Microchat:
la chat a microservizi

Thomas Angelini
thomas.angelini@studio.unibo.it

Luca Deluigi
luca.deluigi3@studio.unibo.it

Simone Magnani
simone.magnani4@studio.unibo.it

Febbraio 2022

1 Scopo e requisiti

Gli scopi del progetto possono essere suddivisi tra teorici e pratici. Gli scopi pratici sono relativi allo sviluppo del progetto nel rispetto dei suoi requisiti, mentre lo scopo sul piano teorico/accademico è quello di realizzare l'applicazione con un'architettura a microservizi che comunicano ad eventi, in modo scalabile. Tale infrastruttura rappresenta la frontiera dei sistemi distribuiti scalabili, modulari e cloud-friendly. Saranno inoltre analizzati i vantaggi di un protocollo real time per la comunicazione client-server quale ad esempio Web Socket.

1.1 Requisiti obbligatori

I seguenti requisiti sono imposti dal gruppo nella realizzazione del progetto affinché lo si possa considerare completato.

1.1.1 Funzionali

- possibilità di **dialogare con un utente** attraverso messaggi testuali.
- **La ricerca di un utente** per poter iniziare il dialogo.
- **Visualizzare il nome del profilo** con cui si sta comunicando.
- **Evidenziare le principali statistiche** della chat. Verranno mostrati:
  - *il numero totale dei messaggi scambiati nella chat.*
  - *Il numero medio giornaliero dei messaggi scambiati nella chat.*
  - *Il numero medio settimanale dei messaggi scambiati nella chat.*
- **Identificare la data di trasmissione del messaggio**.

1.1.2 Non funzionali

- il sistema sarà a **microservizi** che comunicano ad **eventi** asincroni (scambio di messaggi).
- **L'applicazione sarà scalabile orizzontalmente** in proporzione al numero di utenti.
- Il protocollo di **comunicazione client-server** sarà **bidirezionale e real time**.
- Integrazione nel progetto di **Continous Integration**, **Continous Delivery** e possibilmente **Continuous Deployment**.

1.2 Requisiti opzionali

In aggiunta a quanto elencato precedentemente è possibile la realizzazione dei seguenti obiettivi aggiuntivi.

1.2.1 Funzionali

- permettere all'utente di **rispondere ad uno specifico messaggio**. Tale funzionalità, come nel caso delle moderne applicazioni di messaggistica, consentirà di evidenziare il messaggio al quale l'utente vuole rispondere.
- Creazione di **chat di gruppo** dove il numero utenti può essere superiore a 2.
- Mostrare nella chat se l'utente con cui si dialoga **sta scrivendo un messaggio**.

2 Piano di lavoro

- CI/CD basata su GitHub Actions, così come l'organizzazione dei task, issue e pull request. GitHub sarà usato come hosting dei sorgenti principale per questo progetto.
- L'intero processo di sviluppo si ispira ai principi e pattern del Domain Driven Design[1] e ai pattern dei sistemi distribuiti scalabili
- Per i microservizi sarà usato il linguaggio C#, usando librerie di supporto quali ad esempio EasyDesk.CleanArchitecture, SignalR, ASP.NETCore, EFCore, Mediatr, Rebus...
- Il frontend sarà realizzato con Angular 13.

2.1 Suddivisione dei compiti

Dopo l'analisi del dominio e la progettazione della soluzione saranno delineati con maggior dettaglio i compiti specifici di ciascuno.

2.2 Organizzazione dello sviluppo

I membri del team lavoreranno sia in parallelo che in pair programming (specialmente all'inizio) organizzando i compiti in task settimanali/bisettimanali con un kanban presente sulla pagina del repository GitHub (GitHub Projects).

3 Analisi dei Requisiti

Il gruppo inizialmente si è riunito impersonando sia gli sviluppatori che gli stakeholder e gli esperti del dominio per individuare i principali casi d'uso del progetto. L'analisi è stata semplificata per poter sviluppare in tempi utili una Proof Of Concept del sistema con funzionalità di base. In ambito Domain Driven Design questa fase, da ripetere ad ogni iterazione di sviluppo, è detta *Knowledge Crunching*.

3.1 User Stories

Di seguito sono descritte le principali user stories di interazione con l'applicazione. Tale approccio consente di immedesimarsi negli attori del sistema ed ipotizzare tutte le possibili funzionalità da realizzare.

Registrazione Account "Come utente richiedo di *registrare un account* per sfruttare i servizi di Microchat, utilizzando email, password, username e opzionalmente anche nome e cognome."

Cancellazione di un Account "Come utente registrato, qualora non volessi più far parte del servizio Microchat, richiedo la possibilità di poter *cancellare il mio account* insieme a tutte le relative informazioni sensibili."

Accesso Acccount "Come utente registrato voglio poter *accedere* a Microchat con l' account da me creato, utilizzando l'email oppure lo username come identificativo."

Modifica delle credenziali di accesso "Come utente registrato richiedo di poter modificare le mie credenziali di accesso, ovvero *email*, *username* e *password*."

Modifica dei dati Utente "Come utente registrato richiedo di poter modificare i miei dati personali, ovvero *nome*, *cognome* e *username*."

Creazione di una Chat "Come utente registrato richiedo di poter *creare una chat* per poter comunicare con un altro utente registrato."

Spedizione di un messaggio in una chat "Come utente registrato richiedo di poter *inviare un messaggio* in una chat dove sono coinvolto."

Cancellazione di un messaggio "Come utente che ha spedito un messaggio richiedo di poterlo *eliminare*."

Modifica di un messaggio "Come utente che ha spedito un messaggio richiedo di poter *modificare il suo contenuto*."

3.2 Diagramma dei casi d'uso

L'unico attore nel sistema è quindi **User** (l'utente normale) il quale potrà svolgere le azioni esplicitate a figura 1.

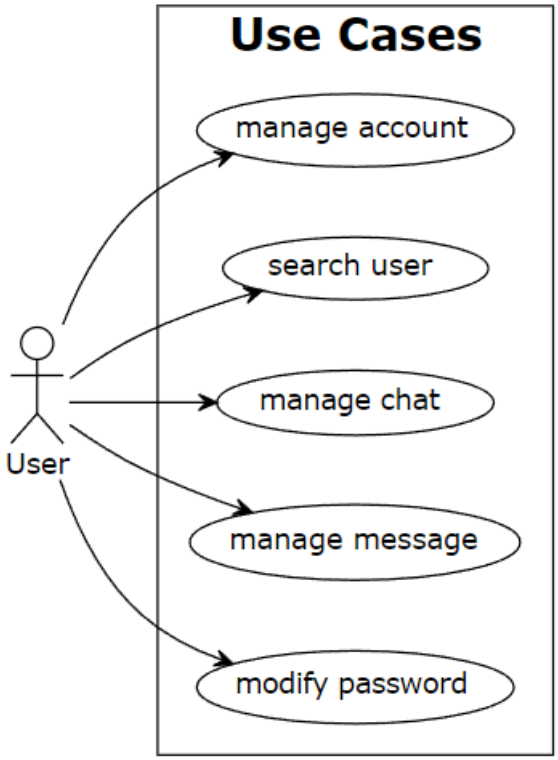

Figura 1: Diagramma dei casi d'uso in UML.

Inoltre la fase di Knowledge Crunching ha portato alla creazione di un *Linguaggio Ubiquo* per evitare imprecisioni e confusioni linguistiche sia nel parlato che nei modelli che nel codice. Il linguaggio ubiquo è uno degli artefatti e in generale degli aspetti più importanti del DDD. Permette una maggior comprensione tra sviluppatori ed esperti del dominio e permette una miglior capacità di validazione del codice, che viene scritto seguendo lo stesso linguaggio.

4 Analisi del Dominio

In questa sezione verranno descritti i principali aspetti inerenti al dominio della messaggistica istantanea e alla sua analisi guidata dai principi e pattern del DDD.

4.1 Introduzione al Domain Driven Design

Di seguito sono illustrati alcuni dei concetti principali legati al DDD necessari per capire i modelli. Tali concetti sono definiti pattern tattici per distinguerli dai pattern strategici. Quest'ultimi descrivono e suggeriscono buone pratiche in ambito di design di massima e architetturale nonché di analisi del dominio, e sono considerati più importanti dei pattern tattici. Inoltre, in generale, il DDD si può considerare un insieme di linee guida più che di dogmi, e la scelta su cosa applicare o meno è sempre lasciata al team di sviluppo.

I diagrammi e la documentazione principale relativa al DDD e al progetto Microchat sono stati creati e scritti in inglese e sono rilasciati automaticamente tramite la CI in un sito disponibile al link `https://ldeluigi.github.io/microchat/index.html`.

Dominio Il dominio (*domain*) è l'insieme di tutte le entità concrete o astratte e di tutte le policy di business intorno al servizio che si vuole implementare. In questo caso il dominio è la *messaggistica istantanea*, o più nello specifico *Microchat* stesso.

Sottodominio Il sottodominio (*subdomain*) è una parte del dominio intorno ad un insieme specifico e ristretto di casi d'uso. In questo caso i sottodomini individuati durante il knowledge crunching sono l'*utente* e la *chat*. I sottodomini si distinguono in *core* (di primaria importanza per il business, su cui concentrarsi), *supportive* (utili per i core ma secondari), *generic* (rivisti in altri problemi e quindi in altre soluzioni, implementabili con soluzioni già sviluppate).

Contesto Il contesto (*bounded context*) rappresenta un confine di integrità di una parte del dominio, al cui interno un'entità o un termine assumono un unico significato, chiaro e preciso. In questo caso sono stati individuati tre contesti, ovvero *chat*, *user* e *account*.

Altri concetti Altri concetti necessari saranno spiegati più avanti (Sezione 5.1.1)

4.2 Modellazione

Tramite l'analisi del dominio che segue il knowledge crunching sono emersi principalmente due sottodomini:

- **Chat** è il sottodominio *core* che riguarda le chat e i messaggi.
- **User** è il sottodominio *supporting* che gestisce i *gli utenti*.

4.2.1 Context Map

La context map in figura 2 mostra i due sottodomini divisi in tre bounded context e le relative dipendenze. Una legenda per l'interpretazione del metalinguaggio è presente in figura 3.

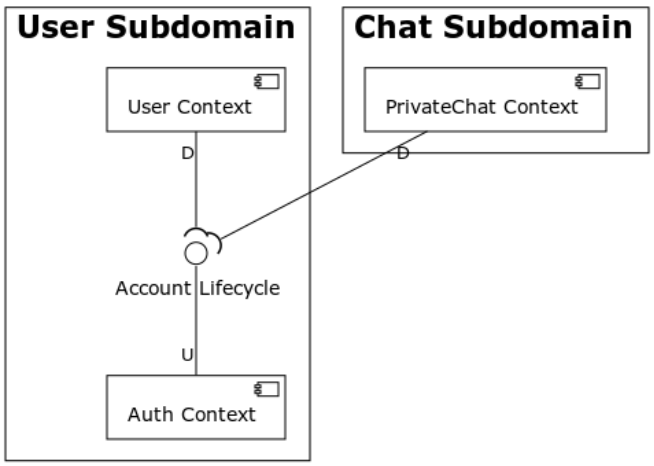

Figura 2: Context Map. U signfica upstream, la fonte di dati. D significa Downstream, colui che riceve i dati. Il cerchio è un'interfaccia di eventi alla quale i contesti dalla parte dell'arco nella relazione si conformano.

Tutte le relazioni presenti tra i contesti sono di tipo **conformist**. Teoricamente, ogni contesto dovrebbe essere assegnato ad un team di sviluppo indipendente. La relazione conformist prevede che il team che usa un'interfaccia si adegui al design proposto da chi la deve implementare. In questo caso il contesto dell'autenticazione (*auth*) emette eventi relativi agli accounti che devono essere ricevuti e interpretati correttamente dal contesto degli utenti e da quello delle chat.

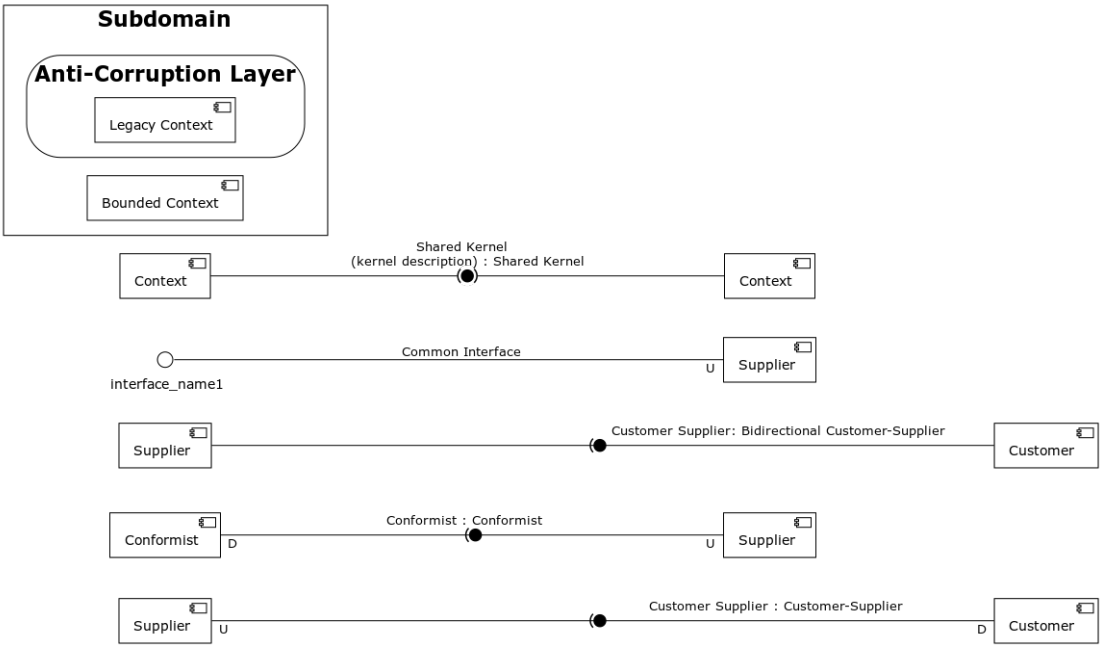

Figura 3: Legenda per il metalinguaggio derivato da UML usato nella context map dal team.

4.3 Sottodominio delle chat: il contesto delle chat private

Il sottodominio delle chat private è *core*, e su questo sono stati concentrati i maggiori sforzi di design e implementazione. Il sottodominio delle chat comprende sia il contesto delle *Chat Private* che quello delle *Chat di Gruppo*. Per questo progetto è stato deciso di ignorare il contesto delle chat di gruppo.

Il contesto delle chat private comprende come casi d'uso:

- **Creazione** e **cancellazione** di una chat con un utente. É ammessa anche la chat con se stessi.
- **Invio di un messaggio** in una chat, quindi ad uno specifico utente.
- **Modifica di un messaggio** qualora l'utente richiedente della modifica coincida con colui che ha effettuato l'invio.
- **Cancellazione di un messaggio** qualora l'utente richiedente della cancellazione coincida con colui che ha effettuato l'invio.

Il diagramma del contesto è visibile in figura 4.

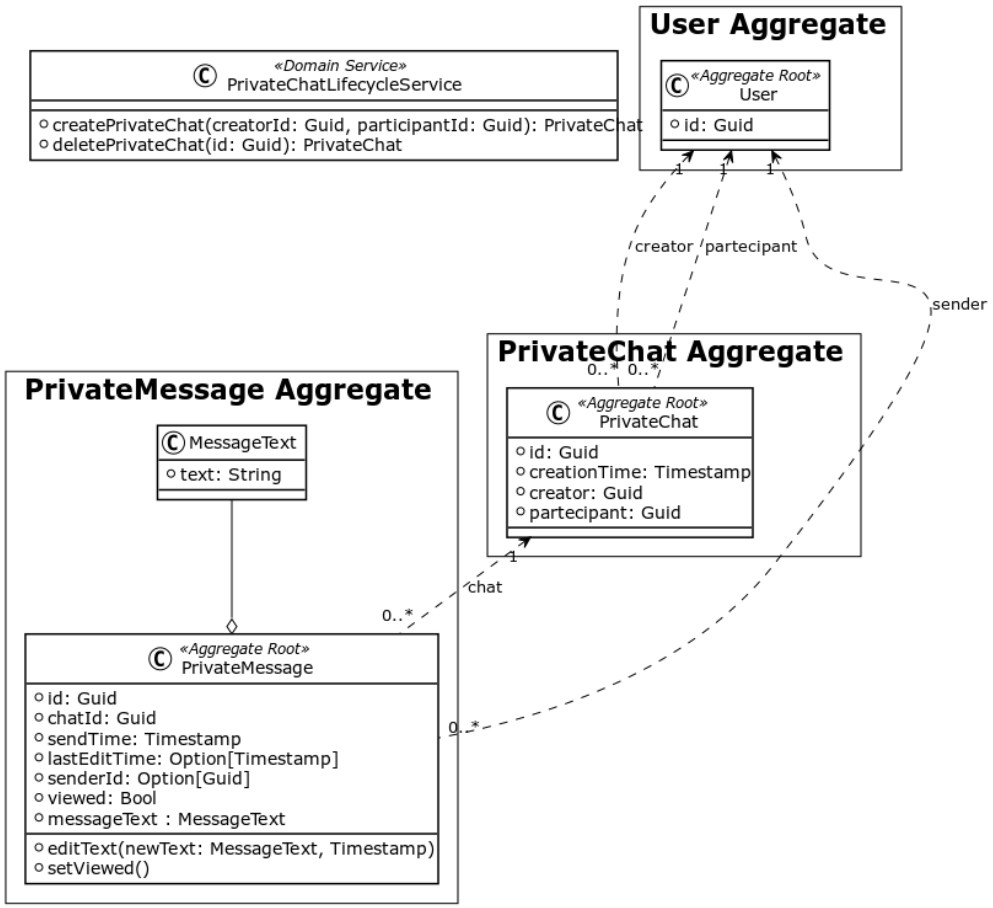

Figura 4: Modello del contesto delle chat private in UML.

4.3.1 Descrizione del modello

Chat Privata è un aggregato [1] con le seguenti proprietà:

- *id*: identificatore della chat;
- *creationTime*: istante di tempo della creazione della chat;
- *partecipant*: identificatore dell'utente invitato a far parte della chat;
- *creator*: identificatore dell'utente che crea la chat.

Una chat privata è una chat che mette concettualmente in comunicazione il creatore e il partecipante.

[1] vedi sezione 5.1.1 per una spiegazione del concetto di aggregato

Messaggio Privato Un messaggio privato ha le seguenti proprietà:

- *id*: identificatore del messaggio;
- *chatId*: identificatore della chat;
- *text*: contenuto del messaggio;
- *senderId*: identificatore del mittente del messaggio;
- *sendTime*: istante di tempo in cui è spedito il messaggio;
- *lastEditTime*: ultimo istante di tempo in cui è stato modificato il messaggio;
- *viewed*: indicatore se, da parte del ricevente del messaggio, questo è stato visualizzato.

Utente L'utente compare in questo contesto solo come fonte di verità sulla propria esistenza. L'entità è composta dall'*id*: l'identificatore dell'utente (lo stesso che ha negli altri contesti).

Chat private, messaggi privati e utenti sono **aggregate root**[2].

4.3.2 Vincoli del modello

I principali vincoli del contesto delle chat private sono i seguenti:

- Le **chat private** hanno solo due partecipanti, uno detto *creator* ovvero colui che crea la chat, e un altro detto *partecipant*.
- Un **utente** può *interagire con una chat* (privata) solamente nel caso in cui sia un partecipante (o creator) della stessa.
- Un **utente** può *spedire messaggi* in una *chat* (privata) solamente nel caso in cui sia un *partecipante* (o creator) della stessa.
- Un **utente** può *modificare o cancellare messaggi* solamente nel caso in cui sia l'originale mandante di tale messaggio.

4.4 Sottodominio degli utenti

Nello User subdomain sono racchiuse le casistiche per le quali è coinvolto univocamente l'utente. Le funzionalità incluse in tale sottodominio sono:

- **creazione** e **cancellazione** dell'utente stesso.
- **autenticazione** e **logout** dell'utente.
- **modifica** delle informazioni relative all'utente.

[2]vedi sezione 5.1.1 per una spiegazione del concetto di aggregate root

Per una migliore analisi il sottodominio dell'utente è stato successivamente scomposto in due contesti:

- **User**: contenente tutte le informazioni dell'utente non necessarie per l'autenticazione.
- **Auth**: contenente tutte le sole informazioni dell'utente necessarie per l'autenticazione.

4.4.1 Descrizioni dei modelli nel contesto degli utenti

Il contesto degli user si concentra sui casi d'uso legati alla ricerca degli utenti.

User Rappresenta l'unica entità del suo aggregato, come mostrato in figura 5, nonché aggregate root e presenta i seguenti campi:

- **id**: identificatore dell'utente.
- **name**: nome dell'utente.
- **surname**: cognome dell'utente.
- **username**: username dell'utente.

Questi campi sono utili per la ricerca di utenti all'interno dell'app.

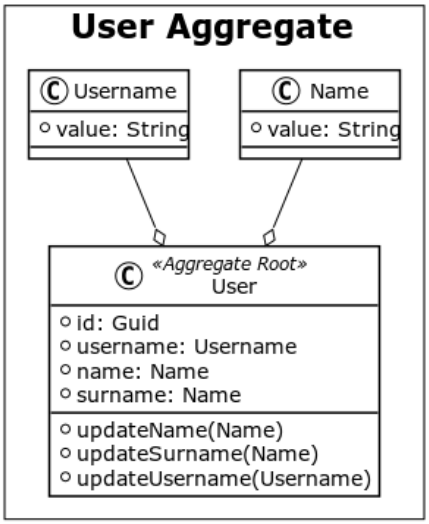

Figura 5: Modello del contesto degli utenti in UML.

4.4.2 Descrizioni dei modelli nel contesto degli account

Il contesto degli account si concentra sui casi d'uso legati all'autenticazione.

Account Rappresenta l'unica entità del suo aggregato, come mostrato in figura 6, nonché aggregate root e presenta i seguenti campi:

- **id**: identificatore dell'account;
- **username**: username dell'account;
- **email**: email dell'account;
- **passwordHash**: composizione di:
  - *password*: password dell'account;
  - *salt*: campo utile a salare la password dell'account;
- **sessions**: set di sessioni, dove ogni sezione comprende:
  - *accessTokenId*: identificatore del token per l'accesso (JWT);
  - *expiration*: durata del token;
  - *refreshToken*: token di rinnovo del JWT.

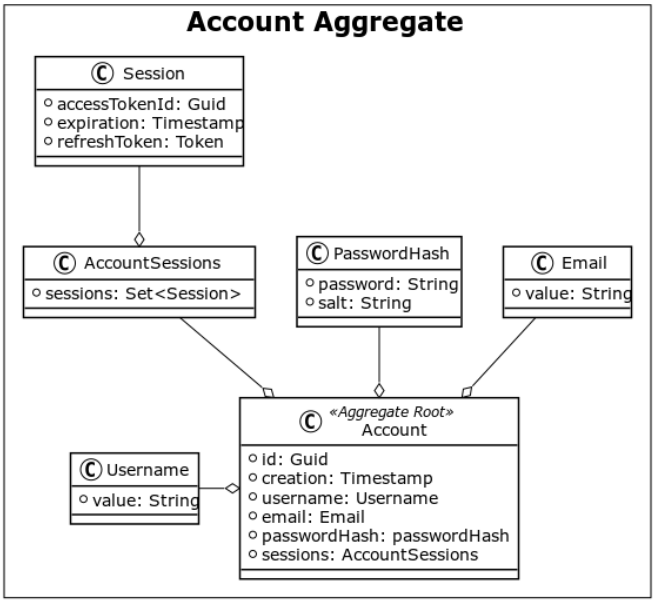

Figura 6: Modello del contesto degli account in UML.

4.4.3 Vincoli del dominio

L'utente, per effettuare le operazioni sul proprio account o sul proprio profilo nel contesto utenti deve essere loggato.

5 Design architetturale

Sempre seguendo i principi del DDD, in particolare quelli strategici, ogni bounded context viene implementato in maniera indipendente e isolata rispetto agli altri. Nel caso di Microchat la relazione tra microservizi e bounded contexts è 1 a 1, quindi dei 3 bounded contexts sono stati realizzati 3 microservizi.

5.1 Architettura di un microservizio

In questa sezione si descrive la struttura generale di ogni microservizio nella sua divisione in *layers* concentrici. Ogni servizio del sistema è diviso in una serie di strati seguendo il design della **Clean Architecture**. Il modello architetturale implementato nella pratica è basato su una libreria, chiamata `EasyDesk.CleanArchitecture`, ispirata al modello originale proposto da Robert C. Martin nel suo libro[2], ma personalizzata.

Di seguito, in figura 7, vengono descritti brevemente gli strati della Clean Architecture implementata e ne si evidenziano anche le dipendenze: ogni strato più esterno dipende da quelli più interni.

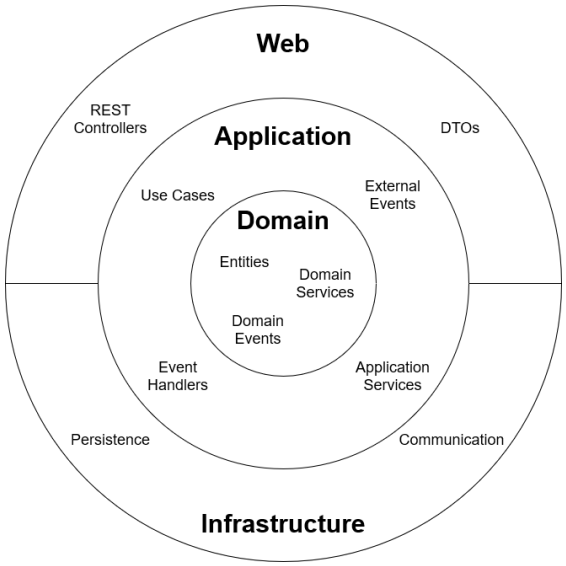

Figura 7: Diagramma a cerchi concentrici (layer) della Clean Architecture impiegata. Il layer Infrastructure dipende dal layer Web solo per l'*Inversion of Control*.

In generale, i microservizi sono *stateless* by design. Ciascuna operazione offerta da un microservizio non mantiene alcuno stato interno dopo il suo termine.

Tale paradigma conferisce vantaggi all'architettura:

- **scalabilità** maggiore;
- **cache** agevolizzata;
- necessitano di **meno spazio di archiviazione**;

- nessuna necessità di associazione del client al server poiché ogni richiesta client fornisce al server tutte le informazioni necessarie.

5.1.1 Il Domain Layer

Il domain layer contiene il mapping 1 a 1 del modello del contesto in codice. Aggregati, servizi, comportamenti e interfacce del dominio sono tutti definiti a questo livello.

Esempi di ciò che si trova in questo livello (pattern *tattici* del DDD):

- Gli **aggregati** sono un insieme autonomo di entità, oggetti e relativi comportamenti individuati e raggruppati sulla base delle *invarianti* del dominio, ovvero vincoli di integrità che collegano più entità insieme. In altre parole, tra aggregati diversi si ha *eventual consistency* mentre all'interno di un aggregato *transactional consistency*.
- Le **aggregate root** è l'unico punto di accesso all'aggregati. Ogni riferimento ad un aggregato si reifica con un riferimento debole (ovvero tramite l'id) all'aggregate root di quell'aggregato. Un aggregato è quindi una sorta di albero il cui accesso deve passare dalla radice.
- Le interfacce dei repository, che permettono al dominio di interagire con la persistenza in maniera agnostica rispetto all'implementazione, che si trova invece a livello infrastrutturale (*infrastructure*).
- I servizi del dominio implementano comportamenti e logiche di business definite dai casi d'uso.

I *domain services* possono notificare attraverso **eventi di dominio** qualsiasi cambiamento rilevante dello stato. Gli eventi, sebbene definiti in questo strato, verranno gestiti in quello applicativo (*application*).

Note sulla semantica dei repository Le operazioni messe a disposizione dai repository di modifica inserimento e cancellazione di un singolo aggregato sono da considerarsi **atomiche**, ovvero eseguite in un'*unica transazione che mantiene la consistenza all'interno della base di dati*. Vale tuttavia il *principio di integrità transazionale solo all'interno di uno stesso aggregato*. Nel caso di Microchat, quindi, è possibile che in un dato istante esistano messaggi senza una chat di appartenenza, oppure chat tra utenti cancellati e quindi assenti. Questo perché messaggio, chat e utente nel contesto delle chat sono aggregati separati. Tra aggregati diversi bisogna accontentarsi di avere *eventual consistency*.

5.1.2 Il layer application

Definisce una collezione di **casi d'uso** esposti dal servizio e le azioni che devono essere eseguite per ognuno di questi tramite una serie di **handlers** che interagiscono con lo strato di dominio e quello *web*. Gli *handlers* possono essere usati anche per gestire la

logica di business in risposta a quando si scatenano degli **eventi di dominio** o alla ricezione di **eventi esterni** e **comandi esterni**.

Gli *eventi esterni* sono emessi da altri contesti e ascoltati dal livello applicativo, qualora questo rappresenti un caso d'uso del contesto in ascolto. Nel caso di Microchat, ad esempio, il contesto delle chat è interessato agli eventi riguardanti la gestione account del contesto degli account. Chi invia un evento esterno è agnostico rispetto a chi lo ascolta, secondo il pattern **publish/subscriber**.

I *comandi esterni* invece sono dei messaggi diretti ad uno specifico destinatario, secondo il pattern **command processor**. Quest'ultimi non sono stati impiegati per Microchat.

Ogni caso d'uso presente a livello applicativo è distinto tra **Query** e **Command**, secondo il pattern strategico **CQRS**.

5.1.3 CQRS

Il pattern *Command/Query Responsibility Segregation* prevede la separazione netta tra i casi d'uso in lettura (query) da quelli in scrittura (command). Siccome le *letture non rischiano in nessun modo la rottura della consistenza* dei dati e la violazione dei vincoli del dominio, possono essere implementate in maniera efficiente direttamente dal layer di infrastruttura, che comunica con la base di dati, senza passare mai il controllo alle entità di dominio. I comandi, invece, per essere eseguiti rispettando i vincoli del dominio, in particolare quelli transazionali, devono essere implementati chiamando la logica del dominio e coinvolgendo i repository per l'accesso alla base di dati, che quindi deve necessariamente avere un supporto alle transazioni *one-read-one-write* di un aggregato.

5.1.4 Il layer infrastructure

Questo strato mantiene l'**implementazione concreta delle interfacce dei repository del dominio e dei servizi di comunicazione del livello applicativo**, nonché l'accesso ai dati e il relativo modello relazionale, l'accesso ai mezzi di comunicazione per i messaggi (comandi o eventi, solitamente tramite un bus di messaggi) e l'implementazione delle query.

5.1.5 Il layer web

Implementa i **punti d'accesso** (endpoint) su cui il client del servizio può effettuare le richieste. I client possono fare richieste usando una ReST API esposta da ogni microservizio, oppure tramite altri protocolli quali WebSocket, se resi disponibili. Questo livello contiene quindi i **DTO** (Data Transfer Object) per rappresentare le interfacce di ogni endpoint. Questo strato è responsabile di gestire l'**autenticazione dei client** per determinare l'identità degli agenti che effettuano le richieste al servizio. Infine il layer web gestisce l'**Inversion of Control** (IoC), ovvero la presenza di un componente che permette di evitare (o volendo astrarre) un design attorno al paradigma procedurale concentrandosi quindi sull'implementazione e la correttezza dei singoli componenti, sapendo che questi verrano "svegliati" o interpellati nei corrispondenti casi d'uso, come

per esempio l'arrivo di una certa richiesta HTTP. Viene anche chiamata **Dependency Injection**.

5.2 La libreria CleanArchitecture e i microservizi in C#

Implementano i layer così definiti usando ASP.NETCore[3] per l'IoC e il layer Web (sia API Socket tramite SignalR[4] che ReST), Rebus[5] per la comunicazione dei messaggi, Entity Framework Core[6] per la comunicazione con la base di dati (solitamente SQL Server di Microsoft), Mediatr[7] per CQRS e l'IoC di handler di qualsiasi tipo e altro ancora.

[3] https://docs.microsoft.com/it-it/aspnet/core/?view=aspnetcore-6.0
[4] https://docs.microsoft.com/it-it/aspnet/core/signalr/introduction?view=aspnetcore-6.0
[5] https://github.com/rebus-org/Rebus
[6] https://docs.microsoft.com/it-it/ef/core/
[7] https://github.com/jbogard/MediatR

6 La comunicazione tra servizi

In questa sezione sono trattati i principali temi relativi alla comunicazione tra i microservizi e alla loro scalabilità.

6.1 Il Teorema CAP

Uno dei fondamentali teoremi dei sistemi distribuiti è il **Teorema CAP**, il quale dichiara che ottenere contemporaneamente *Consistenza*, *Disponibilità* e *Tolleranza ai guasti di rete* è impossibile. Nel caso di Microchat la soluzione è accontentarsi delle proprietà **BASE** (basically available, soft state ed eventual consistency) garantendo il miglior servizio possibile a tutti gli utenti, e al contempo mantenere le proprietà **ACID** (Atomic, Consistent, Isolated e Durable) per tutti i *casi d'uso modellabili con singoli comandi che coinvolgono singoli aggregati.*

In relazione a Microchat e in generale ai microservizi simili questo teorema ha diverse conseguenze a seconda dei tipi di fallimento che possono accadere. Come in ogni buon sistema distribuito, anche per questo si può parlare di *design for failure.* Grazie alla possibilità di avere molteplici repliche per ciascun microservizio, come approfondito nella prossima sezione, è possibile nascondere il fallimento di una o più di queste ridirigendo il traffico su un'altra, grazie al design *stateless.* I fallimenti critici per un microservizio quindi non sono quelli del suo software o della rete tra servizio e client ma piuttosto quelli della rete che lo collega al database, senza il quale il sistema non può chiaramente funzionare. In questo caso si rinuncerebbe all'availability. Un fallimento nella rete tra un servizio (tutte le repliche) e il message bus o il backplane Redis si traduce invece in un possibile stato di temporanea inconsistenza dei dati, che data la natura e il design *for failure* di entrambi questi componenti si risolverà *eventually* una volta ripristinato il collegamento, rinunciando quindi solo temporaneamente alla consistenza dei dati.

Tramite i pattern **outbox** si garantisce che ciascun evento sia correttamente consegnato al bus e gestito completamente dal servizio che lo deve ricevere. Questo pattern è implementato facendo uso delle capacità transazionali del database in cui vengono salvati tutti gli eventi in uscita fino a confermata consegna da parte del message bus. É tuttavia possibile che un fallimento accada durante la fase di conferma di avvenuta gestione, il che significa che alcuni eventi possono essere ricevuti più volte dallo stesso servizio. Col pattern **inbox** implementato tramite una finestra temporale di eventi è possibile scartare gli eventi ripetuti.

In generale, grazie a tutti gli accorgimenti messi in atto, è possibile definire il sistema **partition tolerant**, tranne appunto per catastrofiche situazioni legate alla connessione coi dati.

6.2 Scalabilità

Ciascun microservizio, in quanto stateless, può essere replicato a seconda del carico di richieste provenienti dagli utenti, come visibile in figura 8. Questa capacità di scalare orizzontalmente introduce numerosi problemi all'interno dell'architettura. Per quanto

riguarda la comunicazione tramite scambio di messaggi è valido assumere che il Message Bus sia in grado di sostenere un carico estremamente grande di lavoro in quanto progettato per farlo. Il database, sul quale insistono tutte le repliche, può essere suddiviso (tramite ad esempio lo *sharding*) in quanto i comandi leggono e scrivono in modo transazionale un singolo aggregato per volta, potendoli quindi separare a livello estensionale. Le query invece si possono agilmente ottimizzare con caching e calcolo parallelo. Per quanto riguarda la comunicazione tramite *long-lived TCP/IP connection* come ad esempio le WebSocket sono necessari piccoli accorgimenti da applicare al load balancer che gestisce le repliche (in particolare sono necessarie le cosiddette *sticky connections*) e un sistema di sincronizzazione tra websocket detto *backplane*, che fa sì che utenti connessi a repliche diverse dello stesso servizio possano comunicare tramite socket in pseudo real-time come se fossero connesse alla stessa replica.

Nel caso di Microchat, sebbene le librerie supportino più soluzioni per i componenti, sono stati usati **RabbitMq** come message bus, **Microsoft SQL Server** come database e **Redis** come backplane per le socket del servizio chat. Un diagramma dei componenti che illustra i flussi di dati ed eventi di Microchat è disponibile in figura 9.

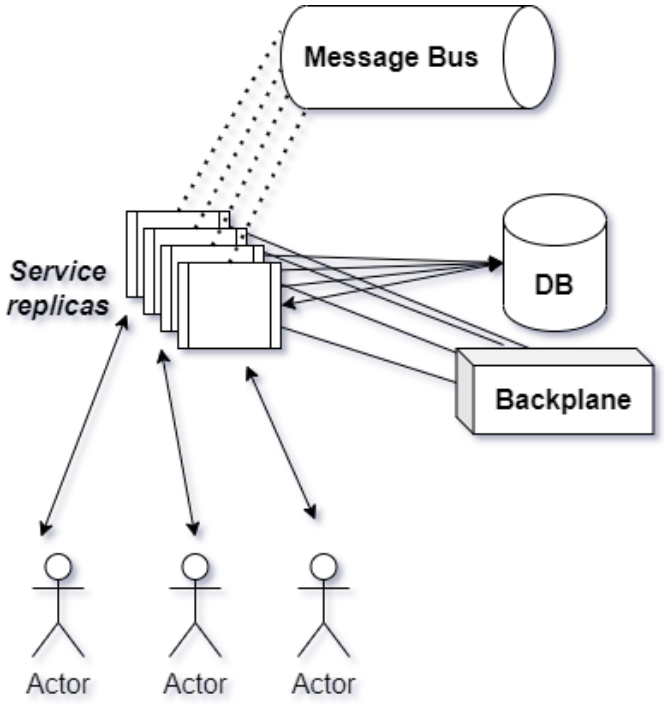

Figura 8: Esempio di come più repliche di un microservizio comunichino con altri servizi attraverso bus, col database, tra di loro tramite un backplane e con i client.

Durante i test è stato usato **Docker** come piattaforma di containerizzazione di tutti i componenti e come DNS/orchestrator per la loro comunicazione.

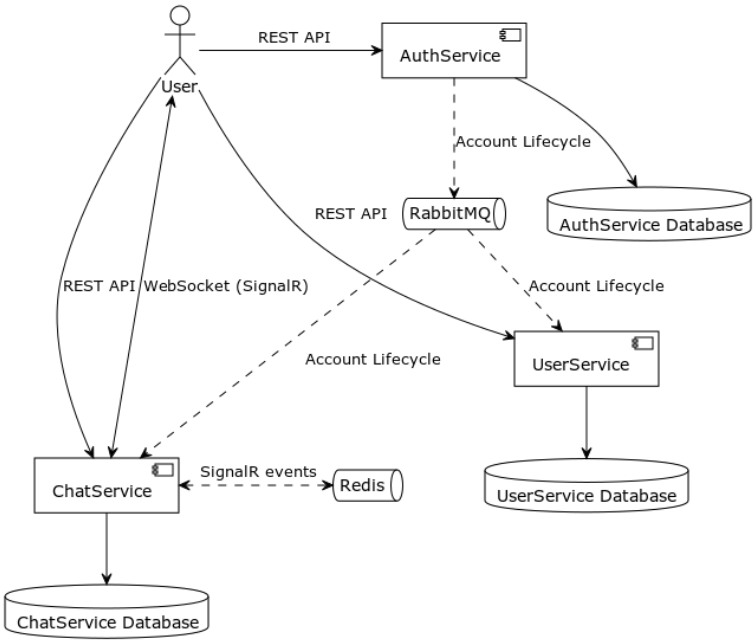

Figura 9: Diagramma dei componenti che mostra anche i flussi di richieste (linee continue) ed eventi (linee tratteggiate).

6.3 Comunicazione ad eventi

All'interno del progetto è stato fatto uso di eventi per propagare ai microservizi interessati la notifica di un cambiamento avvenuto in un contesto: questi eventi sono creati a partire da modifiche del dominio che vengono rese atomiche tramite delle transazioni supportate dal database. Durante il processamento di un comando viene caricato un singolo aggregato dal database al servizio. Tutte le scritture vengono effettuate su quello specifico aggregato e schedulate per quando viene chiusa la transazione. A quel punto il database potrà accettare la transazione se i vincoli non sono violati oppure rifiutarla, cosa che verrebbe comunicata all'utente. Tra le scritture accorpate sono incluse anche la creazione di eventi da inviare al bus, salvati sul database fino a confermata ricezione da parte del bus.

Per la generazione degli eventi ogni microservizio esegue in sequenza le seguenti operazioni, in risposta ad un comando:

1. si **inizia la transazione** che coinvolge un'aggregato;
2. nel caso di una modifica, l'aggregato viene caricato in memoria dal database;
3. vengono effettuati i **cambiamenti del dominio** e viene **creato l'evento (o gli eventi) da propagare**;
4. l'aggregato modificato e gli eventi in uscita vengono salvati sul database come parte della transazione;

5. si **salva la transazione** (commit) al fine di ottenere l'atomicità e il rispetto dei vincoli (che sono sempre interni ad un singolo aggregato);
6. viene chiusa la richiesta e il controllo viene restituito al chiamante;
7. in maniera *out of process*, un task in background comincia una nuova transazione col database per svuotare l'outbox appena riempita;
8. vengono letti gli eventi in uscita e caricati in memoria per l'invio al broker;
9. vengono **pubblicati gli eventi** creati precedentemente, e, se viene confermata la corretta ricezione da parte del service bus, questi vengono rimossi dal database (dall'outbox). In caso contrario un processo in background ogni tanto si sveglia e riprova ad inviare nuovamente il messaggio, finché non ci riesce.
10. viene effettuato il commit anche di questa transazione.

Se qualcosa va storto il processo è interrotto ma si può osservare che valgono le seguenti invarianti:

- I vincoli intra-aggregato non sono violati;
- Gli eventi non sono mai persi

Nei casi di errore vengono effettuati nuovi tentativi periodici fino al corretto svolgimento di tutti gli step. Può capitare quindi che alcuni eventi vengano inviati/ricevuti più volte. Per correggere questo problema è stato implementato il pattern inbox. Se un fallimento avviene durante la prima transazione il client osserverà un errore nella risposta HTTP.

Una descrizione più formale del processo è disponibile in figura 10.

6.4 Comunicazione tramite WebSocket

WebSocket è una tecnologia web che fornisce canali di comunicazione full-duplex e real-time attraverso una singola connessione TCP, detta *long-lived*, tra client e servizio. Per stabilire una connessione Websocket, il client invia una richiesta di handshake HTTP "Upgrade" ed il server invia la risposta affermativa o negativa. In caso di risposta affermativa si crea il canale che verrà utilizzato in maniera bidirezionale. Da questo momento il server potrà contattare direttamente il client senza che quest'ultimo inizi la comunicazione. Questo nuovo modello è detto **push model**, per distinguerlo del più comunemente usato **pull model** di HTTP. La connessione rimarrà aperta finché non si chiuderà il browser o non si effettuerà il refresh della pagina, o finché il server non decide di terminarla.

Il diagramma di figura 11 mostra come viene utilizzata la WebSocket per il caso d'uso dell'invio di un messaggio e della conferma di visualizzazione.

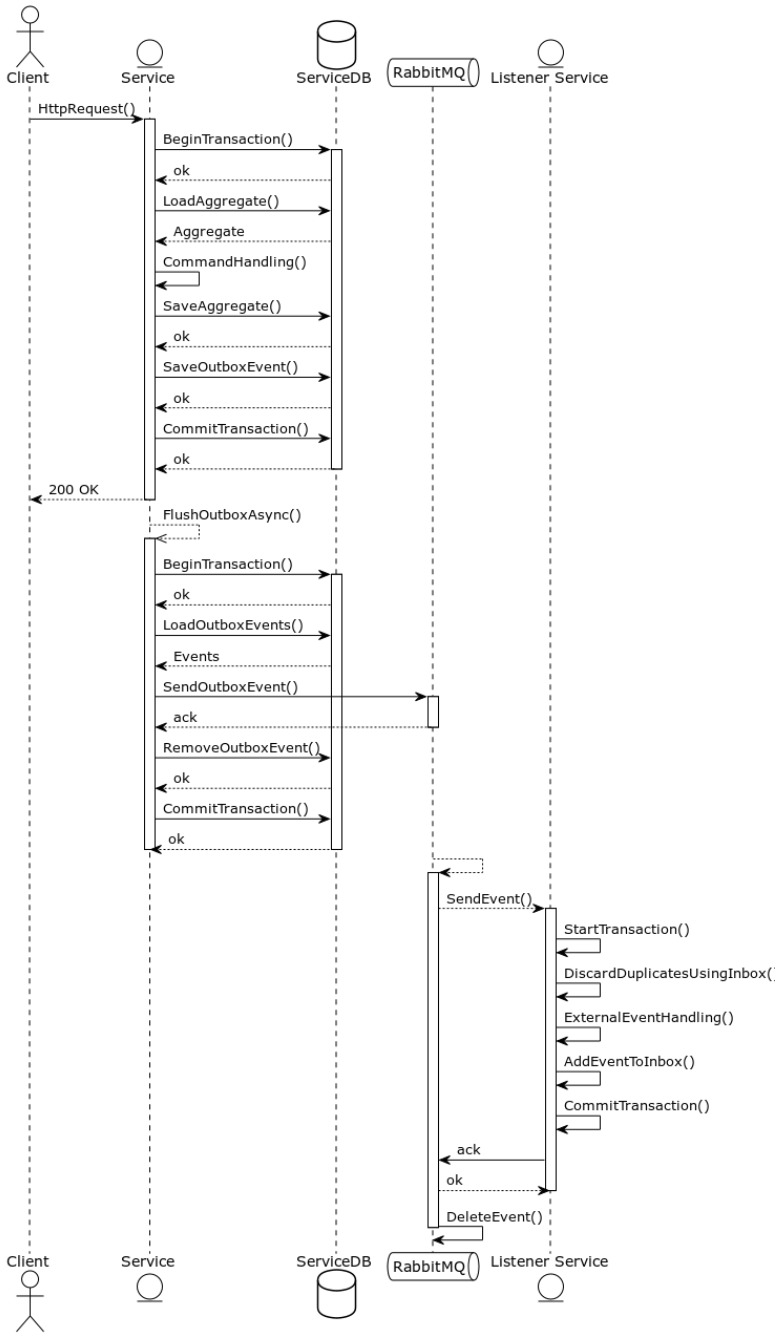

Figura 10: Diagramma UML di sequenza relativo ad un comando e alla generazione di eventi da emettere.

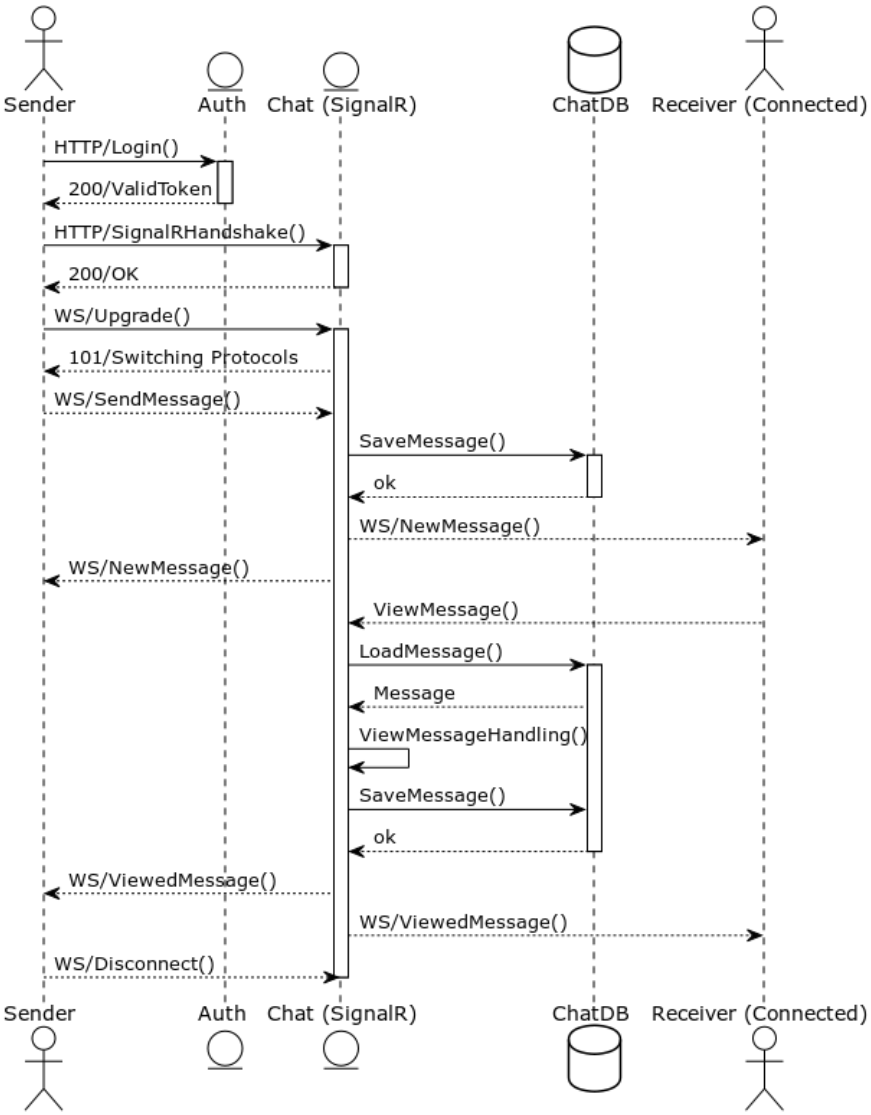

Figura 11: Diagramma UML di sequenza relativo all'invio e la relativa visualizzazione di un messaggio.

7 Tecnologie impiegate

In questa sezione si parla delle varie tecnologie impiegate nello sviluppo di Microchat.

7.1 SignalR

SignalR è una libreria software gratuita e open source per Microsoft ASP.NET che consente al codice server di **inviare notifiche asincrone alle applicazioni Web sul lato client**. Il paradigma adottato da SignalR è detto **push model**. L'idea di SignalR e

di altre librerie simili quale ad esempio Socket.io è quella di iniziare un dialogo tra client e server per contrattare automaticamente quale dovrebbe essere il miglior protocollo di comunicazione bidirezionale da adottare, per poi aprire un canale di quel tipo il tutto in modo trasparente rispetto al codice che usa la libreria sia lato server che lato client.

7.1.1 Funzionamento

SignalR permette la comunicazione attraverso tre protocolli:

- **Websockets**: è la modalità principale in quanto garantisce un *uso efficiente della memoria del server*, una *latenza bassa* e alcune funzionalità extra rispetto agli altri canali.
- **Server Side Events**: in questa modalità il client manda una richiesta dove indica la possibilità di accettare stream di dati. In questo modo il server ogni volta che avrà un dato disponibile potrà creare una connessione e inviare dati. Per permettere al client di comprendere che l'invio dati è finito viene inserito un segnalatore in fondo allo stream.
- **Long polling**: usato come ultima scelta, consiste nel client che effettua una richiesta periodica al server, ripetuta fino al termine della comunicazione. Per ogni richiesta di nuovi eventi il client apre una nuova connessione. Seppur funzionante, questa strategia è un forte spreco di risorse computazionali e di rete.

Se non specificata la scelta tra i tre canali di comunicazione questa viene effettuata da SignalR valutando le risorse e le tecnologie disponibili dal client e dal server.

SignalR offre inoltre delle astrazioni di concetti legati alla messaggistica istantanea: connessioni, utenti e gruppi. Normalmente, per qualsiasi modalità di comunicazione, i client potranno comunicare solamente con altri client connessi allo stesso server, in altre parole alla stessa replica dello stesso server, cosa che ne impedisce la scalabilità in quanto in un sistema come Microchat ciascun utente deve poter comunicare in real time con qualsiasi altro.

7.1.2 Redis

Redis è un **archivio chiave-valore in memoria** che supporta anche un sistema di messaggistica con un modello di publish/subscribe. Le funzionalità publish/subcribe concesse dal backplane Redis permettono ai server di **inoltrare un messaggio ad altri server**. E' possibile integrare Redis con SignalR per sincronizzare le repliche di un servizio tra loro, in modo trasparente rispetto a chi usa la libreria. Quando un client si connette ad un server viene propagata implicitamente tale informazione anche al "backplane" Redis. In questo modo, se un server desidera inoltrare un messaggio ad un client non connesso alla propria istanza lo comunica a Redis che se possibile notificherà l'interessato del messaggio e lo porterà a consegna come si vede in figura 12.

Questo design fa sì che SignalR usi, di Redis, solamente le funzionalità publish/subscribe e non quelle di archivio chiavi-valore.

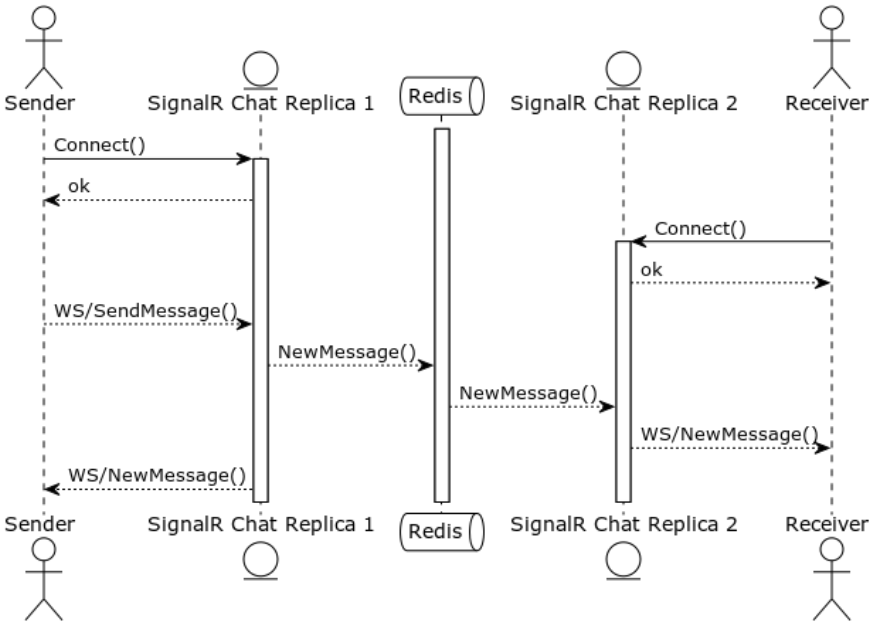

Figura 12: Diagramma UML di sequenza relativo all'inoltro dei messaggi tramite Redis.

7.1.3 Vantaggi

Ecco i principali vantaggi derivanti dall'utilizzo di SignalR:

- perfetta integrazione con ASP.NETCore,
- supporta le WebSocket,
- è free ed open source,
- permette la comunicazione real time, fondamentale in un sistema ad eventi, distribuito come quello dei microservizi.

7.1.4 Svantaggi

SignalR necessita di "*sticky connections*" attraverso il load balancer/reverse proxy, ovvero connessioni persistenti tra un client e una replica del server. Tali connessioni rimangono aperte anche quando il client è inattivo, in modo che messaggi diversi che passano attraverso la stessa connessione con SignalR sono consegnati sempre agli stessi agenti client/server. Tuttavia, il numero di connessioni TCP che un server può effettuare sono limitate. Se un server dovesse terminare il numero di connessioni disponibili, la libreria potrebbe avere un comportamento inatteso.

7.2 Docker

Docker è un progetto open-source di containerizzazione di applicazioni, che da la possibilità di astrarre rispetto all'ambiente di esecuzione necessario per la sua esecuzione. In altre parole con Docker è possibile creare immagini (snapshot) di build pulite del software e il deploy anche su larga scala di queste. Docker inoltre mette a disposizione un servizio di DNS interno tra container per semplificare l'identificazione reciproca dei vari host.

7.3 RabbitMQ

La comunicazione tra microservizi si basa sullo scambio di messaggi asincroni per notificare gli eventi di dominio che sono avvenuti all'interno di un contesto. RabbitMq è la scelta tecnologica implementativa per il Message Bus, la piattaforma di distribuzione di eventi (publish/subscribe pattern) e comandi (command processor pattern). RabbitMQ è un **broker di messaggi** open source, free e pensato per essere leggero e scalabile. Può essere utilizzato per sistemi distribuiti anche a larga scala, e soddisfa i requisiti del progetto di per l'affidabilità e il pattern inbox (non lascia che i messaggi non consegnati vengano dispersi e riprova fino a riuscirci).

8 Implementazione

Questa sezione spiega come sono state organizzate e implementate i vari componenti del sistema.

8.1 Backend

I microservizi lato backend sono stati sviluppati internamente in C# tramite l'utilizzo della libreria **EasyDesk.CleanArchitecture** che segue la struttura della Clean Architecture di Martin.

Per l'interfacciamento a RabbitMQ è stata utilitizzata la libreria Rebus, in particolare **Rebus.RabbitMq**, una libreria nata per astrarre dall'effettiva infrastruttura del broker di messaggi: questa scelta è stata fatta per poter, in un secondo momento, scegliere di cambiare broker (ad esempio per il deploy su Azure con Azure Service Bus).

8.1.1 Docker compose

Docker compose è un tool di Docker per la definizione e l'esecuzione di applicazioni a container multipli. Docker compose è configurabile tramite l'uso di un file YAML[8]; così usando un solo comando è possibile creare e far partire tutti i servizi dell'applicazione.

Docker compose è stato utile nello sviluppo del backend per effettuare le prove di integration test tra i vari servizi che, usando delle configurazioni condivise nel file YAML, erano in grado di comunicare ed esporre le sole porte alle applicazioni frontend.

8.1.2 AuthService

AuthService gestisce gli account e i login tra tutti i microservizi tramite i JSON Web Tokens[9], che sono una valida soluzione scalabile al problema dell'autenticazione. All'interno di un JWT sono codificati i dati fondamentali per identificare un utente (il suo id ad esempio) e sono firmati con un segreto condiviso dai servizi (crittografia simmetrica in questo caso).

Fornisce le funzioni per:

- **registrare ed eliminare** un utente;
- effettuare il **login** e **logout**;
- effettuare il **refresh del JWT**;
- rendere **disponibile all'applicazione l'id e il JWT dell'utente** se quest'ultimo ha effettuato il login;
- effettuare l'**update delle informazioni utente** (email, password e username).

Un diagramma di sequenza della registrazione di un nuovo utente è rappresentata in figura 13.

[8] `https://yaml.org/`
[9] `https://jwt.io/`

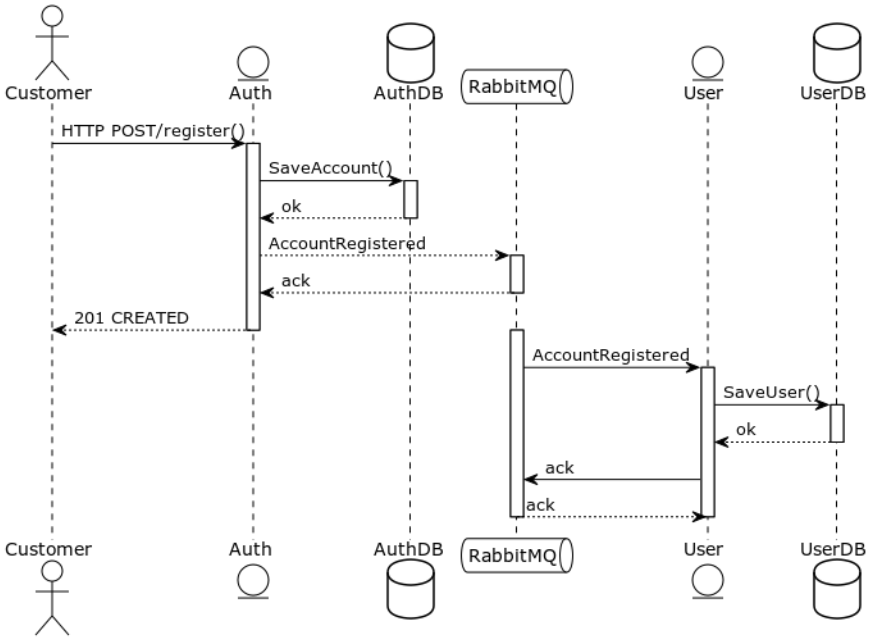

Figura 13: Diagramma di sequenza che rappresenta le interazioni necessarie alla creazione di un nuovo utente.

8.1.3 UserService

UserService si occupa di mantenere i dati di contorno dei profili degli utenti, in questo caso nome e cognome. Permette inoltre una ricerca basata su parole chiave di un utente per poter iniziare una chat con un nuovo contatto.

8.1.4 ChatService

ChatService oltre all'interfaccia restful esposta anche da tutti gli altri servizi deve essere in grado di gestire la comunicazione bidirezionale con il frontend per la messaggistica istantanea. Per farlo utilizza un endpoint collegato ad un **Hub** di SignalR. I gruppi di SignalR non sono usati. L'*invio di un messaggio,* la *modifica* e la *cancellazione* sono a tutti gli effetti dei comandi per il livello applicativo, e come tali vengono portati a termine coinvolgendo un singolo aggregato, in scrittura, in modo transazionale. Dopo l'avvenuta corretta esecuzione del comando viene inviato il messaggio con SignalR. SignalR, a differenza di RabbitMq, non da alcuna garanzia sul corretto invio o ricezione dei messaggi. Quindi, in rarissimi casi (crash di un servizio o partizionamento di rete tra la fine di un comando e la comunicazione con SignalR e il client) è possibile che un client osservi una chat diversa da quella reale (e corretta) che si trova sul database.

Fornisce le funzioni per:

- **ricevere gli eventi relativi** alle chat (creazione, rimozione);

- **ricevere gli eventi relativi** a un messaggio (ricezione, aggiornamento, rimozione) di ogni chat;
- **inviare un messaggio**;
- **modificare o eliminare un messaggio**.

8.2 Frontend

Il frontend, come descritto in precedenza, è stato sviluppato con **Angular**. Si è utilizzato un approccio **Object-Oriented** unito ad una forte separazione delle responsabilità cercando di attribuire ad ogni *Component* o *Service* un compito specifico.

8.2.1 Dettagli sul Frontend

JWTInterceptor e ErrorInterceptor Sono intercettatori di chiamate tra app e backend utilizzati rispettivamente per aggiungere a tutte le chiamate il JWT nel caso in cui l'utente sia connesso, e per intercettare eventuali errori nelle chiamate fatte e, se il problema è risolvibile, sistemare l'errore riscontrato e provare a effettuare nuovamente la chiamata al server.

AuthGuard Rappresenta l'unica guardia presente nel sistema. Essa blocca l'accesso all'area privata utente per tutti gli utenti non registrati.

8.3 DevOps Engineering

I principi e le linee guida DevOps sono state applicate all'intero processo di sviluppo sin dalle prime fasi. Sono state un fattore abilitante per l'efficacia del processo di sviluppo dall'analisi del dominio fino alla fase di integrazione. In questa sezione verranno discussi gli strumenti e le metodologie messe in pratica.

8.3.1 Strategie di DVCS

I sorgenti sono stati gestiti con Git e hostati su GitHub, per essere infine ricaricati tramite un workflow su un repository GitLab.

8.3.2 Software Build Lifecycle

Ogni passo della build del software è stata automatizzata tramite Git e GitHub Actions. GitHub Actions offre dei container (chiamati runners) che eseguono vari tipi di script divisi in flow, task in modo imperativo.

Ogni tipo di sottoprogetto (microservizio, frontend, documentazione) viene gestito da un workflow di CI/CD costruito appositamente per l'automazione; inoltre ogni progetto implementato in C# condivide le stesse politiche di stile.

8.3.3 Controllo qualità del software

Il controllo di qualità del software è implementato da regole stilistiche imposte e warning in compilazione trattati come errori.

8.3.4 Licenza

Ogni artefatto software è rilasciato sotto licenza **MIT**, per renderlo *libero* e *Open Source*.

8.4 Istruzioni per il Deploy

Il deploy del progetto è diviso in due parti:

- il backend, essendo sviluppato tramite container di Docker è facilmente deployabile in locale entrando nella cartella `./services` e, dopo aver avviato docker, si lancia il comando `docker compose up -d --build`.
- Il frontend può essere lanciato entrando nella cartella `./frontend/microchat-frontend` e, dopo aver installato i vari pacchetti npm (`npm install`) tramite il comando `ng serve`, il frontend è disponibile all'indirizzo `http://localhost:4200`.

Per mancanza di tempo non è stato fatto un deploy sul cloud.

8.5 Istruzioni per l'uso

Anche per quanto riguarda l'utilizzo, il progetto può essere diviso in backend e frontend:

- una volta deployato tramite docker compose la parte REST del backend può essere testata anche tramite `swagger` agli indirizzi:
  - `http://localhost:8080/swagger/index.html` per AuthService.
  - `http://localhost:8082/swagger/index.html` per UserService.
  - `http://localhost:8084/swagger/index.html` per ChatService.
- Il frontend è navigabile all'indirizzo `http://localhost:4200`.

8.5.1 Descrizione per user stories

Registrazione Account Per registrare l'account è necessario andare alla pagina */register* (tramite l'apposito bottone) nella pagina di login e riempire la form; una volta completato cliccare sul bottone register.

Accesso Acccount Per effettuare il login occorre inserire username e password nella pagina */login* (pagina iniziale quando non si è autenticati) e cliccare il bottone login.

Cancellazione di un Account Per cancellare l'account, una volta registrato ed effettuato il login, è necessario cliccare sul proprio profilo (immagine in alto a sinistra) e cliccare sul bottone delete user.

Modifica delle credenziali di accesso e dati utente Per modificare le credenziali e i dati utente, una volta registrato ed effettuato il login, è necessario cliccare sul proprio profilo (immagine in alto a sinistra) e cliccare sull'apposita icona di modifica del campo; una volta completato cliccare sul bottone update.

Creazione di una Chat Per creare una chat con un utente, una volta registrato ed effettuato il login, è necessario trovare l'utente con cui si vuole chattare (scrivendo o username o nome o cognome nell'apposita barra di ricerca) e cliccare sul contatto dell'utente.

Spedizione di un messaggio in una chat Per spedire un messaggio, una volta creata una chat, è necessario scrivere un messaggio e cliccare *Enter* o l'apposita icona di invio.

Cancellazione di un messaggio Per cancellare un messaggio è necessario cliccare sul messaggio tenendo premuti *Ctrl+alt*.

Modifica di un messaggio Per modificare un messaggio è necessario cliccare sul messaggio tenendo premuto *Ctrl* scrivere un nuovo messaggio e cliccare *Enter* o l'apposita icona di modifica.

Cancellazione di una chat Per cancellare una chat è necessario cliccare sull'icona delete (in alto a destra).

9 Conclusioni

Il progetto è risultato essere un'esperienza completa ed estremamente formativa per quanto concerne le problematiche, le sfide e i vantaggi di un buon design nell'ambito dei sistemi distribuiti informativi scalabili. Il backend è stato realizzato seguendo i principi di Domain Driven Design che si integrano molto bene con l'architettura a microservizi. I principi del DDD e di DevOps si sono rivelati anche in questo progetto una guida fondamentale per la buona riuscita della soluzione e per il mantenimento di un alto livello di qualità del codice e del processo di sviluppo soprattutto. Il team è molto soddisfatto del lavoro svolto sul progetto, nonostante non abbia avuto il tempo materiale di implementare anche i requisiti opzionali.

9.1 Lavori futuri

Il progetto sarebbe ulteriormente espandibile tramite l'aggiunta di varie funzionalità:

- gestire le chat di gruppo;
- visualizzare lo stato degli altri utenti (online o offline, sta scrivendo...);
- la possibilità di rispondere a uno specifico messaggio;
- la possibilità di inoltrare uno specifico messaggio a qualcun altro;
- e molte altre idee già viste in app come WhatsApp o Telegram, che sono stati presi come spunto di idee inizialmente.

Riferimenti bibliografici

[1] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[2] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Boston, MA: Prentice Hall, 2017. ISBN: 978-0-13-449416-6. URL: https://www.safaribooksonline.com/library/view/clean-architecture-a/9780134494272/.