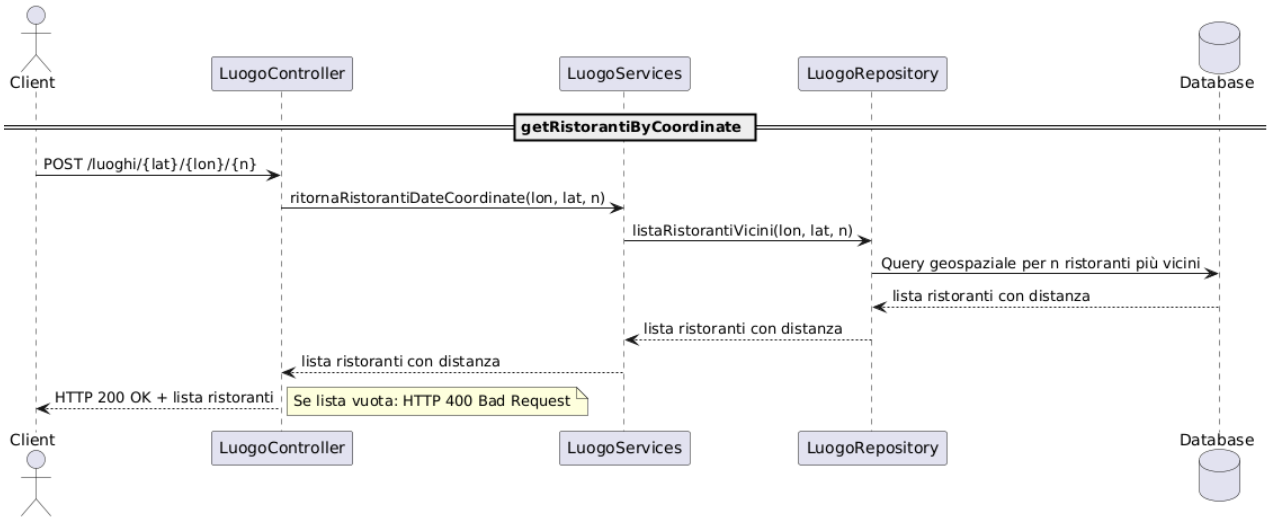
getRistorantiByCoordinate

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

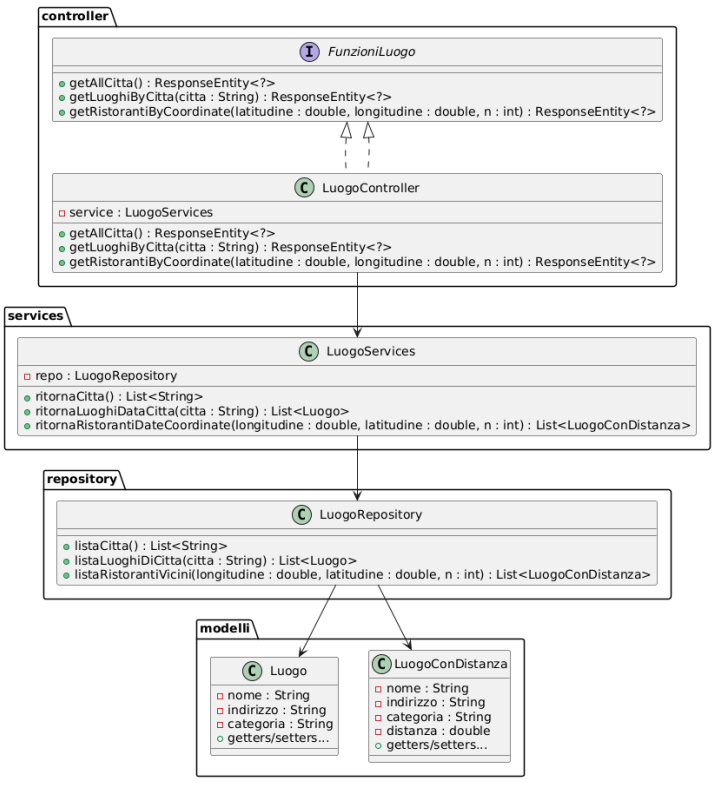
Il controller (*getRistorantiByCoordinate*) passa i parametri necessari al service (*ritornaRistorantiDateCoordinate*). Il service chiama il repository (*listaRistorantiVicini*), che grazie a una query geospaziale calcola quali sono i n ristoranti più vicini.

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.



Tutte le API riguardanti i luoghi sono contenute nel seguente diagramma delle classi:



API-Itinerario

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.

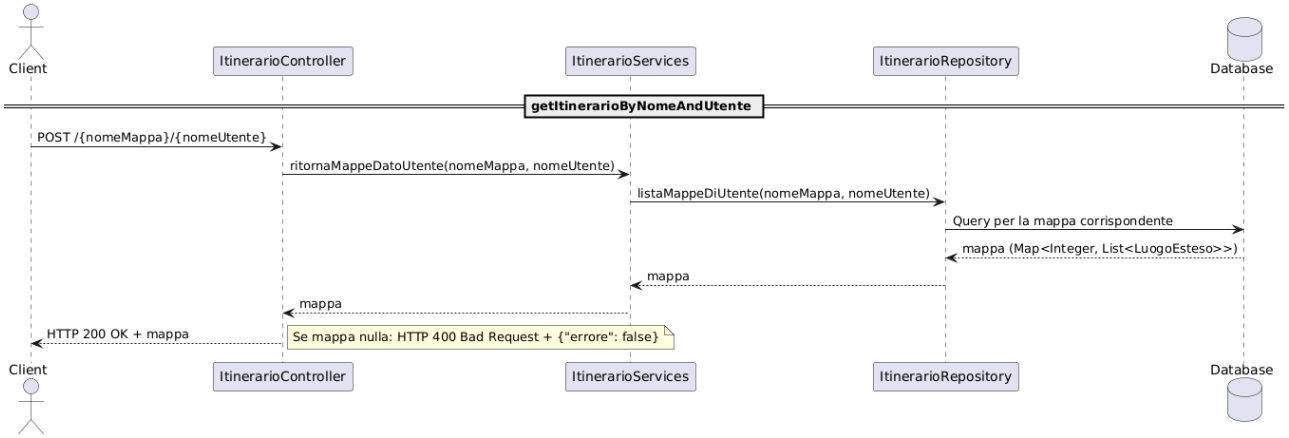
Il controller riceve due parametri come path variable e chiama il metodo *ritornaMappeDatoUtente* nel service.

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.

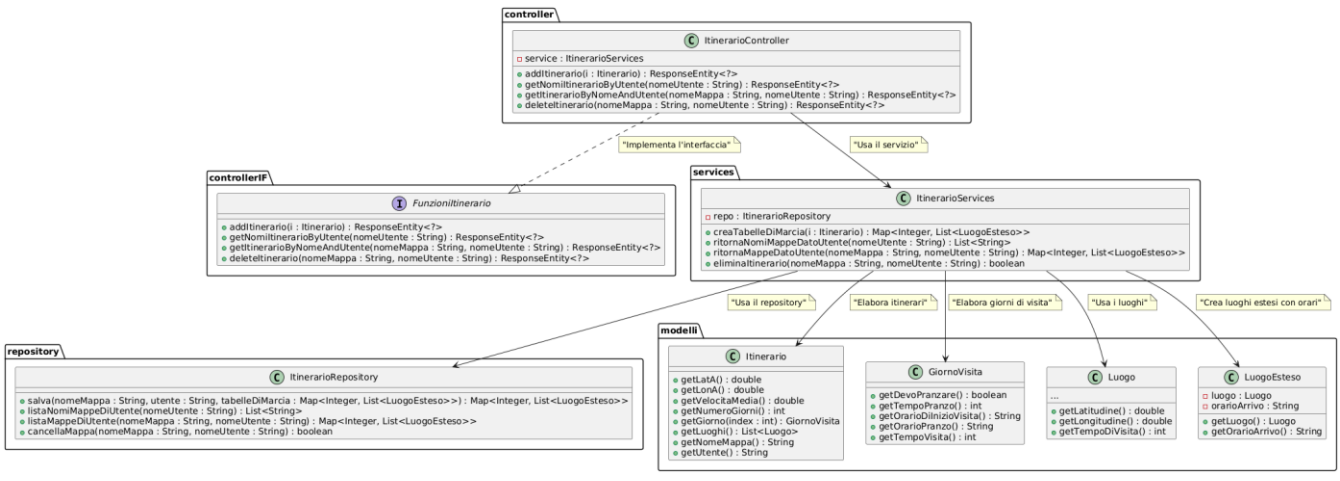
Se la mappa viene trovata, il controller risponde al client con HTTP 200 OK e la mappa stessa (che è rappresentata come una mappa Java, dove la chiave è il numero del giorno e il valore è la lista dei luoghi visitati in quel giorno).

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

Questa API è fondamentale perché consente all'applicazione di caricare e visualizzare in dettaglio un itinerario creato e salvato in precedenza da un utente.



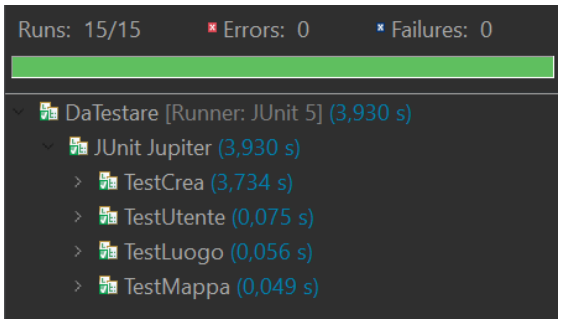
Tutte le API riguardanti gli itinerari sono contenute nel seguente diagramma delle classi:



Testing del software

Come visto nelle iterazioni precedenti, ogni API è stata testata seguendo tre modalità complementari. La prima, più banale, consiste nel verificare a occhio che l'API svolga correttamente la funzione prevista, osservandone il comportamento durante l'esecuzione. La seconda prevede 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. La terza modalità, più strutturata, utilizza JUnit, dove vengono scritti metodi di test per verificare puntualmente le operazioni di lettura e scrittura che le API eseguono sul database.

Concentrandoci su quest'ultimo caso, riportiamo i risultati dell'esecuzione in cascata dei test di tutte le query delle API. Come si può osservare, i 15 metodi definiti all'interno delle apposite classi di test producono tutti i risultati attesi, confermando che le API sono correttamente funzionanti e che le operazioni sul database avvengono senza errori.



Analisi qualità e metriche del software

Per la valutazione della qualità del software e l'analisi delle metriche è stato utilizzato lo strumento CodeMR, che consente di generare report in formato HTML corredati di grafici esplicativi sui risultati ottenuti. Questa analisi ha permesso di monitorare diversi aspetti qualitativi del codice.

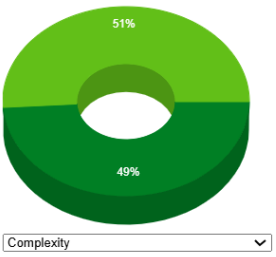
Tra i vari grafici generati riportiamo solo quelli relativi a complessità, accoppiamento, assenza di coesione e size. Più nello specifico: la complessità misura quanto è difficile comprendere e mantenere il codice, il coupling indica il grado di dipendenza tra classi o moduli, l'assenza di coesione evidenzia quanto le responsabilità di una classe siano disperse e poco correlate, mentre la dimensione del codice è il numero di righe o metodi.

Come si può notare dai grafici prodotti, i risultati complessivi sono molto buoni, con valori che evidenziano una buona qualità del software nei vari progetti. L'unico progetto che risulta meno performante è il progetto *database*. Questa anomalia è dovuta al fatto che esso include i pacchetti autogenerati da jOOQ per la gestione del database: tali pacchetti, non realizzati manualmente, presentano una qualità inferiore che ha inevitabilmente peggiorato le nostre ottime statistiche globali.

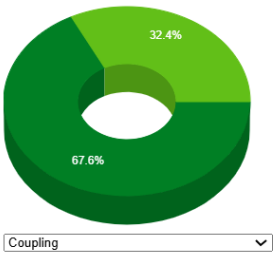
Controller

Distribution of Quality Attributes

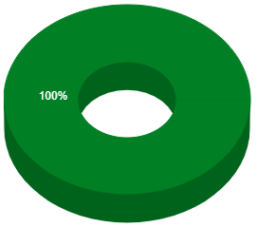
Complexity, Coupling, Cohesion, and Size



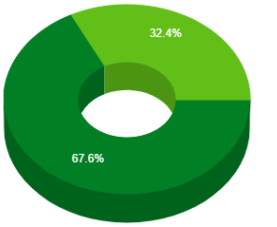
Complexity



Coupling



Lack of Cohesion

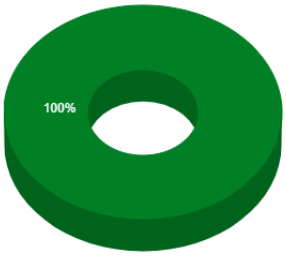


Size

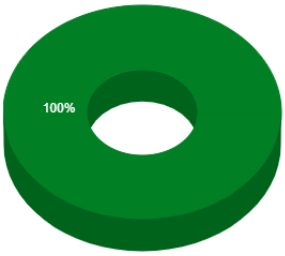
Model

Distribution of Quality Attributes

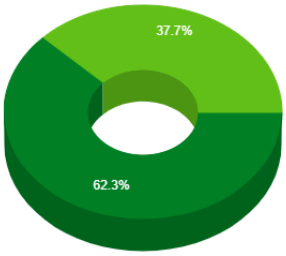
Complexity, Coupling, Cohesion, and Size



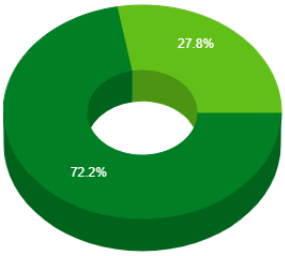
Complexity



Coupling

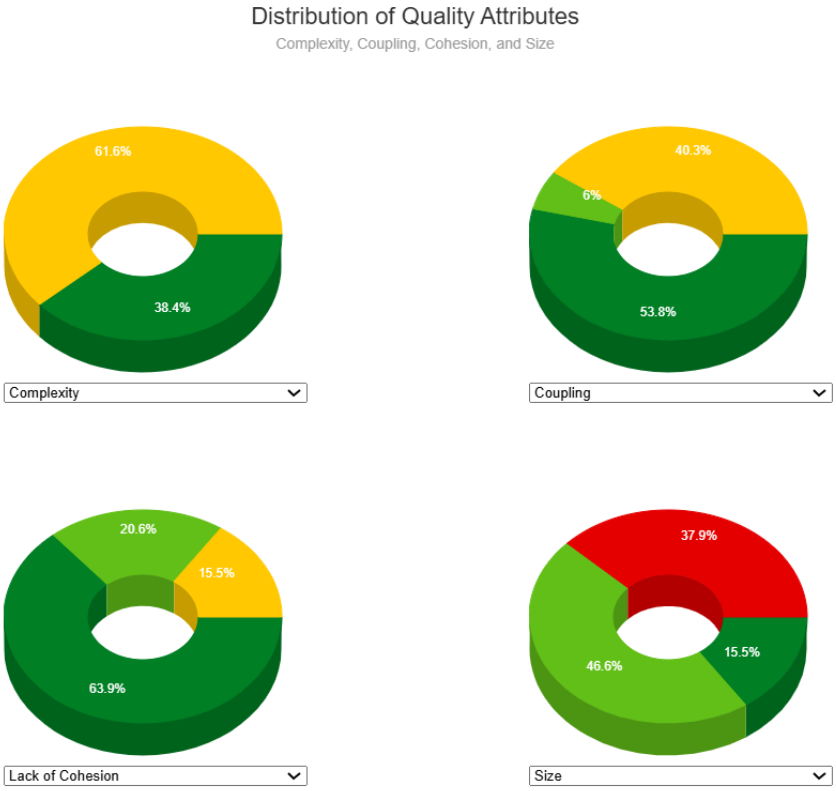


Lack of Cohesion



Size

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.