The Social Network

Giacomo Romagnoli
giacomo.romagnoli4@studio.unibo.it

Giugno 2025

I social network sono applicazioni che permettono agli utenti di interconnettersi su larga scala, condividendo pensieri ed emozioni con il resto della comunity. Con il tempo questa tipologia di applicazioni si è diffusa, affermandosi come strumento di comunicazione e divulgazione a livello globale. Il mercato moderno comprende diversi social network pronti all'uso, distinti l'uno dall'altro principalmente per i contenuti che la piattaforma permette di pubblicare e condividere. Altri fattori discriminanti sono il metodo di condivisione dei contenuti stessi, i quali possono essere pubblici, ristretti ad un gruppo di amici selezionati dall'utenza o spesso proposti da sofisticati algoritmi. The Social Network vuole ricreare un facsimile di questi sistemi, proponendo una soluzione tecnica concentrata sul garantire i quality attribute ritenuti fondamentali; a questo scopo ci si è concentrati sul deploy del sistema finale, che costituisce il contributo di maggior spessore per questo progetto.

1 Goal/Requirements

In questa sezione vengono esposti i requisiti funzionali che definiscono gli obiettivi del progetto. Tali requisiti sono formulati tramite user stories che riportano le esigenze e le aspettative degli utenti.

- **Registrazione**
  Come nuovo utente,
  voglio registrarmi nel sistema,
  così da creare un account e iniziare ad usare l'applicazione.

- **Autenticazione**
  Come utente registrato,
  voglio accedere al sistema,
  così da autenticarmi e usare le funzionalità dell'applicazione.

- **Invio richieste di amicizia**
  Come utente,

voglio inviare richieste di amicizia ad altri utenti,
così da aggiungerli alla mia lista di amici.

- **Accettazione/Riufiuto delle richieste di amizcizia**
  Come utente a cui è stata inviata una richiesta di amicizia,
  voglio accettare o rifiutare la richiesta,
  così da poter decidere di chi essere amico.

- **Notifica di richiesta di amicizia**
  Come utente,
  voglio ricevere una notifica quando ricevo una richiesta di amicizia,
  così da accorgermi subito di averla ricevuta.

- **Chat** Come utente con degli amici,
  voglio inviare e ricevere messaggi privati dai miei amici,
  così da poter comunicare con loro tramite l'applicazione.

- **Notifiche di chat**
  Come utente che riceve un messaggio da un amico,
  voglio ricevere una notifica alla ricezione,
  così da accorgermi subito di aver ricevuto un messaggio da un amico.

- **Pubblicazione di contenuti**
  Come utente con degli amici,
  voglio pubblicare contenuti,
  così da condividere con tutti i miei amici i miei pensieri.

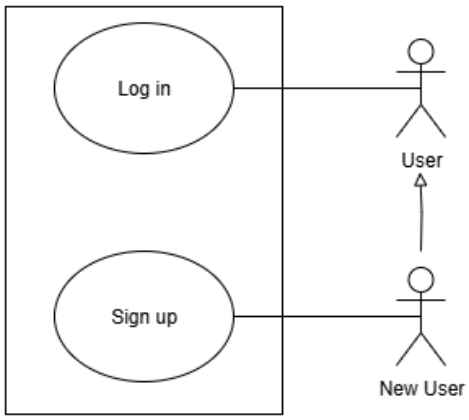
- **Visualizzazione di contenuti**
  Come utente con degli amici,
  voglio visualizzare i contenuti pubblicati dai miei amici,
  così da rimanere aggiornato su cosa condividono.

- **Ricerca con parola chiave**
  Come utente con degli amici,
  voglio cercare fra le pubblicazioni dei miei amici inserendo una parola chiave,
  così da filtrare i contenuti che la contengono.

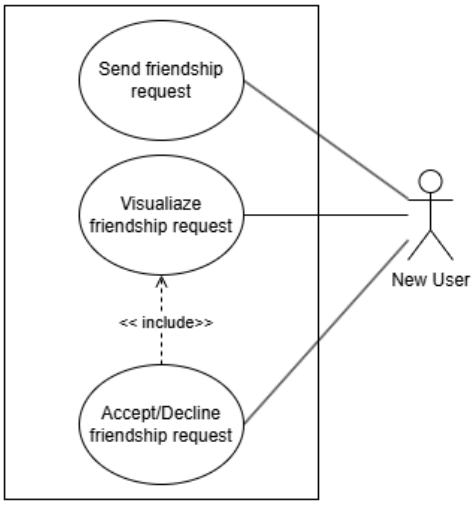
- **Supervisione utenti**
  Come amministratore,
  voglio poter bloccare e sbloccare altri utenti,
  così da mantenere la piattaforma un luogo sicuro per tutti.

1.1 Scenarios

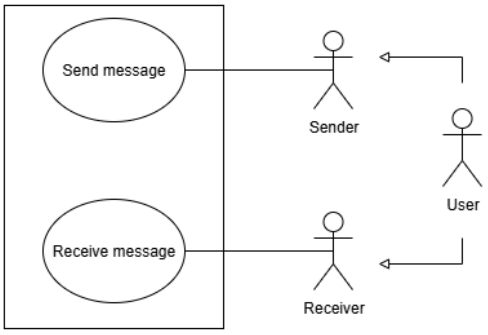
Quando un utente vuole utilizzare l'applicazione, la prima azione richiesta è quella di effettuare il login nel sistema, o registrarsi in caso non si possegga un account.



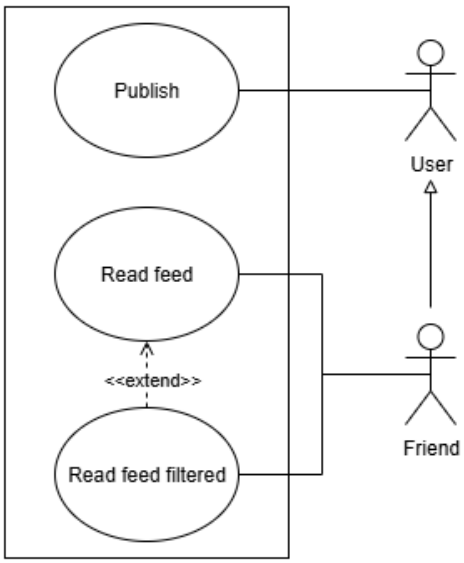
Eseguito l'accesso l'utente è libero di effettuare richieste al sistema usufruendo delle sue funzionalità. Tuttavia la maggior parte di queste dipende dalla rete di amici, per questo un nuovo utente deve prima costruirne una richiedendo o accettando delle richieste di amicizia.



Una volta popolata la lista amici l'utente può iniziare a chattare con i propri amici tramite uno scambio di messaggi testuali privati.



Oltre agli scambi privati l'utente può pubblicare dei contenuti testuali, tali contenuti sono visualizzabili nel feed, risorsa che comprende tutte le pubblicazioni relative alla rete di amicizia di un utente. L'utente può decidere di operare una ricerca all'interno del proprio feed, visualizzando solo quei contenuti che contengono una keyword da lui definita.



2 Requirements Analysis

Un'attenta analisi condotta dagli sviluppatori sulla base dei requisiti funzionali e dell'esperienza maturata nell'utizzo di sistemi simili, ha condotto alla definizione dei quality attribute propri del sistema da realizzare. Di seguito sono elencati i requisiti non funzionali:

- **Affidabilità**
  Il sistema deve rimanere stabile nel tempo evitando crash inaspettati.

- **Consistenza**
  La consistenza del sistema è eventuale. Cioè possono esistere archi temporali, seppur ridotti, in cui una modifica allo stato globale non si è ancora propagata, ma è garantita nel tempo.

- **Disponibilità**
  Il sistema deve rimanere disponibile e raggiungibile dagli utenti per lunghi periodi di tempo, indipendentemente dal carico computazionale e da fault parziali.

- **Tolleranza ai guasti**
  Il sistema prevede un meccanismo di auto recovery nel caso di errori localizzati senza perdita di dati.

- **Performance**
  Il sistema deve garantire prestazioni atte a non far percepire ritardi da parte del client, indipendentemente dal carico computazionale.

- **Scalabiltà**
  Il sistema deve adattarsi ad un andamento crescente del carico computazionale dovuto ad un elevato numero di richieste da parte degli utenti senza degradi percepibili.

- **Estensibilità**
  Il sistema deve prevedere un'architettura che permetta l'aggiunta di nuove funzionalità in modo facile e sicuro.

- **Manutenibilità**
  Il codice sorgente deve essere modulare, ben strutturato e chiaro in modo da permettere interventi di manutenzione.

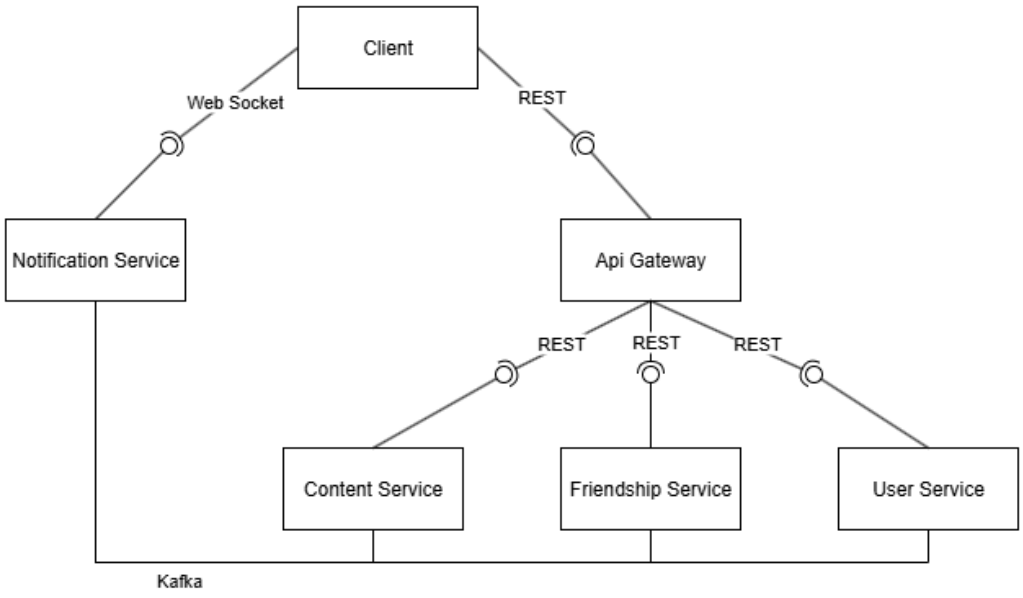
- **Sicurezza**
  Il sistema deve garantire confidenzialità nello scabio di dati sensibili (es. password), verifica dell'identità degli utenti, autorizzazione all'accesso di risorse coerente con le regole di dominio.

- **Distribuibilità**
  Il sistema deve essere facilmente distribuibile (deployabile)

3 Design

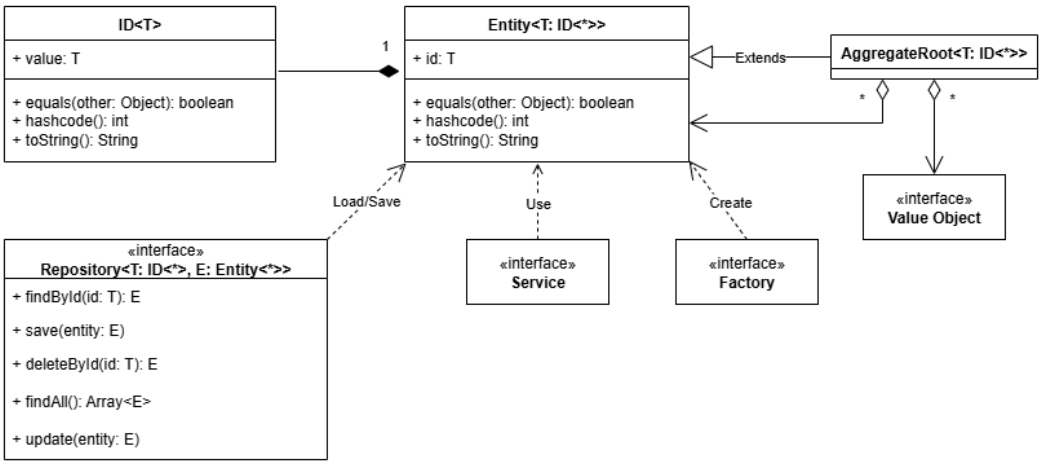
Col fine di raggiungere un buon livello di disponibilità e performance si è adottata un architettura a microservizi. Nella fattispecie l'architettura conta tre microservizi veri e prori: user-service, friendship-service e content-service che gesticono le risorse fondamentali del sistema, rispettivamente: utenti, amicizie/messaggi e contenuti. La persistenza dei microservizi è gestita separatamente seguendo il pattern database per service, tale scelta è giustificata dall' aumento della fault tollerance atto ad evitare che un singolo

guasto si ripercuota sull'intero sistema, ma solo in una sua porzione. Per garantire comunque una consistenza eventuale, l'architettura comprende kafka come middleware, con il compito di fornire un mezzo di coordinamento ai microservizi utile a mantere lo stato di questi ultimi coerente con gli altri. Come mostra il diagramma sottostante, è stato inserito anche un servizio aggiuntivo denominato notification-service, il quale svolge il ruolo di interfaccia fra kafka e il client. Quest'ultimo è infatti interessato a sottoscriversi a quegli eventi che lo riguardano e che vengono gestiti dal middleware (es. destinario sottoscritto all'evento di pubblicazione di un nuovo messaggio per lui). La sottoscrizione diretta, non intermediata, da parte del client, è ritenuta insicura e lascierebbe la possibilità ad un utente esperto di aggirare le regole di dominio intercettando eventi non a lui destinati; per questo motivo il servizio di notifica funge da intermediario, capace di autenticare il client e di impedire accessi non permessi alle risorse. Come punto unico di accesso al sistema, l'architettura comprende un api-gateway, il quale fornisce un unico nodo di autenticazione delle richieste http e un naturale punto di coordinamento per le richieste composte. La scelta di esporre sia notification-service che api-gateway al client è dettata dal requisito delle performance e della scalabilità; infatti sovrapporre i due ruoli (ingresso http e web socket) in un unico nodo può impattare negativamente sulle performance, soprattutto all'aumentare del numero di utenti connessi.



3.1 Structure

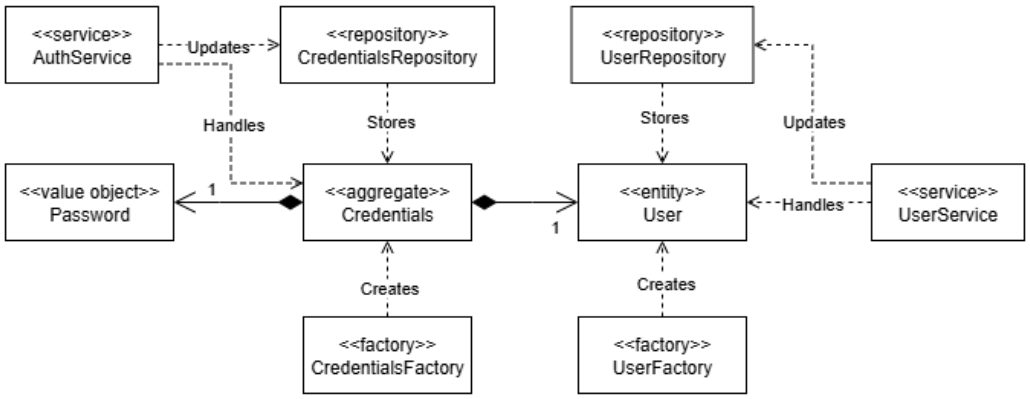
Il dominio è stato modellato seguendo i principi del DDD (Domain Driven Design), i cui building blocks sono rappresintati da entità di prima classe.



Il dominio è stato poi suddiviso in bounded context, ognuno dei quali è stato modellato come dominio a se stante.

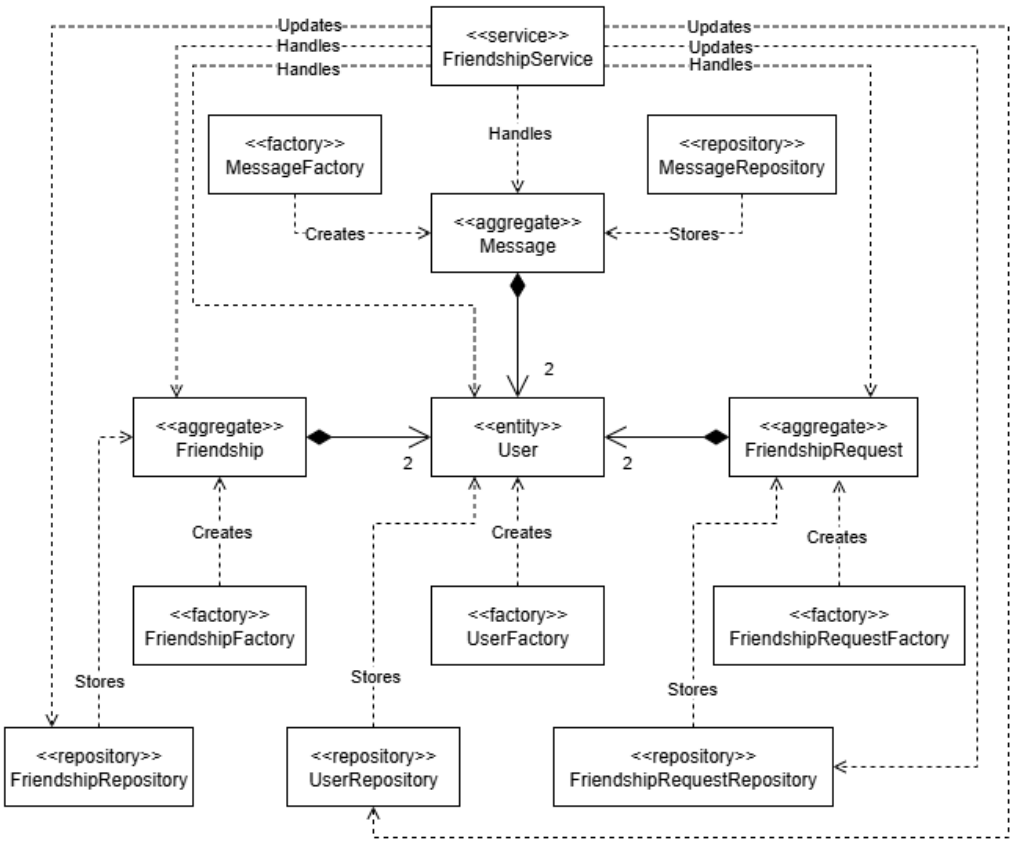
3.1.1 Users

Il contesto Users comprende tutto ciò che è inerente agli utenti e alla loro autenticazione.



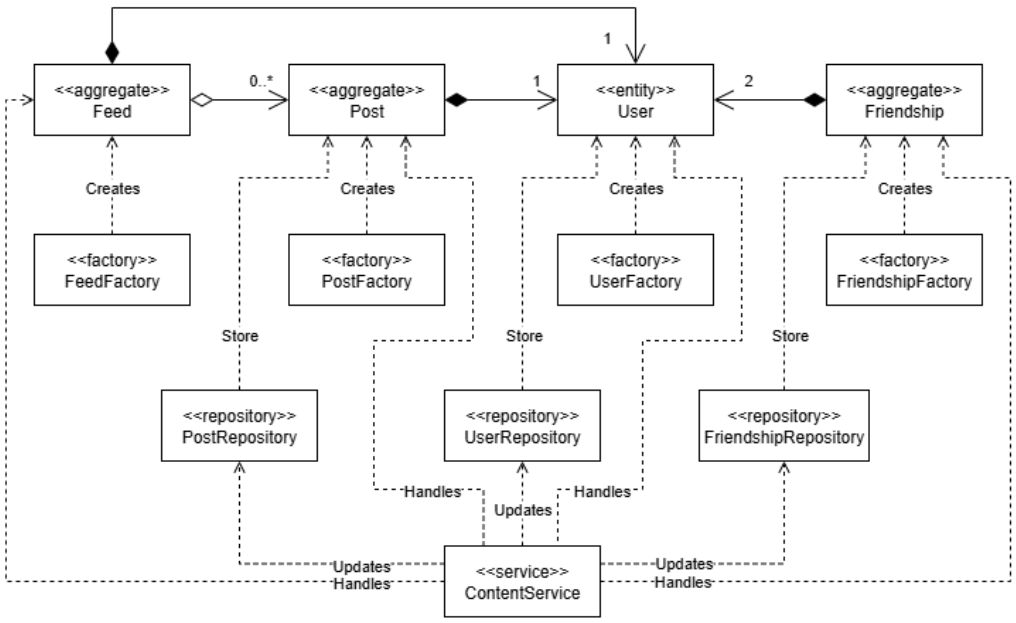
3.1.2 Friendships

Il contesto Friendships comprende tutto ciò che inerente all'amicizia fra gli utenti e i loro messaggi.



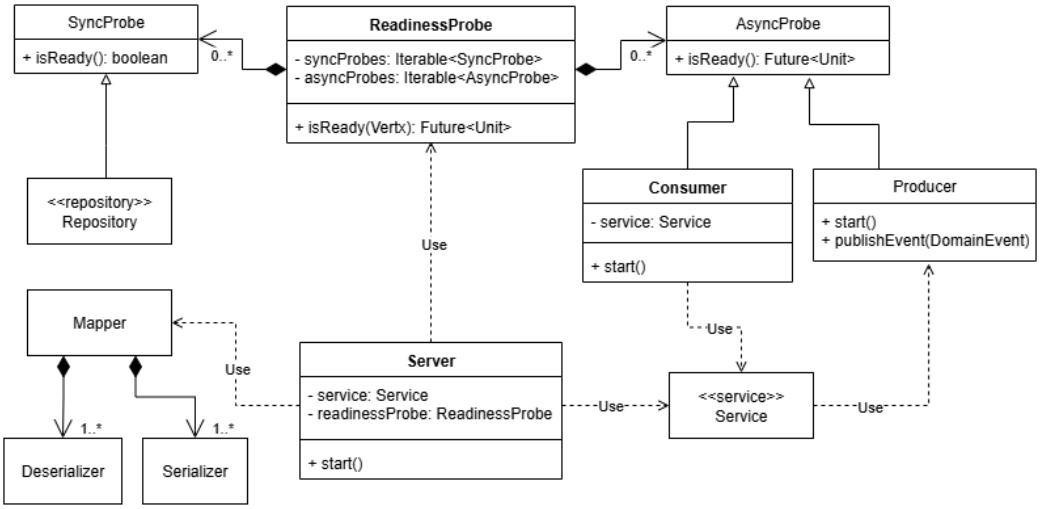
3.1.3 Contents

Il contesto Contents comprende tutto ciò che è inerente ai contenuti pubblicati e visualizzati dagli utenti.



3.1.4 Infrastructure

Ciò che non fa esplicitamente parte del dominio, e che costituisce la parte infrastrutturale dei servizi, è stato progettato in modo analogo per ogni componente del sistema. Di seguito una rappresentazione.



Si noti che all'atto implementativo i servizi possono variare leggermente dall'idea gen-

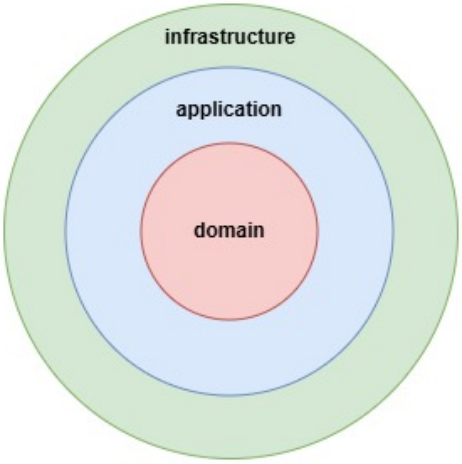
erale. Questo è dovuto principalmente alle esigenze del singolo componente, ma anche alle tecnologie usate per implementarlo.

3.1.5 Clean Architecture

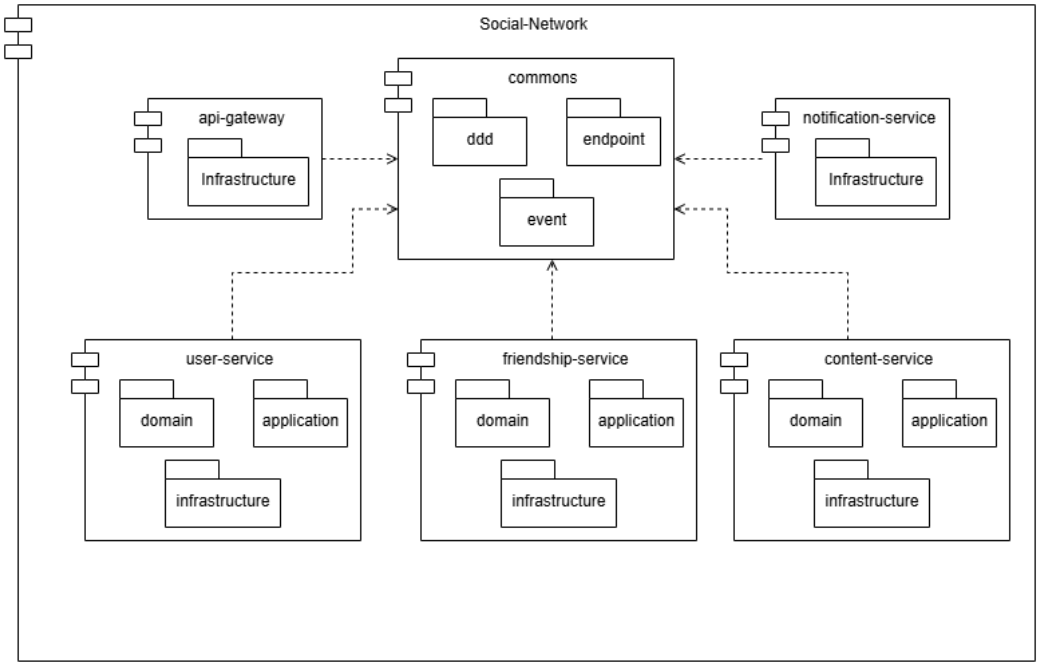
Col fine di migliorare la manutenibilità, i vari microservizi sono sviluppati secondo i principi della clean architecture (o architettura esagonale). Tale architettura suddivide il microservizio in layer e introduce una direzionalità nelle dipendenze fra i livelli. Nello specifico si visualizzano i layer come cerchi concentrici disposti uno dentro l'altro a "matriosca", ogni livello può dipendere solo da quelli che contiene e non viceversa. Nella declinazione specifica di questo progetto, sono stati individuati tre layer:

- **Domain**:
  contiene le entità del dominio, i value object e le factory.
- **Application**
  comprende i servizi e le interfacce a loro indispensabili.
- **Infrastructure**
  include le implementazioni dei repository e in generale tutto ciò che è legato alla comunicazione con l'esterno.

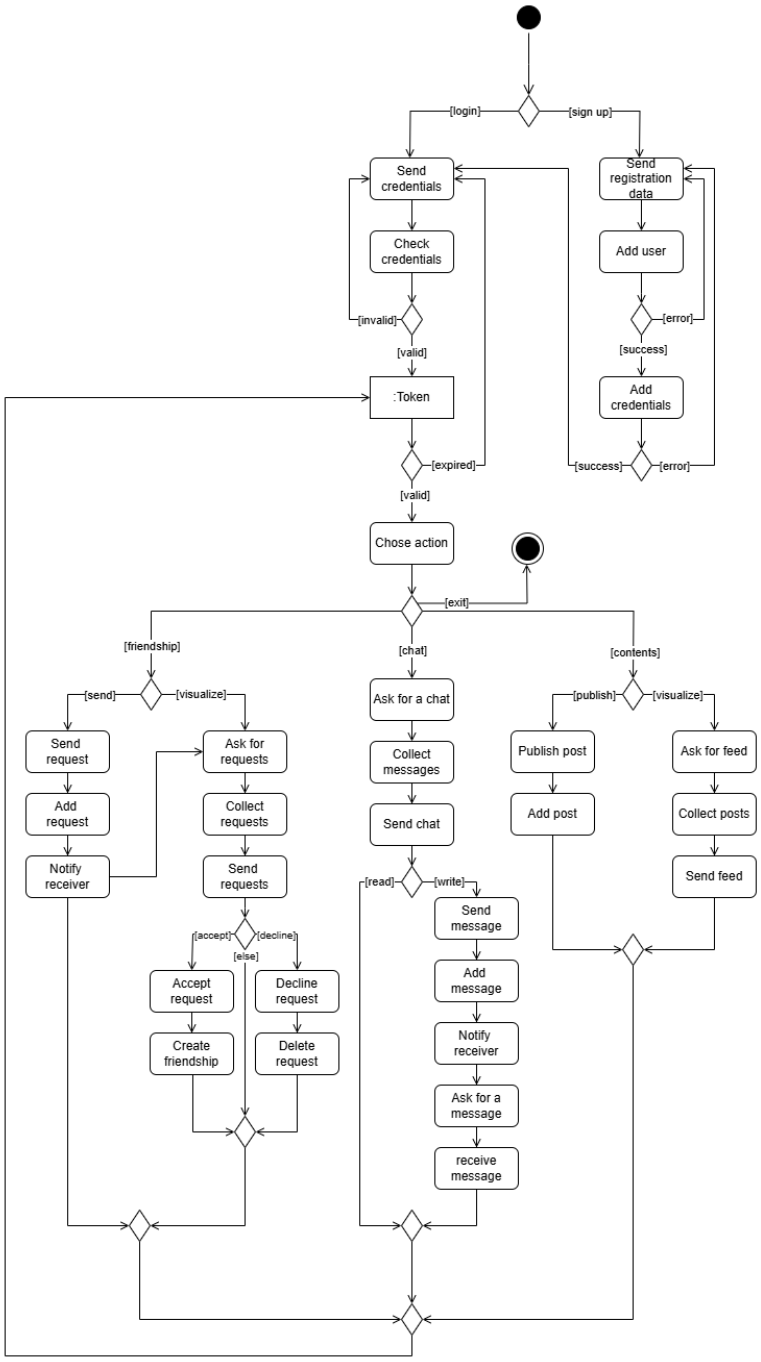
Di seguito un immagine che aiuta a visualizzare i layer e la loro profondità.



A livello di progetto, in senso globale, si è optato per una suddivisione in sotto progetti (moduli), raggruppando le porzioni di codice che sono utilizzate da più moduli in un modulo unico chiamato commons. La restante modularizzazione è naturalmente correlata ai diversi componenti del sistema, come mostra lo schema sottostante.


Social-Network
commons
ddd
endpoint
event
api-gateway
Infrastructure
notification-service
Infrastructure
user-service
domain
application
infrastructure
friendship-service
domain
application
infrastructure
content-service
domain
application
infrastructure

3.2 Behaviour



L'activity diagram mostrato sopra evidenzia come il sistema elabora le azioni prsentate negli use case alla sezione 1.1, sottolineando l'ordine temporale con il quale le azioni, e

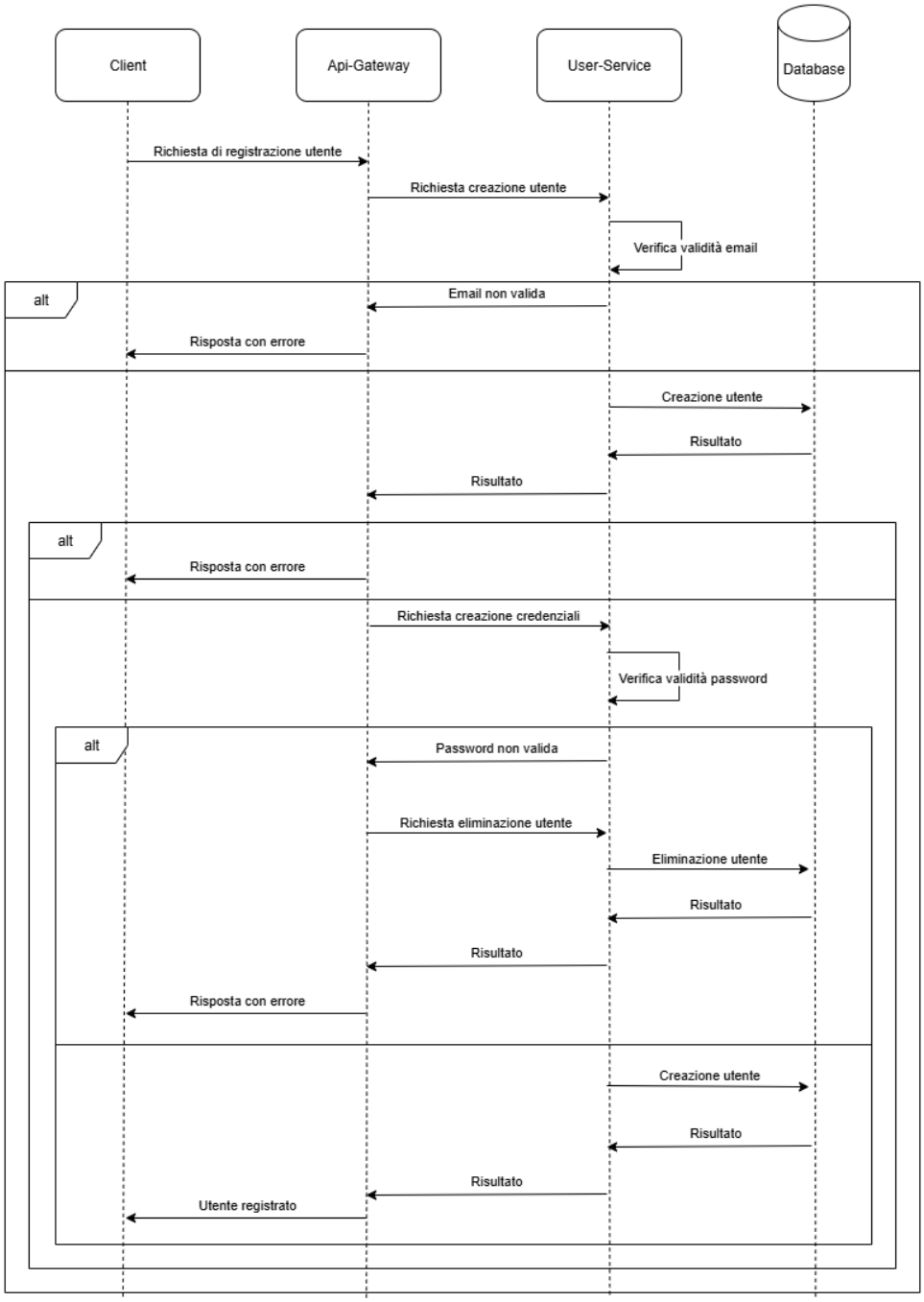
quindi la loro risoluzione, avvengono.

3.3 Interaction

Le modalità di interazione che compaiono nel sistema sono di tre tipologie:

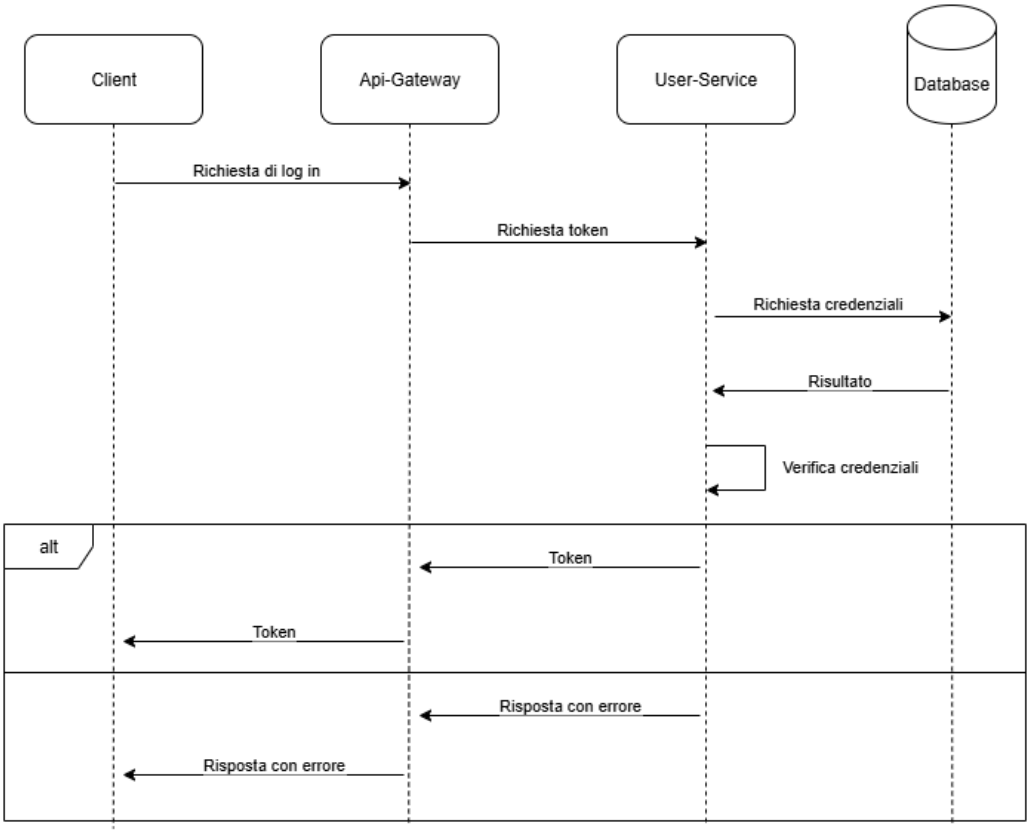
- **REST/HTTP(S)**:
  rappresentata dalla freccia piena (modalità "sincrona")
- **TCP/MYSQL**:
  rappresentata dalla freccia piena verso o da un database (modalità mista)
- **Evento kafka**:
  rappresentata dalla freccia spezzata e dalla notazione `<<evento>>` (modalità asincrona)
- **WebSocket**:
  rappresentata dalla freccia spezzata senza notazioni (modalità asincrona)

I diagrammi seguenti mostrano a livello logico come avvengono le interazioni fra client e servizi usando le rappresentazioni sopra descritte. Si noti che l'intento non è mostrare ogni possibile interazione ma in generale le tipologie possibili. Si specifica, inoltre, che il client comunica con i punti di accesso del sistema via https, mentre all'interno del sistema i servizi usano http.



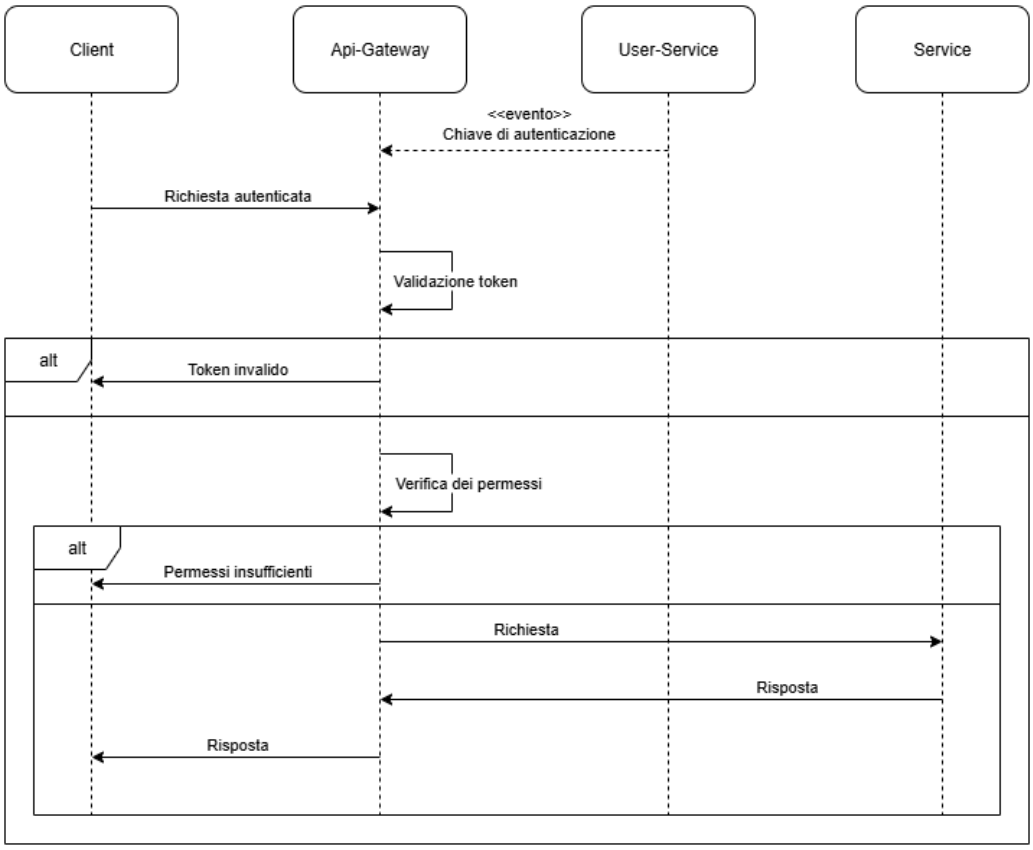
Questo diagramma descrive il processo di registrazione di un utente. Il client invia una richiesta POST con i propri dati (email, password, nome utente) ad api-gateway, il quale, orchestra un pattern SAGA per soddisfare la richiesta del client. Inizialmente api-gateway comunica a user-service di creare un nuovo utente, inviando un sottoinsieme dei

dati forniti dal client. user-Service verifica la validità dei dati (email) per poi rispondere con un errore o creando la nuova risorsa. Se questa fase ha successo, api-gateway prosegue richiedendo la creazione delle credenziali, inviando i dati restanti. Come nello step precedente user-service verifica la validità delle credenziali (password) per poi rispondere con un errore o creando la nuova risorsa. Nel caso in cui l'utente venga creato con successo ma le credenziali risultano invalide, api-gateway richiede un rollback tramite la richiesta di eliminazione della risorsa utente. Alla fine del processo il client riceve una risposta di errore nel caso uno step sia fallito o, altrimenti, una di avvenuta creazione. La scelta di implementare una strategia di composizione di richieste con rollback è nata dall'esigenza di anticipare il cambiamento; in futuro, infatti, potrebbero coesistere diversi servizi, anche esterni, di autenticazione, con il compito di mantenere e verificare le credenziali. Per ciò, una soluzione composta facilita l'adattamento del sistema a questo genere di scenari.

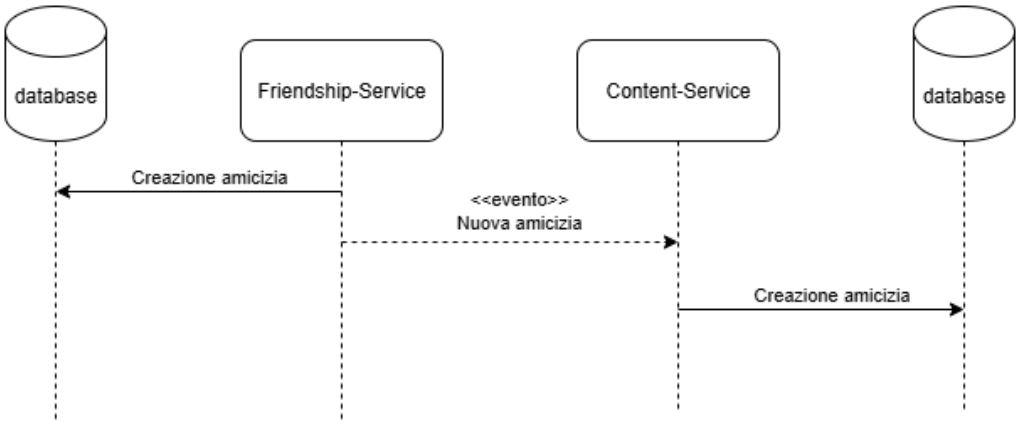


In questa sequenza è mostrato il processo di login. Il client effettua una richiesta POST ad api-gateway, inviando le credenziali (email, password). Dopo l'inoltro a user-service, quest'ultimo si occupa di recuperare le credenziali dell'utente dal database e confrontarle con quelle fornite dal client. In caso le password coincidano viene generato un token e

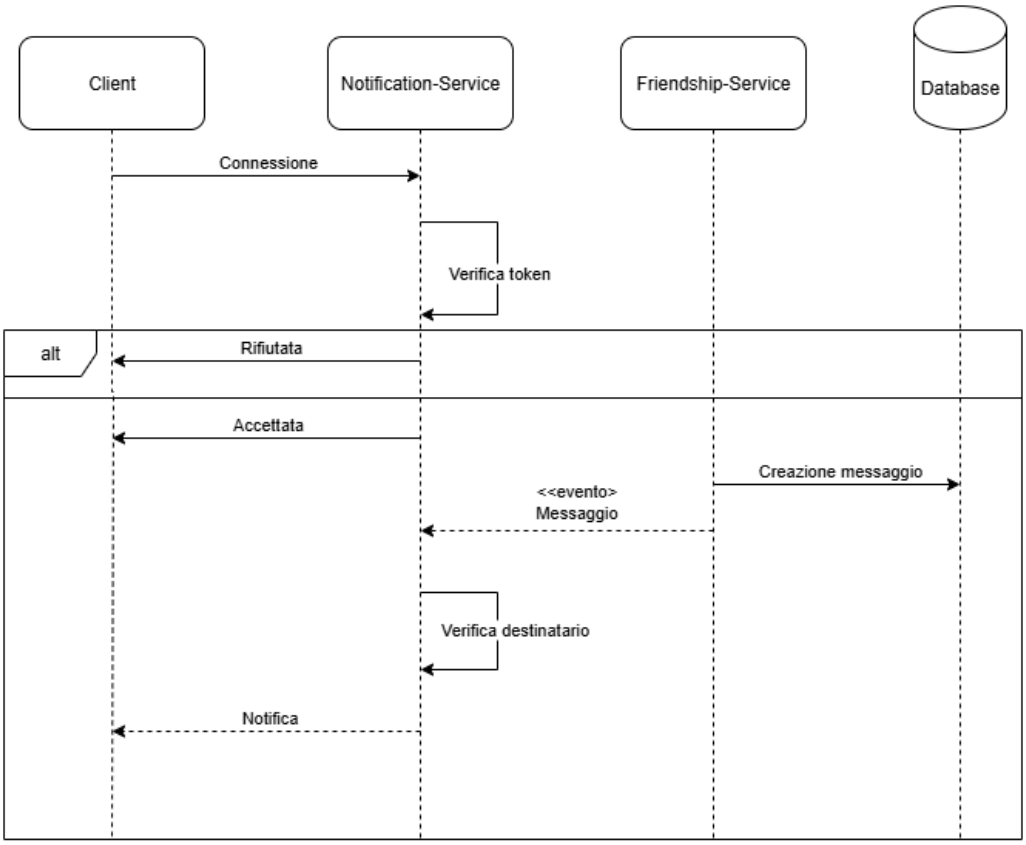
restituito al client, esso rappresenta l'identità dell'utente e dovrà essere inviato nelle richieste successive per garantire una richiesta autenticata. Il token ha una validità temporale che lo rende inutile allo scadere dell'intervallo previsto, dopo di che è necessario rieffettuare il login per ottenere un nuovo token valido.



In questa sequenza si mostra come avviene una richiesta autenticata. Astraendo dalla richiesta in se, che potrebbe coinvolgere diversi servizi, il diagramma mostra che l'autenticazione è delegata ad api-gateway. Quest'ultimo dispone infatti della chiave pubblica generata da user-service, con la quale è possibile validare i token firmati con chiave privata da user-service. Un token valido porta con se diverse informazioni sull'utente, come la sua identità, il suo ruolo e il suo stato. A seconda di queste informazioni e dalla richiesta del client, api-gateway applica le regole di dominio e, in caso i permessi siano sufficienti, procede ad inoltrare la richiesta al servizio pertinente. In caso di invalidità del token o permessi insufficienti, il client riceve una risposta di errore.



Questo diagramma mostra come avviene la comunicazione fra servizi e come si mantiene la consistenza del sistema. Quando una nuova risorsa viene creata, il servizio proprietario di quella risorsa (e solo lui) pubblica un evento kafka che descrive la risorsa. I servizi che per necessità usano quella risorsa, ma non ne sono responsabili, si sottoscrivono agli eventi sulla stessa e quando ricevono notifica di avvenuta creazione o modifica, aggiornano il proprio database.



Questa sequenza mostra come avviene la propagazione delle notifiche. Il client stabilisce una connessione websocket con notification-service, il quale autentica in modo analogo ad api-gateway la richiesta del client. Stabilita la connessione, qual'ora un servizio dovesse pubblicare un evento su un topic per cui notification-service ha un consumer, quest'ultimo applicherà la business logic e inoltrerà via wesocket l'evento al client interessato.

4 Implementation Details

Facendo seguito alle sezioni precedenti dove viene sviscerato il design del progetto, questa sezione fa da corredo per chiarire alcuni aspetti implementativi non trattati esplicitamente.

- **Deploy alternativi**
  Il progetto comprende due deploy distinti con caratteristiche differenti. Ai fini di testing viene mantenuto il deploy docker come versione light del sistema, mentre il deploy kubernetes è la versione completa che implementa i quality attribute esposti in analisi.

- **Risorse limitate**
  È evidente che esistono discrepanze fra quanto riportato nella sezione design e il deploy kubernetes. Nella fattispecie, nello script di deploy viene istanziato un unico cluster per la persistenza invece che uno per servizio; tale differenza è dovuta alle risorse hardware limitate che non consentivano di istanziare il sistema completo garantendone il funzionamento.

5 Self-assessment / Validation

Per valutare la correttezza del software prodotto sono stati prodotti test automatici che costituiscono il criterio di valutazione dell'aderenza del sistema ai requisiti. Questi acceptance test hanno lo scopo di validare il codice dei vari componenti, istanziati in forma "light weight" con docker. I test sono sviluppati in modo da essere indipendenti l'uno dall'altro, reistanziando un sistema vergine ad ogni esecuzione così da isolare i vari use case, evitando conflitti indesiderati e poco significativi. Per quanto riguarda la validazione del deploy, sono stati condotti test manuali atti a verificarne il corretto funzionamento. Si è quindi scelto di non duplicare gli acceptance test, separando la validazione del codice sorgente da quella del deploy. Tale decisione è frutto della tecnologia usata per deployare il sitema; infatti in kubernetes vengono istanziati gli stessi container docker creati per gli acceptance test, rendendo poco significativo e ridondante la produzione di test specifici per questo deploy. Gli acceptance test sono consultabili al path del repository "Social-Network/src/test/social/general/Simulations.kt".

6 Deployment

Il processo di deploy è l'insieme delle operazioni necessarie per rendere disponibile un sistema software in un ambiente di produzione (o test). Include attività come compilazione, configurazione, trasferimento dei file, avvio dei servizi e verifica del funzionamento. Un buon processo di deploy è automatizzato, ripetibile e affidabile, riducendo il rischio di errori e semplificando l'aggiornamento del sistema. In questa sezione si approfondisce il processo di deploy, centrale per garantire alta disponibolità e fault tollerance del sistema. Le tre sotto sezioni seguenti trattano gli aspetti fondamentali della distribuzione di The Social Network, suddivisi in base alle tecnologie utilizzate e al loro scopo.

6.1 Docker

Docker è una piattaforma che consente di creare, distribuire ed eseguire applicazioni all'interno di container, ambienti isolati e portabili che includono tutto il necessario per far funzionare il software. Funziona costruendo un'immagine da un file (Dockerfile), che viene poi eseguita come container. Docker semplifica la distribuzione delle applicazioni, garantendo coerenza tra ambienti di sviluppo, test e produzione. La necessità di usare uno strumento di containerizzazzione nasce dall'esigenza di un deploy portabile

e riproducibile. Ogni servizio in The Social Network è containerizzato attraverso un Dockerfile.

```
FROM openjdk:21-jdk AS build
WORKDIR /Social-Network
COPY ./build/ /Social-Network/
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "./libs/user-service.jar"]
```

Questo Dockerfile mostra come è stato containerizzato uno dei servizi del sistema. Parte da un'immagine base con Java 21 (openjdk:21-jdk), definisce una directory di lavoro (/Social-Network), copia al suo interno i file di build già compilati e espone la porta 8080 per ricevere richieste. Infine, avvia l'applicazione eseguendo il file user-service.jar. Ogni istruzione (FROM, COPY, ENTRYPOINT, ecc.) crea un layer, ovvero uno strato immutabile dell'immagine, rendendo il processo di build efficiente e riutilizzabile.

6.1.1 Kubernetes

Kubernetes è una piattaforma open source per l'orchestrazione di container, progettata per automatizzare il deploy, la scalabilità e la gestione di applicazioni containerizzate in ambienti distribuiti. Funziona eseguendo i container all'interno di pod, l'unità base di esecuzione in Kubernetes, che può contenere uno o più container con accesso condiviso alle risorse di rete e storage. I pod vengono gestiti da oggetti chiamati Deployment, che definiscono il numero di repliche desiderate, le strategie di aggiornamento e il controllo del ciclo di vita dei pod. Per separare la configurazione dal codice, Kubernetes utilizza risorse come le ConfigMap, che permettono di iniettare parametri di configurazione (es. variabili d'ambiente o file di configurazione) nei container in modo dinamico e riutilizzabile. Per esporre i pod all'interno del cluster o all'esterno, si usano le risorse Service, che astraggono l'accesso ai pod e forniscono funzionalità di bilanciamento del carico. Quando è necessario esporre un'applicazione HTTP/S all'esterno del cluster, si usa l'Ingress, che gestisce il routing in base al nome di dominio e al percorso richiesto, spesso dietro un controller come NGINX. Per gestire la scalabilità automatica in base al carico, Kubernetes offre gli Horizontal Pod Autoscaler (HPA), che monitorano metriche come l'utilizzo della CPU e aumentano o riducono il numero di pod in base alla domanda. L'interazione con il cluster avviene tramite lo strumento a riga di comando kubectl, che consente di creare, aggiornare, osservare e cancellare le risorse Kubernetes, interagendo con il server API del cluster. Per sviluppare e testare il progetto in ambiente locale, è stato utilizzato Minikube, uno strumento leggero che crea un cluster Kubernetes su una singola macchina virtuale. Minikube è ideale per scopi didattici e di sviluppo, perché replica in piccolo l'ambiente Kubernetes reale, permettendo di eseguire i comandi kubectl, testare Ingress, Services e Deployments, e simulare scenari di produzione in modo controllato.

6.1.2 Helm

Helm è un package manager per Kubernetes che semplifica il deploy, l'aggiornamento e la gestione delle applicazioni nel cluster. Funziona tramite i chart, pacchetti che descrivono risorse Kubernetes (come pod, service, ingress, deployment, ecc.) usando file YAML con supporto a template e un linguaggio di scripting che permette condizioni, cicli e sostituzioni dinamiche. Questo rende i chart altamente configurabili e riutilizzabili in contesti diversi. Nel progetto i chart sono così organizzati. Il chart chiamato microservice gestisce i servizi containerizzati. È progettato per essere semplice da configurare, consentendo di specificare facilmente il tipo di servizio scegliendo tra valori definiti a priori che rappresentano i vari servizi del sistema. Inoltre, è possibile sovrascrivere alcuni valori di default, ad esempio variabili d'ambiente utilizzate per la connessione a database o a Kafka. Per migliorare affidabilità e prestazioni, questo chart include:

- La configurazione di una readiness probe, che verifica lo stato dei container prima di inviare traffico.
- Un Horizontal Pod Autoscaler (HPA) che scala automaticamente i pod in base al carico.
- Un Service per gestire il load balance delle repliche

Il secondo chart, chiamato ingress, contiene tutte le risorse di tipo Ingress del sistema. Qui la configurazione è minimalista: basta specificare il nome del servizio backend e la porta per instradare correttamente il traffico HTTP/S esterno verso i servizi interni. Per i componenti di terze parti, ho utilizzato:

- Il chart Bitnami Kafka, per gestire il broker Kafka in modo affidabile e facilmente aggiornabile.
- I chart InnoDB Cluster (con l'operatore dedicato), per la gestione del database MySQL in alta disponibilità.

Questa struttura modulare con Helm permette di mantenere flessibilità, semplicità di configurazione e facilità di aggiornamento in tutto il sistema in Kubernetes.

6.1.3 Script

Il progetto è progettato per essere facilmente deployabile tramite l'esecuzione dello script deploy.cmd, che automatizza tutte le operazioni necessarie per installare e configurare l'intero sistema nel cluster Kubernetes. Allo stesso modo, è possibile rimuovere completamente il deploy eseguendo lo script undeploy.cmd, che pulisce tutte le risorse create, garantendo un ambiente pulito e pronto per eventuali nuove installazioni. Per poter eseguire correttamente questi script, è necessario avere installati e configurati i seguenti strumenti:

- **Docker**, per gestire le immagini container.

- **Helm**, per il deploy e la gestione dei chart Kubernetes.
- **Kubectl**, per interagire con il cluster Kubernetes.
- **Minikube**, che consente di creare un cluster Kubernetes locale su cui effettuare il deploy.

Questa soluzione permette di semplificare notevolmente le operazioni di deployment e teardown, rendendo il processo ripetibile, veloce e accessibile anche a chi non ha esperienza diretta con Kubernetes o Helm.

7 Usage Examples

Registrazione

```
curl -i -k https://127.0.0.1/users
-d '{"email":"youremail@domain.org","username":"username","password":"password"}'
-H "Host: api.social.local"
```

La password deve essere lunga almeno 8 caratteri e contenere almeno un carattere speciale, una maiscola e un numero. L'email deve essere nel formato mostrato nell'esempio.

Login

```
jwt=$(curl -s -k https://127.0.0.1/login
-d '{"email":"youremail@domain.org","password":"password"}'
-H "Host: api.social.local")
```

Si consiglia di salvare in una variabile il token, come nell'esempio, siccome serve per effettuare le successive richieste autenticate.

Richiesta di amicizia

```
curl -i -k https://127.0.0.1/friends/requests/send
-d '{"from":"youremail@domain.org","to":"anotheremail@domain.org"}'
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

Visualizzare richieste di amicizia

```
curl -i -k https://127.0.0.1/friends/requests/youremail@domain.org
-d '{"from":"youremail@domain.org","to":"anotheremail@domain.org"}'
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

Accettare richiesta di amicizia

```
curl -i -k -X PUT https://127.0.0.1/friends/requests/accept
-d '{"from":"youremail@domain.org","to":"anotheremail@domain.org"}'
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

Si noti che deve essere l'utente nel campo to ad accettare la richiesta, quindi è necessario che jwt faccia riferimento al suo token.

Inviare un messaggio

```
curl -i -k https://127.0.0.1/friends/messages/send
-d '{"sender":"youremail@domain.org","receiver":"anotheremail@domain.org","content":"cont
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

Sender e receiver devono essere amici per potersi inviare messaggi.

Leggere una chat

```
curl -i -k https://127.0.0.1/friends/messages/chat/youremail@domain.org/anotheremail@doma
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

Pubblicare un contenuto

```
curl -i -k https://127.0.0.1/contents/posts
-d '{"user":{"email":"youremail@domain.org","username":"username"},"content":"content"}'
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

Visualizzare il feed

```
curl -i -k https://127.0.0.1/contents/posts/feed/youremail@domain.org
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

Visualizzare il feed filtrato

```
curl -i -k https://127.0.0.1/contents/posts/feed/youremail@domain.org?keyword=keyword
-H "Host: api.social.local" -H "Authorization:Bearer $jwt"
```

8 Conclusions

Il sistema è stato realizzato coerentemente con i requisiti definiti. Grazie al deploy kubernetes, si è raggiunto un buon livello di tolleranza all'errore e disponibilità, oltre che di performance. Il codice sorgente risulta ben organnizzato e stabile. Nonostante ci sia margine di miglioramento in merito, si è raggiunto un buon livello di manutenibilità, garantendo ammodernamenti poco costosi.

8.1 Future Works

In futuro il sistema potrebbe comprendere le seguenti migliorie:

- Interfaccia grafica che permetta una visualizzazione gradevole e migliore la user experience
- Sistema di refresh token che eviti di dover rieffettuare il login

- Sistema di rotazione delle chiavi di autenticazioni che migliori la sicurezza nel tempo
- Supporto a nuove tipologie di contenuti

8.2 What did we learned

La realizzazione di questo progetto si è dimostrata un occasione per approfondire in modo pratico le riflessioni teoriche fatte dirante il corso in merito ai quality attribute; mi ha dato anche modo di avvicinarmi al mondo dev ops tramite l'utilizzo di docker e kubernetes per il deploy. Inoltre, è stato un banco di prova per consolidare le mie abilità di programmazione e progettazione, in un contesto architetturale moderno come quello dei microservizi.