Università di Bologna

Elaborato di progetto per l'esame di Paradigmi di Programmazione e Sviluppo

Distributed Chat Service

Nicola Piscaglia
nicola.piscaglia2@studio.unibo.it

Mattia Vandi
mattia.vandi@studio.unibo.it

Alessandro Gnucci
alessandro.gnucci@studio.unibo.it

Martina Magnani
martina.magnani8@studio.unibo.it

A.A. 2017/2018

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Processo di Sviluppo

1.1 Metodologia di sviluppo

Relativamente al processo di sviluppo è stato deciso di adottare e testare una metodologia **Agile Scrum-like**.
Nella versione da noi utilizzata i membri del team sono allo stesso livello, non esiste infatti uno *Scrum Leader*, ne uno *Scrum Manager*. Il ruolo del *Product Manager* è stato ricoperto da *Nicola Piscaglia*. Come alternativa ad una vera e propria *Kanban* abbiamo deciso di usare **Trello**: qui, come fasi dello sprint abbiamo usato *"Backlog"*, *"ToDo"*, *"Design"*, *"Develop"*, *"Review"* e *"Done"*.
Abbiamo reputato corretto fare sprints con cadenza settimanale, poichè così è possibile ottenere una buona quantità di feedback dall'utente e numerose opportunità per imparare e migliorare.
All'inizio del processo di sviluppo abbiamo delineato il *Product Backlog*, contenente gli items, con relativa priorità e dimensione stimata; inoltre, è stato fatto un primo sforzo per la modellazione del sistema in termini di servizi e patterns architetturali, producendo: un diagramma relativo alla gerarchia dei servizi, uno schema relazionale ed uno logico dei databases.
All' inizio di ogni settimana abbiamo fatto un meeting, tutti assieme, per decidere quali elementi del *Product Backlog* assegnare al prossimo sprint, strutturando lo *Sprint Backlog*, delineando con più precisione la dimensione degli items.
A seguito di ciò, i membri del team hanno scelto gli items seguendo le priorità già delineate ed hanno iniziato a fare il design ed implementarne le features descritte tramite *TDD*, senza più cambiare le decisioni relative agli items prese all'inizio dello sprint.
Il Test Driven Development è un modello di sviluppo che prevede che la stesura dei tests sia fatta prima dell'implementazione e che lo sviluppo sia orientato esclusivamente all'obiettivo di passare i tests precedentemente predisposti. Più in dettaglio, *TDD* prevede un breve ciclo di sviluppo in tre fasi: nella prima si scrive un test per la nuova funzione da sviluppare, nella seconda si sviluppa il codice necessario per passare il test; infine, nella terza si esegue il refactoring del codice per adeguarlo ad determinati standards di qualità.
Durante lo sviluppo, ogni singolo elemento del team ha la possibilità di fare *Pull Requests* al *Repo verità* (non si può fare push): a quel punto, un altra persona del team controllerà la qualità del codice della singola *PR* e, se tutto risulta a norma, farà merge nel branch dello sprint corrente. Lo strumento di *continuous integration* è stato configurato per permettere il merge di una pull request

solo in caso di procedura di build andata a buon fine e solo se tutti i tests terminano con successo; Infine, è stato impostato lo stesso strumento in modo da generare in automatico l'eseguibile del progetto.
Al termine di ogni *Sprint* i componenti del team hanno preso parte ad un meeting, in cui sono state poste in atto le fasi di *Sprint Review* (per valutare il risultato dello Sprint e stabilire le priorit'a delle prossime feature da portare a termine) e di *Retrospective* (per analizzare il processo di sviluppo allo scopo di migliorare la produttività del team).
Si è reso necessario, a volte, contattare elementi del team tramite Skype o strumenti simili, senza riunire però tutti gli elementi, per risolvere problemi o per procedere con lo sviluppo. Infine, gli items non completati entro la fine dello sprint sono stati semplicemente riportati nello sprint successivo.

1.2 Strumenti adottati

I principali strumenti utilizzati durante l'intero processo di sviluppo sono i seguenti:

- **Git**: come sistema di versioning abbiamo optato per Git per la sua diffusione, velocità e semplicità di utilizzo;
- **GitHub**: è stato scelto GitHub come servizio di hosting; per il version control e per il repository, poichè permette ampia possibilità di integrazione con altri strumenti di sviluppo. Il repository è disponibile al seguente link;
- **GitFlow**: come metodologia di *branching* all'interno del repository abbiamo adottato GitFlow, poichè rende lo sviluppo parallelo molto semplice;
- **Google Sheets**: la creazione e gestione del *Product Backlog*, *Sprint Backlog*, *Sprint Review* e *Sprint Retrospective* sono state fatte tramite Google Sheets, vista la sua semplicità di utilizzo. È possibile visualizzare i documenti predetti al seguente link;
- **Gradle**: per la gestione della build di progetto è stato scelto Gradle, per la sua alta velocità;
- **Travis CI**: è stato inoltre utilizzato Travis CI come strumento di *continuos integration*: esso permette di verificare, ad ogni nuova modifica del repository, che il progetto continui a compilare con successo e a controllare che i tests passino, in un ambiente unico;
- **Trello**: come strumento di tracciamento delle fasi del ciclo di sviluppo. È possibile visionare la *Kanban* al seguente link.

Capitolo 2

Requisiti

L'obiettivo del progetto è realizzare un *servizio di chat distribuito*. Nella nostra applicazione potranno esserci uno o più *utenti*, i quali potranno unirsi o abbandonare una o più *stanze*. Una stanza è rappresentata da un nome univoco e da un insieme di utenti. All'interno di ogni stanza ogni utente può inviare messaggi altri utenti *partecipanti* e riceverne dagli stessi. L'utente avrà la possibilità di creare e modificare il proprio *profilo* utente e di visualizzare quello dei partecipanti delle stanze in cui è presente.

2.1 Requisiti di business

Dall'analisi del problema sono emersi i seguenti requisiti funzionali che ci prefiggiamo di implementare:

- Possibilità da parte di un utente di **unirsi a una o più stanze ed abbandonarle**.
- **Eliminazione di una stanza** (solamente da parte del suo creatore).
- Possibilità da parte di un utente di **visualizzare lo stato (effettiva connessione al sistema) degli utenti cha partecipano alla stanza**.
- Possibilità di **visualizzare il profilo utente** (nome utente, nome, biografia, stato).
- **Aggiornamento real-time ai partecipanti di una stanza sullo stato di scrittura** di un utente che digita.
- Possibilità di **cercare una chat per nome**.
- Possibilità, da parte di un utente, di **rendersi invisibile agli altri utenti**.

Le funzionalità utente sono state descritti in maniera visuale tramite il diagramma dei casi d'uso in Figura 2.1.

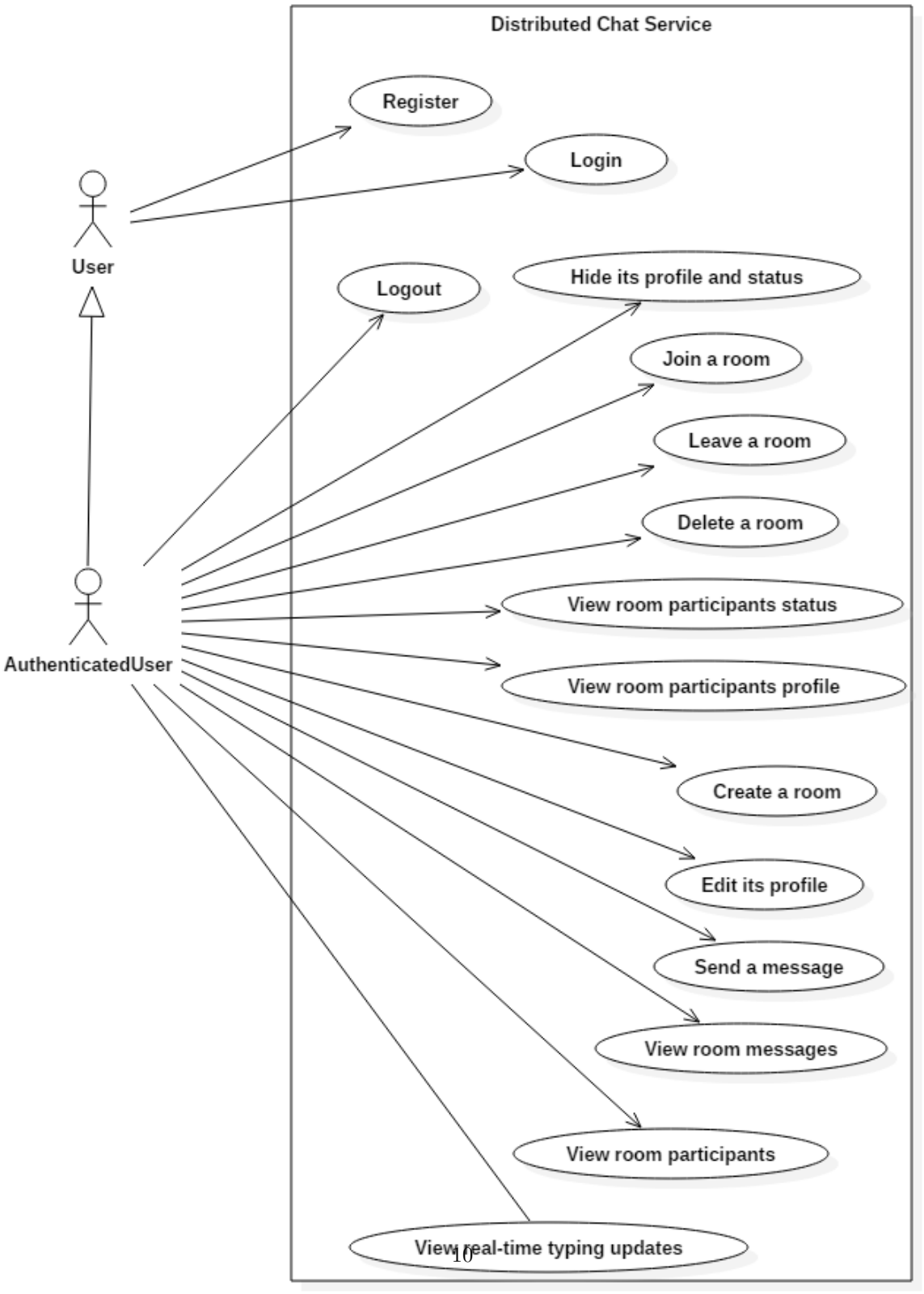

Figura 2.1: Diagramma dei casi d'uso.

2.1.1 Unione ad una stanza ed uscita

L'unione dell'utente ad una stanza comporta la possibilità di visualizzare i messaggi in essa contenuta (anche precedenti al *join* dell'utente), di inviare messaggi e di visualizzare le informazioni sulla stanza (numero, nome e data di join dei partecipanti). È possibile uscire in qualsiasi momento da una stanza a cui ci si era uniti.

2.1.2 Eliminazione di una stanza

È possibile, per il suo creatore, eliminare una stanza indipendentemente dal numero di messaggi e di partecipanti che contiene in un certo momento.

2.1.3 Visualizzazione dello stato degli utenti

Un partecipante ad una stanza può visualizzare lo stato degli altri partecipanti (a meno che lo stato da visualizzare appartenga ad un utente invisibile). Per *stato* si intende l'effettiva *connessione* al servizio di chat in un certo periodo temporale.

2.1.4 Visualizzazione e modifica del profilo utente

Un partecipante ad una stanza può visualizzare il profilo utente degli altri partecipanti (a meno che il profilo da visualizzare appartenga ad un utente invisibile). Il profilo utente è modificabile tramite apposita schermata e comprende: nome utente, nome, biografia e flag di invisibilità.

2.1.5 Aggiornamento real-time sullo stato di scrittura

Nel momento in cui un utente scrive un messaggio all'interno di una stanza, allora agli utenti che partecipano alla stessa stanza verrà visualizzato il fatto che quell'utente sta digitando.

2.1.6 Ricerca di una stanza per nome

Sarà presente un'apposita schermata di ricerca di una stanza. La ricerca avviene digitando il nome della stanza da trovare. Nel momento in cui l'utente inizia a digitare, verranno visualizzate tutte le stanze che contengono nel proprio nome la stringa digitata. Successivamente, sarà possibile effettuare il *join* alla stanza semplicemente cliccando sul suo nome.

2.1.7 Offuscamento del proprio stato agli altri utenti

Tramite la modifica del proprio profilo utente, è possibile impostare un *flag di invisibilità*. Se tale flag è attivo, il proprio profilo ed il proprio stato non saranno visibili da altri utenti. Inoltre, non sarà possibile, per l'utente invisibile, visualizzare lo stato e profilo degli altri utenti.

2.2 Requisiti non funzionali

L'unico requisito non funzionale del sistema è la **sicurezza del sistema di interazione** tra i componenti del sistema, in particolare del meccanismo di autenticazione. Il sistema di autenticazione dovrà:

- Permettere la registrazione, inserendo *username* e *password*, rilasciando un *token* di autenticazione all'utente;
- Memorizzare in maniera persistente le informazioni inserite;
- Permettere la cancellazione di un utente, fornendo il suo *token* di autenticazione;
- Permettere il login, inserendo *username* e *password*, rilasciando un *token* di autenticazione all'utente;
- Permettere la validazione di un *token* di autenticazione, ottenendo la descrizione dell'utente in risposta.

Alcune interazioni (etichettate come protette) tra i componenti del sistema saranno eseguibili soltanto previa autenticazione; quindi necessiteranno dell'invio, contestuale alla richiesta, di un token precedentemente rilasciato dal sistema di autenticazione.

2.3 Requisiti di implementazione

Per lo sviluppo del progetto sono stati definiti a priori i seguenti requisiti implementativi:

- **Scala**: il linguaggio da utilizzare per lo sviluppo della maggior parte del progetto deve essere *Scala*. Imponiamo questo requisito in quanto vogliamo sperimentare il paradigma funzionale, appreso nel corso di *Paradigmi di programmazione e sviluppo*, ed i vantaggi di *Scala* rispetto ai linguaggi studiati in corsi precedenti.
- **Vertx**: data la natura dell'applicazione, abbiamo scelto *Vertx* come piattaforma scalabile, concorrente, non bloccante e distribuita su cui realizzare i servizi di *back-end* del sistema. *Vertx* permette di realizzare applicazioni reattive fornendo buone performance ed un consumo di risorse ridotto.
- **TDD**: dato l'utilizzo di una metodologia di sviluppo *Agile* e la necessità, vista la complessità del sistema, di un uso intensivo di strumenti di test da affiancare al puro sviluppo di funzionalità, è stato deciso di applicare quando possibile un modello di sviluppo *test-driven*. In questo modo è possibile:
  - ottenere, dopo una prima stesura, un codice in gran parte già testato.
  - verificare in maniera automatica se successive modifiche al codice comportano una perdita di correttezza.
  - ottenere una specifica delle funzionalità che il software prodotto deve realizzare.

- **Angular**: per la realizzazione del *front-end* della web application è stato scelto il framework *Angular*. Quest'ultimo è stato progettato per fornire uno strumento facile e veloce per sviluppare applicazioni che girano su qualunque piattaforma, inclusi smartphone e tablet. Infatti, le applicazioni web in Angular, in combinazione con il toolkit open source *Bootstrap* o *Angular Material* sono naturalmente responsive, ossia il design del sito web si adatta in funzione alle dimensioni del dispositivo utilizzato.

- **MySQL**: la gestione della persistenza è stata realizzata attraverso un *Database MySQL* con relativo deployment su piattaforma online *GearHost*. La scelta di tale piattaforma è stata condizionata dall'esigenza di reperire uno spazio di hosting gratuito.

Capitolo 3

Design architetturale

3.1 Pattern architetturali

L'architettura del sistema è una classica **Client-Server**.
La parte server è stata suddivisa in **microservizi RESTFul**, a basso coupling, per aumentare la modularità;
Il lato Server è stato realizzato secondo i principi della **Clean Architecture**, un pattern architetturale utile per ottenere una buona separazione dei compiti.
Il client adopera inoltre un'architettura **Component-Based**.

3.1.1 Clean Architecture

Per ottenere la separazione dei compiti, tale architettura impone la suddivisione del codice in differenti strati: ogni strato può conoscere, chiamare ed utilizzare solo gli strati più interni, imponendo così la *Dependency Rule*: in tal modo i cambiamenti di codice in uno strato esterno non causano modifiche agli strati interni. La Dependency Rule permette, dunque, di creare un sistema intrinsecamente modificabile e testabile, con tutti i benefici che ciò comporta. Inoltre, seguendo il *Single Responsability Principle* e dipendendo solo da astrazioni, diventa semplice sostituire le implementazioni.

Gli strati che compongono l'architettura sono i seguenti:

- *View*: prende gli eventi utente e li passa al presenter, mostrando inoltre i dati ricevuti dal presenter e permettendo azioni relative all'interfaccia grafica, come la gestione delle animazioni;
- *Presenter*: fa da "middle man" tra la View e gli Use Cases; è il Presenter che formatta e passa i dati da visualizzare alla view e che gestisce gli eventi utente in arrivo dalla view, chiamando poi il giusto Use Case. Nel codice del Presenter non bisogna inserire nulla di relativo ad un singolo framework, poichè deve essere indipendente da essi;

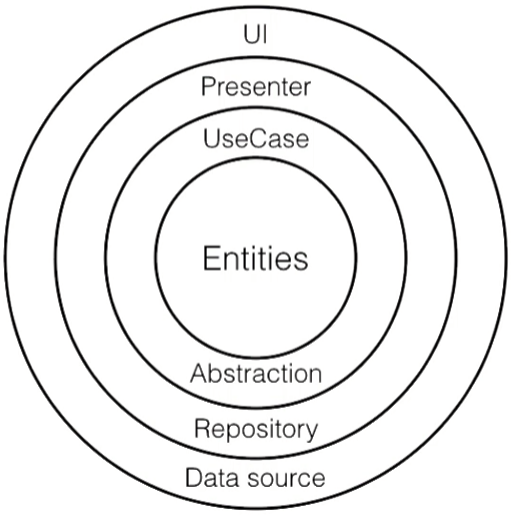

Figura 3.1: Rappresentazione della Dependency Rule e degli strati della Clean Architecture.

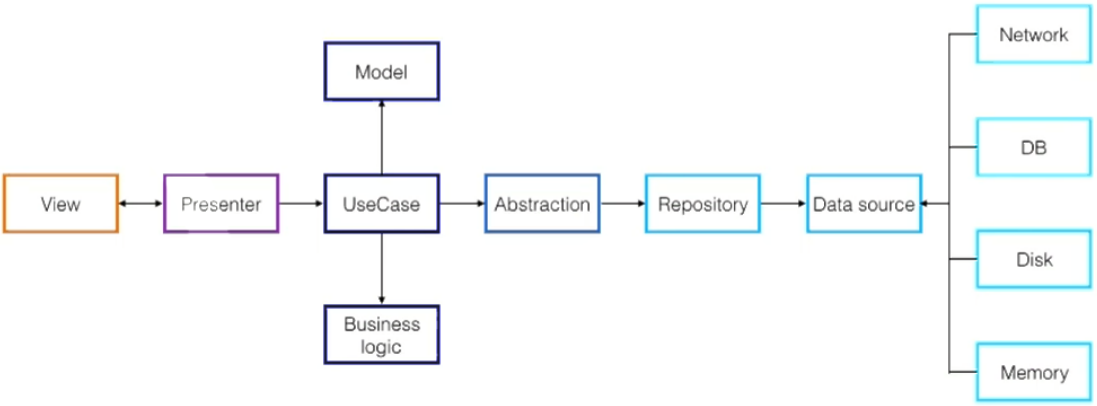

Figura 3.2: Diagramma contenente tutti gli strati della Clean Architecture e le loro relazioni

- *Use Case*: sono classi contenenti la logica necessaria per implementare un use case, ovvero uno specifico compito di business che l'utente cerca di portare a termine. Gli Use Case adoperano le classi di *Model* e sfruttano uno o più abstractions. Dunque, gli Use Cases ottengono i dati tramite le abstractions; a seguito di ciò essi usano le più appriopriate classi di business logic, per poi restituire un risultato, in modo asincrono, tramite un *Subscriber* che il Presenter ha fornito;
- *Abstraction*: astrazione per componenti specifici ad un framework;
- *Repository*: un Repository è un data provider; al suo interno si sceglie quale *Data Store* usare o, in caso se ne usi più di uno in contemporanea, se ne unisce i risultati (ad esempio si sceglie la sorgente dati che per prima restituisce il risultato). Il repository mappa in modo opportuno il risultato ottenuto, in modo che sia comprensibile dagli strati superiori, facendone in caso anche caching.
- *Data Store*: è un'implementazione di una specifica sorgente dati; è dunque possibile realizzare multipli Data Source, uno per ogni tipologia di sorgente dati: Internet, Database, Disco Rigido, Memoria Centrale.

Il primo strato, ovvero la View, non è presente lato Server, poichè è il Client che se ne occupa.

La Clean Architecture permette di ottenere i migliori risultati se usata assieme ai seguenti elementi, i quali sono stati da noi adottati:

- Un framework o tecnica di *Dependency Injection*, per estrarre il codice di creazione delle dipendenze dalle classi, ottenendo codice più riusabile, testabile, leggibile e con maggiore separazione dei compiti. Qui la scelta è ricaduta sul *Constructor Injection*, per la sua immediatezza nell'utilizzo;
- *Reactive Extensions*, per operare su sequenze di dati asincroni in modo dichiarativo e funzionale;
- Un ampio uso delle lambdas, per ridurre la quantità di codice boilerplate.

3.1.2 Microservizi

Per architettura a microservizi si intende un'architettura formata da una collezione di servizi tra loro cooperanti e debolmente accoppiati, ovvero che non richiedono uno stack tecnologico specifico per l'interoperabilità, a meno delle principali tecnologie web.
Il beneficio principale della scomposizione di un unico software monolite in più servizi è l'aumento di modularità, rendendo così più facile lo sviluppo: infatti, tale architettura permette di sviluppare e testare parti più piccole del sistema. Inoltre, permette lo sviluppo in maniera parallela e autonoma da parte di più teams.
Infine, questa architettura rende più facile il raggiungimento di un buon livello di resilienza e scalabilità, poichè, in caso di bisogno di replicazione, è possibile replicare la sola parte interessata, senza dover replicare l'intero monolite; infatti, spesso parti diverse del monolite hanno caratteristiche e necessità diverse, dunque è giusto permettere ad ogni parte di essere ridondata in un numero corretto di volte.

3.1.3 Architettura del Client

Il client è stato realizzato secondo un'architettura *Component-Based*, che comprende i seguenti elementi:

- *Template*: definisce come renderizzare un componente, specificandone gli elementi grafici e i data bindings (mono o bidirezionali) dei relativi dati;
- *Component*: è il codice che gescisce una sezione della vista, aggiornandone i dati e reagendo agli eventi, chiamando, se necessario, uno o più *Services*. Un Component è sempre associato ad un template. I vari Component si distribuiscono ad albero, infatti ogni component può a sua volta contenere ulteriori Components, formando così la struttura della view.
- *Service*: è un'ampia categoria che comprende un qualsiasi valore, funzione o funzionalità di cui l'applicazione ha bisogno.

Infine, è stato fatto abbondante uso di *Dependency injection* sia nei Components sia nei Services, tramite i loro costruttori.

3.2 Componenti del sistema distribuito

Come già introdotto nella sezione precedente, il sistema vuole realizzare una *Web Application* basata sul classico modello architetturale Client-Server. Mentre la parte Client del sistema è rappresentata da un'unica applicazione monolitica, il sistema Server è stato scomposto in più componenti seguendo il modello a microservizi. Dopo una prima fase di design del back-end del sistema, sono stati identificati 4 microservizi principali:

- *WebApp Service*
- *User Service*
- *Room Service*
- *Authentication Service*

Inoltre è stato successivamente introdotto anche un *servizio di Heartbeat* per permettere lo sviluppo della feature di visualizzazione dello stato effettivo di connessione al sistema di un utente. Infine, viene descritta la struttura dei *DB* utilizzati da ogni microservizio ai fini della persistenza delle informazioni. Prima di descrivere ogni componente, viene successivamente definito il modello di riferimento delle entità di business del sistema. La definizione di tale modello infatti impatterà la struttura dei dati utilizzata da ogni componente.

3.2.1 Model

Prima di addentrarsi nell'analisi di ogni singolo componente del sistema distribuito, sono stati modellati i concetti di business principali tramite un diagramma delle classi *UML*. Questa fase sarà utile poi per il design di ogni componente poichè permette di definire lo schema di base della struttura dei dati. Le entità di dominio e le relative associazioni sono espresse in Figura 3.3.

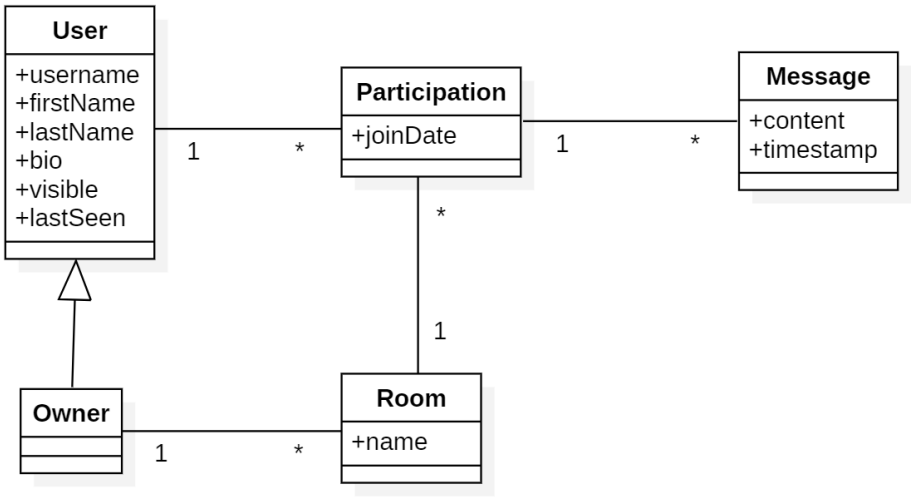

Figura 3.3: Diagramma delle classi delle entità di business.

3.2.2 Web Client

Rappresenta la parte del sistema utilizzabile dall'utente tramite un browser. Questo componente permette la visualizzazione delle schermate tramite le quali l'utente può interagire col sistema ed effettuare tutte le operazioni di business definite in fase di analisi del problema. Il *Web Client* interagisce con il *Server* sia in maniera *pull* sia in maniera *push*: questa duplice modalità di interazione permette di creare un sito web reattivo ai cambiamenti in maniera efficiente. Non vi è infatti, la necessità di effettuare *polling* sul server per verificare se i dati sono stati aggiornati. Nelle seguenti sottosezioni vengono definiti i componenti del sistema *Client* considerando le astrazioni fornite dal framework *Angular*.

3.2.2.1 Components

I principali componenti *Angular* del *front-end* identificati sono:

- ***Registration***: incapsula la grafica della schermata di registrazione di un utente e la logica di controllo degli eventi scatenati dalla relativa interfaccia.
- ***Login***: incapsula la grafica della schermata di login e la logica di controllo degli eventi scatenati dalla relativa interfaccia.
- ***Add-room***: incapsula la grafica della schermata di creazione di una stanza e la logica di controllo degli eventi scatenati dalla relativa interfaccia.
- ***Edit-profile***: incapsula la grafica della schermata di *editing* del proprio profilo utente, la logica di controllo degli eventi scatenati dalla relativa interfaccia e l'attivazione del recupero delle informazioni dell'utente corrente.

- ***Room-info***: incapsula la grafica della schermata di visualizzazione delle informazioni di una stanza, la logica di controllo degli eventi scatenati dalla relativa interfaccia e l'attivazione del recupero delle informazioni della stanza selezionata.
- ***Rooms***: incapsula la grafica della lista delle stanze a cui l'utente si è unito, la logica di controllo degli eventi scatenati dalla relativa interfaccia e l'attivazione del recupero delle stanze.
- ***Search-rooms***: incapsula la grafica della lista delle stanze ricercate, la logica di controllo degli eventi scatenati dalla relativa interfaccia e l'attivazione del recupero delle stanze.
- ***Sidebar***: incapsula la grafica della schermata di visualizzazione delle stanze cercate o a cui l'utente si è unito e la logica di controllo degli eventi scatenati dalla relativa interfaccia.
- ***Top-navbar***: incapsula la grafica della barra di navigazione e la logica di controllo degli eventi scatenati dalla relativa interfaccia.
- ***User-profile***: incapsula la grafica della schermata di visualizzazione del profilo di un utente e l'attivazione del recupero delle informazioni sull'utente visualizzato.

3.2.2.2 Services

I servizi *Angular* identificati sono:

- ***Authentication***: mantiene il riferimento all'utente che si è autenticato e fornisce le funzionalità di *login*, *registrazione*, *logout*.
- ***Chat***: permette di effettuare le principali chiamate, riguardanti la *Chat*, al *Server* e di intercettarne le relative risposte in maniera asincrona.
- ***User***: offre le funzionalità di recupero/modifica del profilo utente e permette di aggiornare le informazioni sullo stato di connessione del *Client*.
- ***Event-bus***: permette di utilizzare l'*Event Bus* di *Vertx* per interagire in maniera non bloccante secondo il modello *publish-subscribe*. Viene utilizzato per intercettare gli eventi asincroni provenienti dal *Server*.

3.2.2.3 Template

Il template di base del front-end è la versione di *SB Admin* realizzata tramite componenti dell'UI framework *Angular Material*. Sono stati utilizzati alcuni componenti del predetto template, editandoli in maniera opportuna per addattarli alle nostre esigenze grafiche. L'utilizzo di tale framework ha permesso di creare uno sito web accessibile e naturalmente *responsive*. In Figura 3.4 viene mostrata la schermata iniziale del template precedentemente citato.

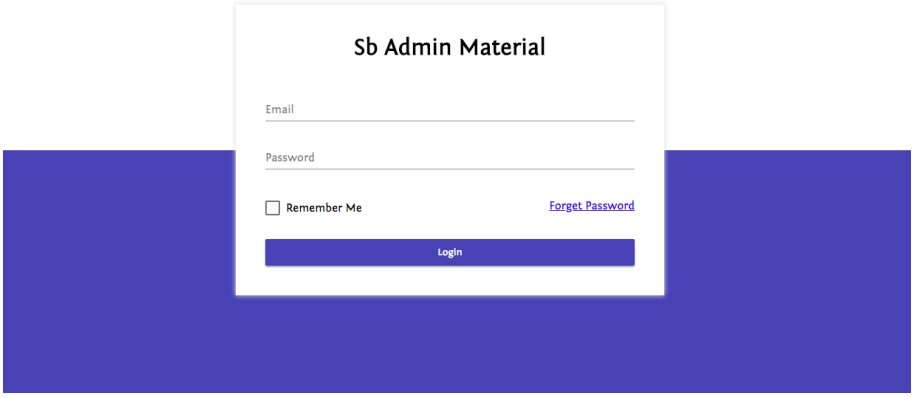

Figura 3.4: Schermata di login del template SB Admin realizzato tramite il framework UI *Angular Material*.

3.2.3 Web Server

3.2.3.1 Struttura

Si elencano e descrivono di seguito, i componenti nei quali il *Web Server* è stato scomposto. Come detto in precedenza infatti, questa parte del sistema è stata progettata secondo il modello a microservizi.

3.2.3.1.1 WebApp Service

Il *WebApp Service* rappresenta il microservizio che gestisce l'interazione tra il client ed i restanti servizi di back-end. Il suo ruolo principale è quello di intercettare le richieste da parte dei client ed eseguire le operazioni necessarie per elaborare e restituire il relativo risultato. Nel perseguire il suo obiettivo il *WebApp Service* tipicamente interagisce con uno o più microservizi di back-end. La struttura del *WebApp Service* è descritta tramite un diagramma UML delle classi in Figura 3.5. In tale figura sono stati omessi alcuni dei casi d'uso per facilitare la lettura del resto del diagramma. Per completezza, si riporta la lista completa dei casi d'uso del *WebApp Service* e le relative associazioni ai repositories:

- ***LoginUserUseCase*** $\longrightarrow$ *AuthenticationRepository*, *UserRepository*
- ***RegisterUserUseCase*** $\longrightarrow$ *AuthenticationRepository*, *UserRepository*, *RoomRepository*
- ***LogoutUserUseCase*** $\longrightarrow$ *AuthenticationRepository*, *UserRepository*
- ***JoinRoomUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***LeaveRoomUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*

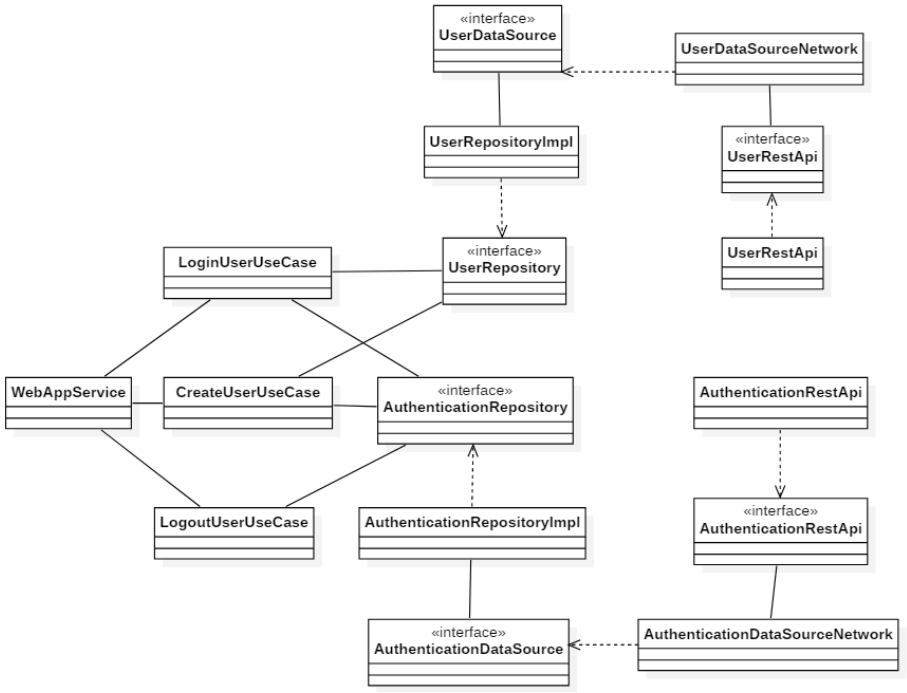

Figura 3.5: Diagramma delle classi dell'*WebApp Service* (alcuni casi d'uso sono stati omessi per facilitare la lettura dello schema).

- ***NotifyTypingInRoomUseCase*** $\longrightarrow$ *AuthenticationRepository*
- ***SendMessageUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***GetUserUseCase*** $\longrightarrow$ *AuthenticationRepository*, *UserRepository*
- ***GetUserParticipationsUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***GetRoomsUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***GetRoomParticipationsUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***GetMessagesUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***EditUserUseCase*** $\longrightarrow$ *AuthenticationRepository*, *UserRepository*
- ***DeleteRoomUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***CreateRoomUseCase*** $\longrightarrow$ *AuthenticationRepository*, *RoomRepository*
- ***UserOfflineUseCase*** $\longrightarrow$ *UserRepository*

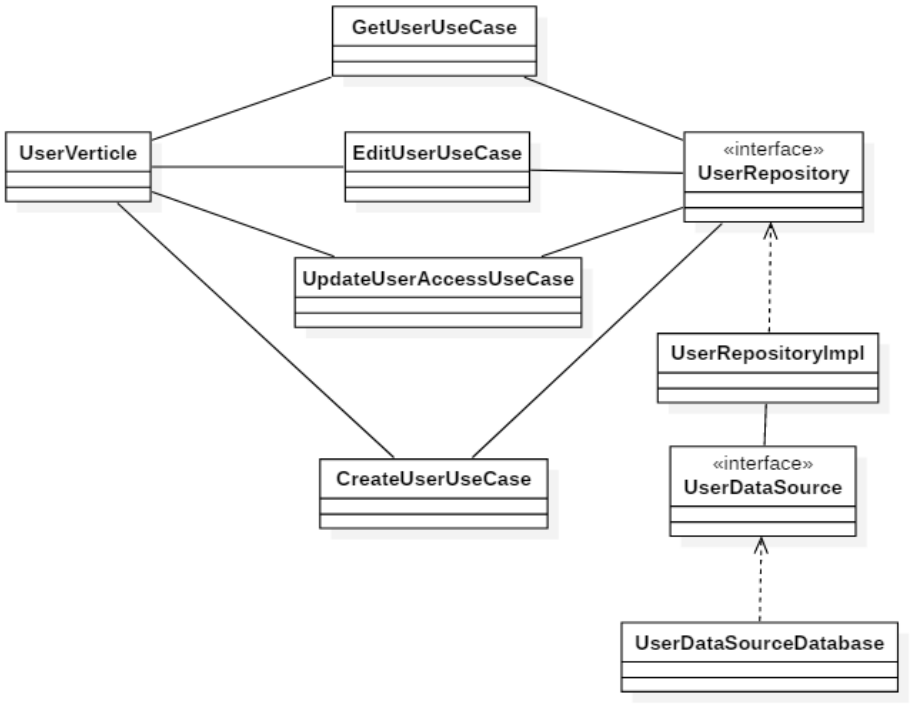

Figura 3.6: Diagramma delle classi dell'*User Service*.

3.2.3.1.2 User Service

Lo *User Service* è il servizio che si occupa della persistenza delle informazioni degli utenti. Offre le classiche operazioni di lettura, scrittura ed aggiornamento sui dati utente. Esempi di operazioni che coinvolgono il servizio sono: reperimento della biografia utente, modifica del nome e cognome, ecc.

3.2.3.1.3 Room Service

Il *Room Service* è il servizio che si occupa della persistenza delle informazioni delle stanze. Offre le classiche operazioni di lettura, scrittura ed aggiornamento sui dati di una stanza. Operazioni tipiche che coinvolgono il Room Service sono ad esempio: reperimento dei messaggi e partecipanti di una stanza, invio di un messaggio, uscita dalla stanza, ecc. La struttura di questo servizio è descritta dal diagramma delle classi in Figura 3.7. In tale figura sono stati omessi alcuni dei casi d'uso per facilitare la lettura del resto del diagramma. Per completezza, si riporta la lista completa dei casi d'uso del *Room Service*:

- ***GetRoomsUseCase***
- ***SendMessageUseCase***
- ***DeleteRoomUseCase***

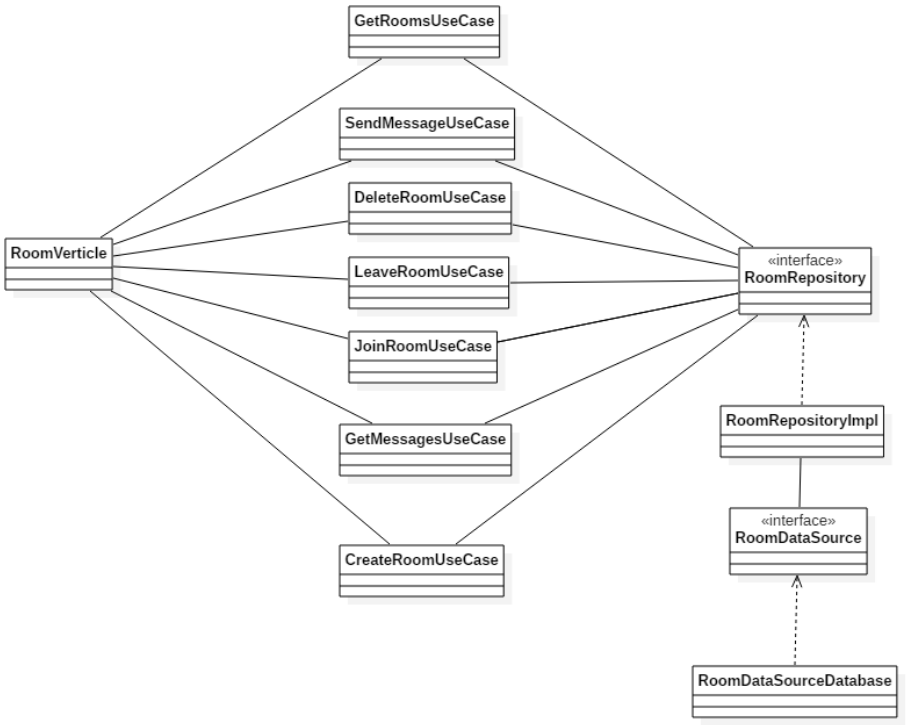

Figura 3.7: Diagramma delle classi del *Room Service* (alcuni casi d'uso sono stati omessi per facilitare la lettura dello schema).

- ***LeaveRoomUseCase***
- ***JoinRoomUseCase***
- ***GetMessagesUseCase***
- ***CreateRoomUseCase***
- ***GetUserParticipationsUseCase***
- ***GetRoomParticipationsUseCase***
- ***CreateUserUseCase***

3.2.3.1.4 Authentication Service

L' *Authentication Service* è il servizio che si occupa della gestione dell'autenticazione dell'utente e della protezione delle rotte di navigazione per le quali bisogna essere loggato per accedervi. Gestisce la persistenza delle credenziali degli utenti ed offre il servizio di creazione e validazione dei *token*. La struttura di questo servizio è descritta dal diagramma delle classi in Figura 3.8.

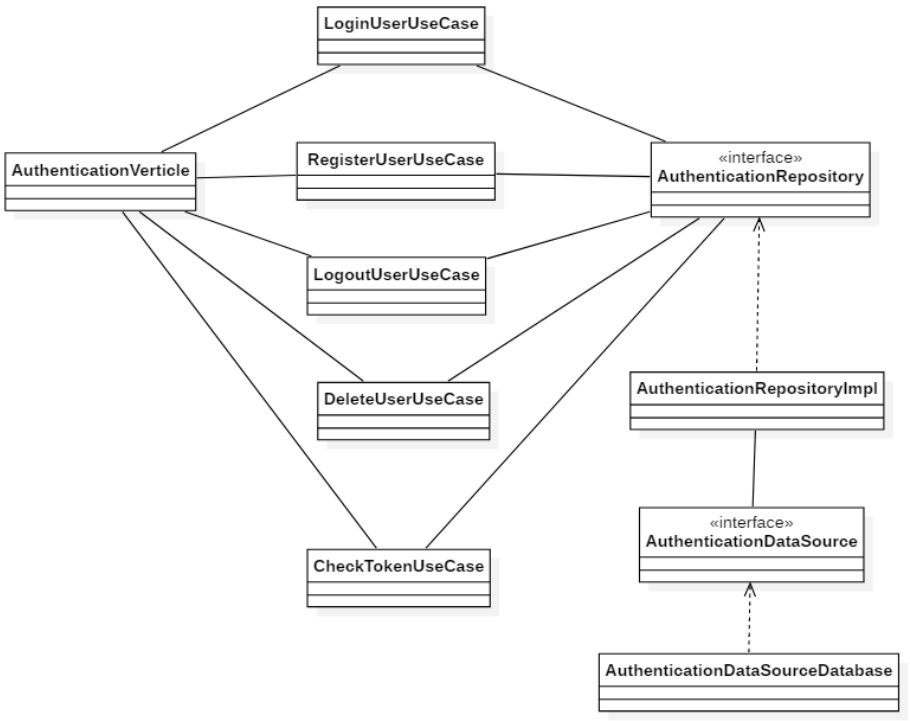

Figura 3.8: Diagramma delle classi dell'*Authentication Service*.

3.2.3.1.5 Heartbeat Service

L'*Heartbeat Service* è il microservizio che si occupa di monitorare lo stato del client. Questo servizio monitora periodicamente i client connessi al sistema verificando se sono ancora connessi. In questo modo il sistema *Server* riuscirà ad inviare ai vari client le informazioni riguardo lo stato di altri utenti.

3.2.3.2 Interazione

In primo luogo, viene riportata la descrizione della gerarchia dei microservizi che ne regola l'interazione. Infine, vengono presentati alcuni scenari di interazione tra i componenti del sistema.

3.2.3.2.1 Gerarchia dei microservizi

La radice della gerarchia dei microservizi è rappresentata dal *WebApp Service*. Quest'ultimo servizio infatti, funge da entry-point del sistema *Server* e quindi da collante tra *Client* e *Server* (il *Client* contatta sempre il *WebApp Service*). Il *WebApp Service* a sua volta comunicherà, a seconda della funzionalità richiesta, con il *Room Service*, *Authentication Service* e *User Service*. I servizi *Room* e *User* presentano una dipendenza verso il servizio di *Authentication* poichè per effettuare alcune operazioni è necessario essere autenticati; quindi l'Authentication Service dovrà prima validare il *token* dell'utente richiedente. Come è possibile vedere in Figura 3.9, il WebApp Service interagisce anche con un *servizio di Heartbeat*. Quest'ultimo è stato pensato per verificare e restituire lo stato di connessione dei *Client* alla *Web Application*.

3.2.4 Database

Il modello scelto per il design del *Database* è quello *relazionale* poichè la natura dei dati è prevalentemente di tipo strutturata. Per ogni microservizio che necessita di persistenza, è stato modellato uno *schema E/R* che ne descriva la struttura dei dati. Secondo tale modello infatti, ogni servizio gestirà il proprio database che implementerà il relativo schema relazionale. Di seguito vengono riportati gli schemi E/R dei servizi che utilizzano un *Database*.

3.2.4.1 Room Service DB

Il database del *Room Service* presenta 4 relazioni:

- *User*: identificata dal solo *username*. Uno *User* può partecipare a più stanze e può essere il creatore di una stanza.
- *Participation*: identificata dalla *data di unione* ad una stanza, da un *utente* e da una *stanza*. Una partecipazione riguarda solamente un utente ed una stanza. Ad una partecipazione possono essere associati 1 o più messaggi.
- *Message*: identificata da una partecipazione e da un timestamp. Presenta l'attributo *content* che rappresenta il testo del messaggio. Ad ogni messaggio è associato una sola partecipazione.

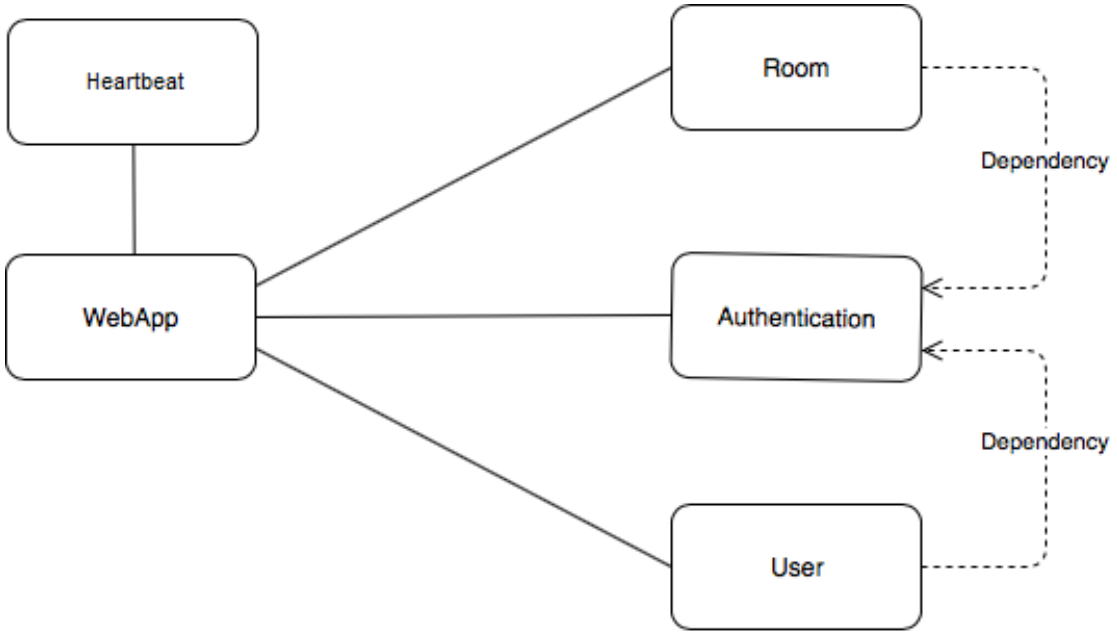

Figura 3.9: Gerarchia dei microservizi identificati per il back-end.

- *Room*: identificata dal solo *nome della stanza*. Ogni stanza ha un solo creatore e può essere associata ad una o più partecipazioni.

3.2.4.2 User Service DB

Il database dello *User Service* presenta una singola relazione *User*. Essa è identificata dallo *username* dell'utente ed è caratterizzata da:

- Nome
- Cognome
- Biografia
- Flag di visibilità
- Ultimo accesso

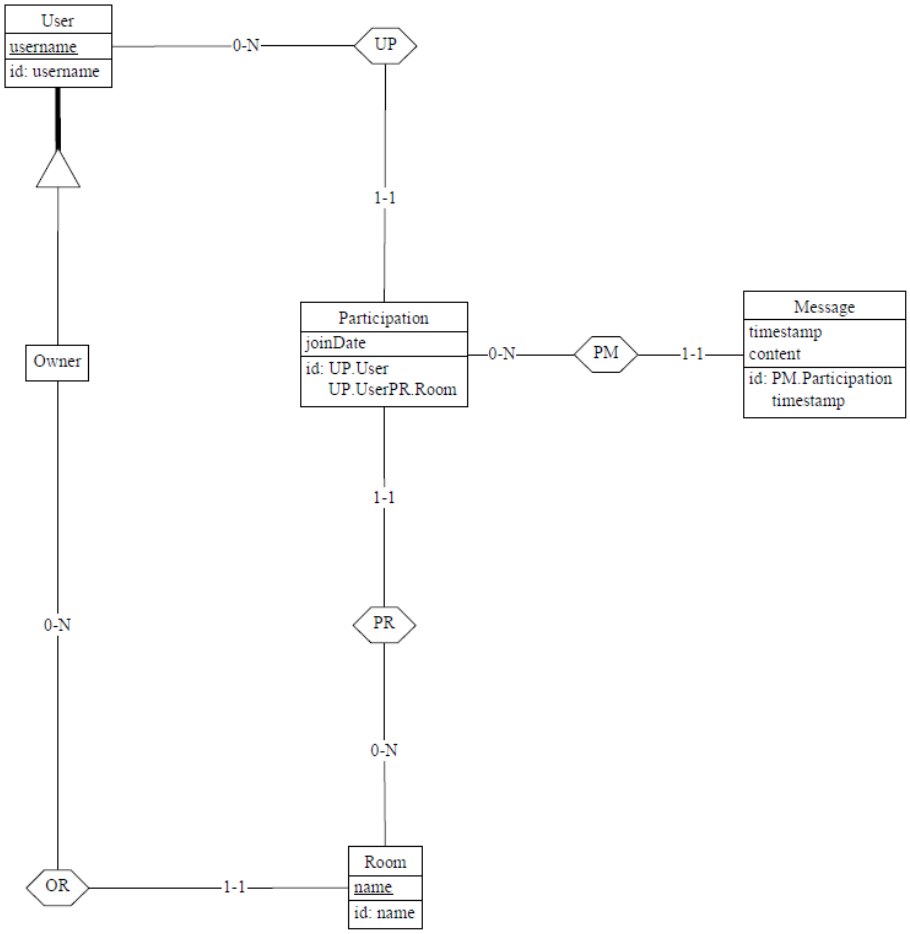

Figura 3.10: *Schema E/R* del Database gestito dal *Room Service*.

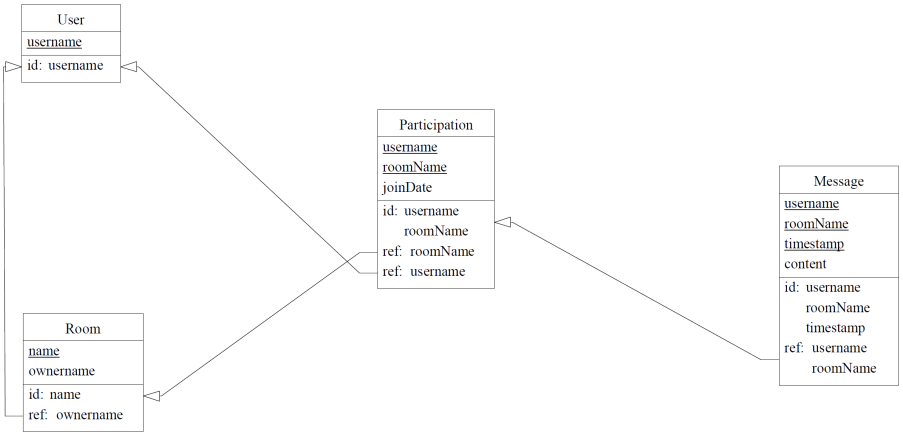

Figura 3.11: *Schema logico* del Database gestito dal *Room Service*.

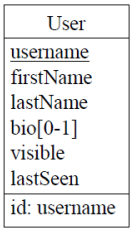

Figura 3.12: *Schema E/R* del Database gestito dal *User Service*.

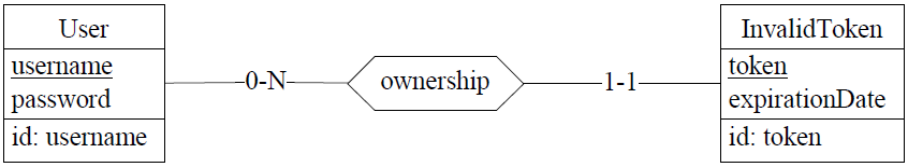

Figura 3.13: *Schema E/R* del Database gestito dall'*Authentication Service*.

Figura 3.14: Schema logico del Database gestito dall'*Authentication Service*. L'associazione *ownership* non è stata tradotta in chiave esterna della tabella *invalid_tokens* in quanto non necessaria; infatti il token intrinsecamente contiene l'informazione relativa allo *username*.

3.2.4.3 Authentication Service DB

Il database dell'*Authentication Service* presenta 2 relazioni:

- *User*: identificato dallo username. È caratterizzata dall'attributo rappresentante la *password* dell'utente. Ad ogni utente possono corrispondere uno o più token invalidi. Quest'ultimi infatti, hanno una scadenza e possono comunque essere invalidati dall'*Authentication Service*.
- *InvalidToken*: identificata dalla stringa del token stesso. Presenta l'attributo corrispondente alla data di scadenza del token.

3.2.5 Esempi di interazione

Riportiamo i diagrammi di sequenza riportanti alcuni degli scenari di interazione più interessanti che coinvologono i componenti core del sistema:

- **Login di un utente** (Figura 3.15)
- **Logout di un utente** (Figura 3.16)
- **Registrazione di un utente** (Figura 3.17)
- **Partecipazione ad una stanza** (Figura 3.18)

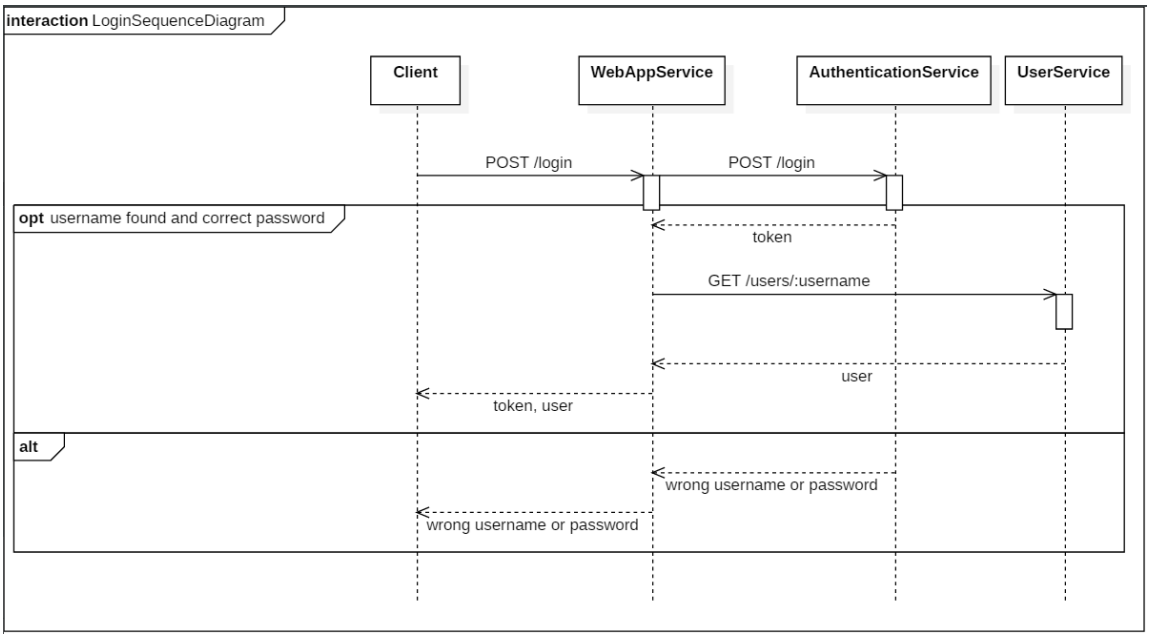

Figura 3.15: Diagramma di sequenza del login di un utente.

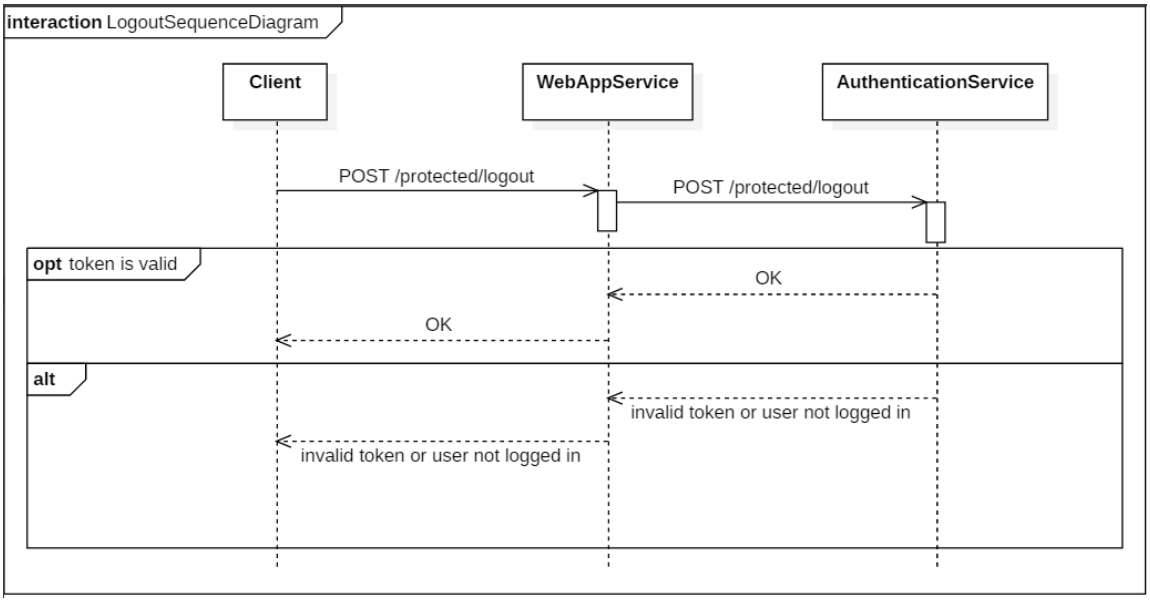

Figura 3.16: Diagramma di sequenza del logout di un utente.

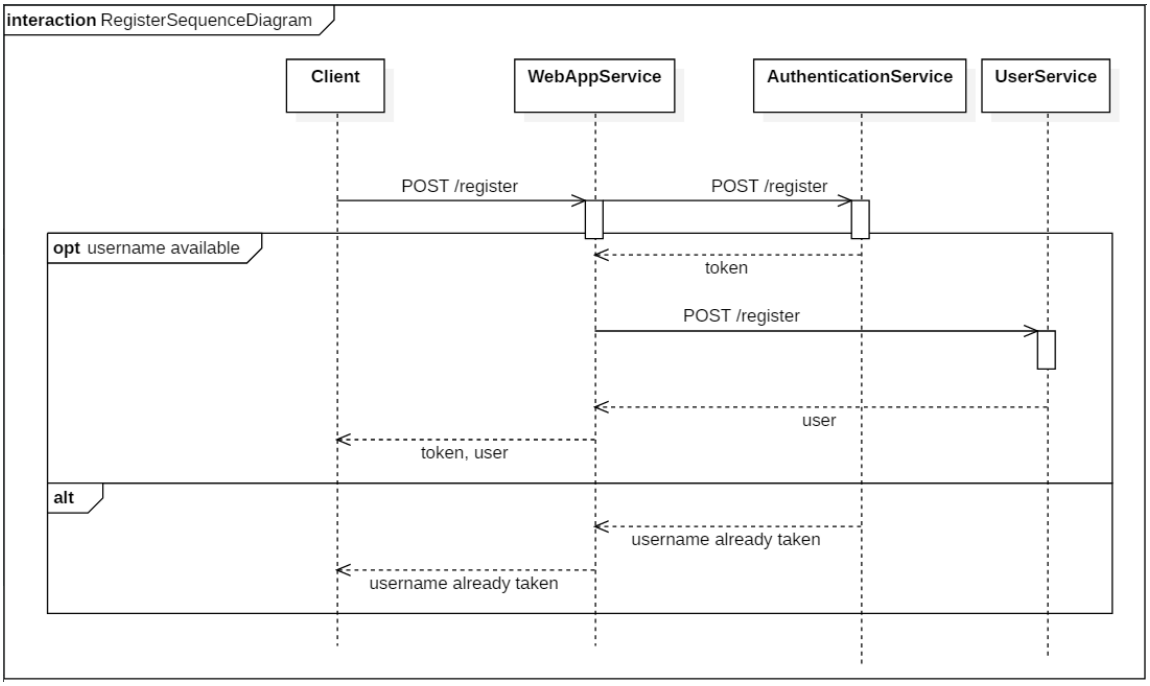

Figura 3.17: Diagramma di sequenza della registrazione di un utente.

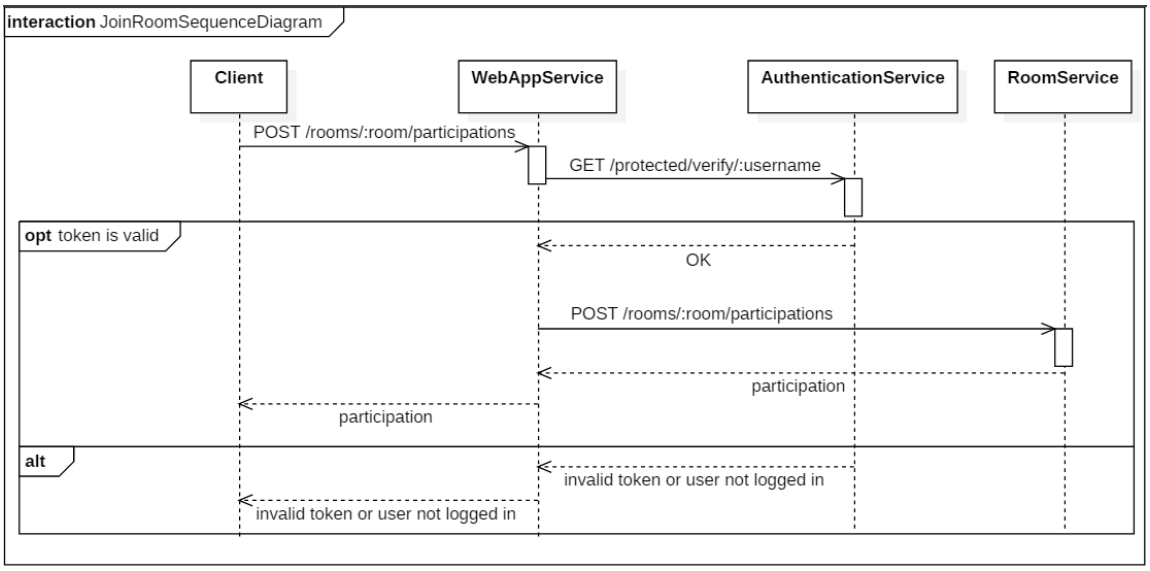

Figura 3.18: Diagramma di sequenza dell'unione ad una stanza da parte di un utente.

3.3 Scelte tecnologiche cruciali ai fini architetturali

3.3.1 Vertx

Vertx è un framework *Event-Driven* poliglotta, utilizzabile per costruire applicazioni reattive sulla JVM. È inoltre non bloccante, modulare e particolarmente veloce, favorendo così la concorrenza e la scalabilità.
Il seguente è un elenco delle principali funzionalità e componenti sfruttati, resi disponibili da Vertx:

- *Verticle*: Vertx permette l'uso di un modello di concorrenza e deployment semplice, scalabile e Actor-like, realizzabile facendo uso dei Verticles. Un'applicazione Vertx è formata da uno o più Verticles, i quali possono essere *Standard*, ovvero basati su un event loop, oppure *Worker*, usando così uno dei thread dal Worker Pool, oppure *Multi-threaded worker*, rendendo il codice eseguibile da più threads;
- *EventBus*: è il "sistema nervoso" di Vertx, infatti permette diverse parti dell'applicazione di comunicare tra loro tramite scambio di messaggi asincrono, indipendentemente dal loro linguaggio e istanza Vertx in cui eseguono. L'EventBus supporta publish/subscribe e messaggi sia punto a punto sia request-response; esso è suddiviso in canali, potendo così registrare *handlers* e pubblicare messaggi per uno specifico topic. È inoltre possibile creare un "ponte" con Javascript, rendendo dunque possibile l'uso dell'EventBus nei client Web: anche quest'ultima funzionalità è stata utilizzata;
- *Clustering*: con questa funzionalità è possibile eseguire un insieme di istanze con capacità di alta disponibilità, dati distribuiti ed un Event Bus distribuito. Ciò è necessario per fare il deploy di Verticles su più istanze di Vertx, permettendo loro di collaborare;
- *Vertx-Web*: è un insieme di funzionalità per la costruzione di applicazioni web, facilitando il routing, il passaggio di parametri, la gestione di cookies e sessioni, ecc;
- *Service Discovery*: questo componente fornisce un'infrastruttura per pubblicare e scoprire servizi e microservizi, i quali possono contenere risorse di varia natura;
- *Vertx-Jdbc*: è un client che permette di interagire con ogni database compatibile con JDBC, usando un'API asincrona.

Vertx è stato dunque scelto per le sue spiccate caratteristiche di velocità, scalabilità e capacità di fornire costrutti di alto livello utili e potenti.

3.3.2 Angular

Angular 2+ (o semplicemente Angular) è una piattaforma open source per lo sviluppo di applicazioni web con licenza *MIT*, evoluzione di *AngularJS*. E' stato sviluppato principalmente da Google, la sua prima release è avvenuta il 14 settembre 2016. Il linguaggio di programmazione usato per *Angular* è *TypeScript*.

Le applicazioni sviluppate in Angular vengono eseguite interamente dal web browser dopo essere state scaricate dal web server. Questo comporta il risparmio di numerose richieste che dovrebbero

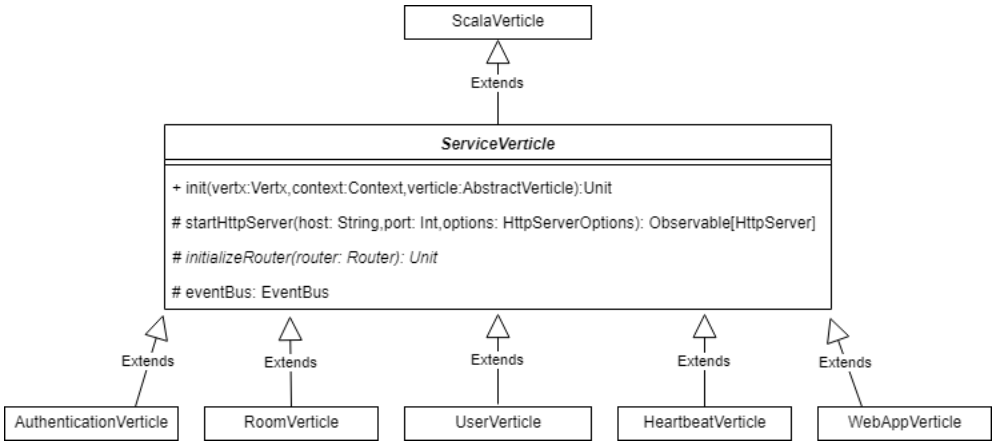

Figura 3.19: Diagramma delle classi che mostra le principali classi del progetto che estendono da *Verticle*.

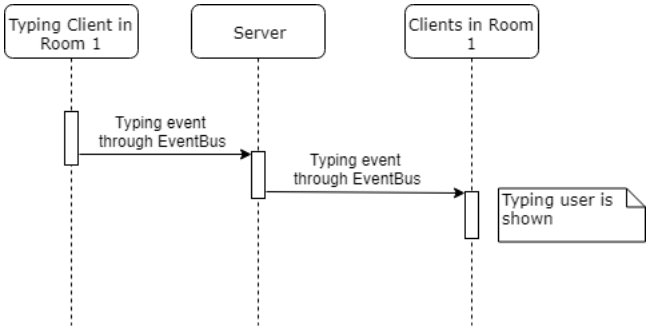

Figura 3.20: Diagramma di sequenza di un esempio di uso dell'*EventBus*: l'aggiornamento in real-time dello stato di scrittura di un utente in una stanza.

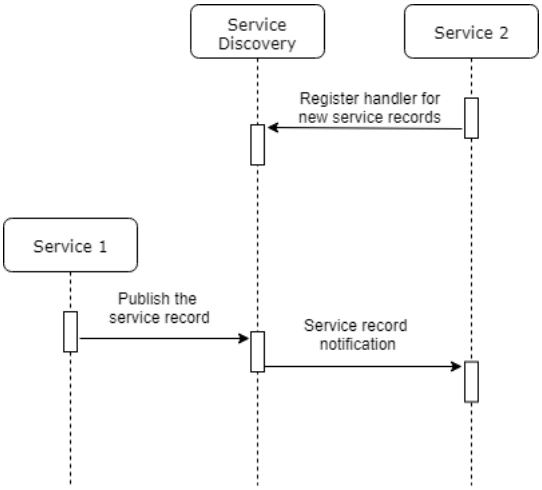

Figura 3.21: Diagramma di sequenza di un esempio di uso del servizio di *Service Discovery*.

essere altrimenti effettuate ogni volta che c'è una richiesta di azione da parte dell'utente. Il codice generato da Angular gira su tutti i principali web browser moderni quali ad esempio *Chrome*, *Microsoft Edge*, *Opera*, *Firefox*, *Safari*, ecc. I vantaggi dell'utilizzo di tale framework sono davvero considerevoli:

- **TypeScript**: Le applicazioni *Angular* sono costruite usando il linguaggio *TypeScript*, un linguaggio costruito sopra JavaScript, che garantisce maggiore sicurezza in quanto supporta tipi (primitive, interfacce, ecc.) in fase di compilazione. Inoltre, aiuta a catturare ed eliminare gli errori in anticipo durante la scrittura del codice o l'esecuzione di attività di manutenzione.
- **UI espressa in modo dichiarativo** tramite l'uso di *HTML*.
- **POJO**: Con Angular, non sono necessarie ulteriori funzioni getter e setter. Dato che ogni oggetto che usa è POJO (Plain Old JavaScript Object), che abilita la manipolazione degli oggetti fornendo tutte le funzionalità JavaScript convenzionali. È possibile rimuovere o aggiungere proprietà dagli oggetti, oltre a eseguire il loop su questi oggetti quando richiesto.
- **Pattern MVC incorporato**: lo sviluppatore *Angular* non deve creare i classici blocchi previsti dall'architettura *MVC* ma deve solo preoccuparsi di suddividere i componenti SW secondo i concetti forniti dal framework.
- **Struttura modulare**: realizzata tramite la divisione del SW in *components*, *services* e *templates*. In Figura 3.22 si può vedere un esempio di servizio *Angular*. In particolare è stato scelto di rappresentare la struttura e le dipendenze del *ChatService*. Si riporta, in Figura 3.23, anche un esempio di un componente *Angular*. Nello specifico, è stato rappresentato il componente *room-info*, responsabile delle visualizzazione delle informazioni riguardanti una stanza.
- **Consistenza del codice**

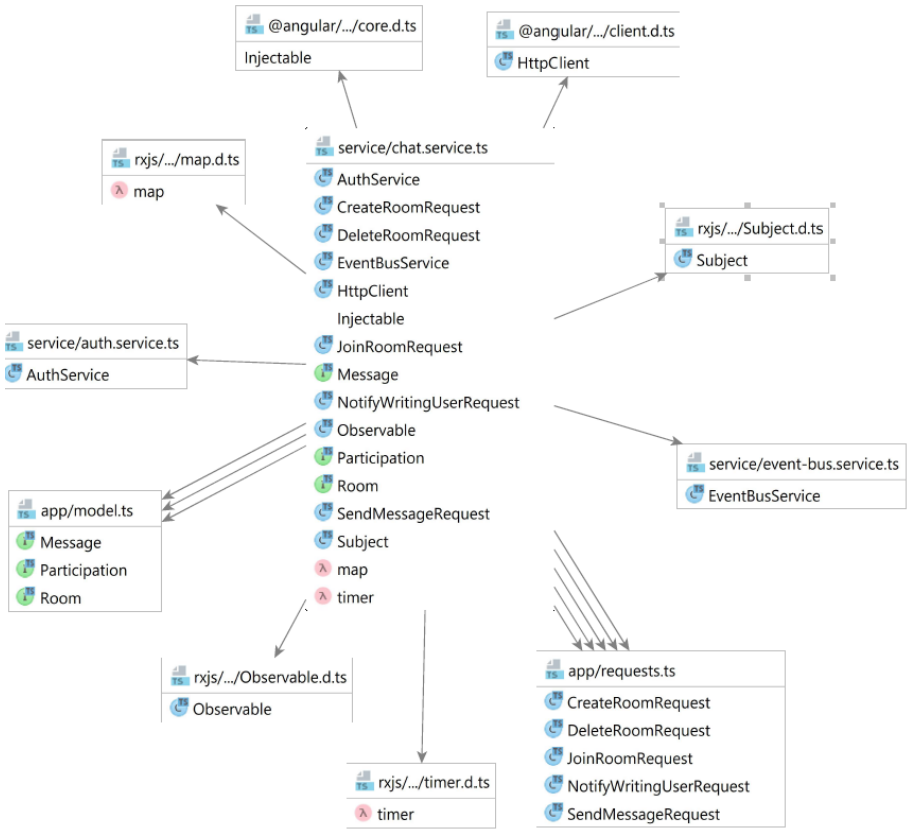

Figura 3.22: Diagramma delle dipendenze del servizio *Angular ChatService*.

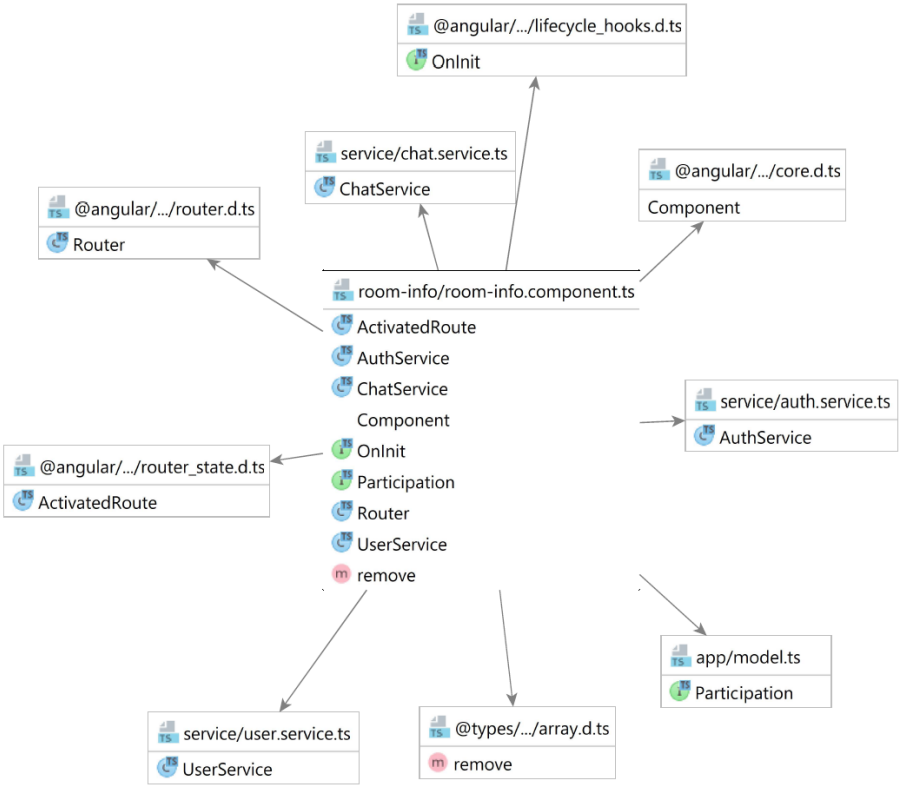

Figura 3.23: Diagramma delle dipendenze del componente *Angular Room-info*.

  - **Riusabilità**
  - **Migliore leggibilità**
  - **Facilità di manutenzione**
  - **Unit-Testing semplificato**

- **Performance**: in quanto *Angular* mette a disposizione una modalità di deployment detta *Production* nella quale esegue una serie di ottimizzazioni importanti (e.g. minificazione dei file css, js, ecc.).

Capitolo 4

Design di Dettaglio

In questo capitolo vengono analizzate le scelte operate a livello di design nel dettaglio. In particolare viene analizzata la struttura data a ciascuno dei servizi ed i principali pattern di progettazione adottati.

4.1 Suddivisione in moduli

Il nostro progetto è stato suddiviso in moduli in quanto permette un'organizzazione *by-need* dei singoli moduli. Questo permette ad ogni modulo di includere come dipendenza solo i moduli di cui necessita, ottimizzando l'incapsulamento.
I moduli definiti sono:

- *Commons*: contiene tutte le parti di codice riutilizzate all'interno dei diversi moduli.
- *Data Access*: contiene tutte le parti di codice riutilizzate per l'accesso (in lettura e scrittura) ad una base di dati relazionale.
- *Exceptions*: contiene una rappresentazione di tutte le eccezioni di dominio e fornisce costrutti utili per la loro gestione.
- *Interactors*: contiene i costrutti comuni per la creazione di casi d'uso.
- *Service Commons*: contiene i costrutti comuni per la creazione, pubblicazione e scoperta di servizi.

4.2 Parti comuni

In questa sezione vengono descritte le parti di codice comune che vengono riutilizzate all'interno di altri moduli.

4.2.1 Commons

All'interno del modulo *Commons* sono state fattorizzate principalmente:

- Alcune funzionalità generiche:
  - logging;
  - lettura di file.
- Funzionalità per agevolare la trasformazione da oggetti descritti in formato *JSON* a costrutti Scala.
- Funzionalità per la conversione da costrutti di Vert.x a `Scheduler` RxScala e classi implicite per arricchire gli `Observable` RxScala con funzionalità aggiuntive.
- Funzionalità per convertire una chiamata asincrona ad un metodo fornito da Vert.x in un `Observable` RxScala, metodi di conversione e utilizzo del pattern **Pimp my Library** per arricchire di nuove funzionalità le classi `AsyncResult`, `EventBus` e `Json`.
- Funzionalità per agevolare l'estrazione di parametri dal contesto di una richiesta HTTP e una classe implicita per estendere il costrutto `HttpServerResponse` di Vert.x.
- Altri costrutti più avanzati sono descritti in dettaglio nelle successive sotto-sezioni.

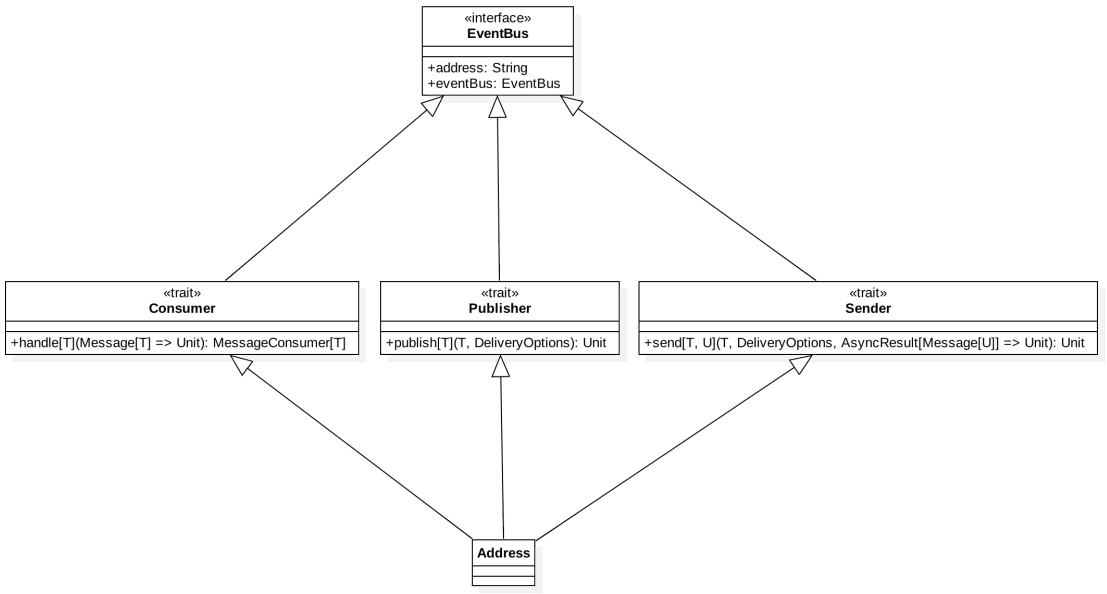

Figura 4.1: Utilizzo del *mix-in* per aggiungere funzionalità alla classe `Address`.

4.2.1.1 Validazione

Per realizzare la validazione dell'input utente è stato utilizzato il pattern *builder*. La scelta di utilizzare questo pattern progettuale è legata al fatto che non è possibile determinare a priori quante (e quali) regole dovranno essere applicate per effettuare la validazione di un input.

La classe `ValidatorBuilder` è incaricata di gestire la costruzione di un oggetto `Validator`, per fare ciò ò espone il metodo `addRule` che permette di aggiungere al builder un ulteriore regola di validazione. `ValidatorBuilder` espone due overload del metodo `addRule`, il primo prende in input una regola da applicare, il secondo prende in input un predicato e un'eccezione da sollevare nel caso l'oggetto passato in input al validatore non sia soddisfatta.

Per la creazione di oggetti di tipo `Validator` è stato utilizzato il pattern *Execute Around* che permette, tramite un espressione lambda, di eseguire un blocco di codice ed incapsulare all'interno del metodo eventuali operazioni di creazione e *clean-up*.

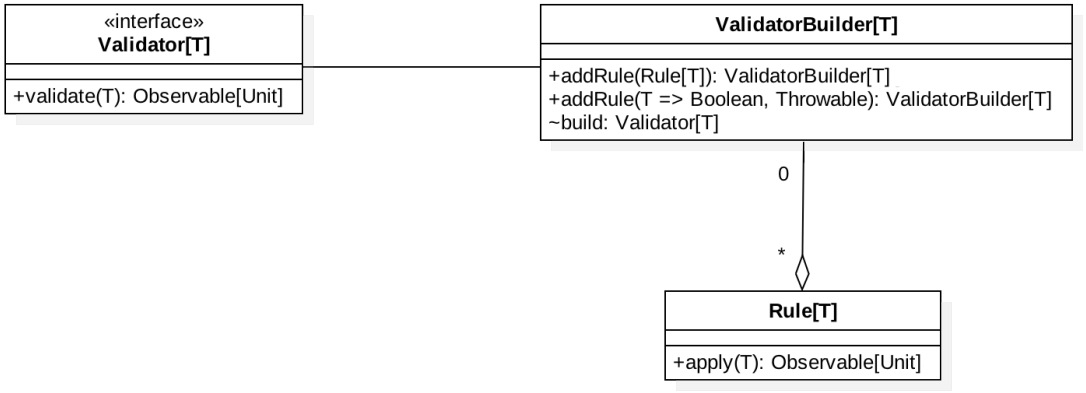

Figura 4.2: Diagramma delle classi per la classe per la costruzione di un `Validator`.

4.2.1.2 Configurazione

Per gestire la configurazione di oggetti complessi è stata utilizzata la classe `Configurator`, la quale permette di applicare in cascata un insieme di funzioni ad un'istanza passata in input.

4.2.2 Data Access

All'interno del modulo *Data Access* sono state inserite tutte le parti di codice che sono state riutilizzate all'interno di altri moduli per semplificare l'interazione con la base di dati sottostante, inoltre espone alcune conversioni per agevolare la traduzione di tipi di dato Scala in tipi di dato interpretabili MySQL (nello specifico `Boolean` e `java.util.Date`). Utilizzo del pattern **Pimp my Library** per arricchire la classe `ResultSet` aggiungendovi un metodo che permette di ottenere una rappresentazione JSON di ogni record restituito dall'interrogazione.

4.2.3 Exceptions

All'interno del modulo *Exceptions* sono contenute tutte le possibili eccezioni che l'applicazione potrebbe sollevare. Fornisce metodi utili per la conversione delle eccezioni in oggetti JSON (e viceversa), oltre ad un metodo per determinare il tipo di errore HTTP a partire dall'eccezione sollevata.

4.2.4 Interactors

All'interno del modulo *Interactors* sono contenuti i costrutti comuni per la creazione di casi d'uso.

La classe `UseCase`, classe padre di tutti i casi d'uso, sfrutta il pattern progettuale *Template method* per raccogliere a fattor comune le modalità di interazione e lasciare alle sottoclassi la definizione del comportamento.

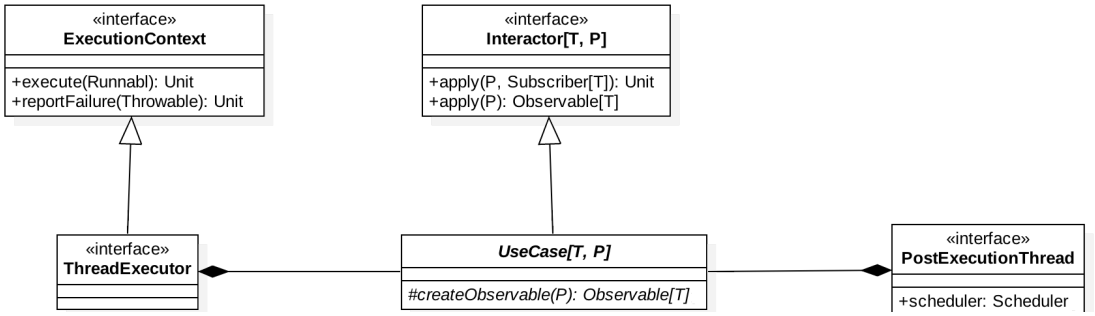

Figura 4.3: Diagramma delle classi per la classe `UseCase`.

4.2.5 Service Commons

All'interno del modulo *Service Commons* sono contenuti i costrutti comuni per la definizione, pubblicazione e scoperta di un servizio.

È utilizzato il pattern *Template method* nella classe `ServiceVerticle` al fine di raccogliere a fattor comune le modalità di inizializzazione del componente e demandare alle sottoclassi la definizione delle proprie rotte.

4.3 Suddivisione dei servizi

Nel nostro progetto abbiamo rispettato il pattern architetturale di suddivisione del progetto a *microservizi* (come discusso nel capitolo 3), in quanto fornisce notevoli vantaggi rispetto al classico approccio dell'applicativo server monolitico. Di seguito verranno elencati tutti i servizi del progetto, per poi andare ad analizzarli nel dettaglio nelle sezioni successive:

- *Authentication Service*: gestisce l'accesso, la registrazione, il logout e l'autenticazione delle richieste effettuate da un utente.
- *Room Service*: gestisce la creazione e l'eliminazione di una stanza, la partecipazione di un utente in una stanza e la persistenza dei messaggi scambiati in ciascuna stanza.
- *User Service*: gestisce la creazione, eliminazione e modifica dei profili utente.
- *Web App Service*: funge da intermediario fra i client e i servizi di cui sopra.

4.3.1 Scelte rilevanti

Durante la fase di analisi del problema abbiamo ritenuto opportuno che ogni microservizio avesse una sua specifica visione del dominio, in modo tale che ognuno lo vedesse al proprio livello di dettaglio d'interesse. Questa scelta ha portato ad una duplicazione consapevole di alcune entità del dominio.
Per validare le richieste ricevute i servizi usufruiscono di casi d'uso ausiliari. Tale scelta consente di non "sporcare" l'implementazione del caso d'uso principale e favorire il riuso dello stesso validatore per richieste effettuate a casi d'uso diversi.
Laddove si è ritenuto vantaggioso è stato applicato il pattern **Factory Method** utilizzando i *Companion Object* di *Scala*, in particolare.

4.4 Authentication Service

Authentication Service espone le funzionalità necessarie per gestire l'autenticazione degli utenti nel sistema. In particolare si occupa della registrazione, del login, del logout e della verifica di validità dei *token* di autenticazione.
Questo servizio è autonomo e non dipende da altri microservizi.
Le risorse messe a disposizione dal servizio, specificate nella tabella 4.1, permettono di:

- **Registrazione** di un utente: l'utente fornisce username e password e chiede di registrarsi. Se l'operazione viene conclusa con successo il servizio risponde confermando la creazione *Create* e inviando un *token di autenticazione* all'utente; altrimenti la richiesta viene rifiutata con messaggio di errore *Bad Request*. Se username o password non sono presenti nella richiesta, il servizio restituisce il messaggio di errore *Precondition Failed*.
- **Eliminazione** di un utente: l'utente invia la richiesta fornendo un *token* di autenticazione. Se la richiesta viene soddisfatta il servizio conferma con un messaggio di accettazione con codice *OK* altrimenti se il token non è valido il servizio risponde con un messaggio di errore *Unauthorized*. Se l'operazione non viene conclusa correttamente il servizio risponde con *Bad Request*. Se il token non è presente il servizio restituisce il messaggio di errore *Precondition Failed*.
- **Login** di un utente: l'utente fornisce la coppia username e password ed il servizio, se username e password sono corretti, il servizio risponde inviando il *token di autenticazione*. Se le

credenziali non sono valide, viene inviato un messaggio di errore *Unauthorized*. Se l'operazione non va a buon fine il servizio risponde con *Bad Request*. Se username o password non sono presenti il servizio restituisce il messaggio di errore *Precondition Failed*.

- Effettuare il **logout** di un utente: l'utente fornisce il proprio username e comunica la volontà di disconnettersi dall'applicazione. Se il token è valido, l'utente può disponnettersi e il servizio segna il token come non più utilizzabile, *invalid*. Se il token non è valido il servizio risponde con un messaggio di errore *Unauthorized*. Se l'operazione non viene conclusa correttamente il servizio risponde con *Bad Request*. Se il token non è presente il servizio restituisce il messaggio di errore *Precondition Failed*.

- **Validità** di un token: questa funzionalità è utilizzata dagli altri servizi per controllare che l'utente sia effettivamente presente nel sistema e che il suo token sia valido. Se il token è valido il servizio risponde con un messaggio di conferma con codice *OK*. Se il token non è valido il servizio risponde con un messaggio di errore *Unauthorized*. Se l'operazione non va a buon fine il servizio risponde con *Bad Request*. Se lo username o il token non sono presenti, oppure se il nome utente non coincide con quello presente nel token, il servizio restituisce il messaggio di errore *Precondition Failed*.

HTTP verb	Resource	Header	Request Body	Response Body	Responses	Description
POST	/register	-	username, password	token	201, 412, 500	Registrazione utente
DELETE	/protected/users/:username	token	-	-	200, 401, 412, 500	Eliminazione utente
POST	/login	-	username, password	token	200, 412, 500	Login utente
DELETE	/protected/logout	token	-	-	200, 401, 412, 500	Logout utente
GET	/protected/verify/:username	token	-	-	200, 401, 412, 500	Verifica token utente

Tabella 4.1: Risorse esposte dall'*Authentication Service*.

Figura 4.4: Diagramma E/R del servizio *Authentication Service*.

4.5 Room Service

Room Service è il servizio che gestisce le *chat room* di conversazione degli utenti. In particolare si occupa della creazione, eliminazione, adesione alle stanze e della persistenza dei messaggi. Questo servizio è autonomo, non dipende da altri servizi.
Le risorse messe a disposizione dal servizio, specificate nella tabella 4.2, permettono di:

- **Creare** una stanza: l'utente fornisce il proprio *username* e il *nome* della stanza che vuole creare. Se l'operazione viene conclusa con successo il servizio risponde confermando la creazione *Create* e inviando il *nome* della stanza appena creata. Se l'operazione fallisce viene inviato un messaggio di errore *Internal Server Error*. Se esiste già una stanza con quel nome, *Room Service* risponde con messaggio di errore *Conflict*. Se *username* e/o *name* non sono presenti nella richiesta, il servizio restituisce il messaggio di errore *Precondition Failed*.

- **Eliminare** una stanza: l'utente fornisce il proprio *username* e il *nome* della stanza come *parametri* della richiesta. Il servizio controlla l'effettiva esistenza della stanza: se esiste il servizio conferma l'eliminazione con messaggio di accettazione *OK*, altrimenti risponde con messaggio di errore *Not Found*. Se *username* e/o *name* non sono presenti nella richiesta, il servizio restituisce il messaggio di errore *Precondition Failed*. Se l'operazione non viene conclusa correttamente il servizio risponde con *Internal Server Error*.

- **Unirsi** ad una stanza: l'utente fornisce il proprio *username* e il *nome* della stanza come *parametri* della richiesta. Se la stanza esiste, il servizio conferma la creazione della partecipazione dell'utente nella stanza *Create*, altrimenti risponde con messaggio di errore *Not Found*. Se *username* e/o *name* non sono presenti nella richiesta, il servizio restituisce il messaggio di errore *Precondition Failed*. Se l'operazione non va a buon fine il servizio risponde con *Internal Server Error*.

- **Abbandonare** una stanza: l'utente fornisce il proprio *username* e il *name* della stanza che vuole abbandonare. Se il token è valido, l'utente può disponnettersi e il servizio segna il token come non più utilizzabile, *invalid*. Se i valori richiesti non sono presenti, il servizio risponde con un messaggio di errore *Precondition Failed*. Se lo *username* dell'utente o il *name* della stanza non risultano presenti nel *database*, il servizio restituisce il messaggio di errore *Not Found*. Se l'operazione non viene conclusa correttamente il servizio risponde con *Internal Server Error*.

- **Inviare** messaggi in una stanza: questa funzionalità permette la condivisione dei messaggi all'interno delle stanze. Infatti all'invio di un messaggio, viene creata una richiesta con *username* dell'utente e *content* del messaggio. Se l'operazione va a buon fine *Create*, il *RoomService* genera una risposta con il *content* e il *timestamp* del messaggio, il *name* della stanza e lo *username* dell'utente. Se i campi richiesti non sono presenti o non sono corretti, il servizio risponde rispettivamente con il messaggio di errore *Precondition Failed*, oppure con messaggio di errore *Not Found*. Se l'operazione non va a buon fine il servizio risponde con *Internal Server Error*.

HTTP verb	Resource	Request Body	Response Body	Responses	Description
POST	/rooms/	username, name	name	201, 412, 500	Creazione stanza
GET	/rooms/	-	list of Room	200	Recupera la lista delle stanze
DELETE	/rooms/:name	username	name	200, 404, 412, 500	Eliminazione stanza
POST	/rooms/:name/participations	username	participation	201, 404, 412, 500	Adesione alla stanza
GET	/rooms/:name/participations	-	list of Participations	200	Recupera la lista delle partecipazioni della stanza data
GET	/users/:username/participations	-	list of Participations	200	Recupera la lista delle partecipazioni dell'utente dato
DELETE	rooms/:name/participations/:username	-	name, username	200, 404, 412, 500	Abbandono della stanza
POST	rooms/:name/messages	content, username	content, timestamp, username, name	201, 404, 412, 500	Invio messaggio
GET	rooms/:name/messages	-	list of Messages	200	Recupera la lista dei messaggi presenti nella stanza data

Tabella 4.2: Risorse esposte dal *Room Service*.

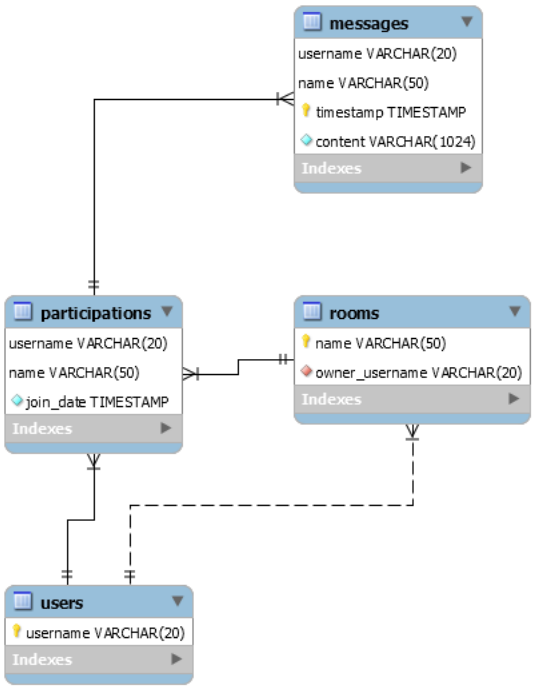

Figura 4.5: Diagramma E/R del servizio *Room Service*.

4.6 User Service

User Service è il servizio che gestisce il *profilo* di ogni utente registrato all'applicazione. In particolare si occupa della creazione, eliminazione e aggiornamento dei profili utente. Questo servizio è autonomo, non dipende da altri servizi.
Le risorse messe a disposizione dal servizio, specificate nella tabella 4.3, permettono di:

- **Creare** un utente: l'utente fornisce il proprio *nome utente*, *nome* e *cognome*. Se l'operazione viene conclusa con successo il servizio risponde confermando la creazione con il messaggio *Create* e inviando il *profilo* dell'utente appena creato. Se l'operazione fallisce il servizio risponde con il messaggio di errore *Internal Server Error*. Se esiste già un utente col *nome utente* fornito in input il servizio risponde con il messaggio di errore *Conflict*. Se *nome utente*, *nome*, o *cognome* non sono presenti nella richiesta il servizio risponde con il messaggio di errore *Precondition Failed*.

- **Eliminare** un utente: l'utente fornisce il proprio *nome utente*. Se l'operazione viene conclusa con successo il servizio risponde confermando l'eliminazione dell'utente con il messaggio *(OK)* e inviando il *nome utente* dell'utente appena eliminato. Se l'operazione fallisce il servizio risponde con il messaggio di errore *Internal Server Error*. Se non esiste un utente con il *nome utente* specificato il servizio risponde con il messaggio di errore *Not Found*. Se il *nome utente* non è presente nella richiesta il servizio risponde con il messaggio di errore *Precondition Failed*.

- **Aggiornare** il profilo utente: l'utente fornisce il proprio *nome utente* oltre al nuovo *nome*, *cognome* e *biografia*. Se l'operazione viene conclusa con successo il servizio risponde confermando l'aggiornamento del profilo utente rispondendo con il messaggio *OK* e inviando il *profilo* aggiornato dell'utente che ha effettuato la richiesta. Se l'operazione fallisce il servizio risponde con il messaggio di errore *Internal Server Error*. Se non esiste un utente con il *nome utente* specificato il servizio risponde con il messaggio di errore *Not Found*. Se *nome utente*, *nome*, *cognome* o *biografia* non sono presenti nella richiesta il servizio risponde con il messaggio di errore *Precondition Failed*.

- **Recuperare** il profilo utente: l'utente fornisce il proprio *nome utente*. Se l'operazione viene conclusa con successo il servizio risponde confermando l'aggiornamento del profilo utente rispondendo con il messaggio *OK* e inviando il profilo dell'utente che ha effettuato la richiesta. Se l'operazione fallisce il servizio risponde con il messaggio di errore *Internal Server Error*. Se non esiste un utente con il *nome utente* specificato il servizio risponde con il messaggio di errore *Not Found*. Se *nome utente* non è presente nella richiesta il servizio risponde con il messaggio di errore *Precondition Failed*.

HTTP verb	Resource	Header	Request Body	Response Body	Responses	Description
POST	/users	-	username, firstName, lastName	User profile	201, 409, 412, 500	Creazione di un utente
GET	/users/:username	-	-	User profile	200, 404, 412, 500	Recuperare il profilo di un utente
PUT	/users/:username	-	firstName, lastName, bio, visible	User profile	200, 404, 412, 500	Aggiornamento del profilo di un utente
DELETE	/users/:username	-	-	username	200, 404, 412, 500	Eliminazione di un utente
PUT	/users/:username/access	-	-	-	200, 500	Aggiornamento ultimo accesso

Tabella 4.3: Risorse esposte dallo *User Service*.

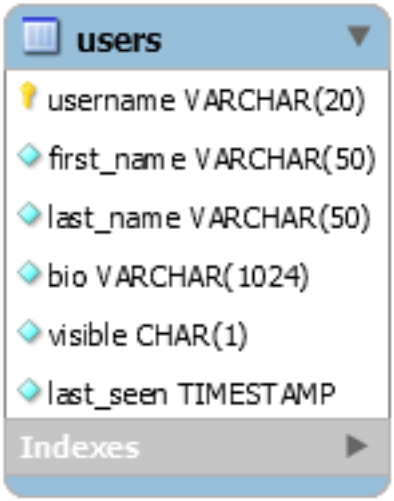

Figura 4.6: Diagramma E/R del servizio *User Service*.

4.7 Web App Service

Web App Service è il servizio che si occupa di inoltrare le richieste effettuate dai client ai relativi servizi, di gestire la parte di aggiornamento in tempo reale dei singoli client (per es. propaganzione di un messaggio a tutti gli utenti attivi presenti all'interno di una stanza) e di servire l'applicazione web all'utente.

Per erogare tutte le funzionalità esposte il *Web App Service* si appoggia sui servizi di back-end descritti in precedenza. Per esempio per registrare un utente si appoggia sull'*Authentication Service* per archiviare le credenziali di autenticazione e sullo *User Service* per archiviare il profilo utente.

Le risorse messe a disposizione dal servizio sono specificate nella tabella 4.4.

HTTP verb	Resource	Header	Request Body	Response Body	Responses	Description
POST	/register	-	username, firtname, lastname, password	User	201, 412, 500	Creazione utente
POST	/login	-	username, password	User	200, 404, 412	Login utente
DELETE	/logout	token	username	-	204, 401, 404, 412, 500	Logout utente
GET	/rooms	token	-	list of Rooms	200, 401, 404, 412, 500	Recupera la lista delle stanze in cui l'utente non partecipa
POST	/rooms	token	usernane, name	name	200, 401, 404, 412, 500	Creazione stanza
DELETE	/rooms/:name	token	username	-	204 401, 404, 412, 500	Eliminazione stanza
POST	/rooms/:name/participations	token	username	Participation	201, 401, 404, 412, 500	Adesione alla stanza
GET	/rooms/:name/participations	token	-	list of Participations	200, 401, 404, 412, 500	Recupera la lista delle partecipazioni della stanza data
DELETE	rooms/:name/participations/:username	token	-	name, username	200, 404, 412, 500	Abbandono della stanza
POST	rooms/:name/messages	token	content, username	Message	201, 404, 412, 500	Invio messaggio
GET	rooms/:name/messages	token	-	list of Messages	200, 401, 404, 412, 500	Recupera la lista dei messaggi presenti nella stanza data
GET	/users/:username	token	-	User	200, 404, 412, 500	Recupera l'utente dato il suo username
PUT	/users/:username	token	firtname, lastname, bio, visible	User	200, 404, 412, 500	Update utente
DELETE	/users/:username	token	-	username	204, 401, 404, 412, 500	Eliminazione utente
GET	/users/:username/participations	token	-	list of Participations	200, 401, 404, 412, 500	Recupera la lista delle partecipazioni di un dato utente

Tabella 4.4: Risorse esposte dallo *User Service*.

4.8 Client

Il front-end del sistema è un'Applicazione Web realizzata in Angular e servita dal `WebAppService`. L'applicazione si presenta con una prima pagina di *login*, dove l'utente potrà inserire il suo *username* e la *password*. Nel caso l'utente non fosse ancora registrato, ha a possibilità di registrarsi e creare un proprio profilo.

Una volta effettuato l'accesso la pagina presenta: (1) un menù laterale in cui sono visibili le stanze a cui l'utente ha ito, (2) una funzionalità di *ricerca* delle stanze, (3) una funzionalità che permette la *creazione* di nuove stanze, (4) un bottone per il logout e la modifica del profilo utente, (5) la chat di messaggistica che mostra i messaggi scambiati dagli utenti, (6) un bottone che permette di vedere le informazioni riguardanti una stanza, in particolare i partecipanti alla chat e il loro profilo.

Capitolo 5

Implementazione

In questo capitolo sono analizzati nel dettaglio i contributi apportati da ciascun componente del gruppo.
Avendo impostato il sistema basandoci sulla *Clean Architecture*, in cui la modellazione delle funzionalità del sistema viene trattata in termini di casi d'uso (*interattori*), gran parte del carico di lavoro è stato suddiviso tra i membri del team attraverso la **spartizione dei casi d'uso** da implementare.
Per la creazione dei micro-servizi sono state utilizzate le librerie messe a disposizione da Vert.x.

5.1 Componenti creati in cooperazione

Bootstrapping dell'Architettura:

- Identificazione e design dei servizi richiesti
- Identificazione delle sorgenti necessari per ogni servizio e design dei *DB*
- Identificazione e preparazione della piattaforma DBaaS
- Creazione degli Use Case Diagram in UML
- Creazione dei Class Diagram in UML
- Creazione dei Sequence Diagram in UML

5.2 Alessandro

Lo studente *Alessandro Gnucci* si è occupato dello sviluppo dei seguenti items, definiti nel *Product Backlog*:

- **Architecture bootstrapping** (con il resto del team). Questa parte si articola in:
  - Identificazione dei servizi richiesti;

  - Identificazione delle sorgenti dati per i servizi e design dei *DB*;
  - Identificazione e preparazione della piattaforma *DBaaS*, inoltre design dell'*UML Use Case Diagram*.
  - Design dell'*UML Class Diagram*.
- **Registrazione di un utente**
  - Design del diagramma *UML* di sequenza (con il resto del team).
  - Implementazione *TDD* del *Authentication Service*.
- **Login di un utente**
  - Design del diagramma *UML* di sequenza (con il resto del team).
  - Implementazione *TDD* dell'*Authentication Service*.
- **Logout**
  - Design del diagramma *UML* di sequenza (con il resto del team).
  - Implementazione *TDD* dell'*Authentication Service*.
- **Controllo di validità dei tokens Jwt**: implementazione *TDD* dell'*Authentication Service*.
- **Rimozione di una stanza**:
  - implementazione *TDD* del *WebApp Service*.
  - implementazione *TDD* del *Room Service*.
  - Implementazione *TDD* del *Web Client*.
- **Ricerca di una stanza**: implementazione *TDD* del *Room Service*.
- **Cancellazione di un account**: implementazione *TDD* dell'*Authentication Service*.
- **Refactoring del codice dei servizi**: refactoring del codice dell'*Authentication Service*.
- **Abbandono da parte di un utente di una stanza**:
  - implementazione *TDD* del *WebApp Service*.
  - implementazione *TDD* del *Room Service*.
  - Implementazione *TDD* del *Web Client*.
- **Modifica del profilo utente**:
  - implementazione *TDD* del *WebApp Service*.
  - implementazione *TDD* del *User Service*.
  - Implementazione *TDD* del *Web Client*.
- **Aggiornamento in real-time sullo stato di scrittura degli utenti nelle stanze**:
  - implementazione *TDD* del *WebApp Service*.
  - Implementazione *TDD* del *Web Client*.

5.3 Nicola

All'interno del progetto *Distributed Chat Service*, lo studente *Nicola Piscaglia* si è occupato dello sviluppo dei seguenti item definiti nel *Product Backlog*:

- **Project bootstrapping**: in particolare dell'inizializzazione e configurazione di *Travis CI*.
- **Architecture bootstrapping** (in cooperazione con gli altri membri del team). In particolare questa fase si è articolata in:
  - Identificazione dei servizi richiesti (si è scelto di utilizzare l'architettura a microservizi).
  - Identificazione delle sorgenti dati necessarie per ogni servizio e design del *DB*.
  - Identificazione/creazione della piattaforma *DBaaS* e design dell'*UML Use Case Diagram*.
  - *UML Class Diagram* Design.
- **Registrazione di un utente**
  - Design del diagramma *UML* di sequenza (in cooperazione con gli altri membri del team).
  - Implementazione *TDD* del *WebApp Service*.
- **Login**
  - design del diagramma *UML* di sequenza (in cooperazione con gli altri membri del team).
  - Implementazione *TDD* del *WebApp Service*.
- **Logout**
  - Design del diagramma *UML* di sequenza (in cooperazione con gli altri membri del team).
  - Implementazione *TDD* del *WebApp Service*.
- **Creazione di una stanza**: implementazione *TDD* del *WebApp Service*.
- **Rimozione di una stanza**: implementazione del *Web Client*.
- **Validazione dell'input**: implementazione della validazione delle richieste di ogni servizio.
- **Partecipazione ad una stanza**:
  - implementazione del *Web Client*.
  - implementazione *TDD* del *WebApp Service*.
- **Refactoring del codice dei servizi**: refactoring del codice del *WebApp Service*.
- **Standardizzazione delle Rest API**: in particolare sono state uniformate le chiamate REST e le rotte in base al design formalizzato tramite specifica *Swagger*. Tali cambiamenti hanno impattato i seguenti componenti:
  - *WebApp Service*
  - *Room Service*

  - *User Service*
  - *Authentication Service*
  - *Web Client*

- **Visualizzazione delle informazioni di una stanza**
  - Implementazione *TDD* del *WebApp Service*.
  - Implementazione *TDD* del *Room Service*.
  - Implementazione del *Web Client*.

- **Visualizzazione del profilo di altri utenti**
  - Implementazione *TDD* del *WebApp Service*.
  - Implementazione *TDD* dello *User Service*.
  - Implementazione del *Web Client*.

5.4 Martina

All'interno del progetto *Distributed Chat Service* la studentessa *Magnani Martina* si è occupata dello sviluppo delle seguenti parti:

- Registrazione di un utente:
  - Implementazione TDD del micro-servizio *User Service*.
- Login di un utente:
  - Implementazione TDD del micro-servizio *User Service*.
- Creazione di una stanza:
  - Implementazione TDD del micro-servizio *Room Service*.
  - Implementazione del *Web Client* in cooperazione con *Vandi Mattia*.
  - Debug e testing (Sevice & Web Client) della funzionalità ai fini della *release*.
- Refactoring del codice dei servizi:
  - Refactoring del codice del micro-servizio *User Service*.
- Ricerca di una stanza:
  - Implementazione TDD del micro-servizio *WebApp Service*.
- Partecipazione ad una stanza:
  - Implementazione TDD del micro-servizio *Room Service*.
- Invio di un messaggio:

  - Implementazione TDD del micro-servizio *Room Service*.
  - Implementazione TDD del micro-servizio *WebApp Service*.
  - Implementazione del *Web Client*.
- Ricezione di un messaggio:
  - Implementazione TDD del micro-servizio *Room Service*.
  - Implementazione TDD del micro-servizio *WebApp Service*.
  - Implementazione del *Web Client*.
- Ricezione dei messaggi passati per le stanze in cui l'utente partecipa:
  - Implementazione TDD del micro-servizio *Room Service*.
  - Implementazione TDD del micro-servizio *WebApp Service*.
  - Implementazione del *Web Client*.

5.5 Mattia

All'interno del progetto *Distributed Chat Service* lo studente *Vandi Mattia* si è occupato dello sviluppo delle seguenti parti:

- Bootstrapping del progetto:
  - Inizializzazione del repository.
  - Inizializzazione della configurazione multi-progetto Gradle.
  - Inizializzazione di Trello.
- Registrazione di un utente:
  - Implementazione TDD nel micro-servizio *User Service*.
  - Implementazione del *Web Client*.
- Creazione di una stanza:
  - Implementazione del *Web Client*.
- Eliminazione di una stanza:
  - Implementazione TDD nel micro-servizio *Room Service*.
- Ricerca di una stanza:
  - Implementazione del *Web Client*.
- Refactoring del micro-servizio *Room Service*.
- Broadcasting degli eventi di creazione e eliminazione di una stanza.

  - Implementazione TDD nel micro-servizio *Web App Service*.
  - Implementazione del *Web Client*.
- Definizione di una descrizione formale dei micro-servizi utilizzando il tool Swagger.
- Invisibilità dell'utente:
  - Implementazione TDD nel micro-servizio *Room Service*.
  - Implementazione TDD nel micro-servizio *Web App Service*.
  - Implementazione del *Web Client*.

Capitolo 6

Retrospettiva

6.1 Processo di sviluppo

Il lavoro è stato organizzato secondo le filosofie della metodologia Agile, adottando un approccio *Scrum-like*. Non dispondendo della figura dello scrum manager il team si è autorganizzato e autogestito per la definizione dei task da portare a termine negli Sprint settimanai e per la loro distrubuzione tra i membri.

Al termine di ogni Sprint i componenti hanno preso parte a un meeting, durante il quale sono state poste in atto le fasi di *Sprint Review* (per valutare il risultato dello Sprint e stabilire le priorità delle prossime feature da portare a termine) e di *Sprint Retrospective* (per analizzare il processo di sviluppo allo scopo di migliorare la produttività del team).

Il codice è stato sviluppato usando tecniche di *Test-Driven-Development* (TDD).

Per aiutarci nella pianificazione degli Sprint settimanali abbiamo utilizzato *Google Sheets* per tenere traccia sia del product backlog complessivo sia del product backlog degli Sprint settimanali. Abbiamo utilizzato *Trello* per tracciare l'andamento dei task asseganti.

6.2 Andamento degli Sprint

6.2.1 Sprint 1

Le funzionalità che ci siamo prefissi di sviluppare all'interno di questo Sprint sono:

- Possibilità di registrazione di un nuovo utente.
- Possibilità di effettuare l'accesso da parte di un utente già presente all'interno del sistema.
- Possibilità, da parte di un utente che ha già effettuato l'accesso, di uscire dall'applicazione.

Nella fase iniziale dello *Sprint* ci siamo concentrati sul bootstrapping del progetto, in particolare è stato inizializzato il repository e gli strumenti di supporto alla *continuous integration* (Travis CI, Gradle, Trello e Google Sheets).

Successivamente siamo passati ad una fase di analisi e modellazione del problema, in particolare abbiamo identificato l'architettura del sistema e definito lo schema delle *basi di dati*. Sono stati prodotti i diagrammi UML di massima del dominio applicativo (casi d'uso e diagramma delle classi).

In seguito, abbiamo modellato ciascun caso d'uso di competenza dello Sprint con dei diagrammi di sequenza e l'architettura di ogni microservizio con diagrammi delle classi. Questa parte ha richiesto più tempo del previsto e ha sottratto diverso tempo alla fase successiva di implementazione di ogni servizio. Quest'ultima fase è stata iniziata ma non terminata all'interno di questo Sprint.

6.2.1.1 Retrospettiva

Analizzando il processo di sviluppo adottato in questo Sprint è emerso che alcuni task sono stati sottostimati. In particolare, abbiamo attribuito una grandezza inferiore alle fasi di modellazione e sviluppo, di conseguenza negli Sprint successivi verranno calibrati meglio i task per ogni sprint. In particolare task troppo onerosi saranno scomposti in sotto-task più piccoli da distribuire in più sprint.

6.2.2 Sprint 2

In questo Sprint è stata terminata l'implementazione lato *server* delle funzionalità che non erano state completate nello Sprint precedente ed è stato applicato in maniera sistematica lo *unit-testing* durante la fase di sviluppo.

Inoltre è stata iniziata l'implementazione del client in *Angular*: architettura generale, componenti principali e impostazione del *template*.

6.2.2.1 Retrospettiva

I risultati di questo Sprint sono stati abbastanza soddisfacenti, in quanto è stato applicato il processo di TDD per tutti i test delle Web API di ogni servizio.

A livello di processo di processo di sviluppo, negli Sprint successivi, saranno effettuate *pull request* più frequenti al fine di risolvere un minor numero di conflitti durante la fusione delle singole *pull request*.

6.2.3 Sprint 3

In questo Sprint sono state aggiunte le seguenti funzionalità:

- Aggiunta di una nuova stanza.
- Eliminazione di una stanza.
- Controllo della validità del token passato da un utente.

In questo Sprint ogni membro del team ha terminato i task assegnati. È stata ultimata l'implementazione dei microservizi in merito ai casi d'uso che erano stati prefissati.

6.2.3.1 Retrospettiva

Gli elementi assegnati a questa settimana si sono dimostrati correttamente quantificati per questo abbiamo ritenuto che l'aggiunta di ulteriori task per i successivi Sprint non sia necessaria. Tutti i membri sono riusciti a completare i propri compiti nell'arco della settimana.

Per lo Sprint successivo ci cercherò di portare a termine lo sviluppo di un prototipo funzionante in base ai casi d'uso implementati finora.

6.2.4 Sprint 4

In questo Sprint ci siamo dedicati al *debug* delle funzionalità sviluppate fino a questo momento, sia lato client sia lato servizi.

Abbiamo cambiato il *layout* dell'applicazione web applicando un nuovo *template* che consente una prototipazione più facile dell'interfaccia utente.

Abbiamo introdotto la validazione delle richieste effettuate dall'utente in ogni servizio.

È stato uno Sprint particolarmente impegnativo dal punto di vista del carico di lavoro; in particolare per la validazione dell'input delle richieste di ogni servizio. Siamo comunque riusciti a completare con successo tutti i tasks prefissati. È stato raggiunto l'obiettivo di realizzare un prototipo funzionante che implementasse i casi d'uso finora delineati.

6.2.4.1 Retrospettiva

I tasks assegnati sono stati completati anche se non è rimasto tempo per ulteriori migliorie o refactoring del codice. Nel prossimo Sprint vogliamo riservare spazio anche per aumentare il numero di test e riportare la qualità del codice ad un livello più alto.

6.2.5 Sprint 5

In questo Sprint sono state aggiunte le seguenti funzionalità:

- Ricerca di una stanza da parte di un utente.
- Possibilità, da parte di un utente, di unirsi ad una stanza già esistente.

È stato aggiunto il broadcasting dell'evento di creazione di una stanza, in modo tale che ogni client possa essere informato, durante la fase di ricerca di una stanza quando una nuova stanza è stata aggiunta senza aver bisogno di effettuare una nuova ricerca.

I task per questo Sprint sono stati completati con successo, a meno della ricerca delle stanze. In particolare è incompleta la parte client. È stata definita, inoltre, una specifica standard per le rotte dei servizi *REST*, tramite *Swagger*. Quindi nel prossimo Sprint, si dovranno adattare le rotte già implementate alla specifica delineata.

6.2.5.1 Retrospettiva

In questo Sprint è stato aggiornato il backlog con nuovi task in modo incrementale; l'obiettivo per lo Sprint successivo sarà quello di terminare i task prefissati al suo inizio, senza aggiungerne di nuovi nel corso dello Sprint. Complessivamente, lo Sprint si è concluso senza particolari problemi.

6.2.6 Sprint 6

In questo Sprint sono state aggiunte le seguenti funzionalità:

- Invio di un messaggio da parte di un utente.
- Ricezione di un messaggio da parte dell'utente.
- Possibilità da parte di utente di abbandonare una stanza.
- Possibilità da parte di utente di visualizzare le informazioni relative ad una stanza stanza.

È stata ultimata la standardizzazione delle interfacce *REST* di ogni servizio ed è stata completata la ricerca delle stanze da parte dell'utente.
È stato uno Sprint a carattere decisamente implementativo, durante il quale sono state sviluppate diverse feature fondamentali per la web application (e.g. invio/ricezione di un messaggio, ricerca di una stanza, visualizzazione dei dettagli di una stanza, ecc.). I compiti assegnati sono stati interamente completati.

6.2.6.1 Retrospettiva

Riteniamo che il team abbia previsto un carico di lavoro adeguato alle ore a disposizione. Lo sviluppo è stato particolarmente produttivo, grazie anche al maggior tempo a disposizione per l'implementazione di nuove feature. Non è stato riscontrato nessun problema particoalare. La sfida dei prossimi Sprint, a livello di processo, sarà definire un piano di lavoro che porti alla terminazione dell'implementazione del sistema e alla scrittura della relazione di progetto, in tempo utile per la consegna finale.

6.2.7 Sprint 7

In questo Sprint sono state aggiunte le seguenti funzionalità:

- Possibilità da parte di un utente di rendersi invisibile agli altri utenti.
- Possibilità di visualizzare il profilo degli altri utenti presenti in una chat.
- Possibilità di visualizzare in tempo reale l'aggioranmento sullo stato di scrittura da parte di utente.
- Possibilità di recuperare i messaggi precedentemente scambiati all'interno di una stanza.

In questo Sprint sono state sviluppate con successo tutte le funzionalità rimaste, escluse le opzionali, ottenendo così come risultato il sistema completo.

6.2.7.1 Retrospettiva

I tasks assegnati per questo Sprint si sono dimostrati correttamente quantificati. L'obbiettivo per lo Sprint successivo sarà quello di scrivere la relazione in maniera dettagliata, completa ed esaustiva.

6.2.8 Sprint 8

In questo Sprint il team si è dedicato alla stesura della Relazione di progetto. È stato deciso di dividere il team in gruppi di due persone, in modo tale che alla fine del lavoro ogni gruppo ha potuto revisionare la parte dell'altro. La relazione secondo le regole d'esame doveva essere formata da 6 capitoli; formando due gruppi, ognuno ha potuto scrivere esattamente 3 capitoli della relazione.

6.2.8.1 Retrospective

La divisione del lavoro è risultata corretta ai fini del completamento dello Sprint e della Relazione. Ci siamo resi conto che alcuni diagrammi UML realizzati in precedenza dovevano essere ampliati, in quanto definiti durante il primo Sprint e di conseguenza non coprivano ancora tutte le funzionalità che il sistema avrebbe implementato.

6.3 Commenti finali

Avendo impostato il progetto seguendo la filosofia della *Clean Architecture*, basata su una modellazione delle funzionalità in termini di casi d'uso (*interattori*), il nostro codice risulta molto pulito, leggibile, riutilizzabile e facilmente manutenibile.

L'utilizzo delle *pull request* messe a disposizione da *GitHub* è risultato molto efficace durante lo sviluppo, in quanto ciascun membro del team ha potuto lavorare autonomamente nel proprio repository. I vantaggi derivanti sono stati: (1) un minor numero di conflitti durante lo sviluppo concorrente, (2) il testing automatico di ciascuna feature prima di essere caricata (*Continuous Integration* di *TravisCI*) e (3) autonomia nello sviluppo.

Anche l'approccio Agile è risultato molto comodo, ci ha permesso di organizzare al meglio i task da svolgere e l'obiettivo di soddisfare gli Sprint ha motivato tutto il gruppo a lavorare regolarmente e costantemente. È stato riscontrato qualche problema solo durante le prime settimane, in quanto dovevamo capire il giusto peso da attribuire ai vari task e imparare il corretto approccio a questa metodologia di sviluppo, a noi nuova.

Ogni settimana, a partire dalla quarta, è stato rilasciato un prototipo di progetto che è stato arricchito di nuove funzionalità in maniera incrementale. Durante i primi Sprint ci siamo concentrati nella creazione della struttura base di tutti i servizi e del client. Successivamente abbiamo proceduto all'implementazione di tutti i casi d'uso seguendo una scaletta in ordine d'importanza decrescente.

Al termine del progetto il team si ritiene soddisfatto del lavoro svolto; tutti i requisiti funzionali sono stati completati ed è possibile utilizzare *Distributed Chat Service* in maniera distribuita con aggiornamenti in tempo reale. I servizi possono trovarsi su macchine differenti senza causare alcun problema all'esecuzione della chat.

6.4 Estensioni future

Pensando ad un prossimo sviluppo, ci piacerebbe estendere *Distributed Chat Service* con le seguenti funzionalità:

- Trasferimeto di file multimediali nelle chat.
- Suoni personalizzati all'arrivo dei messaggi a seconda del mittente e della stanza.
- Invio di note vocali.
- Integrazione di emoji.
- *Chatbot* helper e di compagnia.