

# Università degli Studi di Napoli Federico II



## Dipartimento di Ingegneria Elettrica edelle Tecnologie dell'Informazione

*Scuola Politecnica e delle Scienze di Base*

*Corso di Laurea Magistrale in Ingegneria Informatica*

## Documentazione

## Elaborato Software Architecture Design

### **Studenti:**

Danilo Romano (M63001542)

Mario Tranchese (M63001533)

Alfonso D'avino (M63001493)

Valerio Maietta (M63001407)

### **Docenti:**

Prof.ssa Anna Rita Fasolino

## Sommario

<b>Capitolo 1: Processo di sviluppo e strumenti utilizzati .....</b>	<b>4</b>
1.1 Processo di sviluppo .....	4
1.2 Approfondimento: Storie Utente .....	8
1.3 Strumenti utilizzati per la documentazione .....	8
1.3.1 Tecnologie di supporto .....	10
1.4 Mockup .....	12
<b>Capitolo 2: Analisi del dominio .....</b>	<b>14</b>
Analisi preliminare .....	14
2.1 Specifica dei requisiti .....	14
2.1.1 Storie utente .....	14
2.1.2 Requisiti funzionali .....	15
2.1.3 Requisiti non funzionali .....	16
2.2 Scenari di funzionamento .....	17
2.2.1 Scenario Uno - Registrazione .....	17
2.2.2 Scenario Due - Login .....	17
2.2.3 Scenario Tre – Invio di un messaggio ad un altro utente .....	18
2.2.4 Scenario Quattro – Ricezione di un messaggio da un altro utente .....	18
2.3 Product Backlog .....	19
2.4 Modellazione Casi d'uso .....	20
2.4.1 Diagramma dei Casi d'uso .....	21
2.5 Diagrammi di Attività .....	22
2.5.1 <i>Registrazione</i> .....	22
2.5.2 <i>Login</i> .....	23
2.5.3 <i>Invio/Ricezione messaggio</i> .....	23
2.5.4 <i>Ricerca Chat</i> .....	24
2.5.5 <i>Logout</i> .....	24
2.6 Diagramma di Contesto .....	25
2.7 Diagrammi di Sequenza .....	26
2.7.1 <i>Registrazione</i> .....	26

2.7.2 Login .....	27
2.7.3 Ricerca Chat .....	28
2.7.4 Invio/Ricezione Messaggio .....	29
2.7.5 Logout .....	29
<b>Capitolo 3: Architettura e progettazione .....</b>	<b>30</b>
3.1 Tecnologie usate per l'implementazione della web app .....	30
3.1.1 Implementazione backend: NodeJS .....	30
3.1.2 Implementazione frontend: React.....	31
3.1.2 Messaggistica in tempo reale: Socket.IO .....	34
3.2 Architettura a microservizi .....	34
3.2.1 Utilizzo di Docker per la containerizzazione dei microservizi.....	36
3.3 Pattern MVC .....	36
3.4 Diagrammi .....	39
3.4.1 Diagramma dei componenti .....	39
3.4.2 Diagrammi di Sequenza Raffinati.....	39
3.4.3 Diagramma dei package .....	45
3.4.5 Diagramma delle classi .....	49
3.4.6 Diagramma di deployment .....	55
<b>Capitolo 4: Manuale di Installazione .....</b>	<b>56</b>
4.1 Installazione .....	56
4.2 Installazione alternativa con Docker: .....	57
<b>Capitolo 5: Testing e Sicurezza .....</b>	<b>58</b>
5.1 Testing e2e tramite Cypress .....	58
5.1.1 Come funziona Cypress? .....	58
5.1.2 Implementazione dei casi di test per ogni microservizio .....	59
5.2 Sicurezza tramite JWT e bcrypt .....	68
5.2.1 Gestione sicura dell'autenticazione e dell'autorizzazione: utilizzo di JWT .....	68
5.2.2 Hashing della password tramite Bcrypt.....	69

# Capitolo 1: Processo di sviluppo e strumenti utilizzati

## Avvio del progetto

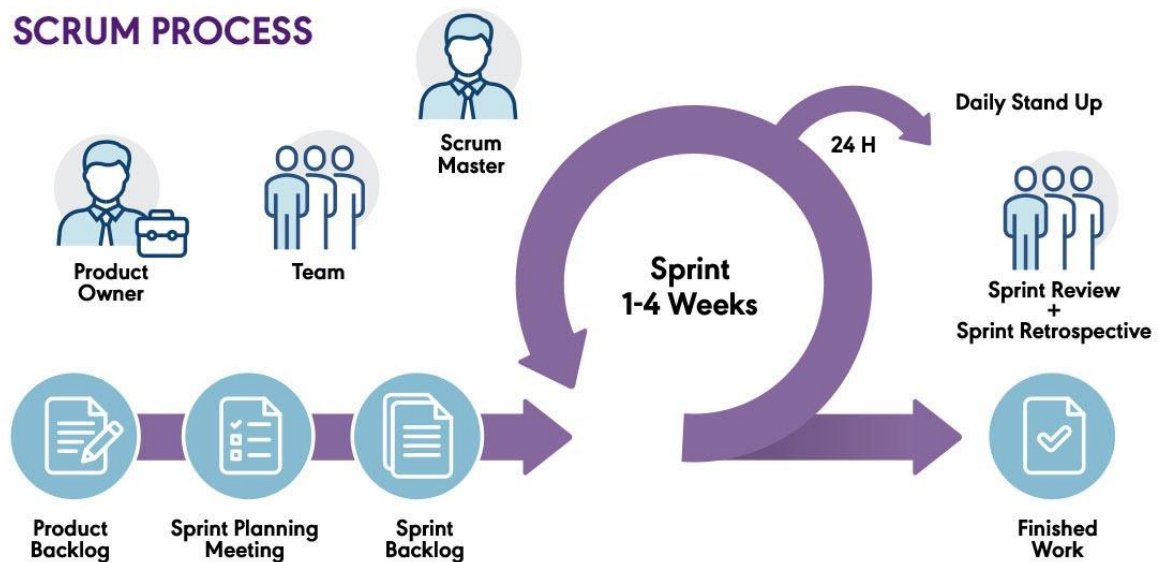
L'idea alla base di CrossChat è quella di creare una piattaforma di messaggistica innovativa e versatile, specificamente progettata per facilitare la comunicazione tra i membri di un team all'interno di un'azienda. CrossChat propone un approccio nuovo e dinamico alle interazioni e alla collaborazione online, offrendo un ambiente di chat intuitivo e funzionale che migliora la produttività e la condivisione delle informazioni tra i colleghi.

### 1.1 Processo di sviluppo

Per la realizzazione del progetto presentato, abbiamo deciso di lavorare mediante l'uso del processo di sviluppo Agile, grazie anche all'opportunità avuta durante il corso di approcciarci a questo nuovo *modus operandi* che ci ha permesso di gestire in modo molto efficiente il nostro lavoro. L'approccio Agile è di tipo iterativo e il suo obiettivo è, infatti, quello di consentire a un team di sviluppo di riuscire ad organizzare le proprie attività in modo più veloce ed efficiente rispetto ad altri approcci classici (es. waterfall). In questo modo, piuttosto che avere lunghe fasi di sviluppo totalmente sequenziali l'una all'altra, l'approccio Agile ci ha permesso di lavorare mediante iterazioni (o sprint), al termine delle quali vengono realizzati degli "incrementi" del prodotto finale in modo tale da aggiungere nuove funzionalità che abbiano valore agli occhi del cliente finale.

Abbiamo iniziato a lavorare inizialmente secondo un approccio classico basato sull'analisi dei requisiti funzionali e non. Abbiamo virato su un approccio **Agile**, in particolare abbiamo

suddiviso il lavoro mediante metodologie **SCRUM**, ed abbiamo implementato una product backlog con le storie utente che andremo poi a sviluppare in codice.



*Figura 1: Scrum process*

### Cos'è SCRUM?

**SCRUM** è il metodo Agile più diffuso, particolarmente indicato per progetti complessi ed innovativi. Si tratta di un framework che indica l'insieme delle pratiche da effettuare per dividere al meglio il processo di gestione di un progetto. In particolare, tale processo viene diviso in sprint per permettere di dare l'adeguata attenzione alle esigenze degli stakeholders gestendo parallelamente anche lo sviluppo vero e proprio del prodotto. Questi sprint, in genere, hanno breve durata, tipicamente di circa 2-4 settimane.

## Artefatti

Il processo di sviluppo porta alla realizzazione di una serie di “arteфatti”, tra cui ritroviamo:

- **Product Backlog:** rappresenta una vera e propria lista delle attività ordinate per priorità e necessarie per la realizzazione del progetto. Questa lista viene stilata da una figura specifica, che è il Product Owner, il quale ha il compito di capire come organizzare al meglio il lavoro, scinderlo in attività dalla mole più contenuta e capire come schedulare tali attività sulla base del tempo a disposizione e delle competenze del team con cui lavora;
- **User Story:** permette di descrivere in modo semplice, informale ed intuitivo quelli che sono i requisiti che si richiedono.

## Coordinazione e divisione del lavoro

Con cadenza periodica, abbiamo organizzato ognuno di questi eventi col fine di coordinare al meglio il nostro lavoro:

- **Iteration Planning:** è una riunione della durata di circa 1 ora svolta all’inizio di ogni iterazione. Con questa riunione si ha l’importante compito di stabilire come migliorare il proprio prodotto e, soprattutto, di attribuire la priorità alle varie storie utente e di schedulare i vari compiti da svolgere. Questa riunione è anche tipicamente nota col nome di Sprint Planning.
- **Coordination Meeting:** questo tipo di riunione, invece, nasce per essere di breve durata, ovvero all’incirca 15 minuti, in modo tale da essere effettuata giornalmente col fine di sincronizzare il lavoro svolto dai vari membri del team e di aiutarsi insieme

per superare eventuali ostacoli riscontrati. Essa rappresenta a tutti gli effetti quella che si può definire come “riunione di allineamento”. Questa riunione è tipicamente nota col nome di Daily Scrum;

- **Sprint Review:** questa riunione è generalmente di tipo informale e ha una durata che può andare dai 30 ai 60 minuti; viene svolta alla fine di un’iterazione e in essa si mostra agli stakeholders il risultato raggiunto, magari facendo uso di una demo col fine di raccogliere i primi feedback;
- **Sprint Retrospective:** è una riunione della durata di circa 1 ora, anch’essa svolta alla fine di un’iterazione e viene svolta successivamente alla Sprint Review così i membri del team hanno l’opportunità di discutere su cosa sia andato bene o cosa sia andato male dopo essersi interfacciati con gli stakeholders. L’obiettivo è quello di migliorare sempre di più la soddisfazione dei clienti e di supportare, al tempo stesso, un buon rapporto lavorativo all’interno del team.

Per il nostro progetto, a causa dei vari impegni personali e universitari di ogni membro del nostro gruppo, non è stato possibile effettuare riunioni ogni giorno, eppure, nonostante questo impedimento, abbiamo compiuto ogni sforzo per essere proattivi ed emulare al meglio il ciclo di sviluppo appena descritto.

## 1.2 Approfondimento: Storie Utente

Le user stories sono un potente mezzo tramite il quale si cerca di “estrapolare” le esigenze degli utenti finali. La grande potenza di questo tipo di strumento è data dalla estrema semplicità e immediatezza con cui esprimono le caratteristiche, le funzioni e i requisiti che si vogliono realizzare.

Nello specifico, ogni storia descrive in modo semplice:

- **Che funzionalità** si sta trattando;
- **A chi è rivolta** quest’ultima;
- **Per quale motivo** è necessario che ci sia.

Ovviamente, il punto di vista della descrizione è quello di chi sta richiedendo quella specifica funzionalità. Il motivo principale delle storie utente è soprattutto quello di permettere ai membri del team di effettuare una stima del lavoro richiesto per la realizzazione del requisito in esame. Tale stima viene quantificata mediante un punteggio attribuito alla storia.

## 1.3 Strumenti utilizzati per la documentazione

### Visual Studio Code

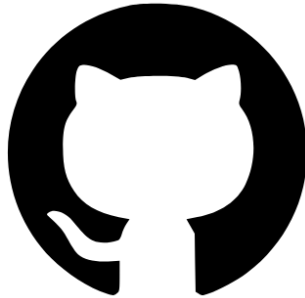
**Visual Studio** fornisce agli sviluppatori uno strumento per lavorare con Java e applicazioni Web.





## **GitHub**

**GitHub** è molto noto per i servizi di hosting per progetti software. Abbiamo usato un repository condiviso raggiungibile con questo link: [ValerioMaietta/ProgettoSAD2024 \(github.com\)](https://github.com/ValerioMaietta/ProgettoSAD2024)



## **Microsoft Teams**

Abbiamo usato questa piattaforma per tenere una traccia condivisa di tutte le riunioni e di tutto il materiale utilizzato. Inoltre, per gestire le attività da fare abbiamo usato le schede di Attività messe a disposizione dalla piattaforma stessa.



## **Draw.io**

Abbiamo usato questo software per la realizzazione di alcuni tipi di diagrammi; come, per esempio, i diagrammi di attività e delle componenti.



### 1.3.1 Tecnologie di supporto

**React:** Un framework JavaScript per la creazione di interfacce utente basate sul concetto di componenti riutilizzabili.



**Tailwind:** Framework CSS per la personalizzazione grafica dell'interfaccia utente



**Socket IO:** Libreria JavaScript per la comunicazione real-time per la parte di messaggistica istantanea



**Cypress:** uno strumento potente per l'automazione dei test end-to-end (E2E) di applicazioni web



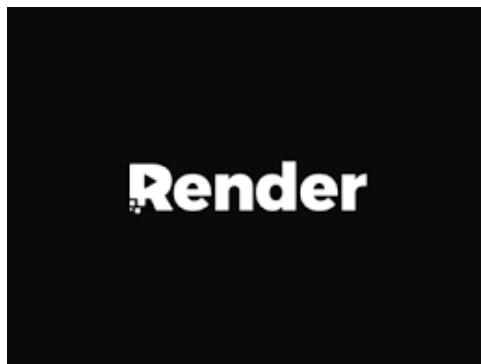
**MongoDB:** per l'archiviazione dei dati della web app



**NodeJS:** Il tutto verrà orchestrato da un server sviluppato con l'ausilio di NodeJS

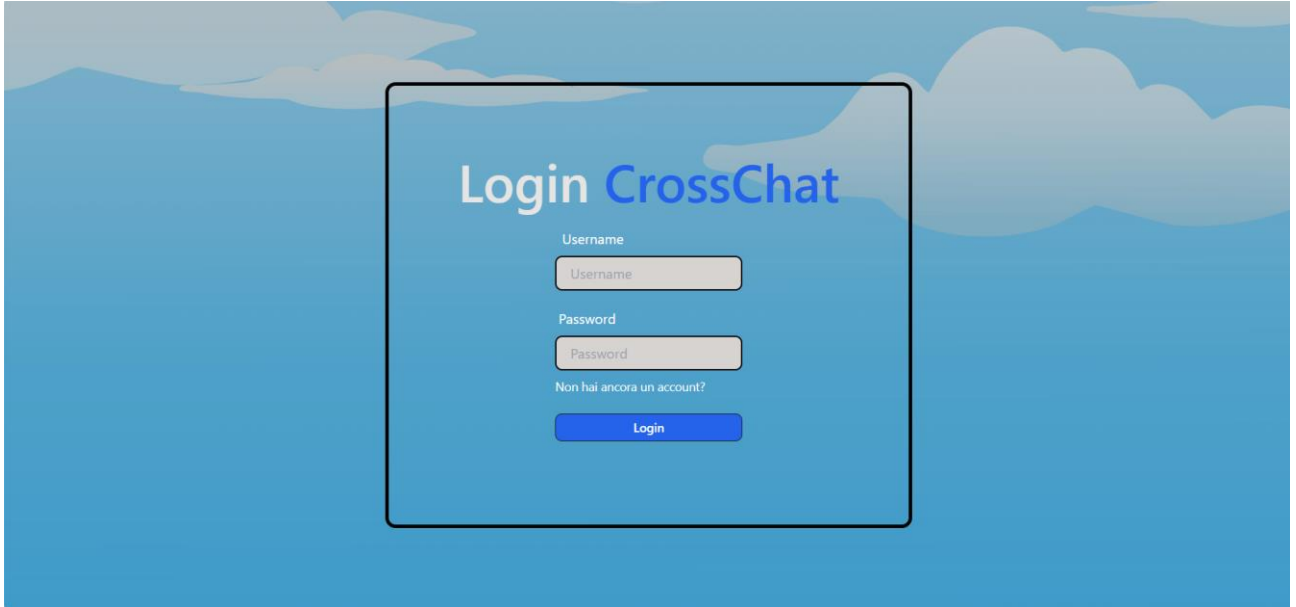


**Render:** per effettuare il deployment della web app e renderla utilizzabile da chiunque e in qualsiasi momento.



## 1.4 Mockup

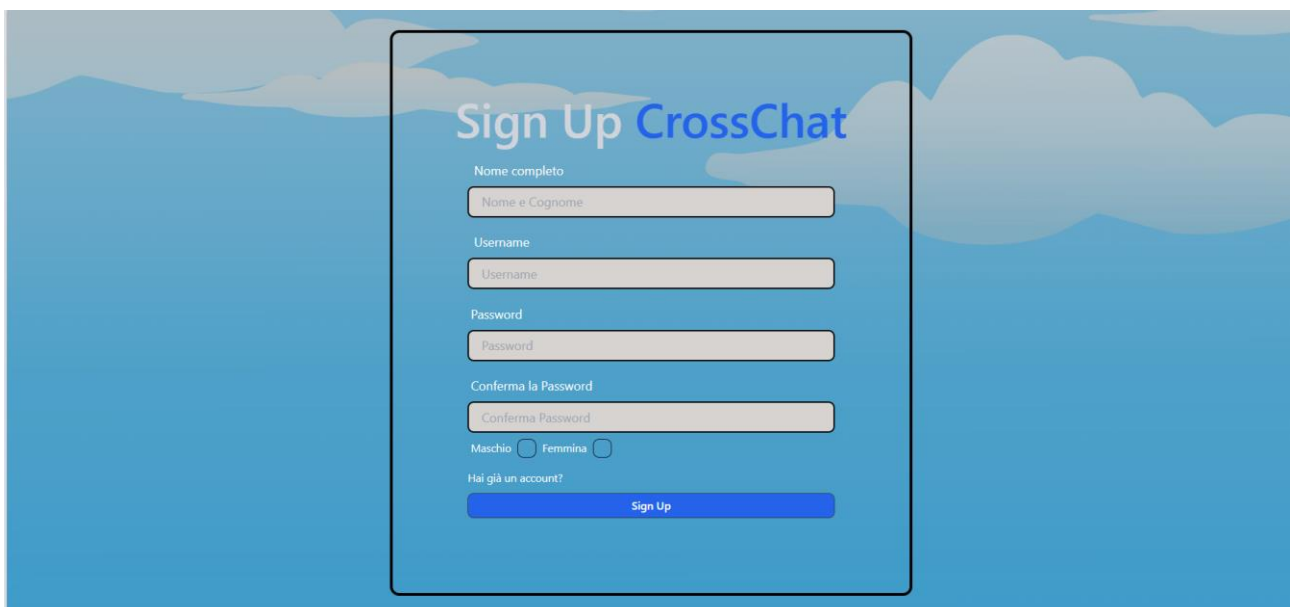
- Schermata Login



A login form titled "Login CrossChat" centered on a blue background with a stylized mountain and cloud illustration. The form is enclosed in a black rectangular border. It contains the following elements: a "Username" label above a text input field with "Username" as a placeholder; a "Password" label above a text input field with "Password" as a placeholder; a link "Non hai ancora un account?" below the password field; and a blue "Login" button at the bottom.

Figura 2: Mockup Login

- Schermata Registrazione



A sign-up form titled "Sign Up CrossChat" centered on a blue background with a stylized mountain and cloud illustration. The form is enclosed in a black rectangular border. It contains the following elements: a "Nome completo" label above a text input field with "Nome e Cognome" as a placeholder; a "Username" label above a text input field with "Username" as a placeholder; a "Password" label above a text input field with "Password" as a placeholder; a "Conferma la Password" label above a text input field with "Conferma Password" as a placeholder; a "Maschio" label with an unchecked radio button, followed by a "Femmina" label with an unchecked radio button; a link "Hai già un account?" below the gender options; and a blue "Sign Up" button at the bottom.

Figura 3: Mockup SignUp

- Schermata Home

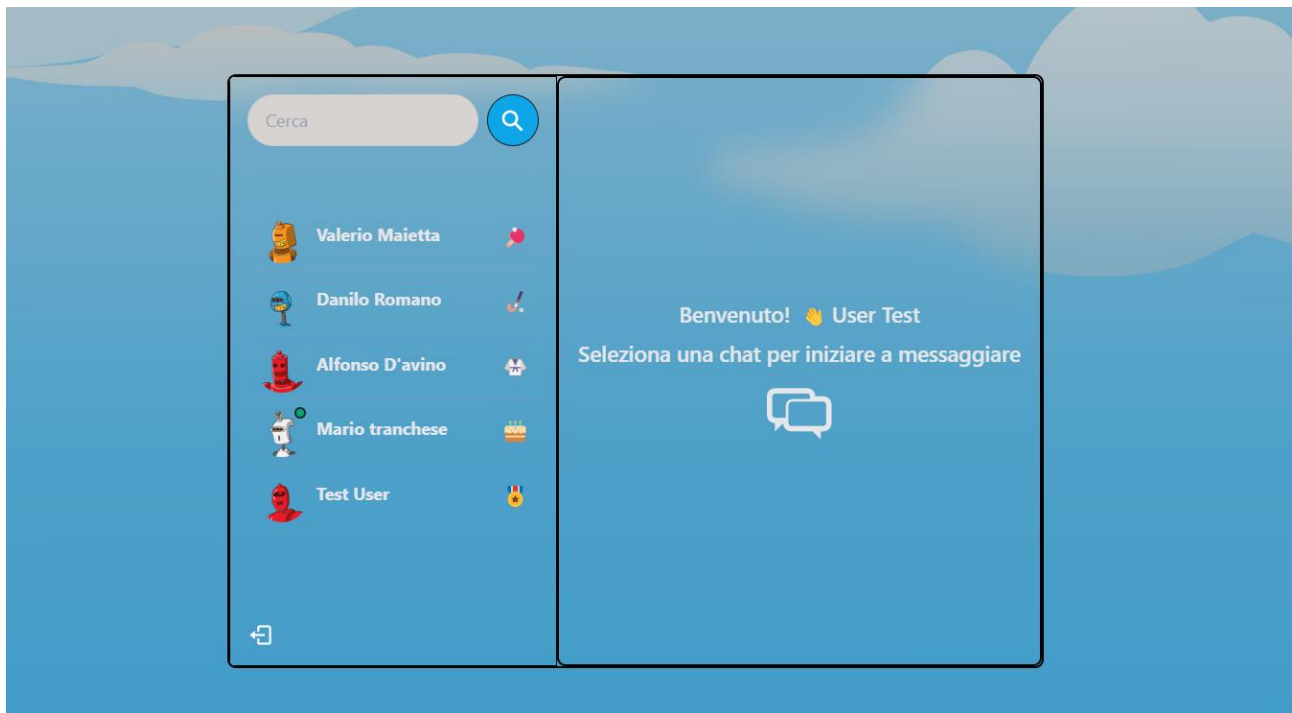


Figura 4: Mockup Schermata Home

- Schermata Chat

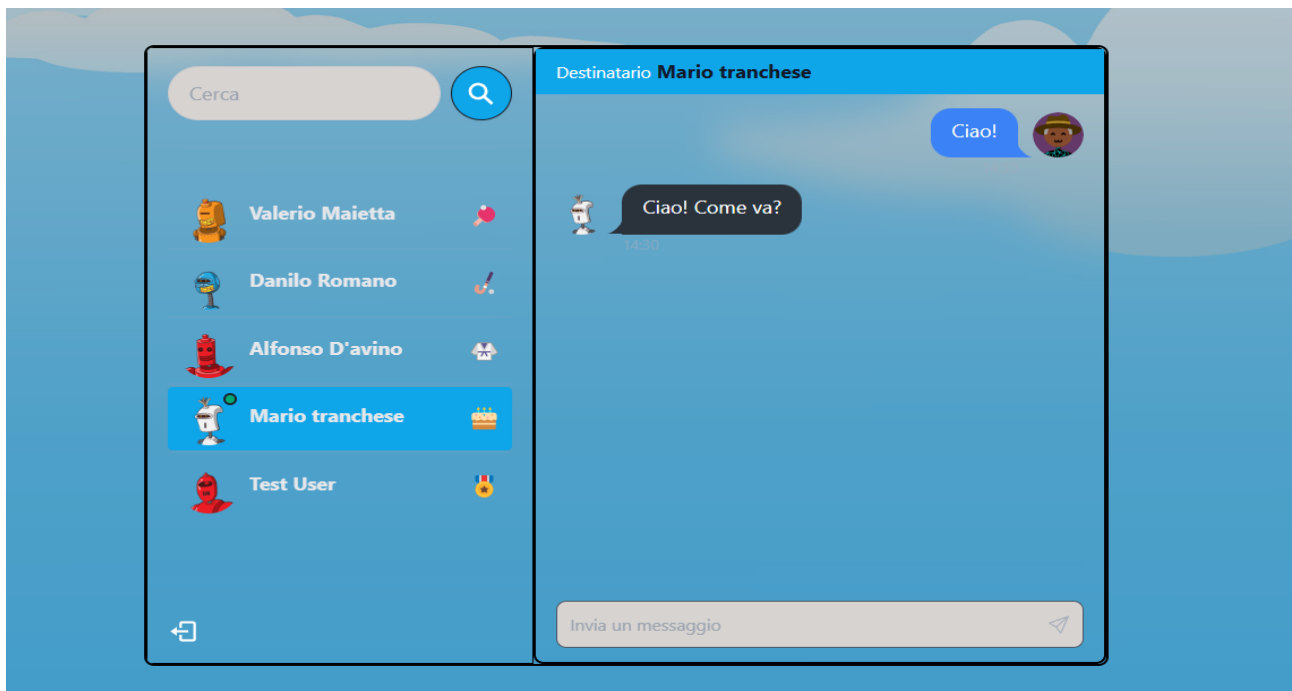


Figura 5: Mockup Schermata Chat

## Capitolo 2: Analisi del dominio

### Analisi preliminare

L'analisi preliminare del dominio ha permesso di individuare i requisiti funzionali e non funzionali, essenziali al fine di specializzare il progetto scelto.

Abbiamo tre attori principali: **l'utente non registrato**, **l'utente (registrato) non autenticato** e **l'utente autenticato**.

Attore	Descrizione
Utente non registrato	Colui che per accedere all'applicazione ha bisogno di registrarsi.
Utente non autenticato	Colui che è registrato e che per accedere all'applicazione ha bisogno di effettuare il login.
Utente autenticato	Colui che accede all'applicazione e può messaggiare con altri utenti autenticati.

### 2.1 Specifica dei requisiti

Per arrivare alla specifica dei requisiti si è partito dalle **user stories**, in cui vengono identificate le cose che un utente può e non può fare.

#### 2.1.1 Storie utente

Abbiamo individuato le seguenti storie:

1. Come utente voglio potermi registrare al sistema per poter chattare.
2. Come utente voglio poter effettuare il login per accedere alle conversazioni.
3. Come utente voglio poter cercare altri utenti tramite una barra di ricerca.
4. Come utente voglio poter chattare con altri utenti attraverso chat private.
5. Come utente voglio poter effettuare il logout.

## 2.1.2 Requisiti funzionali

I requisiti funzionali sono fondamentali per la **costruzione** dei **casi d'uso**. A partire dalle storie utente, sono stati estratti i seguenti requisiti:

### 1) Registrazione

Gli utenti devono poter creare un account fornendo nome, cognome, nome utente e password.

### 2) Login

Gli utenti devono poter effettuare il login utilizzando nome utente e password.

### 3) Creazione di Conversazioni

Gli utenti devono poter iniziare una nuova conversazione con un altro utente.

### 4) Comunicazione real-time

Gli utenti devono poter inviare e ricevere messaggi di testo in tempo reale.

### 5) Ricerca Utenti

Gli utenti devono poter cercare conversazioni con altri utenti tramite un'apposita barra di ricerca.

### 6) Sicurezza delle credenziali

Le credenziali devono essere conservate in modo sicuro (criptazione delle password).

### 7) Autenticazione Necessaria

L'utente deve essere autenticato prima di poter usufruire del servizio.

### 8) Salvataggio credenziali nel database

Il sistema salva nel database i dati relativi alla registrazione da parte dell'utente, in modo da poter accedere senza problemi nei successivi login.

## 9) Verifica Credenziali

Il sistema ad ogni registrazione e/o login effettua una verifica delle credenziali in modo da confermare il corretto inserimento dei dati da parte dell'utente.

## 10) Logout

Gli utenti devono poter effettuare il logout dall'applicazione.

## 11) Visualizzazione stato utente

Gli utenti devono poter visualizzare lo stato degli altri utenti con cui si chatta, ossia se è online o meno.

## 12) Notifiche

Gli utenti devono poter ricevere notifiche sonore per le conversazioni in modo da essere aggiornati sullo stato di una conversazione.

## 2.1.3 Requisiti non funzionali

I requisiti non funzionali permettono di **definire** quelle **caratteristiche** del sistema che non si traducono in un caso d'uso ma che ne influenzano le proprietà. Di seguito vengono riportati i requisiti individuati:

### 1) Consistenza dei dati e univocità degli identificativi

Il sistema deve garantire la consistenza dei dati e che gli identificativi siano univoci.

### 2) Usabilità

Il sistema deve garantire un buon grado di usabilità per renderlo facilmente fruibile dall'utente.

### 3) Portabilità

Il sistema deve essere portabile, ovvero deve essere compatibile su browser diversi, indipendentemente dal sistema operativo e dal dispositivo in uso.



## 2.2 Scenari di funzionamento

Durante la stesura degli scenari di funzionamento, sono stati individuati, a partire dai requisiti funzionali, quattro **scenari di base**: il primo riguardante la **registrazione**, il secondo riguardante il **Login**, il terzo riguardante l'**invio di un messaggio ad un altro utente**, ed il quarto riguardante **la ricezione del messaggio da parte dell'utente**.

### 2.2.1 Scenario Uno - Registrazione

- **Attore principale**: Utente

**Precondizioni**

Nessuna

**Scenario principale di successo (flusso base):**

1. L'utente inserisce nome, cognome, username e password
2. Il sistema verifica che i campi inseriti siano validi e successivamente permette la registrazione.
3. L'utente risulta registrato, autenticato e può iniziare a chattare

**Flussi alternativi (username e/o password non validi)**

1. L'utente inserisce nome, cognome, username e password
2. Il sistema verifica che almeno uno dei campi non è valido e quindi nega la registrazione.
3. L'utente visualizza l'eccezione e riesegue il flusso base.

### 2.2.2 Scenario Due - Login

- **Attore principale**: Utente

**Precondizioni**

L'utente deve essere registrato.

**Scenario principale di successo (flusso base):**

1. L'utente inserisce username e password

2. Il sistema verifica la validità delle credenziali e permette l'accesso
3. L'utente risulta autenticato e può iniziare a chattare

#### **Flussi alternativi (username e/o password non validi)**

1. L'utente inserisce username e password
2. Il sistema verifica la validità delle credenziali e nega l'accesso.
3. L'utente visualizza l'eccezione e riesegue il flusso base.

### **2.2.3 Scenario Tre – Invio di un messaggio ad un altro utente**

- **Attore principale:** Utente

#### **Precondizioni**

L'utente deve essere loggato.

#### **Scenario principale di successo (flusso base):**

1. L'utente dopo aver loggato visualizza le chat con gli altri utenti.
2. L'utente apre la chat desiderata e scrive un messaggio nella barra apposita.

#### **Flussi alternativi (Problematiche varie):**

1. L'utente dopo aver loggato visualizza le chat con gli altri utenti.
2. L'utente apre la chat desiderata e scrive un messaggio nella barra apposita.
3. Il messaggio non viene inviato a causa di possibili problematiche (es. di rete o sistema).
4. L'utente provvederà ad un nuovo invio in seguito alla loro risoluzione.

### **2.2.4 Scenario Quattro – Ricezione di un messaggio da un altro utente**

- **Attore principale:** Utente

#### **Precondizioni**

L'utente deve essere loggato.

#### **Scenario principale di successo (flusso base):**

1. L'utente dopo aver loggato visualizza le chat con gli altri utenti.

2. L'utente riceve una notifica di ricezione di un messaggio.
3. L'utente apre la chat e visualizza il messaggio nella conversazione.

#### **Flussi alternativi (Problematiche varie):**

1. L'utente dopo aver loggato visualizza le chat con gli altri utenti.
2. L'utente non riceve una notifica di ricezione di un messaggio a causa di problematiche varie (ad es. di rete o sistema).

## **2.3 Product Backlog**

Una product backlog è un elemento chiave dell'approccio Agile. È una lista prioritaria di funzionalità, miglioramenti e correzioni di bug che il team di sviluppo deve implementare, costantemente aggiornata per riflettere le esigenze e le priorità del progetto.

ID	User Story	Descrizione	Requisiti	Attività
1	Registrazione	Come utente, devo potermi registrare.	Campo Nome Completo, Campo Username, Campo password, Messaggi di errore	Implementazione form di registrazione, Implementazione logica backend
2	Login	Come utente, devo potermi loggare.	Campo Username, Campo password, Gestione sessione utente, Messaggi di errore	Implementazione form di login, Implementazione logica backend, Gestione delle sessioni utente
3	Accesso Sicuro	Come utente, voglio accedere a CrossChat tramite autenticazione sicura.	Criptazione delle password e gestione sicura della sessione	Hashing delle password tramite Bcrypt e gestione della sessione utente tramite JWT
4	Creazione di Conversazioni	Come utente, voglio creare una nuova conversazione.	Campo messaggio, spazio di messaggistica	Implementazione logica backend
5	Ricerca di Utenti	Come utente, voglio cercare conversazioni con vari utenti usando il loro username.	Campo di ricerca	Implementazione funzionalità di ricerca

6	Sistema di notifiche	Come utente, voglio ricevere notifiche alla ricezione di un nuovo messaggio.	Sistema di notifiche	Implementazione sistema di notifiche
7	Profilo personalizzato	Come utente, a seconda del genere, si genererà una diversa immagine di profilo.	Cambio immagine profilo	Collegamento con API che fornisce una selezione randomica di immagini per il profilo
8	Logout	Come utente voglio poter chiudere la mia sessione	L'utente deve essere loggato	Implementazione logica backend

## 2.4 Modellazione Casi d'uso

ID Caso d'Uso	Nome Caso d'Uso	Attori	Descrizione	Pre-condizioni	Post-condizioni
UC1	Effettua Registrazione	Utente	L'utente può creare un nuovo account fornendo le informazioni richieste (nome, cognome, username e password)	Nessuna	Account utente creato
UC2	Effettua Login	Utente	L'utente può accedere al proprio account fornendo username e password.	Utente registrato	Utente autenticato e sessione attiva
UC3	Invio/Ricezione Messaggio	Utente	L'utente deve poter avere accesso alla funzionalità di messaggistica.	L'utente è autenticato nel sistema.	L'utente legge/scrive il messaggio correttamente
UC4	Ricerca chat	Utente	L'utente può cercare nella barra apposita lo username di altri utenti	L'utente deve essere registrato e loggato	L'utente vedrà a schermo la chat ricercata
UC5	Effettua Logout	Utente	L'utente può uscire dall'applicazione cliccando sul tasto di logout	L'utente deve essere loggato	L'utente dovrà inserire nuovamente le sue credenziali per accedere all'applicazione

### 2.4.1 Diagramma dei Casi d'uso

Di seguito viene riportato il rispettivo grafico in UML dei casi d'uso, implementato a partire dalla rappresentazione tabellare, dai requisiti funzionali e dagli scenari con i flussi base e alternativi.

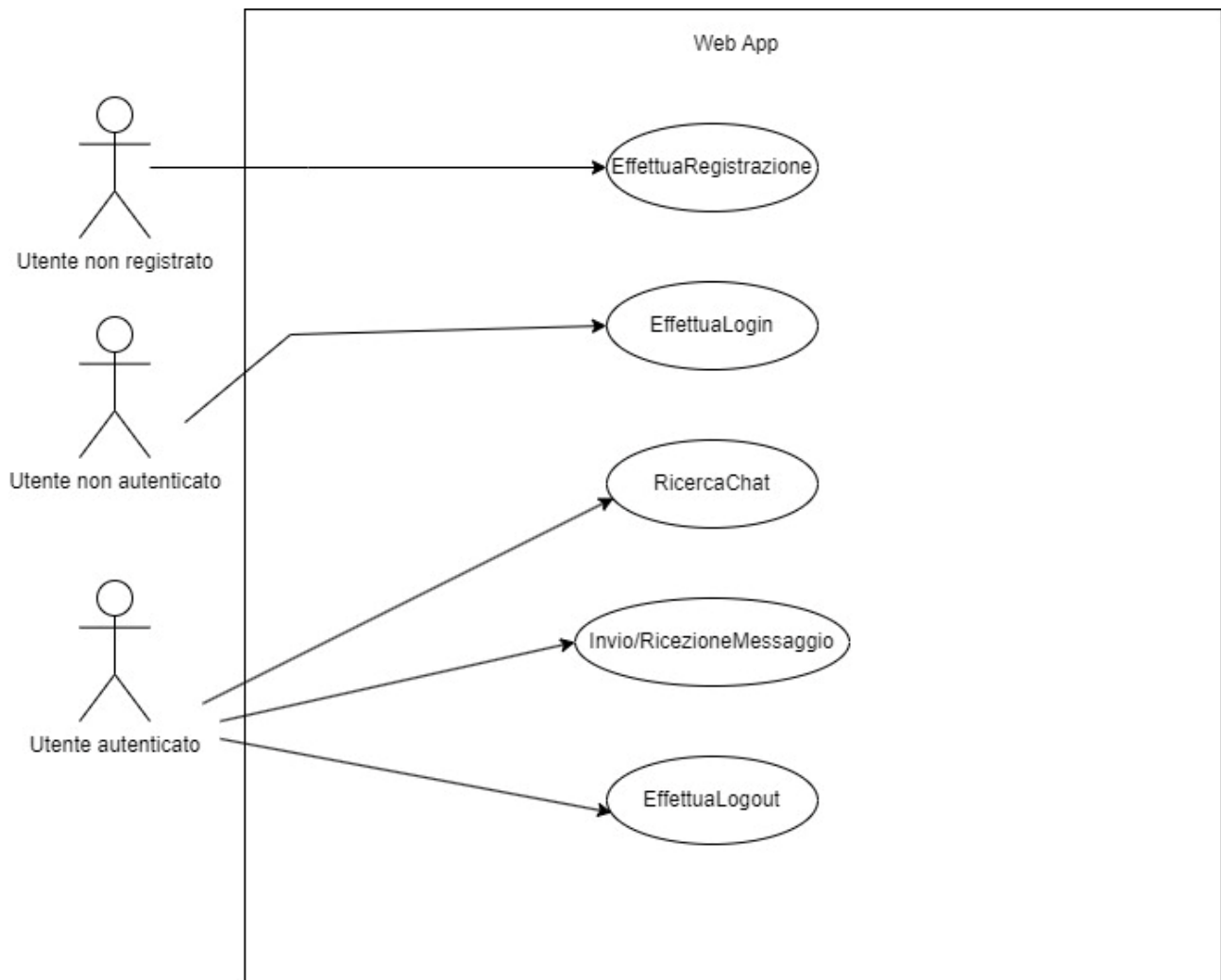


Figura 6: Diagramma casi d'uso

## 2.5 Diagrammi di Attività

I diagrammi di attività sono uno dei tipi di diagrammi UML (Unified Modelling Language) utilizzati nella progettazione del software per descrivere visivamente il flusso di lavoro o il comportamento di un sistema. Questi diagrammi sono particolarmente utili per modellare processi, procedure e attività all'interno di un sistema software. Ecco i diagrammi riguardanti la web app sotto esame:

### 2.5.1 Registrazione

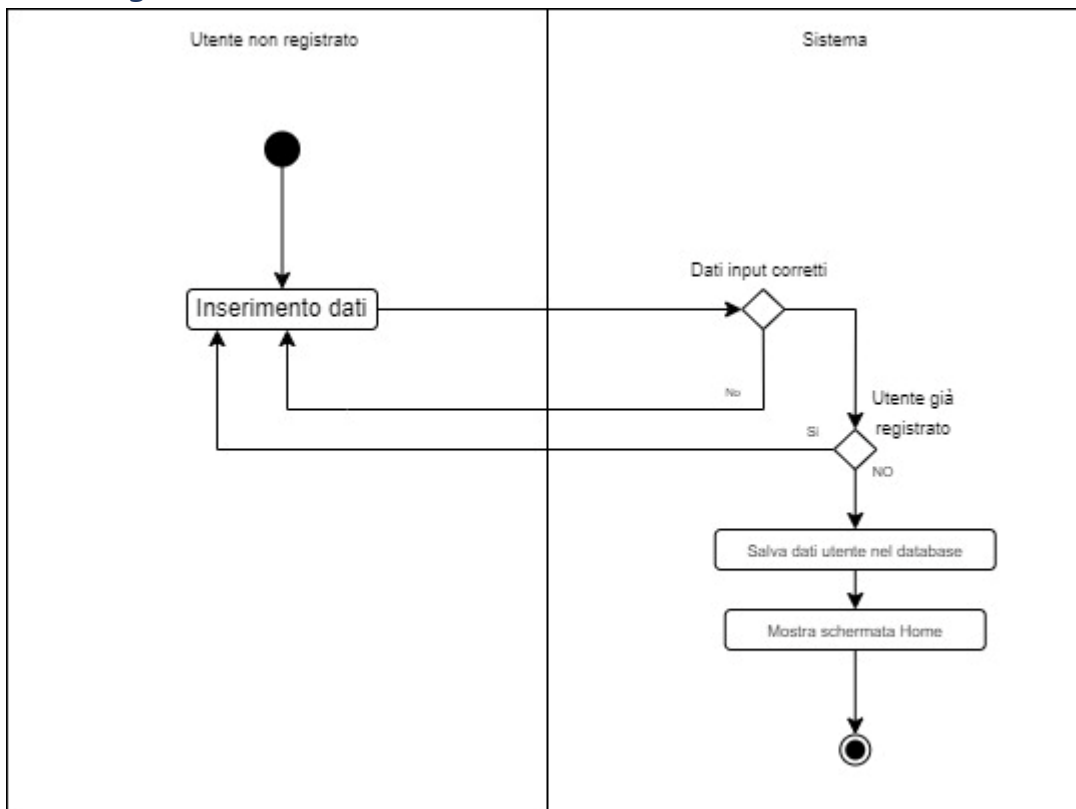


Figura 7: Registrazione Diagramma d'attività

### 2.5.2 Login

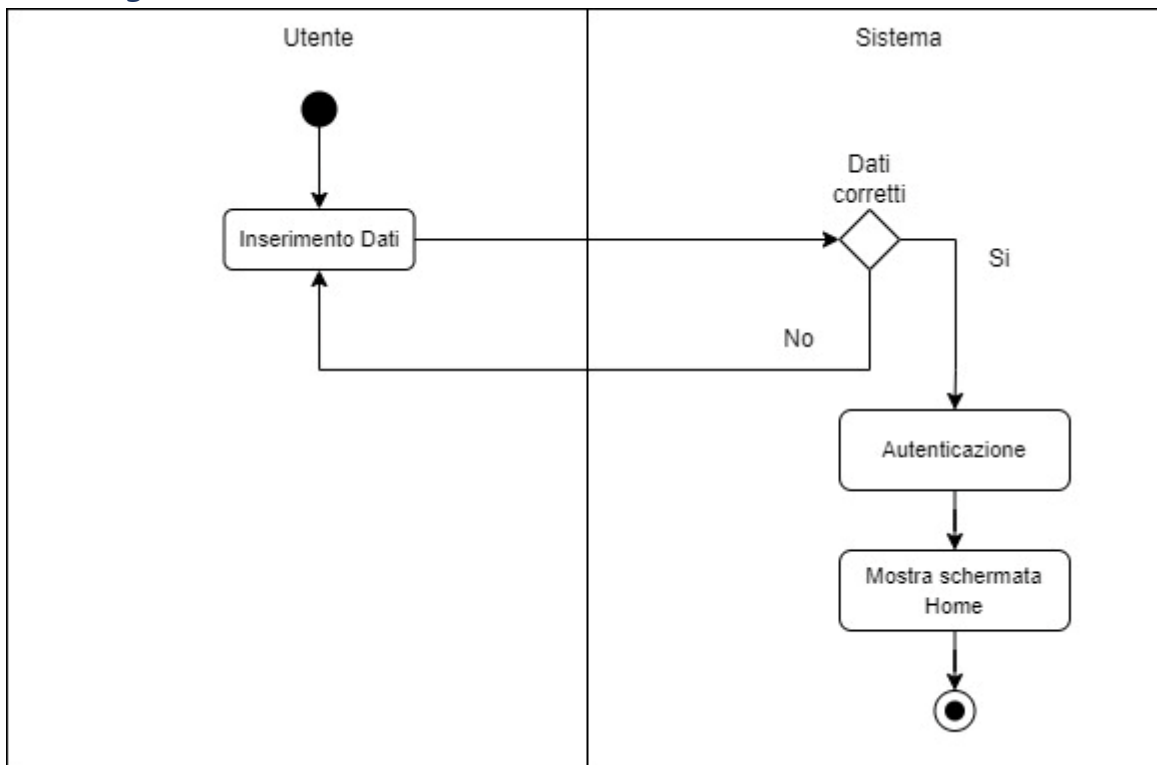


Figura 8: Login Diagramma d'attività

### 2.5.3 Invio/Ricezione messaggio

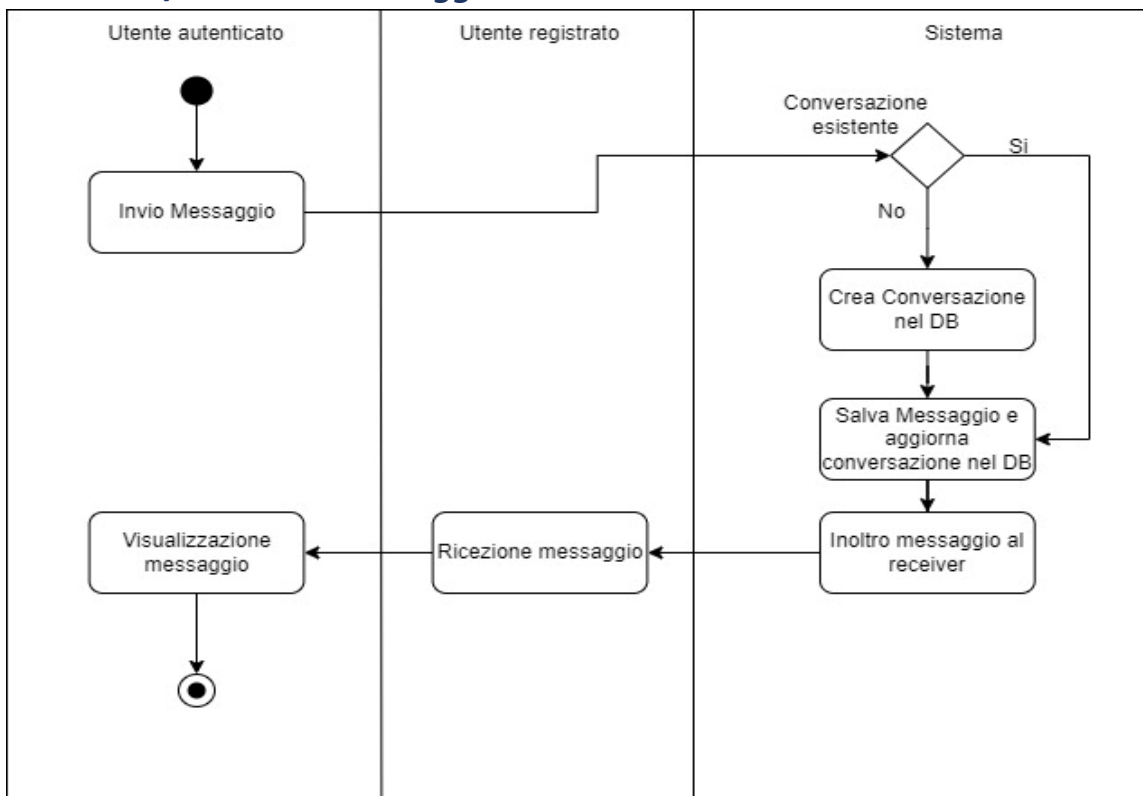


Figura 9: Invio/Ricezione messaggio Diagramma d'attività

### 2.5.4 Ricerca Chat

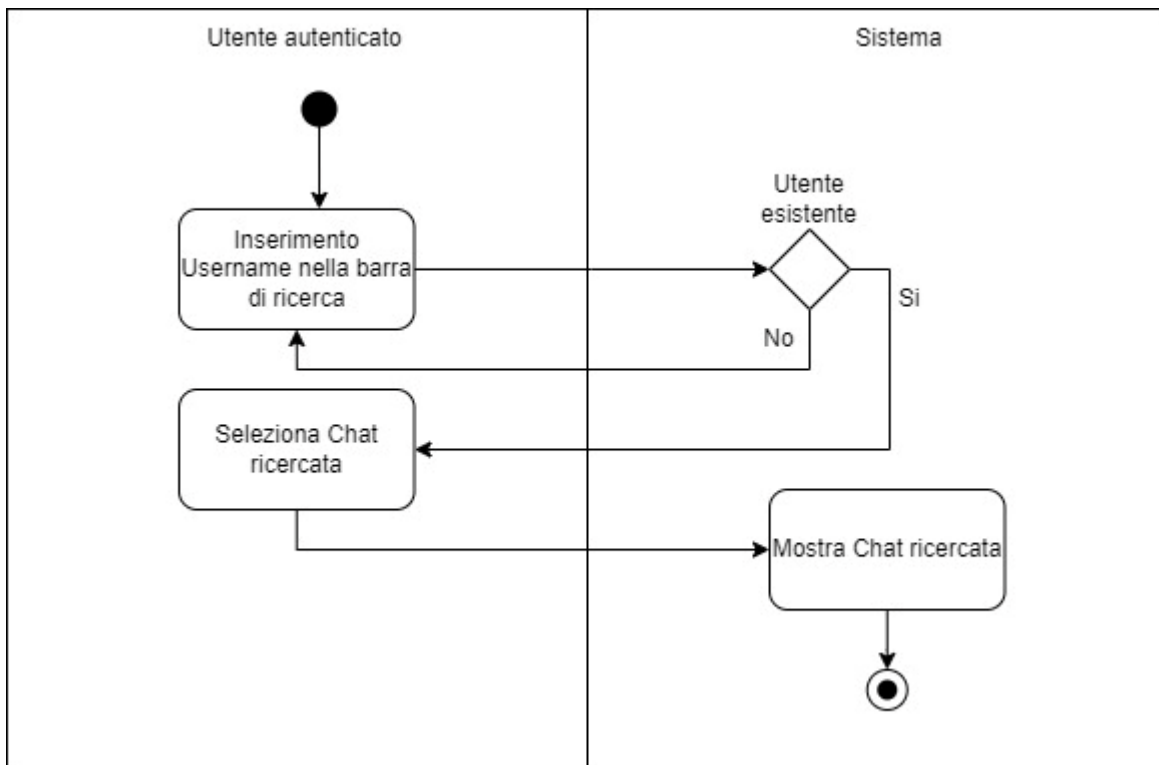


Figura 10: Ricerca Chat Diagramma d'attività

### 2.5.5 Logout

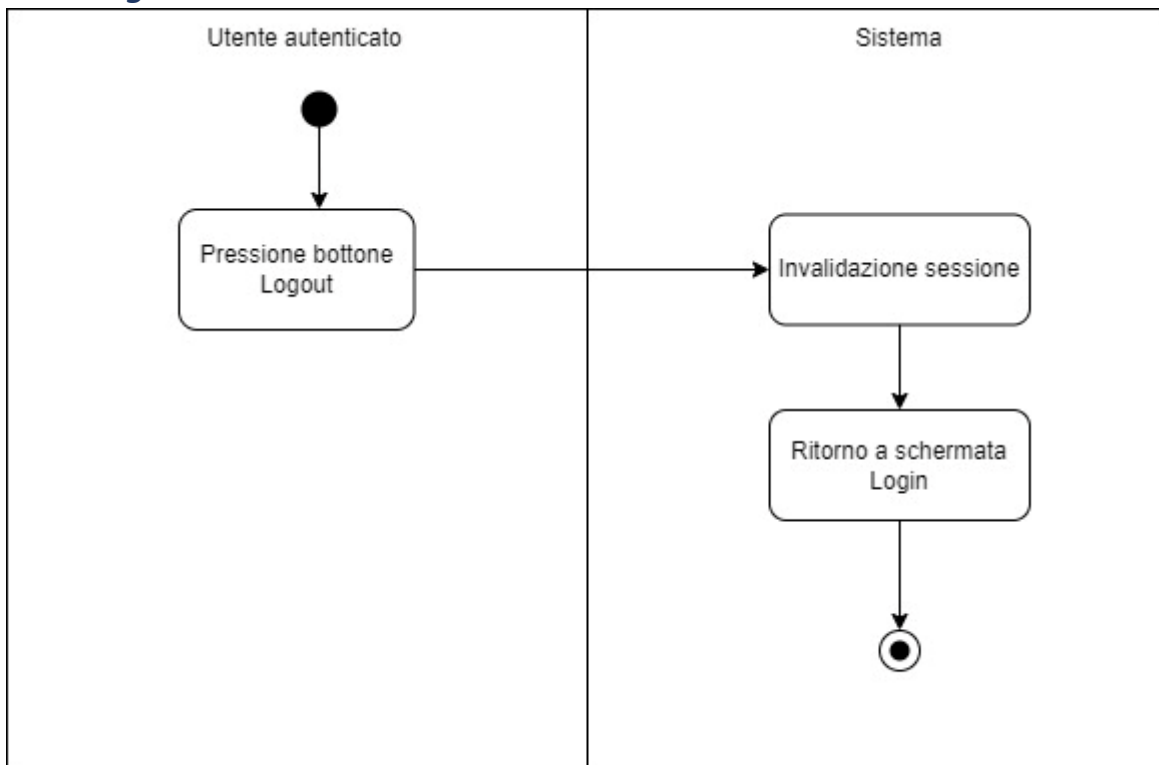


Figura 11: Logout Diagramma d'attività



## 2.6 Diagramma di Contesto

Il diagramma di contesto è un tipo di diagramma che risulta essere molto utile per avere una visione di insieme di alto livello sul sistema. Esso permette di mappare il flusso di informazioni tra il sistema e l'ambiente esterno.

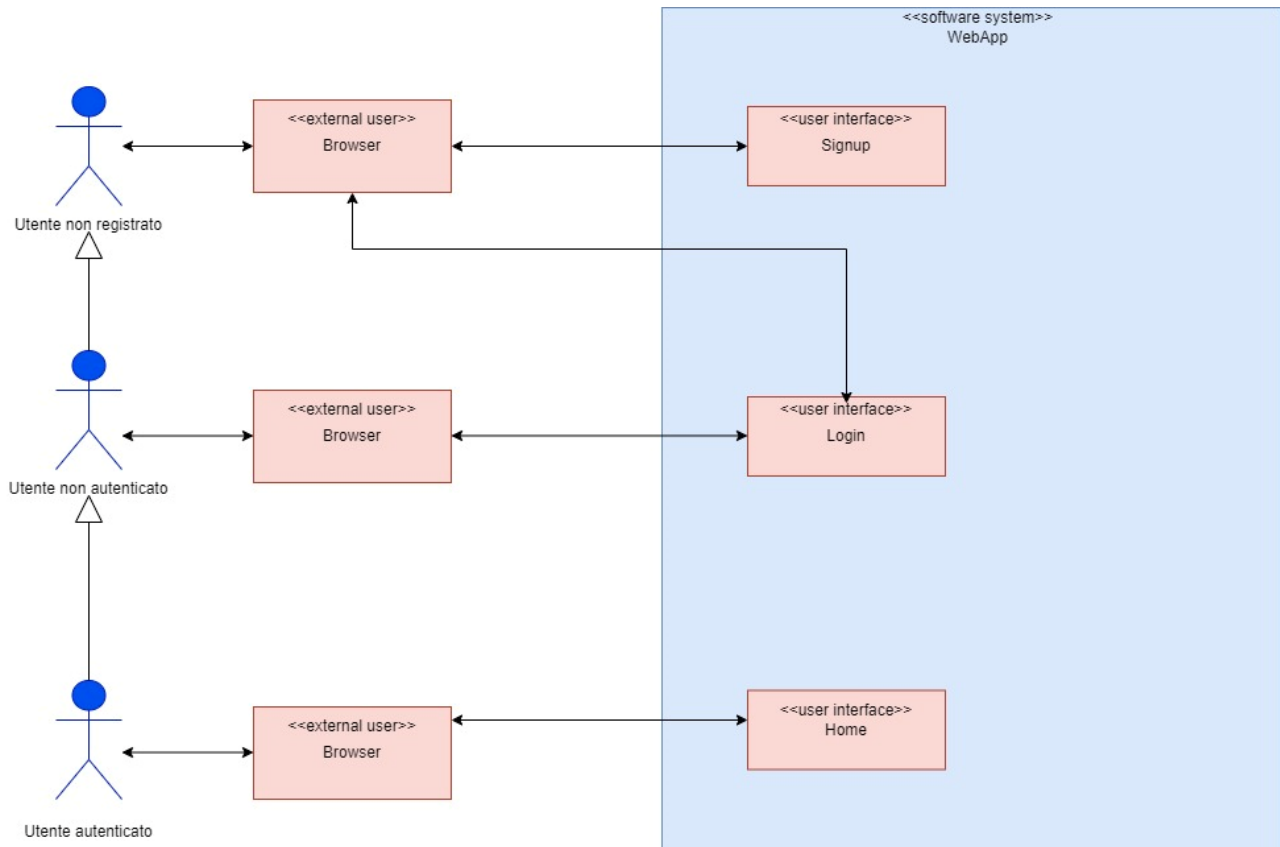


Figura 12 - Diagramma di contesto

## 2.7 Diagrammi di Sequenza

I diagrammi di sequenza in UML rappresentano il flusso di interazioni tra gli oggetti di un sistema in modo lineare e chiaro. Questi diagrammi mostrano come gli oggetti comunicano tra loro attraverso una serie di messaggi nel tempo, aiutando a visualizzare il comportamento del sistema e a comprendere le interazioni fondamentali tra i componenti.

### 2.7.1 Registrazione

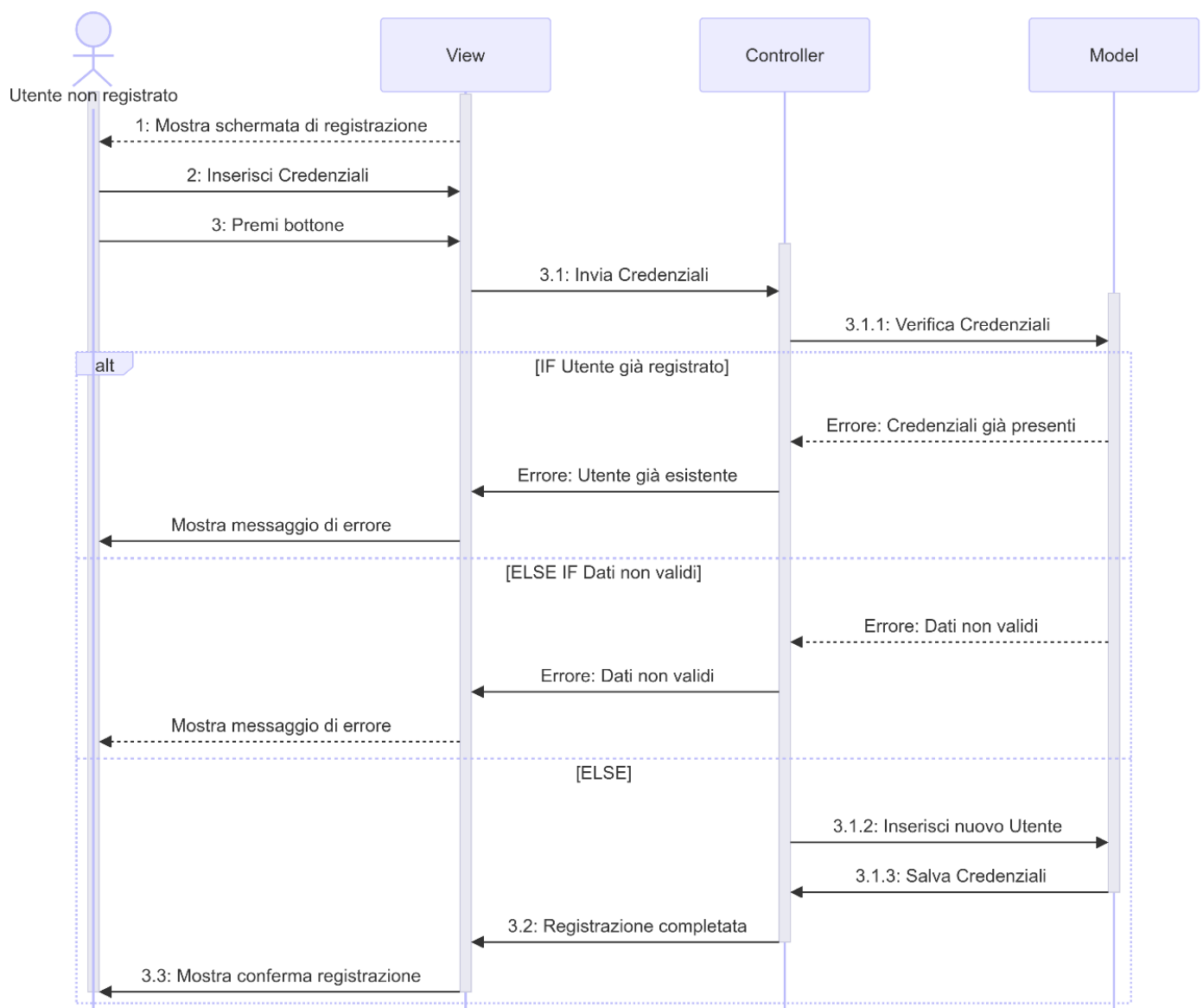


Figura 13: Registrazione Diagramma di sequenza non raffinato

## 2.7.2 Login

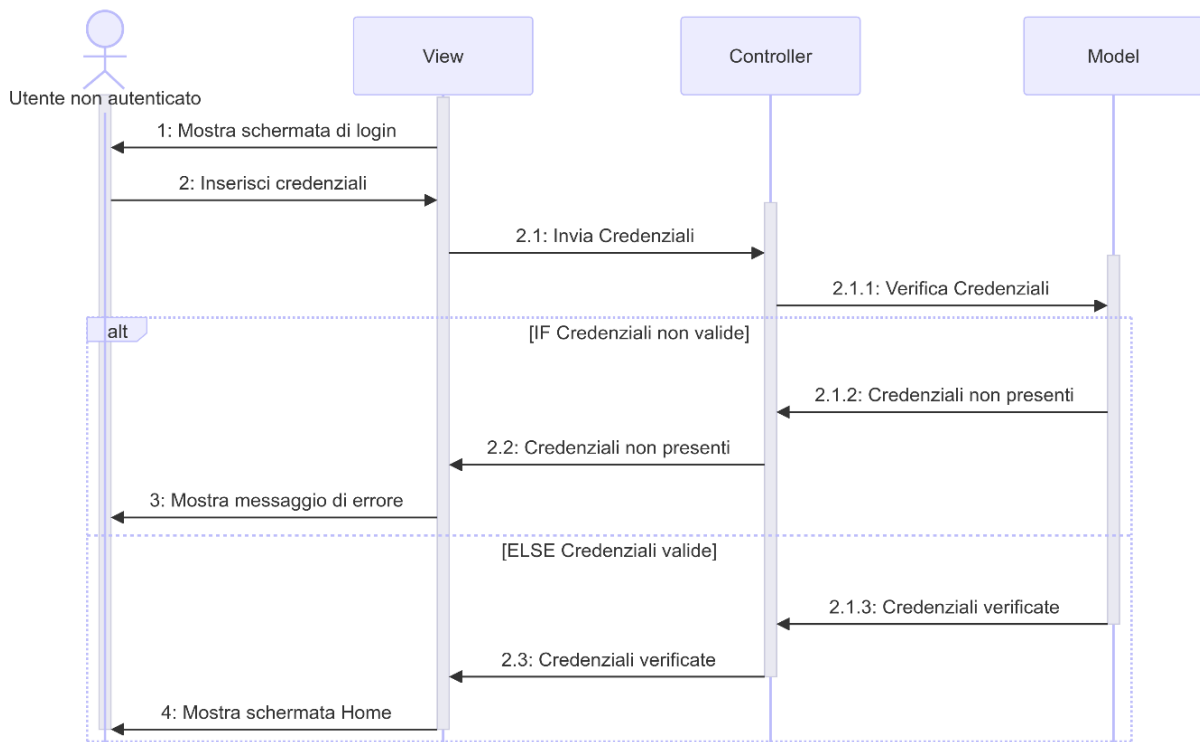


Figura 14: Login Diagramma di sequenza non raffinato

### 2.7.3 Ricerca Chat

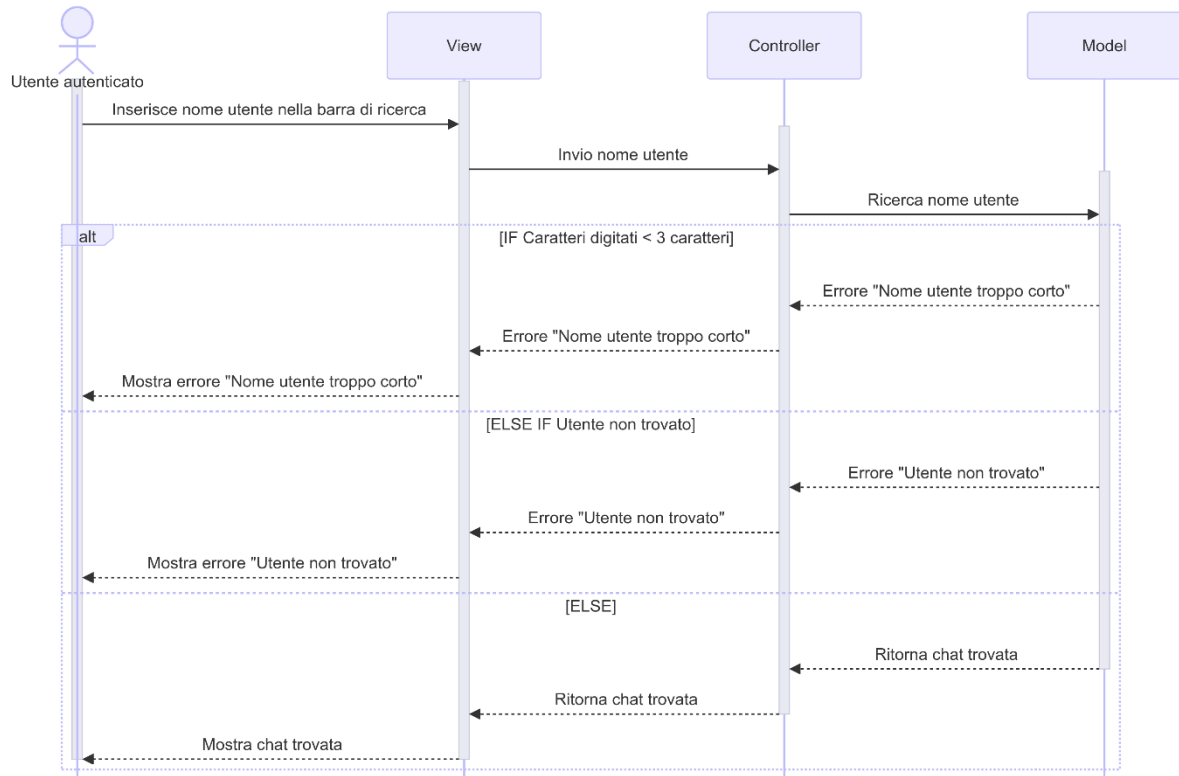


Figura 15: Ricerca Chat Diagramma di sequenza non raffinato

## 2.7.4 Invio/Ricezione Messaggio

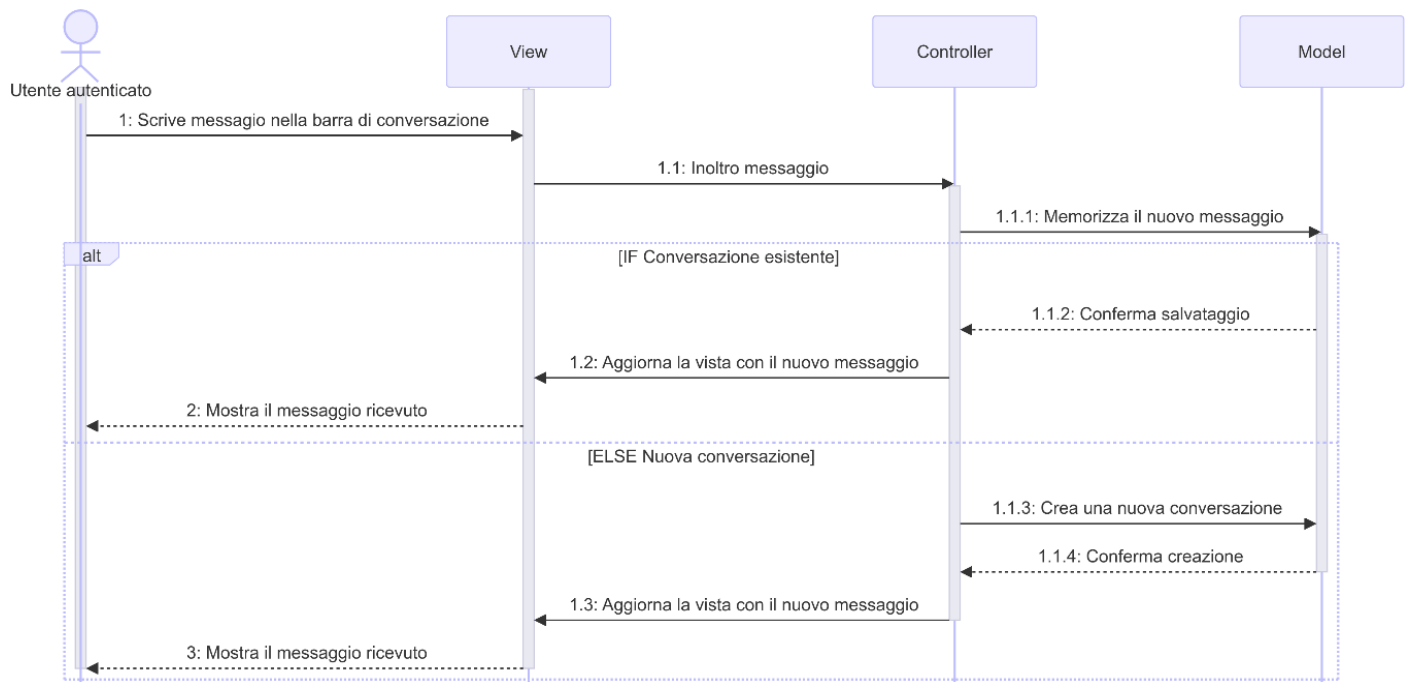


Figura 16: Invio/Ricezione Messaggio Diagramma di sequenza non raffinato

## 2.7.5 Logout

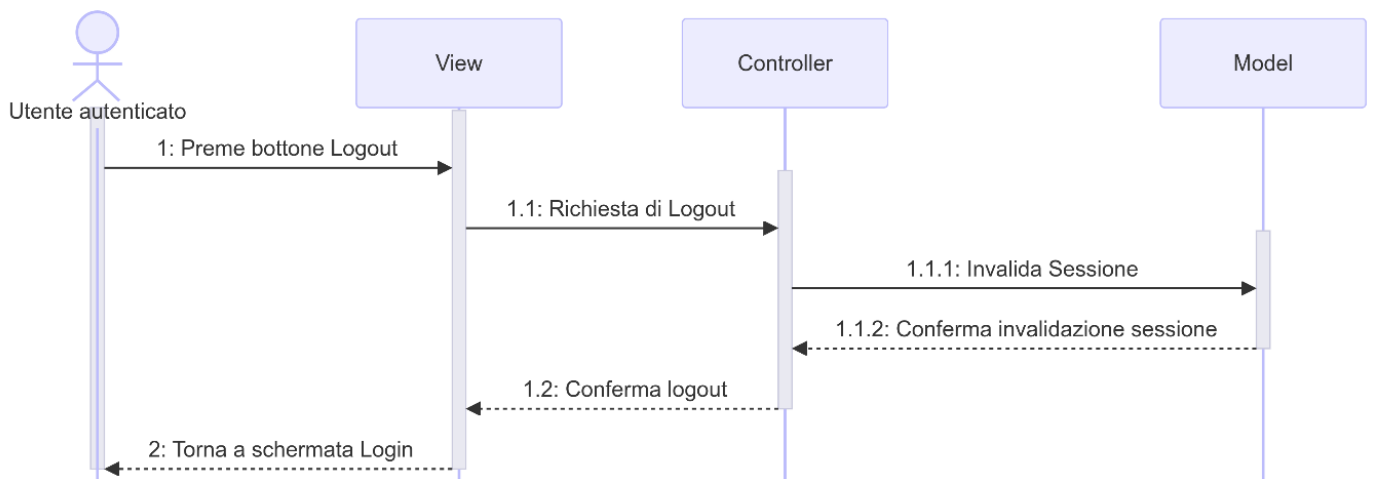


Figura 17: Logout Diagramma di sequenza non raffinato

## Capitolo 3: Architettura e progettazione

### 3.1 Tecnologie usate per l'implementazione della web app

Questo capitolo descrive l'implementazione della web app di messaggistica *CrossChat* utilizzando lo stack **MERN** (MongoDB, Express, React, Node.js). Analizzeremo i componenti chiave, inclusa l'integrazione di **Socket.io** per la gestione della messaggistica in tempo reale.

#### 3.1.1 Implementazione backend: NodeJS

Il backend della nostra web app di messaggistica è stato sviluppato utilizzando Node.js, una piattaforma JavaScript che permette di costruire applicazioni server-side performanti e scalabili. In questo sottocapitolo, esamineremo i dettagli dell'implementazione del backend, soffermandoci sui seguenti aspetti:

##### 1. Configurazione del Server

Il server è stato configurato nel file *server.js* utilizzando il framework Express, che semplifica la gestione delle richieste HTTP e l'organizzazione delle routes.

##### 2. Gestione delle Routes

Le routes sono gestite tramite Express e organizzate in modo modulare per mantenere il codice pulito e manutenibile. Di seguito un esempio di definizione di alcune routes di base:

```
router.post("/signup", signup);  
router.post("/login", login);  
router.post("/logout", logout);
```

##### 3. Sicurezza e Middleware

Abbiamo implementato un middleware per migliorare la sicurezza e la funzionalità dell'applicazione. Ad esempio, *protectRoute.js* è un middleware che verifica l'autenticità del token JWT presente nei cookie delle richieste in arrivo. Questo middleware:

- Verifica l'esistenza del token.

- Decodifica il token per ottenere l'ID dell'utente.
- Recupera l'utente dal database escludendo la password.
- Se il token è valido e l'utente esiste, supera il controllo.

La cartella “*utils*” contiene il file *generateToken.js*, utilizzata per generare token JWT per l'autenticazione degli utenti. La sicurezza dell'applicazione è ulteriormente migliorata utilizzando middleware come CORS per permettere o restringere le richieste provenienti da domini differenti (cross-origin).

#### 4. Gestione delle Variabili di Ambiente

Per garantire la sicurezza e la configurabilità del progetto, abbiamo utilizzato un file “*.env*” per memorizzare variabili di ambiente sensibili. Il file “*.env*” include:

- **JWT\_SECRET:** La chiave segreta utilizzata per firmare e verificare i token JWT. Questa chiave è stata generata randomicamente per evitare possibili attacchi e migliorare la sicurezza.
- **PORT:** La porta su cui è attivo il servizio backend.
- **MONGODB\_URI:** L'URI per la connessione al database MongoDB.

#### 5. Gestione delle Dipendenze

Le dipendenze necessarie per il funzionamento del backend sono state installate tramite `npm install` e sono elencate nei file “*package.json*” e “*package-lock.json*”. Queste dipendenze includono, ma non sono limitate a, Express per la gestione delle routes, Mongoose per l'interazione con MongoDB, Socket.io per la comunicazione in tempo reale, e vari middleware per la sicurezza e l'autenticazione. Tutte le dipendenze installate sono memorizzate nella cartella “*node\_modules*”, che contiene i pacchetti necessari per l'esecuzione dell'applicazione.

### 3.1.2 Implementazione frontend: React

React è un framework JavaScript per la creazione di interfacce utente, sviluppata da Meta. È particolarmente utilizzata per lo sviluppo di applicazioni web moderne e dinamiche. React è noto per la sua efficienza, modularità e facilità d'uso, grazie a concetti come i componenti, lo stato e il virtual DOM.

## Caratteristiche principali di React:

- Componenti e modularità: in React, l'interfaccia utente è suddivisa in piccoli componenti riutilizzabili, ciascuno dei quali gestisce la propria logica e il proprio rendering.
- **Sintassi JSX:** JSX è una sintassi che consente di scrivere HTML all'interno di JavaScript. È opzionale ma comunemente usato in React per rendere il codice più leggibile.
- **Stato e Props:**
  - o Stato (State): Ogni componente può avere un proprio stato, che rappresenta i dati che cambiano nel tempo. Quando lo stato cambia, React ri-renderizza solo quel componente e non tutta l'interfaccia.
  - o Props (Properties): I props sono valori passati ai componenti parent per configurare i componenti child. I props sono immutabili all'interno del componente child.
- **Virtual DOM:** React mantiene una copia in memoria del DOM chiamata Virtual DOM. Quando lo stato cambia, React aggiorna il Virtual DOM e poi effettua una comparazione (diffing) con il DOM reale per minimizzare le operazioni di manipolazione del DOM.

Per la creazione di un progetto React è possibile usare il modulo JavaScript **ViteJS**, tramite il comando `"npm create vite@latest"`, che andrà ad effettuare la creazione del progetto utilizzando l'ultima versione disponibile del framework. Insieme ad esso verranno create anche directory e file utili al funzionamento della web app. Tra queste troviamo la cartella **dist** o **utils**, che conterranno funzioni di utilità che sono generiche e riutilizzabili in diverse parti della web app o la cartella **assets**, utilizzata per organizzare e gestire risorse statiche come Immagini, icone, ecc. Di fondamentale importanza in React sono gli **hooks**, che sono funzioni che consentono di agganciarsi alle funzionalità dei componenti React. Un altro concetto molto importante in React è quello del contesto (**context**), esso è un meccanismo che permette di passare dati attraverso la gerarchia dei componenti senza dover passare esplicitamente le props attraverso ogni livello. Questo è particolarmente utile per dati che



devono essere accessibili da molti componenti, come nel nostro caso le informazioni di autenticazione.

### **Caratteristiche principali del nostro progetto React:**

Essendo il nostro progetto React una web app di messaggistica istantanea, abbiamo pensato di creare tre pagine:

- **Signup**, la schermata in cui è possibile effettuare la registrazione;
- **Login**, la schermata in cui è possibile effettuare l'accesso inserendo le credenziali;
- **Home**, la schermata in cui è possibile messaggiare ed effettuare la ricerca degli utenti. Questa schermata è visibile solo per gli utenti autenticati.

e tre componenti principali:

- **Messages**, il folder che contiene tutta la parte relativa all'UI dei messaggi;
- **Sidebar**, che contiene la parte relativa all'UI del menu laterale contenente a sua volta gli utenti con i quali è possibile scambiare messaggi;
- **Skeleton**, che contiene la parte relativa alla schermata di caricamento dei messaggi.

In fine, nel nostro progetto sviluppato con React, abbiamo utilizzato anche **Zustand**, che è una libreria di gestione dello stato per applicazioni React, ovvero il processo di mantenimento delle informazioni che cambiano nel tempo e che determinano come l'interfaccia utente viene renderizzata e come si comporta. Lo stato può includere qualsiasi cosa, dai dati dell'utente, ai risultati di una chiamata API, ai valori dei campi di input e altro ancora.

Insieme a React, per lo sviluppo della UI abbiamo utilizzato **TailwindCSS**, un framework CSS progettato per facilitare la creazione di interfacce utente personalizzabili. La comodità del suo utilizzo sta nel fatto che esso è altamente configurabile e personalizzabile e che non fornisce componenti UI predefiniti, ma è possibile prototipare rapidamente componenti direttamente nel JSX, senza scrivere CSS aggiuntivo.

### 3.1.2 Messaggistica in tempo reale: Socket.IO

Socket.IO è una libreria per Node.js che consente di abilitare una comunicazione bidirezionale in tempo reale tra due utenti senza dover aggiornare la pagina web.

**Logica di funzionamento:**

- 1) **Connessione:** Il client si connette al server Socket.IO.
- 2) **Scambio di eventi:** Entrambi possono emettere e ascoltare eventi.

```
// io.emit() è usato per mandare events a tutti gli utenti connessi
io.emit('getOnlineUsers', Object.keys(userSocketMap));

// socket.on() è usata per ascoltare gli utenti. può essere usata sia lato client che server
socket.on('disconnect', () => {
  console.log('user disconnected', socket.id);
  delete userSocketMap[userId];
  io.emit('getOnlineUsers', Object.keys(userSocketMap));
});
```

In questo esempio, possiamo notare l'emissione di un evento tramite *io.emit()* e l'ascolto di un evento tramite *socket.on()*.

- 3) **Disconnessione:** Il client può disconnettersi e il server può gestire questa disconnessione.

## 3.2 Architettura a microservizi

In questa parte dell'elaborato esploreremo l'architettura a microservizi utilizzata per la nostra web app. L'architettura a microservizi è un approccio che scompone un'applicazione monolitica in una serie di servizi indipendenti e interconnessi. Ogni microservizio è responsabile di una funzione specifica e può essere sviluppato, distribuito e scalato in modo indipendente dagli altri. Questo approccio offre numerosi vantaggi, tra cui maggiore flessibilità, facilità di manutenzione e scalabilità.

L'architettura a microservizi scompone l'applicazione in unità più piccole e autonome. Questa struttura consente di migliorare l'agilità del team di sviluppo, poiché i microservizi possono essere sviluppati e distribuiti indipendentemente. Inoltre, permette una maggiore

resilienza del sistema: un problema in un microservizio non compromette necessariamente l'intera applicazione. La nostra web app è stata suddivisa in tre microservizi principali, ognuno dei quali svolge un ruolo specifico:

- Authentication-Service
- Message-Service
- User-Service

Vediamoli nello specifico:

### **Authentication-Service**

Il microservizio di autenticazione si occupa di gestire tutto ciò che riguarda l'autenticazione e l'autorizzazione degli utenti. Include funzionalità come la registrazione, il login, la gestione delle sessioni e la verifica dei token JWT. Utilizzando un servizio dedicato per l'autenticazione, possiamo garantire che queste operazioni critiche siano sicure e scalabili, riducendo il carico sui servizi principali dell'applicazione.

### **Message-Service**

Il microservizio di messaggistica gestisce lo scambio di messaggi tra gli utenti. È responsabile della creazione, dell'invio, della ricezione e dell'archiviazione dei messaggi. Questo servizio è progettato per essere altamente scalabile, in modo da poter gestire un alto volume di traffico e garantire tempi di risposta rapidi, indipendentemente dal numero di utenti che utilizzano il sistema simultaneamente.

### **User-Service**

Il microservizio per la gestione degli utenti si occupa di tutte le operazioni relative agli utenti, ad eccezione dell'autenticazione. Separando queste responsabilità dal microservizio di autenticazione, possiamo mantenere una chiara separazione delle preoccupazioni, facilitando la manutenzione e l'aggiornamento delle singole componenti.

L'adozione di un'architettura a microservizi ha portato numerosi benefici alla nostra web app, tra cui una maggiore flessibilità nello sviluppo, una migliore scalabilità e una resilienza complessiva del sistema. Ogni microservizio può essere sviluppato, distribuito e scalato indipendentemente, permettendo al team di rispondere rapidamente alle esigenze degli utenti e ai cambiamenti del mercato. Questo approccio ci ha permesso di creare una piattaforma robusta e adattabile, capace di evolversi con le esigenze future.

### 3.2.1 Utilizzo di Docker per la containerizzazione dei microservizi

Questi microservizi sono stati opportunamente dockerizzati, utilizzando Docker per la gestione dei container. Docker permette di impacchettare ogni microservizio insieme alle sue dipendenze software in un container leggero e isolato, garantendo che possano essere eseguiti in modo consistente su diversi ambienti di sviluppo, test e produzione. Questo approccio facilita la distribuzione dei microservizi e assicura coerenza tra gli ambienti, contribuendo alla stabilità e alla scalabilità dell'architettura a microservizi adottata per la nostra web app.



















<input type="checkbox"/>	Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	 <a href="#">peaceful_albattani</a> 586cc0b4d1b6 	<a href="#">user-service</a>	Running	<a href="#">10000:10000</a> 	0.01%	25 seconds ago	  
<input type="checkbox"/>	 <a href="#">optimistic_murdock</a> a7e12fbc72b8 	<a href="#">message-service</a>	Running	<a href="#">8000:8000</a> 	0.01%	24 seconds ago	  
<input type="checkbox"/>	 <a href="#">eloquent_galois</a> 85ac21c36b46 	<a href="#">authentication-service</a>	Running	<a href="#">8090:8090</a> 	0.01%	23 seconds ago	  

Figura 18 - Containerizzazione dei microservizi tramite Docker

## 3.3 Pattern MVC

Per la strutturazione dell'applicazione 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.

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;

**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.

### **Vantaggi del MVC:**

La scelta di tale Pattern Architettuale 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 programmatori.
- 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.

## 3.4 Diagrammi

### 3.4.1 Diagramma dei componenti

Il diagramma dei componenti serve per avere una visione d'insieme del sistema implementato in modo tale da documentare le relazioni tra i vari singoli componenti individuati e sottolineare anche le interfacce che vengono esposte per rendere possibile la comunicazione fra essi. I componenti rappresentati incapsulano al loro interno tutte le strutture complesse che li caratterizzano, mentre i contatti con gli altri componenti, che ne permettono, di fatto, la comunicazione, è possibile solo tramite l'uso di interfacce.

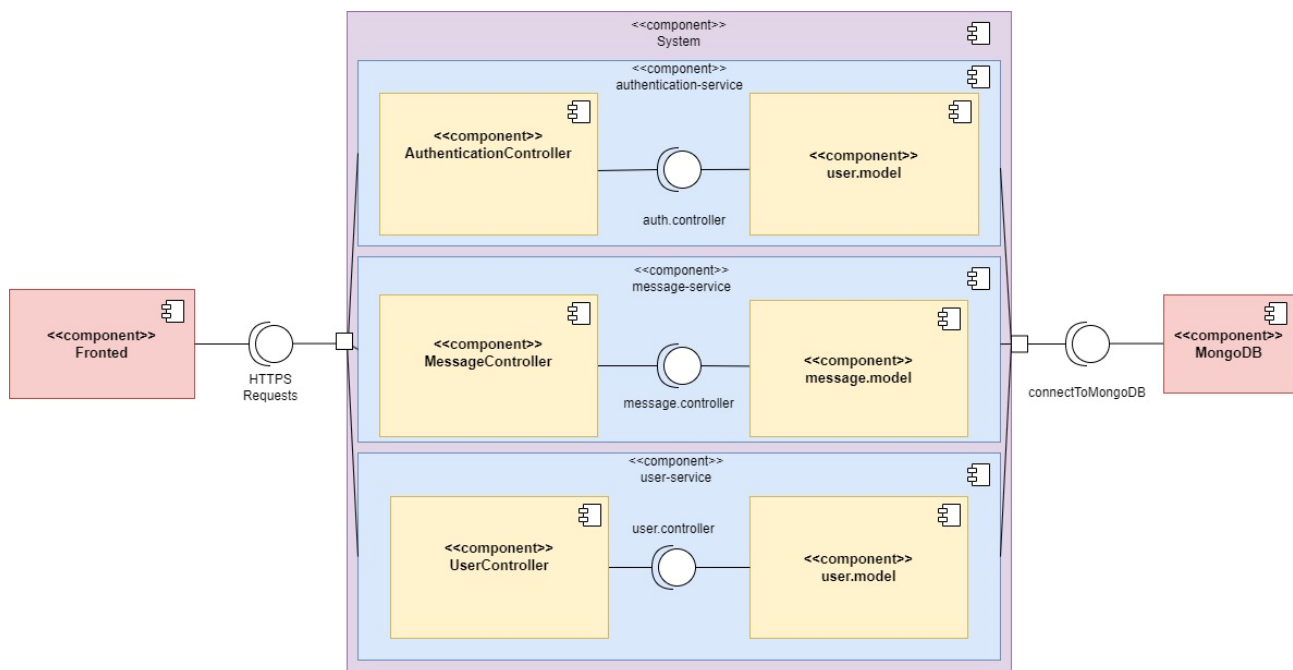


Figura 19: Diagramma dei componenti

### 3.4.2 Diagrammi di Sequenza Raffinati

I diagrammi di sequenza raffinati in UML sono strumenti potenti per visualizzare e analizzare le interazioni dettagliate tra gli oggetti all'interno di un sistema software. Questi diagrammi rappresentano il flusso di messaggi e le operazioni che si verificano nel tempo, offrendo una visione chiara e precisa di come i componenti del sistema collaborano per eseguire una funzione specifica.

In questo capitolo, ci concentreremo su come i diagrammi di sequenza raffinati possono essere utilizzati per modellare le interazioni complesse all'interno della nostra architettura.

Andiamo a vederli nello specifico:

### 3.4.2.1 Registrazione

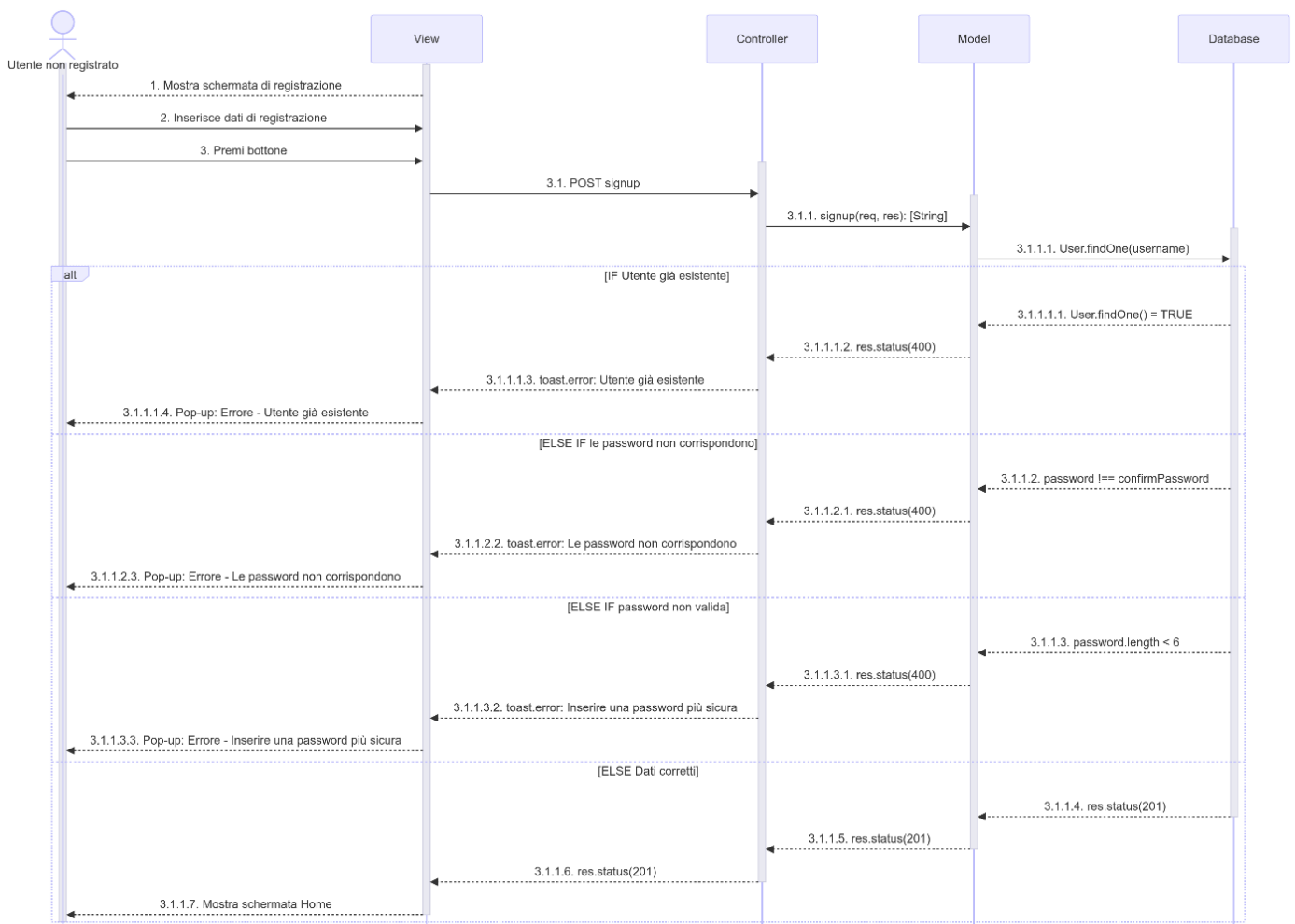


Figura 20: Registrazione - Diagramma di sequenza raffinato



### 3.4.2.2 Login

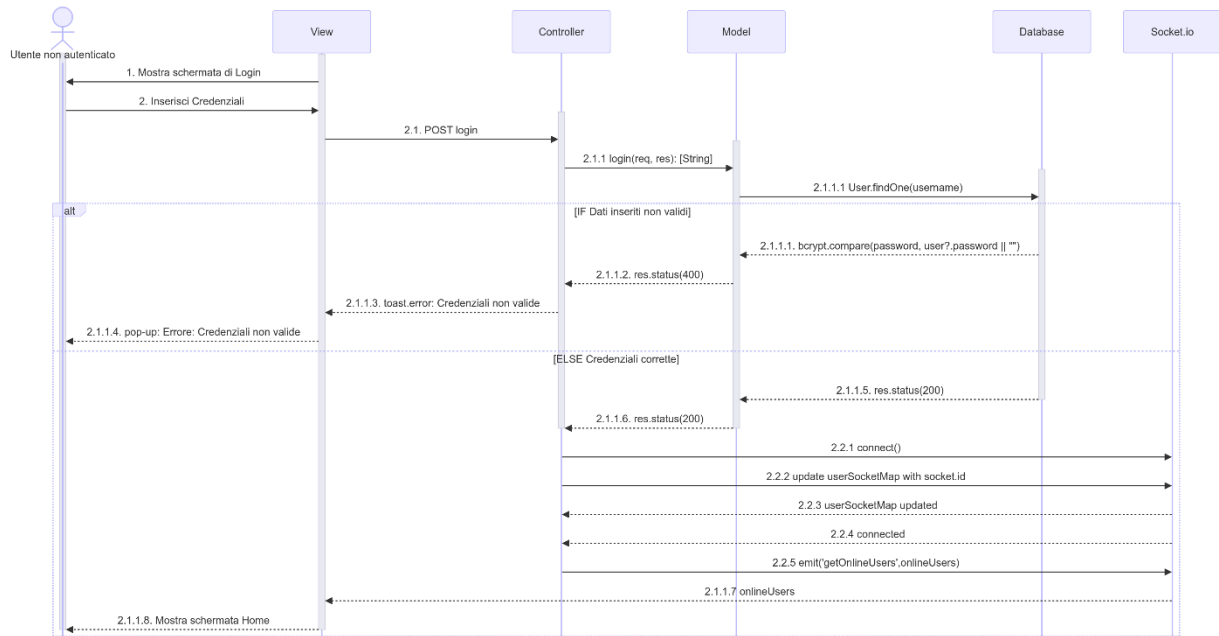


Figura 21: Login - Diagramma di sequenza raffinato

### 3.4.2.3 Ricerca Chat

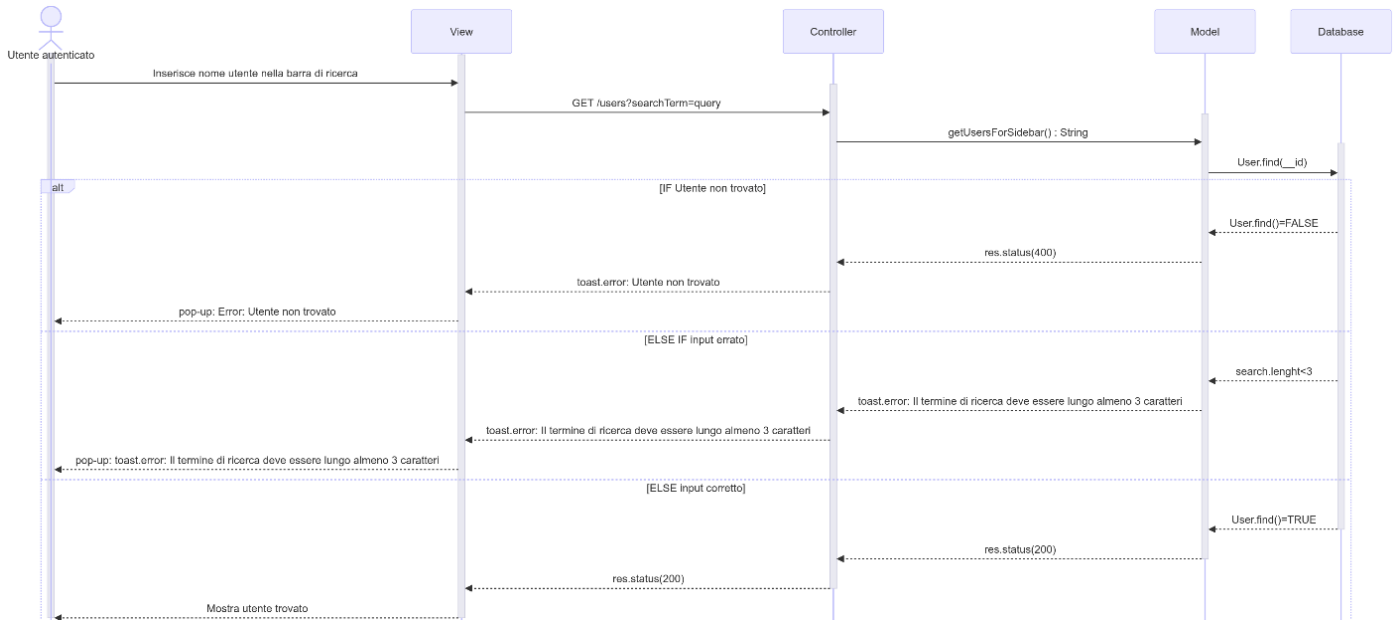


Figura 22: Ricerca Chat - Diagramma di sequenza raffinato

### 3.4.2.4 Invio/Ricezione Messaggio

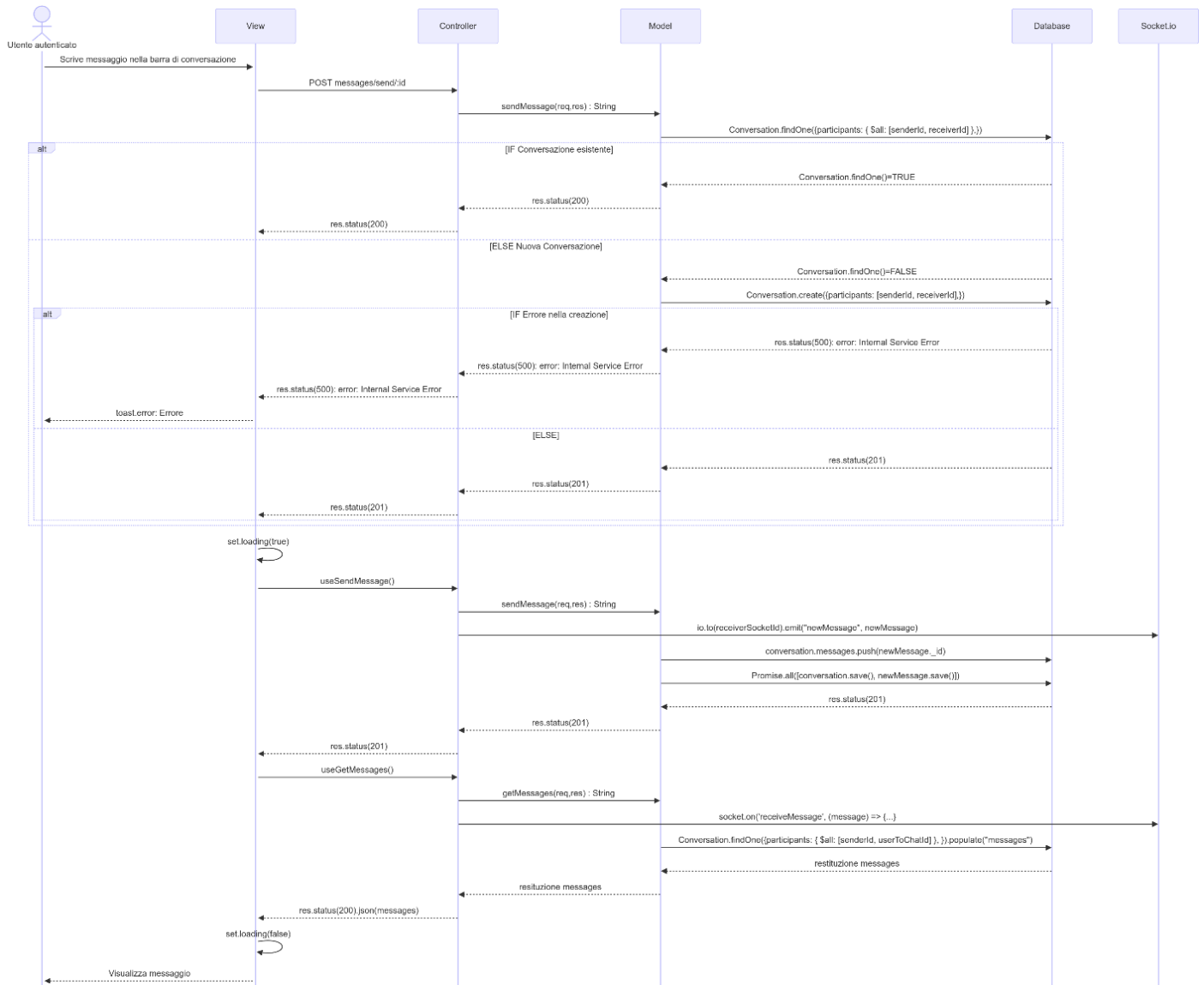


Figura 23: Invio/Ricezione Messaggio – Diagramma di sequenza raffinato

Per questioni di leggibilità, abbiamo scisso la Figura 24 in due immagini:

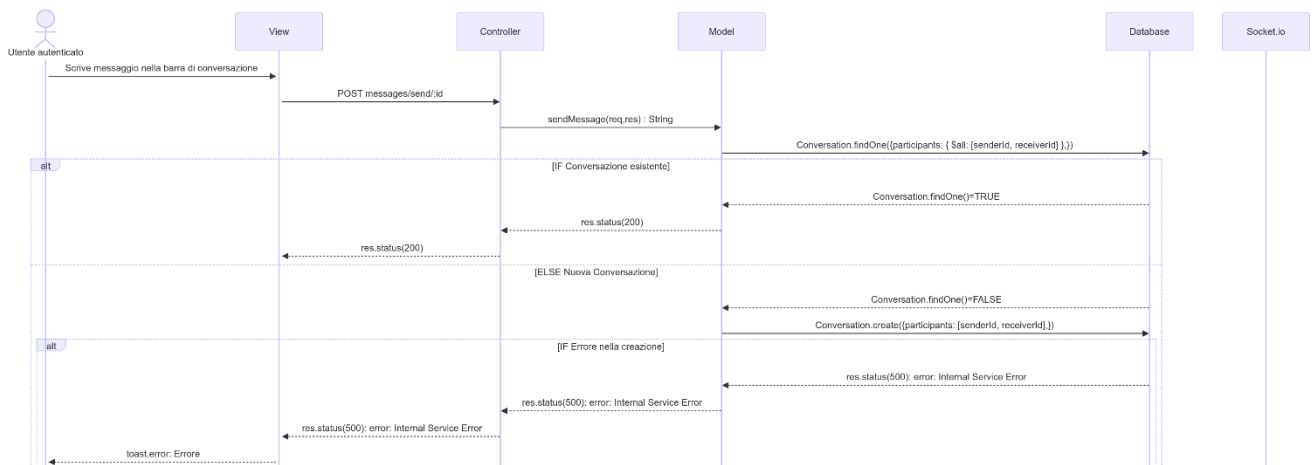


Figura 24 : Diagramma di sequenza raffinato Invio/Ricevi Messaggio -PARTE 1

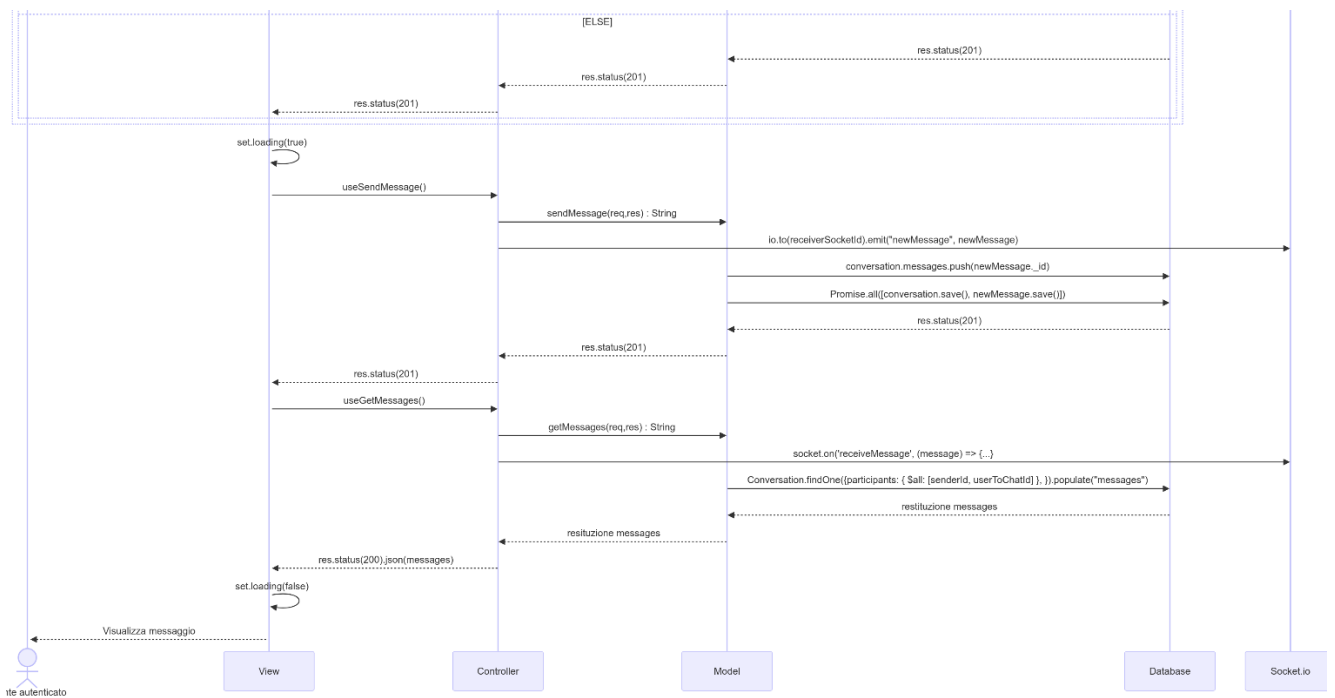


Figura 25 : Diagramma di sequenza raffinato Invio/Ricevi Messaggio -PARTE 2

### 3.4.2.5 Logout

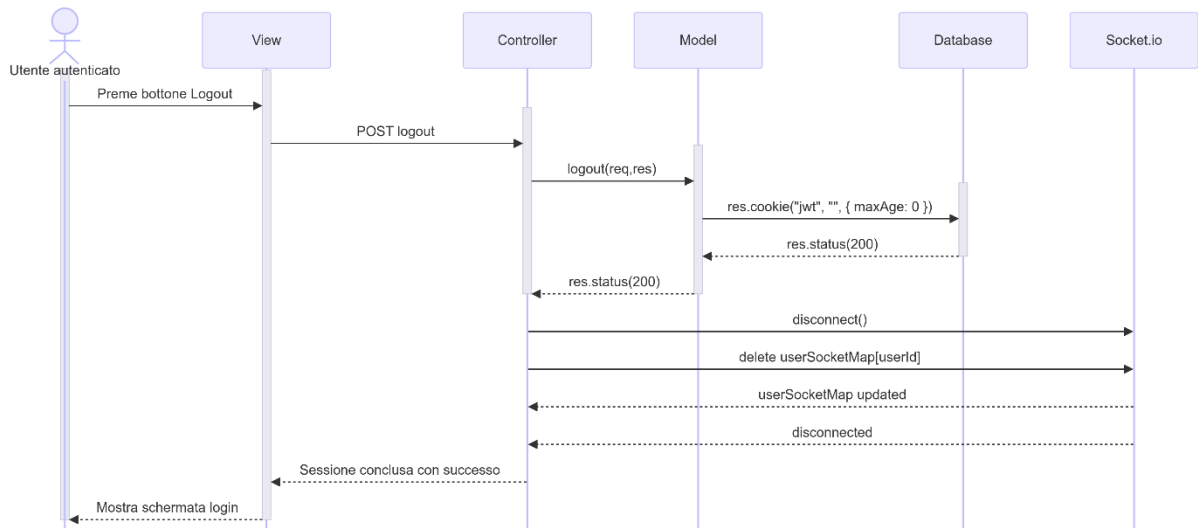


Figura 26: Logout – Diagramma di sequenza raffinato

### 3.4.3 Diagramma dei package

Il diagramma dei package consente di avere una visualizzazione logica di come è composta l'architettura del nostro sistema.

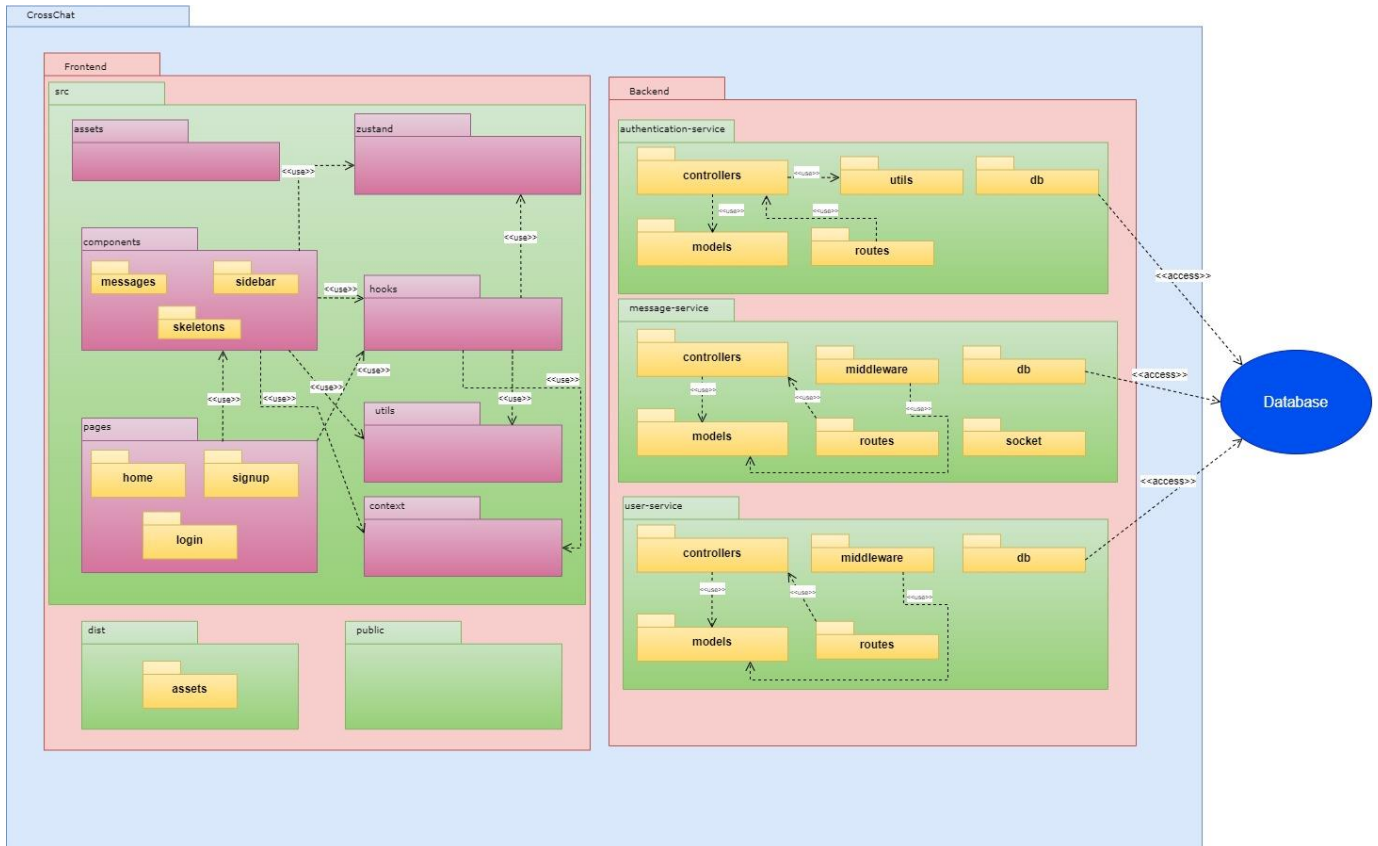


Figura 27 - Diagramma dei package

## 1. Panoramica del Frontend

- **Scopo:** Gestisce l'interfaccia utente e l'esperienza utente dell'applicazione.
- **Struttura:** Diviso in vari sotto-pacchetti come **assets**, **components**, **pages**, **utils**, **hooks**, **context** e **zustand**.

### Pacchetti nel Frontend

- **src/assets**
  - **Scopo:** Contiene file statici come immagini, font e fogli di stile.
  - **Componenti:**
    - Cartella assets per organizzare tutte le risorse statiche.
- **src/components**

- **Scopo:** Contiene componenti UI riutilizzabili.
- **Componenti:**
  - **messages:** Gestisce la visualizzazione dei messaggi di chat.
  - **sidebar:** Contiene gli elementi dell'interfaccia della barra laterale.
  - **skeletons:** Contiene i caricamenti scheletro per migliorare l'esperienza utente durante il recupero dei dati.
- **src/pages**
  - **Scopo:** Contiene le diverse pagine dell'applicazione.
  - **Componenti:**
    - **home:** Pagina principale.
    - **signup:** Pagina di registrazione dell'utente.
    - **login:** Pagina di accesso dell'utente.
- **src/zustand**
  - **Scopo:** Gestisce lo stato globale dell'applicazione utilizzando la libreria Zustand.
  - **Componenti:**
    - **useConversation:** Hook per gestire lo stato delle conversazioni.
- **src/hooks**
  - **Scopo:** Contiene hook personalizzati di React per la logica riutilizzabile.
  - **Componenti:**
    - **useGetConversation:** Recupera le conversazioni.
    - **useGetMessage:** Recupera i messaggi.
    - **useListenMessages:** Ascolta i nuovi messaggi.
    - **useLogin:** Gestisce il login dell'utente.
    - **useLogout:** Gestisce il logout dell'utente.
    - **useSendMessage:** Invia messaggi.
    - **useSignup:** Gestisce la registrazione dell'utente.
- **src/utls**
  - **Scopo:** Contiene funzioni di utilità.

- **Componenti:**
  - **emojis:** Per l'utilizzo degli emoji nella schermata home.
  - **extractTime:** Estrae e formatta le informazioni temporali dai dati.
- **src/context**
  - **Scopo:** Gestisce lo stato globale utilizzando l'API Context di React.
  - **Componenti:**
    - **AuthContext:** Gestisce lo stato di autenticazione.
    - **SocketContext:** Gestisce lo stato e la connessione del socket.

## 2. Panoramica del Backend

- **Scopo:** Gestisce la logica lato server e le interazioni con il database.
- **Struttura:** Diviso in microservizi come authentication-service, message-service e user-service.

### Pacchetti nel Backend

- **authentication-service**
  - **Scopo:** Gestisce l'autenticazione e l'autorizzazione degli utenti.
  - **Componenti:**
    - **controllers:** Definiscono i gestori delle richieste.
    - **models:** Definiscono i modelli di dati e gli schemi.
    - **routes:** Definiscono le rotte API.
    - **utils:** Funzioni di utilità per l'autenticazione.
    - **db:** Logica di interazione con il database.
- **message-service**
  - **Scopo:** Gestisce le funzionalità relative ai messaggi.
  - **Componenti:**
    - **controllers:** Definiscono i gestori delle richieste per la messaggistica.
    - **models:** Definiscono i modelli di dati dei messaggi.
    - **routes:** Definiscono le rotte API per la messaggistica.
    - **middleware:** Middleware per l'elaborazione dei messaggi.

- **socket:** Gestisce le connessioni WebSocket.
  - **db:** Logica di interazione con il database.
- **user-service**
  - **Scopo:** Gestisce le funzionalità relative agli utenti.
  - **Componenti:**
    - **controllers:** Definiscono i gestori delle richieste per la gestione degli utenti.
    - **models:** Definiscono i modelli di dati degli utenti.
    - **routes:** Definiscono le rotte API relative agli utenti.
    - **middleware:** Middleware per l'elaborazione degli utenti.
    - **db:** Logica di interazione con il database.

### 3. Database

- **Scopo:** Archivio centralizzato per tutti i dati dell'applicazione.
- **Struttura:** Interfacciato dai vari servizi backend per memorizzare e recuperare i dati.



### 3.4.5 Diagramma delle classi

Il diagramma delle classi UML rappresenta la struttura statica di un sistema, mostrando le classi, i loro attributi, metodi e le relazioni tra di esse, come ereditarietà, associazioni e dipendenze. Questo diagramma è fondamentale per la progettazione orientata agli oggetti, poiché aiuta a visualizzare e organizzare le componenti del software.

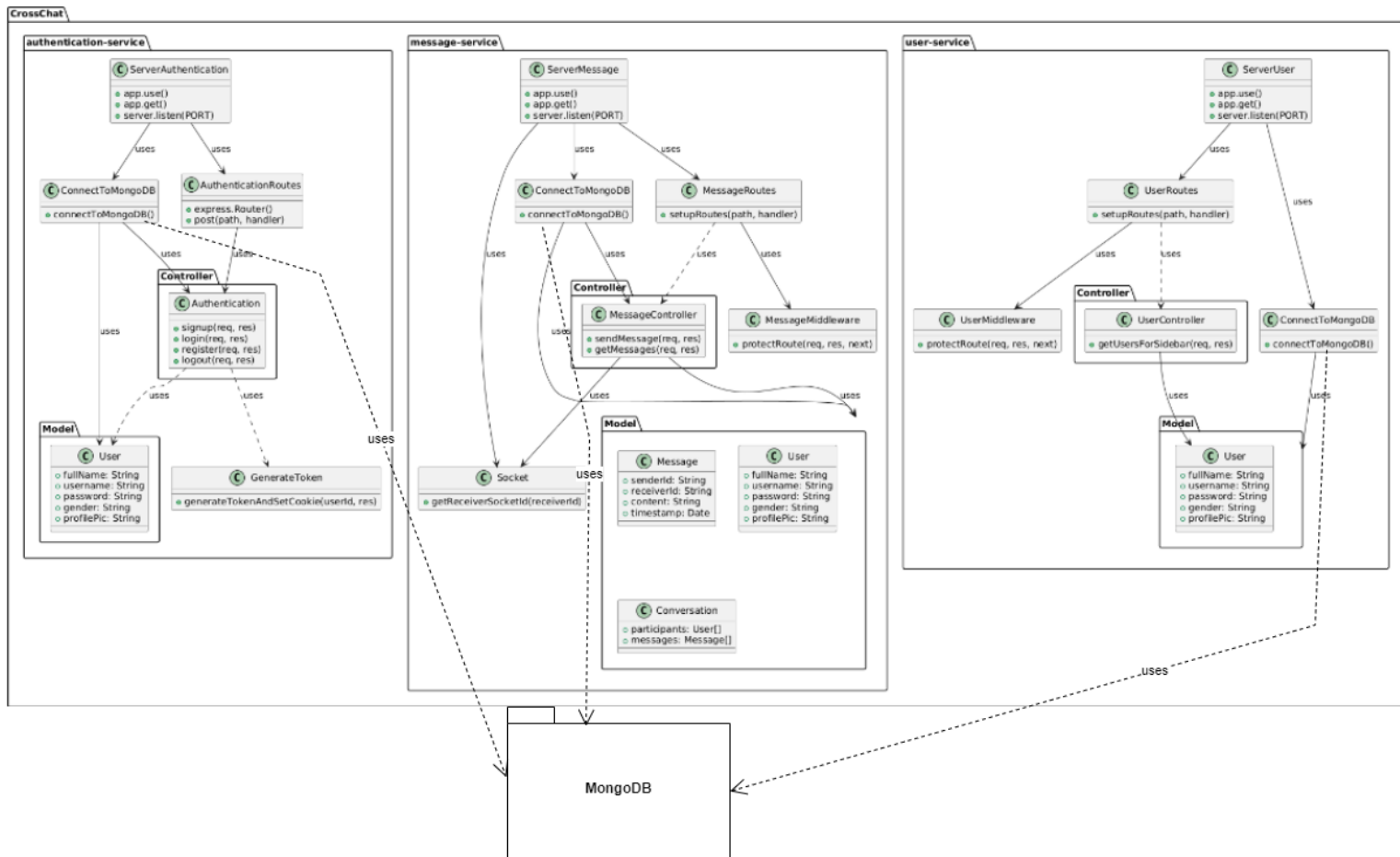


Figura 28 - Diagramma delle Classi

Il diagramma delle classi rappresenta l'architettura di "CrossChat", suddivisa in tre servizi principali: authentication-service, message-service e user-service. Ogni servizio è autonomo e comunica con un database MongoDB comune.

## 1. Authentication Service

Il servizio di autenticazione gestisce la registrazione, l'accesso e il logout degli utenti. È composto dalle seguenti componenti:

- **ServerAuthentication:**
  - Configura il server e imposta le rotte utilizzando `app.use()`, `app.get()` e `server.listen(PORT)`.
- **ConnectToMongoDB:**
  - Gestisce la connessione al database MongoDB tramite la funzione `connectToMongoDB()`.
- **AuthenticationRoutes:**
  - Definisce le rotte per l'autenticazione usando `express.Router()` e `post(path, handler)`.
- **Controller (Authentication):**
  - Implementa le funzionalità di autenticazione:
    - `signup(req, res)`: Gestisce la registrazione di un nuovo utente.
    - `login(req, res)`: Gestisce l'accesso di un utente esistente.
    - `logout(req, res)`: Gestisce la disconnessione dell'utente.

- **Model (User):**
  - Definisce il modello utente con attributi come fullName, username, password, gender, e profilePic.
- **GenerateToken:**
  - Gestisce la generazione e impostazione del cookie del token (generateTokenAndSetCookie(userId, res)).

## 2. Message Service

Il servizio di messaggistica gestisce l'invio e la ricezione dei messaggi. È composto dalle seguenti componenti:

- **ServerMessage:**
  - Configura il server per il servizio di messaggistica utilizzando app.use(), app.get() e server.listen(PORT).
- **ConnectToMongoDB:**
  - Gestisce la connessione al database MongoDB tramite la funzione connectToMongoDB().
- **MessageRoutes:**
  - Definisce le rotte per la gestione dei messaggi (setupRoutes(path, handler)).
- **Controller (MessageController):**
  - Implementa le funzionalità di gestione dei messaggi:

- `sendMessage(req, res)`: Gestisce l'invio di un messaggio.
- `getMessages(req, res)`: Recupera i messaggi.
- **MessageMiddleware:**
  - Protegge le rotte (`protectRoute(req, res, next)`).
- **Model (Message, User, Conversation):**
  - Definisce i modelli di messaggio (`senderId, receiverId, content, timestamp`), utente (`fullName, username, password, gender, profilePic`), e conversazione (`participants, messages`).
- **Socket:**
  - Gestisce le connessioni socket per la messaggistica in tempo reale (`getReceiverSocketId(receiverId)`).

### 3. User Service

Il servizio utenti gestisce la gestione dei profili utente. È composto dalle seguenti componenti:

- **ServerUser:**
  - Configura il server per il servizio utenti utilizzando `app.use()`, `app.get()` e `server.listen(PORT)`.

- **ConnectToMongoDB:**
  - Gestisce la connessione al database MongoDB tramite la funzione `connectToMongoDB()`.
- **UserRoutes:**
  - Definisce le rotte per la gestione degli utenti (`setupRoutes(path, handler)`).
- **Controller (UserController):**
  - Implementa le funzionalità di gestione degli utenti:
    - `getUsersForSidebar(req, res)`: Recupera gli utenti per la sidebar.
- **UserMiddleware:**
  - Protegge le rotte (`protectRoute(req, res, next)`).
- **Model (User):**
  - Definisce il modello utente con attributi come `fullName`, `username`, `password`, `gender`, e `profilePic`.

## Interazioni

- Ogni servizio comunica con il database MongoDB per gestire i dati degli utenti, dei messaggi e delle conversazioni.
- Le rotte e i controller nei vari servizi sono responsabili di eseguire operazioni specifiche e di manipolare i dati tramite i modelli definiti.

- Il middleware protegge le rotte sensibili assicurando che solo gli utenti autenticati possano accedere a certe funzionalità.
- I socket sono utilizzati nel servizio di messaggistica per gestire la comunicazione in tempo reale.

### 3.4.6 Diagramma di deployment

Il diagramma di deployment mostra come i componenti software sono distribuiti su nodi hardware. Esso rappresenta dove e come le applicazioni vengono eseguite nel sistema.

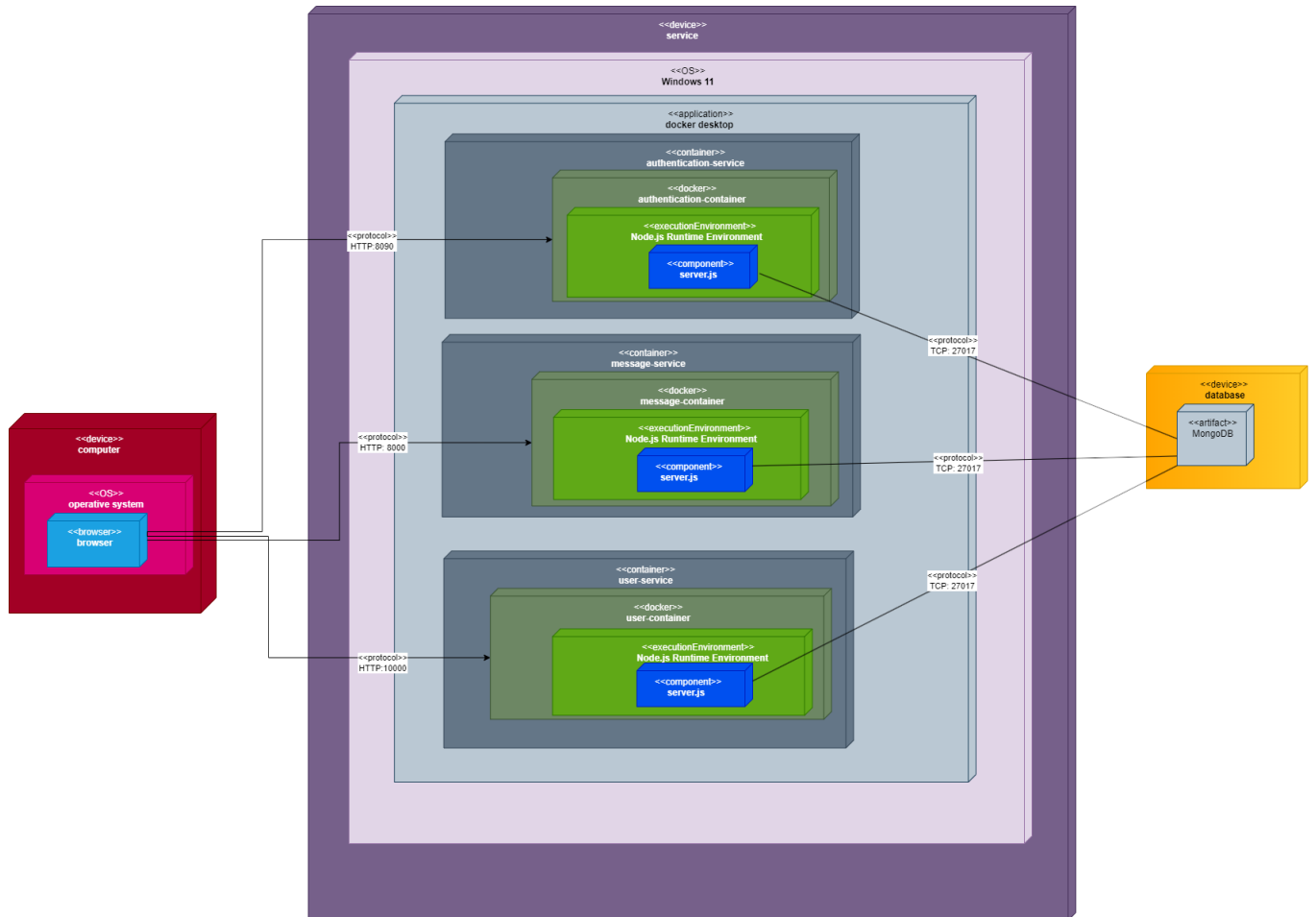


Figura 29: Diagramma deployment

## Capitolo 4: Manuale di Installazione

### 4.1 Installazione

```
## Setup del file .env per ogni servizio del backend e del frontend con porte
diverse

```js
PORT=...
MONGO_DB_URI=...
JWT_SECRET=...
NODE_ENV=...
```

## Installare le dipendenze per ogni servizio e avviarli

### authentication-service

```shell
cd backend/authentication-service
npm install
npm start
```

### message-service

```shell
cd backend/message-service
npm install
npm start
```

### user-service

```shell
cd backend/user-service
npm install
npm start
```

### frontend

```shell
cd frontend
npm install
npm run dev
```
```



## 4.2 Installazione alternativa con Docker:

### ## Setup con Docker

In alternativa è possibile eseguire il setup del progetto con Docker nel seguente modo:

#### ### authentication-service

```
```shell
cd backend/authentication-service
docker build -t authentication-service .
docker run -p [port]:[port] -d authentication-service
```
```

#### ### message-service

```
```shell
cd backend/message-service
docker build -t message-service .
docker run -p [port]:[port] -d message-service
```
```

#### ### user-service

```
```shell
cd backend/user-service
docker build -t user-service .
docker run -p [port]:[port] -d user-service
```
```

#### ### frontend

```
```shell
cd frontend
npm install
npm run dev
```
```

## Capitolo 5: Testing e Sicurezza

In questo capitolo esploreremo due aspetti fondamentali per il successo e l'affidabilità della nostra web app: il **testing** e la **sicurezza**.

Il testing, nello specifico i test end-to-end (E2E) effettuati con Cypress, è un processo cruciale che assicura che tutte le funzionalità dell'applicazione funzionino come previsto, riducendo al minimo i bug e migliorando l'esperienza utente. Illustreremo come Cypress sia stato utilizzato per automatizzare e semplificare il processo di test, garantendo la robustezza e la qualità del nostro software.

Parallelamente, discuteremo delle pratiche di sicurezza implementate per proteggere la web app da potenziali minacce e vulnerabilità. In particolare, esamineremo l'uso dei token JWT (JSON Web Token) per la gestione sicura dell'autenticazione e dell'autorizzazione degli utenti. JWT è un metodo per rappresentare informazioni tra due parti in modo sicuro e compatto.

Parleremo anche della libreria bcrypt utilizzata per l'hashing delle password per evitare attacchi di forza bruta sulle password.

### 5.1 Testing e2e tramite Cypress

Nel nostro elaborato abbiamo utilizzato Cypress, un framework open-source di testing end-to-end per web applications. Tramite questi test siamo andati a simulare il comportamento di un utente reale che naviga le pagine del sito.

#### 5.1.1 Come funziona Cypress?

Cypress è installabile come pacchetto Npm e per farlo è necessaria una versione recente di Node.js. A questo punto basta aprire un terminale all'interno della cartella del tuo progetto e lanciare il seguente comando:

```
npm install cypress
```

Una volta fatto ciò, basterà lanciare il seguente comando:

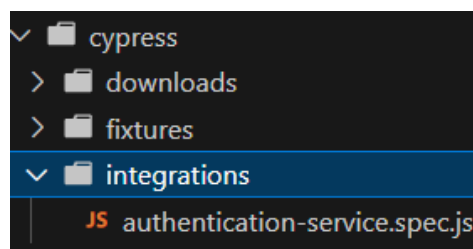
```
npx cypress open
```

per aprire l'interfaccia grafica di Cypress, e generare all'interno della directory in cui si è installato il framework un file di configurazione denominato ***cypress.config.js***.

Questo file ci permette di inserire al suo interno alcune configurazioni che ci semplificheranno la vita durante la creazione dei nostri test.

Ad esempio, possiamo informare Cypress, attraverso questo file di configurazione, che andremo a testare API che hanno come *URL Base* il nostro *http://localhost:{PORT}* così da non dover ogni volta riscrivere da zero il percorso delle pagine web che abbiamo intenzione di testare.

Tra i folder generati nella cartella "cypress", quello su cui ci focalizziamo è "**integrations**", all'interno del quale troviamo un file del tipo ***\*.spec.js***, dove comunicheremo a Cypress le operazioni da simulare.



Definiti i casi di test all'interno di questo file, basterà lanciare il testing di Cypress, da terminale, tramite il comando:

```
npx cypress run
```

o in alternativa, eseguirlo dall'interfaccia grafica di Cypress tramite *npx cypress open*.

### 5.1.2 Implementazione dei casi di test per ogni microservizio

Applicando il procedimento precedente per ogni folder, siamo andati quindi ad implementare i casi di test specificatamente per ogni microservizio indicato: *authentication-service*, *message-service*, *user-service*.

#### 5.1.2.1 Implementazione dei casi di test per authentication-service

Per quanto riguarda authentication-service:

Siamo andati a considerare i seguenti casi di test:

1. Devo registrare un nuovo utente con successo
2. Da utente registrato voglio poter loggare con successo
3. Voglio poter effettuare un logout con successo

Di seguito l'implementazione del file ***authentication-service.spec.js***:

```
// cypress/authentication-service.spec.js
describe('Authentication Service API', () => {

  it('Devo registrare un nuovo utente con successo', () => {
    cy.request('POST', `${Cypress.env('apiBaseUrl')}/signup`, {
      fullName: 'Test User',
      username: 'testuser',
      password: 'password123',
      confirmPassword: 'password123',
      gender: 'male'
    }).then((response) => {
      expect(response.status).to.eq(201); //201 - Created: significa che la
      richiesta ha avuto successo e il server ha creato una nuova risorsa
      expect(response.body).to.have.property('_id');
      expect(response.body).to.have.property('username', 'testuser');
      expect(response.body).to.have.property('fullName', 'Test User');
    });
  });

  it('Da utente registrato voglio poter loggare con successo', () => {
    cy.request('POST', `${Cypress.env('apiBaseUrl')}/login`, {
      username: 'testuser',
      password: 'password123'
    }).then((response) => {
      expect(response.status).to.eq(200); //Codice di stato 200: OK.
      expect(response.body).to.have.property('_id'); //verifica che il corpo
      della risposta (response.body) contenga una proprietà chiamata _id.
      expect(response.body).to.have.property('username', 'testuser'); //verifica
      che il corpo della risposta (response.body) contenga una proprietà chiamata
      username.
    });
  });

  it('Voglio poter effettuare un logout con successo', () => {
    cy.request('POST', `${Cypress.env('apiBaseUrl')}/logout`, {
      username: 'testuser',
      password: 'password123'
    }).then(() => {
      cy.request('POST', `${Cypress.env('apiBaseUrl')}/logout`).then((response)
=> {
```

```

        expect(response.status).to.eq(200);
        expect(response.body).to.have.property('message', 'Logged out
successfully');
    });
});
});
});

```

Di seguito il relativo ***cypress.config.js***:

```

import { defineConfig } from 'cypress';

export default defineConfig({
  e2e: {
    baseUrl: 'http://localhost:8090',
    setupNodeEvents(on, config) {
      // Puoi aggiungere ascoltatori di eventi qui, ad esempio:

      // Puoi aggiungere altre configurazioni o plugin qui
    },
    specPattern: 'cypress/integrations/*.spec.js',
    supportFile: 'cypress/support/e2e.js'
  },
  env: {
    apiBaseUrl: 'http://localhost:8090/api/auth'
  },
  fixturesFolder: 'cypress/fixtures'
});

```

### 5.1.2.2 Implementazione dei casi di test per message-service

#### Per quanto riguarda message-service:

Siamo andati a considerare i seguenti casi di test:

1. Dovrebbe inviare un messaggio con successo
2. Dovrebbe ottenere i messaggi con successo

Di seguito l'implementazione del file ***message-service.spec.js***:

```

describe('Message Service API', () => {
  let authToken = null;

  const receiverId = '668d01cd93b88ac186be2d71'; // Receiver ID for testing

  before(() => {
    cy.request({
      method: 'POST',
      url: 'http://localhost:8090/api/auth/login',

```

```

    body: {
      username: 'user-test',
      password: 'grupposad',
    },
  }).then((response) => {
    expect(response.status).to.eq(200);
    authToken = response.body.token;
    cy.log('Auth Token Acquired');
    console.log('Auth Token Acquired:', authToken);
  });
});

it('Dovrebbe inviare un messaggio con successo', () => {
  cy.request({
    method: 'POST',
    url: `${Cypress.env('apiBaseUrl')}/${receiverId}`,
    headers: {
      'Authorization': `Bearer ${authToken}`,
      'Content-Type': 'application/json'
    },
    body: {
      message: 'Test message from Cypress'
    },
    failOnStatusCode: false
  }).then((response) => {
    if (response.status !== 201) {
      // Gestione personalizzata dell'errore se non è stato restituito un 201
      cy.log(`Errore durante l'invio del messaggio: ${response.status}`);
    } else {
      // Aspettiamo un codice di stato 201 e verifichiamo la risposta
      expect(response.status).to.eq(201);
      expect(response.body).to.have.property('_id');
      expect(response.body).to.have.property('senderId');
      expect(response.body).to.have.property('receiverId', receiverId);
      expect(response.body).to.have.property('message', 'Test message from
Cypress');
    }
  });
});

it('Dovrebbe ricevere i messaggi con successo', () => {
  cy.request({
    method: 'GET',
    url: `${Cypress.env('apiBaseUrl')}/${receiverId}`,
    headers: {
      'Authorization': `Bearer ${authToken}`,
      'Content-Type': 'application/json'
    },
    failOnStatusCode: false
  }).then((response) => {
    if (response.status === 401) {

```

```
// Gestione caso di non autorizzazione
cy.log('Non autorizzato. Verificare il token JWT o l\'autorizzazione.');
```

```
// Puoi anche eseguire altre azioni o asserzioni qui in caso di 401
} else if (response.status === 404) {
  // Gestione caso di risorsa non trovata
  cy.log('Endpoint non trovato. Assicurati che l\'URL sia corretto.');
```

```
// Puoi anche eseguire altre azioni o asserzioni qui in caso di 404
} else {
  // Aspettiamo un codice di stato 200 e verifichiamo la risposta
  expect(response.status).to.eq(200);
  expect(response.body).to.be.an('array').that.is.not.empty;
  response.body.forEach((msg) => {
    expect(msg).to.have.property('_id');
```

```
    expect(msg).to.have.property('senderId');
```

```
    expect(msg).to.have.property('receiverId', receiverId);
  });
}
});
});
});
```

Di seguito il relativo ***cypress.config.js***:

```
import { defineConfig } from 'cypress';

export default defineConfig({
  e2e: {
    baseUrl: 'http://localhost:8000',
    setupNodeEvents(on, config) {
      // Puoi aggiungere ascoltatori di eventi qui, ad esempio:

      // Puoi aggiungere altre configurazioni o plugin qui
    },
    specPattern: 'cypress/integrations/*.spec.js',
    supportFile: 'cypress/support/e2e.js'
  },
  env: {
    apiBaseUrl: 'http://localhost:8000/api/messages'
  },
  fixturesFolder: 'cypress/fixtures'
});
```

### 5.1.2.3 Implementazione dei casi di test per user-service

#### Per quanto riguarda user-service:

Siamo andati a considerare il seguente caso di test:

1. Dovrebbe ritornare una lista di utenti con successo

Di seguito l'implementazione del file ***user-service.spec.js***:

```
describe('API /api/users', () => {
  let authToken = null;

  before(() => {
    cy.request({
      method: 'POST',
      url: 'http://localhost:8090/api/auth/login',
      body: {
        username: 'user-test',
        password: 'grupposad',
      },
    }).then((response) => {
      expect(response.status).to.eq(200);
      authToken = response.body.token;
      cy.log('Auth Token Acquired');
      console.log('Auth Token Acquired:', authToken);
    });
  });

  it('Dovrebbe ritornare una lista di utenti', () => {
    cy.request({
      method: 'GET',
      url: 'http://localhost:10000/api/users',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${authToken}`,
      },
    }).then((response) => {
      expect(response.status).to.eq(200);
      expect(response.body).to.have.length.greaterThan(0);
      expect(response.body[0]).to.have.property('username');
    });
    //per visualizzare la lista di utenti guardalo da npx cypress open, dalla gui del browser
    cy.log('Number of users: ' + response.body.length);
    cy.log('First user: ' + JSON.stringify(response.body[0]));
    cy.log('All users: ' + JSON.stringify(response.body));

    console.log('Number of users:', response.body.length);
    console.log('First user:', response.body[0]);
  });
});
```



```
    console.log('All users:', response.body);
  });
});
});
```

Di seguito il relativo ***cypress.config.js***:

```
import { defineConfig } from 'cypress';

export default defineConfig({
  e2e: {
    baseUrl: 'http://localhost:10000',
    setupNodeEvents(on, config) {
      // Puoi aggiungere ascoltatori di eventi qui, ad esempio:

      // Puoi aggiungere altre configurazioni o plugin qui
    },
    specPattern: 'cypress/integrations/*.spec.js',
    supportFile: 'cypress/support/e2e.js'
  },
  env: {
    apiBaseUrl: 'http://localhost:10000/api'
  },
  fixturesFolder: 'cypress/fixtures'
});
```

#### 5.1.2.4 Risultati ottenuti

Di seguito verranno mostrati gli output di questi test per ogni microservizio:

- Authentication-service:

```
Running: authentication-service.spec.js (1 of 1)

Authentication Service API
  ✓ Devo registrare un nuovo utente con successo (410ms)
  ✓ Da utente registrato voglio poter loggare con successo (183ms)
  ✓ Voglio poter effettuare un logout con successo (63ms)

3 passing (782ms)

(Results)

Tests:      3
Passing:    3
Failing:    0
Pending:    0
Skipped:    0
Screenshots: 0
Video:      false
Duration:   0 seconds
Spec Ran:   authentication-service.spec.js

=====

(Run Finished)

Spec                                Tests  Passing  Failing  Pending  Skipped
✓ authentication-service.spec.js    784ms   3         3        -        -        -
✓ All specs passed!                 784ms   3         3        -        -        -
```

Figura 30: Test authentication

- Message-service:

```
Running: message-service.spec.js (1 of 1)

Message Service API
  ✓ Dovrebbe inviare un messaggio con successo (280ms)
  ✓ Dovrebbe ricevere i messaggi con successo (38ms)

2 passing (376ms)

(Results)

Tests:      2
Passing:    2
Failing:    0
Pending:    0
Skipped:    0
Screenshots: 0
Video:      false
Duration:   0 seconds
Spec Ran:   message-service.spec.js

=====

(Run Finished)

Spec                                     Tests  Passing  Failing  Pending  Skipped
✓ message-service.spec.js                385ms    2         2        -        -        -
✓ All specs passed!                      385ms    2         2        -        -        -
```

Figura 31: Test message

- User-service:

```
Running: user-service.spec.js (1 of 1)

API /api/users
  ✓ Dovrebbe ritornare una lista di utenti (755ms)

1 passing (975ms)

(Results)

Tests:      1
Passing:    1
Failing:    0
Pending:    0
Skipped:    0
Screenshots: 0
Video:      false
Duration:   0 seconds
Spec Ran:   user-service.spec.js

=====

(Run Finished)

Spec                                     Tests  Passing  Failing  Pending  Skipped
✓ user-service.spec.js                  979ms    1         1        -        -        -
✓ All specs passed!                    979ms    1         1        -        -        -
```

Figura 32: Test user

## 5.2 Sicurezza tramite JWT e bcrypt

La sicurezza è un aspetto fondamentale per proteggere la nostra web app da potenziali minacce e garantire la protezione dei dati degli utenti.

### 5.2.1 Gestione sicura dell'autenticazione e dell'autorizzazione: utilizzo di JWT

Per la gestione dell'autenticazione, dell'autorizzazione e quindi della sessione utente, utilizziamo i JSON Web Token (JWT). I JWT rappresentano un metodo sicuro e compatto per trasmettere informazioni tra le parti.

Nella nostra implementazione, generiamo un token JWT che contiene l'ID dell'utente e lo impostiamo come un cookie sicuro. Questo processo avviene tramite una funzione che utilizza la libreria *jsonwebtoken* per creare il token con una chiave segreta e una scadenza di 15 giorni. Il token viene poi inviato al client e memorizzato come un cookie HTTP-only e same-site, prevenendo attacchi XSS (Cross-Site Scripting) e CSRF (Cross-Site Request Forgery). Inoltre, il cookie è impostato come sicuro per essere trasmesso solo su connessioni HTTPS, garantendo ulteriore protezione in ambienti di produzione. Infine, è importante sottolineare la gestione delle sessioni in quanto JWT offre funzionalità per gestire le sessioni degli utenti attraverso cookies e timeout.

Qui di seguito verrà riportato il file “***generateToken.js***”, nella cartella ***utils*** del nostro progetto, il quale si occupa della generazione del token:

```
import jwt from "jsonwebtoken"; //andiamo a usare un token JWT (JSON Web Token) che
è un metodo per rappresentare informazioni tra due parti in modo sicuro e compatto.
// la funzione sottostante genera un token JWT che contiene lo userId
const generateTokenAndSetCookie = (userId, res) => {
  const token = jwt.sign({ userId }, process.env.JWT_SECRET, {
    // JWT_SECRET è una chiave segreta utilizzata per firmare il token. Questa
    chiave dovrebbe essere mantenuta segreta e non condivisa pubblicamente.
    expiresIn: "15d", //indica che il token scadrà in 15 giorni.
  });

  res.cookie("jwt", token, {
    //imposta un cookie chiamato "jwt" con il valore del token generato.
    maxAge: 15 * 24 * 60 * 60 * 1000, // imposta la durata del cookie a 15 giorni,
    in milliseconds.
    httpOnly: true, // previene attacchi XSS (Cross-Site Scripting)
    sameSite: "strict", // prevenire attacchi CSRF (Cross-Site Request Forgery)
```

```

    secure: process.env.NODE_ENV !== "development", //imposta il cookie come sicuro
    (trasmesso solo su HTTPS)
  });
};
export default generateTokenAndSetCookie;

```

### 5.2.2 Hashing della password tramite Bcrypt

Bcrypt è una libreria per la crittografia delle password, progettata per essere computazionalmente costosa da calcolare, rendendo più difficile per gli attaccanti eseguire attacchi di forza bruta sulle password hashate. Bcrypt utilizza un algoritmo di hashing adattivo che consente di specificare un fattore di costo che determina quanto tempo ci vuole per generare l'hash. Questo fattore di costo può essere aumentato nel tempo man mano che le capacità computazionali aumentano, mantenendo così la sicurezza delle password.

Ecco come nel nostro progetto abbiamo utilizzato la libreria Bcrypt:

```

// HASH PASSWORD
    const salt = await bcrypt.genSalt(10); //il salt è una stringa casuale
    utilizzata per aggiungere ulteriore casualità all'hashing delle password con un
    fattore di costo di 10.
    const hashedPassword = await bcrypt.hash(password, salt); //prende la
    password originale e il salt generato nel passaggio precedente e produce un hash
    della password. L'hash risultante è una stringa che rappresenta la password in una
    forma crittografata e sicura.

```

Ed ecco come nel nostro database si può vedere come, per ogni utente, la password venga criptata correttamente:

```

_id: ObjectId('668019145ea00001519fe769')
fullName: "Danilo Romano"
username: "daniloromano"
password: "$2a$10$0A2rqvTIj9uZZYRxV.Jzn.2omrc5djg4VN/Ey04DZjhGh5PKBy0LS"
gender: "male"
profilePic: "https://robohash.org/daniloromano?set=set1"
createdAt: 2024-06-29T14:24:20.848+00:00
updatedAt: 2024-06-29T14:24:20.848+00:00
__v: 0

```