MovieParty with Microservices

Sistemi Distribuiti

Rei Beshiri - 950330
rei.beshiri@studio.unibo.it

Andrea Vaienti - 951454
andrea.vaienti2@studio.unibo.it

29 giugno 2021

Sommario

Nel seguente elaborato si intente simulare la realizzazione di un progetto commissionato da un'azienda fittizia, in cui la prestazione riguarda l'aggiornamento di una piattaforma Web. Questa richiesta deriva da un notevole incremento di popolarità del sistema, in cui si desidera quindi investire ottenendo un software maggiormente scalabile, resiliente ai guasti e con una maggiore indipendenza tra i vari servizi esposti. Il team di sviluppo adotterà quindi varie tecniche di ingegnerizzazione per soddisfare le richieste del cliente, aumentando al contempo il business value del prodotto.

Indice

Capitolo 1

Descrizione del progetto

Nel seguente elaborato si intente simulare la realizzazione di un progetto commissionato da un'azienda fittizia, in cui la prestazione riguarda l'aggiornamento di una piattaforma Web, nonché principale fonte di redditto per il cliente. L'investitore è fortemente intenzionato a migliorare la propria piattaforma in quanto vede in lei un enorme potenziale. Essendo stato informato durante l'implementazione dell'attuale sistema che tale soluzione avrebbe funzionato correttamente solo con un utenza particolarmente limitata, visto l'attuale e repentino incremento di utilizzatori, si è deciso di voler migliorare l'infrastruttura di tale piattaforma. Un aumento così rapido di utenza potrebbe portare infatti ad un sovraccarico del sistema per come è attualmente concepito, causando problematiche che potrebbero allontanare utenti dalla piattaforma. L'azienda considera il problema di gravità elevata e quindi da risolvere al più presto.

1.1 La piattaforma

La piattaforma da aggiornare consiste in una Web App in cui è possibile visionare in compagnia o singolarmente i film presenti all'interno del catalogo offerto dal sito. La caratteristica principale che offre l'applicazione consiste nella creazione di una "stanza privata", denominata party, a cui posso essere invitati solamente gli amici. All'interno del party avverrà la riproduzione sincronizzata del video tra tutti i partecipanti. Chiunque potrà quindi fermare, avanzare o arretrare la riproduzione, in modo tale da riguardare i momenti salienti, avere tempo di preparare i pop-corn o saltare parti noiose. Ma non solo; il party è infatti provvisto di una chat per commentare in tempo reale ciò che si sta visionando, per poter condividere le proprie opinioni, pensieri o giudizi con tutti i partecipanti del party. MovieParty presenta una catalogazione dei vari contenuti in categorie, facilitando così la scelta dell'utilizzatore finale. Sarà inoltre presente una barra di ricerca per cercare in modo facile e veloce contenuti multimediali non presenti all'interno delle categorie. Si può quindi asserire che l'applicazione dispone delle seguenti funzionalità:

- Sistema per la gestione sicura delle credenziali d'accesso. Solamente gli utenti autenticati potranno accedere ai servizi offerti dalla piattaforma.
- Sistema per la gestione della lista amici di ciascun utente.
- Sistema per la gestione di informazioni e servizi in real-time.
- Sistema di notifiche per la visualizzazione di richieste di amicizia o inviti al party.

Precedente architettura del sistema

L'applicativo era stato costruito basandosi sullo stack MERN, che come l'acronimo descrive comprende le seguenti tecnologie:

- MongoDB
- Express
- React
- Node.js

MERN è una delle numerose varianti dello stack MEAN (dove il tradizionale framework frontend Angular.js viene sostituito con React.js) che permette la costruzione di un'architettura a 3 livelli (frontend, backend, database) utilizzando interamente Javascript e JSON.
All'interno del progetto erano state utilizzate diverse tecnologie, differenti da frontend a backend. Per la parte frontend erano state utilizzate Axios, Redux e React; mentre per la parte backend erano state utilizzate Express e Moongose. Per la gestione dei dati era stato selezionato MongoDB come DBMS (Database management system) in quanto grazie alla memorizzazione di documenti JSON, facilitava notevolmente l'archiviazione, la manipolazione e la rappresentazione dei dati ad ogni livello dell'applicazione.

Maggiori informazioni sul design, la struttura, il codice sorgente e la documentazione di tale applicazione sono reperibili qui: MovieParty.

1.1.1 Scenari

Versione Desktop

Per la realizzazione dell'applicazione dal punto di vista dello user interface design, in particolare per quanto riguarda la progettazione visiva e dell'interazione si è cercato di mantenere uno stile quanto più semplice e intuitivo possibile. L'app è progettata per essere fruita al massimo delle sue potenzialità in modalità Desktop, in quanto agevola notevolmente l'utilizzo della chat real-time durante la riproduzione di risorse multimediali. Tuttavia si è prestata attenzione anche all'interfaccia per dispositivi tablet e mobile in modo da rendere l'esperienza il più godibile possibile. Il primo approccio dell'utente con l'applicazione avviene mediante la seguente interfaccia:

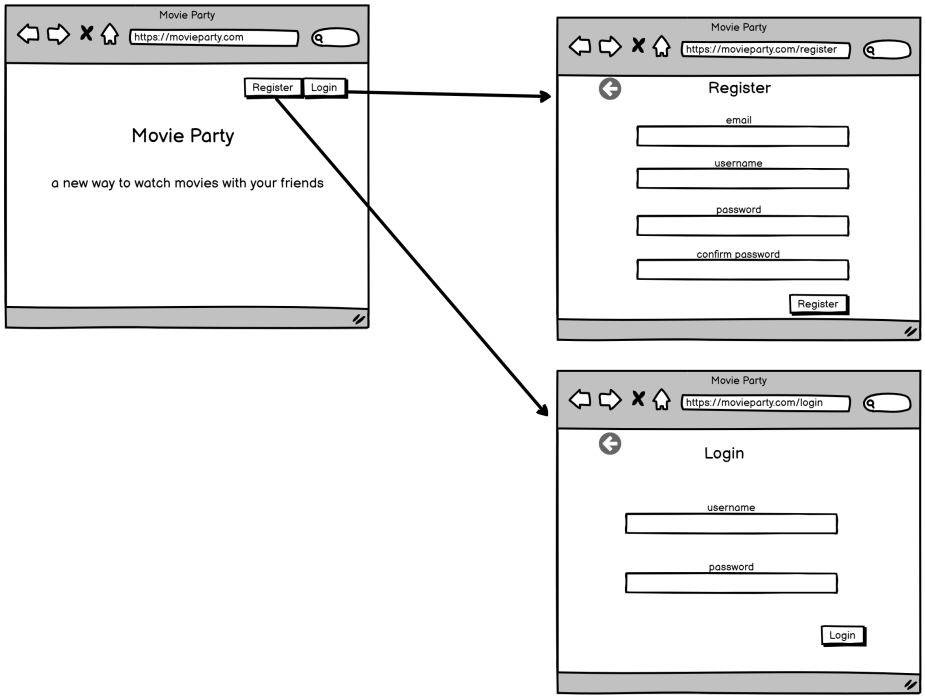


Figura 1.1: Rappresentazione della Homepage dell'app.

Come rappresentato in Figura 1.1 l'applicativo richiede come prima cosa l'autenticazione dell'utente. Questo può avvenire attraverso una registrazione al sito nel caso in cui non si disponga di un account oppure attraverso un operazione di login se già in possesso di credenziali di accesso. Sin da subito è possibile visualizzare come il sito presenti schermate minimal per non trarre in confusione l'utilizzatore.

Effettuato il login si viene reindirizzati nella schermata principale dell'applicazione. Al suo interno vi è presente una barra di navigazione che facilità l'utilizzo di diverse funzionalità messe a disposizione dall'applicativo, come ad esempio la ricerca di film o la visualizzare degli amici attualmente attivi.

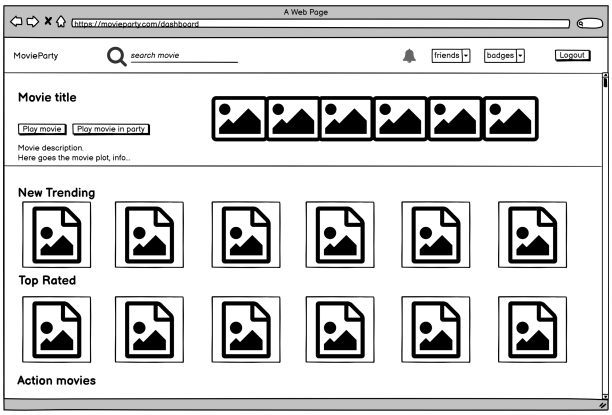


Figura 1.2: Rappresentazione della Schermata principale dell'app.

Per agevolare gli utenti, sempre all'interno dell'homepage dell'applicazione è possibile inviare richieste di amicizia per ampliare l'insieme di conoscenti presenti nel sito o visualizzare le notifiche perse. Oltre a ciò, vi sono presenti anche i badges, ovvero contenuti collezionabili che l'utente potrà sbloccare eseguendo specifiche azioni.

Sotto alla barra di navigazione sono elencati i contenuti multimediali presenti all'interno del sito suddivisi in categorie, ad esempio nei trending vi saranno presenti solamente i film di successo dell'ultimo periodo. Alla selezione di uno specifico film, nella parte inferiore della schermata verranno mostrate le informazioni riguardati tale opera, come il titolo e la trama. Attraverso due bottoni viene offerta la possibilità di riprodurre il contenuto multimediale singolarmente o in modalità party.

Figura 1.3: Rappresentazione del Poster in seguito alla selezione di un film.

La riproduzione del contenuto in modalità singola avvia un player video adibito alla visualizzazione di quest'ultimo, come rappresentato in Figura 1.4.

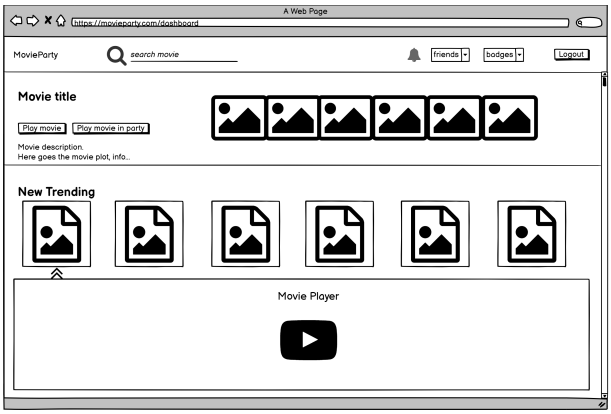


Figura 1.4: Rappresentazione del Player all'interno della schemata principale per la riproduzione in modalità singola.

La riproduzione del contenuto in modalità Party invece reindirizza l'utente in una nuova schermata, detta Lobby, nella quale potrà invitare gli amici attualmente online.

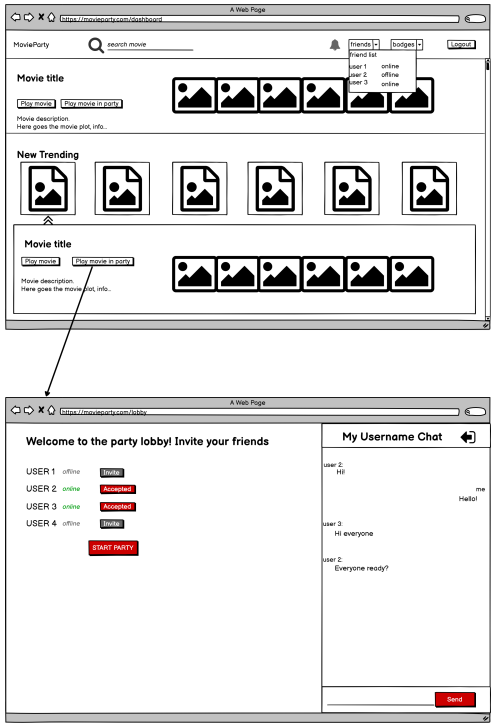


Figura 1.5: Cambiamento di schermata durante l'avvio della modalità Party

All'interno della Lobby è presente una chat che consente a tutti i partecipanti d'interagire, in attesa che il Leader (creatore della Lobby) decida di avviare la riproduzione sincronizzata del film.

Una volta avviata la riproduzione del contenuto multimediale, tutti i partecipanti alla sessione visualizzeranno una nuova schermata.

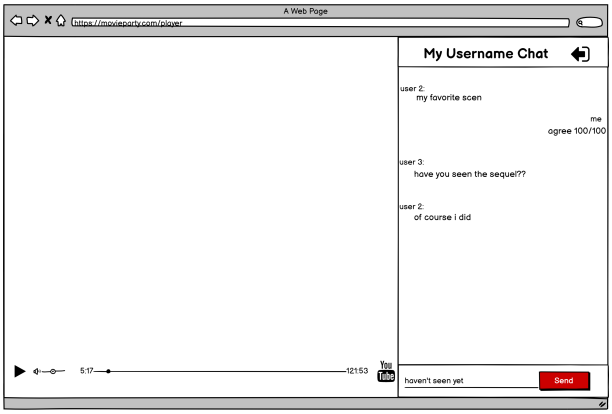
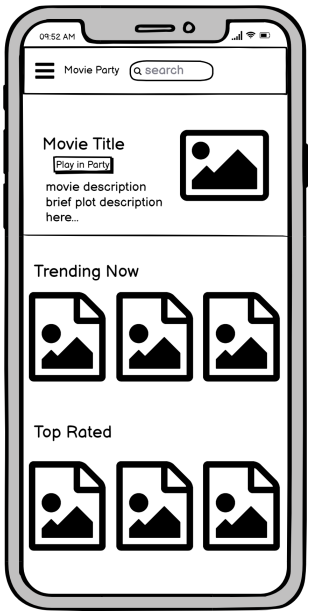


Figura 1.6: player with chat

Anche al suo interno è presente una chat mediante la quale è possibile comunicare durante la visione dell'opera. Gli utenti possono inoltre interagire con la riproduzione del video mettendolo in pausa, avanzando di tempo o indietreggiando. Queste azioni verranno poi contemporaneamente propagate a tutti i partecipanti del party.

Versione Mobile

La versione mobile non si differenzia molto da quella desktop, anche se, per le ridotte dimensioni dello schermo, qualche rivisitazione è presente. La schermata principale infatti risulta come in Figura 1.7a. In seguito al fatto che i dispositivi mobile siano solitamente utilizzati in verticale, si è cercato di strutturare l'applicazione in maniera tale da sfruttare l'intera area disponibile. Si è quindi cercato di rendere la User Experience godibile tanto quanto la versione desktop mantenendo un'interfaccia semplice e intuitiva.



(a) Schermata principale Mobile App.

(b) Schermata Pre lobby

(c) Schermata con la riproduzione in corso

Figura 1.7: Principali schermate da mobile

1.2 Requisiti richiesti dal cliente

In seguito ad una serie di colloqui avvenuti tra l'azienda e il team di sviluppo, si è cercato di capire le funzionalità e le caratteristiche del sistema che potessero accondiscendere i desideri del committente ma che allo stesso tempo permettessero al prodotto di guadagnare business value. Si è quindi concluso che la soluzione dovrà rispondere ai seguenti requisiti:

- Sistema maggiormente scalabile che consenta una facile ripartizione delle risorse nel caso in cui la piattaforma diventi sempre più popolare, evitando così rallentamenti.
- Sistema maggiormente resiliente ai guasti, in modo tale da poter offrire un servizio continuo agli utilizzatori senza disservizi. Si vuole evitare che un singolo errore in un componente possa avere ripercussioni sull'intera applicazione.
- Maggiore indipendenza strutturale, in maniera tale da poter sviluppare, distribuire e ridimensionare ciascun componente individualmente, senza dover influenzare il comportamento degli altri componenti.
- Maggiore predisposizione alla sperimentazione e alla distribuzione di nuovi componenti, in modo tale da accelerare il time-to-market di nuove funzionalità. Si vuole quindi poter collaudare nuove idee e nel caso ripristinare configurazioni precedenti con facilità.

Si è deciso inoltre di mantenere intatto tutto quello che riguarda l'UI (User Interface), in quanto ritenuta estremamente soddisfacente dagli utenti.

1.3 Obiettivi

Il progetto si pone quindi come obiettivo la reingegnerizzazione della piattaforma sopra descritta. Ciò avverrà in seguito ad un'approfondita analisi del sistema, che permetterà di individuare le porzioni del sistema rilevanti che dovranno poi essere modificate per ottenere le caratteristiche richieste dal committente. Gli obiettivi possono quindi essere elencati nel seguente modo:

- Analisi approfondita del sistema attuale e delle sue interazioni Client-Server.
- Modellazione concettuale di una nuova architettura che consenta il raggiungimento dei requisiti richiesti.
- Implementazione di tale soluzione attraverso l'utilizzo di framework che semplifichino il lavoro.
- Creazioni di test per verificare il corretto funzionamento del sistema.

Verrà inoltre eseguito del code refactoring attraverso l'introduzione di framework adatti alle circostanze che consentano di semplificare il lavoro e migliorare l'ingegnerizzazione del sistema. Tale refactoring consentirà una maggiore leggibilità, manutenibilità, riusabilità e estendibilità del codice.
Principalmente l'elaborato prevederà l'intera reimplementazione del server e delle comunicazioni ad esso associate (attraverso l'utilizzo di API Restful), mantenendo praticamente inalterato il client visto che come richiesto la parte grafica deve essere mantenuta intatta, dopo i complimenti ottenuti dalla community.

1.4 Expected Deliverables

Il team di sviluppo si impegna a consegnare:

- documentazione del processo di progettazione e sviluppo del software
- i sorgenti dell'applicazione
- i sorgenti di testing

Capitolo 2

Processo di sviluppo

Per la realizzazione dell'elaborato è stata adottata la metodologia di sviluppo AGILE che permette di realizzare un progetto per fasi, denominate *sprint*, ognuna delle quali focalizzata su nuove funzionalità. Al cliente, simulato dai membri del team, viene mostrato il lavoro svolto con continuità, per verificarne il gradimento, ed è quindi possibile apportare modifiche con estrema velocità. In questo modo si incrementa l'efficienza del gruppo di lavoro e dunque la produttività. In particolare si è fatto riferimento a **Scrum**, effettuando sprint settimanali per determinare i compiti da svolgere per ciascun membro del team e rivedere le funzionalità da implementare.
Il primo periodo del processo di sviluppo è stato caratterizzato dalla modellazione dell'architettura generale del progetto, attraverso la definizione dei requisiti e una prima progettazione mediante schemi concettuali. Successivamente, dopo aver assegnato a ciascun funzionalità individuata durante la fase di progettazione una priorità, si è pianificato il lavoro da svolgere durante le settimane attraverso il compimento di Sprint Planning.

2.1 Meeting

I meeting tra i componenti del team sono avvenuti con frequenza regolare mediante la piattaforma Microsoft Teams. All'inizio di ogni settimana sono stati effettuati in modo congiunto la **Sprint Review** e lo **Sprint Planning**. In questo tipo di meeting è stato passato in rassegna il lavoro svolto nella settimana appena conclusa e sono stati determinati gli obiettivi e i compiti da svolgere per ciascun membro del team. Ogni componente è stato aggiornato sullo stato di avanzamento dei task individuati e la pianificazione del lavoro è stata aggiornata di conseguenza. Altri incontri si sono svolti nel mezzo della settimana per aggiornare i membri del team sui progressi svolti e far emergere problematiche riscontrate allo scopo di studiare insieme delle soluzioni, similmente a come avverrebbe nel **Daily Scrum** a cui si è fatto riferimento.

2.2 Divisione dei task

All'avvio di ciascuno sprint settimanale a ciascun componente del team sono stati assegnati una serie di task da svolgere durante la settimana, in modo da definire obiettivi che ogni membro del team è tenuto a portare a termine. Un task rappresenta uno o più requisiti tra quelli individuati nel primo periodo del processo di sviluppo.

2.3 Revisione dei task

Alla fine di ciascuno sprint, durante la **Sprint Review**, ciascun componente del team ha informato gli altri dei progressi nello svolgimento dei task a lui assegnati, notificando le varie difficoltà avute per lo svolgimento dei task. Sulla base del risultato della review è stata aggiornata e revisionata la pianificazione del lavoro, controllando eventuali ritardi sulla tabella di marcia e conseguentemente adottando relative contromisure.
Generalmente, al compimento di un task, prima della sua effettiva terminazione è stata fatta un'azione di code review dove il codice prodotto dal responsabile del task è stato mostrato a tutti i membri del team con l'obiettivo di individuare possibili refactor atti a migliorare la qualità del codice prodotto, diminuendo così potenziale debito tecnico.

2.4 Scelta degli strumenti

Sono stati utilizzati differenti tool a supporto del processo di sviluppo. L'obiettivo dell'utilizzo di tali tool è quello di servire gli sviluppatori durante tutto il processo di sviluppo automatizzandolo, con lo scopo di migliorarne l'efficienza e di consentire al gruppo di concentrarsi maggiormente sulla risoluzione dei requisiti del progetto stesso.

- **NPM** come strumento di build automation, automatizzando una varietà di compiti software quotidiani durante lo sviluppo del progetto come la compilazione del codice sorgente o l'installazione di librerie.
- **GitLab** come servizio di hosting del codice sorgente e file utilizzati durente il processo di sviluppo.

Capitolo 3

Paradigmi a confronto

Per rispondere alle esigenze del committente si è quindi deciso di effettuare un passaggio dall'attuale architettura monolitica a un architettura a microservizi. Di seguito vengono quindi descritti tali stili architetturali.

3.1 Architettura monolitica

L'architettura monolitica è tradizionalmente considerata la tipologia principale con cui vengono sviluppate le applicazioni. Tale architettura prevede di essere costituita da un'entità unica e indivisibile. Ciò significa che i vari componenti sono strettamente collegati tra di loro, venendo eseguiti come un singolo servizio. Questo comporta un'assenza di latenza di rete per le comunicazioni che avvengono tra i vari servizi, in quanto esse avvengono attraverso chiamate locali. Come è possibile visualizzare nella figura sottostante, l'accesso alle funzionalità servite da tale architettura avviene attraverso un'interfaccia. La suddetta interfaccia definisce l'insieme di operazioni (e la modalità di interazione con esse) che possono essere invocate dalle applicazioni esterne.

Vantaggi

I vantaggi che un architettura monolitica comporta sono i seguenti:

- **Semplicità di implementazione**
  Essendo l'approccio monolitico la modalità standard con cui vengono realizzate le applicazioni, qualsiasi team di sviluppo possiede le giuste conoscenze e capacità per sviluppare un'applicazione di tale tipologia.
- **Facilità di testing**
  Poiché un'app monolitica è una singola unità indivisibile, l'implementazione e l'esecuzione di test risulta facile e veloce.

Svantaggi

Tuttavia presenta i seguenti svantaggi:

- **Comprensione**
  Quando un'applicazione monolitica raggiunge grandi dimensioni, diventa molto complicata da comprendere. Tale sistema, diventa così di difficile gestione.
- **Difficoltà di modifica**
  Risulta più difficile implementare modifiche ad applicazioni ampie e complesse a causa dell'accoppiamento estremamente stretto tra le varie funzionalità. Inoltre, qualsiasi modifica al codice influisce sull'intero sistema, dovendo quindi essere accuratamente progettata e implementata tenendo conto delle possibili dipendenze.
- **Scalabilità**
  Non è possibile ridimensionare indipendentemente i componenti, ma solo ed esclusivamente l'intera applicazione.
- **Barriera tecnologica**
  È estremamente problematico applicare una nuova tecnologia in un'applicazione monolitica poiché di conseguenza l'intera applicazione dovrebbe essere progettata e implementata nuovamente.

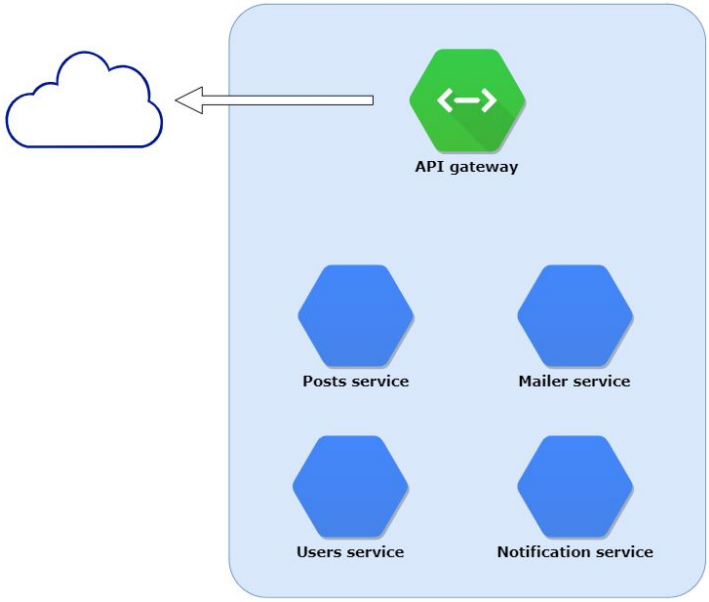


Figura 3.1: Rappresentazione grafica di un'architettura a monolite

3.2 Architettura a microservizi

La costruzione di un sistema a microservizi è uno stile architetturale che prevede la realizzazione di un software costituito da tanti servizi indipendenti e di piccole dimensioni. Per precisione, si definisce un servizio qualsiasi componente software fornito da un endpoint accessibile via rete. I servizi sono realizzati in modo tale da espletare una singola funzionalità nella maniera più approfondita ed esaustiva possibile. Inoltre, sono responsabili della persistenza dei propri dati. Questo comportamento differisce dal modello tradizionale, in cui la persistenza dei dati viene gestita in maniera centralizzata. I servizi possono inoltre comunicare tra loro tramite l'utilizzo di API ben definite.

Caratteristica fondamentale della suddetta architettura è lo scarso accoppiamento (loose coupling) presente tra i vari servizi. Esso deriva da due fattori estremamente importanti:

1. qualsiasi interazione richiesta a un servizio avviene attraverso l'API messa a disposizione dal componente stesso, consentendogli così di incapsulare i dettagli di implementazione

2. qualsiasi cambiamento può essere effettuato all'interno di un servizio senza che esso influisca sul suo utilizzatore

Queste caratteristiche dei microservizi consentono a ciascuno di essi di essere aggiornati, distribuiti e ridimensionati indipendentemente da tutti gli altri, così da poter rispondere prontamente a possibili necessità che potrebbero occorrere.

Nella successiva Figura 6.2 è possibile visualizzare la ben nota architettura a microservizi, in cui ognuno di essi è in esecuzione su un singolo nodo distribuito in rete. Come precedentemente dettagliato le comunicazioni avvengono attraverso l'utilizzo di API. In questo caso, non è possibile considerare la latenza di rete come trascurabile, in quanto i servizi potrebbero essere dislocati su endpoint differenti e quindi dover comunicare attraverso la rete. Sebbene possa comportare alcune difficoltà di interazione e di design, è proprio questa distribuzione dei servizi in rete che consente una maggiore scalabilità, resilienza e tolleranza ai guasti.

Vantaggi dell'architettura a microservizi

- **Disaccoppiamento**
  Ciascun servizio in questo stile architetturale può essere sviluppato, distribuito, eseguito e ridimensionato senza influenzare il funzionamento degli altri componenti. I servizi non necessitano quindi di condividere alcun codice o implementazione con gli altri. Qualsiasi forma di interazione avviene attraverso API ben definite.

- **Specializzazione**
  Ciascun servizio si concentra sulla risoluzione di un problema specifico. Se,

nel tempo, gli sviluppatori dovessero aggiungere del codice aggiuntivo a un servizio rendendolo più complesso, il servizio potrebbe essere scomposto in servizi più piccoli.

- **Agilità**
  Il team di sviluppo agendo in un contesto ridotto e ben delineato può lavorare più rapidamente. Risulta infatti più semplice implementare e sperimentare nuove funzionalità, potendo ripristinare la configurazione precedente nel caso qualcosa vada storto. Questa caratteristica consente di accelerare il time-to-market di nuove funzionalità, riducendo così i tempi del ciclo di sviluppo.

- **Scalabilità e flessibilità**
  I microservizi consentono di scalare ciascun componente in maniera totalmente indipendente dagli altri, consentendo così un adattamento specifico alle possibili esigenze che l'applicazione potrebbe richiedere. Ciò permette agli sviluppatori di ridimensionare facilmente le capacità computazionali dell'infrastruttura nelle modalità più adeguate alle circostanze.

- **Libertà tecnologica**
  Le architetture basate su microservizi non obbligano un'unica tecnologia per l'intera applicazione. Gli sviluppatori possono scegliere differenti approcci tecnologici (es. framework) e implementativi (es. linguaggio di sviluppo) per ogni microservizio. In questo modo è possibile scegliere gli strumenti migliori per risolvere i problemi specifici che si possono riscontrare.

- **Riusabilità**
  Dividere il software in moduli circoscritti e ben definiti permette ai team di riutilizzare stessi servizi per diversi scopi. Un servizio che realizza una specifica funzionalità può essere utilizzato come blocco costruttivo per un'altra. Ciò permette agli sviluppatori di implementare sistemi con funzionalità simili senza dover necessariamente riscrivere il codice da zero.

- **Resilienza**
  Lo scarso accoppiamento dei servizi aumenta la resilienza di un'applicazione in caso di errori. In un'architettura monolitica un errore di un singolo componente comporterebbe ripercussioni sull'intera applicazione. Con i microservizi il sistema può gestire completamente gli errori che possono avvenire in un servizio isolando solamente la funzionalità interessata, senza che ciò intacchi l'intera applicazione.

Svantaggi dell'architettura a microservizi

- **Complessità extra**
  Un'applicazione costituita da microservizi richiede una maggiore distribuzione in rete dei suoi componenti rispetto all'applicazione monolitica equi-

valente. Il singolo servizio è più semplice da sviluppare, tuttavia l'intero sistema nella sua totalità diventa più complesso.

- **Difficoltà di testing**
  L'implementazione di test per servizi distribuiti (sebbene di piccole dimensioni) richiede un approccio totalmente diverso rispetto a quanto avviene per i software tradizionali, soprattutto a causa delle molteplici dipendenze e interazioni che potrebbero essere presenti tra i vari servizi.

- **Mancanza di governance**
  L'approccio decentralizzato ottenuto dalla seguente architettura presenta alcuni vantaggi, tuttavia può anche essere causa di problemi. Difatti, potrebbero essere stati utilizzati così tanti linguaggi e framework nell'implementazione dei vari servizi che poi l'intera applicazione diventa di difficile gestione. Può essere pertanto utile applicare alcuni standard a livello di progetto, senza limitare eccessivamente la flessibilità dei team.

- **Integrità dei dati**
  In un sistema in cui ogni microservizio è responsabile della persistenza dei proprio dati, la coerenza di essi nell'intera applicazione può risultare un problema.

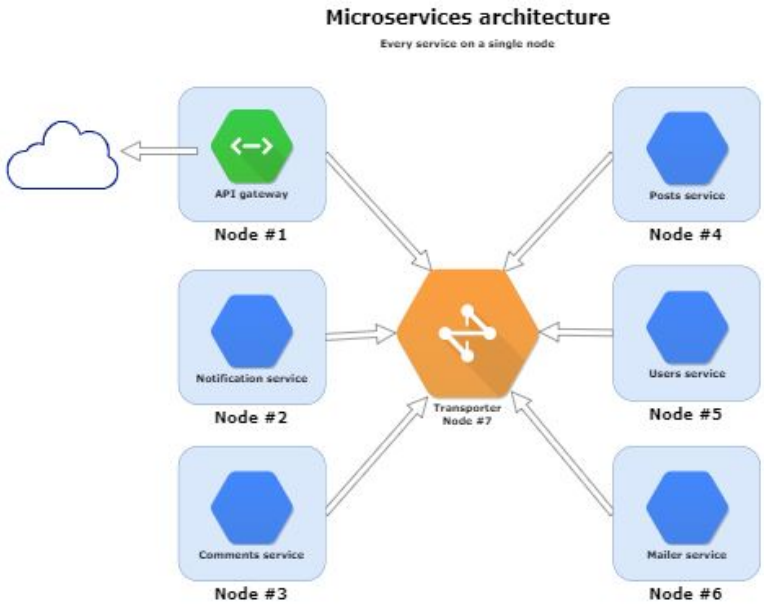


Figura 3.2: Rappresentazione grafica di un'architettura a microservizi

Capitolo 4

Design

Nel seguente capitolo viene illustrato e dettagliatamente analizzato il processo di reingegnerizzazione attuato per far fronte alle necessità riscontrate dal committente. La descrizione è suddivisa nelle tre principali aree di progettazione di un applicazione software: struttura, comportamento e interazione.

4.1 Struttura

Come nel Capitolo 2 annunciato, il team di sviluppo ha deciso di effettuare la trasformazione di un sistema monolitico in un'applicazione a microservizi. Per fare ciò, durante la fase di progettazione e design si è quindi deciso di suddividere le funzionalità attualmente implementate attraverso una semplice architettura client-server in diversi microservizi indipendenti.

4.1.1 Decomposizione del monolite

Lo sviluppo da zero di applicazioni a microservizi è particolarmente raro; è molto più usuale infatti che le organizzazioni tendano ad adottare una soluzione a microservizi partendo da una loro applicazione monolitica. Per tale motivo sono stati realizzati specifici pattern che consentano di risolvere efficientemente ed efficacemente il problema di migrazione di un'applicazione monolitica legacy (obsoleta) a un'architettura di microservizi. Il pattern da cui il team di sviluppo ha preso ispirazione è denominato *Strangler application*, il quale prevede la modernizzazione dell'architettura attraverso lo sviluppo incrementale (vedi Figura 4.1) di un nuovo sistema (detto strangler) attorno a quello di partenza (detto legacy). Tale approccio prevede un processo ben delineato per l'individuazione e l'estrazione dei servizi dall'applicativo monolitico. Chris Richardson, ideatore del pattern, suggerisce infatti di scomporre il sistema in microservizi mediante una decomposizione basata sulle business capability dell'applicazione: ovvero incentrandosi sui concetti o funzionalità che possono generare business value per l'azienda. Seguendo tale suggerimento, successivamente ad una approfondi-

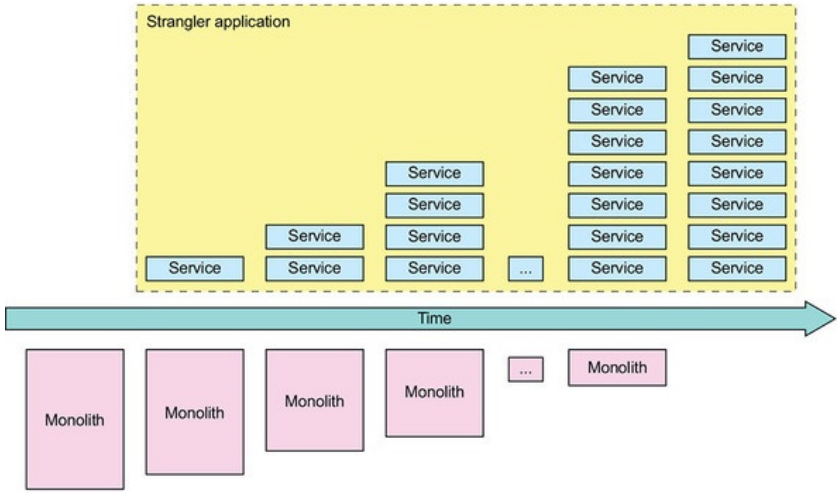


Figura 4.1: Conversione di un applicativo attravero lo Strangler application pattern

ta analisi del software, il team di sviluppo ha identificato le seguenti funzionalità principali:

- Registrazione di un nuovo utente;
- Login di un utente precedentemente registrato;
- Logout di un utente precedentemente registrato;
- Hashing della password per l'adozione di misure di sicurezza;
- Creazione di un Party per la condivisone di contenuti multimediali con altri utenti;
- Invito di utenti alla modalità Party;
- Riproduzione sincronizzata del contenuto multimediale per tutti coloro che partecipano al Party;
- Chat real-time per tutti coloro che partecipano al Party;
- Gestione del concetto di amicizia tra utenti.
- Invio di richieste di amicizia.
- Gestione di elementi di gamification.

Tali funzionalità sono successivamente state suddivise in base alla loro correlazione in vari microservizi.

- **Authentication Service**
  - Registrazione di un nuovo utente
  - Login di un utente precedentemente registrato.
  - Logout di un utente precedentemente registrato.
- **Hash Password Service**
  - Hashing della password per l'adozione di misure di sicurezza.
- **Friend Service**
  - Gestione del concetto di amicizia tra utenti.
  - Invio di richieste di amicizia.
- **Real-time Communication Service**
  - Creazione di un Party per la condivisone di contenuti multimediali con altri utenti;
  - Invito di utenti alla modalità Party;
  - Riproduzione sincronizzata del contenuto multimediale per tutti coloro che partecipano al Party;
  - Chat real-time per tutti coloro che partecipano al Party;
- **Badge Service**
  - Gestione di elementi di gamification.

4.1.2 Progettazione dell'architettura

Identificati i microservizi e le relative interazioni ci si è focalizzati sull'identificazione dell'architettura più vantaggiosa per le esigenze del cliente. Grazie alle funzionalità messe a disposizione dal framework Moleculer (approfondito nella sezione 6.1) è stato possibile realizzare un "architettura mista", ovvero un architettura che potesse combinare i vantaggi delle applicazioni monolitiche e dei microservizi.

Con la riprogettazione del sistema si è quindi pianificato di trasformare l'attuale architettura monolitica:

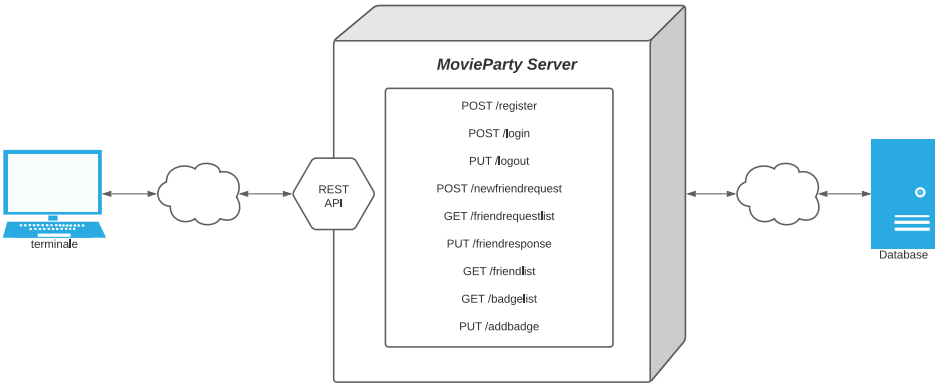


Figura 4.2: Precedente architettura monolitica

nella seguente architettura a microservizi:

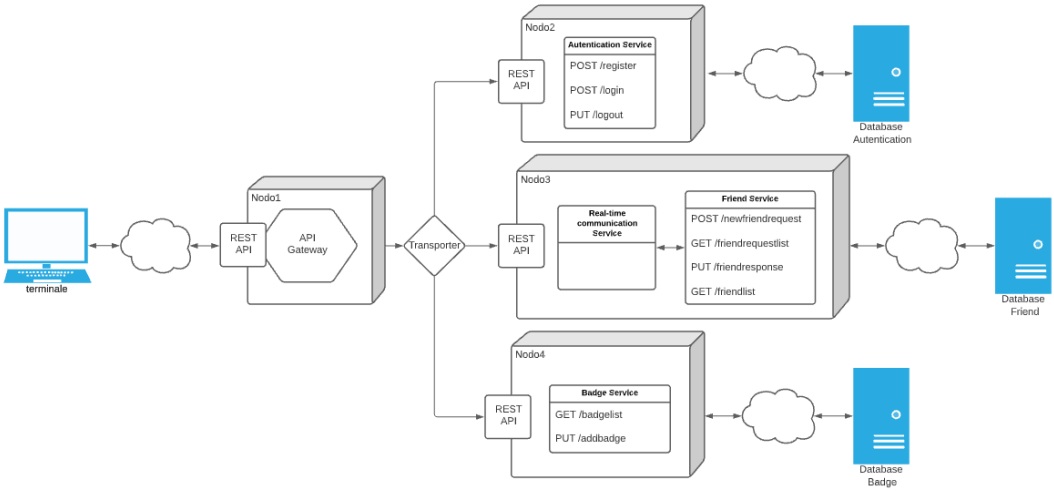


Figura 4.3: Successiva architettura a microservizi

Come rappresentato dalla Figura 4.3 la soluzione prevede due tipologie di componenti:

Servizi

Con servizio si intende un modulo contenente un insieme di funzionalità che consentono la gestione di una parte più o meno complessa dell'applicazione.

Tipicamente le funzionalità presenti all'interno di un servizio sono correlate in modo tale da poter specificare un determinato dominio applicativo. Ogni servizio è isolato e autonomo. Ciò significa che nel caso in cui un servizio abbia un malfunzionamento che comporti un disservizio di una parte o della totalità dello stesso, i restanti servizi rimarrebbero ancora attivi, potendo continuare così a rispondere alle varie richieste.
Come è possibile visualizzare nella figura 4.3, mediante l'utilizzo dei Container Docker e la rispettiva integrazione con il framework Moleculer è stato possibile posizionare più servizi concorrenti sullo stesso nodo. Tale tecnica, sebbene non convenzionale quando si tratta di microservizi, ha permesso di ridurre la latenza di rete delle interazioni (in quanto chiamate locali) che avvengono tra i servizi che risiedono nello stesso nodo. Cercando sempre di mantenere il più possibile il concetto fondamentale dei microservizi, ovvero il disaccoppiamento, solamente i servizi strettamente correlati, tra i quali avvengono un numerose interazioni sono state inserite all'intero dello stesso nodo.

Database

Come da prassi quando si tratta di microservizi, si è fatto uso del pattern *Database per Service*, in cui ogni servizio è responsabile della persistenza dei propri dati. Ciò consente ad ogni componente di poter utilizzare il tipo di database più adatto alle proprie esigenze e assicura ai microservizi di potersi evolvere indipendentemente gli uni dagli altri. Nel progetto si è deciso di continuare con l'utilizzo del DBMS (Database Management System) non relazionale MongoDB. Questo perché il suo orientamento ai documenti in formato JSON, semplifica notevolmente agli sviluppatori la gestione delle informazioni potendo utilizzare direttamente tali dati all'interno dell'applicazione. In particolare si è scelto di utilizzare MongoDB Cloud, in quanto l'esternalizzazione a una azienda altamente specializzata garantisce una disponibilità e scalabilità dei servizi best-in-class, standard di privacy e sicurezza estremamente esigenti e funzionalità considerate best-practices del settore.

4.1.3 Scalabilità e resilienza ai guasti

Il team di sviluppo in seguito all'accurata integrazione della tecnologia Docker con il framework Moleculer e la sua facilità di utilizzo ha optato per tale soluzione per gestire la scalabilità e resilienza ai guasti dell'applicazione. Come rappresentato nella Figura 4.4, attraverso la combinazione di queste tecnologie è stato possibile istanziare più nodi contenenti i medesimi servizi. Ciò consente:

- **Scalabilità**

  Nel caso in cui un servizio diventi sovraccaricato, è possibile istanziare nuovi nodi con lo stesso medesimo servizio per aumentare le capacità di calcolo dell'applicazione. Il bilanciamento del carico fornito da Moleculer partizionerà poi le richieste tra i diversi nodi, consentendo così di suddividere il lavoro di un singolo componente su più componenti distribuiti in

rete. Tali nodi potranno poi essere facilmente rimossi quando il carico di rete tornerà a livelli standard.

- **Bilanciamento del carico**

  Moleculer offre anche la possibilità di usufruire di diverse strategie di bilanciamento del carico, tutte integrate al suo interno. Nello specifico, se uno stesso servizio è in esecuzione su più nodi (come ad esempio l'Authentication Service), il *ServiceRegistry* utilizza la strategia specificata dallo sviluppatore per individuare il nodo alla quale dirigere la richiesta.

- **Maggiore resilienza ai guasti**

  Nel caso in cui un servizio abbia un guasto è possibile istanziare un nuovo nodo perfettamente funzionante con i medesimi servizi oppure ridirigere completamente le richieste su componenti di backup precedentemente predisposti.

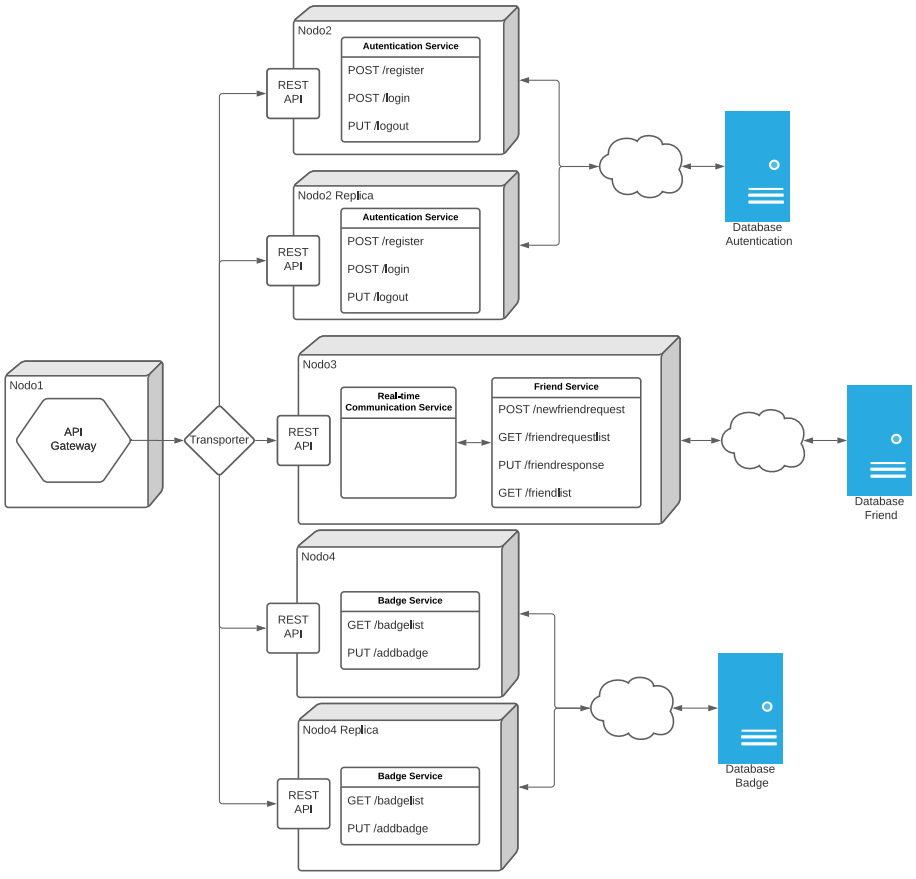


Figura 4.4: Architettura a microservizi con nodi "replica"

4.2 Comportamento

In questa sezione viene analizzato più nel dettaglio il comportamento dei vari microservizi presenti all'interno dell'applicazione. Sostanzialmente il compito di ognuno di essi consiste nella validazione dei parametri passati all'interno della richiesta, per poter così svolgere un'azione che andrà a modificare lo stato del dominio contenuto in un database. Per ogni servizio, nel caso i parametri contenuti nella richiesta non siano corretti, sono stati predisposti opportuni errori che consentano al client di individuare l'errore.

- **API Gateway**
  L'API Gateway non è altro che un usuale servizio Moleculer che consente di esporre agli utenti finali i vari microservizi implementati all'interno dell'applicativo, in particolare, tali servizi vengono pubblicati come RESTful APIs. Si può quindi riassumere che il compito del Gateway a cui si sta facendo riferimento è quello di intercettare le richieste provenienti dai vari client, ridirigendole ai servizi opportuni in attesa di una risposta che verrà successivamente inoltrata al mittente. Tutto ciò consente all'applicativo di possedere un unico punto di ingresso per tutti i client, ottenendo così il vantaggio di isolare i client dal modo in cui l'applicazione è partizionata in microservizi e dal problema di determinare le posizioni delle istanze del servizio. Le app client quindi non devono più sapere in che modo le diverse aree dell'applicazione vengono scomposte in microservizi, e quando si effettua il refactoring o manutenzione di un microservizio interno, non è più necessario modificare anche l'interazione del client con quel microservizio. Un ulteriore vantaggio nell'utilizzo di un API Gateway riguarda la sicurezza. Infatti, senza un suo utilizzo, tutti i microservizi sarebbero esposti all'esterno, rendendo più ampia la superficie di attacco.

- **Authentication Service**
  Modulo del sistema che si occupa dell'autenticazione degli utenti all'interno dell'applicazione. Attraverso le APIs fornite dal seguente microservizio è possibile registrare un nuovo fruitore all'interno della piattaforma oppure effettuare il login, il logout o la cancellazione di un utente precedentemente autenticato. Alla registrazione di un nuovo utente viene inoltre utilizzato un Hash Password Service, che consente di adottare misure di sicurezze idonee per un archiviazione sicura della password. La registrazione e la cancellazione avvengono (dopo opportuni controlli per l'integrità dei dati) mediante l'aggiunta o la rimozione di risorse all'interno del database specifico per la memorizzazione degli utenti. Per quanto riguarda il Login e il Logout invece la base di dati non viene modificata, solamente interrogata per verificare che i parametri (come ad esempio username e password) siano corretti.

- **Friend Service**
  È il servizio che rappresenta il concetto di amicizia all'interno del dominio applicativo. In particolare, questo specifico servizio consente l'invio

di richieste di amicizia tra utenti presenti all'interno della piattaforma, di consultare tutte le richieste di amicizia ancora pendenti, rispondere a tali richieste oppure visualizzare la lista di utenti attualmente amici e nel caso di rimuoverne alcuni. Come accade per ogni servizio presente all'interno del sistema tutto ciò è gestito mediante la memorizzazione delle informazioni necessarie all'interno di un apposito database atto a reificare l'astrazione di amicizia. In aggiunta però, viene utilizzo il Real-time Communication Service per intraprendere uno scambio di informazioni in tempo reale fra gli utenti intenzionati a iniziare una nuova amicizia. Attraverso questa tipologia di interazione vengono infatti inoltrate ai vari destinatari le richieste di amicizia e successivamente le risposte ai mittenti. È bene precisare che i messaggi real-time non vengono mai spediti da Client a Client. Sarà sempre il microservizio in seguito all'arrivo di comunicazioni tramite RESTful APIs che specificano l'intenzione di perseguire una nuova richiesta di amicizia o una risposta a una di esse a memorizzare le informazioni all'interno del database per poi contattare tramite messaggistica in tempo reale il Client interessato. Sebbene i database dei vari microservizi siano indipendenti, per mantenere lo stato dell'applicazione sempre aggiornato e integro su tutte le basi di dati a disposizione vengono utilizzati gli eventi, una funzionalità fornita dal framework Moleculer. Infatti, in seguito alla registrazione di un nuovo utente sulla piattaforma che avviene sull'Authentication Service, non solo verrà aggiunta una risorsa all'interno del database specifico di quel servizio, ma attraverso l'utilizzo degli eventi verranno avvisati tutti gli altri componenti in modo tale che loro possano agire di conseguenza. Di fatti, sia il Friend Service che il Badge Service creerano all'interno dei loro database di riferimento le risorse necessarie per il funzionamento corretto dell'intera applicazione.

- **Badge Service**
  È il servizio ideato per la gestione degli elementi di gamification presenti all'interno della piattaforma. Mediante le APIs pubbliche del servizio è possibile infatti ottenere la lista dei badge di un utente specifico oppure di aggiornarla nel caso in cui vengano raggiunti nuovi obiettivi. Anche in questo caso tutto viene realizzato mediante l'utilizzo di un database in cui vengono memorizzate le informazioni necessarie per gestire correttamente gli elementi di gamification.

- **Real-time communication Service**
  Questo servizio è stato realizzato per contenere al suo interno un server WebSocket, consentendo così di intraprendere ove necessario una comunicazione real-time tramite socket all'interno del sistema. Tali comunicazioni non avvengono tra microservizi, bensì sono utilizzate per comunicare direttamente con i Client. Il Real-time Communication Service viene interpellato:

  – da altri microservizi quando è necessario informare specifici Client di un cambiamento dello stato del sistema avvenuto in seguito ad un

particolare avvenimento. Per esempio, successivamente ad una richiesta effettuata tramite API al Friend Service per una nuova richiesta di amicizia, tale microservizio utilizzerà il Real-time Communication Service per comunicarla al destinatario. È possibile visualizzare tutto ciò nella seguente rappresentazione:

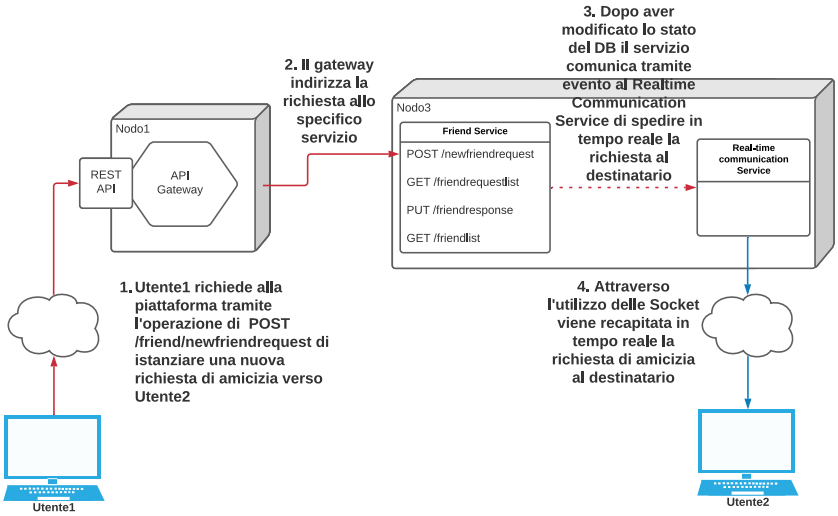


Figura 4.5: Interazione tra microservizio e WebSocket

– direttamente dai Client tramite l'utilizzo delle Socket. Questo risulta utile, a differenza del caso precedente che viene utilizzato per comunicare un cambiamento dello stato del sistema, per coordinare in tempo reale operazioni che riguardano più utenti. Ad esempio, per spedire un messaggio all'interno della chat oppure fermare momentaneamente la riproduzione del contenuto multimediale. Tali informazioni vengono comunicate dal Client al Real-time communication Service, che mediante l'utilizzo di "stanze" fornite dalla libreria Socket stessa riesce a raggiungere attraverso una tipologia di interazione broadcast solamente gli specifici utenti partecipanti al medesimo Party del mittente. Nella figura sottostante è possibile visualizzare come un Client possa comunicare direttamente con il server WebSocket attraverso l'utilizzo di specifici eventi di cui il server è costantemente in ascolto.

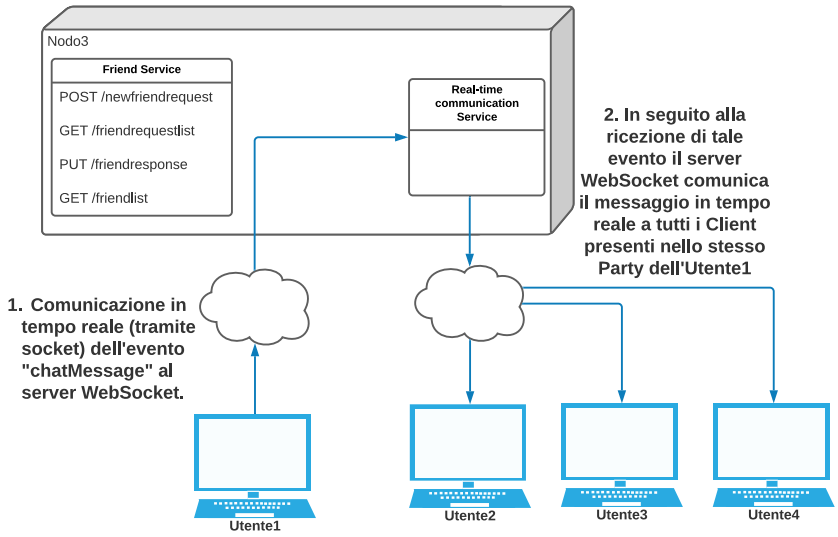


Figura 4.6: Interazione diretta tra il Client e il server WebSocket

Attraverso l'utilizzo di diverse tipologie di comunicazione fornite dalla libreria Socket.Io come la one-to-one o la broadcast questo microservizio è in grado di comunicare inviti, gestire la chat oppure inviare messaggi per la sincronizzazione dei vari utenti partecipanti alla modalità Party presente all'interno della piattaforma, tutto perennemente in tempo reale.

4.3 Interazione

All'interno dell'applicazione è possibile individuare le seguenti tipologie di interazione:

4.3.1 Rest API per la comunicazione client-server

Il client comunica con i vari servizi attraverso l'utilizzo di RESTful API. Ciò comporta che le risorse (identificate tramite URI) esposte dal servizio web siano manipolabili mediante un insieme ben definito di operazioni:

- POST: per la creazione di una nuova risorsa;
- GET: per l'ottenimento dello stato corrente di una determinata risorsa;
- PUT: per l'aggiornamento di una risorsa;
- DELETE: per la cancellazione di una risorsa.

Come protocollo per la rappresentazione delle informazioni è stato definito il formato JSON. Inoltre, secondo quanto richiesto per l'implementazione di un

servizio RESTful l'interazione avviene in modalità *stateless*. Questo richiede l'autocontenimento di ogni messaggio: ovvero che tutte le informazioni necessarie al web-service per eseguire l'operazione debbano essere contenute all'interno del messaggio in quanto nessuno stato di sessione viene mantenuto. Nel nostro caso i servizi da noi ideati espongono le seguenti API (i diagrammi e le relative descrizioni sono state posizionate al termine della relazione per consentire una maggiore leggibilità della stessa):

- **Authentication Service**
  - POST /register: Dettagli in Figura 10.2
  - POST /login: Dettagli in Figura 10.1
  - PUT /logout: Dettagli in Figura 10.3
  - DELETE /delete: Dettagli in Figura 10.4
- **Friend Service**
  - POST /newfriendrequest: Dettagli in Figura 10.5
  - GET /friendrequestlist: Dettagli in Figura 10.6
  - PUT /friendresponse: Dettagli in Figura 10.7
  - GET /friendlist: Dettagli in Figura 10.8
  - DELETE /deletefriendrequest: Dettagli in figura 10.9
  - DELETE /deleteuserfriend: Dettagli in Figura 10.10
- **Badge Service**
  - GET /getbadgelist: Dettagli in Figura 10.11
  - PUT /updatebadgelist: Dettagli in Figura 10.12

4.3.2 RPC per comunicazioni tra servizi

Un servizio può comunicare, sempre mediante i metodi pubblici esposti, con altri servizi. In questo caso l'interazione avviene attraverso delle Asynchronous Remote Procedure Call (RPC). In seguito alla creazione di una RPC, il ServiceBroker del servizio chiamante individuerà il servizio (e il nodo nella quale è in esecuzione) che possiede l'operazione specificata e la invocherà. Il servizio destinatario potrà risiedere all'interno dello stesso nodo del mittente (chiamata locale) oppure in un servizio distribuito in rete (chiamata remota). Nel caso la chiamata avvenga localmente il ServiceBroker inoltrerà lui stesso la richiesta al servizio; in caso contrario si affiderà al Transporter che avrà il compito di ridirigere correttamente la chiamata al destinatario.

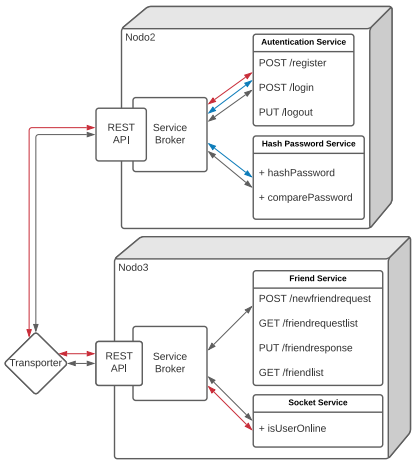


Figura 4.7: Rappresentazione di una chiamata locale (freccia blu) e una chiamata remota (freccia rossa)

4.3.3 Moleculer Events

Ogni servizio creato mediante Moleculer è inoltre in grado di inviare eventi a servizi locali (presenti nello stesso nodo) o remoti (distribuiti in rete) tramite il suo ServiceBroker. Ancora una volta si può intuire come il *ServiceBroker* sia l'anima del seguente framework. All'interno del applicativo gli eventi sono stati utilizzati per comunicare uno specifico cambiamento di stato ai servizi interessati, in modo tale che anch'essi possano reagire alla suddetta modifica.

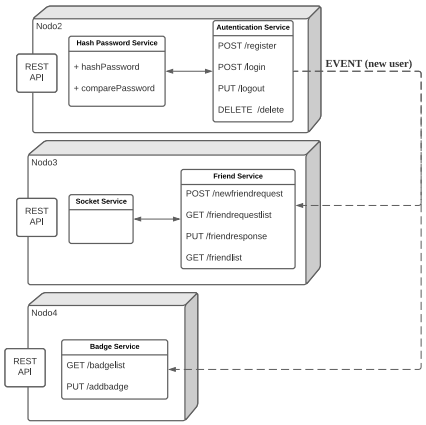


Figura 4.8: Invio di un evento in seguito a un cambiamento di stato ai servizi interessati

Nella Figura 4.8 è possibile visualizzare come l'Authentication Service in seguito alla registrazione di un nuovo utente comunichi tale cambiamento di stato anche

al Friend Service e Badge Service. Così facendo i seguenti servizi reagiranno a tale informazione istanziando tutti i campi necessari all'interno del database per un corretto funzionamento dell'applicazione.

4.3.4 Socket messages per la comunicazione real-time

Il Socket Service è un modulo creato appositamente per la gestione delle comunicazioni real-time basate su eventi tra client e server. In particolare si occupa di tutte le interazioni che devono necessariamente avvenire in tempo reale all'interno della modalità "Party" come ad esempio: l'invito di utenti alla sessione, la gestione della chat o la sincronizzazione della riproduzione multimediale. Al suo interno è quindi possibile riscontrare due tipologie di messaggi:

- **Messaggi privati**
  Messaggi destinati ad un utente specifico. In particolare sono stati utilizzati per tutte le interazioni one-to-one, alla quale quindi partecipavano solamente due specifiche entità.

- **Messaggi broadcast**
  Scambio di messaggi nella quale lo stesso contenuto viene inviato contemporaneamente a tutte le socket attualmente connesse al server. In aggiunta, attraverso l'utilizzo di un canale arbitrario chiamato "stanza" la libreria Socket.IO permette la trasmissione di messaggi solamente a un sottoinsieme di socket collegate al server. Questo consente una gestione estremamente facilitata di tutte le comunicazioni real-time della modalità Party. Il sistema infatti assegna a ogni Party una stanza, alla quale tutti i partecipanti verranno iscritti. Successivamente, quando un utente causerà un cambiamento di stato, tutte le socket appartenenti alla stanza verranno informate dell'aggiornamento, consentendo così una perfetta sincronizzazione dei dati tra tutti i Client.

Capitolo 5

Dettagli Implementativi

5.1 Async/Await Pattern

Per ottimizzare al meglio l'utilizzo delle risorse di cui si dispone, all'interno del codice è stato fatto grande uso di funzioni asincrone. Tale scelta implementativa consente infatti di massimizzare le performance dell'event-loop con la quale i servizi sono realizzati, non bloccando nemmeno momentaneamente il thread esecutore del servizio. Quanto detto è stato attuato mediante l'utilizzo del meccanismo attualmente di riferimento: *Async/Await*. L'utilizzo delle seguenti Keyword consente infatti di andare oltre agli approcci standard quali Callback e Promise, permettendo la scrittura di codice asincrono pur mantenendo una struttura di codice tipico della programmazione sincrona. Tale meccanismo è semanticamente correlato al concetto di coroutine ed è spesso implementato utilizzando tecniche simili. Principalmente consente al thread esecutore dell'event-loop di espletare altro codice in attesa del completamento di un'attività asincrona di lunga durata.

5.2 Scomposizione di Hash Password Service e Authentication Service

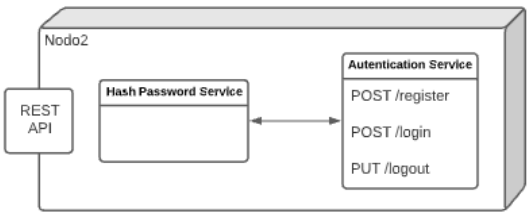


Figura 5.1: Microservizi indipendenti presenti in un unico nodo

La seguente suddivisione, sebbene non necessaria in un sistema reale, è stata ideata per motivi didattici in maniera tale da esplorare completamente le potenzialità del framework Moleculer. La struttura raffigurata precedentemente descrive pienamente il concetto di *Mixed Architecture*: ovvero la combinazione dei vantaggi dell'architettura monolitica con quelli dei microservizi. Tale architettura rende possibile il posizionamento di microservizi che tipicamente interagiscono molteplici volte tra di loro all'interno di uno stesso nodo. Questo rimuovere completamente la latenza di rete necessaria nel caso in cui i due servizi fossero stati distribuiti in rete, consentendo così di ottimizzare la velocità di interazione. Per realizzare tutto ciò non è necessaria alcuna modifica al codice, poiché il Service Broker di Moleculer tenterà sempre di invocare in primo luogo le istanze locali di un servizio, e solamente nel caso in cui non esistano si rivolgerà al distribuito.

Capitolo 6

Tecnologie utilizzate

6.1 Moleculer

Moleculer consiste in un framework per la creazione di sistemi a microservizi in *Node.js*. Moleculer supporta la creazione di sistemi: efficienti, affidabili e scalabili sostenendo il processo di sviluppo in fase di costruzione e gestione dei microservizi. I concetti principali, che saranno poi approfonditi, del framework sono i seguenti:

- **Service**: con servizio si intende un modulo JavaScript che contiene una parte più o meno complessa dell'applicazione. Ogni servizio è isolato e autonomo. Questo significa che nel caso in cui un servizio andasse offline (in seguito ad un crash ad esempio) i restanti servizi rimarrebbero ancora funzionanti potendo così continuare a rispondere alle varie richieste.
- **Node**: il concetto di nodo viene inteso come un semplice processo del S.O. eseguito in un network locale o remoto. Una singola istanza di un nodo può ospitare uno o più servizi. Questo permette di attuare strategie di ottimizzazione nel caso in cui il sistema sia distribuito. Si potrebbe infatti optare per posizionare sullo stesso nodo due servizi con una forte interoperabilità. Ciò consentirebbe una comunicazione locale e non remota, con il vantaggio quindi di una maggiore velocità di interazione.
- **Local Services**: due o più servizi che girano sullo stesso nodo. Condividono hardware e risorse mentre la comunicazione avviene utilizzando il bus locale.
- **Remote Services**: diversamente dai Local Services, i Remote Services sono servizi distribuiti in vari nodi nella rete. La comunicazione in questo caso avviene mediante l'utilizzo di un *trasporter* per la comunicazione, che farà necessariamente subire latenze al sistema dovute al networking.

- **Service Broker**: è il componente principale del framework. Si occupa della gestione e della comunicazione dei servizi sia in locale che in remoto. Ogni nodo ha necessariamente un'istanza di un Service Broker.
- **Transporter**: componente che si occupa del trasporto di messaggi attraverso la rete. È il comunication bus utilizzato dai servizi per lo scambio di messaggi, trasferendo: eventi, richieste e risposte. Questo componente è utilizzato solamente nel caso in cui i processi comunicanti non siano sullo stesso nodo, bensì distribuiti in rete (Remote Services).
- **Gateway**: componente che espone i servizi all'end-user. Sostanzialmente si tratta di un servizio che gestisce le richieste in entrata indirizzandole verso il componente appropriato. Successivamente fornisce la risposta al mittente.

Di seguito viene mostrato un esempio di una semplice architettura a microservizi costruita utilizzando Moleculer.

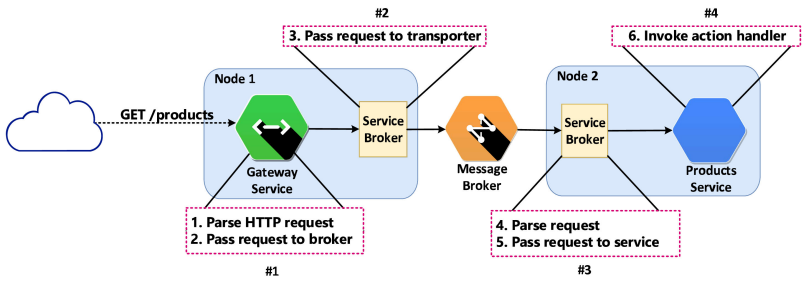


Figura 6.1: Architettura di un sistema a microservizi utilizzando Moleculer

Come mostrato in Figura 6.1, da un punto di vista architetturale si hanno due servizi indipendenti: il servizio *gateway* e il servizio *products*. Il primo servizio è adibito alla ricezione delle richieste degli utenti, che trasmette poi al products service. In questo esempio viene mostrata la comunicazione tra remote services, facendo eseguire i servizi in due nodi separati, il gateway sul nodo 1 e il products sul nodo 2. Siccome i servizi girano su nodi remoti necessitano di un transporter e quindi di un message broker per la comunicazione tra servizi. Nel momento in cui viene ricevuta una richiesta (`GET/products` in figura) questa viene intercettata dal server HTTP nel nodo 1. Il server passa semplicemente la richiesta al *gateway service* che si occupa di processare la richiesta e mapparla in un azione presente nel servizio attivo nel nodo 2. A questo punto il broker riceve la richiesta e viene controllato se il servizio è presente in locale o in remoto; se presente in locale, la richiesta viene passata attraverso il communication bus al servizio. Nel caso in cui il servizio sia remoto, il broker utilizza il modulo di trasporto per spedire la richiesta. Il transporter ha il compito di prendere la richiesta e spedirla al broker service del nodo 2 dov'è presente il products service.

Ricevuta la richiesta il broker del nodo 2 si incarica di eseguire il parsing e di passare la richiesta al products service presente nel nodo. Infine il products service invoca l'azione richiesta e ritorna la risposta, che seguirà il percorso inverso della richiesta.

6.1.1 Broker

Il ServiceBroker è il componente principale del framework Moleculer. Ha il compito di gestire i servizi, chiamare le azioni richieste, emettere eventi e comunicare con nodi remoti. Ogni nodo necessita di un ServiceBroker, che è composto da diversi moduli come mostrato in Figura 6.2.

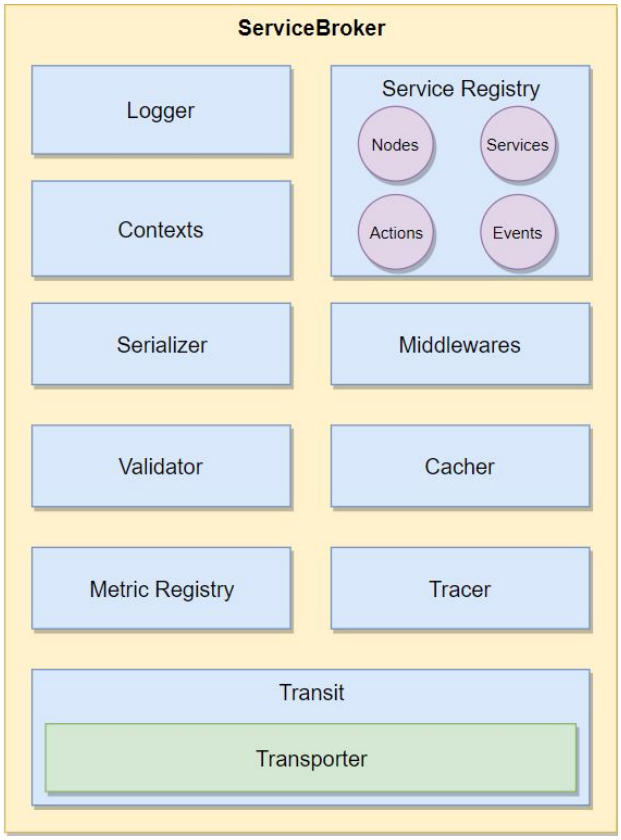


Figura 6.2: ServiceBroker e i suoi moduli principali

Broker lifecycle

Starting logic

Inizialmente il broker ha il compito di stabilire una connessione con il transporter, per poi avviare tutti i servizi mediante l'invocazione dei corrispettivi *started handler*. Una volta avviati correttamente tutti i servizi, il broker pubblica l'elenco dei vari servizi presenti sui nodi remoti. Solamente una volta che tutti i servizi locali sono stati inizializzati e avviati correttamente ai nodi remoti possono essere inviate richieste.
È importante evitare *deadlocks* in questa fase di inizializzazione. Ciò può accadere quando due servizi dipendono l'uno con l'altro. Tipicamente per evitare questa dipendenza circolare si utilizza il meccanismo `waitForServices` offerto direttamente dal framework. Nella Figura 6.3 viene mostrato il lifecylce della fase di start di un broker.

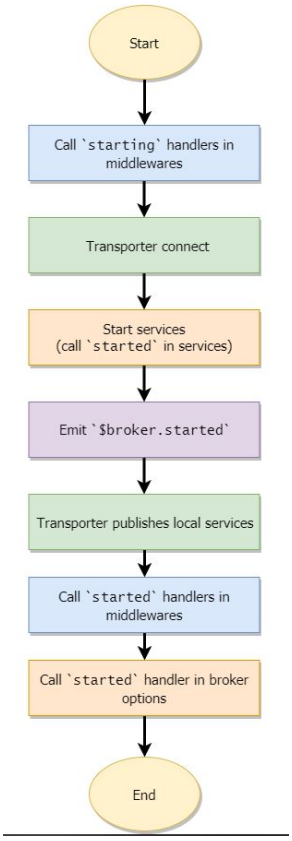


Figura 6.3: Broker start lifecycle logic

Stopping logic

Quando viene richiamata la `broker.stop` o il processo viene interrotto, il broker pubblica un elenco di servizi vuoto sui nodi remoti, in modo tale che le richieste verranno instradate verso altri nodi e non verso il nodo corrente che sta venendo interrotto. Successivamente, il broker inizia a interrompere tutti i servizi locali ed infine, il transporter si disconnette e il processo viene chiuso. Nella Figura 6.4 viene mostrato il lifecylce della fase di stop di un broker.

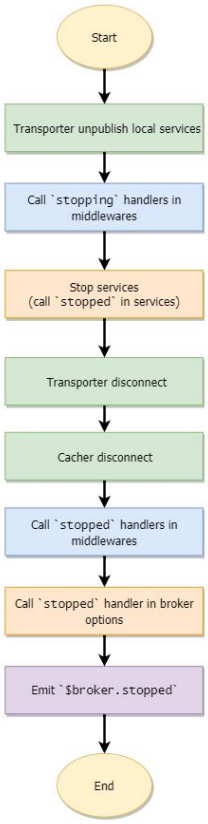


Figura 6.4: Broker start lifecycle logic

6.1.2 Services

Il componente Service di Moleculer rappresenta un microservizio in cui è possibile definire azioni e eventi. I servizi sono gestiti dal Service Registry che si occupa di caricare i servizi all'interno del nodo specificato.

Service lifecycle

Quando un servizio viene avviato o fermato vengono richiamati degli handler sincroni facenti parte del ciclo di vita del servizio. Di seguito vengono elencati gli handler disponibili:

- **Created event handler**: questo handler viene innescato appena l'istanza del servizio viene creata.
- **Started event handler**: questo handler viene innescato in seguito alla chiamata della `broker.start`.
- **Stopped event handler**: questo gestore viene innescato quando viene chiamata la `broker.stop` e il broker avvia l'arresto di tutti i servizi locali. Può essere utilizzato per chiudere le connessioni al database o le istanze delle socket.

Schema

Per la creazione di un servizio è necessario definire uno schema. Lo schema ha diverse componenti principali: name, version, mixins, settings, actions, methods, events.

- *Name*: è una proprietà obbligatoria, quindi deve essere definita. Rappresenta la prima parte della signature di un'azione, nel momento in cui viene chiamata.
- *Version*: è una proprietà opzionale ed è utilizzata nel caso in cui si vogliano eseguire multiple versioni dello stesso servizio.
- *Settings*: proprietà che di fatto rappresenta uno store statico, in cui è possibile memorizzare tutte le impostazioni del servizio.
- *Mixins*: i mixin sono un meccanismo flessibile per gestire funzionalità riutilizzabili per i servizi Moleculer. Il costruttore del servizio unisce questi mixin con lo schema corrente. Quando un servizio utilizza i mixin, tutte le proprietà presenti nel mixin verranno "mixate" nel servizio corrente, ereditando tutte le proprietà e funzionalità del modulo mixato.
- *Actions*: le azioni sono i metodi pubblici e richiamabili del servizio. È possibile richiamarle dal servizio stesso o da altri componenti attraverso la `broker.call`. Infatti anche all'interno delle azioni stesse è possibile chiamare altre azioni innestate in altri servizi. Per invocare una specifica azione è necessario definire prima il nome del servizio poi il nome dell'azione (es. `broker.call("serviceName.action")`).
- *Events*: è possibile definire degli handler da eseguire in seguito all'avvenimento di un evento.
- *Methods*: all'interno del servizio è possibile creare delle funzioni private. Tali funzioni non possono essere invocate da altri servizi, bensì sono riservate al servizio stesso.

Dependencies

Questo costrutto risulta utile se il servizio dipende da altri servizi. Permette infatti di attendere l'avviamento dei servizi dipendenti prima di inizializzare se stesso. Come alternativa alle dipendenze è anche possibile utilizzare il metodo `waitForServices` del ServiceBroker, che restituisce una promise che verrà risolta quando tutti i servizi richiesti saranno disponibili e avviati.

Metadata

Un servizio contiene anche la proprietà *metadata* dove è possibile memorizzare qualsiasi informazione inerente al servizio.

Hot Reloading Services

Moleculer è caratterizzato da una funzionalità built-in per l'hot reloading di un servizio. Il meccanismo dell'hot reloading controlla i file dei servizi e delle loro dipendenze e ogniqualvolta venga rilevata una modifica al file, viene tenuta traccia dei servizi che dipendono da esso e li riavvierà. Questo meccanismo risulta utile durante le fasi di sviluppo in quanto vengono ricaricati solo i servizi modificati, senza dover quindi aspettare il riavvio totale del sistema, riducendo i tempi di attesa e aumentando i tempi di produttività.

6.1.3 Actions

Le actions sono i metodi pubblici e richiamabili di un servizio. Risulta simile ad una richiesta HTTP in quanto ha parametri di richiesta e ritorna una risposta. Se vi sono presenti più istanze di un servizio, il broker si impegnerà a bilanciare le richieste tra i servizi attuando un meccanismo di load balancing (che verrà approfondito in seguito)

Call Action services

Per richiamare un metodo pubblico di un servizio si usa la `broker.call`. Il broker fa scanning dei servizi o dei nodi che hanno l'azione richiesta e richiama. La funzione ritorna una `Promise`, e come si vede dalla Figura 6.5 la struttura della `broker.call` presenta diversi parametri in input.

```
const res = await broker.call(actionName, params, opts);
```

Figura 6.5: Esempio di chiamata di una `broker.call`

- *actionName*: è una stringa con annotazione dot-separated, dove la prima parte della stringa rappresenta il nome del servizio mentre la seconda parte rappresenta il nome dell'azione. Se quindi si ha un servizio di nome "service" e un'azione di nome "action" la stringa finale risulterà "service.action".

- *params*: è un oggetto che viene passato all'azione come parte del `context` e il servizio può accedervi tramite `ctx.params`. Non è un campo obbligatorio e in caso in cui non venga specificato nulla viene passato un oggetto vuoto.
- *opts*: anche questo campo è opzionale ed è costituito da alcuni parametri che andranno a sovrascrivere e impostare alcune opzioni dei parametri di richiesta (ad esempio timeout, retryCount, ecc).

Multiple calls e Streaming

Come visto precedentemente la `broker.call` permette di richiamare un'azione presente in un servizio, tuttavia risulta utile poter avere un costrutto che permetta di richiamare multiple azione allo stesso tempo, ciò è possibile mediante il costrutto `broker.mcall` o `ctx.mcall`. Inotre Moleculer supporta gli stream di Nodejs come parametri di una richiesta o come risposta. Nel momento in cui `params` è una stream non è possibile aggiungervi altre variabili o oggetti, in questo caso per inviare informazioni aggiuntive sarà necessario utilizzare la proprietà `meta`.

Action visibility

L'azione ha una proprietà che consente di gestire la visibilità e la richiamabilità della stessa dai vari servizi. Di default il valore predefinito è null (corrispondente a published), ma vi sono diverse possibilità:

- *published* o *null*: azione pubblica che può essere chiamata sia localmente che da remoto; può essere pubblicato tramite API Gateway.
- *public*: azione pubblica, può essere chiamata localmente e da remoto ma non pubblicata tramite API Gateway.
- *protected*: può essere richiamata solamente in locale (da un servizio locale).
- *private*: può essere chiamata solamente internamente dal servizio stesso via `this.actions.privateAction()`.

6.1.4 Action hooks

Gli action hook sono funzioni middleware collegabili e riutilizzabili che possono essere registrate prima, dopo o sugli errori delle azioni del servizio. Vi sono presenti tre tipi di action hook: before, after ed error hooks.

Before hooks

Il before hook riceve il ctx, può manipolare `ctx.params`, `ctx.meta` o aggiungere variabili personalizzate in `ctx.locals` che saranno poi utilizzabili nei gestori di azioni. I principali utilizzi del before hook sono i seguenti:

- parameter sanitization: modifica l'input per assicurarsi che sia valido
- parameter validation: controlla se l'input soddisfa una serie di criteri
- entity finding
- authorization

After hooks

Negli after hook viene ricevuto il context e la risposta dell'azione, e qui è possibile manipolare la risposta, che sarà poi restituita come versione finale al chiamante. I principali utilizzi del after hook sono i seguenti:

- fare property populating
- rimuovere dati sensibili
- wrappare la risposta in un Object
- convertire la struttura della risposta

Error hooks

Glu error hook vengono chiamati quando viene generato un errore durante il richiamo dell'azione, riceve il context e l'err. Può gestire l'errore e ritornare un'altra risposta (fallback) oppure può lanciare un errore. I principali utilizzi del error hook sono i seguenti:

- error handling
- wrappa gli errori in ulteriori errori
- restituisce risposte fallback

6.1.5 Events

Il broker dispone di un bus di eventi integrato per supportare l'architettura basata sugli eventi e per inviare eventi a servizi locali e remoti. Gli eventi di Moleculer sono di tipo fire-and-forget, il che significa che se il servizio è offline l'evento andrà perso.

Balanced Events

Gli event listeners sono organizzati in gruppi logici, ovvero viene attivato un solo ascoltatore in ogni gruppo. Per inviare eventi bilanciati si utilizza la funzione `broker.emit` e come si denota dalla Figura 6.6 solo un'istanza dei servizi iscritti all'evento riceverà la `emit event`.
Esiste anche la chiamata `broker.broadcast` dove l'evento trasmesso viene inviato a tutti i servizi locali e remoti disponibili. Non è bilanciato, tutte le istanze del servizio lo riceveranno, come si vede a Figura 6.7.

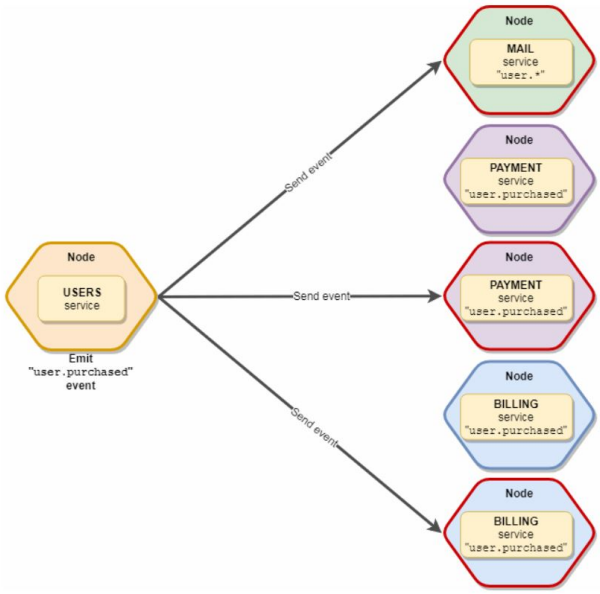


Figura 6.6: Meccanismo di event balancing della `broker.emit`

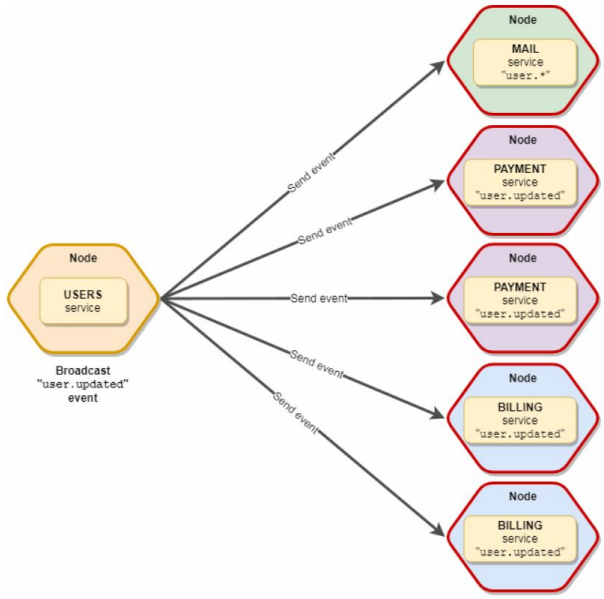


Figura 6.7: Esempio di una `broker.broadcast`

Event parameter validation

Similmente alla validazione dei parametri delle action, anche per gli eventi vi è presente una parameter validation. Nella definizione dell'evento vanno anche definiti i vincoli che si vogliono mettere nei parametri in ingresso, e il validator integrato convalida i parametri negli eventi. Gli errori di validazione non vengono direttamente rispediti al chiamante, ma vengono registrati ed è possibile individuarli con il gestore degli errori globale.

6.1.6 Context

Quando si chiama un'azione o si emette un evento, il broker crea un'istanza Context che contiene tutte le informazioni sulla richiesta e la passa all'azione o al gestore eventi come un singolo argomento, e le informazioni sono accessibili dall'istanza del servizio chiamato. Inoltre il Context può essere utilizzato per accedere l'istanza broker del nodo corrente, l'istanza del chiamante o il suo node ID ecc.

6.1.7 Logging

Tutti i moduli principali di Moleculer hanno un'istanza di logger personalizzata. Vengono ereditati dall'istanza del logger del broker che può essere configurata nelle opzioni del broker; nel progetto viene utilizzato il logger integrato (console di default), che stampa tutti i messaggi di log sulla console. Supporta diversi formattatori incorporati oppure è possibile usare anche un formattatore personalizzato.

6.1.8 Middlewares

Moleculer supporta i middleware. Il middleware è un oggetto con funzioni hook e wrapper. Consente di eseguire il wrapping di gestori di azioni, gestori di eventi, metodi broker ed eventi del ciclo di vita degli hook. Nella Figura 6.8 viene mostrato come Moleculer struttura internamente i vari layer dei middlewares supportati, e anche come vengono integrati con middlewares esterni definiti dall'utente.

6.1.9 Networking

Per poter comunicare con altri nodi (altri ServiceBrokers) è necessario configurare un trasportatore. La maggior parte dei transporter supportati si connettono ad un broker di messaggi centrale che fornisce un modo affidabile per lo scambio di messaggi tra i nodi remoti come mostrato in Figura 6.9. Questi broker di messaggi supportano principalmente il modello di messaggistica del pattern *publish/subscribe*.

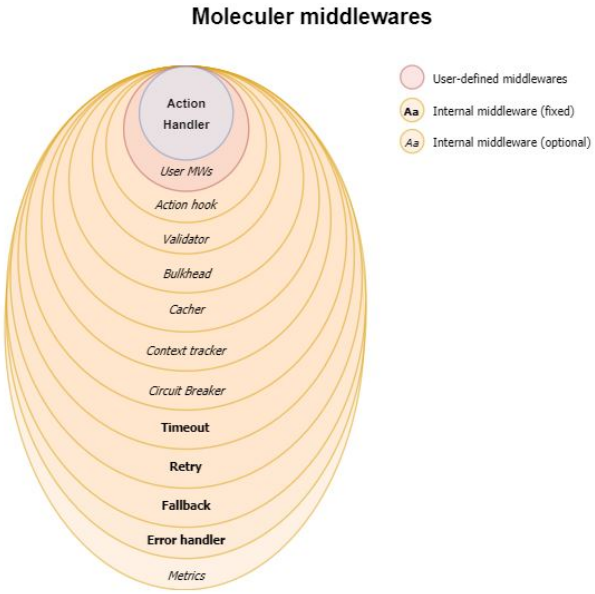


Figura 6.8: Moleculer middlewares architecture

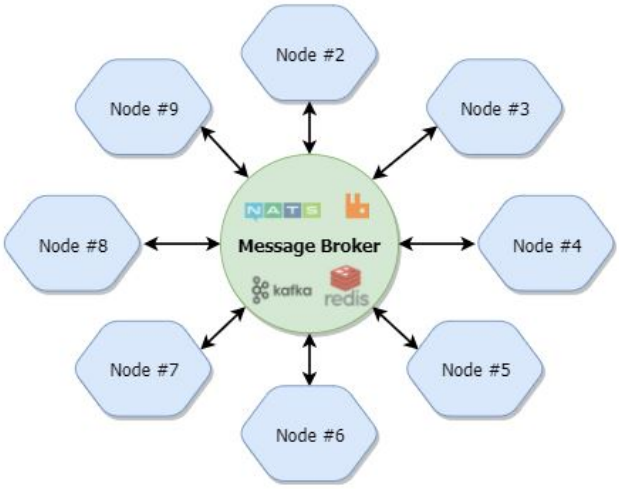


Figura 6.9: Scambio di messaggi tra nodi remoti

6.1.10 Transporters

Il transporter ha il compito di abilitare e facilitare la comunicazione con atri nodi, trasferisce eventi, chiama richieste, elabora risposte, ecc. Se più istanze di un servizio sono in esecuzione su nodi diversi, le richieste verranno bilancia-

te tra loro, l'intera logica di comunicazione è al di fuori delle competenze del transporter. Significa che puoi passare da un trasportatore all'altro senza modificare alcuna riga di codice. Esistono diversi transporter integrati nel framework Moleculer per esempio TCP, NATS, AMQP, ecc.

Serialization

Il transporter necessita di un modulo serializer che serializza e deserializza i pacchetti trasferiti, e il serializzatore predefinito è *JSONSerializer* ma ci sono diversi serializzatori incorporati. Nel progetto viene utilizzato il serializzatore di default JSON che si occupa di serializzare i pacchetti in una stringa JSON e deserializza i dati ricevuti nel pacchetto.

6.1.11 Registry and Discovery

Quando un sistema include più microservizi, i servizi devono essere in grado di trovare la posizione degli altri servizi da cui dipendono. I microservizi devono essere scalabili ed elastici e, se i componenti si guastano, nuove istanze o contenitori devono essere portati online per garantire una disponibilità costante. Ciò significa che gli indirizzi IP delle istanze o dei contenitori in un microservizio possono cambiare costantemente inoltre ogni istanza di un servizio deve inoltre essere continuamente monitorata per verificarne la disponibilità.

Dynamic service discovery

Il framework Moleculer ha un modulo integrato responsabile per la discovery dei nodi e per un heartbeat di verifica periodico. La discovery è dinamica, il che significa che un nodo non ha bisogno di sapere nulla di altri nodi in fase di start. Quando si avvia, annuncerà la sua presenza a tutti gli altri nodi in modo che ognuno possa creare il proprio registro di servizio locale. In caso di stop o di crash inatteso di un nodo, altri nodi lo rileveranno e rimuoveranno i servizi interessati dal loro registro, in questo modo le seguenti richieste verranno instradate ai rimanenti nodi attivi.

Local discovery

La local discovery è l'opzione di discovery di default, si utilizza il modulo transporter per scambiare informazioni sul nodo e pacchetti heartbeat. È il più semplice e il più veloce tra i meccanismi di rilevamento disponibili in quanto non richiede soluzioni esterne, tuttavia questo metodo di rilevamento presenta anche alcuni inconvenienti, soprattutto per distribuzioni su larga scala con centinaia di nodi. I pacchetti heartbeat possono generare una grande quantità di traffico che può saturare il bus di comunicazione e, quindi, influenzare le prestazioni di azioni ed eventi cioè rallentare la consegna di richieste, risposte e event packets.

6.1.12 Load balancing

Con il termine load balancer, andiamo a definire il bilanciamento del carico. In pratica si tratta di una tecnica che consiste nel distribuire il carico di uno specifico servizio e, in questo modo, aumenteremo l'affidabilità e la scalabilità di tutta l'architettura. Un'architettura che sfrutta il load balancer è considerata più affidabile rispetto ad una che non ne fa uso semplicemente perchè se dovessimo avere molte richieste contemporaneamente al nostro server, grazie al load balancer queste verranno ripartite in modo equo tra i diversi nodi disponibili. Qualora uno dei nodi non sia in grado di rispondere alla richiesta, questa verrà indirizzata in un nodo disponibile ed attivo, tale processo prende il nome di *Fault Tollerance*. Tra i vantaggi del load balancer c'è sicuramente quello di avere tempi di caricamento e di risposta più brevi anche in presenza di più richieste contemporanee, inoltre la sicurezza della stabilità dei servizi è un altro dei vantaggi offerti, dal momento che il traffico di un nodo lento/inoperabile viene automaticamente inoltrato ad un altro nodo. Infine tale meccanismo consente la semplificazione della manutenzione dei nodi del sistema, siccome le configurazioni e gli aggiornamenti possono essere effettuati "a caldo" e quindi senza spegnere il sistema e senza che l'utente percepisca nulla, ovviamente a patto che almeno un nodo che offre il servizio da modificare rimanga operativo. Moleculer ha diverse strategie di bilanciamento del carico integrate. Se un servizio è in esecuzione su più istanze di nodi, ServiceRegistry utilizza queste strategie per selezionare un singolo nodo tra quelli disponibili. Per configurare la strategia è necessario impostare la *strategy broker options* nel registro delle proprietà. Esistono diverse strategie integrate: round robin, random, cpu usage-based, sharding, ma è possibile anche integrare strategie personalizzate.

RoundRobin strategy

Abbiamo deciso di utilizzare questa strategia, che seleziona un nodo basandosi sull'algoritmo Round-Robin. Nel nostro caso con Round-Robin si indica un processo circolare in cui le richieste in entrata del server vengono gestite dal load balancer in una lista di attesa e vengono distribuite in ordine tra i nodi presenti. In questo modo ogni nuova richiesta viene assegnata al nodo successivo della serie e gli accessi vengono quindi ripartiti equamente.

6.1.13 Fault tolerance

Moleculer ha diverse funzionalità di fault tollerance integrate, che possono essere abilitate o disabilitate nel *broker options*.

Circuit Breaker

La soluzione built-in di Moleculer, si basa su una soglia che utilizza una finestra temporale per controllare il tasso di richieste non riuscite e una volta raggiunto il valore di soglia, fa scattare l'interruttore. L'interruttore automatico può impedire a un'applicazione di tentare ripetutamente di eseguire un'operazione che

potrebbe non riuscire. Il server ritorna direttamente senza attendere che l'errore venga risolto o sprecando cicli computazionale per un errore che può essere di lunga durata. Il modello circuit breaker consente inoltre di attuare operazioni di monitoraggio rilevando se il guasto è stato risolto e in caso ripubblicare il nodo o il servizio, ritornando richiamabile.

Retry

Questa soluzione può essere utilizzata per recuperare un'operazione che da errore, richiamando nuovamente la stessa operazione finchè non vengono raggiunti i criteri di arresto. I criteri possono essere definiti in *retryPolicy* alla creazione del broker, i principali sono:

- *retries*: numero di tentativi.
- *delay*: ritardo nel richiamare l'operazione
- *check*: una funzione per controllare le richieste non riuscite.

Timeout

Il meccanismo di fault tolerance timeout consente di specificare quanto tempo può impiegare un'operazione per completare la sua esecuzione e di interromperla in caso di timeout. Può essere impostato globalmente nelle opzioni del broker o nelle opzioni di chiamata. Se il timeout è definito e la richiesta è scaduta, il broker genererà un errore *RequestTimeoutError*.

Fallback

È possibile utilizzare anche le fallback che risulta utile quando non vuoi restituire errori agli utenti. Viene chiamata invece un'altra azione o restituire qualche risultato di default. La risposta della fallback può essere impostata nelle calling options o nella definizione dell'azione. La fallback restituisce una Promise e il broker passa gli oggetti `Context` ed `Error` dello scope corrente a questa funzione come argomenti.

6.1.14 Errors

Molecular ha alcuni errori incorporati per generare un errore nei servizi. Gli errori hanno in ingresso come parametri un *message* che definisce il messaggio d'errore, il *code* per il codice d'errore, il *type* per il tipo d'errore ed infine il *data* per inserire qualsiasi dato rilevante per descrivere meglio l'errore. Gli errori più comunemente usati sono stati:

- *MoleculerError*: la classe di errore di base.
- *MoleculerServerError*: per descrivere errori che avvengono all'interno del server.
- *MoleculerClientError*: per descrivere errori derivanti dal client.

6.1.15 API Gateway

Moleculer-web è il servizio gateway API ufficiale per il framework Moleculer, viene usato per pubblicare i servizi come API RESTful. Tra le varie feature che vengono offerte troviamo:

- supporto a HTTP e HTTPS.
- gestione di route multiple.
- alias names
- whitelist
- body parser multipli (json, urlencoded)
- supporta middleware a livello globale, di route e di alias.
- CORS headers
- before after call hooks

Aliases

È possibile utilizzare nomi alias invece di nomi di azioni o di metodi. Il gateway API si occupa di implementare l'azione `listAliases` che elenca gli endpoint HTTP alle mappature delle azioni.

Parameters

Il gateway API raccoglie i parametri dalla stringa di query URL, dai parametri di richiesta e dal corpo della richiesta e li unisce, i risultati vengono inseriti nella `req.params` .

Whitelist

Se non è necessario pubblicare tutte le azioni, è possibile filtrarle con l'opzione whitelist, vengono usate stringhe di corrispondenza o regex nell'elenco delle azioni da pubblicare. Ad esempio per abilitare tutte le azioni, utilizza l'espressione "**".

6.2 Docker

Docker è una piattaforma software che permette di *creare*, *testare* e *distribuire* applicazioni con la massima rapidità. Docker raccoglie il software in unità standardizzate chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime. Con Docker, è possibile distribuire e ricalibrare le risorse per un'applicazione in qualsiasi ambiente, tenendo sempre sotto controllo il codice eseguito. La tecnologia Docker

utilizza il kernel di Linux e le sue funzionalità, per isolare i processi in modo da poterli eseguire in maniera indipendente. Questa indipendenza è l'obiettivo dei container: la capacità di eseguire più processi e applicazioni in modo separato per sfruttare al meglio l'infrastruttura esistente pur conservando il livello di sicurezza che sarebbe garantito dalla presenza di sistemi separati.
Si è detto quindi che Docker è una tecnologia di containerizzazione che consente la creazione e l'utilizzo dei container Linux. Per container linux si intende un insieme di uno o più processi isolati dal resto del sistema. Poiché tutti i file necessari per eseguire tali processi vengono forniti da un'immagine distinta, i container Linux sono portabili e coerenti in tutti gli ambienti, dallo sviluppo, ai test, fino alla produzione. Il container che ospita l'applicazione presenta le librerie, le dipendenze e i file necessari, che consente di passare all'ambiente di produzione, senza alcun impatto negativo. Diversamente dalla virtualizzazione che consente di eseguire più sistemi operativi (Windows o Linux) contemporaneamente su un singolo sistema hardware, i container condividono lo stesso kernel del sistema operativo e isolano i processi applicativi dal resto del sistema, come mostrato dalla Figura 6.10.

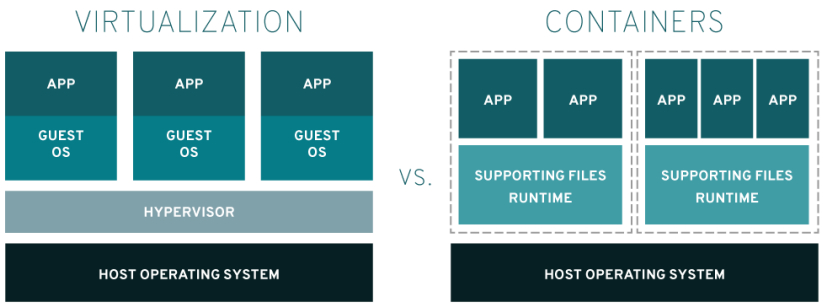


Figura 6.10: Virtualizzation vs Container

I container sono autosufficienti perché contengono già tutte le dipendenze dell'applicazione e non richiedono quindi particolari configurazioni sull'host, ma sono soprattuto portabili perché essi sono distribuiti in un formato standard (le immagini appunto), che può essere letto ed eseguito da qualunque server Docker. Inoltre, i container sono leggeri perché sfruttano i servizi offerti dal kernel del sistema operativo ospitante, invece di richiedere l'esecuzione di un kernel virtualizzato come avviene nel caso delle VM. Ciò permette di ospitare un gran numero di container nella stessa macchina fisica.

6.2.1 I vantaggi dei container Docker

- **Modularità**
  L'approccio Docker alla containerizzazione si basa sulla capacità di estrarre i singoli componenti di un'applicazione, da aggiornare o riparare. Oltre a questo approccio basato sui microservizi, è possibile condividere i

processi tra più applicazioni in modo molto simile a quello usato dalla Service-Oriented Architecture (SOA).

- **Controllo delle versioni delle immagini**
  Ogni file immagine Docker è composto da più strati, che insieme costituiscono una singola immagine. Uno strato viene creato ad ogni modifica dell'immagine. Ogni volta in cui un utente specifica un comando (es. run o copy), viene creato un nuovo strato. Docker riutilizza questi strati per velocizzare il processo di creazione dei container. Le modifiche sono condivise tra le immagini, migliorando ulteriormente la velocità, la dimensione e l'efficienza. Il controllo delle versioni fa parte del processo di stratificazione. Ogni volta che viene apportata una modifica, il registro delle modifiche offre il controllo totale sulle immagini containerizzate.

- **Rollback**
  Uno dei maggiori vantaggi della stratificazione è la capacità di eseguire il rollback. Ogni immagine è composta da strati. Se l'iterazione di un'immagine non è soddisfacente, è possibile riportala alla versione precedente. Ciò consente uno sviluppo agile e aiuta a ottenere l'integrazione e il deployment continui (CI/CD).

- **Deployment rapido**
  I container basati su Docker possono ridurre il deployment a pochi secondi. Creando un container per ogni processo, puoi condividere con rapidità i processi simili con le nuove applicazioni. Poiché non è necessario riavviare un sistema operativo per aggiungere o spostare un container, i tempi per il deployment sono sostanzialmente più brevi. Inoltre, grazie alla velocità del deployment, attraverso i container è possibile creare ed eliminare dati in modo sicuro, semplice ed economico.Dunque, la tecnologia Docker è caratterizzata da un approccio basato sui microservizi granulare e controllabile.

6.2.2 Dockerfile

Docker può creare immagini automaticamente leggendo le istruzioni da un Dockerfile, che è un documento di testo che contiene tutti i comandi che un utente può chiamare sulla riga di comando per assemblare un'immagine. Utilizzando la build docker, gli utenti possono creare una build automatizzata che esegue diverse istruzioni della riga di comando in successione. Nel nostro caso il Dockerfile viene configurato per eseguire i servizi Moleculer. Dovendo utilizzare container multipli ci serve configurare anche il Docker Compose. Compose è uno strumento per la definizione e l'esecuzione di applicazioni Docker multi-container. Con Compose, si utilizza un file YAML per configurare i servizi della applicazione, e infine, con un solo comando, si creano e si avviano tutti i servizi dalla configurazione. Nel nostro caso Docker compone i file per eseguire i servizi Moleculer con NATS (se necessario) e Traefik (bilanciamento del carico del gateway API).

6.3 Socket.IO

Socket.IO consente la comunicazione bidirezionale tra client e server. Le comunicazioni bidirezionali sono abilitate quando un client possiede un'istanza Socket.IO nel browser e un server, con integrato il pacchetto Socket.IO, possiede anch'esso un'istanza Socket.IO con i listener adatti alla risposta. Sebbene i dati possano essere inviati in diversi formati, JSON è il più semplice e quello adottato nel progetto. Per stabilire la connessione e per scambiare dati tra client e server, Socket.IO utilizza Engine.IO e l'architettura generale del framework è riassunta nella Figura 6.11.

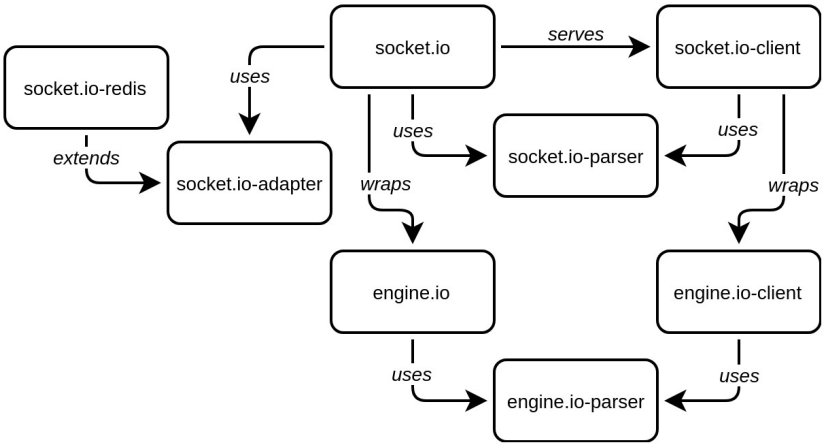


Figura 6.11: Socket.IO Architecture

Per stabilire la connessione Socket.IO utilizza in una prima fase xhr-polling per creare una connessione long-polling, infine, si aggiorna al miglior metodo di connessione disponibile. Nella maggior parte dei casi, ciò comporterà una connessione WebSocket.

6.4 Mongoose

MongooseJS è un Object Document Mapper (ODM) che semplifica l'utilizzo di MongoDB traducendo i documenti di un database MongoDB in oggetti JavaScript. Fornisce una soluzione basata su schemi per modellare i dati dell'applicazione. Include il casting di tipo integrato, convalida, creazione di query, ecc. Come accennato precedentemente Mongoose utilizza schemi per modellare i dati che un'applicazione desidera archiviare e manipolare in MongoDb, lo schema descrive gli attributi delle proprietà (ovvero i campi) che l'applicazione manipolerà. Tra i principali attributi troviamo:

- Data type (ad esempio String, Number, etc.)

- Descrivere se il campo è obbligatori oppure opzionale
- Descrivere se il campo è un valore univoco oppure no

Rispetto all'API nativa di MongoDb in Mongoose vengono semplificate le query. Le query MongoDB sono costituite da BSON strutturato che specifica nomi di proprietà del documento, operatori e valori, che insieme specificano come filtrare i documenti; Mongoose invece utilizza la combinazione e il concatenamento di funzioni, piuttosto che operatori per filtrare i documenti. Sebbene più lunga del suo equivalente MongoDB per le query più semplici, nelle query più complesse la versione Mongoose risulta più facile sia da costruire che da leggere.

6.5 MongoDB Cloud

MongoDB Cloud è considerato un DBaaS, l'equivalente del software-as-a-service (SaaS) con la sola differenza che in questo caso si fa riferimento a un database. Il funzionamento è pressoché identico: si paga a consumo solo ciò che viene utilizzato e nel frattempo si lascia che il sistema gestisca tutti i dettagli di provisioning e scalabilità necessari per soddisfare la domanda, mantenendo allo stesso tempo prestazioni costantemente elevate. Le moderne piattaforme database-as-a-service consentono un accesso facile (ma controllato) ai sistemi cloud tramite API e driver coerenti, semplificando l'accesso alle risorse. La flessibilità dei database NoSQL nel cloud di oggi migliora notevolmente l'agilità nella gestione dei dati e nello sviluppo del software in quanto non risulta più necessario attendere tempi di inattività che si potevano riscontrare per gli aggiornamenti del sistema, il ribilanciamento dei cluster o l'equipaggiamento di risorse hardware più veloci. È infatti possibile contare su:

- *Controlli di sicurezza integrati*: I cluster vengono distribuiti in un unico Virtual Private Cloud protetto da firewall dedicati per un maggiore isolamento di rete. È possibile configurare sofisticate regole di accesso basate sui ruoli per controllare quali utenti e team possono accedere, manipolare ed eliminare i dati nei tuoi database. Tutto il traffico di rete è crittografato utilizzando TLS (Transport Layer Security).
- *Gestione delle performance*: MongoDB permette di scalare il database in qualsiasi direzione, aumentano o diminuendo la capacità di archiviazione dei cluster senza tempi di inattività dell'applicazione. Possiede strumenti per la gestione automatizzata della memoria che gli consente di modificare le dimensioni del database in base alle esigenze. Implementa inoltre strumenti per l'ottimizzazione delle prestazioni e per la visibilità di metriche in tempo reale.
- *Resilienza ai guasti e affidabilità*: MongoDB costruito in modo tale da avere una tolleranza agli errori estremamente elevata anche in sistemi distribuiti implementando meccanismi di ripristino automatico dei dati. Se un nodo primario per qualche motivo dovesse non essere più disponibile,

MongoDB eseguirà automaticamente un failover in pochi secondi, senza necessità di intervento manuale. I database vengono solitamente replicati e sottoposti a backup per impostazione predefinita, in modo che nessun singolo punto di errore possa portare la tua applicazione offline.

Capitolo 7

Testing

"Il testing mostra la presenza di errori, non la loro assenza" — *Dijkstra*

La realizzazione di test è stato un aspetto fondamentale per verificare la correttezza e la qualità del codice, garantendo anche un'ulteriore documentazione del software sviluppato. La loro implementazione non è avvenuta con la metodologia stringente del Test Driven Development (TDD), bensì con un approccio più tradizionale, in cui la funzionalità veniva testata una volta pronta. In particolare è stata usata la libreria Jest.

7.1 Jest

Jest è un framework per le applicazioni Javascript, realizzato da Facebook in ambito OpenSource, orientato alla semplicità d'uso si caratterizza per diverse peculiarità:

- Nessuna configurazione per la maggior parte dei progetti Javascript.
- Utilizzo di snapshots per tenere traccia di oggetti molto grandi.
- Isolamento dei test per permetterne il run in parallelo di diversi processi massimizzando le performance.
- Presenza di molteplici API ben documentate.

Dal punto di vista produttivo si presenta competitivo in quanto semplifica il processo di sviluppo, in particolare è:

- Developer Ready: ovvero una soluzione completa di test JavaScript, ed è pronto per l'uso per la maggior parte dei progetti JavaScript.
- Instant Feedback: presenta la watch mode, ovvero una modalità di visualizzazione veloce e interattiva esegue solo i test relativi ai file modificati.

- Snapshot Testing: una funzionalità che permette di acquisire snapshot di oggetti di grandi dimensioni per semplificare i test e analizzare il loro cambiamento nel tempo.

7.1.1 Code Coverage

La code coverage, o copertura del codice, viene solitamente definita come la percentuale di codice attraversato dei test rispetto al totale della code base (tuttavia non esiste una definizione precisa, infatti bisogna specificare esattamente cosa significa "coprire" una parte di codice e a che livello se ad esempio modulo, classe, metodo etc.). Il testing mostra la presenza di errori, non la loro assenza riprendendo la citazione Dijkstra, per cui, non si è cercato di avere necessariamente un testing con il 100% di coverage, in quanto non sarebbe comunque immune da bugs, lo scopo infatti non è stato quindi aumentare al massimo la coverage, ma testare le parti appropriate dell'applicazione (Figura 7.1). Per lo sviluppo di questi test vengono usati principalmente i seguenti costrutti forniti da Jest:

- *describe*: crea un blocco di codice che raggruppa diversi test correlati, non è necessario infatti si possono scrivere i blocchi di test direttamente al livello superiore, tuttavia il describe è utile se si preferisce organizzare i test in gruppi. Nella scrittura dei test viene utilizzata la *expect* per controllare che i valori soddisfino determinate condizioni, infatti vengono forniti una serie di "matcher" che consentono di convalidare valori diversi.

- *beforeAll*: esegue una funzione prima che venga eseguito uno qualsiasi dei test, e se la funzione restituisce una promise Jest attende che la promise si risolva prima di eseguire i test. Viene utilizzato per creare i broker che servono per richiamare le azioni nei servizi da testare, e in generale impostare uno stato globale che verrà utilizzato da molti test.

- *afterAll*: similmente alla *beforeAll* esegue una funzione dopo che tutti i test sono stati completati, se la funzione restituisce promise, Jest attende che la promise si risolva prima di continuare. Viene utilizzata principalmente dopo aver effettuato test che andassero a scrivere/modificare per ripulire il database da tali file temporanei.

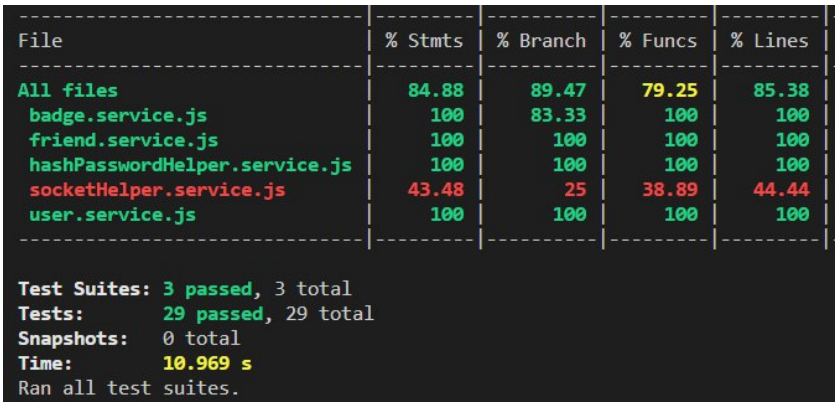


Figura 7.1: Jest test output

7.1.2 Deployment Test

In questa sezione si andranno a descrivere i test effettuati dopo aver eseguito il deployment dei container sulla piattaforma Docker. Dopo aver eseguito lo startup dei container Docker saranno disponibili i servizi visualizzati nella Figura 7.2. Da notare che vengono caricati *movieparty-badge* e *movieparty-badge-replica*, due container che forniscono la stessa funzionalità. Questo permette al meccanismo di load balancing (nel nostro caso è stato scelto il Round Robin come descritto nella sezione 6.1.12) di spartire il carico delle richieste tra i due container. Si può vedere questo meccanismo in azione nella Figura 7.3, dove vengono fatte quattro richieste al servizio badge. È possibile vedere come due richieste vengono innoltrate al *movieparty-badge-replica* e le altre due al *movieparty-badge*. Dopo aver visto questo meccanismo di bilanciamento del carico, viene testata l'affidabilità del sistema; in questo caso viene stoppata dall'esecuzione il container *movieparty-badge-replica*, e il comportamento atteso dal sistema è quello di continuare a funzionare, caricando le richieste al container rimanente ovvero *movieparty-badge*. Come si vede dalla Figura 7.4 una volta terminata l'esecuzione del container *movieparty-badge-replica*, le richieste successive, come da ipotesi, vengono inoltrate a *movieparty-badge* e il sistema risulta perfettamente responsivo. Infine l'ultimo test che viene effettuato è quello di riattivare il container *movieparty-badge-replica*, aspettandoci che dopo la sua messa in funzione il sistema ritornerà al suo funzionamento iniziale (ovvero quello visto nella Figura 7.3). Come atteso infatti nella Figura 7.5 dopo che il servizio è stato ricaricato le richieste vengono inoltrate in maniera alternata tra i due container che contengono il servizio del badge.

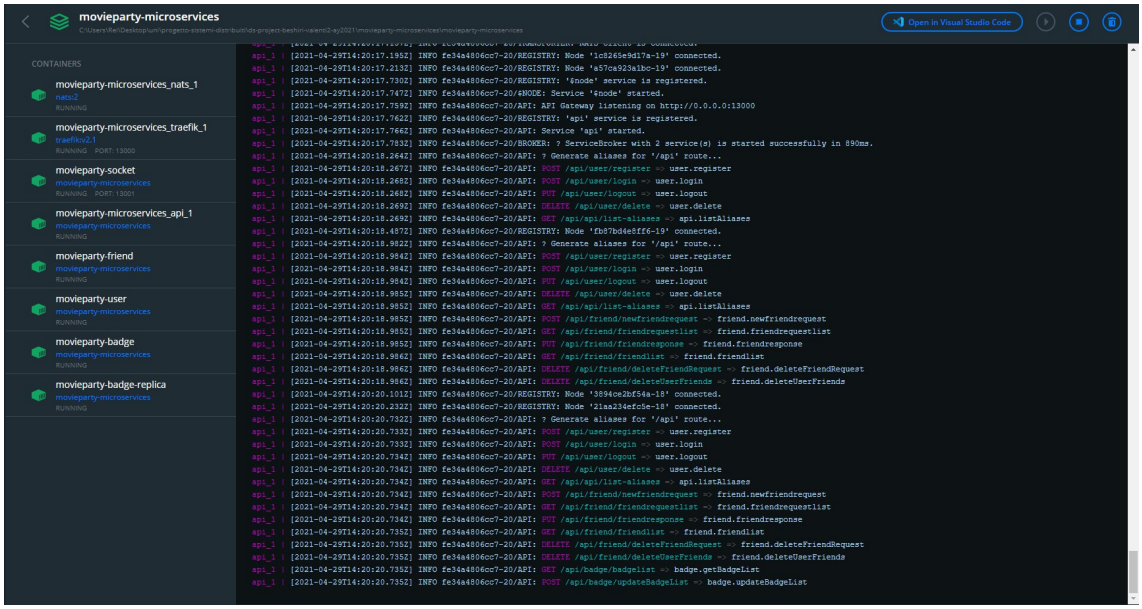


Figura 7.2: Docker Startup

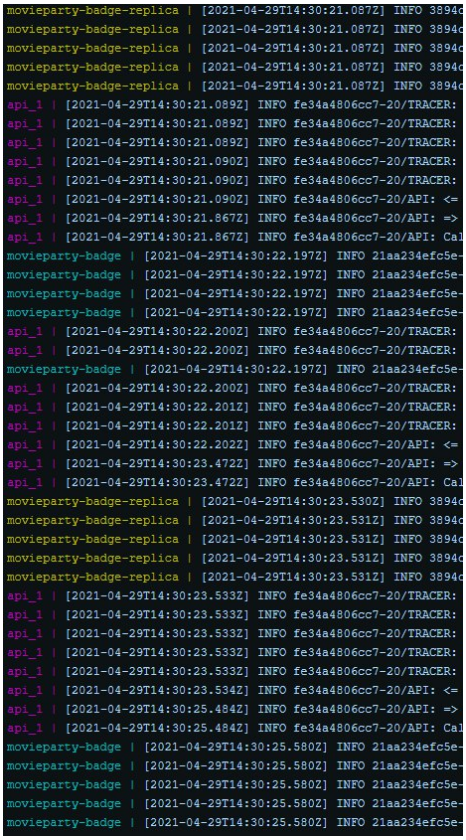


Figura 7.3: Docker Replica Service-Badge Example

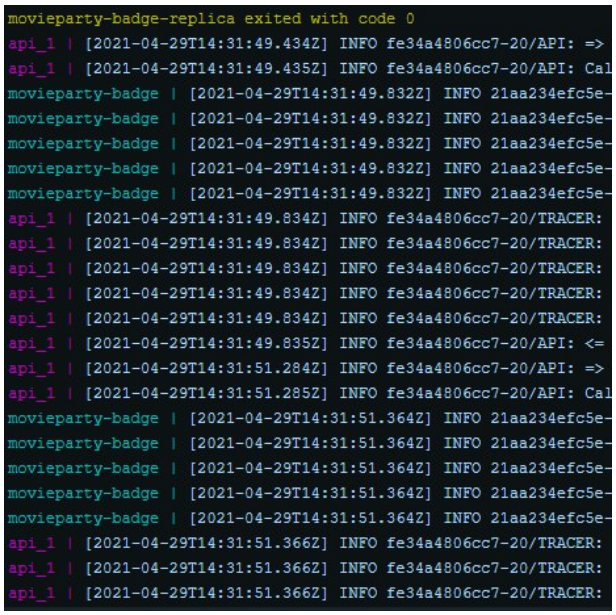


Figura 7.4: Docker Replica Service Stopped

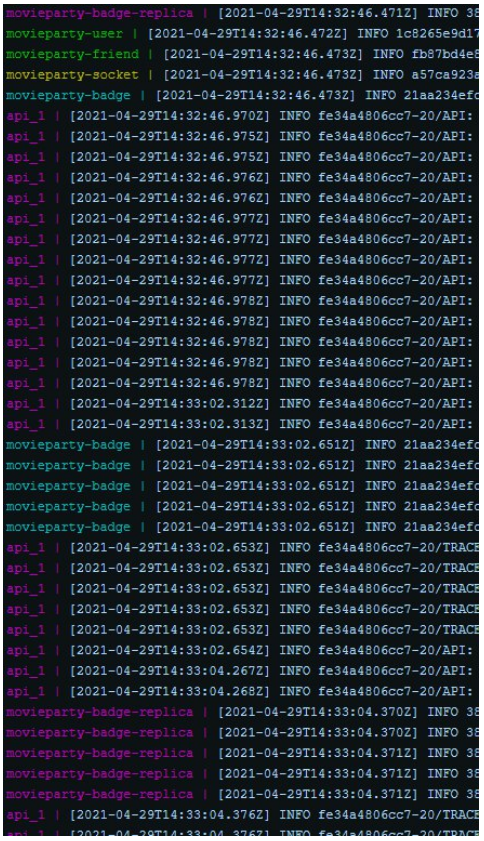


Figura 7.5: Docker Replica Service Reloaded

Capitolo 8

Istruzioni deployment

In questo capitolo verranno descritte le istruzioni per poter deployare l'applicativo, prima però si descrive la strategia scelta per il deployment distribuito su Docker. La scelta è ricaduta sul service instance per container di Docker dove vengono confezionati i package e distribuiti i servizi. Viene creato un pacchetto del servizio come immagine del contenitore Docker e distribuita ogni istanza del servizio come contenitore. Questo permette al servizio di essere distribuibile e scalabile (è semplice aumentare e diminuire un servizio modificando il numero di istanze di container) in modo indipendente e le istanze del servizio sono isolate l'una dall'altra. Inoltre con docker si ha una creazione e distribuzione rapida di un servizio limitando al minimo le risorse come CPU e memoria consumate da un servizio. Infine dalla piattaforma docker è anche possibile monitorare il comportamento di ogni istanza del servizio.

8.1 Istruzioni

In questo paragrafo verranno illustrate le istruzioni su come poter deployare l'applicativo, dopo aver scaricato il progetto saranno presenti due cartelle principali: *movieparty-app* e *movieparty-microservices*.

8.1.1 movieparty-app

Per avviare il client innanzitutto bisogna recarsi nella cartella "movieparty-app" ed eseguire il comando `npm install`. Una volta installate tutte le librerie utilizzate dal client, sarà sufficiente eseguire il comando `npm start` per avviare il client.

8.1.2 movieparty-microservices

Per avviare il server bisogna recarsi prima nella "movieparty-microservices" ed eseguire il comando `npm install`, successivamente ripetere il comando nella

sottocartella "movieparty-microservices". Una volta installate tutte le librerie utilizzate dal server, sarà sufficiente eseguire il comando `npm run dev` per avviare il server. In alternativa per caricare le immagini Docker ed avviare i servizi come container Docker sarà sufficiente eseguire il comando `npm dc:up`.

Capitolo 9

Conclusioni

La realizzazione di questo progetto ha contribuito ad arricchire il bagaglio culturale delle tecniche acquisite durante il corso, dandoci la possibilità di mettere alla prova le nostre conoscenze nella realizzazione di funzionalità che coinvolgano tecnologie diversificate nella stessa soluzione. Lo sviluppo di questo software ci ha permesso di comprendere al meglio ed imparare numerosi aspetti riguardanti lo sviluppo di sistemi distribuiti: analizzando le varie architetture di rete e le varie tipologie di comunicazioni che ci possono essere tra client e server. Un altro aspetto molto importante riguarda la scoperta di nuovi framework, servizi e tecnologie che non avevamo avuto modo di conoscere o approfondire in modo esaustivo prima di questo corso. Nel complesso, all'interno del team non sono state riscontrate grosse problematiche relative alla pianificazione degli Sprint e alla metodologia di sviluppo. Anzi, l'intero team considera che tale progetto sia stata un'esperienza altamente formativa, permettendoci di imbattersi in aspetti di design, progettazione e sviluppo di un applicazione distribuita.

Capitolo 10

Sequence Diagrams

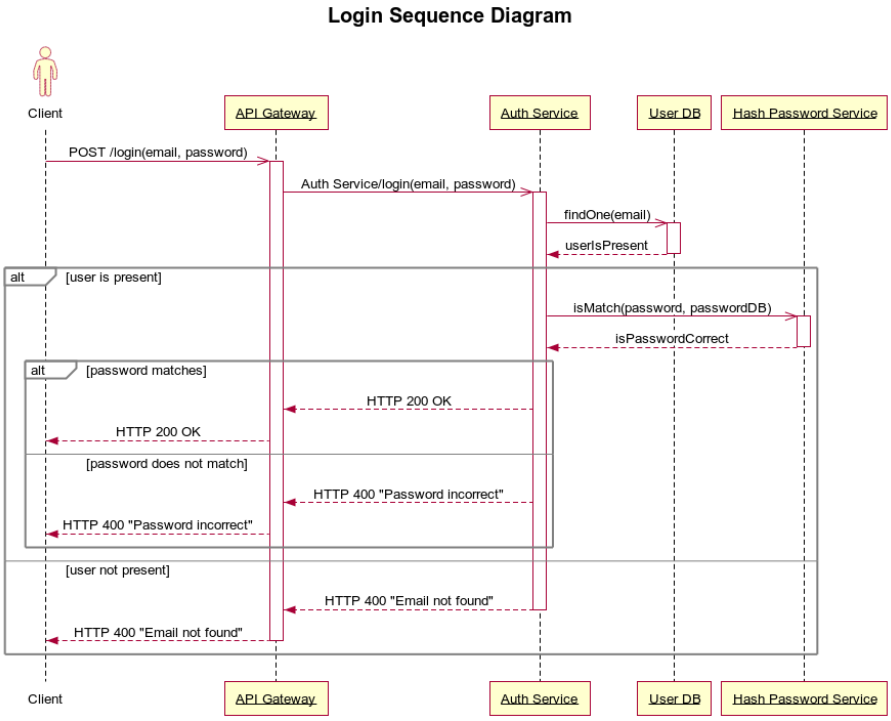


Figura 10.1: Sequence diagram dell'azione di Login

Per l'azione di Login, una volta ricevuta la richiesta di login l'API Gateway inoltra la richiesta all'Authentication Service che si occupa inizialmente di verificare se lo username è presente nel database, successivamente di fare il controllo della password con l'ausilio dell'Hash Password Service. In caso di verifiche andate a buon fine viene poi inviata una risposta di corretto login lato client.

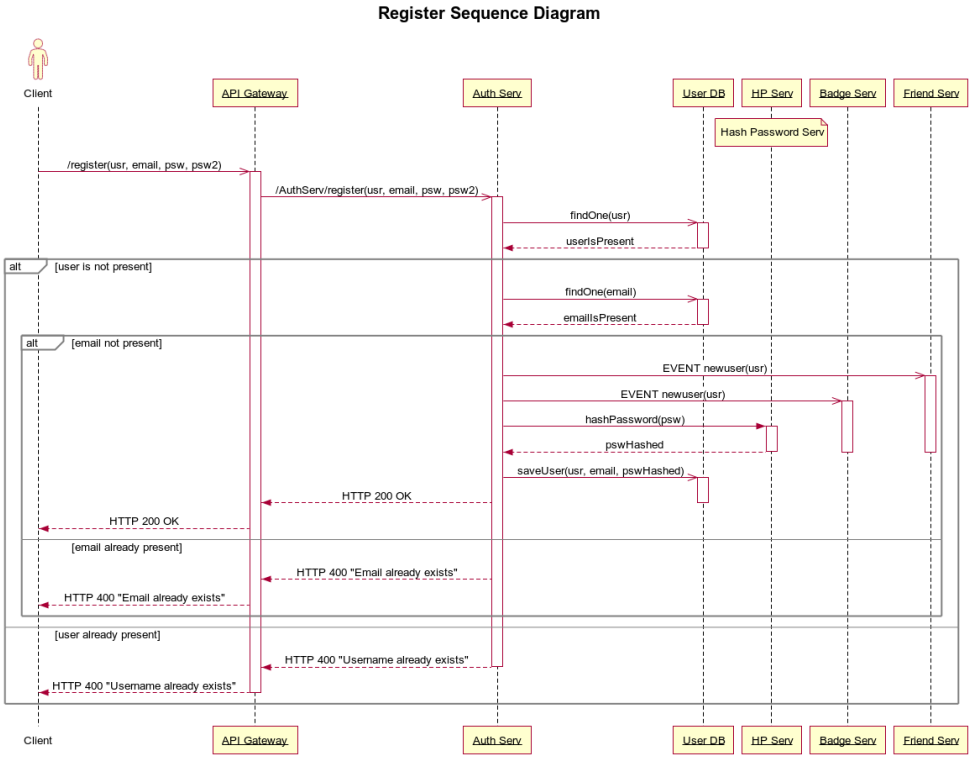


Figura 10.2: Sequence diagram dell'azione di Registrazione

Questo scenario rappresenta la sequenza di azioni coinvolte nell'eseguire la registrazione di un utente. Le entità coinvolte sono molteplici, inizialmente la richiesta di registrazione viene ricevuta dall'API Gateway che si occuperà di inoltrare la richiesta al servizio di competenza, ovvero l'Authentication Service. Prima di registrare le credenziali dell'utente viene fatto un controllo per verificare se l'utente non fosse già presente all'interno del database, e in quel caso ritornare un messaggio di errore. Nel caso in cui l'utente non fosse già registrato, si coinvolgono le entità del Friend Service, Badge Service ed infine Hash Password Service, che rispettivamente hanno il compito di creare nel database la lista di amici, la lista dei badge e l'hashing della password che sarà poi salvata nel DataBase dell'utente. Una volta salvato nello UserDB viene poi spedito lato client una risposta di corretta registrazione dell'utente.

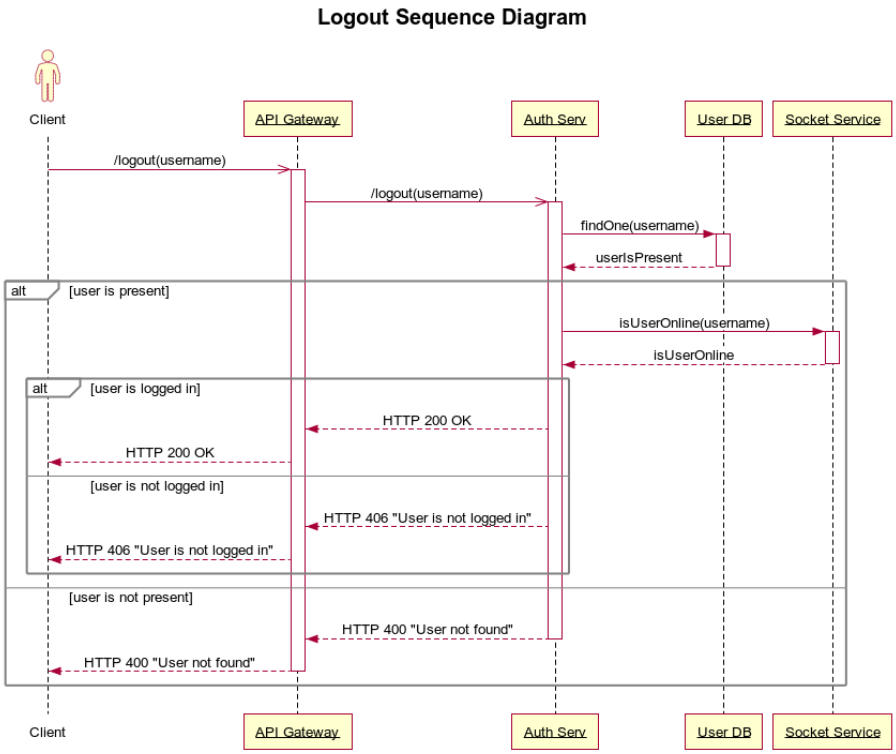


Figura 10.3: Sequence diagram dell'azione di Logout

Come per il Login anche nel caso del Logout viene indirizzata, da parte dell'API Gateway, la richiesta all'Authentication Service che dopo aver controllato l'esistenza dell'utente nello UserDB, procede alla verifica del suo stato. Viene richiamata la Socket Service, e se l'utente risulta effettivamente online si procede a renderlo offline e ad inviare la risposta del corretto cambio di stato lato client.

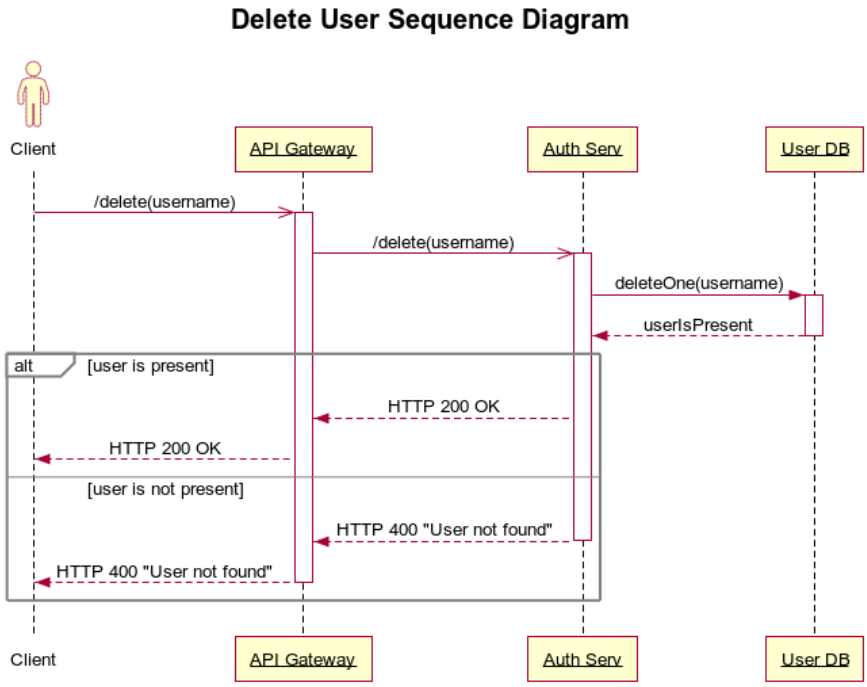


Figura 10.4: Sequence diagram dell'azione di Cancellazione di un utente
Per questa semplice azione l'API Gateway richiama l'Authentication Service. Viene fatta la richiesta di cancellazione allo UserDB e successivamente viene spedita la risposta ottenuta al client.

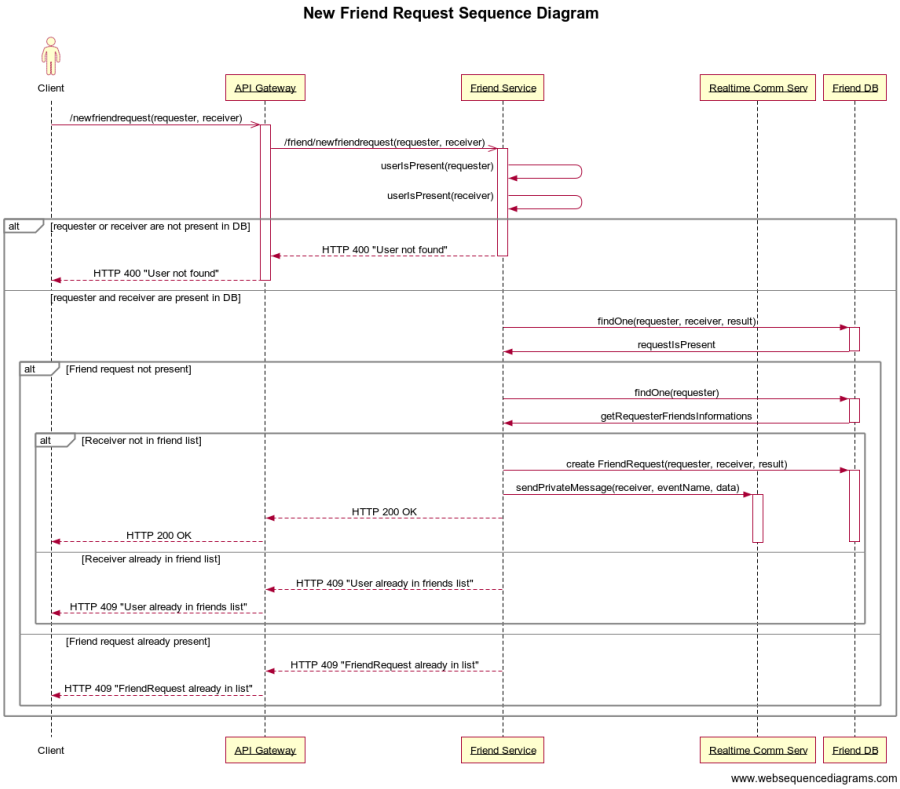


Figura 10.5: Sequence diagram dell'azione di una nuova richiesta di amicizia

Questa azione viene inoltrata da parte dell'API Gateway verso il Friend Service. Prima di procedere alla creazione della richiesta di amicizia da salvare nel FriendDB vengono fatti diversi controlli. Si controlla innanzitutto che il requester e il receiver dell'azione siano presenti nel DataBase, successivamente si controlla che la richiesta di amicizia non sia stata già inviata precedentemente e che i due utenti non siano già amici. Nel caso in cui vengano passate queste verifiche viene aggiunto nel DataBase una richiesta di amicizia e viene richiamato il Socket Service che si occuperà di inviare una notifica real time al receiver della richiesta di amicizia nel caso in cui quest'ultimo sia online al momento dell'invio della richiesta.

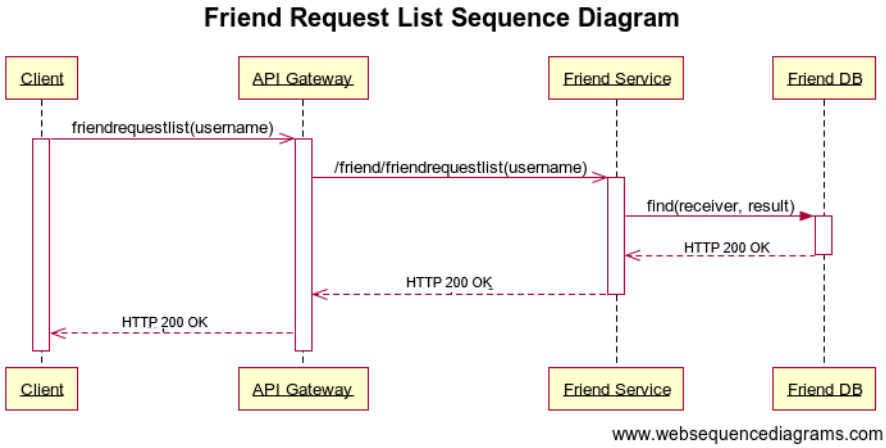


Figura 10.6: Sequence diagram dell'azione per ottenere la lista di richieste di amicizia
Questa semplice azione coinvolge il Friend Service e il FriendDB. L'API Gateway spedisce la richiesta ricevuta al Friend Service che si incarica di contattare il FriendDB per farsi consegnare la lista di richieste di amicizia di un utente specifico. La lista di richieste di amicizia viene poi consegnata come risultato della richiesta al client.

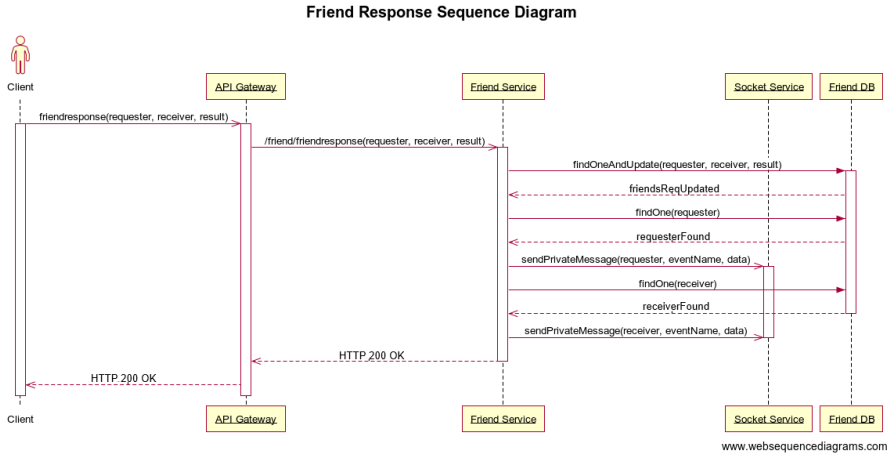


Figura 10.7: Sequence diagram per la risposta ad una richiesta di amicizia
Quando si ha una risposta ad una richiesta di amicizia l'API Gateway richiama il Friend Service che si occupa di contattare il FriendDB per aggiornare la richiesta di amicizia precedentemente inviata, e una volta completata questa azione viene richiamato il Socket Service per inviare sia al requester che al receiver il corretto completamento della richiesta.

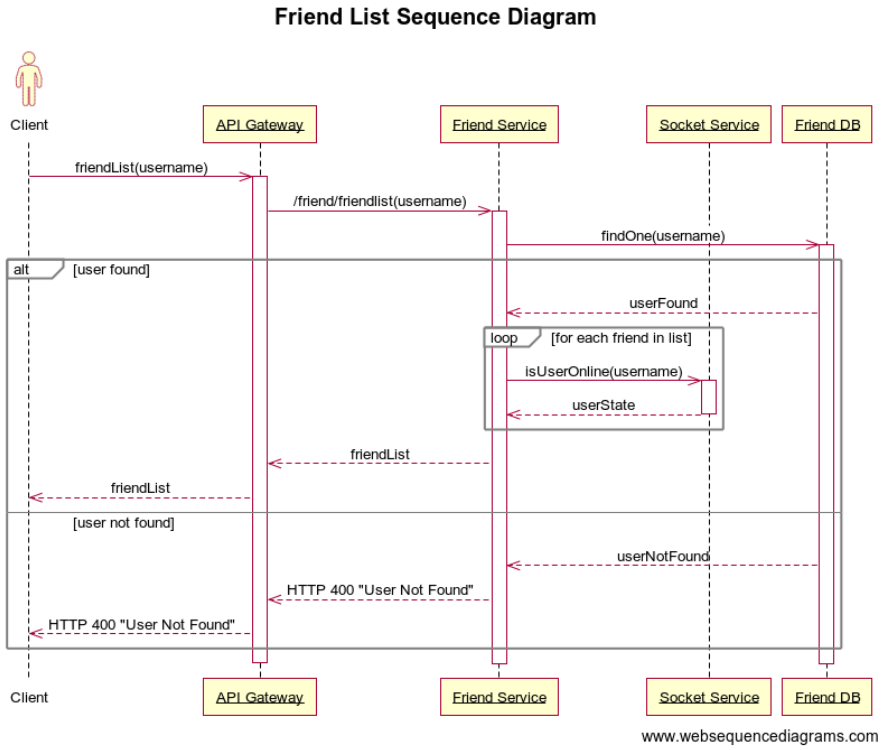


Figura 10.8: Sequence diagram dell'azione di richiesta della lista amici di un utente

Questa azione coinvolge il Friend Service che richiede la lista degli amici di uno specifico utente al FriendDB. Una volta ricevuta la lista degli amici prima di spedire la risposta della chiamata, viene coinvolto il Socket Service. La sua funzione è quella di verificare se un utente sia online oppure offline, questo permette poi di etichettare la lista degli amici di un utente con l'informazione aggiuntiva riguardante il loro stato. Viene poi spedita come risposta al client la lista degli amici e il loro stato.

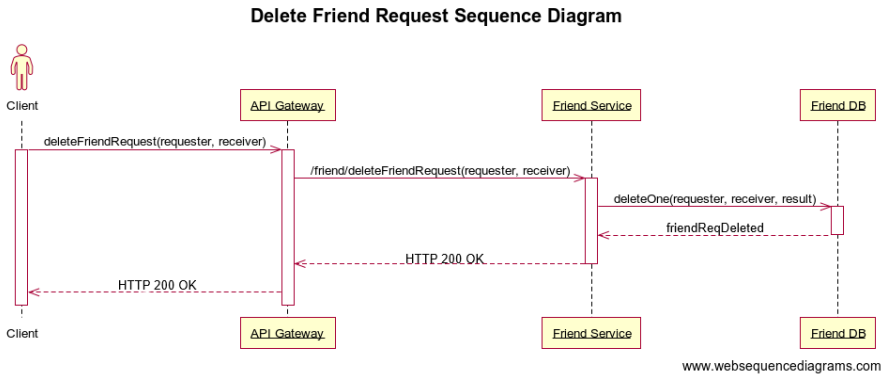


Figura 10.9: Sequence diagram dell'azione di cancellazione di una richiesta di amicizia
In questa semplice azione l'API Gateway inoltra la richiesta al Friend Service che si incarica di contattare il FriendDB e di cancellare una richiesta di amicizia dal DataBase.

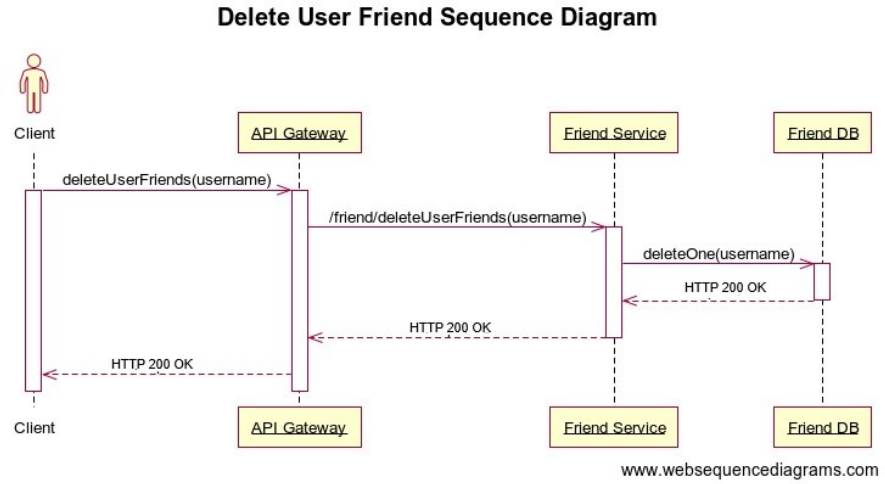


Figura 10.10: Sequence diagram dell'azione di cancellazione di una amicizia
In maniera similare all'azione di cancellazione di una richiesta di amicizia, anche in questa azione vengono coinvolti il Friend Service e il FriendDB. In questo caso però il Friend Service contatta il FriendDB per cancellare un amico dalla lista degli amici del richiedente.

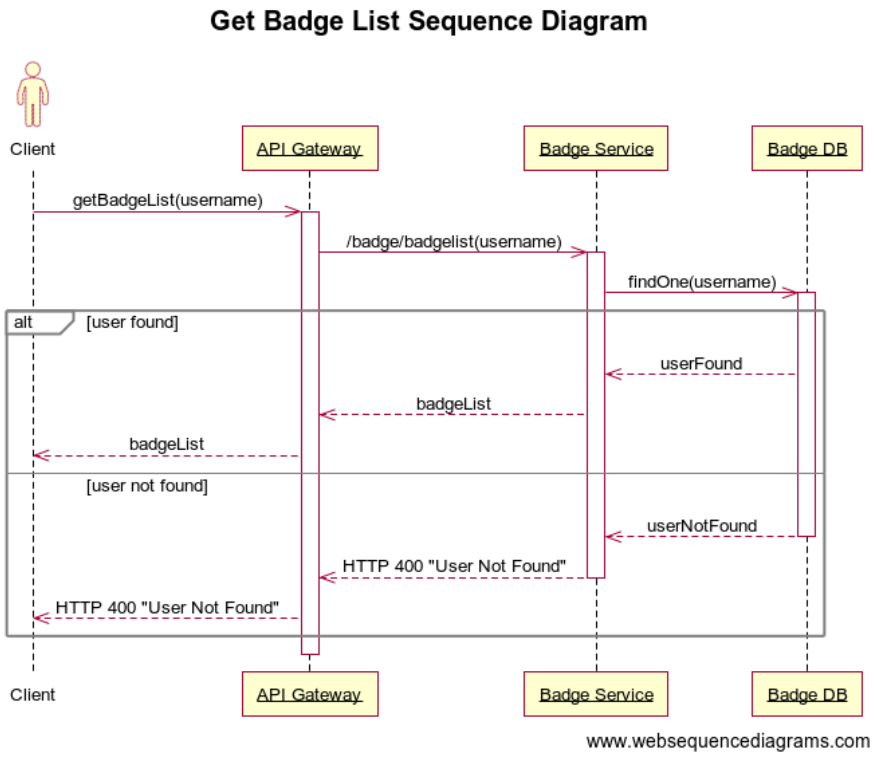


Figura 10.11: Sequence diagram dell'azione di richiesta della badge list di un utente
L'API Gateway alla ricezione della richiesta inoltra al Badge Service il compito di contattare il BadgeDB da cui ricavare la badge list di uno specifico utente. Una volta ottenuta la lista, questa viene mandata come risposta al client.

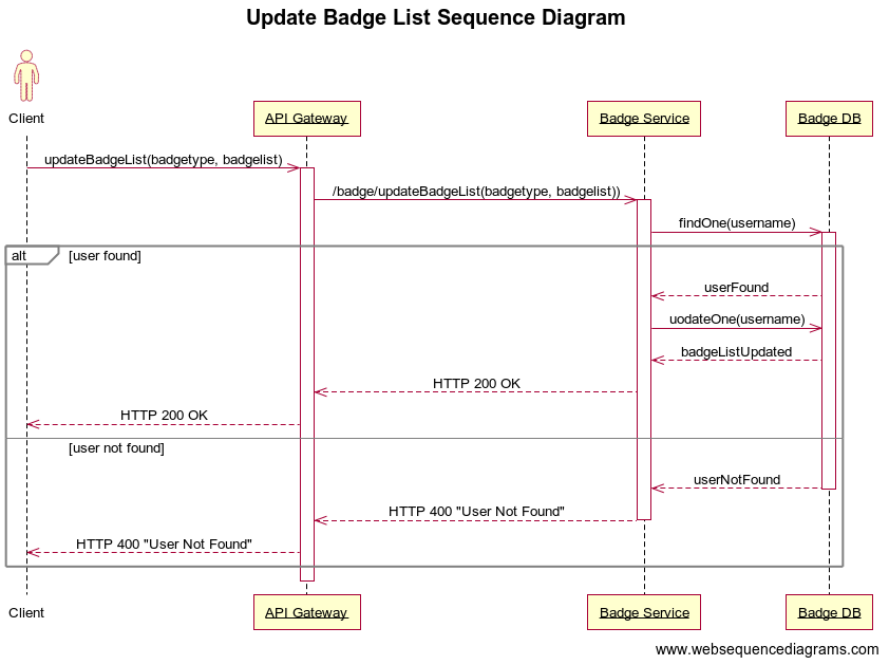


Figura 10.12: Sequence diagram dell'azione di aggiornamento della badge list di un utente
Questa azione viene richiamata quando un utente esegue un'azione che permette di sbloccare un nuovo badge della sua collezione; l'API Gateway richiama il Badge Service, che si occuperà di interagire con il BadgeDB. Viene prima controllata l'esistenza dell'utente, successivamente viene aggiornata la lista di badge.

Bibliografia

[1] Docker site. https://www.docker.com/.

[2] Jest site. https://jestjs.io/.

[3] Moleculer site. https://moleculer.services/.

[4] Mongodb cloud site. https://www.mongodb.com/cloud.

[5] Mongoose site. https://mongoosejs.com/.

[6] Socket.io site. https://socket.io/.