



Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Esame di Software Architecture Design

Documentazione di progetto

Hermes Messenger

Anno Accademico 2021/2022

Alfonso Conte M63001378
Matteo Conti M63001317
Daniele Fazzari M63001384
Francesco Iannaccone M63001324

Indice

1 Avvio del progetto	1
1.1 Parti interessate	2
1.1.1 Utenti	2
1.2 Tabella Attori-Obiettivi	2
1.3 Requisiti generali	3
1.4 Storie Utente	4
1.5 Requisiti non funzionali	4
1.5.1 Sicurezza	5
1.5.2 Testability	6
1.5.3 Usabilità	7
1.5.4 Modifiability	7
1.6 Vincoli	8
1.7 Glossario	9
1.8 Stima dei costi	9
1.9 System Context Diagram	12
2 Processo di sviluppo	14
2.1 UP	14
2.2 SCRUM	16
2.3 eXtreme Programming	17
2.4 Tool per la condivisione del lavoro	18
2.5 Tool e tecnologie per lo sviluppo	21
3 Fase di Analisi	26
3.1 Textual Analysis	26
3.2 Mockup e Design Concepts	27
3.3 Modello dei Casi d'Uso	28
3.3.1 Casi d'uso in formato dettagliato	28
3.3.2 Use Case Diagram	34
3.4 System Sequence Diagram	35
3.4.1 Registrazione	35
3.4.2 Login	36
3.4.3 Invia Messaggio	36
3.4.4 Modifica dati profilo	37
3.4.5 Logout	38
3.4.6 Crea chat	39
3.4.7 Elimina chat	39
3.4.8 Blocca utente	40
3.5 System Domain Model	41

3.6	State Machine Diagram degli stati utente	42
4	Architettura e Progettazione	44
4.1	Client-Server Pattern	44
4.2	Hermes Client e Pattern MVC	45
4.2.1	Vantaggi del MVC	46
4.3	Server e Microservizi	46
4.4	Vista Componenti e Connettori	48
4.5	Dinamica dei componenti	49
4.5.1	Crea chat	50
4.5.2	Effettua login	52
4.5.3	Effettua registrazione	53
4.5.4	Invia messaggio	54
4.6	Client Class Diagram	57
4.6.1	View	57
4.6.2	Controller	58
4.6.3	Model	59
4.6.4	Servizi	60
4.7	Chat Handler Class Diagram	64
4.7.1	Interfaccia logica	68
4.8	AuthService Class Diagram	71
4.8.1	Interfaccia logica	72
4.9	MessageService Class Diagram	74
4.9.1	Interfaccia logica	75
4.10	UserDataService Class Diagram	77
4.10.1	Interfaccia logica	78
4.11	NotifyService Class Diagram	80
4.11.1	Interfaccia logica	81
5	Implementazione	82
5.1	Framework di sviluppo	82
5.2	Descrizione dei file realizzati	83
5.3	Design pattern utilizzati	87
5.3.1	Pattern GRASP	87
5.3.2	Pattern Gof	88
5.4	Dinamica del Sistema	89
5.4.1	Avvio del client	89
5.4.2	Blocca utente	91
5.4.3	Crea chat	92
5.4.4	Effettua login	93
5.4.5	Effettua logout	94
5.4.6	Effettua registrazione	95
5.4.7	Elimina chat	96
5.4.8	Invia messaggio	97
5.4.9	Caricamento immagine profilo	98
5.5	Deployment Diagram	99
5.6	Entity Relationship diagram	100
5.6.1	ER Client	100
5.6.2	ER Server	101

5.7	Manuale di utilizzo	102
5.7.1	Configurazione e Installazione	102
5.7.2	Utilizzo User	102
6	Testing	106
6.1	Testing dei servizi	106
6.1.1	Testing microservizio AuthService	107
6.1.2	Testing microservizio MsgService	108
6.1.3	Testing microservizio UserDataService	108
6.1.4	Testing microservizio NotifyService	109
6.1.5	Testing ChatHandler	110
6.2	Testing funzionale	111
6.2.1	Testing operazione di login	111
6.2.2	Testing operazione di registrazione	112
6.2.3	Testing operazione di invio messaggio	113
6.2.4	Testing servizio di notifica	114
6.2.5	Testing della funzionalità delle immagini profilo	115
6.2.6	Consegna immagine profilo	115
6.3	Video demo	116

Capitolo 1

Avvio del progetto

Si vuole realizzare un'applicazione di messaggistica istantanea open source basata sulla sicurezza e sulla privacy degli utenti che la utilizzano.

Gli obiettivi dell'applicazione sono principalmente:

- Permettere ad un utente di iscriversi al servizio, di inviare e ricevere messaggi da altri utenti iscritti
- Garantire la consegna del messaggio inviato anche qualora il destinatario sia in quel momento offline
- Garantire che la registrazione e la comunicazione avvengano in totale sicurezza, attraverso una gestione crittografata delle chiavi d'accesso

Allo scopo deve essere garantita la possibilità a nuovi utenti di registrarsi alla piattaforma. Una volta effettuata la registrazione, in qualità di utenti autenticati, sarà possibile inviare e ricevere messaggi la cui confidenzialità e integrità dovrà essere garantita tramite meccanismi di crittografia; Il sistema software dovrà garantire la consegna real-time dei messaggi agli utenti online, e il successivo recapito, al momento della connessione al servizio a quelli offline. Ciascun utente registrato è caratterizzato da un profilo contenente come informazioni: nome utente, email e foto profilo. Eventualmente sarà possibile per un utente modificare alcune delle informazioni legate al suo profilo. Sarà, inoltre, possibile per gli utenti loggati, bloccare eventuali utenti indesiderati, con la possibilità di sbloccarli in futuro.

Il servizio di messaggistica sarà utilizzabile tramite un'apposita applicazione mobile che garantirà l'accesso alle funzionalità supportate con l'ausilio di un'interfaccia grafica.

L'architettura del sistema è basata sul modello Client-Server, in cui ogni utente potrà richiedere informazioni ai vari Server, a partire dal Server di autenticazione e login. La comunicazione sarà inoltre mantenuta tramite un Chat Handler, che agirà in funzione di Broker tra l'entità del client e quella del server.

1.1 Parti interessate

Gli utilizzatori del servizio si dividono in anzitutto in due tipologie principali di utenti: quelli registrati al servizio e quelli non registrati. Una volta registrati gli utenti possono inviare e ricevere messaggi.

1.1.1 Utenti

Utente non registrato

Gli utenti non registrati non hanno accesso ai servizi fondamentali dell'applicazione. Al momento dell'accesso all'applicazione sarà data loro la possibilità di effettuare la registrazione. Al momento della registrazione, l'utente specifica un username, con il quale sarà raggiungibile dagli altri utenti, ed una password che sarà necessaria per accedere in modo sicuro all'applicazione. Effettuata la registrazione potrà effettuare il login al servizio.

Utente registrato

Gli utenti registrati hanno accesso alle funzionalità del sistema.

A seguito di una fase di login questi saranno reindirizzati alla schermata iniziale dell'applicazione, dalla quale potranno visualizzare le conversazioni attive con gli altri utenti, avviare nuove conversazioni o aggiornare le informazioni del loro profilo.

Un aspetto fondamentale che il sistema deve garantire è il corretto recapito dei messaggi inviati sia ad utenti sia online che offline.

1.2 Tabella Attori-Obiettivi

Per l'individuazione degli attori primari, può essere utile stilare la cosiddetta *Tabella Attori-Obiettivi*, in cui si inseriscono gli attori principali del sistema congiuntamente ai possibili obiettivi perseguitibili, ciascuno dei quali può corrispondere ad un caso d'uso:

ATTORE	OBIETTIVO
Utente Non Registrato	Effettuare la registrazione
Utente Registrato	Effettuare il login
Utente Autenticato	Inviare e ricevere un messaggio; Modificare dati profilo; Eliminare una chat; Effettuare Logout; Bloccare un utente; Sbloccare un utente; Effettuare login automatico;

1.3 Requisiti generali

1. L'applicazione deve consentire agli utenti, che si possono connettere tramite l'applicazione Android, di inviare e ricevere messaggi, bloccare ed eliminare chat con altri utenti.
2. L'applicazione deve notificare ed inoltrare in tempo reale i messaggi agli utenti online
3. Ogni utente è registrato con un username e una password e, all'atto della registrazione, gli saranno assegnate, secondo l'algoritmo di crittografia asimmetrica RSA, una chiave pubblica e una chiave privata, le quali verranno utilizzate per il criptaggio end-to-end dei messaggi
4. Nel momento in cui un messaggio viene mandato ad un utente offline, il server rileva l'evento e memorizza il messaggio trasmesso in un database. Non appena tale utente ritorna online, effettuando il login, si vedrà notificati i messaggi memorizzati.

1.4 Storie Utente

Storie utente
Al primo avvio dell'applicazione, come utilizzatore non registrato, voglio accedere alla pagina di registrazione dell'applicazione, dove selezionerò un username ed una password.
All'avvio dell'applicazione, se ho già effettuato in precedenza la registrazione, voglio essere in grado di autenticarmi all'applicazione.
Come utente autenticato voglio ricevere in tempo reale i messaggi che mi sono inoltrati da altri utenti.
Come utente registrato, quando effetto il login alla piattaforma voglio ricevere tutti i messaggi che avrei dovuto ricevere mentre ero offline.
Come utente autenticato voglio poter trasmettere ad altri utenti, indicando il loro username, messaggi testuali.
Come utente loggato voglio poter inserire in black-list uno o più utenti, non ricevendo messaggi e notifiche da questi.
Come utente loggato voglio poter eliminare la chat con altri utenti.
Come utente voglio poter visualizzare le informazioni pubbliche di altri utenti, come ad esempio l'ultimo accesso o la foto profilo.

1.5 Requisiti non funzionali

Gli attributi di qualità del software rappresentano delle proprietà testabili e misurabili di un sistema che indicano a che grado il sistema stesso soddisfa le esigenze dei suoi clienti al di là delle funzioni basiche del sistema. Costituiscono, dunque, dei parametri da tenere sotto controllo al fine di ottenere un software di qualità. Gli scenari consentono di specificare e dichiarare gli attributi di qualità del sistema software (o in generale dei suoi componenti) attraverso una rappresentazione che consiste in 5 elementi:

- **Stimulus:** Evento che arriva al sistema o al progetto
- **Stimulus source:** Una entità che genera lo stimolo
- **Response:** Attività che occorre come risultato dell'arrivo di uno stimolo (ciò che si intende soddisfare)
- **Response measure:** Misura la risposta e testa lo scenario
- **Environment:** L'insieme di circostanze in cui lo scenario avviene
- **Artifact:** Target a cui arriva lo stimolo

Usando questo approccio, ogni team di sviluppo ha una rappresentazione (e misurazione) dei requisiti di qualità utile per il testing ma anche per il confronto con gli stakeholders (in fase di analisi) a seconda del livello di dettaglio.

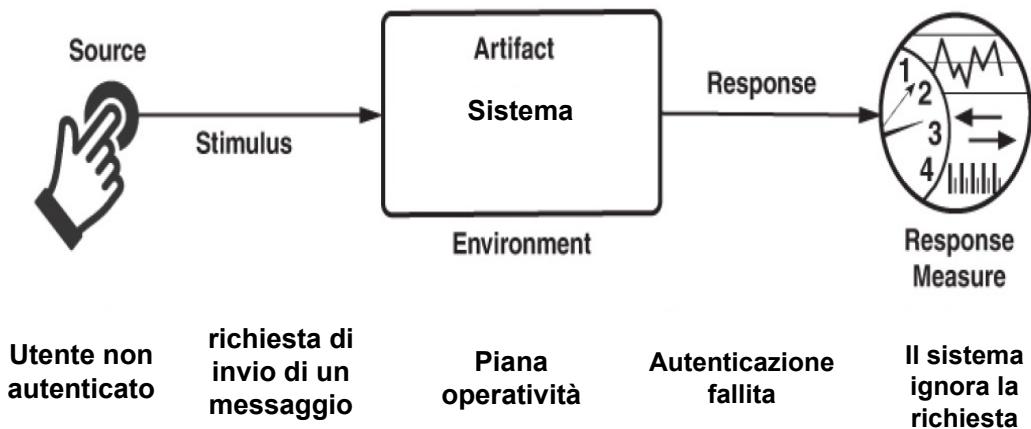
1.5.1 Sicurezza

La sicurezza è la misura dell'abilità di un sistema di proteggere i dati e le informazioni da accessi non autorizzati fornendo al contempo accesso a persone e sistemi con autorizzazione. In particolare per il seguente sistema software, dal punto di vista della security, si richiedono i seguenti attributi qualità:

- Autenticazione: un utente non autenticato non deve poter accedere al servizio, dunque innanzitutto non deve essere in grado di inviare un messaggio ad un altro utente.
- Riservatezza dei messaggi: Un messaggio deve essere visibile in chiaro solo al mittente ed al destinatario dello stesso, dunque deve essere pensato un apposito meccanismo di crittografia dei messaggi.
- Integrità: un messaggio non deve poter essere manipolato da utenti non autorizzati

Autenticazione

Il primo scenario preso in considerazione prevede il tentativo da parte di un utente non autorizzato di inviare di un messaggio. Se l'autenticazione non è andata a buon fine allora il sistema ignora la richiesta.



Riservatezza dei messaggi

Il secondo scenario analizzato prevede un utente malintenzionato che tenta di accedere al messaggio trasmesso tramite la rete. Esso può intercettare il messaggio ma non deve essere in grado di decifrarlo.

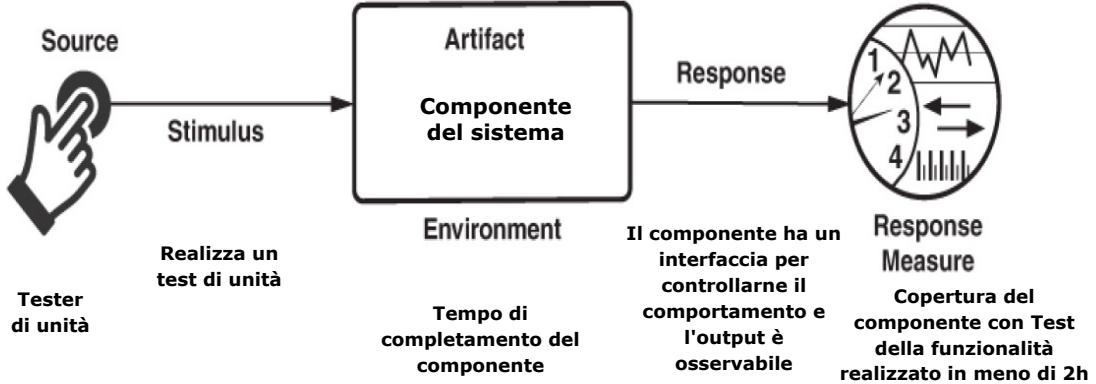


1.5.2 Testability

La testabilità del software è il grado in cui un artefatto software (ad esempio un sistema o un modulo software) supporta il test in un determinato contesto. La testabilità dei componenti software (moduli, classi) è determinata da diversi fattori quali:

- Controllabilità: il grado in cui è possibile controllare lo stato del componente sottoposto a test (CUT).
- Osservabilità: il grado in cui è possibile osservare i risultati del test (intermedio e finale).
- Isolabilità: il grado in cui il componente sottoposto a test può essere testato in isolamento.
- Separazione delle preoccupazioni: il grado in cui il componente sottoposto a test ha una responsabilità unica e ben definita.
- Comprensibilità: il grado in cui il componente sottoposto a test è documentato.
- Automatizzabilità: il grado in cui è possibile automatizzare il test del componente sottoposto a test.

Se la testabilità dell'artefatto software è elevata, è più facile trovare i guasti nel sistema mediante testing. Si riporta di seguito uno scenario per il testing di unità.



1.5.3 Usabilità

L’usabilità rappresenta la capacità del prodotto software di essere capito, appreso, usato e gradito all’utente. Nel caso specifico è desiderabile che il software da realizzare abbia le seguenti caratteristiche:

- Comprensibilità (understandability): capacità del prodotto software di permettere all’utente di comprendere se il software è appropriato, e come esso possa essere usato per particolari compiti e condizioni d’uso
- Apprendibilità (learnability): capacità del prodotto software di permettere all’utente di imparare ad usare l’applicazione.
- Attrattività (attractiveness): capacità del prodotto software di essere attraente all’utente (cioè avere un livello di gradimento nell’utilizzo).

1.5.4 Modifiability

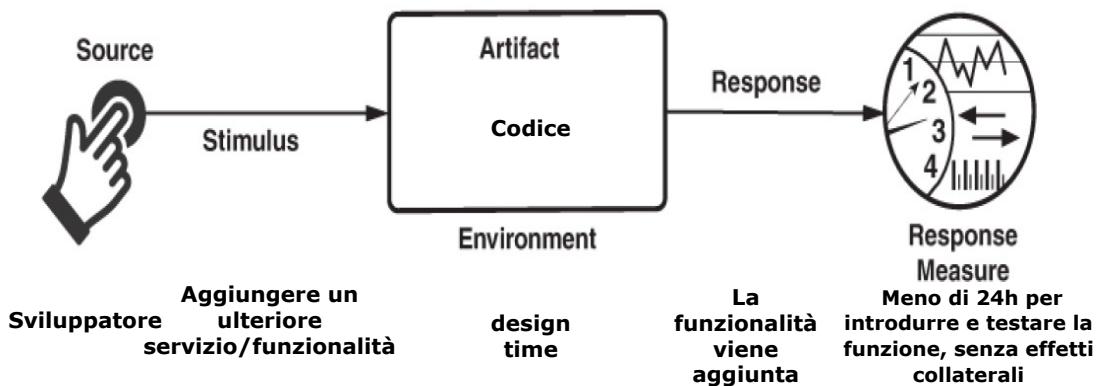
La maggior parte dei costi associati ad un sistema software sono concentrati nella fase successiva al rilascio dello stesso. Difatti, cambiamenti dopo il primo rilascio, possono avvenire al fine di aggiungere nuove funzionalità, modificare quelle già esistenti, fixare bug, migliorare le performance, migliorare la security e altro. In particolare potranno essere usate le seguenti tattiche al fine di ottenere un buon livello di modificabilità:

- **Incremento della coesione (cohesion)**: ottenuta attraverso la suddivisione in moduli software coesi funzionalmente con una opportuna distribuzione di responsabilità tra di essi.
- **Riduzione accoppiamento (coupling)**: ottenuta con opportune tecniche di encapsulamento dei dati e delle funzionalità (attraverso interfacce di accesso ad essi) e riducendo il più possibile le dipendenze tra i componenti con opportune scelte progettuali, come anche la possibilità di introdurre degli intermediari.

Buona parte delle funzionalità previste dal nostro software si prevede non cambieranno durante il suo ciclo di vita. Ma, soprattutto in termini di qualità del servizio, oltre che per le possibili funzionalità da aggiungere, il sistema potrebbe mutare, per questo, si vuole fornire un alto grado di modificabilità.

Aggiunta di un servizio/funzionalità

Lo scenario presentato in figura prevede uno sviluppatore che intende aggiungere un servizio/funzionalità a design time. Lo sforzo richiede meno di 24 ore e non ha effetti collaterali.



Scalability

La scalabilità si riferisce alla capacità da parte del sistema di adattarsi a cambiamenti nel carico di utilizzo.

Ad esempio nel caso di un sistema software con architettura Client-Server riguarda l'adattamento ad aumenti di richieste da parte dei client o aumenti del numero di client stessi connessi al server.

- Horizontal Scalability: non è possibile ottenerla se, ad esempio, nel caso di un'architettura client-server, il server risulta essere disponibile in una sola istanza single-threaded.
- Vertical Scalability: nel caso di un'architettura Client-Server, è possibile aumentare lo storage del database atto a mantenere la persistenza dei dati del sistema o aumentare la dimensione della memoria centrale (RAM) del client al fine di aumentare la capacità di gestione dei messaggi in locale.

1.6 Vincoli

Per poter usare l'applicazione, l'utente deve registrarsi al servizio.

L'applicazione è supportata sulle piattaforme Android dalla versione 5.0 in poi.

L'applicazione deve essere sviluppata rispettando le regole imposte dal GDPR (General Data Protection Regulation) per il rispetto della privacy dell'utente e i termini contrattuali relativi al Google Play Store. I Terms of Service e la Privacy Policy verranno letti e accettati dall'utente in fase di registrazione.

1.7 Glossario

Si riporta di seguito la descrizione del glossario, contenente tutti i termini specifici utilizzati durante lo sviluppo del progetto:

Termine	Alias	Descrizione	Formato
Hermes		Nome dell'applicazione	
Username	Nome Utente	Nome che indica un utente	Stringa alfanumerica
Token		Strumento per la verifica dell'autenticazione	Stringa alfanumerica
Token di notifica	NotifyToken	Token usato per ricevere le notifiche sullo smartphone	Stringa alfanumerica
PuK	Publik Key	Chiave pubblica per l'implementazione di RSA	Stringa alfanumerica
PrK	Private Key	Chiave Privata per l'implementazione di RSA	Stringa alfanumerica
RSA	Rivest-Shamir-Adleman	Algoritmo a crittografia asimmetrica	
Crittografia end-to-end		Metodo per crittografare i messaggi scambiati	
Android		Un sistema operativo per dispositivi mobili	

1.8 Stima dei costi

Viene ora presentata una stima dei costi, in termini di fattibilità, attraverso il calcolo degli Use Case Points.

Unadjusted Use-Case Weight (UUCW)

Elenco de casi d'uso con complessità stimata associata:

Use-case	Complessità
Effettua registrazione	complex
Effettua login	complex
Invia messaggio	complex
Modifica dati profilo	average
Effettua logout	average
Elimina chat	simple
Blocca/sblocca utente	complex

Tabella UUCW

Complessità	Peso	Numero UC	Prodotto
Simple	5	1	5
Average	10	2	20
Complex	15	4	60
Totale			85

Unadjusted Actor Weight (UAW)

Tipi di attori:

Tipo	Esempio	Peso
Semplice	sistema esterno tramite API	1
Medio	sistema esterno tramite protocollo	2
Complesso	persona tramite interfaccia grafica	3

Elenco attori con tipo stimato associato:

Attore	Tipo
Utente non registrato	3
Utente registrato	3
Utente autenticato	3

Tipo	Peso	Numero Attori	Prodotto
Semplice	1	0	0
Medio	2	0	0
Complesso	3	3	9
Totale			9

Unadjusted Use-Case Points (UUCP)

$$UUCP = UUCW + UAW = 85 + 9 = 94 \quad (1.1)$$

Aggiustamenti per la Complessità Tecnica (requisiti non funzionali)

Fattori	Peso	Valutazione	Impatto
Sistema distribuito	2	4	8
Obiettivi di performance	2	5	10
Efficienza end-user	1	4	4
Elaborazione complessa	1	1	1
Codice riusabile	1	1	1
Facile da installare	0.5	5	2.5
Facile da usare	0.5	4	2
Portable	2	2	4
Facile da modificare	1	3	3
Uso concorrente	1	4	4
Sicurezza	1	4	4
Accesso a terze parti	1	0	0
Necessità addestramento	1	0	0
Totale			43.5

Calcolo del Technical Complexity Factor

$$TCF = 0.6 + (0.01 \times TFactor) = 0.6 + (0.01 \times 43.5) = 0.6 + 0.435 = 1.035 \quad (1.2)$$

Aggiustamenti per la complessità dell'ambiente di lavoro

Fattore	Peso	Valutazione	Impatto
Familiare con il processo di sviluppo	1.5	1	1.5
Esperienza sull'applicazione	0.5	4	2
Esperienza sull'Object-Orientation	1	4	4
Capacità dell'analista	0.5	1	0.5
Motivazione	1	5	5
Requisiti stabili	2	4	8
Staff part-time	-1	0	0
Linguaggio di programmazione difficile	-1	3	-3
Totale			18

$$EF = 1.4 + (-0.03 \times EFactor) = 1.4 + (-0.03 \times 18) = 1.4 - 0.54 = 0.86 \quad (1.3)$$

Calcolo Finale degli Use Case Points (UCP)

$$UCP = UUCP \times TCF \times EF = 94 \times 1.035 \times 0.86 = 83.2652 \quad (1.4)$$

Calcolo delle ore di lavoro totali

$$UCP \times 16 = 1332.2432 \quad (1.5)$$

Nota: supponendo che un singolo UC venga sviluppato in 16 ore complessive mediamente.

Numero iterazioni

Per un lavoratore 60 ore a settimana (Wh), per 4 lavoratori 240 ore a settimana (TWh). Se **Th** è il numero di ore di lavoro totali necessarie a terminare il progetto e **TWh** il numero di ore a settimana del gruppo di lavoro (team), allora il numero di settimane necessarie per terminare il lavoro è dato da: $\frac{Th}{TWh} = 5,55$. Dunque si stima che occorreranno circa 5 settimane e mezzo, ovvero 3 iterazioni, per sviluppare questo progetto nella sua interezza. (*Nota:* si sta supponendo che un lavoratore lavori 8 ore al giorno tutti i giorni della settimana, più qualche ora aggiuntiva la mattina o la sera durante la settimana).

1.9 System Context Diagram

Un diagramma di contesto è impiegato per evidenziare i confini del sistema col suo ambiente e mostrare come le **entità esterne** al sistema stesso interagiscano con esso. Tale diagramma può essere realizzato in fase di analisi del problema per aiutare i progettisti ad avere una visione di insieme dello scopo del sistema software, al fine ,ad esempio, di poter capire quale possa essere il modo migliore di progettarlo. Si tratta chiaramente di un diagramma di alto livello, quindi non approfondisce gli aspetti interni di dettaglio di progettazione (ancora da stabilire), bensì si limita ad una vista semplice e chiara dall'esterno del sistema software, in tal caso mostrando unicamente le **User Interfaces** con le quali le entità esterne interagiscono (sfruttando appositi dispositivi). Difatti tale diagramma deve potere essere compreso dagli stakeholders in quanto si tratta di un primo strumento di confronto diretto con essi. Le frecce utilizzate per collegare entità ed interfacce rappresentano un flusso di dati (un system context diagram è il più alto livello di astrazione che si può avere per un diagramma data flow), difatti veicolano gli input forniti dall'esterno dagli attori e gli output corrispondenti forniti dal sistema software.

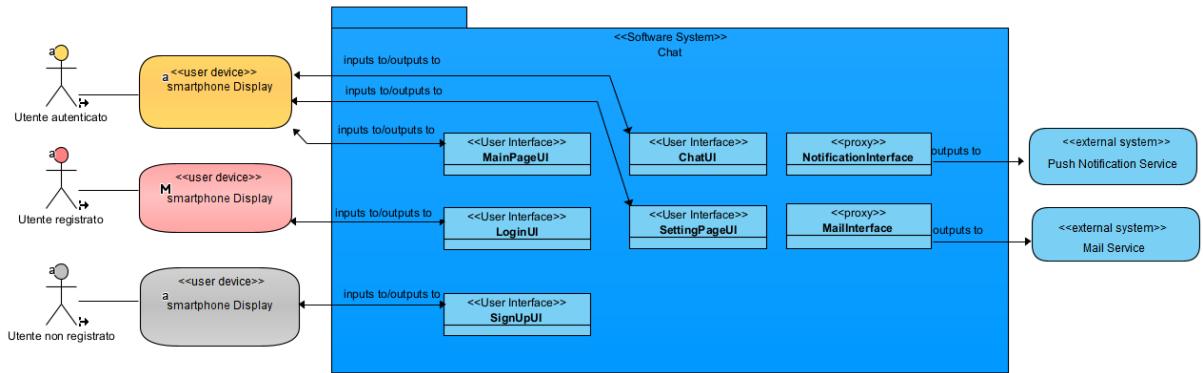


Figura 1.1: System Context Diagram

Tramite il display dello smartphone *l'utente non registrato* può fornire in input all'apposita interfaccia (*SignUpUI*) le informazioni con le quali intende registrarsi. Quest'ultima, in caso di corretta registrazione, mostrerà all'utente la pagina di log-in attraverso la quale esso potrà effettuare l'accesso al servizio. A questo punto *l'utente registrato* può fornire alla *LoginUI* i propri dati di accesso e, se corretti, essere rediretto verso la interfaccia principale contenente la lista delle conversazioni di cui fa parte (*MainPageUI*). Da qui *l'utente autenticato*, interagendo tramite il display dello smartphone con la pagina principale, può selezionare una chat ed accedere alla *ChatUI* oppure visualizzare l'interfaccia per le impostazioni (*SettingPageUI*). Inoltre il sistema sfrutta dei servizi esterni per il meccanismo di notifica dell'arrivo di un nuovo messaggio e della verifica dell'utente tramite e-mail. Difatti grazie al proxy *"NotificationInterface"* il sistema software interagisce con il servizio esterno "**Push Notification Service**" passandogli le informazioni necessarie all'invio delle notifiche. Tramite il proxy *MailInterface*, invece, si utilizza il servizio esterno "**Mail Service**" per inoltrare la mail di autenticazione dell'account.

Capitolo 2

Processo di sviluppo

Un Processo Software è un insieme strutturato di attività necessarie per lo sviluppo di un sistema software (specificazione, progettazione, sviluppo, validazione, evoluzione dopo il rilascio...).

Al fine di poter realizzare il progetto, è stato necessario utilizzare strumenti e pratiche ad hoc per organizzare al meglio il lavoro. Per questo motivo si mostrano in questo capitolo tutte le tecnologie ed i tool utilizzati durante lo sviluppo del progetto e le pratiche agili adoperate per migliorare lo sviluppo dell'applicazione. In particolare, si è deciso di adottare un processo di sviluppo di tipo Agile (e quindi non plan-based/guidato dai piani) per cui la pianificazione è incrementale e risulta più semplice modificare il processo in modo tale da riflettere e adattarsi alle mutevoli esigenze del cliente. A supporto di questa metodologia, durante il processo di progettazione, sono stati realizzati dei prototipi ad-hoc allo scopo di realizzare al meglio i requisiti e per prendere familiarità con le tecnologie utilizzate.

2.1 UP

UP (Unified Process) è un framework di processo di sviluppo software iterativo ed incrementale. Infatti, è una metodologia che prevede lo sviluppo del software come un'attività guidata dalla definizione dei requisiti funzionali, espressi attraverso i casi d'uso. L'analisi dei casi d'uso, di conseguenza, permette di definire le caratteristiche dell'architettura software che li realizza in modo integrato. Esso è basato sull'ampliamento e sul raffinamento di un sistema attraverso diverse iterazioni, con feedback e adattamenti ciclici. Il sistema è sviluppato in maniera incrementale col passare del tempo, iterazione per iterazione, e infatti questo approccio è anche conosciuto come sviluppo di software iterativo ed incrementale. Le iterazioni sono divise su quattro fasi ciascuna delle quali consiste in una o più iterazioni (figura 2.1):

- **Inception:** la prima e la fase più breve nel progetto. E' utilizzata per preparare la base del progetto, che include: stabilire lo *scope* del progetto, definire i *vincoli*, creare la tabella *Attori-Obiettivi*, delineare i *requisiti* chiave e le possibili soluzioni architettoniche insieme ai *compromessi*

di progettazione. Una durata eccessivamente prolungata della fase di inception potrebbe essere sintomo di una mancata chiarezza, da parte degli stakeholders, della visione e degli obiettivi del progetto. Senza obiettivi e visione chiari, il progetto molto probabilmente è destinato a fallire. In questo scenario è meglio prendere una pausa all'inizio del progetto per raffinare visione e obiettivi. In caso contrario, ciò potrebbe portare a ritardi di organizzazione non benevoli per le fasi successive.

- **Elaborazione:** durante questa fase, il team deve elencare la maggior parte dei requisiti di sistema (per esempio, nella forma di use case), eseguire una analisi dei rischi identificati e definire un piano di risk management per ridurne o eliminarne l'impatto sulla schedule finale e sul prodotto, stabilire la progettazione e l'architettura (utilizzando class diagram di base, package diagram o deployment diagram), creare un piano (calendario, stime dei costi, ecc.) per la fase successiva (costruzione).
- **Costruzione:** la fase più lunga e più ampia di UP. Durante questa fase, la progettazione del sistema viene finalizzata e perfezionata e il sistema viene costruito utilizzando le basi create durante la fase di elaborazione. La fase di costruzione è suddivisa in più iterazioni, ognuna delle quali deve portare a un rilascio eseguibile del sistema. L'iterazione finale della fase di costruzione permette di ottenere il sistema completo, che deve essere distribuito durante la fase di transizione.
- **Transizione:** fase finale del progetto che consegna il nuovo sistema agli utenti finali. La fase di transizione comprende anche la migrazione dei dati dai sistemi legacy e la formazione degli utenti.

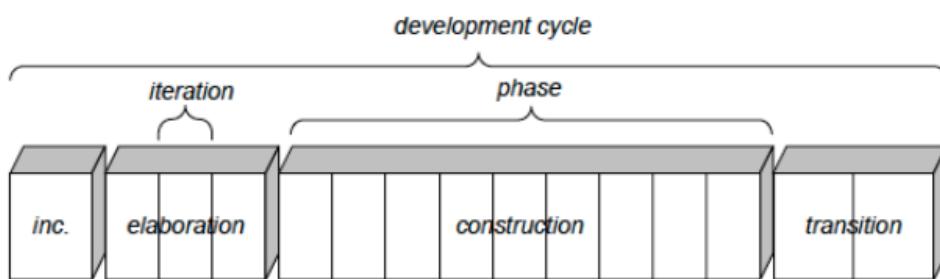


Figura 2.1: Fasi dell'Unified Process

Chiaramente questo modello di sviluppo non è restrittivo né sulla realizzazione di queste fasi, né sul numero di iterazioni da compiere in ogni fase. UP è quindi aperto all'uso di pratiche agili e prevede l'uso di **VCS** (Version Control System) per mantenere sempre una versione funzionante del software. Il ciclo di sviluppo dell'applicazione sfrutta le idee di UP, in quanto la prima fase, che corrisponde a quella di Inception in UP, ha permesso di porre le basi del progetto, con la scrittura del Vision Document, del Glossario e di una prima scrittura dei casi d'uso in formato breve. Da lì in poi lo sviluppo è sempre avvenuto in maniera incrementale e attraverso iterazioni, ma utilizzando un mix di tecniche riprese da due metodi di sviluppo software, SCRUM e l'eXtreme Programming (XP).

2.2 SCRUM

Scrum è un framework agile per la gestione del ciclo di sviluppo del software, iterativo ed incrementale, concepito per la gestione dei progetti e dei prodotti software. Non è una tecnica o un processo, ma un framework all'interno del quale è possibile usare più processi e tecniche.

Per quanto riguarda il progetto, Scrum rappresenta il framework da cui sono stati attinti più processi e tecniche. In primis, dopo la fase di Inception, sono stati sviluppati 2 cicli di Sprint, per cui in ogni ciclo viene realizzato un incremento del sistema.

Lo Sprint è un periodo di tempo limitato durante il quale viene creato un incremento di prodotto utilizzabile e potenzialmente rilasciabile. Gli Sprint hanno durata costante durante il progetto, nel nostro caso equivale a 14 giorni. Inoltre, alla chiusura di uno Sprint, si avvia immediatamente il successivo.

Un'altra tecnica di Scrum utilizzata nel ciclo di sviluppo è il Product Backlog, un elenco di voci per portare a termine il prodotto, ordinato per priorità, che viene aggiornato man mano che il lavoro avanza. Quindi, all'inizio di ciascuno Sprint, il Team seleziona dal Backlog un insieme di voci da sviluppare in quella iterazione (lo Sprint Goal) e crea lo Sprint Backlog, ovvero i compiti da svolgere per arrivare all'obiettivo dello Sprint. Ciò è stato semplificato dall'utilizzo di una Task Board, offerta da Trello per tenere traccia del lavoro svolto. Inoltre, per avere una stima dell'effort per ogni attività, è stata utilizzata la feature di inserimento delle etichette di Trello.



Figura 2.2: Esempio del Sprint Backlog del progetto. Si noti come ad ogni colore è associato un certo grado di difficoltà del requisito

Infine, tra uno Sprint ed il successivo sono state organizzate delle Sprint Retrospective. Una Sprint

Retrospective è un meeting, in cui il team analizza le proprie attività e crea un piano di miglioramenti da attuare per il prossimo Sprint. In questo contesto, si cerca di aumentare la qualità del prodotto migliorando i processi di lavoro. In particolare, è stato discusso:

- Cosa è andato bene durante lo Sprint
- Cosa potrebbe essere migliorato
- Cosa ci si impegna a migliorare nel prossimo Sprint

2.3 eXtreme Programming

XP è una delle prime metodologie agili a essersi affermate nella comunità software, adottando un approccio estremo per lo sviluppo iterativo. Infatti, nuove versioni del sistema possono essere sviluppate, integrate e testate diverse volte al giorno. Gli incrementi vengono rilasciati ai clienti ogni 2 settimane e tutti i test devono essere eseguiti ad ogni build del sistema.

XP è caratterizzato da 13 pratiche:

1. I requisiti sono “user stories”
2. La metafora di riferimento
3. Il “planning game”
4. “Prima i test, poi il codice”
5. Programmazione a coppie
6. Integrazione continua
7. Refactoring
8. Progettazione semplice
9. Proprietà collettiva del codice
10. Cliente On-site
11. Piccoli rilasci
12. Settimana di 40 ore di lavoro
13. Uso sistematico di standard di codifica

Di questi, nel progetto sono stati utilizzati:

- **Integrazione continua:** integrazione giornaliera del codice effettuata tramite il repository condìsivo su GitHub, da cui venivano importate in locale le modifiche eseguite il giorno precedente;
- **Refactoring:** pulizia del codice per migliorarne alcune caratteristiche non funzionali senza modifi-carne il comportamento esterno; nel caso di questo progetto, veniva effettuata una continua pulizia dei cosiddetti “bad smells” del codice in team, attraverso delle riunioni su Microsoft Teams;
- **Programmazione a coppie:** tecnica nella quale due programmatore lavorano insieme alla stessa postazione di lavoro; il primo scrive il codice e l’altro svolge il ruolo di supervisore, tecnica che è stata di fondamentale importanza per il debug e per la risoluzione di errori di codice.

2.4 Tool per la condivisione del lavoro

Microsoft Teams Questo strumento (figura 2.3) ha permesso al team di lavorare online, per cui i membri del team hanno comunicato attraverso delle semplice call vocali sul software, ognuno installato sul proprio PC personale, per la maggior parte degli appuntamenti del ciclo di sviluppo del progetto. Inoltre, è stata ampiamente sfruttata la funzionalità di condivisione schermo, grazie alla quale ogni membro del gruppo poteva condividere facilmente il proprio lavoro agli altri colleghi.

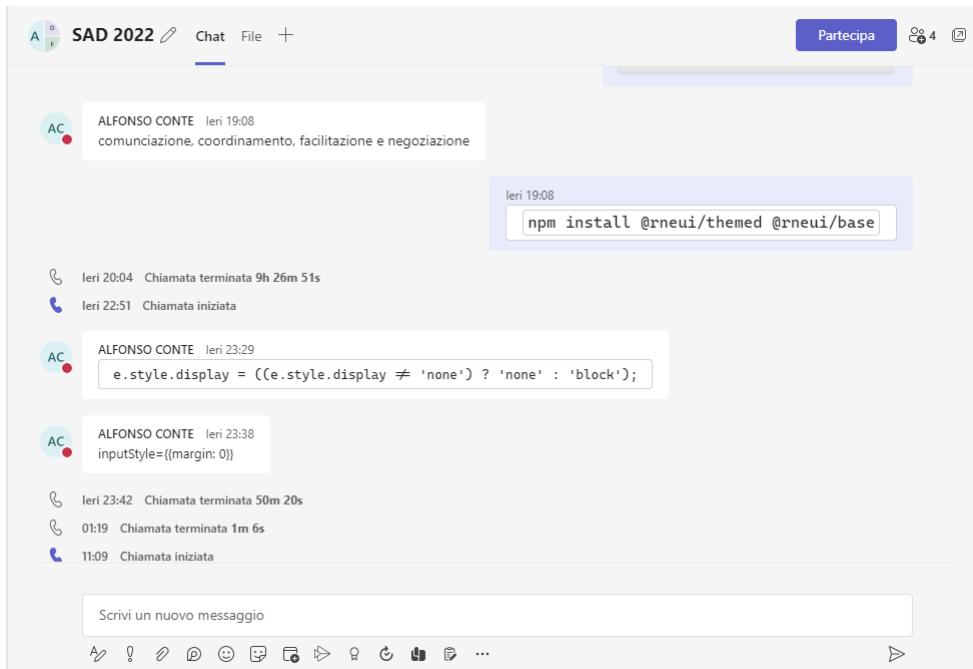


Figura 2.3: Esempio di utilizzo di Microsoft Teams

Trello Trello (figura 2.4) è un tool gratuito per il project management estremamente semplice da utilizzare e particolarmente utile per lavorare in maniera agile. Esso ha consentito la divisione del lavoro in iterazioni, in cui volta per volta venivano raffinati analisi dei requisiti e sviluppo del codice. Trello è stato utilizzato per tenere traccia dei compiti da realizzare in ogni Sprint: infatti esso mette a disposizione

CAPITOLO 2. PROCESSO DI SVILUPPO

delle lavagne virtuali all'interno delle quali è possibile creare delle schede personalizzabili, in cui sono state inserite tutte le attività da fare per il progetto, indicandone anche la data di scadenza e la difficoltà.

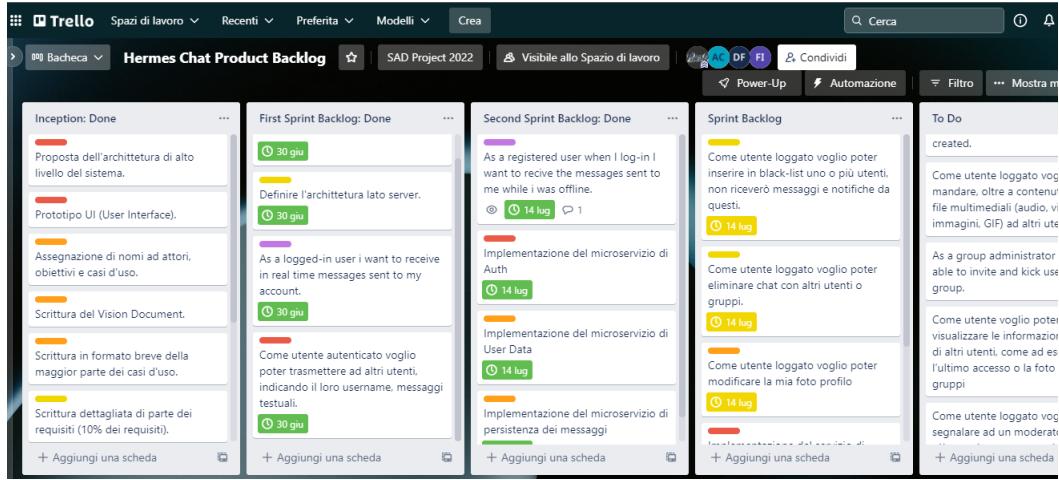


Figura 2.4: Esempio di utilizzo di Trello per lo sviluppo tramite Sprint Backlog

Nel nostro caso, sono state create due bacheche relative ai requisiti completati in ognuna delle due iterazioni. In più, avevamo una bachecca per lo Sprint Backlog attuale e un'ultima bachecca in cui avevamo raccolto inizialmente tutti i requisiti da realizzare. Da quest'ultima ne venivano selezionati una parte per essere inseriti nello Sprint Backlog. Per quanto riguarda una singola scheda (figura 2.5), come già detto, essa comprende un singolo requisito. Inoltre, riporta un'etichetta che a seconda del colore rappresenta un determinato livello di difficoltà, una data di scadenza e il tag dei membri interessati nello sviluppo di quella determinata feature. Opzionalmente, può riportare anche una descrizione aggiuntiva relativa ai dettagli di progettazione o di sviluppo.



Figura 2.5: Esempio di scheda utilizzata su Trello, con scadenza e relativa difficoltà

GitHub GitHub (figura 2.6) è stato utilizzato per la gestione del codice, in particolare per la condivisione dello stesso tra i diversi membri del team e per ottenere un repository chiaro ed ordinato di tutta l'implementazione del sistema. Ogni volta che un membro effettua una modifica dei file memorizzati nel repository, esegue una push sul repository condiviso. A questo punto, gli altri membri possono eseguire una pull per ottenere, in locale, le modifiche effettuate. Inoltre, una funzionalità molto utile di GitHub è quella di visualizzare tutti i commit effettuati, in modo è possibile ottenere praticamente tutte le versioni del sistema, dal primo prototipo fino alla release finale. Infatti, questa feature garantisce una semplice risoluzione dei conflitti di modifiche, laddove presenti, e anche la possibilità di ritornare indietro (rollback) nelle scelte effettuate. Per semplificare ulteriormente il workflow del progetto, si è fatto uso della versione desktop di GitHub, GitHub Desktop, scaricata in locale da ognuno dei membri del team.

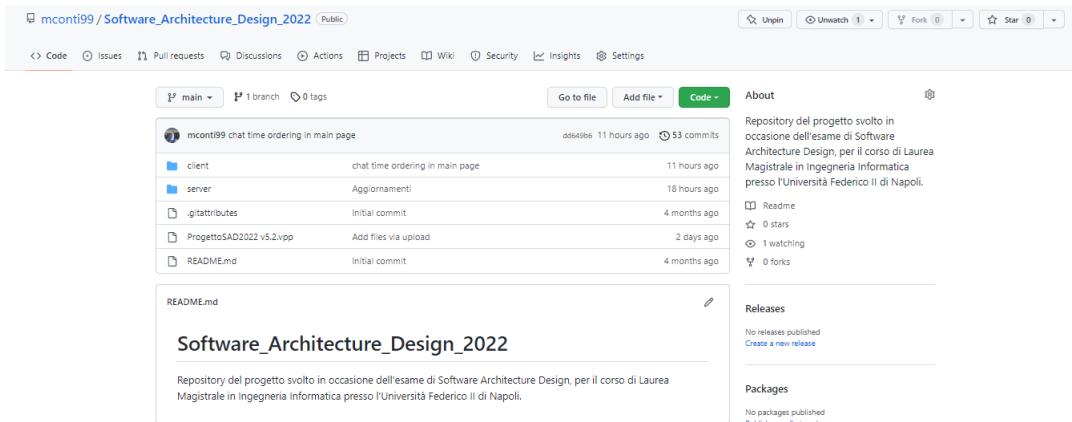


Figura 2.6: Repository GitHub pubblico

Di seguito viene riportato un codice QR che reindirizza direttamente alla home del repository creato su GitHub.



Overleaf Overleaf è un editor L^AT_EX collaborativo basato su cloud utilizzato per scrivere, modificare e pubblicare documenti. Nel progetto è stato utilizzato al fine di poter lavorare contemporaneamente su di un unico documento condiviso al fine di realizzare la documentazione progettuale.



2.5 Tool e tecnologie per lo sviluppo

Visual Paradigm Per progettare diagrammi aderenti allo standard UML che rappresentassero le viste più adeguate alla Architettura Software è stato utilizzato Visual Paradigm. Esso, attraverso la creazione di un progetto, ha consentito di creare Class Diagram, Sequence Diagram, Activity Diagram e Use Case Diagram per la documentazione relativa al sistema (figura 2.7). La modellazione effettuata su Visual Paradigm è allegata a questa documentazione nella folder di progetto.

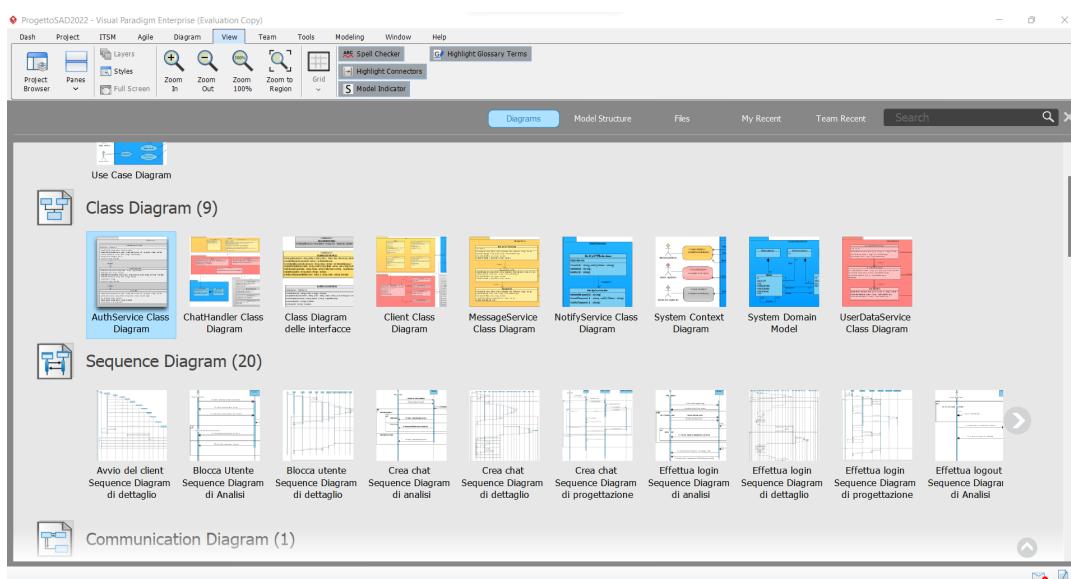


Figura 2.7: Schermata di visualizzazione diagrammi creati

React Native React Native è un framework per applicazioni mobili open-source creato da *Meta Platforms Inc.* Viene utilizzato per sviluppare applicazioni per Android e iOS consentendo agli sviluppatori di utilizzare un'unica codebase scritta in JavaScript. Nel progetto è stato utilizzato principalmente per le interfacce grafiche dell'applicazione mobile.



JavaScript JavaScript è un linguaggio di programmazione multi paradigma orientato agli eventi, comunemente utilizzato nella programmazione Web lato client per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da *eventi*. Tali eventi sono innescati a loro volta, in vari modi dall'utente sulla pagina web in uso. Trova, inoltre, molte applicazioni nell'implementazione di servizi lato server. E' stato il linguaggio più utilizzato nel progetto, sia per quanto riguarda la programmazione del client e della sua logica di business, ma soprattutto per quanto riguarda il Chat Handler e i vari microservizi, scritti quasi interamente in JavaScript. E' stato scelto sia per l'elevata quantità di documentazione ed esempi in rete, sia per la sua facilità d'uso.



Node.js Node.js è un runtime system open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript. Molti dei suoi moduli base sono scritti in JavaScript. *Node.js* consente di utilizzare JavaScript per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al browser dell'utente. *Node.js* in questo modo permette di implementare il cosiddetto paradigma "*JavaScript everywhere*" (JavaScript ovunque), unificando lo sviluppo di applicazioni Web intorno ad un unico linguaggio di programmazione. Nel progetto è stato infatti utilizzato per lo sviluppo lato server dell'app.



Il modello di networking su cui si basa *Node.js* è I/O event-driven: ciò vuol dire che Node richiede al sistema operativo di ricevere notifiche al verificarsi di determinati eventi e rimane, quindi, in sleep fino alla notifica stessa. Solo in tale momento torna attivo per eseguire le istruzioni previste nella funzione di **callback**, così chiamata perché da eseguire una volta ricevuta la notifica indicante che il risultato dell'elaborazione del sistema operativo è disponibile. Tale modello di networking è ritenuto più efficiente

in situazioni critiche, come ad esempio il caso in cui si verifica un elevato traffico di rete, evento non raro in una applicazione di messaggistica come *Hermes Messenger*.

Expo Expo è un framework che estende React Native. Il suo Software Development Kit rende accessibili sia funzionalità native del dispositivo (per esempio la fotocamera e i sensori hardware) sia librerie di utilità standard (come ad esempio la predisposizione per l'autenticazione tramite i principali provider come Apple e Google).

Expo, nel modello operativo *managed workflow*, ha reso il team operativo in pochi minuti: grazie ad Expo CLI e all'app Expo Go, è stato possibile eseguire e fare debug dell'app sfruttando il simulatore iOS o Android, tramite browser Web oppure tramite un dispositivo reale. Tutto ciò senza alcuna necessità di utilizzare software di sviluppo nativo Apple (Xcode) e Android (Android Studio), grazie allo sviluppo che sfrutta React Native. Il managed workflow, che rappresenta la vera forza del framework, è particolarmente adatto a piccoli team di sviluppo, che non possono contare su personale dedicato alle attività di setup e manutenzione.

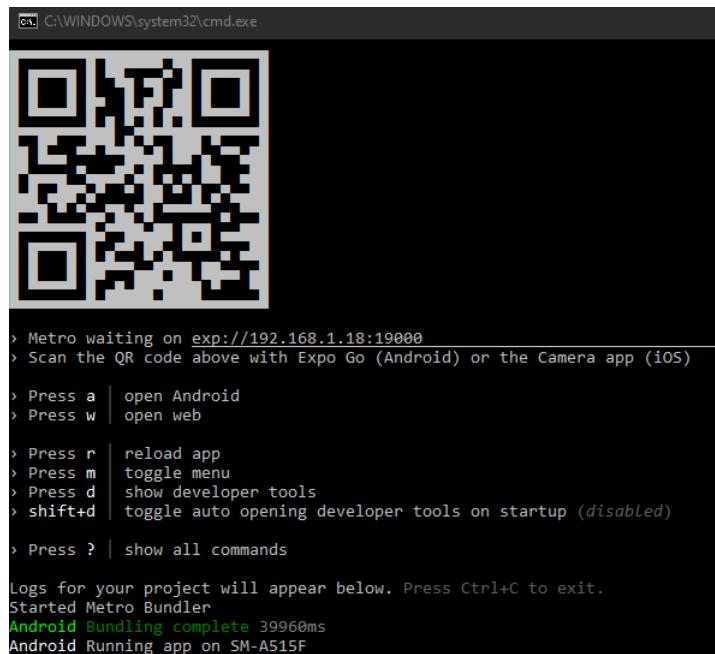


Figura 2.8: Command Prompt per utilizzo Expo

Expo Go (figura 2.9) accelera il processo di sviluppo e test poiché permette di eseguire codice JavaScript senza attendere il processo di build; permette inoltre di invitare utenti tester in modo estremamente semplice, utilizzando un codice QR o un link, e cominciare così a ricevere feedback in tempi molto brevi. Sempre per quanto riguarda il processo di sviluppo, un grande beneficio arriva dall'utilizzo di *Expo Application Services*, un servizio Cloud che automatizza il processo di build (creazione di un eseguibile) e di rilascio su App Store e Google Play Store, permettendo di risparmiare tempo altrimenti dedicato ad attività di setup e configurazione nei rispettivi ambienti di sviluppo nativi.

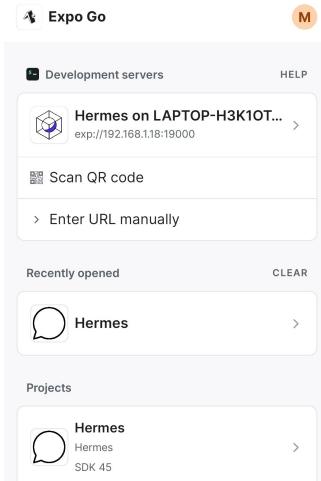


Figura 2.9: Esempio di utilizzo dell'app di Expo Go

L’aspetto più svantaggioso è che Expo limita l’utilizzo di librerie di terze parti poiché si integra unicamente con componenti resi compatibili. Nonostante questi siano in costante aumento, tale caratteristica rende necessaria un’importante riflessione nella fase di analisi iniziale di un nuovo progetto, in cui vengono definiti requisiti e specifiche.

SQLite SQLite è una libreria software scritta in linguaggio C che implementa un DBMS SQL pensato per essere estremamente leggero e adeguato per lo storage locale nel caso di applicazioni mobile. L’interfacciamento a tale database avviene attraverso delle apposite librerie fornite dall’ambiente React Native.



MySQL Si tratta del RDBMS (Relational Database Management System) utilizzato dall’ applicazione lato server. Esso è composto da un client a riga di comando e un server. MySQL è ampiamente utilizzato per la realizzazione di server web in molte applicazioni commerciali. Nel nostro caso, viene usato per la gestione della persistenza dei microservizi. Per l’accesso e la gestione del database tramite moduli Node.js viene utilizzata una apposita libreria chiamata Node-SQL.



Capitolo 3

Fase di Analisi

In questo capitolo verrà affrontata la fase di Analisi del sistema, che ha come scopo il chiarimento e la documentazione delle funzioni e dei servizi che vengono offerti dalla nostra applicazione. In questa fase sono stati realizzati, in notazione UML e sfruttando i concetti tipici della progettazione ad oggetti, diversi diagrammi. Tali diagrammi sono stati continuamente raffinati durante le varie iterazioni del processo di sviluppo adottato fino al raggiungimento di una stabilità e conformità ai requisiti specificati. L'input di questa fase è costituito dall'insieme di requisiti informali già precedentemente presentati. Si tratta di un'analisi di alto livello che ha come obiettivo la produzione di documentazione che sia, per ora, del tutto indipendente dall'implementazione.

3.1 Textual Analysis

È stata realizzata una Textual Analysis a partire dai requisiti descritti nel Documento di Visione allo scopo di trovare le principali entità di dominio e i principali requisiti funzionali dell'applicazione.

Si vuole realizzare un'applicazione di messaggistica istantanea open source basata sulla sicurezza e sulla privacy degli utenti che la utilizzano. Gli obiettivi dell'applicazione sono principalmente:

- Permettere ad un utente di **iscriversi** al servizio, di **inviare** e ricevere **messaggi** da altri utenti iscritti
- Garanzia di consegna del messaggio inviato anche qualora il destinatario sia in quel momento offline
- Garantire che la registrazione e la comunicazione avvengano in totale sicurezza, attraverso una gestione critografata delle chiavi d'accesso
- Allo scopo deve essere garantita la possibilità a nuovi utenti di registrarsi alla piattaforma. Una volta effettuata la registrazione, l'utente potrà effettuare il **login** e, in qualità di utenti autenticati, potrà **inviare** e ricevere **messaggi** la cui confidenzialità e integrità dovrà essere garantita tramite meccanismi di crittografia.
- In particolare, la creazione di una **chat** con un utente può essere creata specificando l'username del destinatario.
- Ad un utente loggato viene chiaramente data la possibilità di effettuare il logout in qualsiasi momento egli voglia.
- Il sistema software dovrà garantire la consegna real-time dei **messaggi** agli utenti online, e il successivo recapito, al momento della connessione al servizio a quelli offline.
- Ciascun utente registrato è caratterizzato da un profilo contenente, come informazioni: nome utente, email, foto profilo.
- Eventualmente sarà possibile per un utente modificare alcune delle informazioni legate al suo profilo.
- I **messaggi** trasmessi possono essere di differenti tipologie e supportare, oltre che informazioni testuali, anche **contenuti multimediali**, quali **immagini**, audio e video **messaggi**.
- Sarà, inoltre, possibile per gli utenti loggati **bloccare** eventuali utenti indesiderati, con la possibilità di **sbloccarli** in futuro.
- Il servizio di messaggistica sarà utilizzabile tramite un'apposita applicazione mobile che garantirà l'accesso alle funzionalità supportate con l'ausilio di un'interfaccia grafica.

Figura 3.1: Textual Analysis

No.	Candidate Class	Extracted Text	Type	Description	Occurrence	Highlight
1	iscriversi	iscriversi	Use Case		1	
2	inviare	inviare	Use Case		2	
3	modificare	modificare	Use Case		1	
4	login	login	Use Case		1	
5	bloccare	bloccare	Use Case		1	
6	sbloccarli	sbloccarli	Use Case		1	
7	chat	chat	Class		1	
8	creazione	creazione	Use Case		1	
9	contenuti multimediali	contenuti multimediali	Class		1	
10	utenti iscritti	utenti iscritti	Actor	Synon: Utente Registrato	1	
11	nuovi utenti	nuovi utenti	Actor	SYnon: Utente non registrato	1	
12	utente autenticato	utente autenticato	Actor		0	
13	messaggi	messaggi	Class		5	

Figura 3.2: Output Textual Analysis

3.2 Mockup e Design Concepts

In fase di Inception ci è stato utile visualizzare i principali requisiti funzionali dell'applicazione tramite un supporto grafico. Per questo motivo abbiamo realizzato, tramite VisualParadigm, dei Mockup dell'Applicazione in modo da avere un'idea del possibile risultato finale.

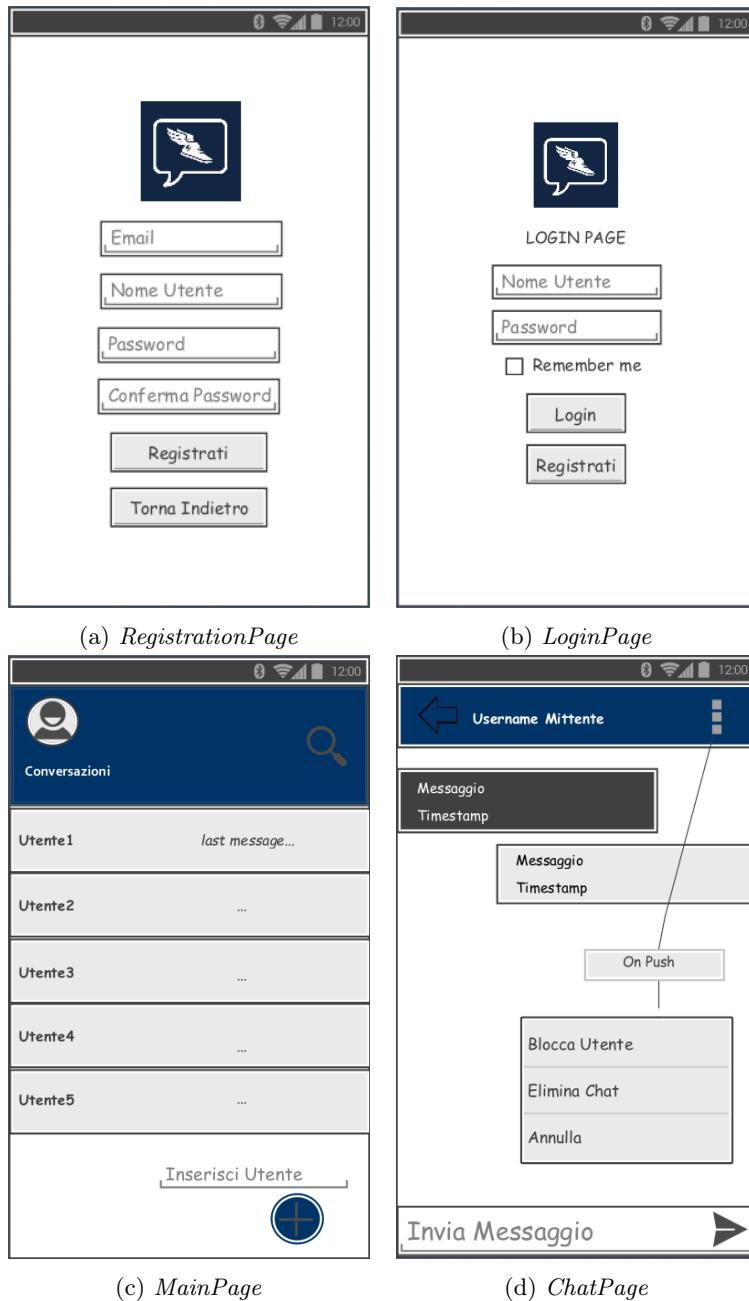


Figura 3.3: Mockups

3.3 Modello dei Casi d’Uso

I casi d’uso sono storie scritte, testuali di qualche attore che utilizza il sistema per il raggiungimento di specifici obiettivi.

3.3.1 Casi d’uso in formato dettagliato

In seguito all’individuazione dei casi d’uso, si è proceduto, durante la fase di ideazione, alla specifica in formato dettagliato di quelli con maggior valore.

Registrarsi al servizio

Livello:	Obiettivo utente
Attore primario:	Utente non registrato
Attore secondario:	Mail Service
Parti interessate e interessi:	Utente non registrato (vuole registrarsi)
Pre-condizioni:	l'utente non è registrato
Garanzia di successo:	l'utente si è registrato
Scenario Principale:	<p>-Un utente non registrato accede alla homepage dell'app</p> <p>1) Attraverso l'interfaccia grafica, clicca sul bottone "Registra ti" per effettuare la registrazione</p> <p>2) L'utente inserisce un nuovo indirizzo email</p> <p>3) Inserisce un nuovo nome utente</p> <p>4) Inserisce una password di almeno 8 caratteri</p> <p>5) Inserisce nuovamente la password</p> <p>6) Clicca sul bottone "Conferma" per confermare l'inserimento dei dati e procedere alla registrazione</p> <p>7) il sistema confermerà la avvenuta richiesta di registrazione con un messaggio a video</p> <p>8) Il sistema richiede al Mail Service di inviare una email con un link per confermare la registrazione</p> <p>9) L'Utente conferma la registrazione tramite il link ricevuto via mail</p>
Estensioni:	<p>6.a: Il nome utente già è stato utilizzato da un altro user per l'iscrizione</p> <p>7) Il sistema segnala l'errore</p> <p>8) L'utente inserisce un nuovo nome utente fino a quando esso non sarà valido</p> <p>6.b: La password non è di almeno 8 caratteri</p> <p>7) Il sistema segnala l'errore</p> <p>8) L'utente inserisce una nuova password fino a quando essa non sarà di almeno 8 caratteri</p> <p>6.c: L'utente non ha ancora confermato la registrazione tramite l'email inviata</p> <p>7) - Il sistema segnala l'errore</p>

Login

Livello:	Sottofunzione
Attore primario:	Utente registrato
Parti interessate e interessi:	Utente registrato (vuole autenticarsi)
Pre-condizioni:	L'utente è registrato
Garanzia di successo:	<p>l'utente è autenticato ed ha accesso a tutte le funzioni dell'app.</p> <p>La data di ultimo accesso viene aggiornata.</p>
Scenario Principale:	<ol style="list-style-type: none"> 1) Un utente non registrato accede alla homepage dell'app 2) Attraverso l'interfaccia grafica, inserisce il suo nome utente nell'apposita barra vuota di inserimento testo 3) Inserisce la sua password nell'apposita barra vuota 4) Clicca sul bottone "Accedi" per effettuare l'accesso 5) Il sistema confermerà l'autenticazione con un messaggio a video 6) L'utente accede alla main page dell'app, da cui potrà accedere alle varie chat attualmente disponibili ricevendo eventuali messaggi non ancora consegnati
Estensioni:	<ol style="list-style-type: none"> 4.a: Il nome utente inserito non è corretto 5) Il sistema segnala l'errore 6) L'utente inserisce nuovamente lo username fino a quando non è corrispondente a quello di un utente registrato 4.b: La password non è corretta 5) Il sistema segnala l'errore 6) L'utente inserisce una nuova password fino a quando essa non sarà corrispondente all'utente registrato 4.c: La connessione col server è fallita 5) Il sistema segnala l'indisponibilità del server 6) Il Client tenta la riconnessione al server dopo un timeInterval 4.d: L'utente non ha confermato la mail 5) Il sistema segnala un messaggio di errore 6) L'Utente verifica la mail e riprova l'accesso

Invio di un messaggio

Livello:	Obiettivo Utente
Attore primario:	Utente Autenticato
Parti interessate e interessi:	Utente Autenticato (vuole inviare un messaggio)
Pre-condizioni:	L'utente si è autenticato
Garanzia di successo:	Il messaggio viene inviato comparando all'interno della chat page
Scenario Principale:	<ul style="list-style-type: none"> - Un utente Autenticato accede alla MainPage dell'app 1) Attraverso l'interfaccia grafica, clicca sul pulsante di ricerca utente 2) Inserisce l'username dell'utente a cui vuole inviare un messaggio 3) L'utente clicca sul bottone di creazione della chat 4) Il sistema crea la chat page con l'utente destinatario 5) L'utente inserisce all'interno della barra il messaggio 6) L'utente clicca sul bottone "Invia" 7) Il sistema codifica e trasmette il messaggio al destinatario 8) Il sistema richiede al Push Notification Service di inviare una notifica al destinatario 9) Il sistema salva in locale il messaggio 10) Il sistema mostra sulla chat page il messaggio appena trasmesso
Estensioni:	<ul style="list-style-type: none"> 3.a: L'Username destinatario non esiste 4) Il sistema segnala l'errore 5) L'utente inserisce un nuovo username fino a quando esso non sarà corrispondente a un utente registrato 6.a: Il messaggio non è stato ricevuto correttamente 7) Il sistema segnala l'errore 8) L'utente ritenta la trasmissione del messaggio

Modifica dati profilo

Livello:	Sottofunzione
Attore primario:	Utente Autenticato
Parti interessate e interessi:	Utente Autenticato (vuole modificare i dati del profilo)

Pre-condizioni:	L'utente si è autenticato
Garanzia di successo:	I dati vengono aggiornati comparando nel profilo utente
Scenario Principale:	<ol style="list-style-type: none"> 1) Un utente Autenticato accede alla MainPage dell'app 2) Attraverso l'interfaccia grafica, clicca sul bottone "Profilo" 3) Il sistema mostra i dati attuali del profilo 4) L'utente carica i nuovi dati 5) Il sistema aggiorna i dati dell'utente 6) Il sistema conferma l'avvenuta modifica con un messaggio a video
Estensioni:	<ol style="list-style-type: none"> 4.a: I dati inseriti non sono validi 5) Il sistema segnala l'errore 6) L'utente ritenta l'inserimento dei dati

Logout

Livello:	Sottofunzione
Attore primario:	Utente Autenticato
Parti interessate e interessi:	Utente Autenticato (vuole effettuare il logout)
Pre-condizioni:	L'utente si è autenticato
Garanzia di successo:	Viene effettuato il logout con il reindirizzamento alla login page
Scenario Principale:	<ol style="list-style-type: none"> 1) Un utente Autenticato accede alla MainPage dell'app 2) Attraverso l'interfaccia grafica, clicca sul bottone "Logout" 3) Il sistema effettua la disconnessione 4) L'utente viene reindirizzato alla login page
Estensioni:	<ol style="list-style-type: none"> 2.a: La connessione col sistema è stata persa 3) Il sistema segnala l'errore 4) L'utente ritenta la disconnessione cliccando nuovamente sul bottone di "Logout"

Crea chat

Livello:	Obiettivo Utente
Attore primario:	Utente Autenticato
Parti interessate e interessi:	Utente Autenticato (vuole creare una chat con un altro Utente)
Pre-condizioni:	L'utente si è autenticato

Garanzia di successo:	Viene mostrata nella Main Page la chat col destinatario prescelto
Scenario Principale:	<ol style="list-style-type: none"> 1) Un utente Autenticato accede alla MainPage dell'app 2) Attraverso l'interfaccia grafica, clicca sul pulsante "Crea chat" 3) L'utente inserisce il nome dell'utente destinatario con cui vuole creare la chat e preme invio 4) L'utente trova nella Main Page la nuova chat
Estensioni:	<ol style="list-style-type: none"> 3.a: La chat col destinatario scelto già esiste 4) Il sistema segnala l'errore 5) Viene mostrata nuovamente la Main Page 3.b: L'username del destinatario prescelto non esiste 4) Il sistema segnala l'errore 5) Viene mostrata nuovamente la Main Page

Elimina chat

Livello:	Obiettivo Utente
Attore primario:	Utente Autenticato
Parti interessate e interessi:	Utente Autenticato (vuole eliminare una chat)
Pre-condizioni:	L'utente ha una chat
Garanzia di successo:	Viene eliminata la chat selezionata
Scenario Principale:	<ol style="list-style-type: none"> 1) Un utente Autenticato seleziona la chat da eliminare 2) Il sistema mostra la Chat Page corrispondente 3) L'utente clicca sul tasto "Impostazioni" e seleziona l'opzione "Elimina chat" 4) Il sistema mostra a video un messaggio di avvenuta eliminazione 5) L'utente viene reindirizzato alla Main Page

Blocca utente

Livello:	Obiettivo Utente
Attore primario:	Utente Autenticato
Parti interessate e interessi:	Utente Autenticato (vuole bloccare un altro utente)

Pre-condizioni:	L'utente ha creato una chat con l'utente che vuole bloccare
Garanzia di successo:	L'utente prescelto viene bloccato
Scenario Principale:	<ol style="list-style-type: none"> 1) Un utente Autenticato seleziona la chat dell'utente che vuole bloccare 2) Attraverso l'interfaccia grafica, clicca sul pulsante "Impostazioni" e quindi seleziona l'opzione "Blocca utente" 3) Il sistema notifica l'avvenuto blocco
Estensioni:	<ol style="list-style-type: none"> 2.a: L'operazione di blocco fallisce 3) Il sistema segnala l'errore con un messaggio a video

3.3.2 Use Case Diagram

Al fine di sintetizzare graficamente i casi d'uso definiti in fase di specifica dei requisiti, è stato realizzato un diagramma dei casi d'uso. Esso mette in luce le funzionalità offerte dal sistema, permettendo al contempo di visualizzare i diversi tipi di attori del sistema e come essi interagiscono con il sistema stesso. Ciascun caso d'uso del diagramma racchiude una serie di scenari con cui l'attore interagisce per il raggiungimento di uno specifico obiettivo.

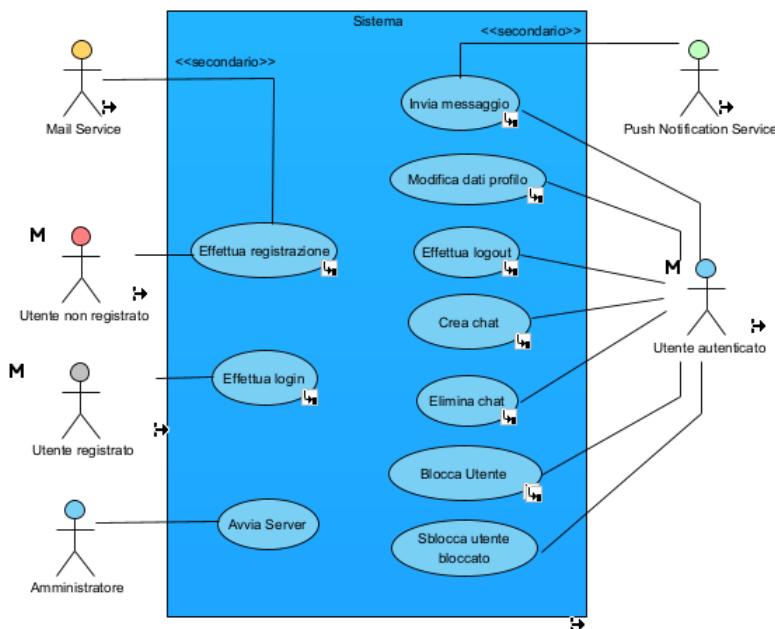


Figura 3.4: Use Case Diagram

All'interno di tale diagramma (figura 3.4), viene elencato l'insieme completo di funzionalità e servizi offerti dal sistema. Tra questi, si è poi proceduto al raffinamento e all'implementazione solo di quelli ritenuti di maggior valore.

Inoltre possiamo notare due ulteriori attori, *Mail Service* e il *Push Notification Service*, i quali rappresentano servizi esterni di cui la nostra applicazione si avvale allo scopo di inviare la mail per confermare la registrazione nel primo caso e trasmettere eventuali notifiche di ricezione di messaggi nel secondo caso.

3.4 System Sequence Diagram

Per i casi d'uso principali, dettagliati in fase di implementazione, si è fatto ricorso ad una ulteriore rappresentazione grafica che evidenzi le interazioni tra attore e sistema. La tipologia di diagramma utilizzato è il System Sequence Diagram che modella ogni scenario come una sequenza di interazioni.

3.4.1 Registrazione

Nel caso d'uso seguente si riporta l'interazione tra l' *Utente non registrato* e il *Sistema*. L'obiettivo, come detto, è consentire la registrazione. A tal scopo l'utente, cliccando sul bottone di sign-up, inserisce i dati nel formato corretto, confermando poi la registrazione tramite il link di conferma trasmesso tramite email. Vengono inoltre riportati anche 3 scenari alternativi dovuti ad un errato formato nell'inserimento dei dati da parte dell'utente.

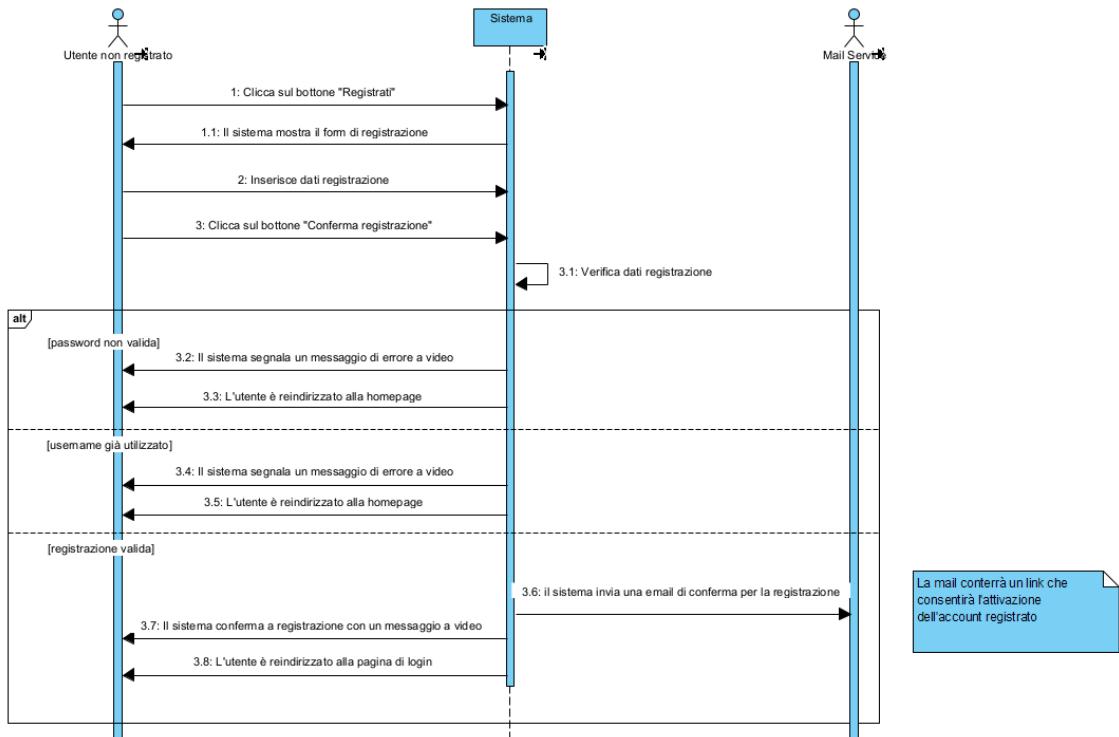


Figura 3.5: Registrazione SSD

3.4.2 Login

Altro caso d'uso che è opportuno specificare è quello relativo al Login da parte di un *Utente Registrato*. Quest'ultimo, accedendo alla homepage può inserire le proprie credenziali e cliccare sul bottone di "Login". Qualora il sistema riscontri degli errori relativi ai dati di accesso o ad una mancata conferma di registrazione, lo segnalera con un messaggio a video. Infine, se la procedura di Login avviene correttamente, l'utente si aspetta di ricevere dal sistema i messaggi ricevuti mentre era offline.

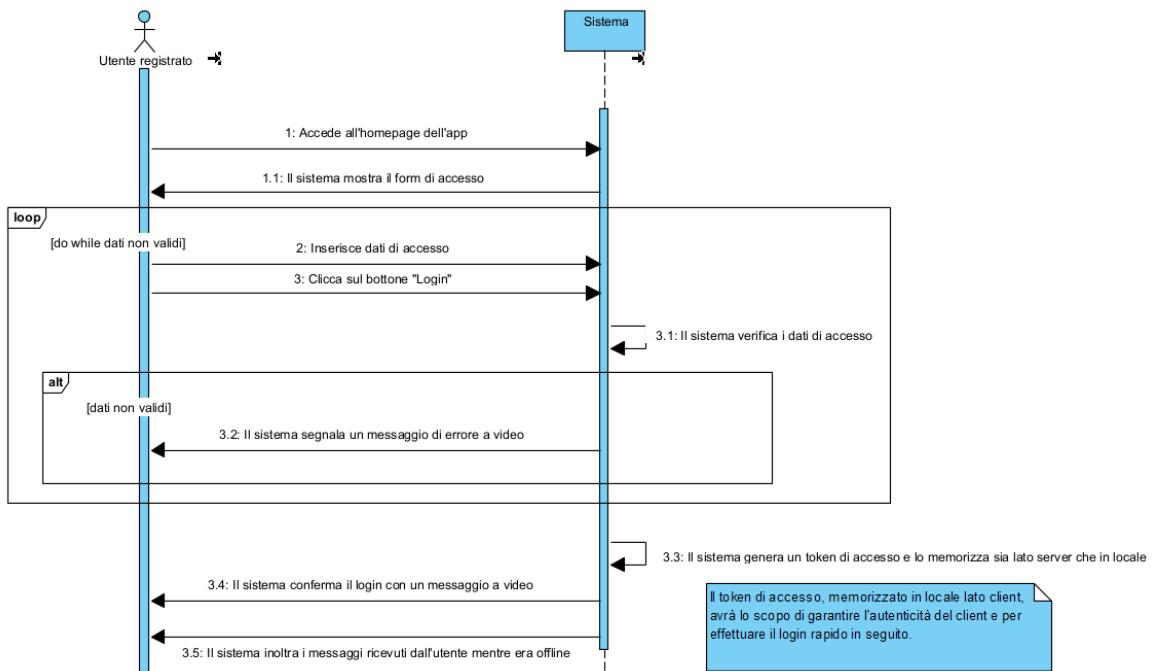


Figura 3.6: Login SSD

3.4.3 Invia Messaggio

Scopo principale di un *Utente autenticato* è, chiaramente, quello di inviare un messaggio. A tal scopo, l'utente seleziona il destinatario, il sistema creerà la Chat Page corrispondente, e a questo punto l'utente può comporre il messaggio da trasmettere. Il messaggio composto inviato al sistema sarà codificato, salvato in locale e trasmesso al destinatario (solo qualora quest'ultimo non abbia bloccato il mittente). Se tutta la procedura avviene correttamente, il sistema notifica l'utente riportando all'interno della ChatPage il messaggio appena inoltrato, in caso contrario mostrerà un messaggio di errore a video.

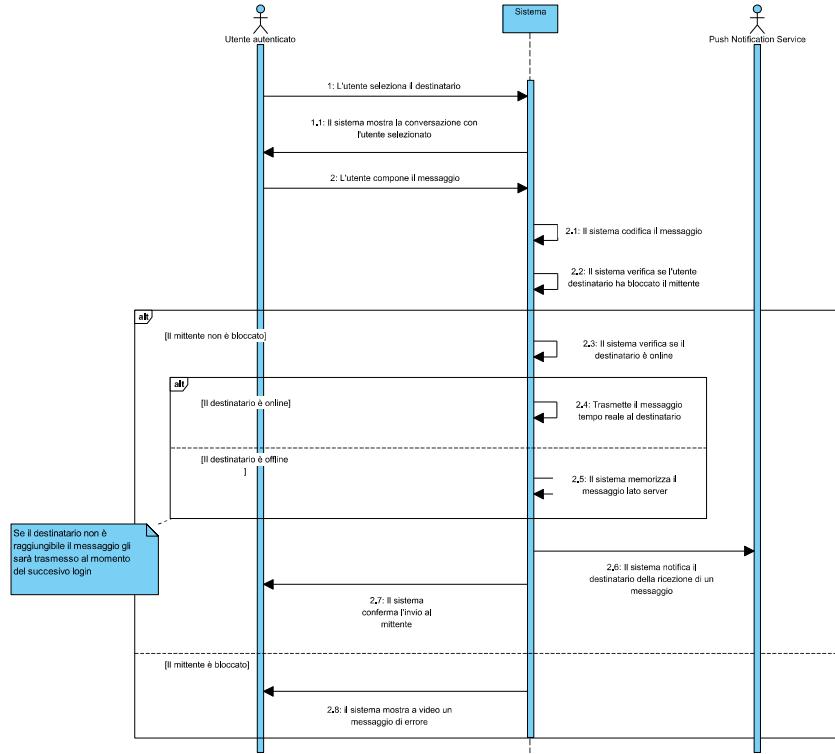


Figura 3.7: Invia Messaggio SSD

3.4.4 Modifica dati profilo

Per quanto riguarda il caso d'uso di modificare i dati del profilo utente, l'utente autenticato dalla Main Page dell'app clicca sul bottone "Profilo". Dalla schermata che appare, l'utente potrà caricare i nuovi dati. Dopo aver premuto il bottone "Conferma", il sistema aggiornerà i dati dell'utente, confermando la modifica con un messaggio a video. Nel caso vengano inseriti dei dati errati, come un file con estensione non valida, il sistema mostrerà un messaggio di errore

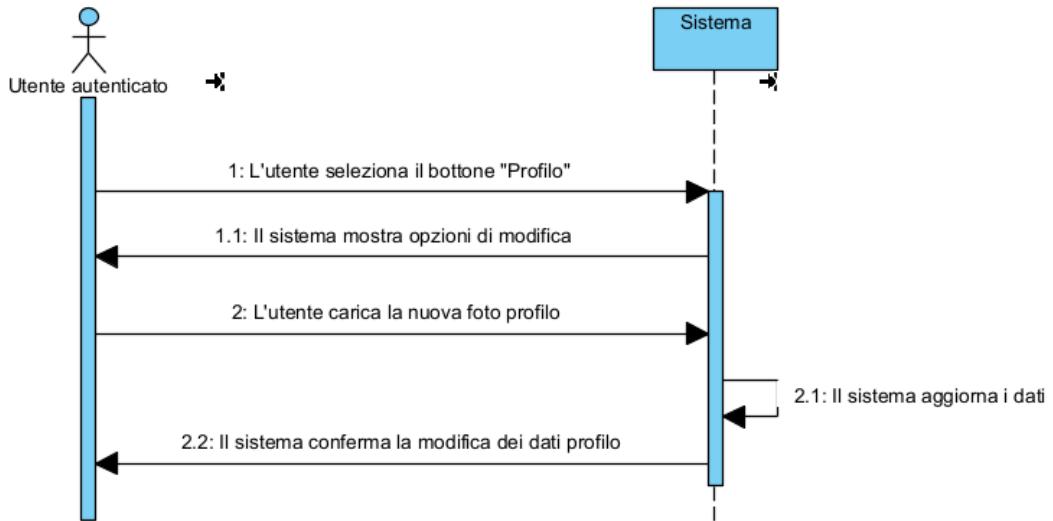


Figura 3.8: Modica immagine profilo SSD

3.4.5 Logout

Il caso d'uso di Logout è molto semplice: essendo che il tasto di Logout è presente solo nella Main Page, viene inserito un alternative fragment che fa tornare l'utente nella Main Page. Successivamente, l'utente, utilizzando il tasto di Logout, consentirà al server di chiudere la connessione col client e aggiornare i dati della connessione. Infine, all'utente verrà mostrata nuovamente la Login Page.

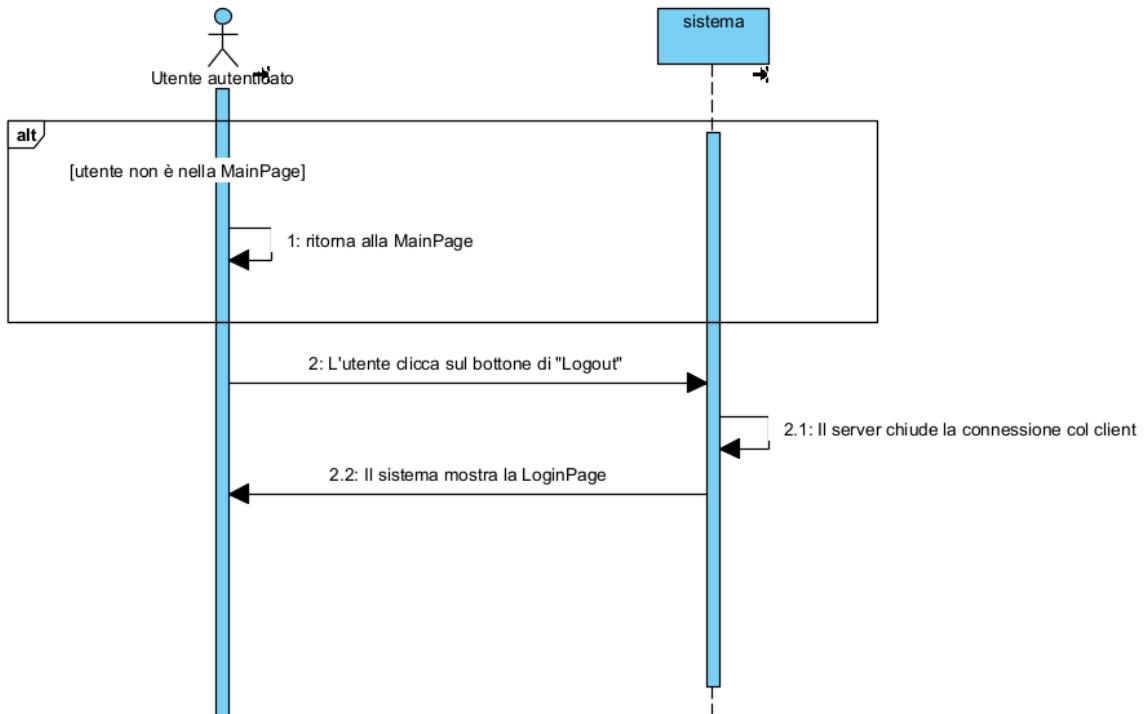


Figura 3.9: Logout SSD

3.4.6 Crea chat

Un utente autenticato può creare una chat, attraverso l'interfaccia grafica, utilizzando il bottone apposito e inserendo il nome utente del destinatario. Il sistema verificherà i dati del destinatario e ritornerà un messaggio di errore all'utente se già esiste una chat con quel destinatario o se il nome utente selezionato non è associato a nessun account. In tutti gli altri casi, la chat viene creata e il sistema mostrerà all'utente la nuova chat nella Main Page.

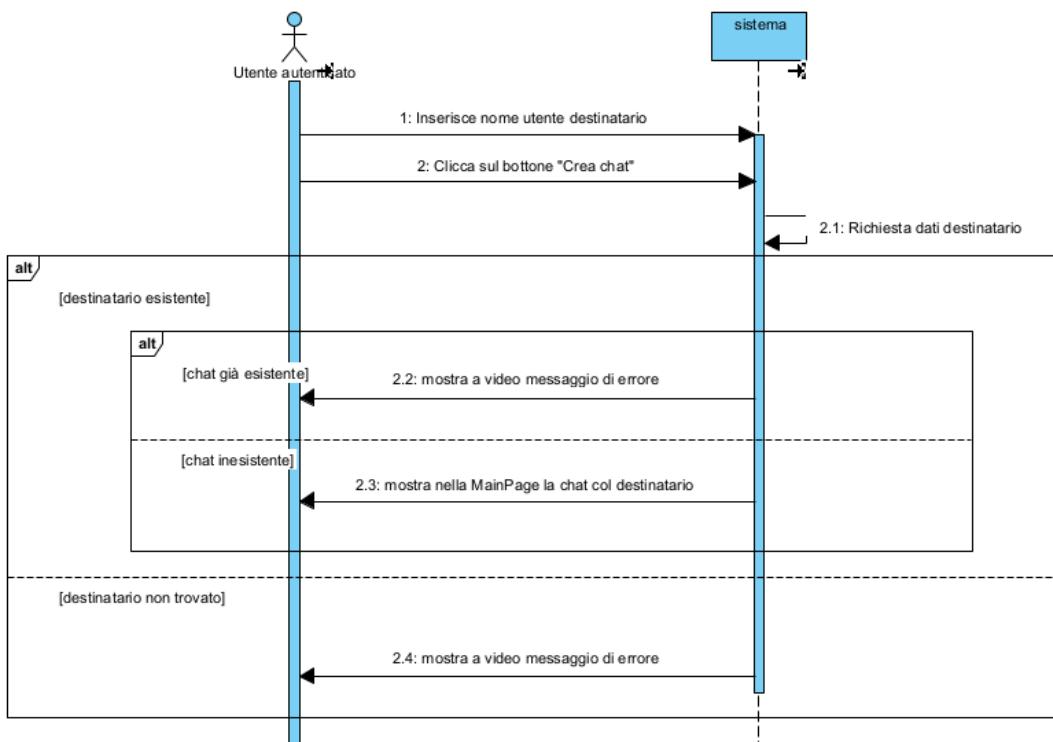


Figura 3.10: Crea chat SSD

3.4.7 Elimina chat

Per il caso d'uso di elimina chat, l'utente autenticato seleziona nella Main Page la chat da eliminare. Successivamente, nella Chat Page viene cliccato il bottone "Elimina chat", il sistema aggiorna i propri dati memorizzati localmente e mostra a video un messaggio di avvenuta eliminazione. Infine, l'utente, che ha eliminato la chat, viene reindirizzato alla Main Page.

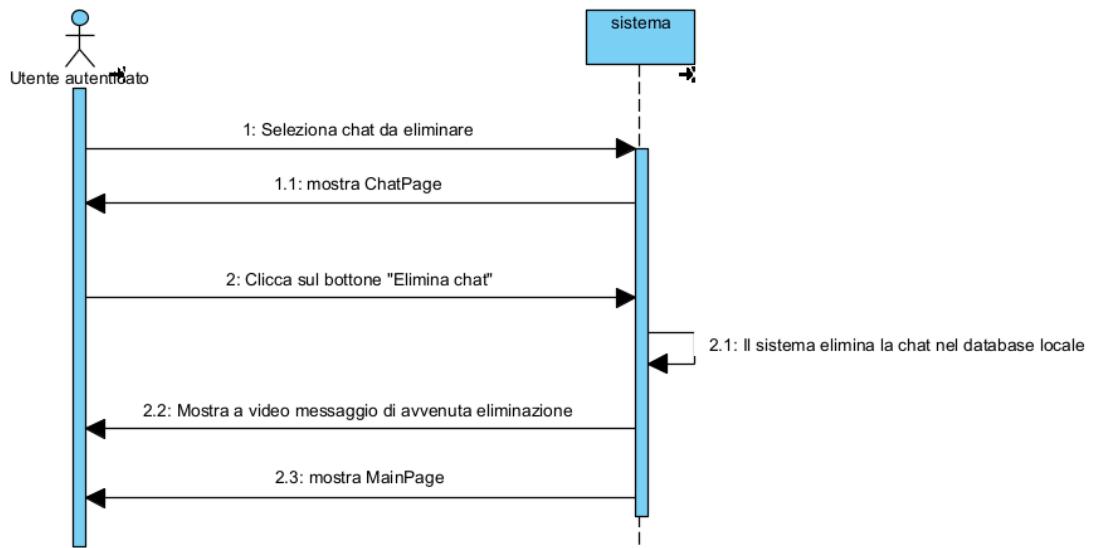


Figura 3.11: Elimina chat SSD

3.4.8 Blocca utente

Innanzitutto, l'utente seleziona la chat con l'utente da bloccare. In questa Chat Page, viene cliccato il bottone "Blocca Utente", il sistema aggiorna i dati e, se il blocco è avvenuto con successo, notifica l'utente con un messaggio di operazione avvenuta. Ciò implicherà non ricevere più messaggi da questo utente né visualizzare le informazioni del suo profilo. Nel caso il blocco fallisca, verrà ritornato un messaggio di errore.

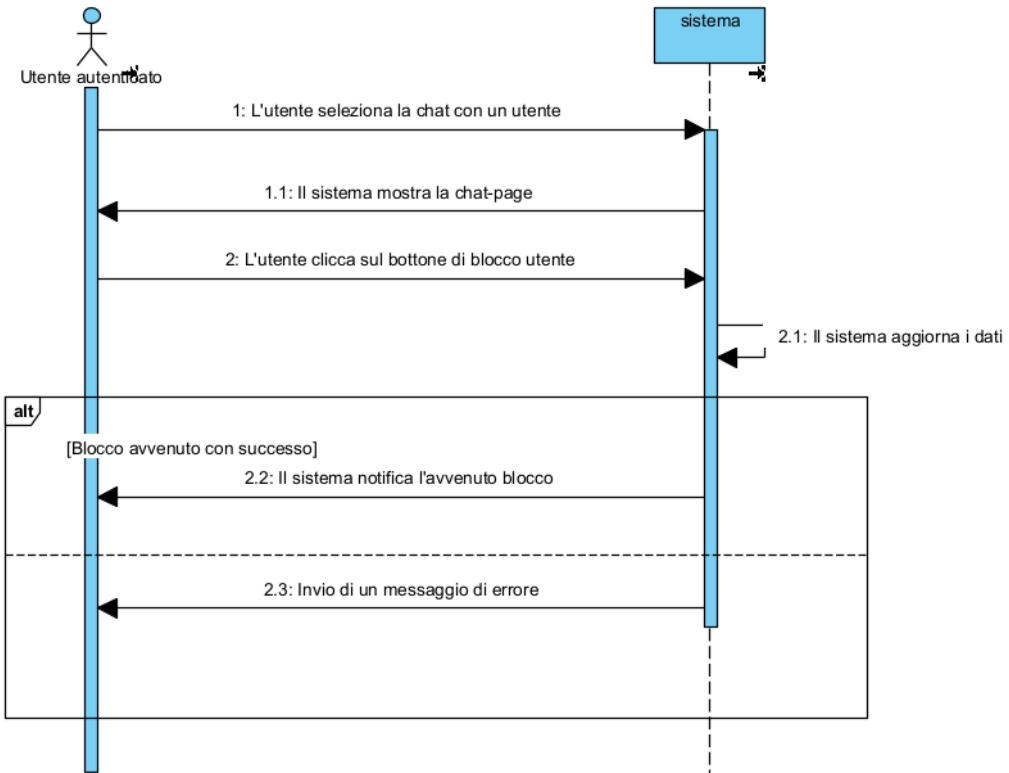


Figura 3.12: Blocca utente SSD

3.5 System Domain Model

Durante la fase di analisi del sistema è stato sviluppato anche il diagramma di dominio del sistema software. In particolare un modello di dominio è un **modello concettuale** che astrae e rappresenta gli aspetti (concetti, elementi o entità) principali pertinenti al dominio stesso (in tal caso il sistema software che si vuole realizzare), descrivendo le relazioni tra essi, oltre che i loro attributi e ruoli. Si è utilizzato dunque un UML Class Diagram per realizzare il modello di dominio (essendo il sistema object oriented questo tipo di notazione si presta più che bene) di alto livello del sistema software.

Sono state modellate dunque le seguenti entità principali componenti il dominio del sistema software:

- **Utente**: si tratta dell'utilizzatore del servizio di messaggistica. Esso è univocamente identificato da un *Id*, ed accede al servizio utilizzando la coppia *username-password* determinata in fase di registrazione. Dispone inoltre di una immagine associata al profilo (*immagineProfilo*) e di una coppia *chiave pubblica* e *chiave privata* per le funzioni di crittografia. Ogni utente chiaramente può essere mittente o destinatario di 0 o più messaggi, e può dunque essere in 0 o più conversazioni con altri utenti.

- **Elenco Utenti:** information expert degli utenti registrati al servizio, ne mantiene difatti una lista ed ha dunque la responsabilità di registrare un nuovo utente o di cercarne uno già registrato.
- **Conversazione:** ad ogni utente possono essere associate 0 o più conversazioni (ovvero chat con altri utenti), ognuna identificata da un campo *id*. Tale classe ha ,ad esempio, la responsabilità di aggiungere un nuovo messaggio alla conversazione, difatti mantiene una lista di 0 o più messaggi.
- **Messaggio:** classe che modella il singolo messaggio scambiato tra due utenti nell'ambito di una conversazione, identificato da un *timestamp* (ovvero l'istante, in termini di ore e minuti, dell'invio del messaggio stesso), un *tipo* (indicante se il messaggio è stato ricevuto o inviato) e il contenuto del messaggio *text*. Chiaramente ogni messaggio ha un solo destinatario ed un solo mittente. Inoltre ad ogni messaggio può essere eventualmente associato un contenuto multimediale (codificato in un determinato formato).

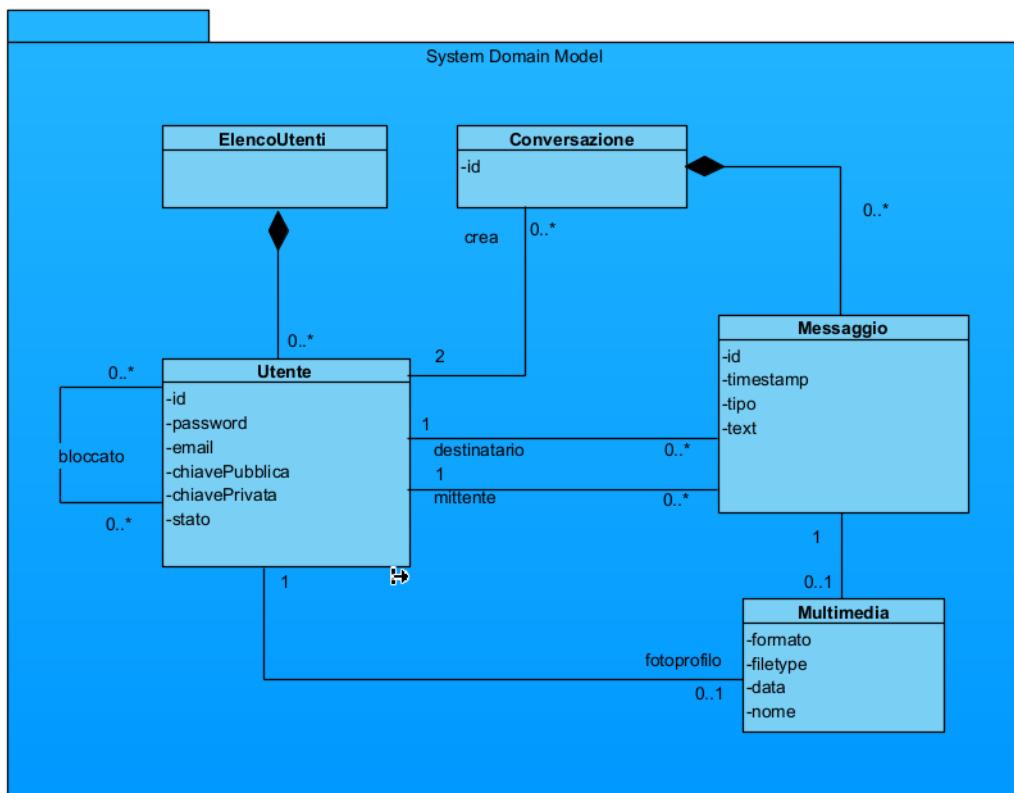


Figura 3.13: System Domain Model

3.6 State Machine Diagram degli stati utente

E' stato progettato uno SMD per modellare i diversi stati dell'utente nel momento in cui si registra e accede all'app.

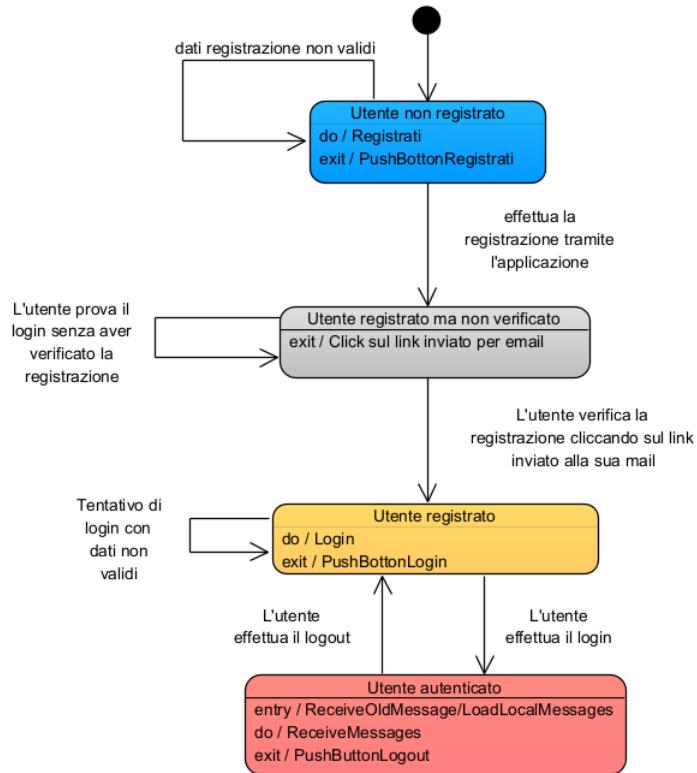


Figura 3.14: State Machine Diagram degli stati utente

Tale diagramma, chiaramente di alto livello, permette di capire tutti i vari passaggi che ci sono da quando l'utente apre l'applicazione nel Client la prima volta fino a quando è autenticato all'interno dell'app. Vengono mostrati quindi tutti gli stati, ovvero le condizioni in cui si può trovare l'utente in un determinato istante, e contemporaneamente tutti gli eventi che causano una transizione da uno stato all'altro.

Inizialmente, l'utente non è registrato e finché i dati non sono validi, ovvero non vengono rispettati i requisiti richiesti per ognuno dei campi della registrazione, rimane in tale stato. L'exit activity di tale stato è, quindi, il push sul bottone "Registrati".

Dopodichè, l'utente effettua la registrazione tramite l'applicazione, ma non è ancora verificato. Transiterà nello stato di utente registrato solo quando verificherà la registrazione cliccando sul link inviato alla mail. Da questo stato, che ha come exit activity il click sul link inviato tramite mail, si passerà allo stato di utente autenticato solo dopo aver effettuato correttamente il login e quindi aver inserito i dati validi.

Nello stato di utente autenticato, avremo come entry activities il ricevere i messaggi inviati quando l'utente era offline e il caricamento dei messaggi ricevuti precedentemente, che sono salvati in locale. Invece, la do activity sarà quella di ricevere nuovi messaggi nel caso arrivino e come exit activity il push sul bottone "Logout", che farà ritornare l'utente nello stato di utente registrato.

Capitolo 4

Architettura e Progettazione

La fase successiva riguarda l'architettura del software e la sua progettazione. Risulta dunque fondamentale, a questo punto, avviare l'individuazione di un'architettura complessiva che evidensi i componenti che costituiscono il sistema con i loro collegamenti e interazioni. Diversi dei requisiti presentati nel capitolo precedente sono dunque stati raffinati e dettagliati; tuttavia il processo di sviluppo impiegato non consente di avere una visione definitiva dell'architettura data la possibile variabilità dei requisiti stessi e delle possibili funzionalità richieste. Per questa ragione, alcuni dei diagrammi presentati costituiscono soltanto una versione non definitiva e potrebbero subire delle modifiche nella fase finale di progetto.

4.1 Client-Server Pattern

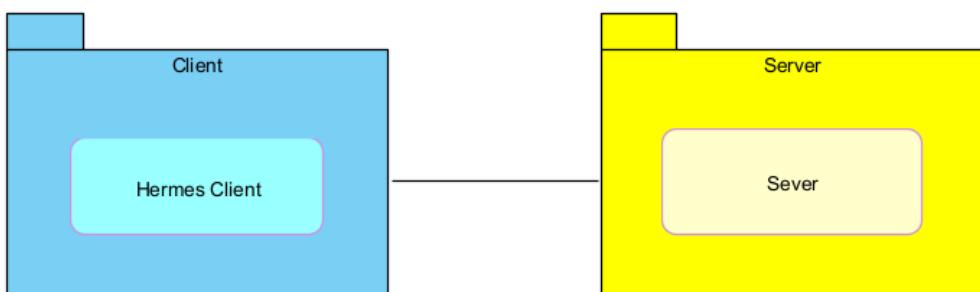


Figura 4.1: Client Server

Come già riferito precedentemente, l'applicazione Hermes è stata realizzata seguendo il pattern Client-Server. Secondo quest'ultimo, infatti, il Server è in grado di fornire ai vari Client collegati diverse funzionalità anche avvalendosi di servizi esterni, tramite un'interazione di tipo richiesta-risposta. Tale scelta architettonica è giustificata da numerosi vantaggi, di seguito elencati:

- Connessione tra Client e Server stabilita dinamicamente.
- Basso accoppiamento tra Server e i Client.
- Il numero di Client può scalare facilmente, anche se ciò dipende dalle capacità del Server, il quale può scalare a sua volta.
- Client e Server possono evolvere indipendentemente.
- L'interazione con l'utente è circoscritta al solo Client.

Ovviamente, si è consapevoli dei limiti che tale scelta comporta soprattutto in termini di possibili ritardi dei messaggi dovuti alla congestione/degradazione della rete.

NOTA: E' necessario specificare che, in questo caso, data la necessità per il Server di trasmettere messaggi verso il Client e dunque di creare una connessione verso quest'ultimo, il Client non rispetta in maniera esatta la sua definizione esponendo, come sarà specificato più avanti nella trattazione, una interfaccia usata dal Server stesso.

4.2 Hermes Client e Pattern MVC

Per la strutturazione dell'Hermes Client si è scelto come pattern di riferimento l'MVC (Model- View - Controller). Tale pattern si adatta perfettamente alle nostre esigenze con la possibilità di separare in maniera netta i componenti di presentazione dei dati da quelli che gestiscono i dati stessi.

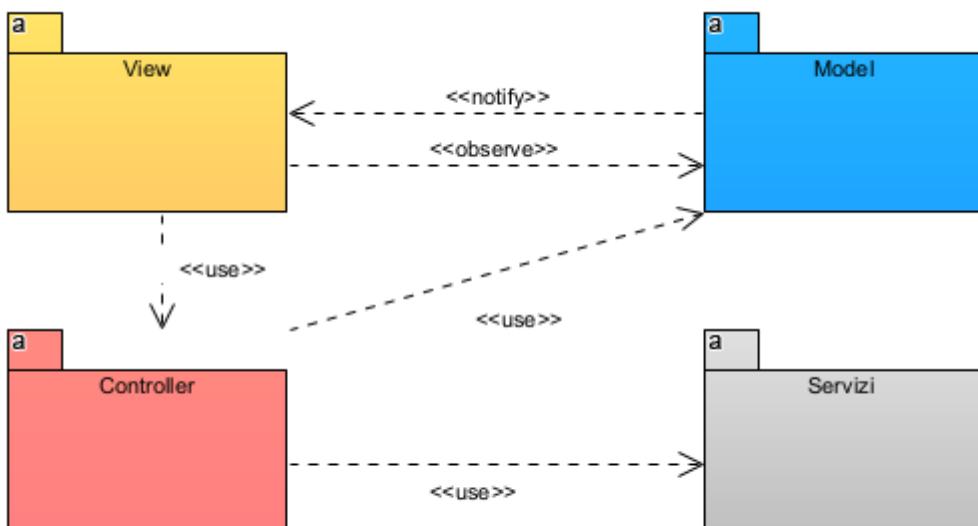


Figura 4.2: Client Package Diagram

I componenti principali di tale pattern sono:

- *Model*: contiene le classi le cui istanze rappresentano i dati da manipolare e visualizzare sulle varie page dell'applicazione. Questo non deve essere dipendente né dal livello View né da quello Controller, pur esponendo ai due le funzionalità per l'accesso e l'aggiornamento dei dati stessi. Inoltre, esso ha la responsabilità di notificare ai componenti della View eventuali aggiornamenti verificatisi in seguito a richieste del Controller per permettere alle View di presentare dati sempre aggiornati;
- *View*: gestisce la logica di presentazione (UI) dei dati contenuti nel Model. Essa interagisce con la strategia push coi dati del Model, implementando il pattern Observer;
- *Controller*: contiene gli oggetti che controllano e gestiscono l'interazione sia con il livello View che con il Model. Esso realizza la corrispondenza tra input utente e le azioni eseguite dal Model, selezionando anche le schermate della View da presentare.
- *Servizi*: contiene le classi di utilità come quella per la crittografia o per l'interfacciamento col database. Queste classi vengono utilizzate a supporto delle funzionalità del Controller.

4.2.1 Vantaggi del MVC

La scelta di tale Pattern Architetturale per il Client è giustificata dai seguenti vantaggi:

- Indipendenza dei vari componenti, che permette la suddivisione del lavoro.
- Esiste la possibilità di scrivere viste e controllori diversi utilizzando lo stesso modello di accesso ai dati e, quindi, riutilizzando parte del codice già scritto precedentemente.
- Supporto a nuovi tipi di Client con la semplice riscrittura di alcune View e alcuni Controller.
- Utilizzo di un modello rigido e di regole standard nella stesura del progetto, cosa che facilita un eventuale lavoro di manutenzione e agevola la comprensione da parte di altri programmatore.
- La possibilità di avere un controllore separato dal resto dell'applicazione, rende la progettazione più semplice e permette di concentrare gli sforzi sulla logica del funzionamento.
- Interattività, fortemente richiesta dalla nostra applicazione.

4.3 Server e Microservizi

Per quanto riguarda il lato Server ci si è orientati verso un'architettura a **microservizi**.

Tale scelta architetturale è stata realizzata al fine di consentire una più corretta gestione dei picchi di traffico. La decomposizione funzionale in *microservizi*, i quali gestiscono una sola responsabilità, consente di raggiungere i seguenti obiettivi:

- Agilità del team
- Aumento della scalabilità del sistema, creando più istanze di uno stesso servizio al crescere della domanda
- Tolleranza agli errori con unità isolate che evitano la propagazione di eventuali errori
- Modularità, infatti ciascun microservizio è caratterizzato da un'API che costituisce un confine impermeabile

Si riporta, di seguito, la descrizione ad alto livello della struttura dell'architettura del sistema:

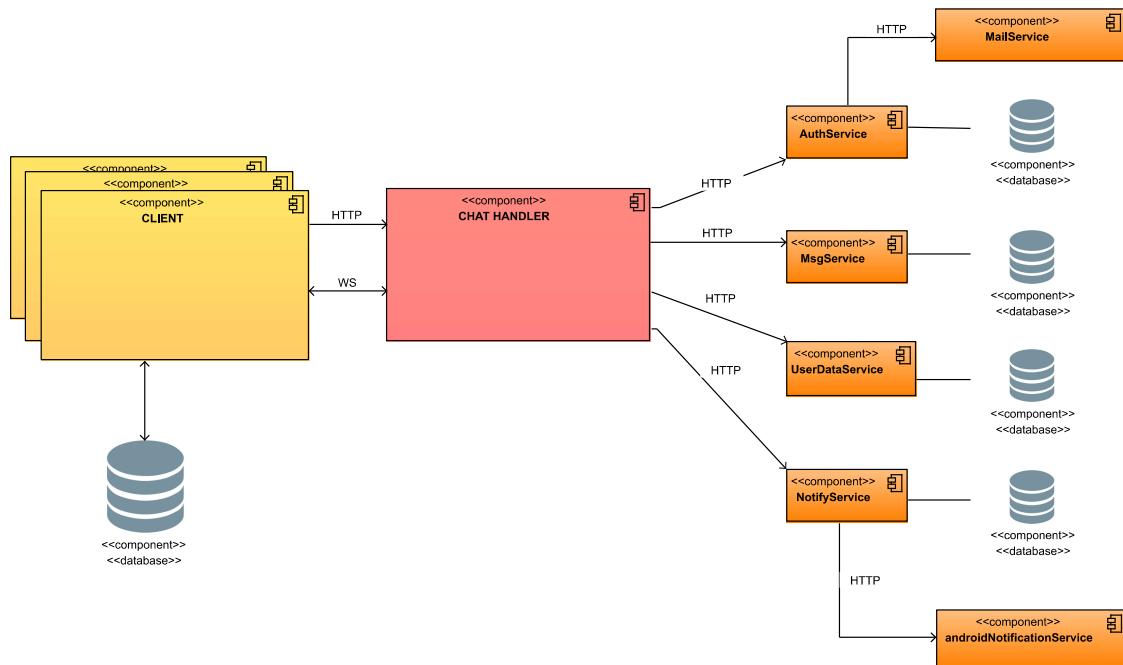


Figura 4.3: Diagramma architetturale di alto livello

Nel nostro caso, i *microservizi* sono stati sviluppati ex-novo e strutturati a livelli. Ciascun livello costituisce una "Macchina Virtuale" che fornisce un set coeso di servizi tramite un'opportuna interfaccia. In particolare, è noto che, in una *closed layered architecture*, ciascun livello è abilitato a utilizzare solo i livelli sottostanti. Ciò consente di:

- Ridurre l'accoppiamento tra le parti
- Avere livelli di astrazione crescenti
- I cambiamenti di un layer possono influire solo su quelli superiori

- Modificabilità, a patto che l’interfaccia non cambi
- Interfacce standard per librerie e framework

4.4 Vista Componenti e Connettori

Nella definizione dell’architettura complessiva, è opportuno valutare la strutturazione del sistema nei vari componenti e le modalità di interazione per la realizzazione delle funzionalità richieste, tenendo sempre in conto anche i requisiti non funzionali specificati. A tal scopo può essere utile rappresentare il sistema tramite una vista *Componenti e Connettori*.

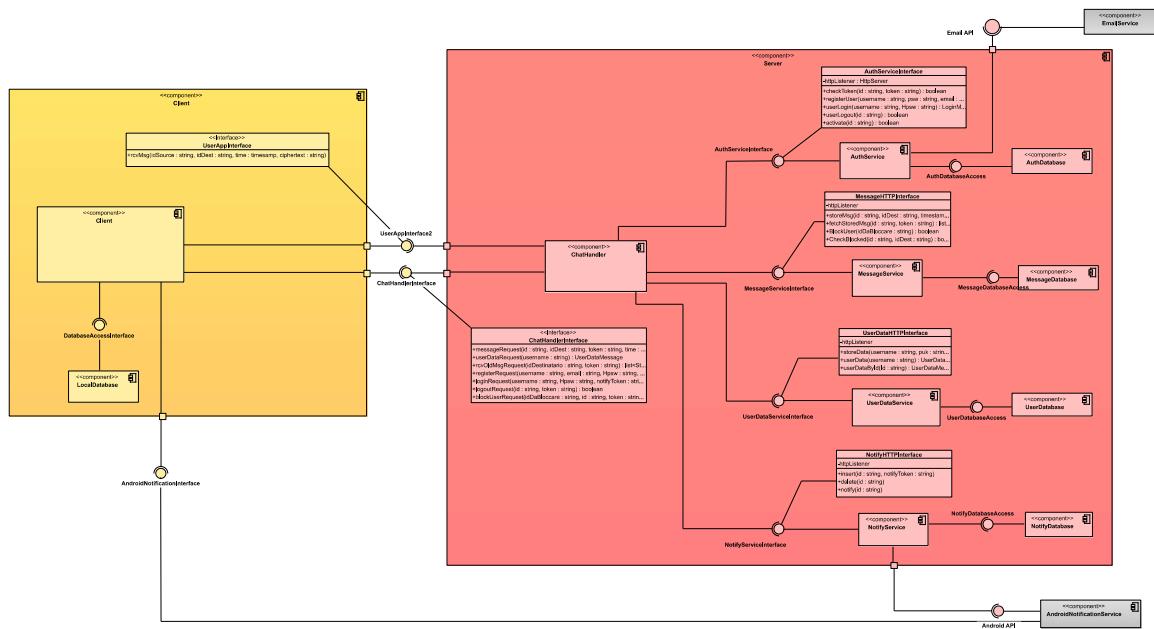


Figura 4.4: Vista Componenti e Connettori

Nel caso specifico, i **componenti** rappresentati sono:

- *Client*: utilizzatore del servizio di messaggistica con al suo interno la presenza di un *Local Storage* per l’immagazzinamento dei dati necessari al caricamento delle chat ed ulteriori informazioni associate allo specifico client
- *Chat Handler*: server di gestione delle operazioni effettuabili sulle chat. Esso ha il compito di mappare le richieste provenienti dal client in opportune sequenze di chiamate necessarie a implementare le funzioni richieste; si occupa, inoltre, di garantire la correttezza del formato della richiesta e l’autenticazione del richiedente.
- *AuthService*: componente per realizzare il servizio di autenticazione e garantire i requisiti di sicurezza specificati.

- *UserDataService*: componente che ha il compito di gestire, richiedendo e memorizzando, le informazioni relative a tutti gli utenti utilizzatori del software
- *Message Service*: componente che svolge la funzione di gestione dei messaggi.
- *Email Service*: servizio esterno per l'invio della mail di conferma della registrazione
- *NotifyService*: servizio che consente la ricezione di notifiche push, si interfaccia a sua volta con l'*AndridNotificationService* per realizzare questa funzionalità.

Per quanto riguarda i **connettori** invece:

- L'interazione *Client-ChatHandler* è stata specificata tramite la notazione a lollipop. In particolare è prevista un'interfaccia HTTP per l'invio delle richieste dal Client verso il Server e una ulteriore interfaccia che sfrutta le Web Socket la quale garantisce un canale di comunicazione bidirezionale tra le due parti.
- Anche l'interazione tra Il *ChatHandler* e i *Microservizi* indicati sfrutta una interfaccia HTTP per l'invio di richieste e la ricezione di risposte.
- Tutti i componenti dotati di un database prevedono, inoltre, la presenza di una ulteriore interfaccia con funzioni specifiche per l'implementazione delle classiche operazioni C.R.U.D.

La principale limitazione di questa architettura è legata alla replicabilità del componente Chat Handler, componente che fa da gateway tra lato client e lato microservizi, in quanto esso mantiene una connessione con gli utenti attivi, quindi è necessario gestire uno stato condiviso tra più componenti Chat Handler. Si è scelto, quindi, di progettare un'architettura con un unico Chat Handler che ci ha permesso di evitare qualsiasi tipo di problema per la comunicazione tra più componenti.

4.5 Dinamica dei componenti

Al fine di chiarificare ulteriormente le funzionalità richieste al nostro software, si è scelto di realizzare ulteriori diagrammi con un livello medio di astrazione, detti di progettazione. Tali diagrammi sfruttano i componenti già individuati tramite la vista Componenti e Connitori 4.4 e mostrano le loro interazioni.

4.5.1 Crea chat

Sequence Diagram di progettazione

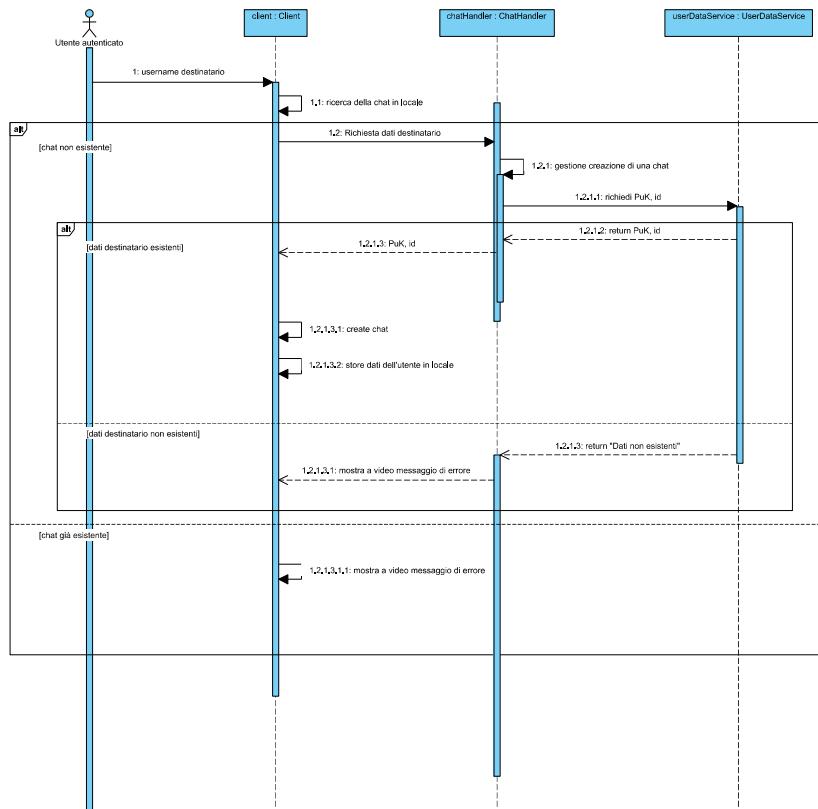


Figura 4.5: CreaChatProgettazione

Dal diagramma in figura si vede come l'interazione parta dall'Utente autenticato, il quale inserisce lo username del destinatario. Qualora la chat non esista in locale occorre prelevare i dati dell'Utente ricevente necessari alla trasmissione del messaggio. Se il destinatario non corrisponde ad un utente registrato viene ritornato un messaggio di errore, in caso contrario viene creata la chat e memorizzate in locale le informazioni dell'utente a cui si trasmette il messaggio.

Activity Diagram

Per specificare meglio questo caso d'uso, abbiamo aggiunto una seconda vista dinamica che ne descriva meglio il comportamento. Tale Activity Diagram specifica la sequenza di azioni richieste per la creazione di una chat. Si distingue, innanzitutto, tra i casi in cui la chat sia già presente per cui viene ritornato un messaggio di errore e il caso in cui la chat non esista ancora, in cui è necessario fetchare i dati dell'utente destinatario specificato per la creazione. Se durante tale processo viene riscontrato un problema il sistema mostrerà un opportuno messaggio di errore a video.

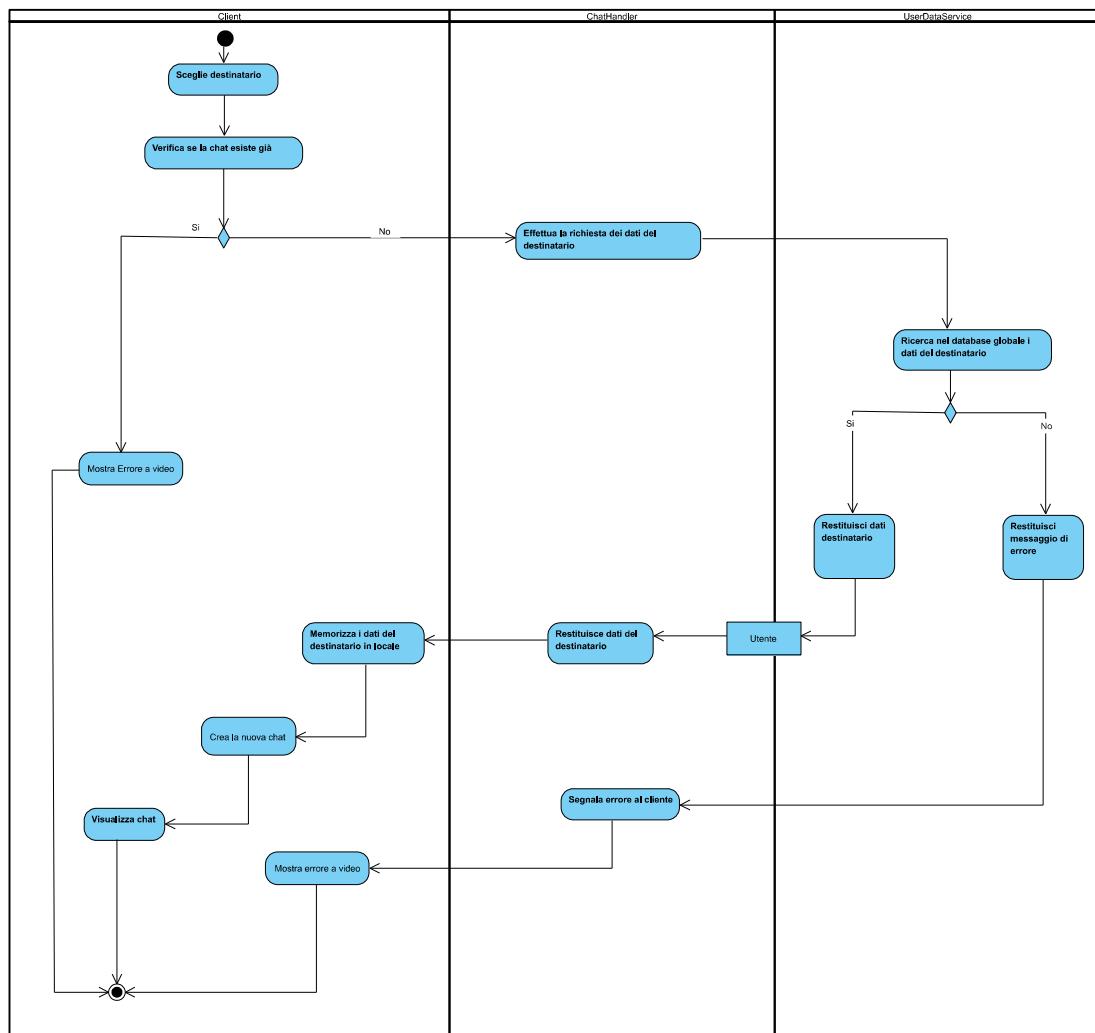


Figura 4.6: Creazione di una Chat AD

4.5.2 Effettua login

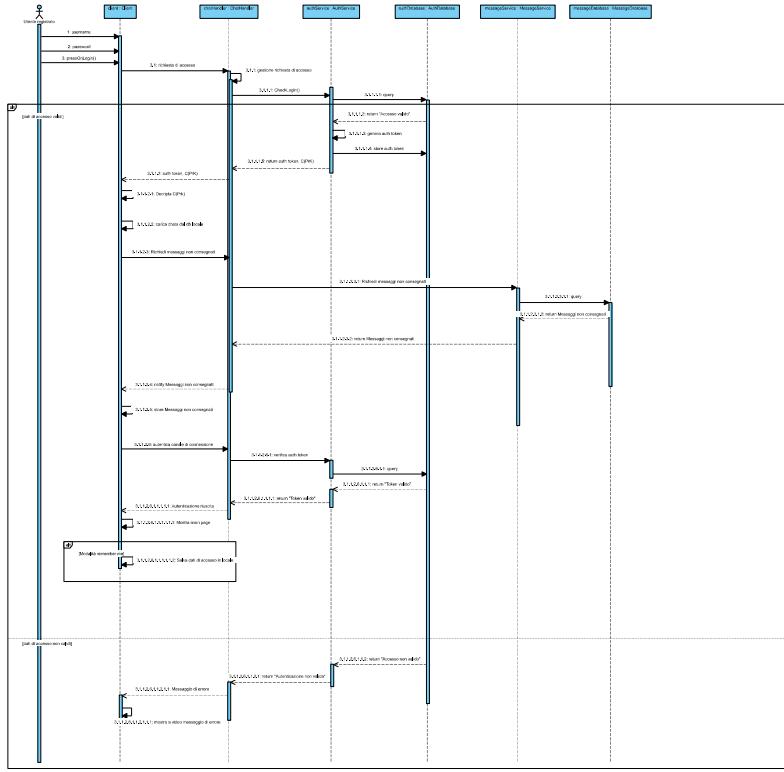


Figura 4.7: EffettuaLoginProgettazione

Per il login, l'Utente registrato inserisce le proprie credenziali. Cliccando su un apposito bottone, la richiesta viene indirizzata al ChatHandler il quale interroga l'AuthService per la verifica dei dati inseriti. Se i dati di accesso sono validi, questo ha inoltre il compito di memorizzare e generare un Token da assegnare all'utente. L'AuthService trasmette, oltre al Token, anche la chiave privata cifrata dell'utente necessaria per decifrare eventuali messaggi. A questo punto vengono caricate la chats precedentemente memorizzate in locale ed effettuata una richiesta al MessageService dei messaggi eventualmente trasmessi mentre si era offline. Non resta che autenticare il canale di connessione (web-socket) con il proprio Token e salvare in locale le credenziali di accesso se la checkbox "Remember Me" è stata spuntata. Quest'ultima funzionalità ci consente di effettuare il login automatico, permettendo all'utente di essere reindirizzato direttamente alla MainPage senza passare per la LoginPage all'avvio dell'app.

4.5.3 Effettua registrazione

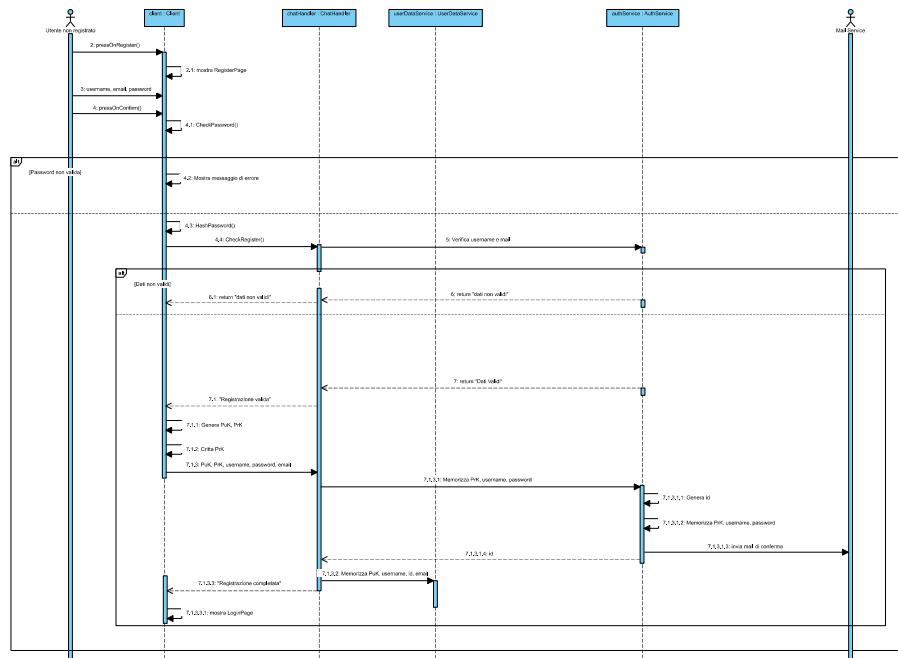


Figura 4.8: EffettuaRegistrazione Progettazione

Per il caso d'uso della registrazione, un utente non registrato preme sul bottone "Registrati" affinchè il componente Client possa mostrargli la RegisterPage. Da lì, l'utente potrà inserire username, mail e password e premere sul bottone "Conferma" per avviare il processo di registrazione.

Innanzitutto, il Client controlla se la password rispetta i requisiti richiesti, ovvero che deve contenere almeno 8 caratteri, di cui almeno una lettera e un numero. Se la password non è valida viene mostrato un messaggio di errore dal client. Altrimenti, il chatHandler verificherà la registrazione delegando il compito all'UserService, attraverso una query sull'UserDatabase. La query controllerà username e mail, nel caso uno di questi già sia stato utilizzato verrà ritornato al client un messaggio di errore, altrimenti un messaggio di registrazione valida.

A questo punto, il client genera chiave privata e chiave pubblica ed esegue il processo di crittografia della chiave privata attraverso la propria password. Dopo aver fatto ciò, invia le due chiavi al chatHandler, che gestisce la memorizzazione della chiave privata, mediante il servizio di AuthService. Questo in primis memorizza la chiave privata su AuthDatabase e poi dà l'ok all'invio della mail di conferma tramite il servizio esterno di Mail Service.

Successivamente, vengono memorizzate anche le informazioni pubbliche dell'utente appena registrato, tramite il servizio di userService, e, infine, viene ritornato al client un messaggio di registrazione completata, che porterà l'utente dalla RegisterPage alla LoginPage.

4.5.4 Invia messaggio

Sequence Diagram di progettazione

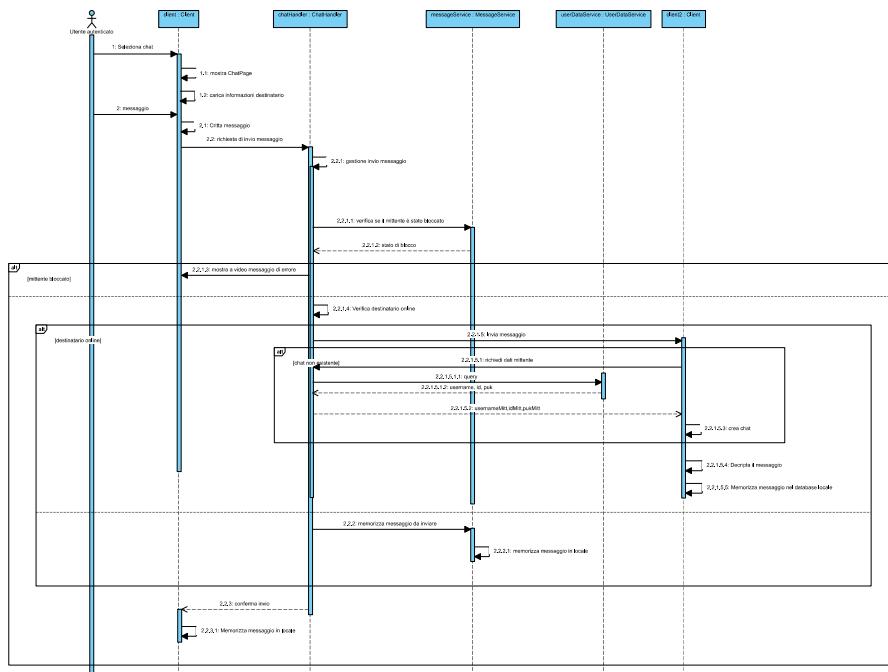


Figura 4.9: InviaMessaggio

Invia messaggio è uno dei casi d’uso più importanti della applicazione. Un utente autenticato seleziona una chat con un utente a cui vuole inviare un messaggio, il Client quindi mostra la chatPage corrispondente e carica le informazioni dell’utente ricevente, come username e messaggi precedenti. Quindi l’utente scrive un messaggio e, dopo averlo inviato, il Client lo cripta.

Il messaggio criptato viene quindi mandato al chatHandler, che gestirà l’invio del messaggio al secondo client. Innanzitutto, bisogna verificare se il mittente è stato bloccato: questo controllo avviene tramite il MessageService, che ritornerà al chatHandler lo stato di blocco. Se il mittente è stato bloccato, sarà mostrato un messaggio di errore, altrimenti si passa al secondo controllo, in cui si verifica se il destinatario è online.

Se è connesso, il chatHandler invia il messaggio al client ricevente. Se il messaggio ricevuto dal client è il primo e quindi la chat non esisteva prima, allora esso richiede i dati del mittente al chatHandler tramite l’id del mittente contenuto all’interno del messaggio ricevuto. Sarà poi il chatHandler a fare una query sull’UserDataService per ottenere i dati richiesti e a rimandarli al client, che così finalmente potrà creare la chat con il mittente del messaggio. In ogni caso, sia se la chat è "nuova" sia se già esistente, il destinatario decripta il messaggio e memorizza tale messaggio decriptato nel suo database locale.

Se il destinatario non è connesso, il chatHandler memorizza il messaggio inviato utilizzando il micro-servizio MessageService, che salverà nel suo database locale. I messaggi memorizzati saranno recapitati

al destinatario al suo successivo login al servizio.

Infine, il chatHandler invierà al client una conferma dell'invio del messaggio, sotto forma di risposta alla richiesta HTTP inviata inizialmente dal Client.

Activity diagram

Per il caso d'uso *Invia Messaggio* è stato realizzato un ulteriore diagramma per modellarne il processo dinamico. In particolare, viene inserita una ulteriore istanza di Client per la modellazione dell'intero processo di invio e ricezione del messaggio, oltre al ChatHandler e al MessageService. Tale diagramma ci consente di evidenziare come la memorizzazione del messaggio nel microservizio MessageService avvenga solo quando il destinatario è offline in modo da garantire la proprietà di persistenza dei messaggi.

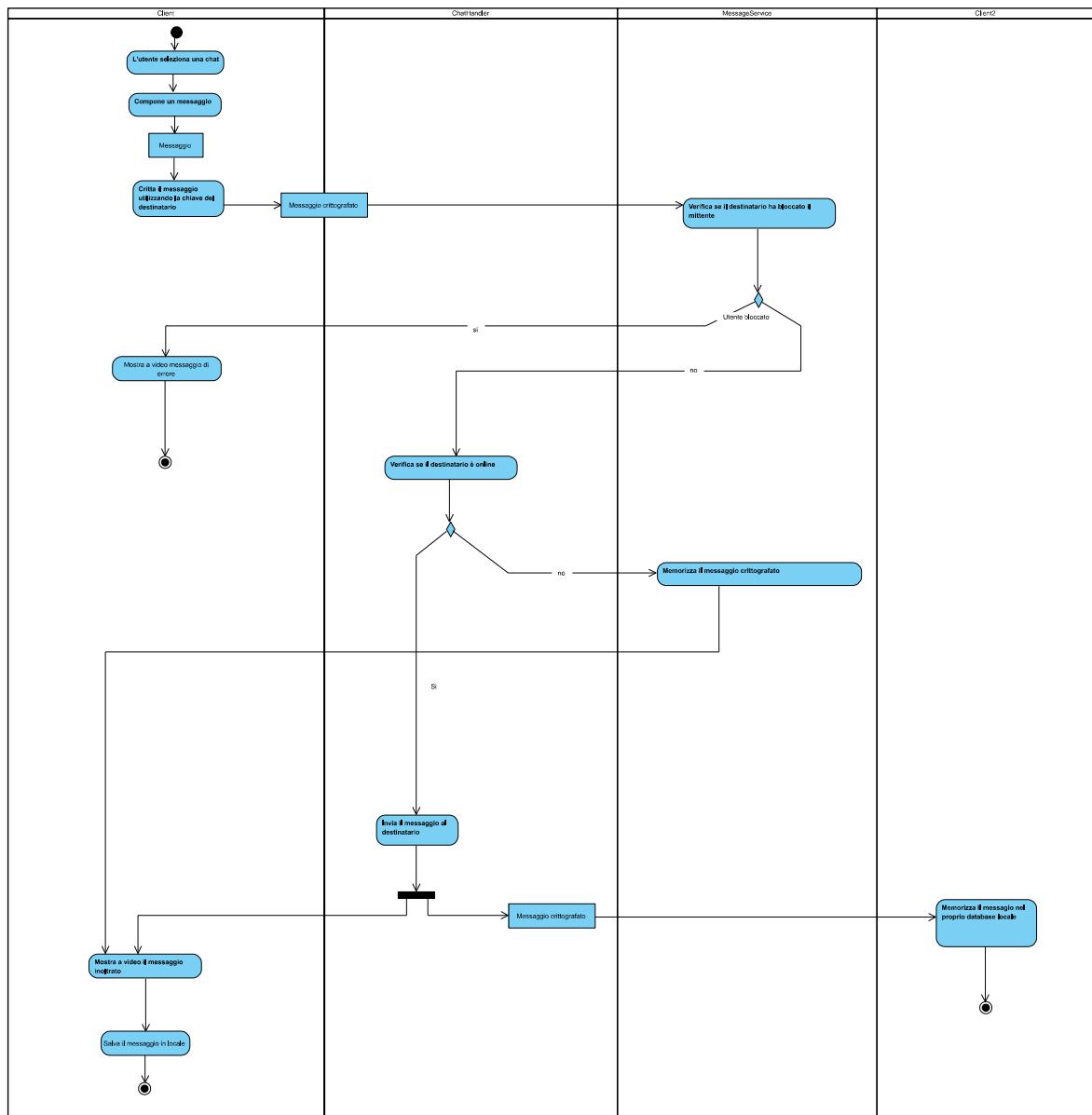


Figura 4.10: Invia Messaggio AD

Communication Diagram

Il Communication Diagram è un diagramma che è stato utilizzato nel nostro progetto per mostrare gli scambi di messaggi e i protocolli tra i vari componenti dell'architettura. Essendo un diagramma di alto livello, permette di ottenere una rappresentazione generale di come i vari componenti interagiscono tra di loro in un determinato caso d'uso, in questo caso quello dell'invio di un messaggio. Prima di descrivere il diagramma bisogna fare delle premesse per questo caso d'uso:

- entrambi gli utenti sono online.
- l'utente mittente possiede già i dati del destinatario.
- il destinatario non conosce i dati del mittente, ovvero il mittente è il primo ad inviare il messaggio nella conversazione (che dunque non esiste ancora lato destinatario).

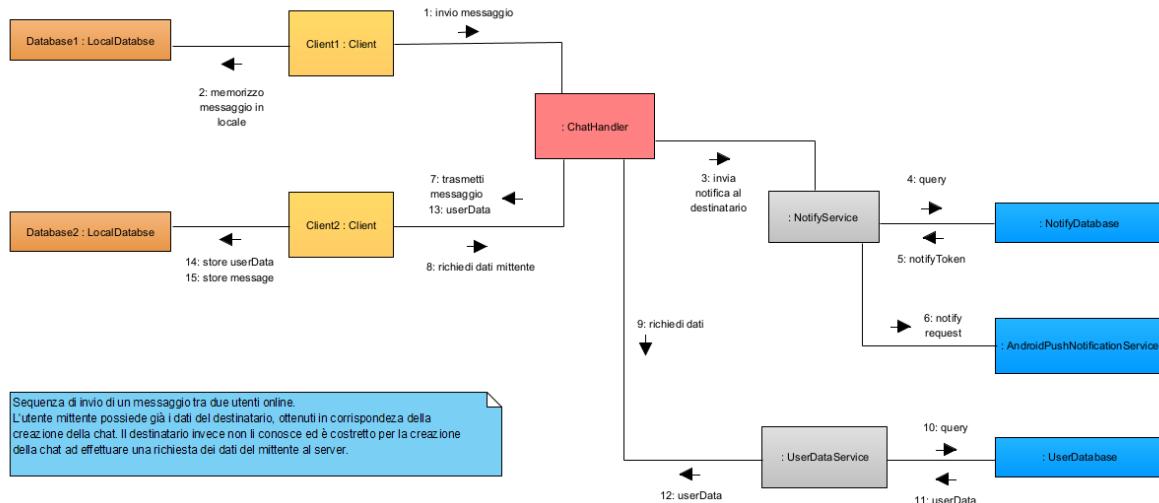


Figura 4.11: Communication Diagram invio messaggio

Come si vede in figura, il client mittente, raffigurato nella figura come Client1, invia il messaggio al ChatHandler che ha il compito di gestire gli scambi di messaggi tra i vari client e l'interfacciamento con i microservizi. Viene quindi inviata la notifica del messaggio al destinatario mediante il microservizio di NotifyService, che farà una query sul NotifyDatabase per ottenere il notifyToken e sfrutterà il servizio esterno di AndroidPushNotificationService per fare la richiesta di notifica. Successivamente, il chatHandler trasmette il messaggio al Client2 e, visto che il destinatario non conosce i dati del mittente, esso per la creazione della chat effettua una richiesta dei dati del mittente al ChatHandler. Viene quindi soddisfatta la richiesta dei dati del mittente, effettuata dal Client2, che viene inoltrata al microservizio UserDataService, incaricato proprio di memorizzare i dati identificativi di Client1. Infine, dopo aver ottenuto i dati dal ChatHandler, il destinatario memorizza sul suo database locale sia i dati dell'utente mittente che il messaggio inviato.

4.6 Client Class Diagram

Il client, come già descritto, è stato progettato attraverso il pattern architetturale MVC. Il diagramma ci mostra come, oltre ai 3 soliti package, ci sia anche un quarto package, chiamato "Servizi", che contiene delle classi di utilità relative all'interconnessione col server e col database locale e che offre i vari servizi di crittografia (per la password dell'utente e per i vari messaggi). In particolare, questo package verrà utilizzato dal Controller, come è possibile notare anche dalla dipendenza d'uso.

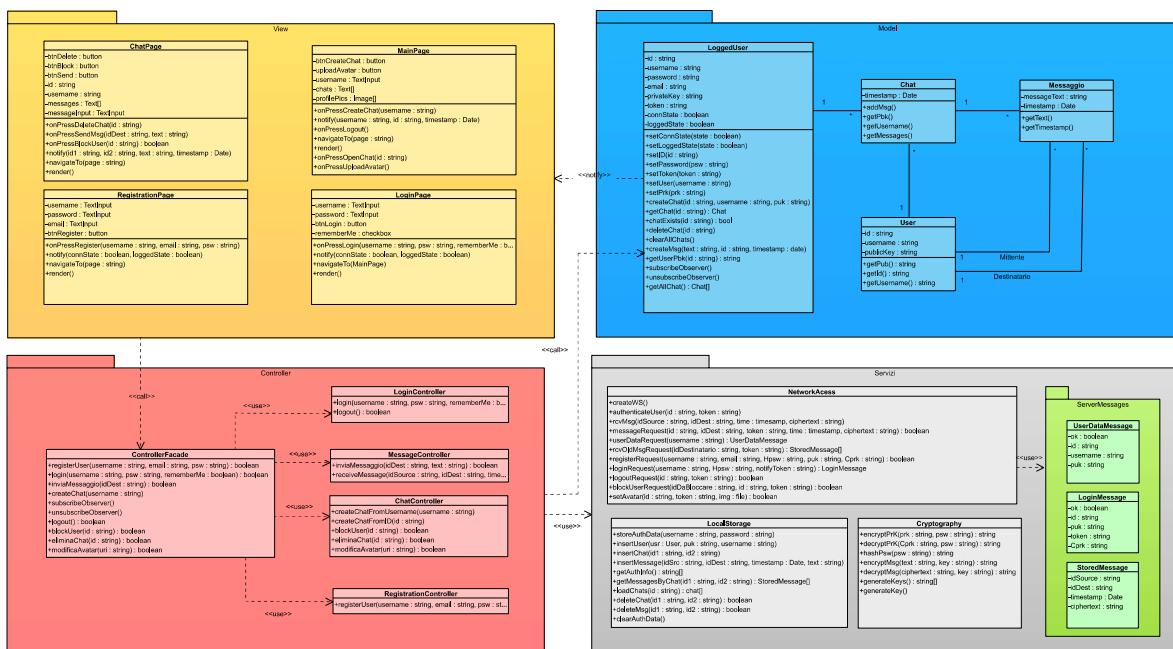


Figura 4.12: ClientClassDiagram

Di seguito, andremo ad analizzare nel dettaglio ognuno dei package del diagramma, descrivendo com'è stato progettato e il suo legame con gli altri package.

4.6.1 View

Nella View sono presenti le diverse pagine visualizzabili dall'utente mentre naviga nell'applicazione. Esse saranno caratterizzate da diversi elementi grafici, come button e text input, con i corrispettivi handler. Si può notare come tutte le classi abbiano un metodo `notify()`, che viene invocato da `LoggedUser` e che modifica l'aspetto grafico del componente in relazione ad una modifica dei dati del Model. In particolar modo, le notify della `RegistrationPage` e della `LoginPage` segnalano l'effettiva connessione al server o lo stato di login compiuto. Invece, la notify della `ChatPage` notifica la ricezione di un nuovo messaggio, mentre quella della `MainPage` la creazione di una nuova chat. Ad ogni bottone è poi associato l'handler per la gestione del click sul bottone stesso.

- RegistrationPage presenta tre text input per i tre campi richiesti per la registrazione, email, username e password, e un bottone per confermare la registrazione.
- LoginPage ha due text input per username e password e un bottone per confermare il login.
- MainPage è la pagina principale dell'app, presenta un text input in cui inserire il nome utente per creare una chat e il bottone corrispondente per confermare. Inoltre, presenta un bottone per il logout e per modificare le informazioni del profilo. La MainPage ha inoltre un vettore di Text e di immagini, associati ad ognuna delle chat presenti nel Model. L'URL utilizzato per caricare la risorsa remota associata all'immagine è calcolato in base all'ID dell'utente con cui è stata creata la chat.
- ChatPage rappresenta la pagina della chat con un altro utente, quindi contiene un bottone per eliminare la chat, uno per bloccare l'utente destinatario ed un terzo per inviare il messaggio. Chiaramente ha anche un text input dove poter scrivere il messaggio da inviare. E' presente un vettore di Text corrispondente all'insieme dei messaggi associati alla chat aperta.

4.6.2 Controller

Il controller viene chiamato dalle classi della View. Per semplificare queste chiamate, e ridurre l'accoppiamento, abbiamo implementato un *facade* per il package Controller che ha tutti i metodi delle classi del Controller, esponendoli come interfacce. Il pattern facade nella programmazione ad oggetti indica una classe che raccoglie le interfacce di diversi sottosistemi in modo da diminuire l'accoppiamento tra i package e, inoltre, permette di istanziare in maniera ben definita le classi del package Controller. Nel nostro caso, il ControllerFacade possiede un riferimento alla classe LoggedUser del Model, quindi sono stati implementati due metodi, `SubscribeObserver()` e `UnsubscribeObserver()`, che implementano il pattern Observer tra View e Model. Infatti, le view chiamano questi metodi per iscriversi a LoggedUser ed essere notificati in caso di cambiamenti che avvengono sul Model.

In aggiunta al facade, le classi del Controller sono:

- LoginController, che si occupa delle operazioni di Login e Logout, con l'aggiunta dell'opzione di rememberMe, qui modellata attraverso un parametro di input di tipo boolean
- MessageController, dedito all'invio e alla ricezione dei messaggi
- ChatController, per la gestione di una chat, sia per quanto riguarda la sua creazione che per quanto riguarda il blocco dell'utente destinatario. In particolare si noti come sia possibile creare la chat dato l'username del destinatario ma anche dato l'id. Questo per garantire che la chat venga creata anche quando un messaggio viene ricevuto, in quanto in questo caso il destinatario conosce solo

l'id del mittente. Inoltre tale controller si occupa anche di gestire l'eliminazione di una chat o la modifica dell'Avatar associato all'utente.

- RegistrationController, che si occupa del caso d'uso di registrazione

4.6.3 Model

Il package Model, come già detto in precedenza, ha al suo interno le entità del dominio da rappresentare. In base ai suoi cambiamenti, ha il compito di trasmettere delle notifiche alla View, mostrando quindi all'utente che sta utilizzando l'applicazione tali modifiche. Come vediamo dalla figura 4.12, il package è caratterizzato dalla classe LoggedUser, implementata attraverso il pattern Singleton. Nella programmazione ad oggetti, il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza. La classe LoggedUser quindi rappresenta esattamente l'unico utente attualmente connesso e loggato in uno specifico client dell'applicazione. Ha numerose variabili membro: id, che è una stringa identificativa, username, password, email, token, una stringa identificativa che viene assegnata quando si effettua il login nell'applicazione, connState e loggedState, che rappresentano lo stato attuale dell'account e riguardano rispettivamente lo stato di connessione all'app e lo stato di login. Si noti come a prima vista id e token possano sembrare identici: in realtà la differenza consiste nel fatto che il token è valido fino al logout e ne è a conoscenza solo l'utente autenticato, mentre l'id rimarrà per sempre e costituisce una informazione pubblica. Inoltre, la variabile membro privateKey è una variabile di tipo string, che viene utilizzata per decriptare i messaggi in arrivo da altri utenti. Per quanto riguarda i metodi della classe, oltre alle varie get e set delle variabili membro, abbiamo: *createChat*, metodo utilizzato per istanziare sia la chat sia, eventualmente, l'utente (con il puk dato in ingresso); *getChat*, che implementa una ricerca delle chat e permette di ritornare quella con l'id posto in ingresso; *chatExists* verifica se una chat è già esistente con l'id dell'utente in input alla funzione; *clearAllChats* elimina tutte le chat collegate all'utente e verrà utilizzata quando l'utente effettuerà il logout; *createMsg* permette di istanziare un messaggio col suo text e il timestamp di invio; *getUserPbk* presenta in output la chiave pubblica dell'utente con cui si sta messaggiando, utilizzata per criptare i messaggi da inviare; ancora, *subscribeObserver* e *unsubscribeObserver* rappresentano metodi chiamati dal controller allo scopo di "iscrivere" la View ai cambiamenti effettuati sul Model e quindi sull'entità del dominio. Infine, i due metodi *deleteChat* e *getAllChat* utilizzati uno per eliminare una chat e l'altro per prelevare tutte quelle associate all'utente. Da notare come il LoggedUser sia il Builder rispetto alle classi di Chat e User. Nella programmazione ad oggetti, il design pattern Builder, separa la costruzione di un oggetto complesso dalla sua rappresentazione cosicché il processo di costruzione stesso possa creare diverse rappresentazioni. Oltre alla classe Singleton, che svolge le funzioni di Facade per il package Model, abbiamo le seguenti classi, che rappresentano i dati del nostro sistema e che sono state ottenute dal System Domain Model, descritto in figura 3.13:

- **Chat** che possiede un riferimento a LoggedUser ed uno alla classe User, ha anche come variabile membro un vettore di elementi di tipo Messaggio e ha i metodi per aggiungere un messaggio a questo vettore, le get per la public key e l'username dell'User associato alla chat e getMessages che restituisce interamente il vettore dei messaggi. Da notare come la classe Chat sia un Information Expert della classe Messaggio. Il pattern Information Expert fornisce i modelli generali associati all'assegnazione delle responsabilità agli oggetti e stabilisce che la responsabilità deve essere assegnata all'Information Expert (la classe che possiede tutte le informazioni essenziali), in questo caso, appunto, la classe Chat. Quindi, questo pattern favorisce la decentralizzazione delle responsabilità e delle decisioni nel sistema software, migliorando l'incapsulamento e riducendo così l'accoppiamento tra le varie classi. I sistemi che usano in maniera appropriata il pattern Information Expert sono più facili da capire, mantenere ed espandere.
- **User** caratterizzato dalle variabili membro id, username e publicKey. Possiede anche un vettore di chat e due vettori di messaggi, indicati dalle due associazioni 1 a molti tra User e Messaggio.
- **Messaggio** modellato con le variabili membro messageText e timestamp, con un riferimento a Chat, per la chat in cui appartiene tale messaggio, e due riferimenti alla classe User, sia per quanto riguarda il mittente che il destinatario di un dato messaggio.

4.6.4 Servizi

Il package Servizi è un package di utilità utilizzato dal Controller che espone funzionalità necessarie per la realizzazione dei compiti e delle responsabilità del package Controller, come la funzionalità di crittografia, l'interconnessione col database locale al Client e, infine, l'interfacciamento col Server, sia tramite WebSocket che tramite il protocollo applicativo HTTP.

Servizio di accesso alla rete: Network Access

NetworkAccess è la classe che ha lo scopo di gestire la connessione tra Client e Server. Com'è possibile notare dal diagramma presentato in figura 4.12, tale classe utilizza un package denominato ServerMessages, esso è stato introdotto per modellare i diversi tipi di messaggi di risposta che il client può ricevere a seguito di richieste effettuate al server. Da notare come questi messaggi siano differenti dai messaggi scambiati tra gli utenti. In particolare, abbiamo tre diversi tipi di Server Messages:

- *LoginMessage*, messaggio ricevuto dal client in risposta ad una richiesta di login e che contiene i campi *ok*, una variabile booleana che identifica se l'operazione di login è andata a buon fine, il campo *id*, che rappresenta l'ID univoco dell'utente associato al suo username, il campo *token*, il campo *puk* (*public key*) e il campo *Cprk*, ovvero la chiave privata cifrata, che potrà essere decriptata

dall’utente tramite la sua password e la cui utilità è principalmente legata alla possibilità da parte dell’utente di decifrare i messaggi cifrati che vengono ricevuti.

- *UserDataMessage* classe che rappresenta i dati di un utente di cui sono state richieste le informazioni da parte del client. E’ da notare che questo è un messaggio sempre inviato dal Server.
- *StoredMessage*, classe che memorizza le informazioni necessarie alla costruzione del nuovo messaggio ricevuto dall’utente, cifrato mediante la chiave pubblica del destinatario.

La classe NetworkAccess non ha variabili membro e presenta i seguenti metodi:

- *createWS*: metodo che inizializza una WebSocket, ovvero un canale di comunicazione bidirezionale, verso il ChatHandler. Una volta autenticato, questo canale sarà utilizzato dal Server per inviare nuovi messaggi in arrivo al client.
- *authenticateUser*: metodo di autenticazione della WebSocket, su cui viene trasmesso un messaggio contenente l’id e il token identificativo dell’utente.
- *recvMsg*: metodo invocato remotamente dal ChatHandler tramite la WebSocket e che serve per inviare al Client un nuovo messaggio.
- *registerRequest*: richiesta di registrazione effettuata dalla RegisterPage che ottiene le informazioni necessarie per tali operazioni (username, Email, password hashata, public Key e private Key cifrata)
- *messageRequest*: metodo utilizzato dal client per effettuare una richiesta di messaggio indirizzato ad uno specifico destinatario.
- *userDataRequest*: metodo utilizzato per fare una richiesta dei dati dell’utente in base al suo username. Questo metodo ritorna un server message di tipo *UserDataMessage*.
- *recvOldMsgRequest*: richiesta fatta dal client per ottenere i messaggi ricevuti mentre esso era offline. Per fare questa richiesta, l’utente deve essere loggato e il metodo avrà come ritorno una lista di stored messages, memorizzati nel server che gestisce la persistenza.
- *loginRequest*: richiesta effettuata dall’utente al login con tutti i dati validi per accedere all’app, ovvero l’username, la password hashata e il notifyToken, stringa identificativa di ogni dispositivo che consente di inviare e/o ricevere notifiche.
- *logoutRequest*: metodo richiamato dal Controller quando viene effettuato il logout. In ingresso ha bisogno dell’id e del token dell’utente, in uscita ritorna una variabile booleana per confermare o meno il logout dell’utente.

- *blockUserRequest*: metodo utilizzato per il caso d'uso di bloccare un utente, che prende in ingresso l'id e il token del bloccante e l'id dell'utente da bloccare.
- *setAvatar*: funzione richiamata per la modifica dell'immagine del profilo dell'utente.

Servizio di memorizzazione in locale: Local Storage

LocalStorage è una classe che gestisce l'accesso del client al database locale. In particolare, verranno effettuate delle operazioni di insert, get e delete sul database attraverso le chiamate dei metodi di questa classe. Essa è una classe senza variabili membro e con i seguenti metodi:

- *insertAuthInfo*: funzione di inserimento dei dati legati all'autenticazione, come username e password
- *insertUser*: funzione di inserimento di un utente all'interno della tabella degli utenti
- *insertChat*: metodo utilizzato per inserire una nuova chat nel database, dati gli id del mittente e del destinatario.
- *insertMessage*: metodo per inserire nella tabella nuovi messaggi, avente come parametri in input gli id del mittente e del destinatario, il testo del messaggio e il timestamp corrispondente all'invio.
- *getMessagesByChat*: metodo che ritorna una lista di messaggi, identificata una data chat tramite gli id del mittente e del destinatario
- *loadChats*: funzione utilizzata al login per ritornare tutte le chat attualmente attive di un determinato utente, riconosciuto con la variabile dell'id data in input.
- *deleteMsg*: metodo che richiama la delete sul database per eliminare un dato messaggio tra due utenti.
- *deleteChat*: metodo che consente l'eliminazione di un'intera chat dallo storage locale.
- *clearAuthData*: essa viene richiamata quando viene effettuato il logout, per eliminare i dati relativi all'utente appena disconnesso dal client. Di conseguenza, consiste in una delete sul database locale.

Servizi criptografici: Cryptography

La classe Cryptography ha lo scopo di garantire i requisiti di security specificati. In particolare, i metodi da essa implementati consentono di ottenere nella comunicazione: **confidenzialità** proteggendo i dati da accessi non autorizzati e **integrità** che evita la manipolazione da parte di soggetti non autorizzati. Per garantire tali proprietà vengono sfruttati due algoritmi di cifratura: AES (Advanced Encryption Standard) a chiave simmetrica e RSA (Rivest–Shamir–Adleman) algoritmo di cifratura asimmetrico che

si basa sulla generazione di due chiavi, Puk e Prk, le quali saranno assegnate a ciascun utente registrato. La classe Cryptography presenta dunque i seguenti metodi:

- *hashPsw*: ha il compito di hashare la password, utilizzando l'algoritmo SHA256. Tale password hashata sarà poi trasmessa al Server per essere memorizzata e garantire l'autenticazione dell'utente. Il Server non avrà mai visione della password in chiaro.
- *encryptPrk*: funzione che viene utilizzata per criptare la chiave privata dell'utente. Nel caso specifico, tale cifratura viene realizzata da parte del Client sfruttando la password in chiaro da esso posseduta sfruttando AES. Anche la Prk cifrata sarà trasmessa al Server per consentire la sua ritrasmissione al Client qualora questo si connettesse da un altro dispositivo. In questo modo, seppur la chiave criptata è memorizzata lato server, il server non dispone delle informazioni necessarie per decriptarla, garantendo la riservatezza dei messaggi trasmessi.
- *decryptPrk*: funzione utilizzata per decifrare la chiave privata che a sua volta viene impiegata per decifrare i messaggi ricevuti.
- *generateKeys*: funzione per la generazione di chiave pubblica (Puk) e chiave privata (Prk)
- *encryptMsg* e *decryptMsg*: funzioni che vengono impiegate per cifrare e decifrare un messaggio
- *generateKey*: funzione per la generazione di una chiave K casuale.

E' utile a tal proposito valutare come viene implementato il servizio di crittografia per la trasmissione di un messaggio: l'utente mittente genera una chiave K casuale per applicare AES. Tale chiave K viene utilizzata per cifrare un messaggio MSG, la stessa chiave K viene però cifrata con la chiave pubblica del destinatario richiesta al ChatHandler e l'insieme di $Puk(K)+K(MSG)$ vengono trasmessi. In ricezione, il destinatario sfrutta la sua chiave privata Prk per decifrare la chiave K che viene a sua volta utilizzata per decifrare il messaggio MSG, ottenendo quello in chiaro.

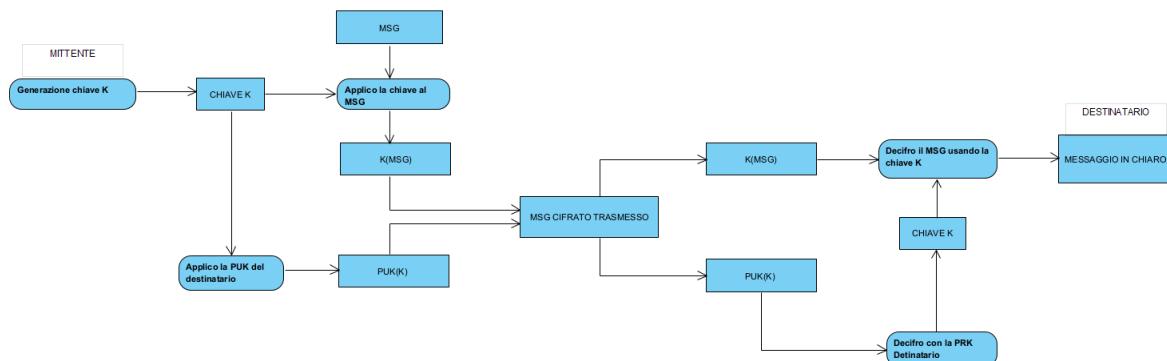


Figura 4.13: Cifratura e Decifratura di un Messaggio

4.7 Chat Handler Class Diagram

Il chat handler è il componente software che si occupa di fare da **proxy** tra i client ed i microservizi. Difatti esso funge da server per i client, che ad esso effettuano le richieste di servizi (ad esempio registrarsi al servizio, effettuare il login, inviare un messaggio etc.), e da client nei confronti dei microservizi che effettivamente implementano le funzionalità offerte dal servizio.

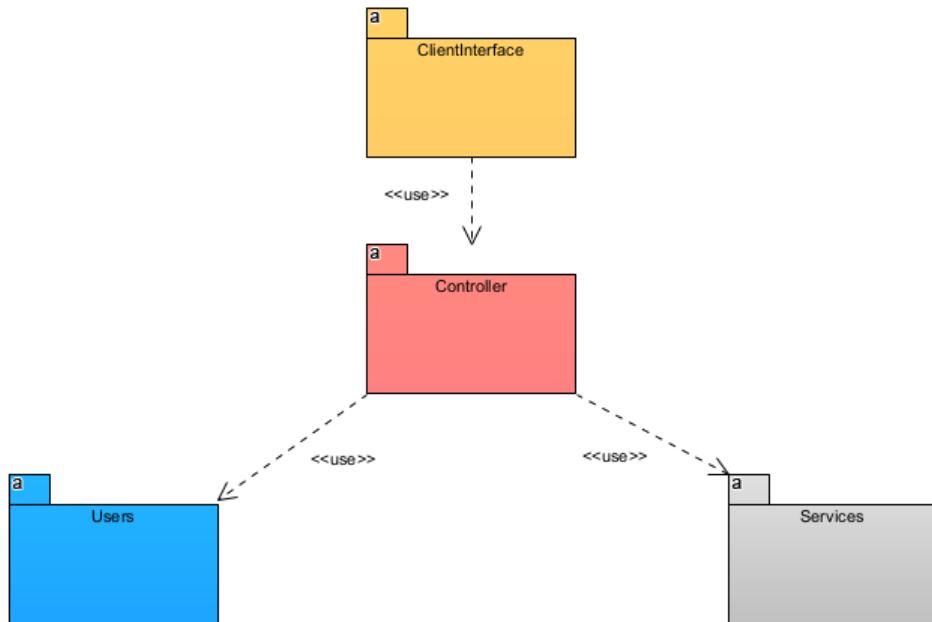


Figura 4.14: ChatHandler Package Diagram

Per il chat handler è stato usato uno *stile architetturale* di tipo **layered**. In particolare, risulta evidente, attraverso livelli di astrazione crescenti, la distinzione tra l'interazione con un generico client tramite l'interfaccia *ClientInterface*, la gestione della logica di controllo delle richieste tramite il livello *Controller* e l'interazione con i microservizi che forniscono le effettive funzionalità richieste. Nell'ultimo layer, è inoltre presente il package *Users* il quale si occupa di modellare le entità del sistema rilevanti per il componente chat handler, ovvero una classe *ConnectionRegister* la quale mantiene una lista di riferimenti ad istanze della classe *ConnectedUser*, che serve per modellare un utente connesso al servizio.

A livello di responsabilità, il Chat Handler deve smistare richieste del client ai vari servizi, mantenere una connessione con tutti gli utenti online e, infine, garantire l'autenticità delle richieste utente.

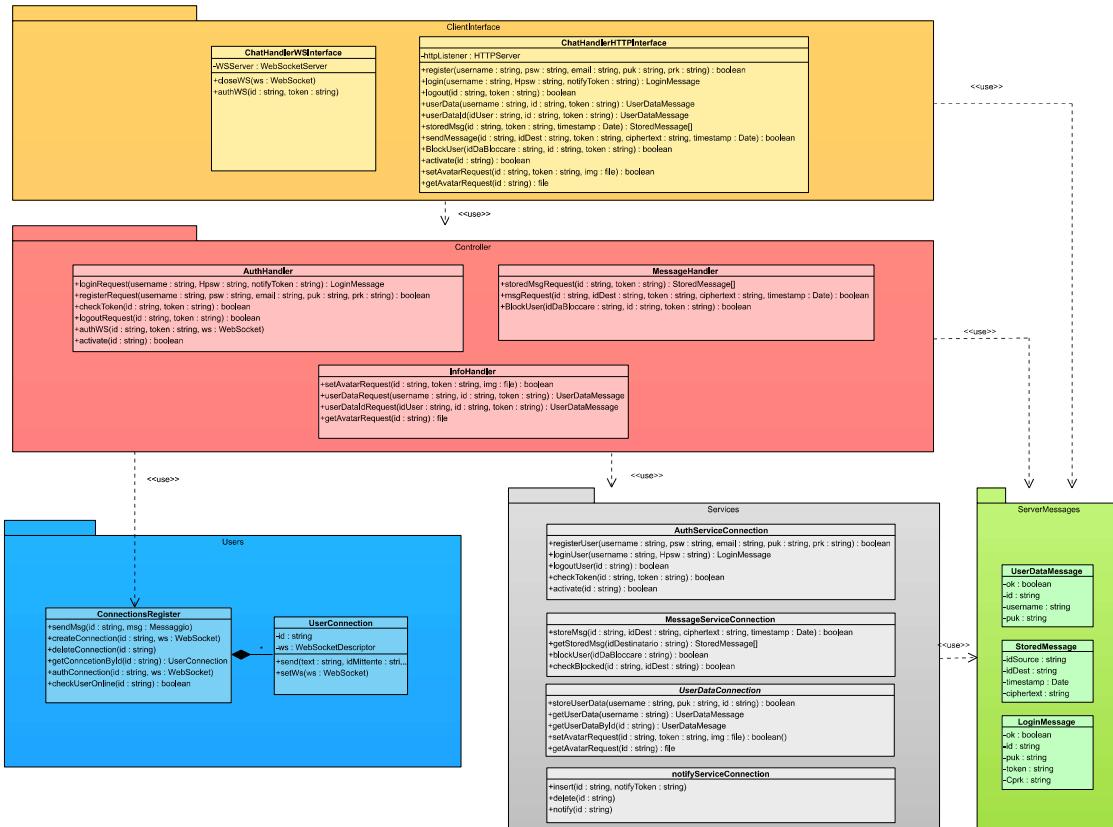


Figura 4.15: ChatHandlerClassDiagram

• ClientInterface:

- **ChatHandlerWSInterface:** Interfaccia tramite la quale un client può effettuare una richiesta di autenticazione della WebSocket e di chiusura della stessa per implementare una disconnessione dell’utente dal servizio.
- **ChatHandlerHTTPInterface:** Interfaccia di tipo HTTP tramite la quale un client può effettuare innanzitutto richieste di registrazione (metodo *register*), login (metodo *login*) e logout (metodo *logout*). Espone inoltre i metodi per richiedere i dati di un utente (in tal caso username, id e chiave pubblica) tramite i metodi *userData* e *userDataId* (il primo sarà implementato ricercando a partire dall’username e il secondo a partire dall’ID). Il metodo *storedMsg* può essere usato per richiedere una lista degli eventuali messaggi non ancora ricevuti dall’utente (identificato da id e token), i quali sono memorizzati nel database associato al microservizio msgService. Il metodo *sendMessage* permette di effettuare una richiesta HTTP di invio messaggio, il cui body è costituito dai campi forniti in ingresso al metodo (id, id utente destinatario, token, messaggio crittografato e timestamp). Ancora, il metodo *BlockUser* permette ad un client di bloccare un utente, fornendo in ingresso al metodo l’ID dell’utente

da bloccare, il proprio Id e il proprio token, ed infine il metodo di attivazione dell'account (*activate*) di un utente tramite una apposita mail ricevuta nella casella postale dell'indirizzo specificato durante la compilazione del form di registrazione.

- **Controller:**

- **AuthHandler:**

- * Esecuzione operazione di Login di un utente registrato (*loginRequest*). Quando il server HTTP riceve una richiesta di *login* viene invocato il metodo *loginRequest* dell'AuthHandler, il quale a sua volta invoca il metodo *loginRequest* dell'AuthServiceConnection (passandogli username e password dell'utente, prelevati dal body della richiesta HTTP) che restituirà all'AuthHandler un LoginMessage associato all'utente. Se tale richiesta va a buon fine allora l'utente viene iscritto al meccanismo di notifiche tramite il suo *notifyToken* (richiamando la *insert* del *NotifyServiceConnection*).
 - * Esecuzione operazione di Registrazione di un utente (*registerRequest*). Quando il server HTTP riceve una richiesta di *registrazione* viene invocato il metodo *RegisterRequest* dell'AuthHandler, il quale a sua volta invoca il metodo *RegisterRequest* dell'AuthServiceConnection il quale farà una richiesta HTTP al servizio di autenticazione che si occuperà di memorizzare nel proprio database le informazioni dell'utente (userName, password, email, puk, prk) ed associargli un id univoco. In caso di successo poi la classe *UserDataConnection* si occuperà di richiedere al servizio di gestione dei dati pubblici degli utenti la memorizzazione di username, chiave pubblica e id dell'utente appena registrato.
 - * Esecuzione operazione di Logout di un utente registrato (*logoutRequest*). Quando il server HTTP riceve una richiesta di *logout* viene invocato il metodo *LogoutRequest* dell'AuthHandler, il quale a sua volta invoca il metodo *LogoutRequest* dell'AuthServiceConnection che si occuperà di richiedere al servizio di autenticazione degli utenti la rimozione del token associato all'utente che vuole fare il logout. Successivamente verrà rimosso l'utente dal *ConnectionsRegister* e verrà disiscritto dal servizio di notifiche.
 - * Esecuzione operazione di check del token di un utente (*checkToken*). Quando il server HTTP riceve una richiesta di *check* del token viene invocato il metodo *checkToken* dell'AuthHandler, il quale a sua volta invoca il metodo *checkToken* dell'AuthServiceConnection, il quale richiamerà il servizio di autenticazione degli utenti per chiedere se il token associato all'id fornito nella richiesta HTTP è quello corretto.
 - * Esecuzione operazione di autenticazione della WebSocket associata ad un utente (*authWS*).

- * Esecuzione operazione di attivazione dell'account di un utente registrato tramite la sua e-mail. (*activate*). Il metodo è invocato in corrispondenza di una richiesta GET (fatta tramite il browser) al link trasmesso nella mail.

– **MessageHandler:**

- * Esecuzione richiesta di ricezione dei messaggi non ancora ricevuti da un utente (*storedMsgRequest*). La richiesta verrà inoltrata, se il token è corretto, al *MessageServiceConnection* chiamando il suo metodo *getStoredMsg* che richiederà la lista dei messaggi non ancora ricevuti dal client associato all'id specificato come parametro in ingresso (idDestinatario) al servizio MessageService.
- * (*msgRequest*). La richiesta verrà inoltrata, se il token è corretto, al *MessageServiceConnection* chiamando il suo metodo *storeMsg* che richiederà la rispettiva funzionalità al MessageService
- * Esecuzione richiesta da parte di un utente di bloccare un altro utente indesiderato (*BlockUser*).

– **InfoHandler:**

- * Esecuzione operazione per inserire un avatar di un utente dato il suo id e che ritorna una variabile booleana in base al suo esito *setAvatarRequest*.
- * Esecuzione operazione per avere come output l'avatar di un utente, dato l'id in ingresso *getAvatarRequest*
- * Esecuzione operazione di richiesta dei dati di un utente, con ricerca a partire dallo username di quest'ultimo (*userDataRequest*).
- * Esecuzione operazione di richiesta dei dati di un utente, con ricerca a partire dall'id di quest'ultimo (*userDataIdRequest*).

- **Users:** Tale package è stato introdotto per tenere traccia degli utenti connessi al servizio. In particolare ritroviamo due classi:

- **UserConnection:** modella il singolo utente connesso cui è associato oltre che un *id* anche la corrispondente WebSocket tramite cui è possibile trasmettere i vari messaggi. Tra le operazioni definite, infatti, ritroviamo oltre che le get e set classiche la funzione di *send*.
- **ConnectionsRegister:** altro non è che un insieme di utenti connessi, esso è necessario proprio per consentire ad un qualsiasi utente di ricercare e individuare la socket tramite cui trasmettere uno specifico messaggio. Infatti attraverso la *sendMsg* viene richiamata la *send* dello specifico utente per l'invio tramite la socket. Esso tiene traccia degli utenti online e tramite la funzione *checkUserOnline* viene deciso se il messaggio possa essere inviato direttamente al destinatario

oppure esso debba essere memorizzato per una consegna futura. Inoltre, il ConnectionRegister in quanto contenitore di Utenti Connessi ha la responsabilità di crearli, eliminarli o ricercarli.

- **Services:** Il package Services contiene le classi le cui funzioni saranno mappate nelle corrispondenti funzionalità dei microservizi. In generale, verranno effettuate le richieste HTTP necessarie e settati opportunamente i campi di tali richieste. Tale package costituisce uno strato fondamentale per l'adattamento alle interfacce dei microservizi stessi.
- **ServerMessages:** Tale package contiene le classi che definiscono le tipologie di messaggi ritornati come risposta dal Server. Essi sono già stati precedentemente esplicitati in 4.6.4

4.7.1 Interfaccia logica

Funzione esposta	Tipo di ritorno
register()	boolean
login()	LoginMessage
logout()	boolean
userData()	UserDataMessage
userDataById()	UserDataMessage
sendMessage()	boolean
storedMessage	list<StoredMessage>
blockUser()	boolean
activateUser()	string
authWS()	void
closeWS()	void
setAvatarRequest()	boolean
getAvatarRequest()	file

Interfaccia fisica

I metodi esposti al client sono implementati tramite due interfacce fisiche del ChatHandler:

- una interfaccia RESTful HTTP
- una interfaccia WebSocket

Tabella 4.1: Metodi interfaccia HTTP

METHOD	URL	REQUEST BODY	RESPONSE BODY
POST	/register	{ "username": <username>, "email": <email>, "password": <password>, "privateKey": <privateKey>, "publicKey": <publicKey> }	{ "ok": <result> }
POST	/login	{ "username": <username>, "password": <password>, "notifyToken": <notifyToken> }	{ "ok": <result>, "id": <id>, "token": <token>, "PrK": <PrK> }
POST	/logout	{ "id": <id>, "token": <token> }	{ "ok": <result> }
POST	/userData	{ "id": <id>, "token": <token> }	{ "ok": <result>, "id": <id>, "username": <username>, "PuK": <PuK> }
POST	/userDataById	{ "id": <id>, "token": <token>, "idUser": <idUser> }	{ "ok": <result>, "id": <id>, "username": <username>, "PuK": <PuK> }
POST	/sendMessage	{ "id": <id>, "idDest": <idDest>, "timestamp": <timestamp>, "cipherText": <cipherText>, "token": <token> }	{ "ok": <result> }
POST	/storedMessage	{ "id": <id>, "token": <token> }	{ "messages": [..., { "idMittente": <idMittente>, "idDest": <idDest>, "time- stamp": <timestamp>, "cipher- Text": <cipherText> }, ...] }

Continuo della Tabella 4.1			
METHOD	URL	REQUEST BODY	RESPONSE BODY
POST	/blockUser	{ "id":<id>, "token":<token>, "idBlocked":<idBlocked> }	{ "ok":<result> }
GET	/activate	{ "id":<id> }	{ "ok":<result> }
GET	/avatar/:id		imagefile
POST	/setAvatar	imagefile, { "id":<id>, "token":<token> }	{ "ok":<ok> }

TYPE	MESSAGE BODY
onMessage	{ "id":<id>, "token":<token> }
onClose	{}

4.8 AuthService Class Diagram

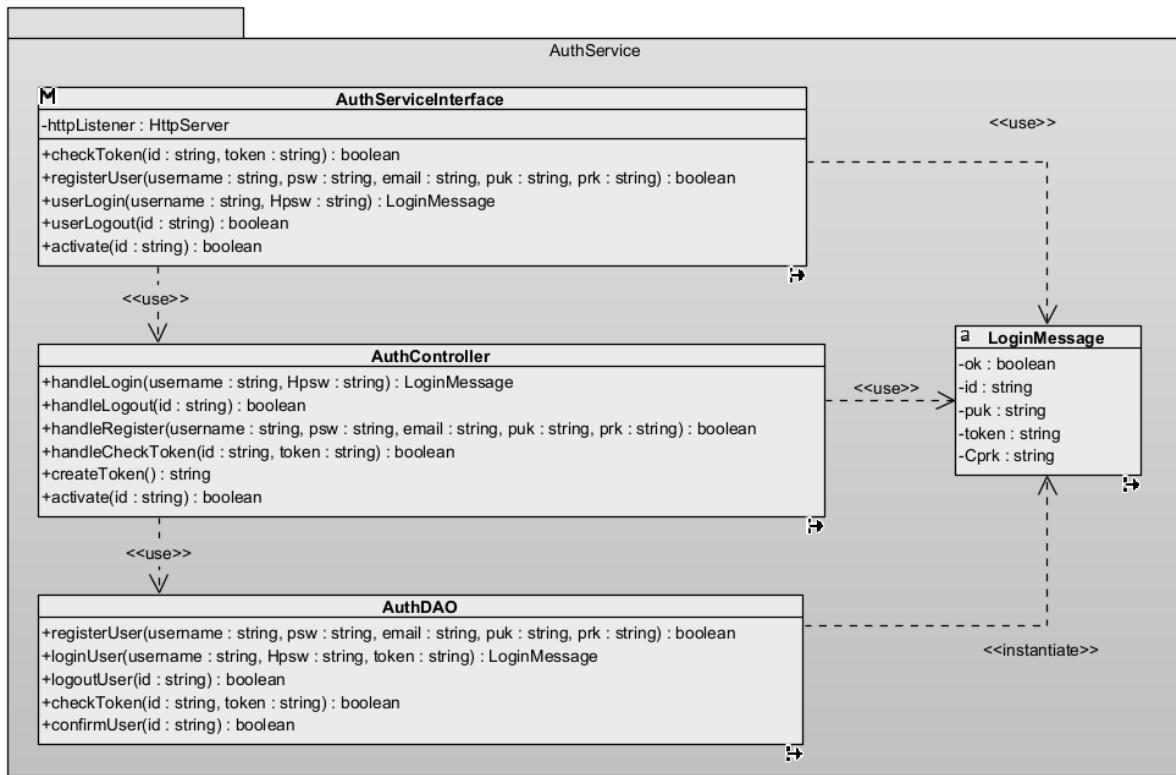


Figura 4.16: AuthServiceClassDiagram

Il microservizio chiamato AuthService gestisce le informazioni dell'account riguardanti l'accesso e l'autenticazione nell'app, memorizza ed utilizza quindi, i dati privati che non possono essere visualizzati dagli altri utenti. Collegato tramite una interfaccia HTTP al ChatHandler, esso è stato progettato attraverso una architettura a livelli, in cui il primo strato offre un'interfaccia predefinita, il secondo implementa la logica di business del servizio e il terzo si dedica alla gestione della persistenza dei dati. Come già descritto in precedenza, ciascuno strato è in comunicazione diretta con quello inferiore ovvero richiede servizi allo strato sottostante. Le responsabilità del microservizio sono:

- gestire la registrazione dell'utente
- attivare la registrazione di un account tramite la conferma della mail
- gestire le operazioni di login e logout di un utente
- controllare la validità del token dell'utente
- generare il token identificativo di un utente, essendo questa una informazione privata e relativa all'autenticazione

Inoltre, è da notare come venga utilizzato dalle classi del microservizio il Server Message di tipo LoginMessage, descritto nei paragrafi precedenti e istanziato proprio dal DAO di questo microservizio.

L'architettura del microservizio è composta quindi da tre classi:

- *AuthServiceInterface*: la classe ha una variabile membro *httpListener* su cui può ricevere richieste HTTP dal ChatHandler. Come metodi possiede *checkToken*, che utilizzando l'id e il token di utente verifica se quel token è valido, ritornando una variabile booleana; *registerUser*, utilizzata per la registrazione e che ritorna una variabile booleana in caso di registrazione avvenuta con successo; *userLogin*, che prende in ingresso l'username dell'utente e la sua password hashata e fornisce in output un oggetto *LoginMessage*; *userLogout*, che viene usato per il caso d'uso di effettua logout dell'utente autenticato; *activate*, metodo utilizzato per l'attivazione di un account, dato in ingresso il suo id.
- *AuthController* ha la funzione *handleLogin*, che viene richiamata da *userLogin* per la gestione del login dell'utente. Stesso discorso vale anche per *handleRegister*, *handleLogout*, *activate* e *handleCheckToken*, che hanno la stessa firma delle funzioni del package sovrastante. Si può notare come effettivamente la dipendenza d'uso espressa tra i due strati adiacenti venga realizzata nell'implementazione delle due classi. Infine, abbiamo la funzione di *createToken*, che si occupa di generare la stringa del token identificativo di un utente.
- *AuthDAO*: anche in questo caso le funzioni di *registerUser*, *confirmUser*, *loginUser*, *logoutUser* e *checkToken* vengono chiamate dalle funzioni della classe appena descritta, andando in questo caso ad operare delle operazioni sul database locale al microservizio. Nel primo caso verrà implementata un'operazione di scrittura (e quindi di insert); nel secondo caso invece un'operazione di aggiornamento (update), in cui verrà modificato il campo di conferma di registrazione dell'account; nei casi rimanenti, invece, più semplicemente una operazione di lettura (e quindi di get).

4.8.1 Interfaccia logica

Funzione esposta	Tipo di ritorno
<i>checkToken()</i>	boolean
<i>registerUser()</i>	boolean
<i>userLogin()</i>	<i>LoginMessage</i>
<i>userLogout()</i>	boolean
<i>activate()</i>	boolean

Interfaccia fisica

METHOD	URL	REQUEST BODY	RESPONSE BODY
POST	/checkToken	{ "id":<id>, "token":<token> }	{ "ok":<result> }
POST	/register	{ "username": <username> , "email":<email>, "password":<password>, "privateKey":<privateKey>, "publicKey":<publicKey> }	{ "ok":<result> }
POST	/login	{ "username":<username>, "password":<password> }	{ "ok":<result>, "id": <id>, "token":<token>, "privateKey":<privateKey> }
POST	/logout	{ "id":<id> }	{ "ok":<result> }
GET	/activate	{ "id":<id> }	{ "ok":<result> }

4.9 MessageService Class Diagram

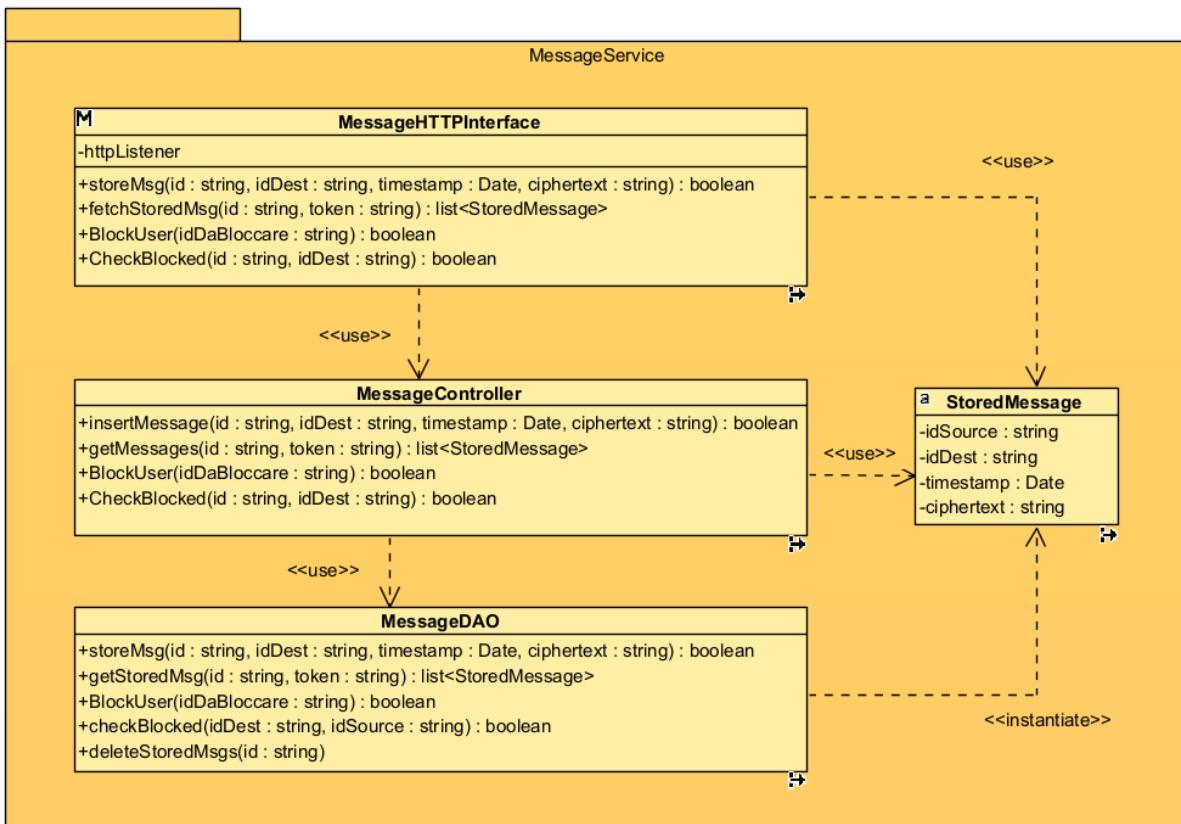


Figura 4.17: MsgServiceClassDiagram

Il microservizio **MessageService** si occupa della persistenza dei messaggi, qualora essi siano indirizzati ad un utente attualmente offline, ma anche di tener tracce di eventuali utenti bloccati. L'architettura a livelli è la stessa del microservizio di **AuthService**, con l'unica differenza, chiaramente, nei metodi che svolgeranno compiti diversi. Inoltre, è possibile notare come i metodi di uno strato superiore richiedano i servizi dello strato inferiore, a dimostrazione della dipendenza d'uso esistente tra le classi.

Le responsabilità di questo microservizio sono:

- memorizzare messaggi diretti ad utenti offline, che quindi non possono ricevere alcun tipo di messaggio
- recuperare tutti i messaggi memorizzati appena l'utente ritorna online in modo da inviarglieli
- bloccare un utente in modo che un utente non riceva più messaggi dall'utente bloccato
- controllare se l'id associato ad un certo utente è stato bloccato o meno

E' stato deciso di assegnare la responsabilità di bloccare un utente a questo microservizio in quanto è il servizio adibito a gestire la consegna dei messaggi.

Per quanto riguarda i Server Messages utilizzati, le classi di MessageService utilizzano il tipo di messaggio `StoredMessage`, istanziato dalla classe DAO, che si trova nell'ultimo strato dell'architettura. Come notiamo dalla figura, la strutturazione a livelli del microservizio ci permette di identificare 3 diverse classi:

- *MessageHTTPInterface*: classe di interfaccia col ChatHandler, con cui comunica attraverso il protocollo HTTP. Essa ha una variabile membro `httpListener`, su cui poter ricevere le richieste HTTP. Per quanto riguarda i metodi il più importante è `storeMsg`, che, dati in ingresso gli id mittente, id destinatario, testo cifrato e timestamp, si occupa di memorizzare il messaggio in input nel database. Nel caso l'operazione avvenga con successo ritornerà una variabile booleana di valore true. `fetchStoredMsg` è il metodo che si occupa di ritornare una lista di `StoredMessage` da trasmettere all'utente appena esso ritorna online. Il metodo `BlockUser`, invece, implementa il caso d'uso di blocca utente, prendendo in ingresso l'id dell'utente da bloccare e ritornando un boolean di valore true in caso l'operazione avvenga con successo. Infine, il metodo `CheckBlocked` il quale verifica se il mittente è stato bloccato dall'utente al quale vuole inviare il messaggio.
- *MessageController* rimappa i metodi del livello superiore `MessageHTTPInterface` nelle sequenze di chiamate necessarie per implementare la logica richiesta.
- *MessageDAO*, che ha il compito di fare da interfaccia verso il database locale al microservizio. In questo caso i metodi della classe, chiamati, come prima, dai metodi della classe di layer superiore, sono mappati in azioni CRUD sul database: in particolare, `storeMsg` e `BlockUser` effettueranno un'operazione di scrittura; `getStoredMsg` e `checkBlocked` eseguiranno un'operazione di lettura (get); infine, `deleteStoredMsg` effettuerà un'operazione di eliminazione dei dati memorizzati nel database in quanto col ritorno online dell'utente i vecchi messaggi saranno consegnati e non c'è più necessità di memorizzarli.

4.9.1 Interfaccia logica

Funzione esposta	Tipo di ritorno
<code>storeMsg()</code>	boolean
<code>fetchStoredMsg()</code>	<code>StoredMessage[]</code>
<code>blockUser()</code>	boolean
<code>checkBlocked()</code>	boolean

Interfaccia fisica

METHOD	URL	REQUEST BODY	RESPONSE BODY
POST	/store	{ "idMittente":<idMittente>, "idDest":<idDest>, "timestamp":<timestamp>, "cipherText":<cipherText> }	{ "ok":<result> }
POST	/fetchStored	{ "id": <id> }	{ "messages": [...,{ "idMittente":<idMittente>, "idDest":<idDest>, "timestamp":<timestamp>, "cipherText":<cipherText> },...] }
POST	/blockUser	{ "id":<id>, "idBlocked":<idBlocked> }	{ "ok":<result> }
POST	/checkBlocked	{ "id":<id>, "idBlocked":<idBlocked> }	{ "ok":<result> }

4.10 UserDataService Class Diagram

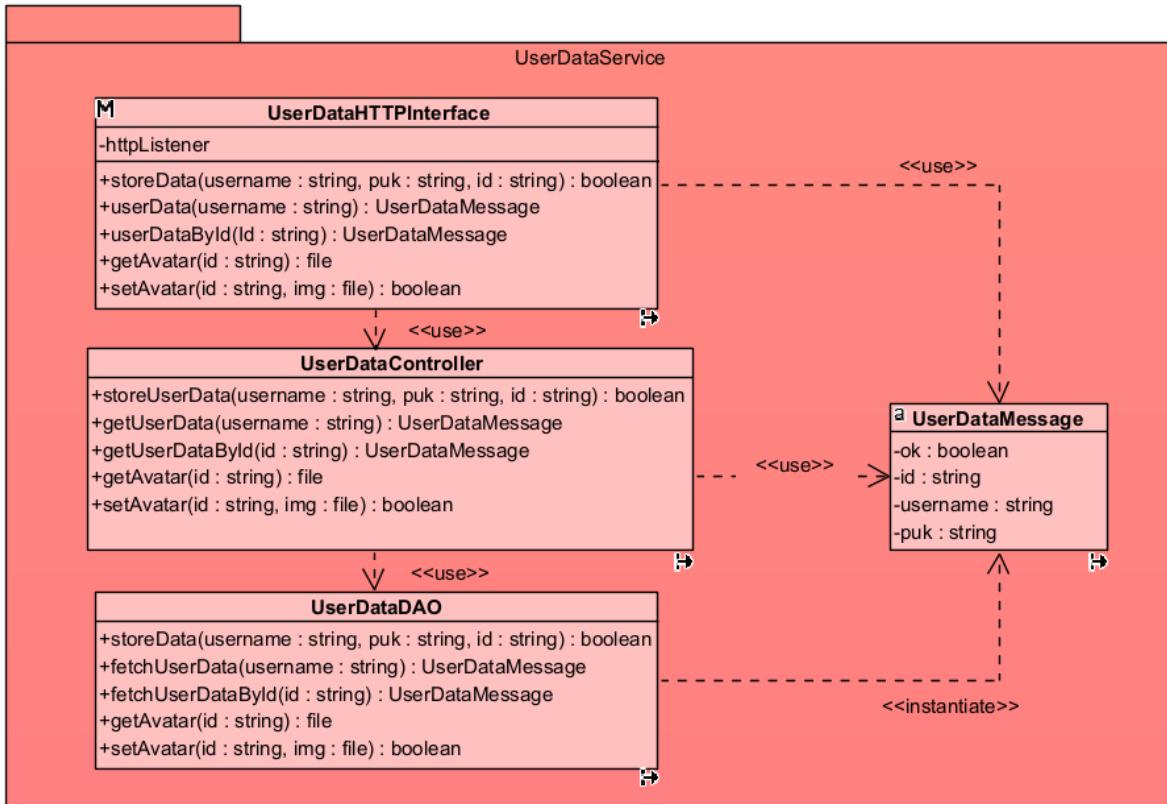


Figura 4.18: UserDataService

UserDataService è il microservizio che ha il compito di memorizzare tutte le informazioni pubbliche di un dato utente, in modo che qualsiasi altro utente, autenticato all'interno dell'app, possa accedervi. Ottenere i dati di un altro utente può servire, per esempio, a creare una chat con esso oppure a visualizzare le informazioni del suo profilo.

L'architettura del microservizio è multi-layered con tre layer, ciascuno dei quali richiede i servizi del layer sottostante. UserDataService comunica attraverso un'interfaccia HTTP con il ChatHandler, da cui può ricevere, come tutti gli altri microservizi, delle richieste di dati.

Le responsabilità assegnate a questo microservizio sono:

- memorizzare i dati pubblici relativi ad un utente quando esso si registra.
- distribuire i dati pubblici di un utente come il suo Username o la sua chiave pubblica, informazioni necessarie al client per avviare una nuova chat. Il servizio consente di ricavare queste informazioni per un dato utente specificandone o l'username, funzionalità importante quando un utente crea manualmente una nuova chat, o l'id, funzionalità utilizzata quando, alla ricezione di un messaggio, è necessaria la creazione automatica di una nuova chat tramite l'id del mittente, ottenuto dal messaggio appena ricevuto.

Il tipo di Server Message utilizzato, come si vede in figura 4.18, è l'UserDataMessage, istanziato sempre dalla classe DAO del microservizio. Questo è un messaggio che, chiaramente, contiene i dati pubblici di un utente, come id, username e chiave pubblica.

Le classi di questo microservizio sono:

- UserDataHTTPInterface: classe interfaccia col chatHandler, da cui riceve richieste di dati attraverso l'httpListener, variabile membro della classe. I metodi della classe sono essenzialmente molto semplici: storeData, che ha il compito di inserire nel database i dati di un utente appena registrato (username, id e publicKey) e che ritorna una variabile booleana in base al successo o al fallimento dell'operazione; userData e userDataById, che ritornano un UserDataMessage rispettivamente in base all'username o all'id di un utente specifico; setAvatar e getAvatar, metodi utili per la gestione dell'avatar di un utente;
- UserDataController: come per gli altri microservizi, anche qui i metodi di questa classe sono chiamati dai metodi della classe del layer superiore, come si può vedere anche dalla stessa firma che hanno le funzioni;
- UserDataDAO: classe che gestisce l'interfacciamento col database locale al microservizio. Nello specifico, il metodo storeData eseguirà un'operazione di insert sul database mentre gli altri due metodi, fetchUserData e fetchUserDataById, effettueranno un'operazione di get.

4.10.1 Interfaccia logica

Funzione esposta	Tipo di ritorno
storeData()	boolean
userData()	UserDataMessage
userDataById()	UserDataMessage
getAvatar()	file
setAvatar()	boolean

Interfaccia fisica

METHOD	URL	REQUEST BODY	RESPONSE BODY
POST	/storeData	{ "username":<username>, "puk":<puk>, "id":<id> }	{ "ok":<result> }
POST	/userData	{ "username": <username> }	{ "ok": <result>, "id":<id>, "username": <username>, "puk": <puk> }
POST	/blockUser	{ "id":<id> }	{ "ok": <result>, "id":<id>, "username": <username>, "puk": <puk> }
GET	/avatar/:id		imagefile
POST	/setAvatar/:id	imagefile	{ "ok": <resp> }

4.11 NotifyService Class Diagram

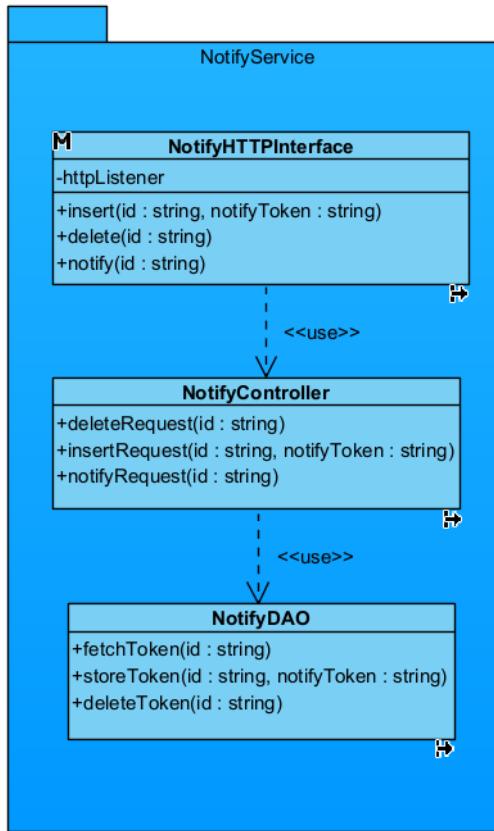


Figura 4.19: NotifyService

Il microservizio *NotifyService* è responsabile della gestione delle notifiche push trasmesse all'atto della ricezione di un messaggio. Ancora una volta l'architettura di riferimento per il microservizio è un'architettura a livelli. L'intera gestione delle notifiche, in particolare, si basa sulla generazione da parte del Client stesso di un *NotifyToken* il quale è necessario per individuare univocamente il dispositivo di un dato Utente. Tale Token viene generato all'atto del Login e trasmesso nella richiesta di Login consentendo all'Utente di impostare la sua preferenza circa le modalità di ricezione delle notifiche stesse. Le responsabilità di questo microservizio sono:

- Inserimento di un *NotifyToken*
- Eliminazione di un *NotifyToken*
- Trasmissione di una notifica

Come già riportato nei microservizi precedenti la strutturazione a livelli rimanda a 3 classi principali:

- *NotifyServiceInterface* con una variabile membro `httpListener` e le tre funzioni di `notify`, `insert` e `delete`.

- *NotifyController* sul quale vengono mappate le funzioni definite nell’interfaccia HTTP.
- *NotifyDAO* classe che tiene traccia del mapping tra l’id utente e il NotifyToken associato.

In realtà, tale microservizio per esplicitare la funzione desiderata si avvale di un ulteriore servizio esterno, indicato come *AndroidNotificationService*, al quale viene trasmessa una richiesta HTTP apposita, a livello Controller, contenente le informazioni necessarie per l’invio della notifica.

4.11.1 Interfaccia logica

Funzione esposta	Tipo di ritorno
insert()	void
delete()	void
notify()	void

Interfaccia fisica

METHOD	URL	REQUEST BODY	RESPONSE BODY
POST	/insertToken	{ "id":<id>, "token":<token> }	{}
POST	/deleteToken	{ "id": <id> }	{}
POST	/notifyUser	{ "id":<id> }	{}

Capitolo 5

Implementazione

5.1 Framework di sviluppo

React Native

Per l'implementazione del client abbiamo utilizzato il framework React Native, un framework che consente, lavorando in JavaScript, di realizzare applicazioni android e iOS. Abbiamo scelto questo framework perché offre moltissime possibilità per quanto riguarda la realizzazione di interfacce utente interattive ma lascia allo stesso tempo libertà per quanto riguarda la realizzazione della logica implementativa dell'applicazione.

I componenti React sono particolari classi che ereditano dalla classe *React.Component*. Queste classi ridefiniscono un metodo *render()* che specifica l'insieme dei componenti che la vista mostra a schermo. Il metodo *render()* è invocato al momento della creazione della vista oppure quando una delle variabili membro è modificata con il metodo *setState()*, ereditato dalla classe *React.Component*. Le viste React sono organizzate gerarchicamente: una vista nel suo metodo render può infatti instanziare un altro componente o vista React passandogli dei parametri di ingresso, i **props**, che diventeranno parte dello stato del componente. L'applicazione all'avvio istanzia un *React.Component* di nome **App** utilizzato a sua volta per istanziare le classi del controller e seleziona la prima view da mostrare all'utente.

Express

Express è un framework per NodeJS utilizzato allo scopo di realizzare server web. Il framework fornisce una classe che implementa le funzionalità di un server web HTTP e che può essere specializzata ridefinendone i metodi di post e get associando a richieste su specifici URL delle chiamate a funzioni opportunamente definite, le quali sono invocate allo scopo di produrre la risposta attesa dai client. Express è inoltre compatibile con la tecnologia delle WebSocket, utilizzata per la realizzazione del canale di connessione bidirezionale tra user application e ChatHandler.

5.2 Descrizione dei file realizzati

Viene riportata una spiegazione sintetica (ma sufficientemente dettagliata ai fini di una buona comprensione) di tutti i file realizzati per l'applicazione:

Client>Model

- **messaggio, user, chat**: rappresentano le entità del dominio di interesse, in particolare chat è Information Expert di messaggio.
- **loggedUser**: è Information Expert di chat; attraverso una particolare chat riusciamo ad ottenere informazioni direttamente riguardo l'utente destinatario e, inoltre, questa classe svolge il ruolo di *Facade* per l'intero package.

Client>View

- **registrationPage**: pagina della registrazione che renderizza dei Text, TextInput e Button opportunamente stilizzati attraverso la creazione di una *StyleSheet*. La pressione del tasto di registrazione innesca la chiamata del controller che effettua la routine di registrazione. All'interno del componente è stato aggiunto un event listener nel caso in cui l'utente "torni indietro" sul suo smartphone, utilizzando la **BackHandler API** di React Native.
- **loginPage**: pagina del login in cui vengono renderizzati i TextInput per il login, l'icona per mostrare o meno la password ed una checkbox per selezionare la modalità "Remember me" al prossimo login; da notare come sia stato definito uno state, per quanto riguarda il mostrare o meno la password, se mostrare o meno la registrationPage o se è stato selezionato il "RememberMeLogin" come modalità di accesso; questi stati verranno settati attraverso il corrispondente metodo setState, che permetterà di renderizzare nuovamente tutto il componente React.
- **mainPage**: pagina principale in cui sono mostrate tutte le chat, un bottone per fare il Logout, uno per creare una nuova chat con il textView associato dove scrivere il nome utente destinatario e infine un bottone per cercare tra le varie chat esistenti. Le chat sono visualizzate come componenti separati, come vedremo successivamente, inserite all'interno di una ScrollView, ovvero un contenitore che è possibile scorrere. Gli stati di questa page sono il vettore delle chat attualmente attive e un booleano per indicare se è stata aperta o meno una particolare chat; in particolare questa variabile booleana diventerà true quando viene creata una nuova chat o cliccata una già esistente, false quando la chat viene chiusa.
- **chatPage**: pagina in cui viene renderizzata una particolare chat, in cui è possibile inviare nuovi messaggi, bloccare l'utente destinatario o eliminare la chat in locale; da notare come ogni messaggio

sia visualizzato come un component separato e il contenitore dei messaggi scorra in automatico in altezza ogni volta che un nuovo messaggio viene inviato/ricevuto; lo stato di questo componente comprende il messaggio che si sta scrivendo e laltezza della pagina; laltezza viene settata in base a degli eventi, come mostrare o nascondere la tastiera, su cui sono stati inseriti opportunamente degli event listener.

Client>Controller Contiene i controllori che gestiscono la logica dell'applicazione lato client. Tra questi riportiamo:

- **Controller.js**: facade del package controller.
- **ChatController.js**: gestisce le funzionalità di creazione ed eliminazione delle chat e blocco di utenti.
- **LoginController.js**: gestisce logica di login, remember me e notifiche.
- **MessageController.js**: gestisce le richieste di invio e ricezione di un messaggio.
- **RegistrationController.js**: si occupa di realizzare la registrazione.

Client>Services

- **cryptoService.js**: presenta le funzioni crittografiche per implementare **RSA** e **AES** utilizzando la libreria crypto-Js.
- **LocaslStorage.js**: gestisce l'accesso al database locale per la memorizzazione delle informazioni di interesse. Utilizza, per l'accesso al database locale, la libreria SQLite.
- **networkAccess.js**: si occupa di effettuare tutte le richieste previste all'*API Rest* del server, utilizzando il metodo *fetch()* offerto da javascript. Gestisce inoltre la websocket utilizzata per la consegna in tempo reale dei messaggi da parte del server. Implementa quindi la funzione di autenticazione del canale di comunicazione ed una callback che gestisce la ricezione di un messaggio.
- **networkConfig.js**: file di configurazione che memorizza le informazioni sull'indirizzo ip del chatandler e sul porto in cui questo è in ascolto.

Client>components

- **Message**: componente per la visualizzazione di un messaggio, con un diverso stile dipendentemente dal fatto se è un messaggio inviato o ricevuto.
- **Conversation**: componente per la visualizzazione di una chat, che renderizza a schermo l'avatar e l'id del destinatario, con l'anteprima dell'ultimo messaggio e l'orario di invio di questo; si noti

come tutto il componente sia racchiuso in un *TouchableOpacity*, un componente React Native che diventa opaco quando premuto.

server>chatHandler

chatHandler>interface

- **ClientInterface**: permette di istanziare il server HTTP e il WebSocketServer; vengono definiti tutti gli URL delle richieste HTTP possibili che può mandare il client, specificando le funzioni da chiamare per ogni richiesta, attraverso i metodi post e get.

chatHandler>controller

- **controllerFacade**: classe Facade che permette di istanziare tutti i controller del package.
- **authHandler, infoHandler, msgHandler**: classi controller, che ritornano una risposta con Header impostato tramite il metodo setHeader e con contenuto una stringa JSON ottenuta mediante il metodo JSON.stringify

chatHandler>services

- **authServiceConnection, msgServiceConnection, notifyServiceConnection, userDataConnection**: classi di interfacciamento con i quattro microservizi. Esse utilizzano una funzione *fetch()* per fare richieste HTTP ai microservizi, specificando il metodo (GET o POST), gli header e il body, sempre ottenuto come stringa JSON di tutti i campi attraverso il metodo JSON.stringify; infine, della risposta alla richiesta ci interessa solo la parte json, ottenuta tramite il metodo apposito

chatHandler>users

- **userConnection**: classe che modella la connessione utente; è possibile inviare un messaggio sulla WebSocket tramite il metodo send, che prende in ingresso una stringa JSON contenente i campi del messaggio.
- **connectionsRegister**: classe Information Expert di connectedUser.

server>authService

- **AuthServiceInterface**: classe di interfaccia HTTP su cui può ricevere richieste dal chatHandler; chiaramente, viene utilizzato un port differente rispetto all'interfaccia HTTP del chatHandler
- **AuthController**: classe controller del microservizio; da notare come sia per la generazione dell'id in fase di registrazione che quella del token in fase di login venga utilizzata una funzione della libreria

"**uuid**". Le risposte, fornite dal DAO, vengono inviate come stringhe JSON ottenute tramite il metodo `JSON.stringify`. Per quanto riguarda l'invio della mail di conferma a valle della registrazione, viene creato un oggetto "*transporter*" attraverso il metodo `createTransport` della libreria **node-mailer** e su tale oggetto viene chiamato il metodo `sendMail`, avendo già settato opportunamente una variabile `mailOptions` dove è indicato mittente, destinatario, oggetto e testo.

- **AuthDAO** : mantiene la persistenza del servizio di autenticazione.

server>notifyService

- **notifyServiceInterface**: classe di interfaccia HTTP per ricevere richieste dal ChatHandler
- **requestController**: gestisce e implementa le funzioni per il servizio di notifica interfacciandosi anche col servizio esterno per la trasmissione effettiva della notifica push.
- **DAO**: mantiene la corrispondenza tra gli utenti e il notifyToken corrispondente.

server>msgService

- **msgServiceInterface**: interfaccia HTTP per la ricezione delle richieste dal parte del chatHandler
- **requestController**: realizza la logica per la memorizzazione di un messaggio, la ricezione di messaggi inviati offline e il blocco di utenti.
- **DAO**: Gestisce la consistenza relativa ai messaggi trasmessi e tiene traccia degli utenti bloccati.

server>UserdataService

- **userdataServiceInterface**: interfaccia HTTP per la ricezione delle richieste inoltrate dal chatHandler. Riceve richieste HTTP di tipo POST per funzioni di `storeData`, `userData` e `userDataById`.
- **requestController**: realizza la logica per la memorizzazione delle informazioni pubbliche di un utente (id, username e chiave pubblica) e per la ricerca di tali informazioni (controlla una operazione di ricerca per id ed una per username).
- **DAO**: Mantiene la persistenza relativamente ai dati pubblici degli utenti registrati al servizio. Espone i metodi `storeData` (memorizzazione id, username e chiave pubblica di un utente), `userData` (ricerca delle info di un utente tramite il suo username) e `userDataById` (ricerca delle info di un utente tramite il suo username).

5.3 Design pattern utilizzati

5.3.1 Pattern GRASP

Information Expert

La classe LoggedUser funge da information expert per molte delle classi contenute nel package Model del client: è infatti in grado di risalire, dato l'id di un utente, alla corrispondente chat e quindi ai messaggi scambiati e alle informazioni dell'utente considerato.

Allo stesso modo, la classe ConnectionsRegister del ChatHandler fa da information expert per i ConnectedUser, consentendo di risalire, dato l'identificativo dell'utente, ad informazioni quali il token di autenticazione o il descrittore della socket creata.

Creator

In molti casi abbiamo utilizzato il pattern Creator allo scopo di definire chiaramente la responsabilità di creare specifici oggetti nella nostra architettura.

Alcuni esempi di applicazione di questo pattern sono:

- la classe LoggedUser, responsabile della creazione delle Chat
- La classe Chat che ha la responsabilità di creare i nuovi Messaggi
- Il controller facade del client è responsabile della creazione dei controller presenti nel package, ma anche delle classi di utilità presenti nel package Servizi.

Controller

In vari punti della nostra architettura abbiamo fatto ricorso al pattern controller allo scopo di distinguere chiaramente le responsabilità di interfacciamento con l'utente o con altri servizi da quella di implementazione della business logic.

Il client, seguendo il pattern MVC, presenta un package che implementa questo pattern, il Controller, e che prende in carico il compito di elaborare gli input forniti dall'utente per realizzare le funzionalità offerte dalla user application.

Inoltre, tutti i microservizi implementano questo pattern per disaccoppiare le funzionalità di interfacciamento con gli altri componenti da quelle di gestione delle richieste.

5.3.2 Pattern Gof

Pattern Observer

Seguendo il pattern architetturale MVC, le classi della View e del Model implementano il pattern observer. Le modifiche di dati del Model effettuate tramite l'information expert LoggedUser sono notificate alle classi della View che aggiornano il loro stato interno e quindi le informazioni mostrate all'utente.

Nel nostro caso specifico la classe MainView che implementa l'interfaccia utente in cui sono presenti le conversazioni, è observer delle Chat contenute nel loggedUser, quando vi è una modifica in queste il loggedUser notifica la view che si aggiorna di conseguenza mostrando le nuove conversazioni o i nuovi messaggi. Allo stesso modo, la classe chatPage è Observer dell'insieme dei messaggi presenti nel Model. Le View si registrano come Observer dello stato del LoggedUser tramite una chiamata ad un metodo del Controller nel momento in cui sono istanziate, annullando la registrazione quando verranno derenderizzate.

La classe App, classe fondamentale per il framework React Native utilizzato, ha il compito fondamentale di istanziare le viste mostrate all'utente ed è inoltre Observer dello stato di connessione, memorizzata come variabile membro del loggedUser. In base allo stato di connessione corrente questa seleziona la vista da renderizzare e viene notificata quando lo stato cambia.

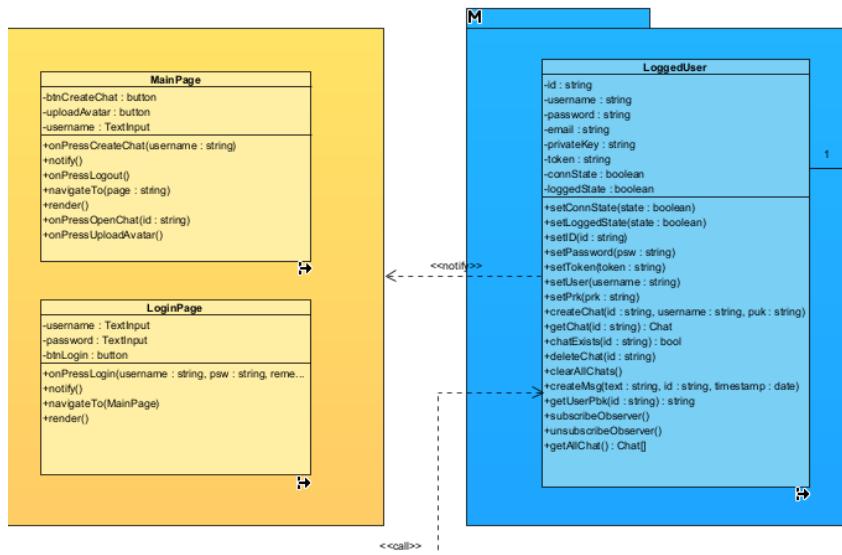


Figura 5.1: Esempio di observer

Pattern Proxy

Il componente ChatHandler implementa il pattern Proxy poiché riceve le richieste dai client e le ridirige agli appositi microservizi.

Pattern Facade

Nel package Controller del client abbiamo optato per l'inserimento di un Facade che consenta agli elementi della view di accedere a tutte le funzionalità offerte dai controller senza doverne conoscere i dettagli implementativi, consentendo un basso accoppiamento tra i due package.

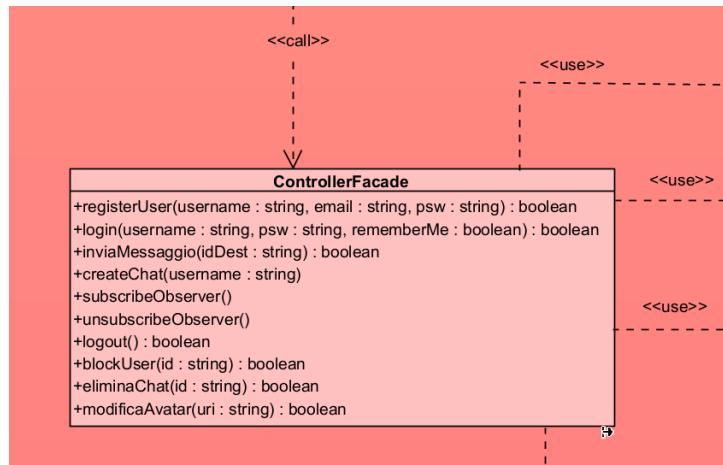


Figura 5.2: Esempio di Facade

Pattern Singleton

Il Singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza.

Nella nostra implementazione, il pattern è stato utilizzato, per esempio, per la classe LoggedUser: infatti, essa rappresenta un utente attualmente connesso e loggato, quindi, dato che solo un utente può essere connesso contemporaneamente, avremo solo una istanza di questa classe a run time.

5.4 Dinamica del Sistema

Riportiamo i diagrammi dinamici di dettaglio commentati che descrivono come sono implementate le funzionalità offerte dal sistema.

5.4.1 Avvio del client

Si riporta di seguito la descrizione dell'avvio dell'applicazione utente in corrispondenza dell'apertura dell'applicazione da parte dell'utente. Innanzitutto viene generata una istanza della classe App che instanzi il **controllerFacade**, il quale a sua volta nel suo costruttore, oltre che ad istanziare il loggedUser (oggetto che rappresenterà l'utente stesso all'interno del sistema client), si occuperà di invocare i costruttori delle classi del package *Services*, ovvero: cryptography (per adempiere alle funzionalità di crittografia), LocalStorage (una istanza rappresenta lo storage locale al client) e NetworkAccess. Il con-

trollerFacade poi userà i riferimenti a tali oggetti istanziati come parametri di ingresso nella invocazione dei costruttori dei controller, ovvero: ChatController, LoginController, RcvMsgController, Registration Controller e MessageController. L'app poi invocherà i metodi *SubscribeObserver* e *RememberMeLogin* del controllerFacade, da questo momento in poi le modifiche dello stato del *LoggedUser* saranno notificate agli elementi della view.

Il metodo RememberMeLogin del LoginController, a questo punto, richiederà al LocalStorage (di cui dispone un riferimento) username e password dell'utente (ricevendo in risposta innanzitutto un booleano che indica il successo o meno dell'operazione): Se i dati del login dell'utente sono già presenti nel LocalStorage (dunque l'utente era già loggato in precedenza e non ha effettuato il logout), allora viene effettuato direttamente il login (tramite i dati memorizzati) e sarà quindi istanziata e visualizzata la mainPage. In caso contrario viene istanziata e visualizzata una LoginPage.

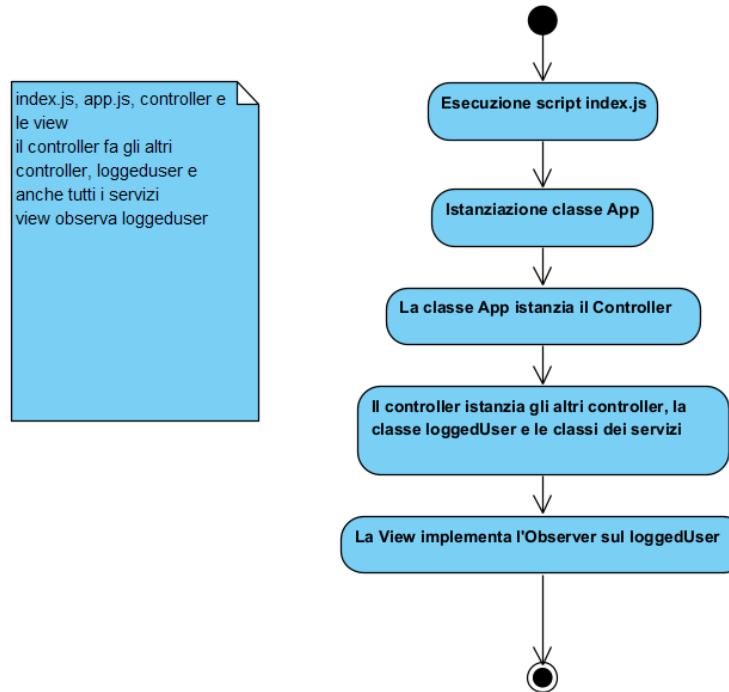


Figura 5.3: AvvioClient Activity Diagram

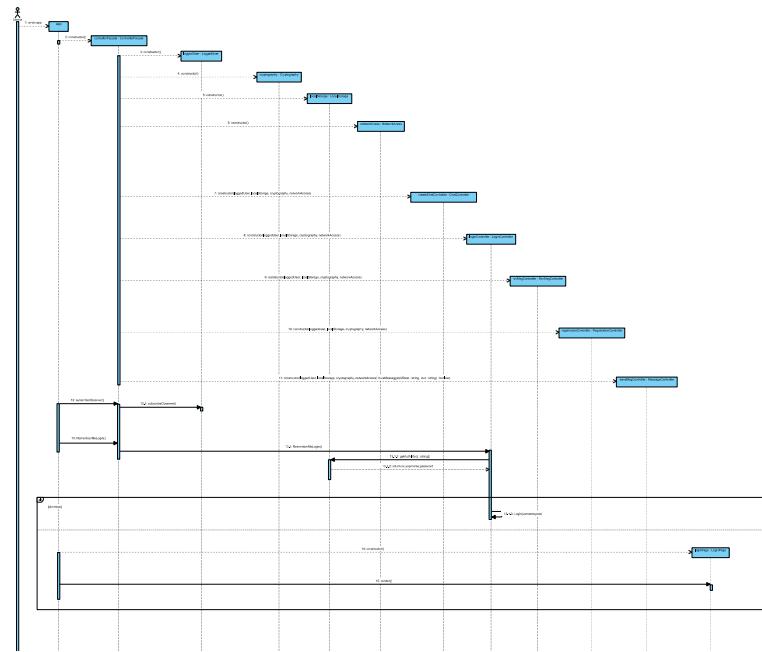


Figura 5.4: Avvio del Client SD di Dettaglio

5.4.2 Blocca utente

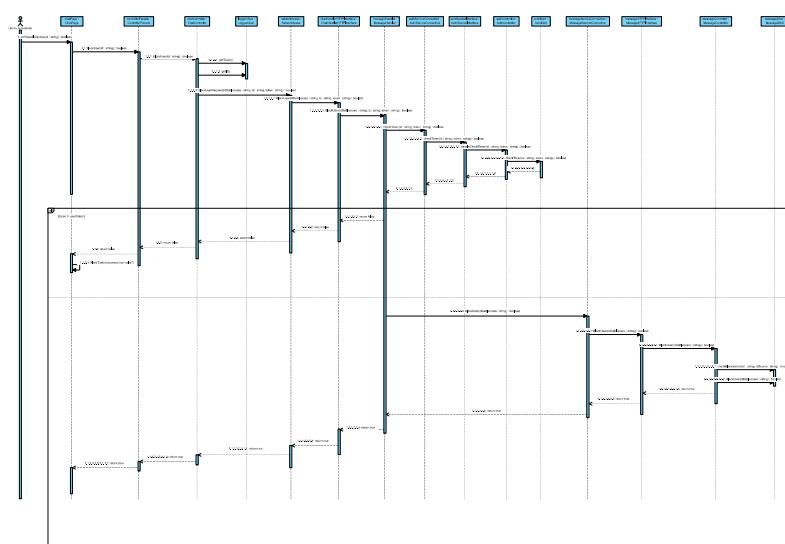


Figura 5.5: BloccaUtente SD di dettaglio

Il diagramma riportato mostra nel dettaglio l'interazione necessaria per bloccare un utente. In particolare, l'utente che si trova nella chat-page e che desidera bloccare un contatto con cui ha creato una chat, alla pressione del pulsante di impostazioni avrà la possibilità di selezionare da un menù a tendina se bloccare l'utente con cui ha avviato la chat. La pressione di tale bottone, porta la *blockUserRequest* dal *Network Access* del Client al *MessageHandler* del *ChatHandler* fino al servizio *MessageService* che serve la richiesta.

5.4.3 Crea chat

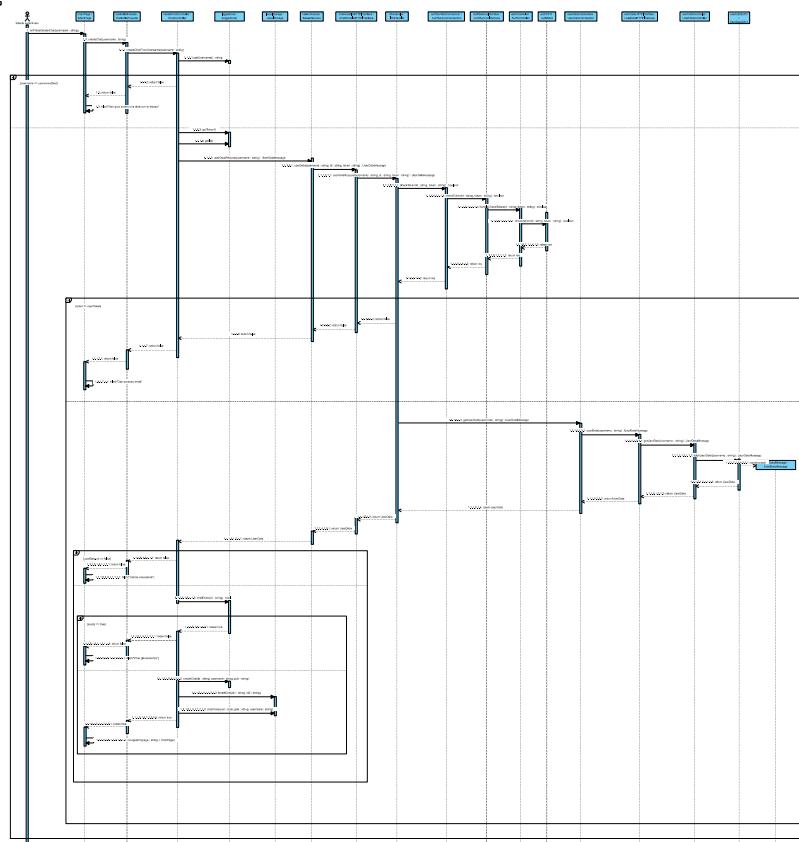


Figura 5.6: CreaChat SD di dettaglio

La creazione di una chat è una operazione che consiste nel reperimento, da parte della user application, delle informazioni pubbliche di un utente, necessarie per inviare a questo nuovi messaggi.

Un utente può innescare la creazione di una nuova chat scrivendo l'username del destinatario e premendo sul bottone dedicato alla creazione della chat. Ciò provoca la chiamata dell'handler del bottone, che si interfaccia direttamente col facade del Controller del Client. Eseguendo una semplice get sul LoggedUser, verifichiamo che l'username dell'utente selezionato come destinatario non sia lo stesso del mittente: nel caso il controllo fosse affermativo, viene mostrato sulla MainPage un alert di errore.

Successivamente, viene effettuata una UserData request, attraverso il chatHandler. Questa richiesta serve chiaramente per ottenere i dati dell'utente con cui bisogna creare la chat. La classe del controller del chatHandler dedita ad eseguire questo tipo di richieste è InfoHandler, che controlla inizialmente l'autenticità del token del mittente grazie all'authService. Fatta questa verifica, si interfaccia con l'userDataService sempre tramite una richiesta HTTP, che verrà gestita dall'UserDataController. Da questa elaborazione verrà quindi ritornato un userDataMessage, costruito dall'userDataDAO e contenente tutte le informazioni dell'utente destinatario.

Quindi, viene controllato dal CreateChatController il campo ok dell'userDataMessage: se è "false",

l'utente ricercato non esiste nell'app; altrimenti, il controller verifica se la chat che si vuole creare già esiste.

Se la chat non esiste, viene chiamato il metodo `createChat` di `LoggedUser`, che è `Information Expert` di tutte le sue chat attualmente attive, e infine i metodi `insertChat` e `insertUser` di `LocalStorage` per memorizzare in locale la chat appena creata.

La creazione di una chat può essere però anche eseguita all'istante di ricezione di un messaggio inviato da un utente con il quale non esiste una chat e di cui è quindi necessario ottenere le informazioni (quali `id` e `public key`) dal server.

5.4.4 Effettua login

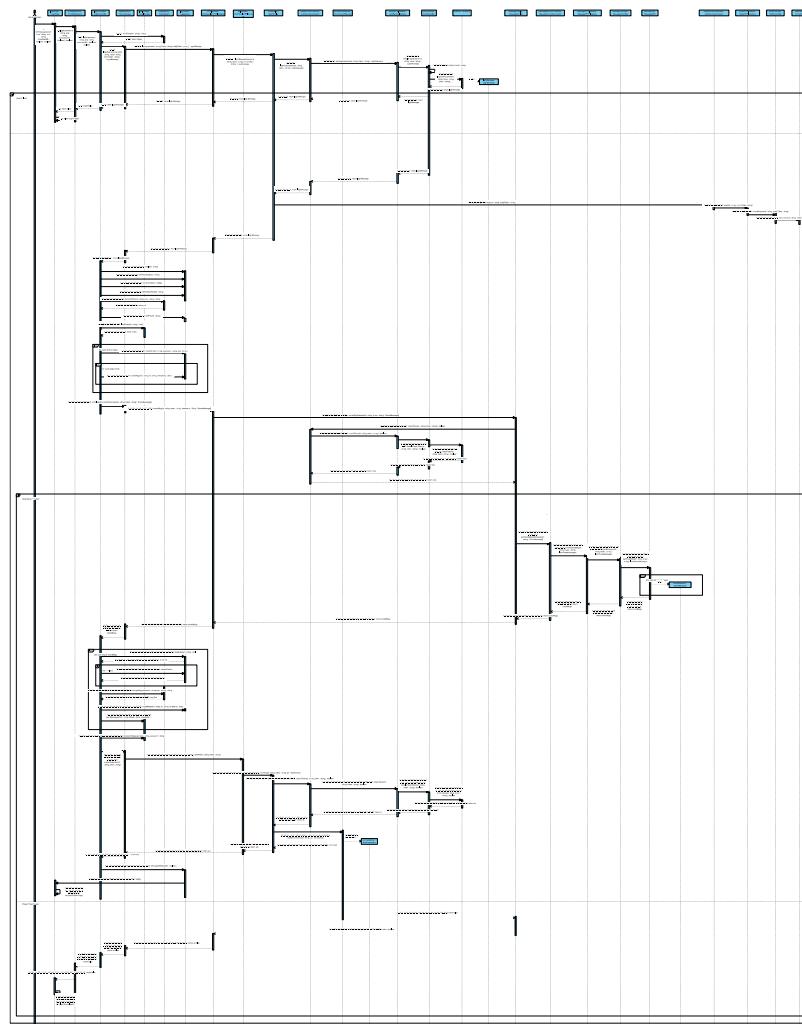


Figura 5.7: EffettuaLogin SD di dettaglio

In seguito all'inserimento delle proprie credenziali, l'utente preme l'apposito bottone di Login. Passando per il `ControllerFacade`, la richiesta viene inoltrata al `LoginController` il quale, dopo aver hashato la

password (ricordiamo che la password è utilizzata per criptare la private key dell'utente e quindi non è mai trasmessa al server) sfruttando le funzioni di crittografia, è pronto per trasmettere la richiesta all'*AuthService* (tramite l'*AuthServiceConnection* del *ChatHandler*) il quale restituisce un *LoginMessage*. A questo punto viene effettuata anche la registrazione al servizio di notifica richiedendo l'inserimento del *NotifyToken* generato dall'apposito servizio *NotifyService*. Una volta settati i parametri che caratterizzano *loggedUser*, viene richiesto al *LocalStorage* il caricamento delle chats salvate in locale. Per ognuna delle chat caricate dal database viene invocato il metodo di creazione di una chat del *LoggedUser*. Infine il *LoginController* richiede al *MessageService* tramite l'interfaccia HTTP del *ChatHandler* i messaggi non ancora consegnati. Una volta verificato opportunamente che il token sia effettivamente corrispondente all'utente di cui si richiedono le chat, viene consegnato all'utente un insieme di *StoredMessage*, ovvero messaggi non consegnati all'utente e ciascuno di questi viene aggiunto ad una chat pre-esistente o creata ad hoc. I messaggi ricevuti, una volta decifrati ed aggiunti alla chat corrispondente, vengono memorizzati in locale utilizzando un metodo esposto dalla classe *LocalStorage*. Avviene, infine, l'autenticazione della WebSocket (che sarà successivamente utilizzata dal server per consegnare i messaggi al client mentre questo è connesso). Il *LoginController*, tramite la *WSInterface* del *ChatHandler*, richiede l'autenticazione del canale di connessione, trasmettendo su questo il suo id ed il suo token. Al completamento dell'autenticazione del canale, il *ChatHandler* memorizza le informazioni relative alla socket in un nuovo oggetto *UserConnection*. Quando viene ricevuto l'ack di autenticazione del canale, lo stato del *LoggedUser* viene settato a "logged".

5.4.5 Effettua logout

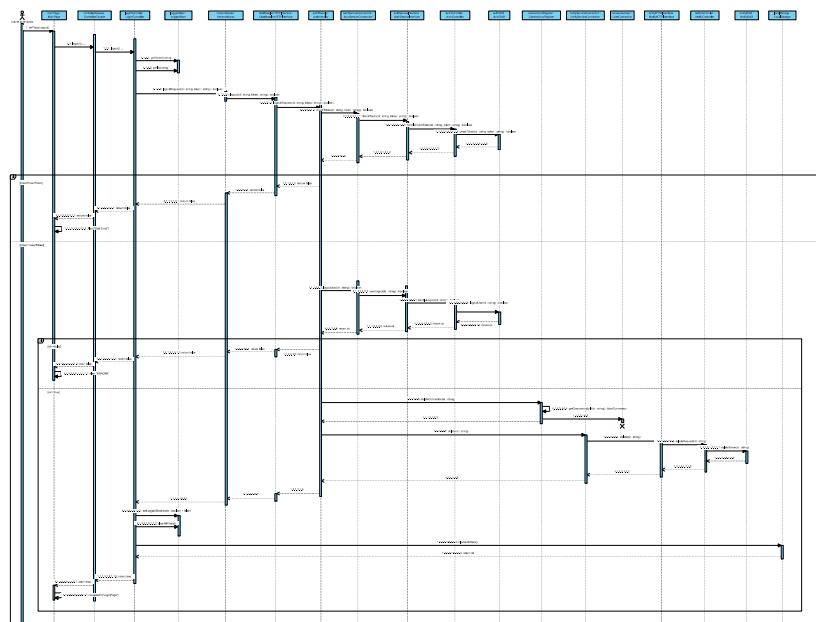


Figura 5.8: EffettuaLogout SD di dettaglio

Il sequence diagram di dettaglio in figura mostra le azioni richieste per consentire ad un utente di effettuare il logout. A partire dalla main-page, e con la pressione del pulsante di logout, viene trasmessa la richiesta passando allo stesso modo tra il *NetworkAccess* e l' *AuthHandler* del *ChatHandler* fino ad arrivare al servizio *AuthService* che serve la richiesta, invalidando il token di accesso fino a questo punto utilizzato. Se non si sono verificati errori l'*AuthHandler* si occupa di richiedere l'eliminazione della connessione con l'utente (non più loggato) dall'insieme delle connessioni mantenute in *ConnectionRegister*. Segue una richiesta di eliminazione dal gruppo di notifica del dispositivo dell'utente . Avviene poi il reset dell'oggetto *loggedUser* e la cancellazione degli oggetti chat, messaggi ed user caricati. Sono inoltre cancellate dal database locale le informazioni utilizzate per il login automatico (che viene disattivato da un esplicito logout dell'utente). Il reindirizzamento alla LoginPage termina la funzione.

5.4.6 Effettua registrazione

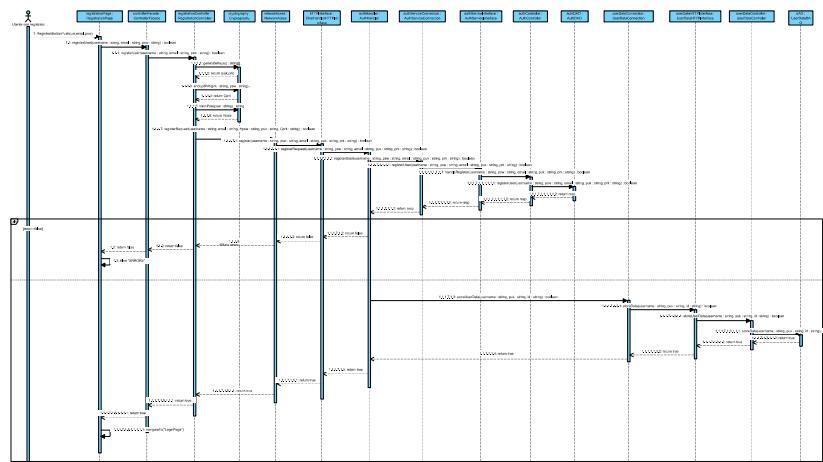


Figura 5.9: EffettuaRegistrazione SD di dettaglio

Dopo l'inserimento, da parte dell'utente, dei dati necessari alla registrazione e alla pressione del Bottone di Sign-Up, il *RegistrationController* si occupa, lato client, di richiamare le funzioni di Crittografia necessarie alla generazione delle chiavi (generate lato client per garantire la riservatezza), cifratura della chiave privata (che non sarà mai trasmessa al server in chiaro) e hashing della password (necessario per rendere impossibile il decriptaggio della chiave privata dell'utente).

E' quindi possibile trasmettere la richiesta all'*AuthService* che serve la richiesta memorizzando le informazioni e trasmettendo la mail necessaria all'attivazione dell'account. La mail di conferma contiene un link che, visitato tramite il browser, causa l'invocazione dell'metodo di autenticazione dell'utente del ChatHandler. Infine, se la procedura è andata a buon fine, le informazioni pubbliche (come username, id, e chiave pubblica) sono memorizzare informazioni nel database del *UserDataService*.

5.4.7 Elimina chat

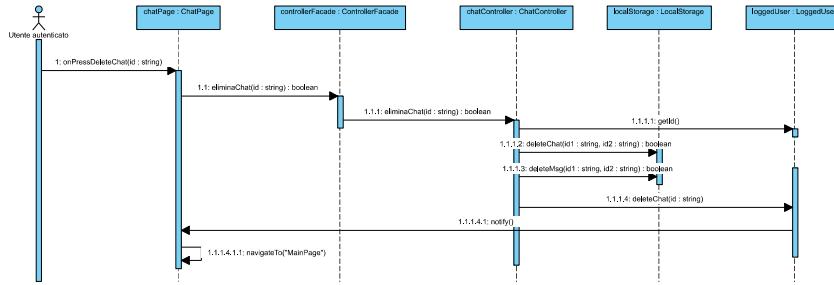


Figura 5.10: EliminaChat SD di dettaglio

Si riporta di seguito la dinamica dell'eliminazione di una chat da parte di un utente. In particolare a partire dalla chat page, alla pressione dell'apposito bottone di elimina chat (accessibile tramite un menu contestuale alla chatPage) una cascata di metodi porterà all'invocazione del metodo *EliminaChat* del ChatController, con in ingresso come unico parametro l'id dell'utente destinazione associato alla chat che l'utente sorgente desidera eliminare. Tale metodo dunque richiederà alla istanza di LoggedUser l'id dell'utente corrente, il quale servirà per effettuare una *deleteChat* e una *deleteMsg*) per l'eliminazione dei messaggi della chat sul LocalStorage. Successivamente verrà chiamato il metodo *DeleteChat(idUtenteChatDaEliminare)* che rimuoverà il riferimento alla chat da eliminare dal logged user. In seguito poi ad una notify di tale aggiornamento nel model verrà mostrata all'utente la view della pagina principale (MainPage) contenente tutte le chat tranne, ovviamente, quella appena eliminata.

5.4.8 Invia messaggio

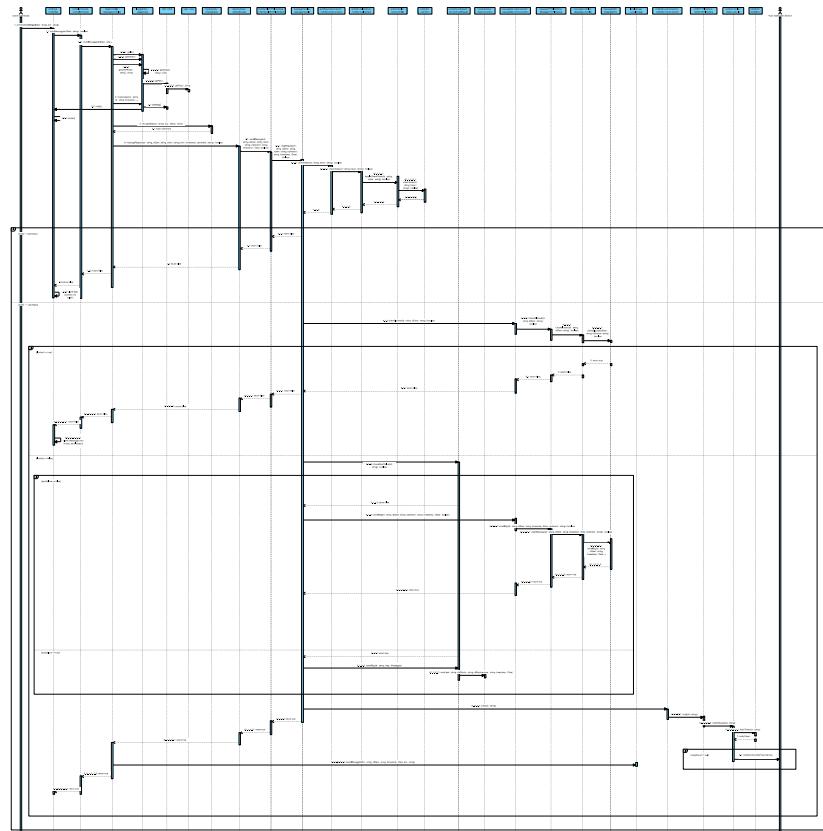


Figura 5.11: InviaMessaggio SD di dettaglio

Come riportato in figura, un utente invia un messaggio cliccando sul bottone apposito nella ChatPage, scatenando l'handler corrispondente che va a richiamare una funzione del ControllerFacade. Il metodo del Facade chiama *inviaMessaggio* del MessageController e da qui parte tutta la logica di business del caso d'uso.

Innanzitutto, si ricerca la chiave pubblica del destinatario, attraverso il metodo `getUserPbk` della classe LoggedUser. Questa andrà a chiamare il suo metodo `getPbk` che ritornerà proprio la stringa rappresentante la `publicKey` del destinatario. Dopo aver fatto ciò, verrà creato e aggiunto il messaggio alla lista dei messaggi, variabile membro della classe Chat, e, sempre da LoggedUser, verrà effettuata la `notify` sulla View, ovvero la ChatPage portando a rirenderizzarla. In questo modo verrà mostrato sullo schermo il messaggio appena inviato dall'utente.

Aggiornato il Model del Client, ora si procede al criptaggio del messaggio tramite la classe Cryptography. Da qui, viene il MsgController richiama il metodo di `messageRequest` del Newtwork Access, che effettua la richiesta HTTP verso il ChatHandler. Il ChatHandler prende quindi in carico il compito di consegnare il messaggio al client destinatario, che dovrà ricevere il messaggio ovviamente decriptato. La richiesta verrà gestita dal messageHandler in questo caso: inizialmente, viene verificata l'autenticità

del token del mittente sempre tramite l^authService; in caso di esito positivo, bisogna controllare se il mittente è stato bloccato dal destinatario. Tale controllo viene effettuato chiamando il metodo checkBlocked della classe di interfaccia con messageService, ovvero messageServiceConnection, ed effettuando una richiesta HTTP al microservizio.

In caso l^utente mittente non sia stato bloccato, il messageHandler controlla se l^utente destinatario è online chiamando il metodo checkUserOnline della classe connectionsRegister:

- se è offline, viene chiamata la funzione storeMsg di messageServiceConnection, che permetterà la store del messaggio nel database del microservizio MessageService. Il messaggio sarà consegnato al destinatario al suo prossimo accesso.
- se è online, viene chiamata la funzione sendMsg di ConnectionsRegister e conseguentemente la send della classe UserConnection, che consente l^uinvio vero e proprio del messaggio tramite la WebSocket in tempo reale.

Successivamente, viene utilizzato il servizio notifyService mediante la chiamata del metodo notify della classe notifyServiceConnection. Ciò viene fatto per mandare una notifica push al client destinatario, ovviamente nel caso esso abbia effettuato il login e quindi gli sia stato assegnato un notifyToken. Prima il notifyController esegue una fetchToken sul notifyDAO per ottenere il notifyToken del destinatario e, se questo è diverso da null, viene utilizzato il servizio esterno PushNotificationService per inviare la notifica sul dispositivo.

Infine, viene memorizzato nel database locale del Client mittente il messaggio inviato, attraverso una chiamata di insertMessage della classe LocalStorage.

5.4.9 Caricamento immagine profilo

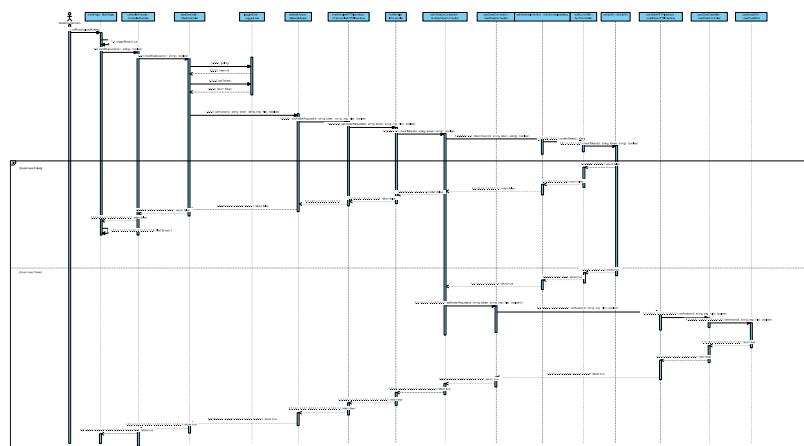


Figura 5.12: ModificaAvatarProfilo SD dettaglio

Dalla mainPage l'utente, premendo su di un apposito pulsante, può caricare una nuova immagine profilo, che sarà visualizzata dagli utenti con i quali esiste una chat. Per l'upload dell'immagine viene utilizzata una funzionalità built-in di react native, l'image picker, che consente all'utente di accedere alla sua galleria e selezionare un'immagine. La funzione restituisce l'URI dell'immagine selezionata. Fatto ciò viene invocato il metodo del chatController di uploadAvatar, che, dopo aver richiesto id e token, informazioni necessarie per validare la richiesta, invia i dati al chatHandler tramite i metodi di utilità offerti dalla classe NetworkAccess. Il chatHandler procede prima alla verifica della validità di id e token, sfruttando le funzionalità dell'authService. Procede poi, se i dati si rivelano validi, all'instradamento della richiesta allo userDataService, che tramite i metodi offerti dalla sua classe DAO memorizza in locale l'immagine, associandola all'id dell'utente.

Successive richieste dell'Avatar del utente saranno servite con la ritrasmissione dell'immagine appena memorizzata.

5.5 Deployment Diagram

Come passaggio finale della fase di progettazione, è utile descrivere quali sono gli eseguibili prodotti e distribuiti agli utenti finali, specificando i nodi fisici del sistema su cui essi andranno eseguiti nonché quelli su cui sono in esecuzione i servizi esterni di cui il sistema si serve. Tali informazioni sono state schematizzate nel diagramma di Deployment riportato in figura.

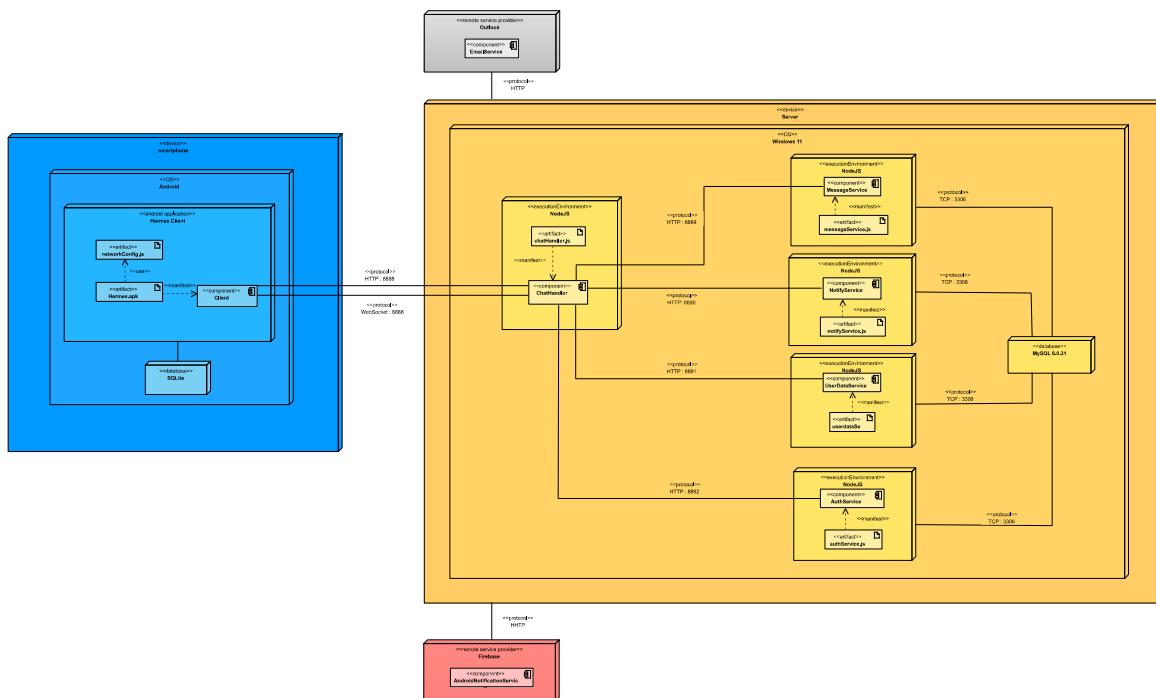


Figura 5.13: DeploymentDiagram

Abbiamo, quindi, un nodo su cui è deployato il client, ovvero uno smartphone con sistema operativo Android: qui troviamo l'app Android con un artefatto chiamato networkConfig, che contiene tutti i parametri di configurazione per la connessione alla rete, e poi l'artefatto che rappresenta l'apk, ovvero l'eseguibile presente sullo smartphone, che "manifesta" il componente Client. Inoltre, l'applicazione Android si basa sul database SQLite, utilizzato proprio per realizzare il database locale del Client attraverso React Native. Il collegamento col server può avvenire o con le Web Socket o tramite il protocollo HTTP.

Per quanto riguarda il Server, esso sarà deployato su un nodo con sistema operativo Windows 11, che nel progetto rappresenta un qualsiasi Personal Computer. Ci saranno diversi Execution Environment di NodeJS, tanti quanti sono i vari microservizi unitamente al Chat Handler. La connessione tra Chat Handler e i microservizi avviene mediante il protocollo HTTP, ognuna su un diverso porto. In particolare, visto che i microservizi sono deployati sullo stesso nodo, saranno tutti collegati tramite TCP allo stesso database, realizzato con MySQL, il che semplifica di gran lunga il deployment dell'applicazione.

I servizi esterni sono implementati su due remote service provider, Outlook e Firebase. Il primo, che comunica tramite HTTP col nodo Server, contiene il componente EmailService, mentre sul secondo è deployato il componente AndroidNotificationService, in comunicazione col Server sempre tramite HTTP.

5.6 Entity Relationship diagram

Un **diagramma entità relazione (ER)** è un tipo di **diagramma di flusso** che illustra come le "entità", quali persone, oggetti o concetti, si relazionano tra loro all'interno di un sistema. Ciascuna entità può essere caratterizzata da diversi attributi che ne definiscono le proprietà. Nel nostro caso il diagramma è stato realizzato per la rappresentazione delle entità in gioco modellando i database sottostanti che vengono sfruttati per la realizzazione delle varie funzionalità.

5.6.1 ER Client

Tramite lo schema ER del database lato client è possibile vedere come siano mantenuti i dati relativi alla autenticazione dell'utente utilizzatore (AuthData), così come tutte le chat (quindi con i rispettivi messaggi che le compongono) aventi un id univoco (che appunto è primary key della rispettiva table) e gli utenti associati alle singole chat (identificati da un id univoco, che appunto è primary key della rispettiva table).

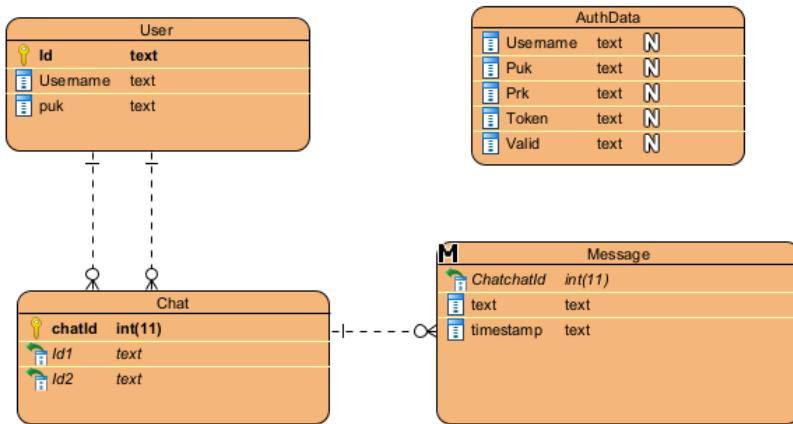


Figura 5.14: ER Client

5.6.2 ER Server

Per il lato server si è deciso di compattare il modello in un unico diagramma che includesse i dati trattati e gestiti da tutti i servizi. Tale diagramma E/R rappresenta la nostra scelta progettuale di deployare tutti i database dei microservizi su un unico database MySQL, come si può vedere dal diagramma di Deployment nel capitolo.

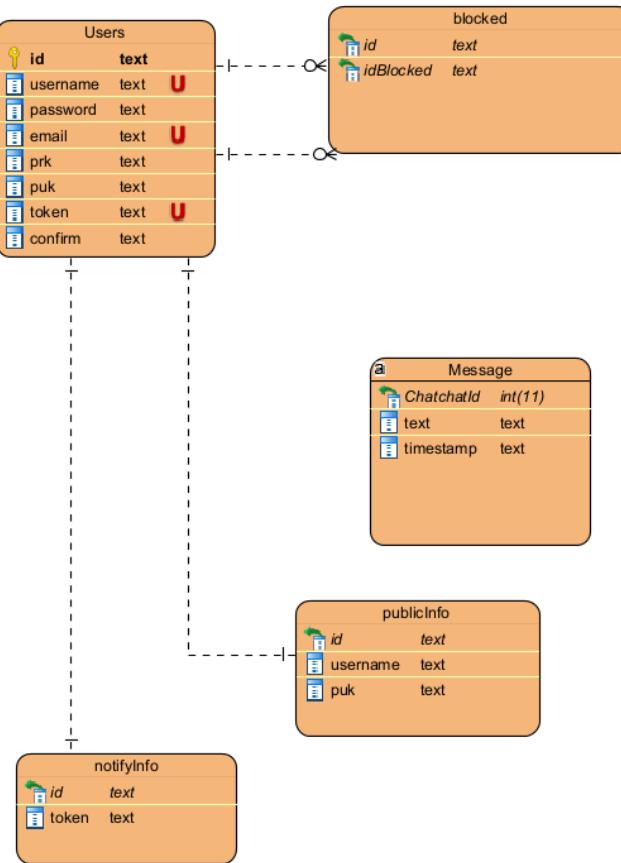


Figura 5.15: ER Server

5.7 Manuale di utilizzo

Presentiamo a questo punto un sintetico **Manuale di utilizzo** del software realizzato.

5.7.1 Configurazione e Installazione

L'installazione e la configurazione sono estremamente semplici. E' stato, infatti, generato un file `.apk` che è possibile installare su un sistema Android con estrema facilità. Tale file va caricato e installato tramite un qualunque File Manager.

5.7.2 Utilizzo User

Accesso e Registrazione

All'apertura dell'applicazione si verrà portati nella schermata in figura per effettuare il Login. Da tale schermata sarà possibile passare a quella di Registrazione cliccando sull'apposito bottone per la creazione di un nuovo Utente.

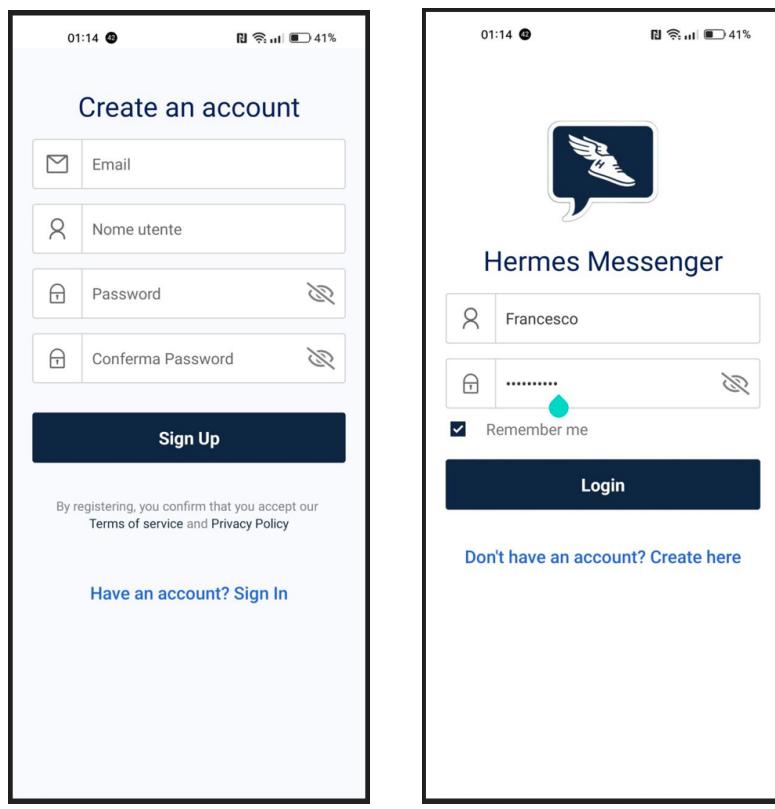


Figura 5.16

Main Page

Una volta effettuato l'accesso correttamente, si verrà reindirizzati alla Main Page dove sarà possibile visionare tutte le chat precedenti e i messaggi che sono stati inviati mentre si era offline.

Da tale schermata è, inoltre, possibile anche effettuare il Logout o cambiare l'immagine del profilo cliccando sull'apposito bottone in alto a sinistra che farà comparire le due opzioni. Qualora l'utente volesse ricercare una chat tra quelle esistenti è sufficiente inserire l'username ricercato all'interno della SearchBar; la Main Page mostrerà solo le chat con il destinatario specificato.

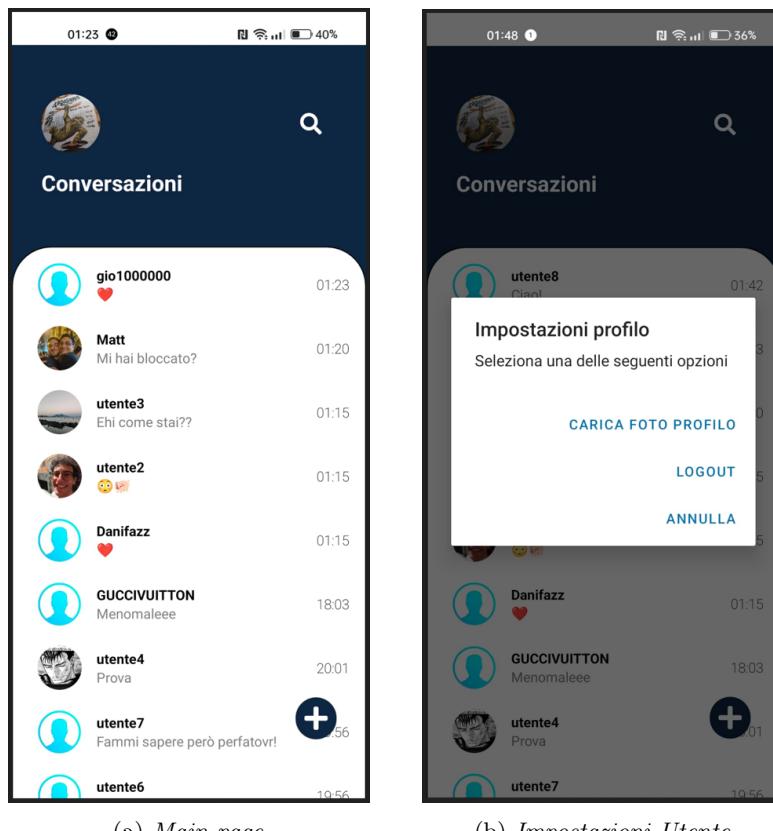


Figura 5.17

Chat Page

Per creare una nuova chat con un utente, è necessario cliccare sul bottone ("+" della main page, la pressione di tale bottone consente la scrittura all'interno field dell'username del destinatario. Completato l'inserimento verrà creata la nuova chat e l'utente sarà reindirizzato ad essa. Qualora si volesse accedere ad una chat precedentemente creata, è necessario cliccare su di essa a partire dalla Main Page.

All'interno della chat page cliccando sul bottone in alto a destra si aprirà un menù a tendina dal quale è possibile selezionare se bloccare l'utente con cui si ha la conversazione o eliminare la chat.

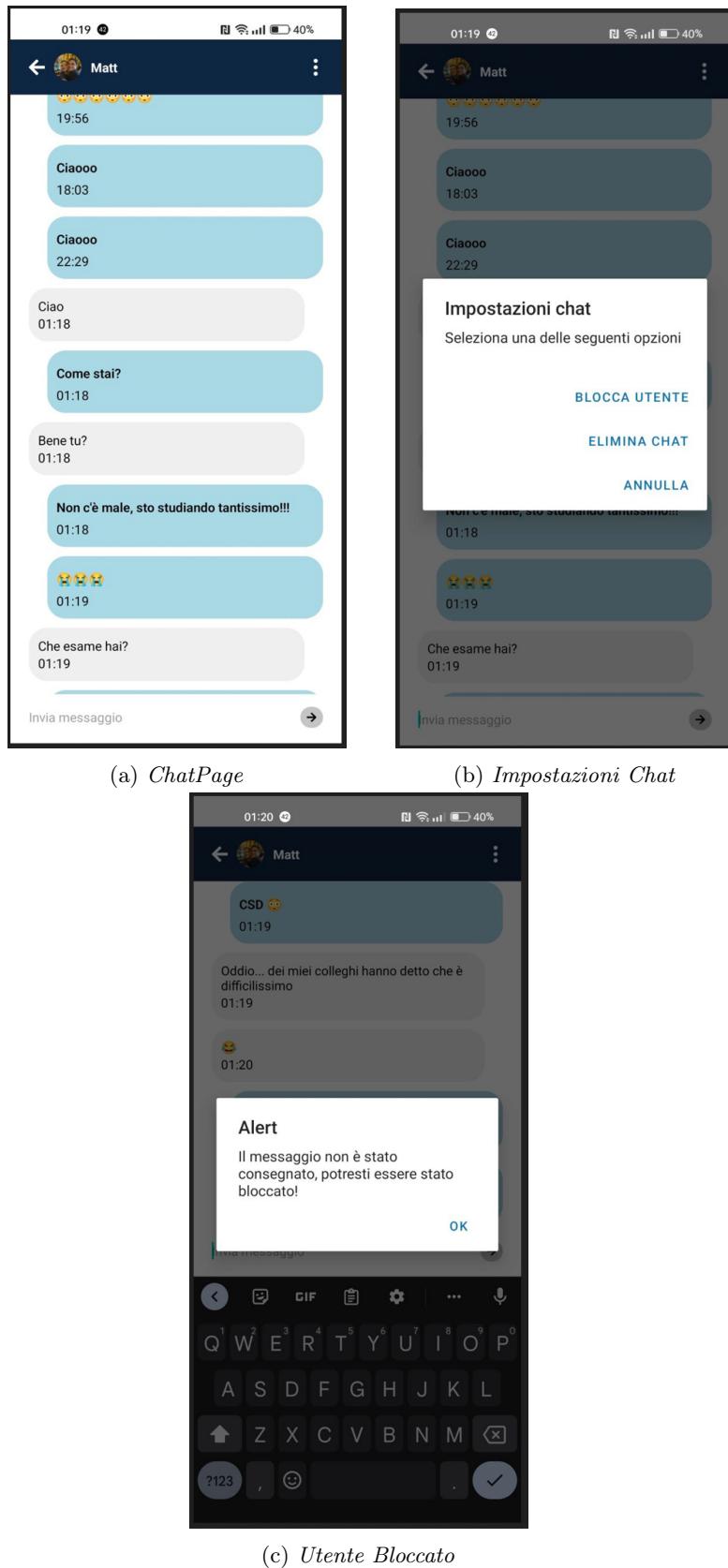


Figura 5.18

Capitolo 6

Testing

Il Testing ha l'obiettivo di trovare difetti e dimostrare che un sistema soddisfa i suoi requisiti funzionali e non-funzionali. Si tratta di un'attività fondamentale per il completamento dello sviluppo di un software. Per questa ragione, è stato eseguito con continuità durante il ciclo di sviluppo, a diversi livelli di dettaglio.

Prima di scrivere uno test effettivo, comprendiamo la struttura generale di un blocco di test:

- Un test è di solito scritto all'interno di un blocco di test dove per prima cosa occorre eseguire il rendering del componente che vogliamo testare.
- Si selezionano gli elementi con cui vogliamo interagire
- Si interagisce con tali elementi
- Si asserisce che i risultati siano come previsto.

La strategia di testing utilizzata per la validazione della nostra applicazione è stata quella di procedere prima con una fase di testing a livello component, testando in maniera singola i microservizi e quindi la correttezza delle funzionalità da loro offerte, e poi una fase di testing funzionale, per garantire la corretta interazione tra l'applicazione utente e i vari servizi lato server.

6.1 Testing dei servizi

L'indipendenza dei microservizi implementati li rende particolarmente adatti al testing e alla validazione autonoma di questi. Per la realizzazione del testing di unità dei componenti del sistema è stato utilizzato il framework JEST, che consente la definizione di un insieme di test case asserendone il risultato atteso e di effettuarli sul sistema in maniera automatica. Nel caso specifico dei microservizi implementati il framework consente di simulare richieste HTTP specificandone il formato ed il contenuto del body e di analizzare la risposta ottenuta dal sistema.

6.1.1 Testing microservizio AuthService

La test suite progettata per il servizio di AuthService prevede i seguenti casi di test:

1. Registrazione con dati validi.
2. Registrazione con email già utilizzata.
3. Registrazione con username già utilizzato.
4. Login con credenziali corrette.
5. Login con username o password non corretti.
6. Verifica dell'account.
7. Login prima della corretta verifica dell'account.
8. Verifica della validità del token corretto.
9. Verifica della validità di un token errato.
10. Tentativo di logout.

Il report prodotto da jest, relativo all'esecuzione di questi casi di test, mostra una elevata coverage. I rami non coperti sono in particolar modo associati a situazioni di errore di accesso al database, condizioni difficilmente simulabili in questo ambiente di test.

```
PASS  __tests__/_authService.test.js
Test di auth service
  ✓ Test di registrazione (249 ms)
  ✓ Test di login prima di aver verificato l'account (46 ms)
  ✓ Verifica dell'account (41 ms)
  ✓ Test di login correttamente eseguito (60 ms)
  ✓ Test di login con credenziali errate (35 ms)
  ✓ test di register con email già utilizzata (47 ms)
  ✓ test di register con username già utilizzato (47 ms)
  ✓ test checktoken con token non valido (69 ms)
  ✓ test checktoken con token valido (373 ms)
  ✓ test di logout (121 ms)

-----|-----|-----|-----|-----|-----|
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 85.86   | 58.33    | 93.33   | 85.86   | 15,23,47-48,65-66,81,86
DAO.js    | 80       | 62.5     | 100     | 80       | 15,23,47-48,65-66,81,86
HTTPInterface.js | 100     | 100     | 100     | 100     |
requestController.js | 83.87   | 50       | 85.71   | 83.87   | 34,46-62,71
-----|-----|-----|-----|-----|-----|
Test Suites: 1 passed, 1 total
Tests:      10 passed, 10 total
Snapshots:  0 total
Time:        2.994 s, estimated 13 s
Ran all test suites.
```

Figura 6.1: Jest testing microservizio AuthService

6.1.2 Testing microservizio MsgService

Per il message service abbiamo progettato i seguenti casi di test:

1. Memorizzazione di un messaggio.
2. Richiesta di messaggi memorizzati.
3. Richiesta di bloccaggio di un utente.
4. Richiesta di sbloccaggio di un utente.

La coverage ottenuta, come mostrato dal report, è molto elevata.

```
PASS  __test__/msgservice.test.js
Test di msgService
  ✓ Test store message (194 ms)
  ✓ Test di request stored msgs (40 ms)
  ✓ Test di block user (49 ms)
  ✓ test di unblock user (43 ms)

-----|-----|-----|-----|-----|-----|
File    | % Stmt | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 96.96 | 100 | 100 | 96.96 |
DAO.js | 93.33 | 100 | 100 | 93.33 | 23-24
HTTPInterface.js | 100 | 100 | 100 | 100 |
requestController.js | 100 | 100 | 100 | 100 |
-----|-----|-----|-----|-----|-----|
Test Suites: 1 passed, 1 total
Tests:        4 passed, 4 total
Snapshots:   0 total
Time:         1.686 s, estimated 2 s
Ran all test suites.
```

Figura 6.2: Jest testing microservizio MsgService

6.1.3 Testing microservizio UserDataService

I casi di test progettati per lo user data service sono i seguenti:

1. Memorizzazione di dati di un nuovo utente.
2. Richiesta di dati di un utente specificando il suo username.
3. Richiesta di dati di un utente specificando il suo id.
4. Richiesta dei dati di un utente il cui username non è registrato.
5. Richiesta di dati di un utente il cui id non è registrato.

```
PASS __test__/userDataService.test.js
Test di userDataService
  ✓ Store data (191 ms)
  ✓ User data by id (27 ms)
  ✓ User data by userName (22 ms)
  ✓ User data by userName non presente (19 ms)
  ✓ User data by id non presente (26 ms)

-----|-----|-----|-----|-----|-----|
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 98.11  | 100      | 100     | 98.11   | 
DAO.js    | 95.45  | 100      | 100     | 95.45   | 21
HTTPinterface.js | 100    | 100      | 100     | 100     |
requestController.js | 100    | 100      | 100     | 100     |

Test Suites: 1 passed, 1 total
Tests:      5 passed, 5 total
Snapshots:  0 total
Time:       1.512 s, estimated 2 s
Ran all test suites.
```

Figura 6.3: Jest testing microservizio UserDataService

6.1.4 Testing microservizio NotifyService

Per il notify service abbiamo progettato i seguenti casi di test:

1. Inserimento al gruppo di notifiche di un utente
2. Invio di notifica ad un utente nel gruppo notifiche
3. Invio di una notifica ad un utente non presente nel gruppo notifica
4. Rimozione di un utente dal gruppo notifica.

```
PASS __test__/notifyservice.test.js
Test di insert notify request
  ✓ Test store message (106 ms)
  ✓ Test di notify request (71 ms)
  ✓ Test di delete notify request (16 ms)
  ✓ test di notify ad un utente non presente (22 ms)

-----|-----|-----|-----|-----|-----|
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 83.33  | 75        | 90.9    | 83.33   | 
DAO.js    | 76.19  | 100      | 100     | 76.19   | 15,30-31,40,49
HTTPinterface.js | 100    | 100      | 100     | 100     |
requestController.js | 78.57  | 50        | 80      | 78.57   | 25,31-39

Test Suites: 1 passed, 1 total
Tests:      4 passed, 4 total
Snapshots:  0 total
Time:       1.978 s, estimated 22 s
Ran all test suites.
```

Figura 6.4: Jest testing microservizio NotifyService

6.1.5 Testing ChatHandler

Per il chat handler abbiamo poi progettato casi di test mirati a verificare il funzionamento delle funzionalità offerte e per verificare i meccanismi di autenticazione e di verifica dell'accesso offerti.

Il report di jest mostra un ottimo livello di copertura del codice nonostante la complessità del componente chathandler.

```
PASS __test__/chathandler.test.js
  Test del chat handler
    ✓ Test di registrazione (135 ms)
    ✓ Test di registrazione con formato errato (10 ms)
    ✓ Test di registrazione con username già utilizzato (20 ms)
    ✓ Test di registrazione con mail utilizzata (20 ms)
    ✓ Test di login con formato errato (23 ms)
    ✓ test di login con username errato (28 ms)
    ✓ test di login con psw errata (52 ms)
    ✓ test di login con dati corretti (39 ms)
    ✓ Test di attivazione di un account (31 ms)
    ✓ Richiesta di userdata con formato errato (9 ms)
    ✓ Richiesta di userdata con username errato (18 ms)
    ✓ Richiesta di userdata con username corretto (38 ms)
    ✓ Richiesta di userdata formato errato (6 ms)
    ✓ Richiesta di userdata by id con id errato (17 ms)
    ✓ Richiesta di userdata by id con id corretto (13 ms)
    ✓ Richiesta di stored message con formato errato (7 ms)
    ✓ Richiesta di stored message con token errato (5 ms)
    ✓ Richiesta di stored message con token corretto (17 ms)
    ✓ Richiesta di stored message id errato (6 ms)
    ✓ Richiesta di invio messaggio con token errato (6 ms)
    ✓ Richiesta di invio messaggio con token corretto (25 ms)
    ✓ Richiesta di invio messaggio con formato errato (8 ms)
    ✓ Richiesta di invio messaggio con id errato (14 ms)
    ✓ test di blocca utente con token non valido (5 ms)
    ✓ test di blocca utente con token valido (27 ms)
    ✓ test di blocca utente con id non valido (11 ms)
    ✓ Registrazione, login ed autenticazione del canale di comunicazione (85 ms)
    ✓ Invio di un messaggio ad un utente online (31 ms)
    ✓ test di logout con token errato (8 ms)
    ✓ test di logout (18 ms)

-----|-----|-----|-----|-----|-----|
File      | % Stmtts | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 84.07   | 79.01   | 89.83   | 84.37   |
controller | 87.12   | 84.61   | 92.3    | 87.12   |
authHandler.js | 86.04   | 79.31   | 100     | 86.04   | 23-24,51,57,71,78
controllerFacade.js | 87.5    | 100     | 84.61   | 87.5    | 25,55
infoHandler.js | 92.3    | 87.5    | 100     | 92.3    | 24
msgHandler.js | 86.2    | 89.28   | 100     | 86.2    | 36-37,54,61
interface | 96.66   | 100     | 85.71   | 100     |
ClientInterface.js | 96.66   | 100     | 85.71   | 100     |
services | 75.8    | 100     | 85.71   | 75.8    |
authServiceConnection.js | 70.58   | 100     | 80       | 70.58   | 56-73
msgServiceConnection.js | 82.35   | 100     | 100     | 82.35   | 28,50,71
notifyServiceConnection.js | 76.47   | 100     | 100     | 76.47   | 41-46
userDataConnection.js | 72.72   | 100     | 66.66   | 72.72   | 8-22
users | 78.78   | 50       | 91.66   | 78.78   |
connectedUser.js | 88.88   | 33.33   | 100     | 88.88   | 11
connectionsRegister.js | 75       | 54.54   | 87.5    | 75       | 21-24,33,50
-----|-----|-----|-----|-----|-----|
Test Suites: 1 passed, 1 total
Tests:      30 passed, 30 total
Snapshots:  0 total
Time:        1.991 s, estimated 7 s
Ran all test suites.
```

Figura 6.5: Jest testing ChatHandler

6.2 Testing funzionale

6.2.1 Testing operazione di login

Tentativo di login con account non registrato

Condizioni di validità dell'input:

1. L'utente che vuole effettuare il login deve utilizzare un username ed una password precedentemente utilizzati durante la fase di registrazione (ovvero deve essere registrato al servizio)

Precondizioni:

1. L'utente deve essere registrato.

TC	Descrizione	Input	Output
TC1	username di un account non registrato	username: MarioRossi18	error: "User does not exist"

Tentativo di login con account non verificato

Condizioni di validità dell'input:

1. L'utente che vuole effettuare il login deve utilizzare un account che sia stato verificato cliccando sul link di registrazione ricevuto via mail.

Precondizioni:

1. L'utente deve essere registrato.

TC	Descrizione	Input	Output
TC2	username di un account non verificato	username: GiulioVerdi10	error: "Wrong password or account not confirmed"

Tentativo di login con password errata

Condizioni di validità dell'input:

1. L'utente che vuole effettuare il login deve utilizzare la password di accesso valida corrispondente all'username dell'account registrato.

Precondizioni:

1. L'utente deve essere registrato.

TC	Descrizione	Input	Output
TC3	password di accesso non corretta o non corrispondente allo username	username: MariaRomano36, password: Romanmary00	error: "Wrong password or account not confirmed"

Tentativo di login con dati esatti

Condizioni di validità dell'input:

1. L'utente che vuole effettuare il login deve utilizzare le credenziali associate ad un utente registrato e che sia stato verificato.

Precondizioni:

1. L'utente deve essere registrato.

TC	Descrizione	Input	Output
TC4	username e password inseriti correttamente e corrispondenti a utente registrato	username: GiulioVerdi10, password: Giulioverdi1	ok:true

6.2.2 Testing operazione di registrazione**Tentativo di registrazione con password non valida**

Condizioni di validità dell'input:

1. L'utente che vuole effettuare la registrazione deve utilizzare una password di almeno 9 caratteri, con almeno una lettera maiuscola e un numero.

TC	Descrizione	Input	Output
TC5	password senza lettera maiuscola	password: pippopippozzo1	error: "Password non valida"

Tentativo di registrazione con username non valido

Condizioni di validità dell'input:

1. L'utente che vuole effettuare la registrazione deve utilizzare un username senza caratteri speciali ma solo lettere minuscole, maiuscole e numeri.

TC	Descrizione	Input	Output
TC6	username contenente un carattere non valido	username: pluto@	error: "Username non valido"

Tentativo di registrazione con username già utilizzato

Condizioni di validità dell'input:

1. L'utente che vuole effettuare la registrazione deve utilizzare un username che non sia stato già precedentemente associato ad un altro utente.

TC	Descrizione	Input	Output
TC7	username già utilizzato	username: pluto	error: "Username o email già utilizzata!"

Tentativo di registrazione con email già utilizzata

Condizioni di validità dell'input:

1. L'utente che vuole effettuare la registrazione deve utilizzare una email che non sia stata già precedentemente associata ad un altro utente.

TC	Descrizione	Input	Output
TC8	email già utilizzata	email: pluto@topolino.it	error: "Username o email già utilizzata!"

6.2.3 Testing operazione di invio messaggio**Tentativo di invio di un messaggio ad utente non registrato**

Condizioni di validità dell'input:

1. L'utente che vuole inviare un messaggio deve inserire l'username di un utente registrato al servizio

TC	Descrizione	Input	Output
TC9	il destinatario non corrisponde a un utente registrato	username: nonno24	error: "User does not exist!"

Tentativo di invio di un messaggio ad utente offline

Precondizioni:

1. L'utente destinatario deve essere registrato al servizio ma offline all'atto dell'invio del messaggio.

TC	Descrizione	Input	Output
TC10	invio messaggio in una chatPage con un utente non attualmente connesso al servizio	username utente destinazione: GiuliaMoccia99, text messaggio: "Ciao Giulia, come va?"	Invio notifica ad utente destinazione (HTTP notify request a NotifyService)

Tentativo di invio di un messaggio da un utente bloccato

Precondizioni:

1. L'utente antipatico1 è stato bloccato da user1

TC	Descrizione	Input	Output
TC11	antipatico1 invia un messaggio ad user1	messaggio [usr:antipatico1, dest:user1]: "Ciao perchè non mi rispondi più?"	error:"Messaggio non consegnato, potrei essere stato bloccato"

6.2.4 Testing servizio di notifica

Verifica di consegna di una notifica ad app aperta

Precondizioni:

1. L'utente ha abilitato la ricezione delle notifiche e l'utente user1 ha l'app aperta

TC	Descrizione	Input	Risultato Atteso
TC12	Alla ricezione di un messaggio non arriva la notifica push	messaggio [dest:user1]	Ricezione del messaggio ma non della notifica push

Verifica di consegna di una notifica ad app in background

Precondizioni:

1. L'utente ha abilitato la ricezione delle notifiche e l'utente user1 ha l'app in background

TC	Descrizione	Input	Risultato Atteso
TC13	Alla ricezione di un messaggio arriva una notifica push	messaggio [dest:user1]	Ricezione di una notifica push secondo le preferenze impostate dall'utente

Verifica di consegna di una notifica ad app chiusa

Precondizioni:

1. L'utente ha abilitato la ricezione delle notifiche e l'utente user1 ha l'app chiusa

TC	Descrizione	Input	Risultato Atteso
TC14	Alla ricezione di un messaggio arriva una notifica push	messaggio [dest:user1]	Ricezione di una notifica push secondo le preferenze impostate dall'utente

6.2.5 Testing della funzionalità delle immagini profilo

Caricamento immagine profilo

TC	Descrizione	Input	Risultato Atteso
TC15	L'utente carica una nuova immagine di profilo selezionandola dalla galleria	immagine	L'immagine è caricata nel server e sostituita alla precedente immagine profilo

6.2.6 Consegna immagine profilo

TC	Descrizione	Input	Risultato Atteso
TC16	L'utente apre l'applicazione		Riceve l'immagine di profilo dei contatti con i quali condivide una chat.

6.3 Video demo

Un video dimostrativo dell'applicazione realizzata e delle sue funzionalità può essere trovato al seguente link YouTube: <https://www.youtube.com/watch?v=vwgRXt8sa4k>

