

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

**Ingegneria del Software 2021/2022
NATOUR SOFTWARE DOCUMENTATION**

GROUP ID INGSW2122_V_02

SEPTEMBER 12, 2022

<i>Studente</i>	<i>Matricola</i>
Vincenzo Tramo	N86002592



Specifica, progettazione,
implementazione e validazione
del Sistema Informativo
NaTour

GitHub

Indice

Introduzione	5
1 Descrizione del progetto	5
Specifiche dei requisiti	6
2 Requisiti software	6
2.1 Requisiti funzionali	6
2.1.1 Autenticazione	6
2.1.2 Interazioni con un itinerario	6
2.1.3 Gestione caricamento immagini	8
2.1.4 Gestione profilo personale	8
2.1.5 Gestione conversazioni private	8
2.2 Requisiti non funzionali	9
2.3 Modellazione dei Casi d'Uso	10
2.3.1 Sistema NaTour	10
2.3.2 Sistema Login	11
2.3.3 Sistema visualizzazione informazioni sentiero	11
2.3.4 Sistema inserimento sentiero	12
2.4 Tabelle di Cockburn	13
2.4.1 Inserimento di un itinerario	13
2.4.2 Invio di un messaggio	16
2.5 Prototipazione visuale via Mock-up dell'interfaccia utente	18
2.5.1 Login	18
2.5.2 Registrazione	19
2.5.3 Home	20
2.5.4 Aggiunta di un itinerario	21
2.5.5 Visualizzazione schermata di dettaglio di un sentiero	38
2.5.6 Chat privata	39
2.6 Modelli di Dominio	42
2.6.1 Classi, oggetti e relazioni di analisi	42
2.6.2 Diagrammi di sequenza di analisi	42
2.6.3 Diagrammi di attività di analisi	42
2.6.4 Diagrammi di stato di analisi	42
Documento di Design del sistema	43
3 Design del sistema	43
3.1 Analisi dell'architettura e criteri di design	43
3.2 Architettura back-end	44
3.3 Architettura front-end	45
3.4 Diagrammi di design	47
3.4.1 Diagramma delle classi di design	47

3.4.2	Diagramma di sequenza di design	47
Test Plan		48
4 Test Plan		48
4.1	Introduzione	48
4.2	Strategie di testing	48
4.3	Test cases	49
4.3.1	Test Case 1 - Fetch itinerari dal back-end	49
4.3.2	Test Case 2 - Aggiornamento messaggi non letti di una chat back-end	56
4.3.3	Test Case 3 - test della classe <code>MapActionMemory</code> front-end	61

Introduzione

1 Descrizione del progetto

NaTour è un sistema complesso e distribuito finalizzato ad offrire un moderno social network per appassionati di escursioni. Il sistema consiste in un back-end sicuro (realizzato con il framework **Spring Boot** e **tecnologie cloud-based**), performante e scalabile, e in un client mobile (**Android**) attraverso cui gli utenti possono fruire delle funzionalità del sistema in modo intuitivo, rapido e piacevole. Il servizio consente agli utenti di accedere a un database di mappe dei sentieri, che include recensioni e immagini in *crowdsourcing*.

Specifiche dei requisiti

2 Requisiti software

2.1 Requisiti funzionali

Questa sezione della documentazione presenta e descrive i *requisiti funzionali* dell'applicazione NaTour.

2.1.1 Autenticazione

Nome	Descrizione
Registrazione	Un utente può registrarsi indicando e-mail, username e password.

Tabella 1: RF.1

Nome	Descrizione
Accesso	Un utente può accedere nella piattaforma inserendo username e password oppure tramite un accounte di terze parti (Google e Facebook).

Tabella 2: RF.2

2.1.2 Interazioni con un itinerario

Nome	Descrizione
Visualizzazione itinerari	Un utente autenticato può visualizzare tutti gli itinerari caricati sulla piattaforma.

Tabella 3: RF.3

Nome	Descrizione
Inserimento itinerario	Un utente autenticato può inserire nuovi itinerari (sentieri) in piattaforma. Un sentiero è caratterizzato da un nome, una durata, un livello di difficoltà, un punto di inizio, una descrizione (opzionale), un tracciato geografico (opzionale) che lo rappresenta su una mappa e una fotografia principale. Il tracciato geografico deve essere inseribile manualmente (interagendo con una mappa interattiva) oppure tramite file in formato standard GPX.

Tabella 4: RF.4

Nome	Descrizione
Visualizzazione dettagli itinerario	Un utente autenticato può visualizzare una schermata di dettaglio per ciascun sentiero. Questa schermata mostra tutte le informazioni note del sentiero, e visualizza su mappa interattiva il punto di inizio e il tracciato geografico, se disponibile. La schermata di dettaglio mostra le eventuali recensioni degli utenti e fotografie caricate.

Tabella 5: RF.5

Nome	Descrizione
Inserimento di una recensione	Un utente autenticato può lasciare una recensione su un sentiero inserendo un punteggio (da 1 a 5) e una descrizione (opzionale).

Tabella 6: RF.6

Nome	Descrizione
Inserimento di una foto	Un utente autenticato può caricare delle fotografie su un sentiero. Le fotografie corrispondenti a un sentiero vengono mostrate nella pagina di dettaglio di quel sentiero. Inoltre, se la fotografia ha una posizione geografica di scatto salvata nei metadati, è apprezzata la possibilità di visualizzare un marker corrispondente alla fotografia sulla mappa, per mostrare in quale punto del sentiero è stata scattata.

Tabella 7: RF.7

Nome	Descrizione
Aggiunta di un itinerario tra i preferiti	Un utente autenticato può aggiungere itinerari nella propria lista di preferiti.

Tabella 8: RF.8

Nome	Descrizione
Rimozione di un itinerario dai preferiti	Un utente autenticato può rimuovere un itinerario della propria lista di preferiti.

Tabella 9: RF.9

Nome	Descrizione
Scaricare tracciato geografico in formato GPX	Un utente autenticato può scaricare il tracciato geografico di un itinerario in formato GPX.

Tabella 10: RF.10

Nome	Descrizione
Scaricare informazioni itinerario in formato PDF	Un utente autenticato può scaricare le informazioni riepilogative di un itinerario in formato PDF.

Tabella 11: RF.11

Nome	Descrizione
Visualizzazione della propria posizione sulla mappa	L'utente, tramite una apposita spunta, può la propria posizione (se il dispositivo mobile supporta localizzazione GPS) sulla mappa visualizzata nella schermata di dettaglio di un itinerario.

Tabella 12: RF.12

Nome	Descrizione
Visualizzazione della propria posizione sulla mappa	L'utente, tramite una apposita spunta, può la propria posizione (se il dispositivo mobile supporta localizzazione GPS) sulla mappa visualizzata nella schermata di dettaglio di un itinerario.

Tabella 13: RF.13

2.1.3 Gestione caricamento immagini

Nome	Descrizione
Blocco immagini inappropriate	Il sistema deve essere in grado di riconoscere automaticamente (e bloccare) eventuali immagini inappropriate/offensive.

Tabella 14: RF.14

2.1.4 Gestione profilo personale

Nome	Descrizione
Visualizzazione dati personali	Un utente loggato può visualizzare i propri dati personali (username, email, nome e cognome).

Tabella 15: RF.15

Nome	Descrizione
Disconnessione	Un utente autenticato può disconnettersi dall'applicazione.

Tabella 16: RF.16

2.1.5 Gestione conversazioni private

Nome	Descrizione
Avvio di una chat	Un utente può inviare un messaggio privato e avviare una chat con un altro utente per chiedere ulteriori informazioni circa un itinerario da lui inserito. E' possibile rispondere ai messaggi.

Tabella 17: RF.17

2.2 Requisiti non funzionali

Questa sezione presenta e descrive i *requisiti non funzionali* dell'applicazione NaTour.

Nome	Descrizione
Duplicazione username	Il sistema non consente di creare un profilo utilizzando un username già esistente.

Tabella 18: RNF.1

Nome	Descrizione
Singola recensione per itinerario	Il sistema non consente ad un utente di inserire più di una recensione sullo stesso itinerario.

Tabella 19: RNF.2

Nome	Descrizione
Singola chat tra due utenti	Il sistema non consente agli utenti di creare due chat con la stessa persona.

Tabella 20: RNF.3

Nome	Descrizione
Back-end scalabile	Il back-end deve essere messo in opera utilizzando tecnologie allo stato dell'arte quali ad esempio servizi di public Cloud Computing come Azure o AWS, al fine di massimizzare la scalabilità del sistema in vista di un possibile repentino aumento del numero degli utenti nelle fasi iniziali di rilascio al pubblico.

Tabella 21: RNF.4

Nome	Descrizione
Limite massimo fetch itinerari	Per evitare di sovraccaricare di dati l'applicazione mobile, l'intero sistema deve fetchare dal database e presentare al client 10 itinerari alla volta.

Tabella 22: RNF.5

2.3 Modellazione dei Casi d'Uso

In questa sezione verranno presentati i *diagrammi Use Case* dell'applicazione.

2.3.1 Sistema NaTour

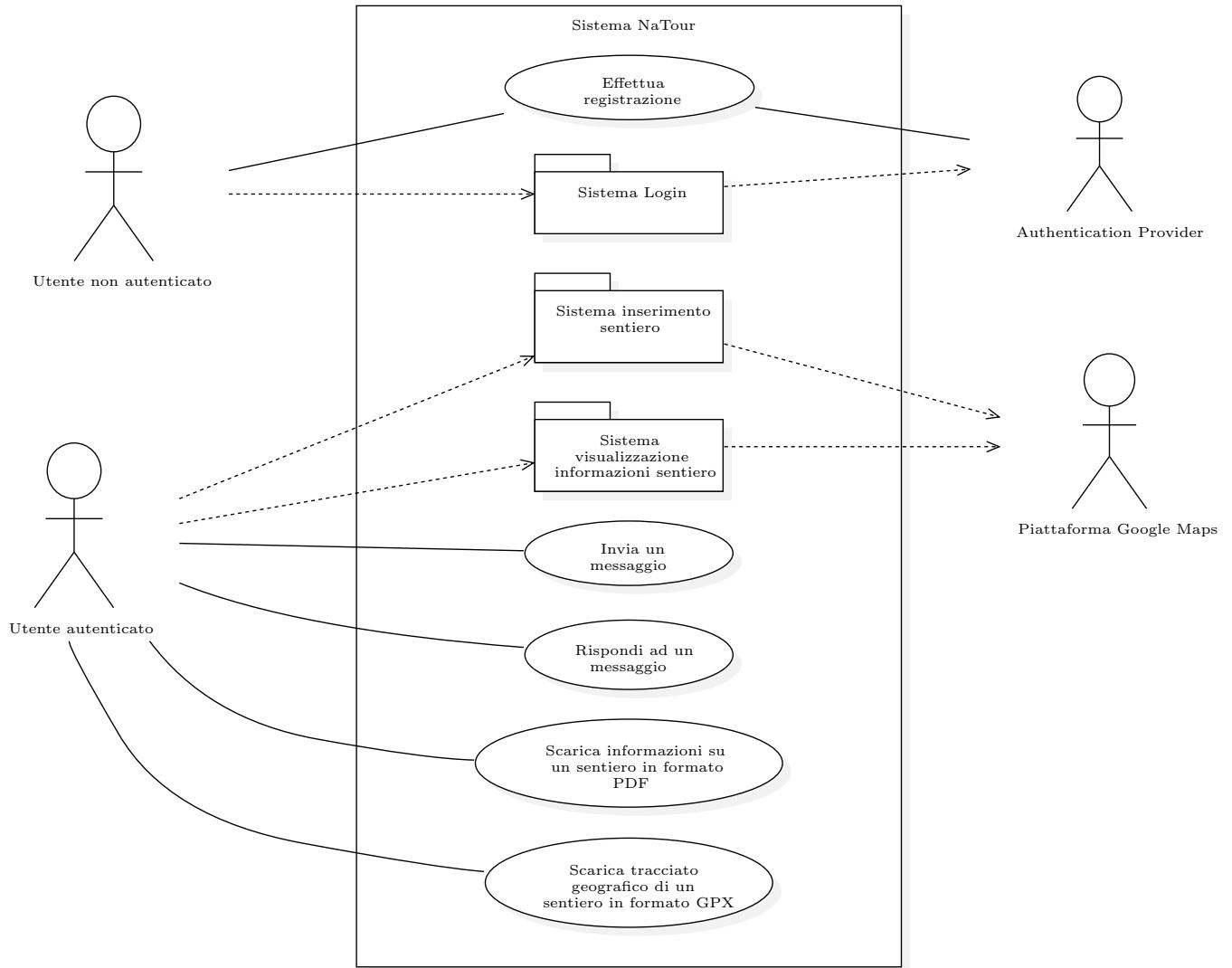


Figure 1: USD.1

2.3.2 Sistema Login

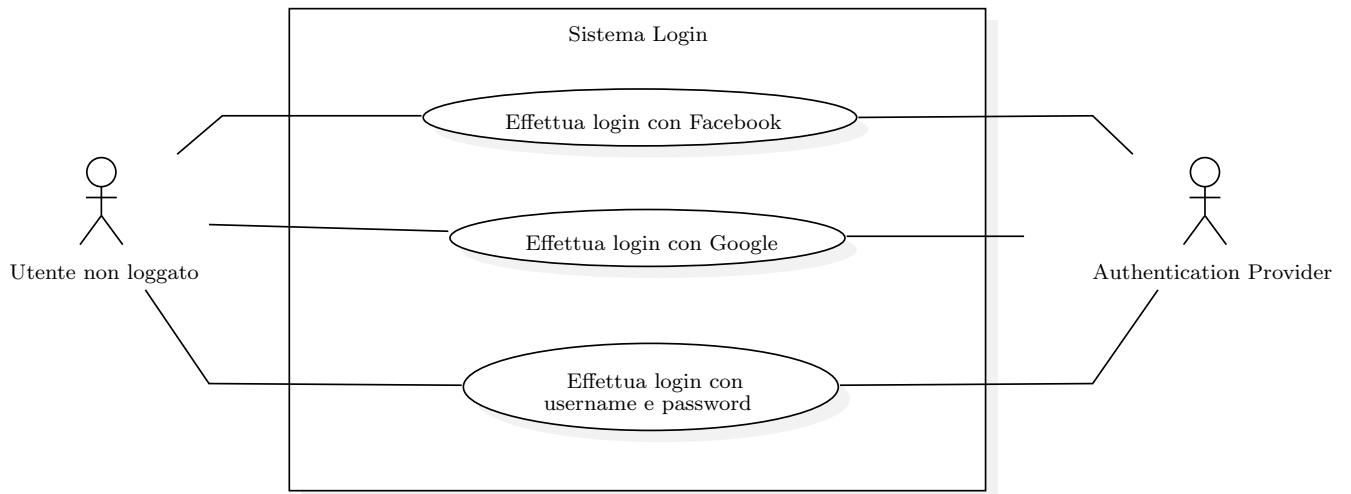


Figure 2: USD.2

2.3.3 Sistema visualizzazione informazioni sentiero

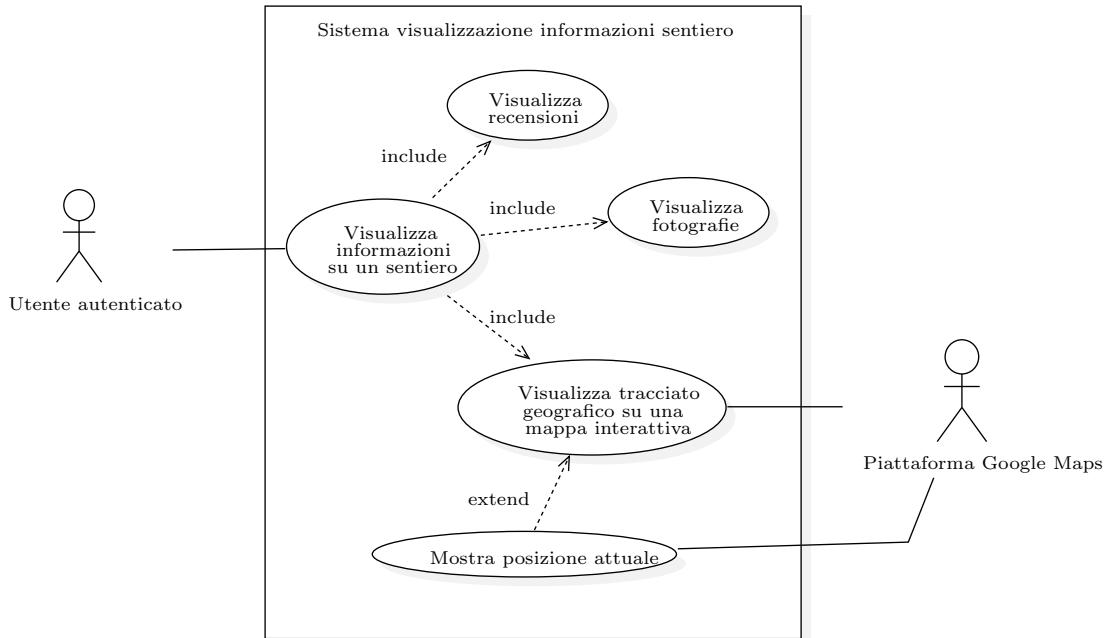


Figure 3: USD.3

2.3.4 Sistema inserimento sentiero

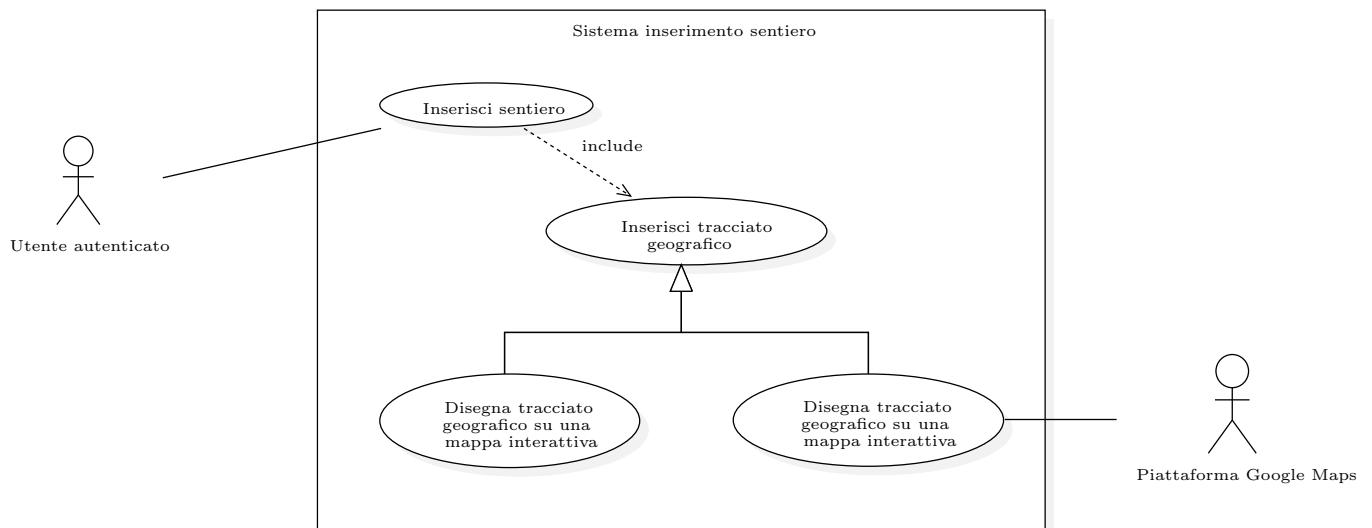


Figure 4: USD.4

2.4 Tabelle di Cockburn

Questa sezione è dedicata alle *tabelle di Cockburn* dei casi d'uso presentati nella precedente sezione (2.3).

2.4.1 Inserimento di un itinerario

USE CASE #1	Inserisce un itinerario			
Goal in Context	L'utente vuole inserire un itinerario in piattaforma			
Preconditions	L'utente è autenticato			
Success End Conditions	L'utente riesce ad inserire il nuovo itinerario			
Failed End Conditions	L'utente non riesce ad inserire il nuovo itinerario			
Primary Actor	Utente autenticato			
Secondary Actor	Google Maps			
Trigger	L'utente preme il bottone "Aggiungi percorso" nella schermata "HomeUI"			
	Step	Utente registrato	Sistema	Google Maps
	1	Preme il bottone "Aggiungi percorso" nella schermata "HomeUI"		
	2		Mostra la schermata "Create RouteUI"	
	3	Inserisce il nome dell'itinerario, durata, difficoltà, un'immagine e una descrizione (opzionale)		

	4	Preme il bottone "Conferma"		
	5		Mostra la schermata "HowDoYou WantAddA RouteUI"	
	6	Seleziona l'opzione "Draw route on map"		
	7		Mostra "DrawRoute OnMapUI"	
	8			Mostra mappa interattiva
	9	Disegna percorso sulla mappa		
	10	Preme il bottone "Conferma"		
	11		Mostra "Route Successfully CreatedUI"	
EXTENSION #1 Inserimento tracciato geografico tramite file GPX	Step	Utente autenticato	Sistema	Google Maps
	6.1	Seleziona "Upload GPX file"		

	6.2		Mostra GPX File Picker	
	6.3	Seleziona un file con estensione .gpx		
	6.4	Preme il bottone "Upload"		
	6.5		Mostra "RouteMap UI"	
	6.6	Preme "Conferma"		
	6.7		Mostra "Route Successfully CreatedUI"	

2.4.2 Invio di un messaggio

USE CASE #2	Invio di un messaggio		
Goal in Context	L'utente vuole inviare un messaggio ad un altro utente		
Preconditions	L'utente è autenticato		
Success End Conditions	L'utente riesce ad inviare un messaggio ad un altro utente		
Failed End Conditions	L'utente non riesce ad inviare il messaggio		
Primary Actor	Utente autenticato		
Trigger	L'utente seleziona un percorso e preme il bottone "Invia un messaggio"		
	Step	Utente registrato	Sistema
	1	Seleziona un percorso	
	2		Mostra la schermata "Trail DetailsUI"
	3	Preme il bottone "Invia un messaggio"	
	4		Mostra la schermata "ChatUI"
	5	Digita il messaggio	
	6	Preme il bottone "Invia"	

	7		Invia messaggio all'altro utente	
EXTENSION #1 Connessione internet assente	Step	Utente autenticato	Sistema	
	6.1		Mostra schermata "Connection Error UI"	
	6.2	Preme il bottone "Riprova"		

2.5 Prototipazione visuale via Mock-up dell’interfaccia utente

In questa sezione verranno presentati i *Mockups* dell’interfaccia utente inizialmente pensati prima dello sviluppo dell’applicazione.

2.5.1 Login

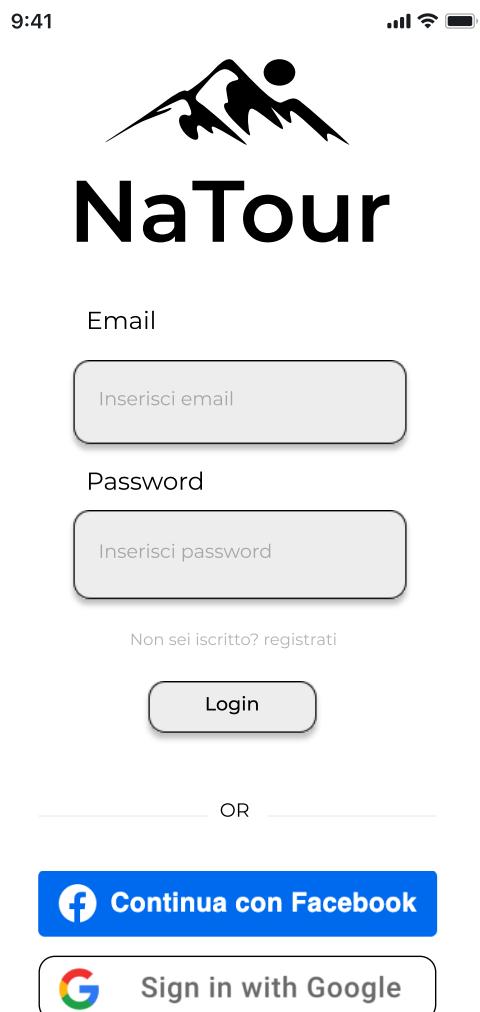


Figure 5: MK.1

2.5.2 Registrazione

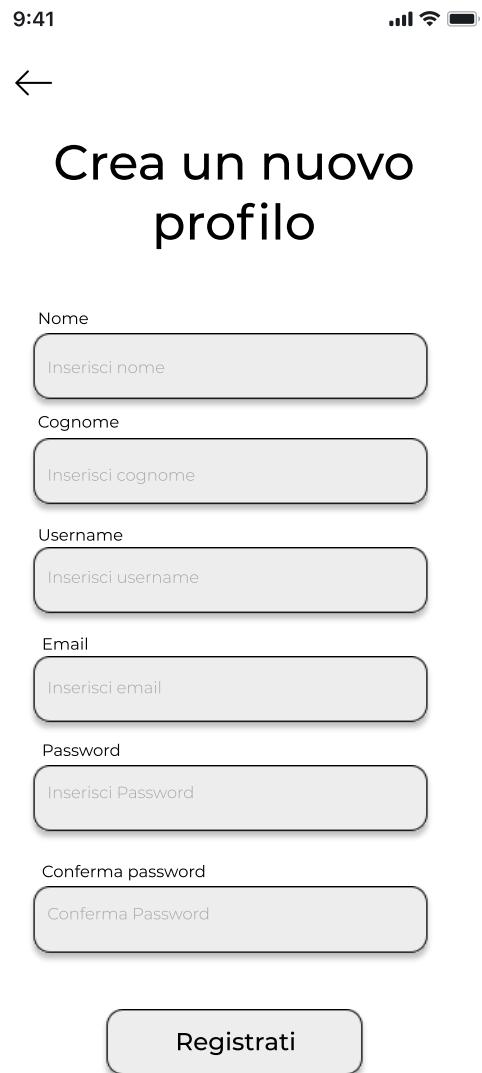


Figure 6: MK.2

2.5.3 Home

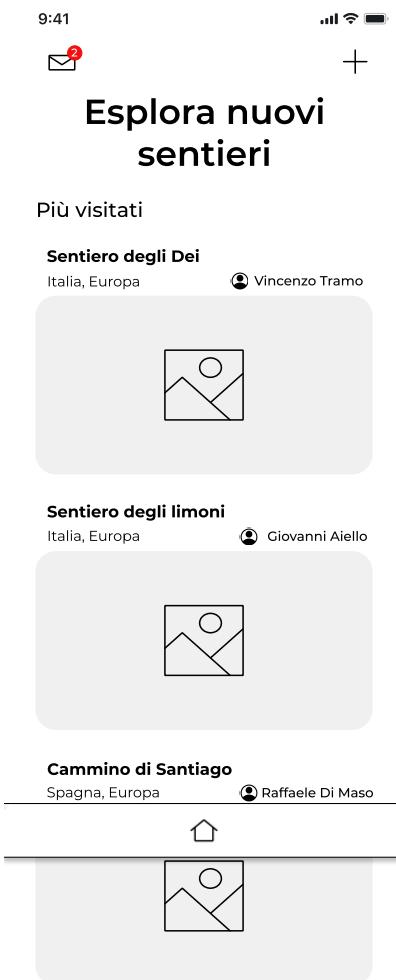


Figure 7: MK.3

2.5.4 Aggiunta di un itinerario

9:41

←

Aggiungi un sentiero

Nome
Sentiero degli dei

Durata

Livello di difficoltà

Punto d'inizio

Descrizione (facoltativa)
Inserisci testo

Conferma

Figure 8: MK.4



Figure 9: MK.5

9:41 . . . Wi-Fi 

Aggiungi un sentiero

Nome

Sentiero degli dei

Durata

0d	0h	0m
----	----	----

Livello di difficoltà

Seleziona ▾

Descrizione (facoltativa)

Inserisci testo

Upload a GPX File



File.gpx



Browse

Conferma

Conferma

Figure 10: MK.6

9:41



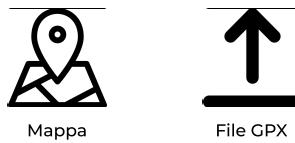
**Vuoi inserire un
tracciato
geografico al tuo
sentiero?**



Figure 11: MK.7



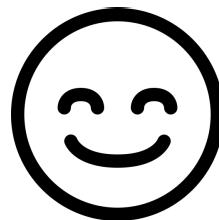
Come vuoi inserire
il tracciato
geografico al tuo
sentiero?



Mappa File GPX

Figure 12: MK.8

**Sentiero inserito
correttamente!**



OK

Figure 13: MK.9



Figure 14: MK.10



Figure 15: MK.10



Figure 16: MK.11



Figure 17: MK.12



Figure 18: MK.13



Figure 19: MK.14



Figure 20: MK.15



Figure 21: MK.16



Figure 22: MK.17



Figure 23: MK.18

**Ci sono stati
problemi durante
l'inserimento del
sentiero.**



OK

Figure 24: MK.19

2.5.5 Visualizzazione schermata di dettaglio di un sentiero

The screenshot displays a mobile application interface for the 'Sentiero degli Dei' (Path of the Gods) trail. At the top, there is a navigation bar with icons for back, download, and share. Below it, the title 'Sentiero degli Dei' is shown, along with the location 'Italia, Europa' and the author 'Vincenzo Tramo'. A rating of 4 stars from 14 votes is displayed. The trail description indicates it is an excursionist route with a duration of 6 hours and 49 minutes. The description text is in Italian. Below the description, there is a map titled 'Mappa' showing the geographical area of the trail, which is located in Tennessee, USA. The map includes various landmarks and roads. Under the map, there is a section titled 'Foto' (Photos) which shows a small thumbnail image of a landscape. At the bottom, there is a section titled 'Recensioni degli utenti' (User reviews) with two entries. The first review is by 'Raffaele Di Masi' (15/02/2022) with a 5-star rating, described as 'Posto magnifico.' The second review is by 'Giovanni Aiello' (13/02/2022) with a 5-star rating, described as 'Posto incantevole, splendido viaggio.'

Figure 25: MK.20

2.5.6 Chat privata



Figure 26: MK.21



Figure 27: MK.22



Figure 28: MK.23

2.6 Modelli di Dominio

In questa sezione verranno presentati i *modelli UML di dominio* del sistema NaTour. I diagrammi sono stati caricati sulla [repository di GitHub di NaTour](#).

2.6.1 Classi, oggetti e relazioni di analisi

Per il class diagram di analisi seguire questo link:

<https://github.com/vtramo/NaTour/tree/main/docs/domain-models/domain-class-diagram>

2.6.2 Diagrammi di sequenza di analisi

Per i sequence diagrams di analisi seguire questo link:

<https://github.com/vtramo/NaTour/tree/main/docs/domain-models/domain-sequence-diagram>

2.6.3 Diagrammi di attività di analisi

Per gli activity diagrams di analisi seguire questo link:

<https://github.com/vtramo/NaTour/tree/main/docs/domain-models/activity-diagrams>

2.6.4 Diagrammi di stato di analisi

Per gli statecharts di analisi seguire questo link:

<https://github.com/vtramo/NaTour/tree/main/docs/domain-models/statechart-diagrams>

Documento di Design del sistema

3 Design del sistema

3.1 Analisi dell'architettura e criteri di design

L'architettura del sistema NaTour è una classica architettura **client-server**. L'organizzazione dell'architettura lato client e dell'architettura lato server è stata decisa in base ai seguenti *obiettivi di design*:

- **Mantenimento:** Estendibilità, Modificabilità, Adattabilità
- **Affidabilità:** Robustezza, Disponibilità, Tolleranza ai fault
- **Performance:** Tempo di risposta

Per raggiungere questi obiettivi (in particolare un alto grado di manutenibilità del sistema) è stato scelto di utilizzare un'**architettura chiusa**: ogni strato comunica solo con quello sottostante. Al fine di massimizzare la *scalabilità del sistema* è stato scelto di utilizzare **servizi di Cloud Computing** e in particolare Amazon Web Services.

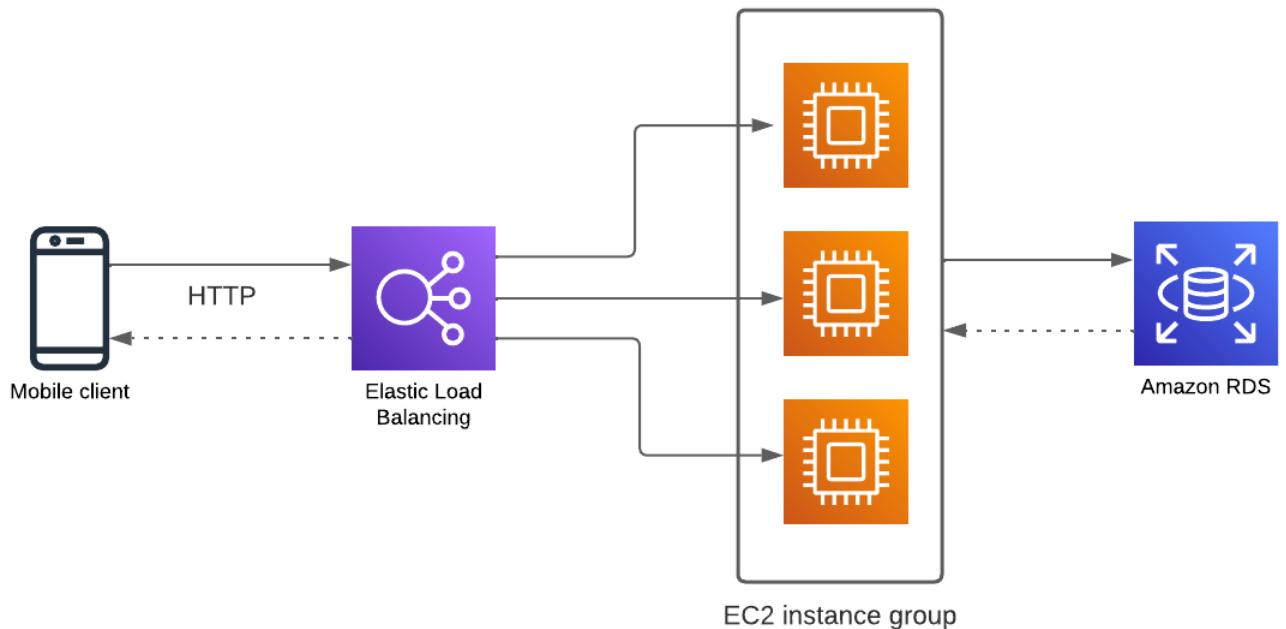


Figure 29: Simple auto-scaling AWS Architecture

3.2 Architettura back-end

Il back-end è stato realizzato con il framework **Spring Boot** utilizzando il linguaggio **Java**. Tale back-end espone **servizi REST** dunque l'intero sistema seguirà questo stile architettonico. Questo consente che ogni risorsa esposta dall'API REST sia unica e indirizzabile usando sintassi universale per uso nei link ipertestuali. E' possibile consultare la documentazione di queste API su [SwaggerUI](#).

Il back-end è suddiviso in tre strati:

- **Controllers (Presentation Layer)**: questo strato si occupa di gestire le richieste HTTP in arrivo inviate ai REST endpoints. Gestisce la presentazione del contenuto e l'interazione con l'utente;
- **Services (Business Layer)**: questo strato è quello centrale e si occupa della logica;
- **Repositories (Data Access Layer)**: questo strato si occupa di salvare/recuperare i dati dalle diverse sorgenti.

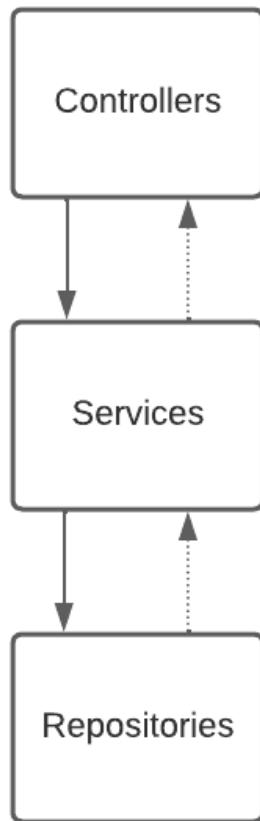


Figure 30: Architettura chiusa back-end

3.3 Architettura front-end

Il front-end è un'applicazione Android che interagisce con il back-end presentato nella Sezione 3.2. Il linguaggio utilizzato per realizzare tale applicativo è **Kotlin**.

Il front-end è suddiviso in:

- **UI Layer** (*Presentation Layer*): il ruolo di questo layer è quello di mostrare i dati dell'applicazione sullo schermo. Ogni qualvolta i dati cambiano (a causa dell'interazione dell'utente o input esterni) l'interfaccia grafica si aggiorna per riflettere i cambiamenti. Tale layer è suddiviso a sua volta in:
 - *UI Elements*: elementi dell'interfaccia grafica che renderizzano i dati sullo schermo.
 - *ViewModels* (o più in generale *State holders*): preparano, contengono ed espongono i dati da fornire agli elementi dell'interfaccia grafica.
- **Data Layer**: questo strato la *business logic* e determina come l'applicazione crea, memorizza e cambia i dati. Tale layer è fatto di *repositories* che possono contenere zero o più *data sources*.
 - *Repositories*: viene creata una repository per ogni tipo di dato. Le responsabilità di questo layer sono:
 - * Esporre i dati al resto dell'applicazione;
 - * Modificare i dati;
 - * Risolvere i conflitti tra più data sources;
 - * Astrarre le sorgenti dati dal resto dell'applicazione;
 - * Contiene la logica di business.
 - *Data sources*: ogni data sources deve avere la responsabilità di lavorare solo con una sorgente dati (file, network o un database locale).

E' stato omesso il *Domain Layer* dato che la logica di business è risultata essere molto semplice.

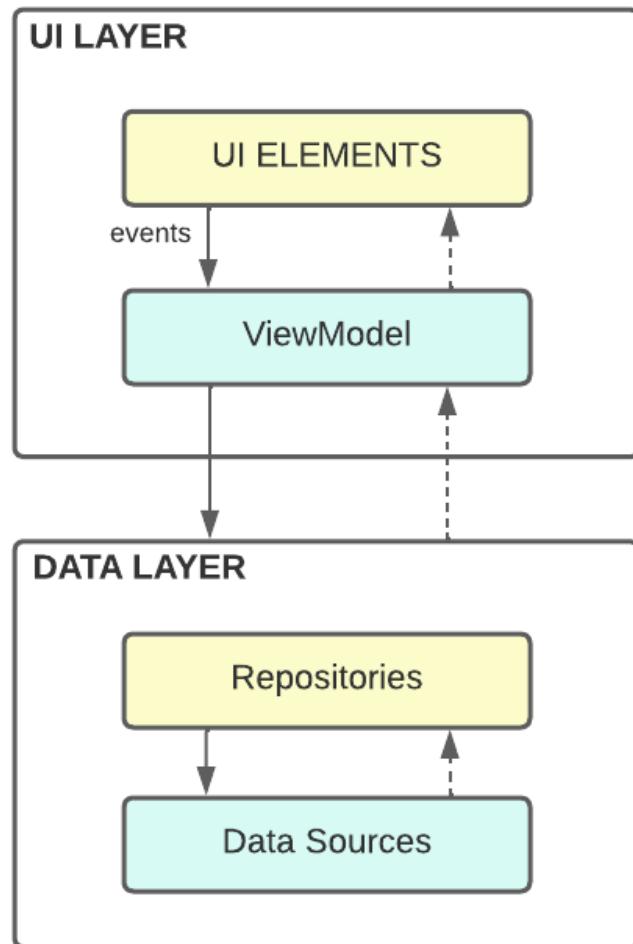


Figure 31: Architettura chiusa front-end

3.4 Diagrammi di design

In questa sezione verranno presentati i *diagrammi UML di design* del sistema NaTour. I diagrammi sono stati caricati sulla [repository di GitHub di NaTour](https://github.com/vtramo/NaTour/tree/main/docs).

3.4.1 Diagramma delle classi di design

Per il class diagram di design seguire questo link:

<https://github.com/vtramo/NaTour/tree/main/docs>

3.4.2 Diagramma di sequenza di design

Per i sequence diagrams di design seguire questo link:

<https://github.com/vtramo/NaTour/tree/main/docs>

Definizione di un piano di testing

4 Test Plan

In questa sezione verrà mostrato cosa è stato testato del sistema e verranno elencate le metodologie usate per farlo.

4.1 Introduzione

L'obiettivo del piano di testing è quello di testare alcuni requisiti funzionali (descritti in Sezione 2.1). Precisamente verranno testate alcune parti del sistema necessarie al funzionamento dei seguenti requisiti funzionali:

Nome	Descrizione
Visualizzazione itinerari	Un utente autenticato può visualizzare tutti gli itinerari caricati in piattaforma.

Tabella 23: RF.3

Nome	Descrizione
Inserimento itinerario	Un utente autenticato può inserire nuovi itinerari (sentieri) in piattaforma. Un sentiero è caratterizzato da un nome, una durata, un livello di difficoltà, un punto di inizio, una descrizione (opzionale), un tracciato geografico (opzionale) che lo rappresenta su una mappa e una fotografia principale. Il tracciato geografico deve essere inseribile manualmente (interagendo con una mappa interattiva) oppure tramite file in formato standard GPX.

Tabella 24: RF.4

Nome	Descrizione
Avvio di una chat	Un utente può inviare un messaggio privato e avviare una chat con un altro utente per chiedere ulteriori informazioni circa un itinerario da lui inserito. E' possibile rispondere ai messaggi.

Tabella 25: RF.17

4.2 Strategie di testing

La strategia di testing utilizzata si focalizza solo sul testare le API esposte dal sistema e si basa sulla scelta di determinati input per stressare e mettere alla prova i moduli sottoposti al test (*input-base coverage*). In particolare la strategia di testing utilizzata è una strategia **blackbox testing** utilizzando come criterio di scelta degli input **SECT** (*All Combinations of Blocks*). Il livello di granularità dei test è al livello dei metodi di una classe.

4.3 Test cases

Questa sezione contiene l'elenco dei casi di test insieme al codice che li implementa. Verranno presentati tre test cases che rappresentano tre metodi diversi di tre funzionalità diverse. I test sono stati realizzati seguendo un approccio black box.

4.3.1 Test Case 1 - Fetch itinerari dal back-end

`public List<TrailResponseDto> getTrails(int page)`: questo metodo fa parte del business layer (back-end) e appartiene all'interfaccia `TrailService` che fornisce servizi riguardo gli itinerari del sistema NaTour. L'implementazione di tale interfaccia è `TrailServiceImpl` che implementa il metodo `getTrails(int page)`. L'obiettivo di questo test è di controllare il corretto funzionamento dell'implementazione del metodo `getTrails(int page)`. L'implementazione è la seguente:

```

@Service
@Log
public class TrailServiceImpl implements TrailService {

    @Autowired
    private TrailRepository trailRepository;
    @Autowired
    private TrailDtoConverter trailDtoConverter;

    private static final int TRAILS_PER_REQUEST = 10;

    // ...

    @Override
    @Transactional
    public List<TrailResponseDto> getTrails(int page) {
        if (page < 0)
            throw new IllegalArgumentException("Page number must be positive.");

        Pageable pageable = PageRequest.of(
            page,
            TRAILS_PER_REQUEST,
            Sort.by("stars").descending().and(Sort.by("id").ascending())
        );

        return trailRepository.findAll(pageable)
            .stream()
            .map(trailDtoConverter::convertTrailEntityToTrailResponseDto)
            .collect(Collectors.toList());
    }
}

```

Il metodo `public List<TrailResponseDto> getTrails(int page)` è definito dal seguente contratto:

- *Precondition*: il metodo riceve come argomento un intero non negativo.
- *Postcondition*: il metodo ritorna una `List<TrailResponseDto>` contenente un numero piccolo di itinerari (definito dalla costante `TRAILS_PER_REQUEST`) ordinati in ordine decrescente di punteggio recensione e, in caso di parità, in ordine crescente di identificativi (id). I sentieri appartengono alla *i*-esima pagina specificata (parametro `page`). Se la pagina specificata è vuota, ritorna una lista vuota.
- *Penalty*: se viene passato un intero negativo, il metodo lancia `IllegalArgumentException`.

Dal contratto del metodo `getTrails` è possibile ricavare due caratteristiche. Per l'argomento `page` possiamo partizionare l'insieme dei suoi possibili valori in tre blocchi: numeri negativi, positivo e lo zero (caratteristica `segno`, chiamamo questa caratteristica C1).

In più, leggendo la postcondizione, il metodo restituisce un output diverso in base alla quantità di sentieri presenti nella pagina specificata (argomento `page`). Individuiamo quindi un'altra carattestica, chiamiamola C2, che indica proprio *il numero di sentieri presenti nella pagina* partizionata in quattro blocchi: zero, compreso tra zero e dieci, dieci o più di dieci sentieri (scegliamo il numero dieci come "spartiacque" perché specificata dalla costante `TRAILS_PER_REQUEST`).

Name	Characteristic	Blocks
C1	Valore dell'argomento <code>page</code>	{negative, zero, positive}
C2	Numero di sentieri x presenti nella pagina	{0, 0 < $x < 10$, 10, 10 ⁺ }

Tabella 26: Caratteristiche scelte per testare il metodo `getTrails`

L'*input coverage criteria* scelto è *All Combinations Coverage* come specificato prima. Possiamo quindi identificare le seguenti 12 combinazioni:

(C1 = negative, C2 = 0)	(zero, 0)	(positive, 0)
(negative, 0 < $x < 10$)	(zero, 0 < $x < 10$)	(positive, 0 < $x < 10$)
(negative, 10)	(zero, 10)	(positive, 10)
(negative, 10 ⁺)	(zero, 10 ⁺)	(positive, 10 ⁺)

E' stato scritto un test per ogni combinazione, tranne per le prime quattro con argomento negativo dato che basta avere il primo valore uguale a *negative* per la caratteristica C1 per violare la precondizione e portare ad un'eccezione. Possiamo quindi generalizzare e testare una sola combinazione tra le quattro, portando ad un totale di 9 tests.

TEST ID	1
TEST NAME	Trail Service Unit Test - <code>getTrails</code> Method Test
TEST DESCRIPTION	L'obiettivo di questo test è testare il metodo <code>public List<TrailResponseDto> getTrails(int page)</code> dell'interfaccia <code>TrailService</code>
LOCATION	Codice sotto
NOTE	Sono state fornite implementazioni fittizie per le dipendenze della classe contenitrice di questo metodo in modo tale da poter testare la classe in isolamento

INPUT	RISULTATO DESIDERATO	RISULTATO OTTENUTO
(negative, 0)	Lancia <code>IllegalArgumentException</code>	Lancia <code>IllegalArgumentException</code>
(zero, 0)	Ritorna una lista vuota	Ritorna una lista vuota
(zero, $0 < x < 10$)	Ritorna una lista non vuota contenente meno di dieci sentieri nel giusto ordine	Ritorna una lista non vuota contenente meno di dieci sentieri nel giusto ordine
(zero, 10)	Ritorna una lista con dieci sentieri nel giusto ordine	Ritorna una lista con dieci sentieri nel giusto ordine
(zero, 10^+)	Ritorna una lista con dieci sentieri nel giusto ordine	Ritorna una lista con dieci sentieri nel giusto ordine
(positive, 0)	Ritorna una lista vuota	Ritorna una lista vuota
(positive, $0 < x < 10$)	Ritorna una lista non vuota contenente meno di dieci sentieri nel giusto ordine	Ritorna una lista non vuota contenente meno di dieci sentieri nel giusto ordine
(positive, 10)	Ritorna una lista con dieci sentieri nel giusto ordine	Ritorna una lista con dieci sentieri nel giusto ordine
(positive, 10^+)	Ritorna una lista con dieci sentieri nel giusto ordine	Ritorna una lista con dieci sentieri nel giusto ordine

```

@TestInstance(Lifecycle.PER_CLASS)
@ExtendWith(MockitoExtension.class)
@ExtendWith(SpringExtension.class)
@DisplayName("A TrailService")
public class TrailServiceUnitTest {

    @MockBean
    private TrailRepository trailRepository;

    @MockBean
    private TrailDtoConverter trailDtoConverter;

    @InjectMocks
    private TrailService trailService = new TrailServiceImpl();

    @BeforeEach
    public void mockTrailDtoConverter() {
        when(trailDtoConverter.convertTrailEntityToTrailResponseDto(any()))
            .thenAnswer(this::convertEntityTrailToTrailResponseDtoMock);
    }

    TrailResponseDto convertEntityTrailToTrailResponseDtoMock(
        InvocationOnMock invocation
    ) {
        Trail trail = (Trail)invocation.getArgument(0);
        return new TrailResponseDto(trail.getId(), trail.getStars());
    }
}

```

```

@Nested
@DisplayName("When getting trails")
class GetTrailMethodTest {

    @Test
    @DisplayName("should throw an IllegalArgumentException when passing a " +
        "negative page")
    public void testNegativePage() {
        mockTrailRepositoryFindAllPageableMethod(0);

        assertThrows(
            IllegalArgumentException.class,
            () -> trailService.getTrails(-1)
        );
    }

    @Test
    @DisplayName("should return an empty trail list when passing a " +
        "zero page with no trails avaivable")
    public void testZeroPageWithNoTrailsAvaivable() {
        mockTrailRepositoryFindAllPageableMethod(0);

        final List<TrailResponseDto> trails = trailService.getTrails(0);

        assertThat(trails, hasSize(0));
    }

    @Test
    @DisplayName("should return a list of trails in the right order when " +
        "passing a zero page with less than ten trails avaivable")
    public void testZeroPageWithLessThanTenTrailsAvailable() {
        mockTrailRepositoryFindAllPageableMethod(3);
        List<TrailResponseDto> expectedTrails =
            getExpectedMockTrails(3);

        List<TrailResponseDto> trails = trailService.getTrails(0);

        assertThat(trails, is(equalTo(expectedTrails)));
    }

    @Test
    @DisplayName("should return a list of ten trails in the right order when " +

```

```

    "passing a zero page with ten trails avaialble")
public void testZeroPageWithTenTrailsAvailable() {
    mockTrailRepositoryFindAllPageableMethod(10);
    List<TrailResponseDto> expectedTrails =
        getExpectedMockTrails(10);

    List<TrailResponseDto> trails = trailService.getTrails(0);

    assertThat(trails, is(equalTo(expectedTrails)));
}

@Test
@DisplayName("should return a list of ten trails in the right order when " +
    "passing a zero page with more than ten trails avaialble")
public void testZeroPageWithMoreThanTenTrailsAvailable() {
    mockTrailRepositoryFindAllPageableMethod(15);
    List<TrailResponseDto> expectedTrails =
        getExpectedMockTrails(15);

    List<TrailResponseDto> trails = trailService.getTrails(0);

    assertThat(trails, is(equalTo(expectedTrails)));
}

@Test
@DisplayName("should return an empty trail list when passing a " +
    "positive page with no trails avaialble")
public void testPositivePageWithNoTrailsAvaiable() {
    mockTrailRepositoryFindAllPageableMethod(0);

    final List<TrailResponseDto> trails = trailService.getTrails(5);

    assertThat(trails, hasSize(0));
}

@Test
@DisplayName("should return a list of trails in the right order when " +
    "passing a positive page with less than ten trails avaialble")
public void testPositivePageWithLessThanTenTrailsAvailable() {
    mockTrailRepositoryFindAllPageableMethod(6);
    List<TrailResponseDto> expectedTrails =
        getExpectedMockTrails(6);
}

```

```

List<TrailResponseDto> trails = trailService.getTrails(5);

assertThat(trails, is(equalTo(expectedTrails)));
}

@Test
@DisplayName("should return a list of ten trails in the right order when " +
    "passing a positive page with ten trails available")
public void testPositivePageWithTenTrailsAvailable() {
    mockTrailRepositoryFindAllPageableMethod(10);
    List<TrailResponseDto> expectedTrails =
        getExpectedMockTrails(10);

    List<TrailResponseDto> trails = trailService.getTrails(5);
    System.out.println("HEY" + trails);

    assertThat(trails, is(equalTo(expectedTrails)));
}

@Test
@DisplayName("should return a list of ten trails in the right order when " +
    "passing a positive page with more than ten trails available")
public void testPositivePageWithMoreThanTenTrailsAvailable() {
    mockTrailRepositoryFindAllPageableMethod(15);
    List<TrailResponseDto> expectedTrails =
        getExpectedMockTrails(15);

    List<TrailResponseDto> trails = trailService.getTrails(7);
    System.out.println("HEY" + trails);

    assertThat(trails, is(equalTo(expectedTrails)));
}

private void mockTrailRepositoryFindAllPageableMethod(
    final int sizeContentPage
) {
    when(trailRepository.findAll(any(PageRequest.class)))
        .thenReturn(
            sizeContentPage == 0 ? Page.empty()
                : new PageImpl<>(
                    MockTrails.getMockTrails()
                        .stream()
                        .sorted()

```

```
        comparing(Trail::getStars)
            .reversed()
            .thenComparing(Trail::getId)
        ).limit(sizeContentPage)
        .collect(toList())
    )
);
}

private List<TrailResponseDto> getExpectedMockTrails(
    final int sizeContentPage
) {
    return MockTrails.getMockTrails()
        .stream()
        .sorted(
            comparing(Trail::getStars)
                .reversed()
                .thenComparing(Trail::getId)
        ).limit(sizeContentPage)
        .map(trail -> new TrailResponseDto(trail.getId(), trail.getStars()))
        .collect(toList());
}
}
```

4.3.2 Test Case 2 - Aggiornamento messaggi non letti di una chat back-end

`public Chat updateUnreadMessages(long idChat, String username, int totalUnreadMessages)`: questo metodo appartiene al business layer (back-end) e fa parte dell'interfaccia `ChatService` che offre servizi riguardo le chat/messaggi privati tra gli utenti del sistema. L'implementazione di tale interfaccia è `ChatServiceImpl` che implementa il metodo `updateUnreadMessages(long, String, int)`. L'obiettivo di questo test è quello di controllare il corretto funzionamento del metodo. L'implementazione è la seguente:

```

@Service
@Log
public class ChatServiceImpl implements ChatService {

    @Autowired
    private ChatRepository chatRepository;

    // ...

    @Override
    public Chat updateUnreadMessages(
        long chatId,
        String usernameOwner,
        int totalUnreadMessages
    ) {
        Objects.requireNonNull(usernameOwner);

        if (totalUnreadMessages < 0) {
            throw new IllegalArgumentException(
                "The total unread message can't be negative!"
            );
        }

        final Chat chat = findEntityById(
            chatRepository,
            chatId,
            "Invalid chat id (unread messages)"
        );

        if (!chat.communicateWith(usernameOwner)) {
            throw new IllegalArgumentException(
                "The user " + usernameOwner + " does not belong to this chat!"
            );
        }

        chat.setTotUnreadMessagesCounterByUsername(
            usernameOwner,
            totalUnreadMessages
        );
        chatRepository.save(chat);

        log.info("Total unread messages counter username " + usernameOwner
            + " set to " + totalUnreadMessages);
        return chat;
    }
}

```

Il metodo `updateUnreadMessages(long, String, int)` è definito dal seguente contratto:

- *Precondition*: il metodo accetta un intero `chatId` che rappresenta una chat tra due utenti esistente

nel sistema, l'username dell'utente appartenente alla chat specificare a cui aggiornare il numero di messaggi non letti in quella chat e un intero non negativo `totalUnreadMessages` che rappresenta la quantità di messaggi non letti.

- *Postcondition:* il metodo setta il numero di messaggi non letti (`totalUnreadMessages`) alla chat specificata per l'utente specificato. Il metodo ritorna la chat modificata.
- *Penalty:* il metodo lancia `IllegalArgumentException` se viene fornito una quantità di messaggi non letti negativa. Il metodo lancia `EntityNotFoundException` se la chat identificata da `chatId` non esiste. Il metodo lancia `IllegalArgumentException` se l'utente specificato identificato dalla stringa `username` non esiste oppure non appartiene alla chat specificata.

Sono state individuate le seguenti caratteristiche:

Name	Characteristic	Blocks
C1	Valore dell'argomento <code>idChat</code>	$\{< 0, \geq 0\}$
C2	La chat con id uguale a <code>idChat</code> esiste	{esiste, non esiste}
C3	Valore dell'argomento <code>totalUnreadMessages</code>	{negative, zero, positive}
C4	L'utente con l'username specificato appartiene alla chat	{appartiene, non appartiene} height

Tabella 27: Caratteristiche scelte per testare il metodo `updateUnreadMessages`

Utilizzando la strategia *All Combinations Coverage* il totale di tests ammonta a 48. Siccome la maggior parte delle combinazioni porta ad un'eccezione (perché violano le precondizioni) la quantità di tests si riduce drasticamente.

TEST ID	2
TEST NAME	Chat Service Unit Test - <code>updateUnreadMessages</code> Method Test
TEST DESCRIPTION	L'obiettivo di questo test è testare il metodo <code>updateUnreadMessages(long, String, int)</code> dell'interfaccia <code>ChatService</code>
LOCATION	Il codice si trova sotto
NOTE	Sono state fornite implementazioni fittizie per le dipendenze della classe contenitrice di questo metodo in modo tale da poter testare la classe in isolamento

INPUT	RISULTATO DESIDERATO	RISULTATO OTTENUTO
(11, esiste, appartiene, 5)	Aggiorna correttamente i messaggi non letti di quella chat per quell'utente settandoli a 5	Aggiorna correttamente i messaggi non letti di quella chat per quell'utente settandoli a 5
(11, esiste, non appartiene, 100)	Lancia un <code>IllegalArgumentException</code>	Lancia un <code>IllegalArgumentException</code>
(0, non esiste, appartiene, 1)	Lancia un <code>EntityNotFoundException</code>	Lancia un <code>EntityNotFoundException</code>
(11, esiste, appartiene, 0)	Aggiorna correttamente i messaggi non letti di quella chat per quell'utente settandoli a 0	Aggiorna correttamente i messaggi non letti di quella chat per quell'utente settandoli a 0

```

@TestInstance(Lifecycle.PER_CLASS)
@ExtendWith(MockitoExtension.class)
@ExtendWith(SpringExtension.class)
@DisplayName("A ChatService")

```

```

public class ChatServiceUnitTest {

    @MockBean
    private ChatRepository chatRepository;

    @InjectMocks
    private ChatService chatService = new ChatServiceImpl();

    @Nested
    @DisplayName("When updating unread messages from a user's chat")
    class UpdateUnreadMessagesMethodTest {

        private static final Long EXISTING_CHAT_ID = 1L;
        private static final Long UNKNOWN_CHAT_ID = 0L;

        {
            configureChatRepositoryMockBean();
        }

        private void configureChatRepositoryMockBean() {
            when(chatRepository.findById(EXISTING_CHAT_ID))
                .thenAnswer(this::withExistingMockChat);

            when(chatRepository.save(any()))
                .thenReturn(null);
        }

        private Optional<Chat> withExistingMockChat(InvocationOnMock invocation) {
            ApplicationUser vtramo = new ApplicationUser();
            vtramo.setUsername("vtramo");
            ApplicationUser lauraa = new ApplicationUser();
            lauraa.setUsername("lauraa");

            Optional<Chat> optionalChat = Optional.of(
                new Chat(
                    (Long)invocation.getArgument(0),
                    "vtramo",
                    "lauraa",
                    null
                )
            );

            Chat chat = optionalChat.get();
        }
    }
}

```

```

        chat.addUser(vtramo); chat.addUser(lauraa);

        return optionalChat;
    }

    @Test
    @DisplayName("should update them correctly if the chat exists, the " +
        "user belongs to that chat and you pass a positive number.")
    public void testUpdateExistingChatExistingUserWithPositiveNumber() {
        final int expectedTotalUnreadMessages = 5;

        final Chat chat = chatService.updateUnreadMessages(
            EXISTING_CHAT_ID,
            "vtramo",
            5
        );

        assertTrue(
            chat.getTotUnreadMessageCounterByUsername("vtramo")
            == expectedTotalUnreadMessages
        );
    }

    @Test
    @DisplayName("should throw an IllegalArgumentException if the chat exists " +
        "but the provided username doesn't belong to it.")
    public void testUpdateExistingChatButWithNotExistingUser() {
        assertThrows(
            IllegalArgumentException.class,
            () -> chatService.updateUnreadMessages(
                EXISTING_CHAT_ID,
                "unknown",
                100
            )
        );
    }

    @Test
    @DisplayName("should throw an EntityByIdNotFoundException if the chat " +
        "doesn't exist.")
    public void testUpdateNotExistingChat() {
        assertThrows(
            EntityByIdNotFoundException.class,

```

```
() -> chatService.updateUnreadMessages(
    UNKNOWN_CHAT_ID,
    "vtramo",
    1
)
);

}

@Test
@DisplayName("should update them correctly if the chat exists, the " +
"user belongs to that chat and you pass zero.")
public void testUpdateExistingChatExistingUserWithZero() {
    final int expectedTotalUnreadMessages = 0;

    final Chat chat = chatService.updateUnreadMessages(
        EXISTING_CHAT_ID,
        "vtramo",
        0
    );

    assertTrue(
        chat.getTotUnreadMessageCounterByUsername("vtramo")
        == expectedTotalUnreadMessages
    );
}
}
```

4.3.3 Test Case 3 - test della classe MapActionMemory front-end

```
fun addAction(
    type: Map ActionType,
    routePoints: List<Pair<Double, Double>>,
    isFirstPoint: Boolean = false
): Boolean
```

Questo metodo appartiene alla classe `MapActionMemory` che fa parte del layer UI (front-end). L'obiettivo di questa classe è quello di salvare le azioni eseguite dall'utente sulla mappa che implicano una cancellazione (ad esempio un'eliminazione di un punto precedentemente inserito o l'eliminazione dell'interno percorso precedentemente disegnato). Tale salvataggio permette il recupero delle azioni eliminate e quindi riportare la mappa allo stato precedente (operazione *redo*). Il contratto di tale metodo è:

- *Precondition*: la lista di punti `routePoints` deve essere non vuota. Il flag booleano `isFirstPoint` è uguale a `true` solo e soltanto se la lista `routePoints` ha un solo punto e l'operazione `Map ActionType` non è uguale a `DELETE`
- *Postcondition*: memorizza la specificata azione con le sue informazioni etichettandola come ultima azione compiuta.
- *Penalty*: lancia `IllegalArgumentException` se la lista `routePoints` è vuota. Lancia `IllegalArgumentException` se la lista `routePoints` ha più di un punto e il flag `isFirstPoint` è settato uguale a `true`. Lancia `IllegalArgumentException` se il tipo di azione è `DELETE` e il flag `isFirstPoint` è settato uguale a `true`.

Sono state individuate le seguenti caratteristiche:

Name	Characteristic	Blocks
C1	Tipo di azione da aggiungere <code>Map ActionType</code>	{UNDO, DELETE}
C2	Size lista <code>routePoints</code>	{0, 1, 1 ⁺ }
C3	Valore flag booleano <code>isFirstPoint</code>	{true, false}

Tabella 28: Caratteristiche scelte per testare il metodo `addAction` della classe `MapActionMemory`

Generalizzando le combinazioni che violano la precondizione testando una sola di esse, le combinazioni da testare in totale ne sono 7:

(DELETE, 0, false)	(UNDO, 2, true)	(DELETE, 2, true)
(DELETE, 1, false)	(DELETE, 2, false)	(UNDO, 1, true)
(UNDO, 2, false)		

TEST ID	3
TEST NAME	MapActionMemory Unit Test - addAction Method Test
TEST DESCRIPTION	L'obiettivo di questo test è testare il metodo <code>addAction(MapActionType, List<Pair<Double, Double>>, Boolean)</code> della classe MapActionMemory
LOCATION	Il codice si trova sotto
NOTE	...

INPUT	RISULTATO DESIDERATO	RISULTATO OTTENUTO
(DELETE, 0, false)	Lancia un <code>IllegalArgumentException</code>	Lancia un <code>IllegalArgumentException</code>
(UNDO, 2, true)	Lancia un <code>IllegalArgumentException</code>	Lancia un <code>IllegalArgumentException</code>
(DELETE, 2, true)	Lancia un <code>IllegalArgumentException</code>	Lancia un <code>IllegalArgumentException</code>
(DELETE, 1, false)	Memorizza correttamente l'azione specificata come l'ultima azione	Memorizza correttamente l'azione specificata come l'ultima azione
(DELETE, 2, false)	Memorizza correttamente l'azione specificata come l'ultima azione	Memorizza correttamente l'azione specificata come l'ultima azione
(UNDO, 1, true)	Memorizza correttamente l'azione specificata come l'ultima azione	Memorizza correttamente l'azione specificata come l'ultima azione
(UNDO, 2, false)	Memorizza correttamente l'azione specificata come l'ultima azione	Memorizza correttamente l'azione specificata come l'ultima azione

```

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@DisplayName("A MapActionMemory")
class MapActionMemoryUnitTest {

    @DisplayName("When adding an action")
    class AddActionMethodTest {

        private val mMapActionMemory = MapActionsMemory()

        @AfterEach
        private fun resetMapActionMemory() = mMapActionMemory.reset()

        @Test
        @DisplayName("should throw IllegalArgumentException when passing an empty route points list")
        fun shouldThrowIAEWhenPassingAnEmptyRoutePointsList() {
            val mapActionType = MapActionType.DELETE
            val routePoints = listOf<Pair<Double, Double>>()
            val isFirstPoint = false

            assertThrows(IllegalArgumentException::class.java) {
                mMapActionMemory.addAction(mapActionType, routePoints, isFirstPoint)
            }
        }
    }
}

```

```

}

@Test
@DisplayName("should throwIllegalArgumentException when passing a list with more than " +
    "one points and 'isFirstPoint' flag equal to true")
fun shouldThrowIAEWhenPassingAListWithMoreThanOnePointsAndIsFirstPointFlagEqualToTrue(){
    val mapActionType = MapActionType.UNDO
    val routePoints = listOf(Pair(2.0, 3.0), Pair(2.0, 2.0))
    val isFirstPoint = true

    assertThrows(IllegalArgumentException::class.java) {
        mMapActionMemory.addAction(mapActionType, routePoints, isFirstPoint)
    }
}

@Test
@DisplayName("should throwIllegalArgumentException when passing a delete action and " +
    "'isFirstPoint' flag equal to true")
fun shouldThrowIAEWhenPassingADeleteActionAndIsFirstPointFlagEqualToTrue() {
    val mapActionType = MapActionType.DELETE
    val routePoints = listOf(Pair(2.0, 3.0), Pair(2.0, 2.0))
    val isFirstPoint = true

    assertThrows(IllegalArgumentException::class.java) {
        mMapActionMemory.addAction(mapActionType, routePoints, isFirstPoint)
    }
}

@Test
@DisplayName("should add correctly when passing a delete action with one point " +
    "and 'isFirstPoint' flag equal to false")
fun shouldAddCorrectlyWhenPassingADeleteActionWithOnePointAndIsFirstPointFlagEqualToFalse() {
    val mapActionType = MapActionType.DELETE
    val routePoints = listOf(Pair(2.0, 3.0))
    val isFirstPoint = false
    val mapActionExcepted = MapAction(mapActionType, routePoints, isFirstPoint)

    val hasBeenSuccessfullyAdded = mMapActionMemory.addAction(
        mapActionType,
        routePoints,
        isFirstPoint
    )

    assertAll(listOf {
        assertEquals(hasBeenSuccessfullyAdded, true)
        assertEquals(mapActionExcepted, mMapActionMemory.getLastAction())
    })
}

```

```

        })
    }

    @Test
    @DisplayName("should add correctly when passing a delete action with two points " +
        "and 'isFirstPoint' flag equal to false")
    fun shouldAddCorrectlyWhenPassingADeleteActionWithTwoPointsAndIsFirstPointFlagEqualToFalse() {
        val mapActionType = MapActionType.DELETE
        val routePoints = listOf(Pair(2.0, 3.0), Pair(2.0, 2.0))
        val isFirstPoint = false
        val mapActionExcepted = MapAction(mapActionType, routePoints, isFirstPoint)

        val hasBeenSuccessfullyAdded = mMapActionMemory.addAction(
            mapActionType,
            routePoints,
            isFirstPoint
        )

        assertAll(listOf {
            assertEquals(hasBeenSuccessfullyAdded, true)
            assertEquals(mapActionExcepted, mMapActionMemory.getLastAction())
        })
    }

    @Test
    @DisplayName("should add correctly when passing an undo action with one point " +
        "and 'isFirstPoint' flag equal to true")
    fun shouldAddCorrectlyWhenPassingAUndoActionWithOnePointsAndIsFirstPointFlagEqualToString() {
        val mapActionType = MapActionType.UNDO
        val routePoints = listOf(Pair(2.0, 3.0))
        val isFirstPoint = true
        val mapActionExcepted = MapAction(mapActionType, routePoints, isFirstPoint)

        val hasBeenSuccessfullyAdded = mMapActionMemory.addAction(
            mapActionType,
            routePoints,
            isFirstPoint
        )

        assertAll(listOf {
            assertEquals(hasBeenSuccessfullyAdded, true)
            assertEquals(mapActionExcepted, mMapActionMemory.getLastAction())
        })
    }

    @Test
    @DisplayName("should add correctly when passing an undo action with two points " +

```

```
        "and 'isFirstPoint' flag equal to false")
fun shouldAddCorrectlyWhenPassingAUndoActionWithTwoPointsAndIsFirstPointFlagEqualToFalse() {
    val mapActionType = MapActionType.UNDO
    val routePoints = listOf(Pair(2.0, 3.0), Pair(3.0, 3.0))
    val isFirstPoint = false
    val mapActionExcepted = MapAction(mapActionType, routePoints, isFirstPoint)

    val hasBeenSuccessfullyAdded = mMapActionMemory.addAction(
        mapActionType,
        routePoints,
        isFirstPoint
    )

    assertAll(listOf {
        assertEquals(hasBeenSuccessfullyAdded, true)
        assertEquals(mapActionExcepted, mMapActionMemory.getLastAction())
    })
}

}
}
```