Final Report Global Chat

Oleg Konchenkov
oleg.konchenkov@studio.unibo.it

Giovanni Poggi
giovanni.poggi2@studio.unibo.it

Elia Pasqualini
elia.pasqualini@studio.unibo.it

May 2020

Nell'ambito dei Sistemi Distribuiti si fa largo l'esigenza di poter condividere informazioni sempre più velocemente da vari dispositivi ed a fruizione di tutti. Per questo, i servizi di chat hanno conosciuto negli ultimi anni un'enorme diffusione, grazie alla sua capacità di soddisfare efficacemente il bisogno di comunicazione del popolo di internet. Ad esempio, molti videogiochi o piattaforme online necessitano di chat in cui tutti gli utenti possano inviare e leggere messaggi velocemente. Solo recentemente però esso è entrato a far parte di quella classe di servizi che un'azienda può utilizzare per aumentare la propria produttività incoraggiando il dialogo in tempo reale fra i dipendenti appartenenti a reparti o sedi distaccate. Tale ambito di utilizzo, presuppone un'architettura maggiormente affidabile, che ne garantisca la disponibilità con una certa qualità di servizio rispetto ai modelli diffusi in ambiente Internet. A tal proposito, il presente lavoro si è prefissato l'obiettivo di realizzare un servizio di Chat Globale raggiungibile tramite Applicazione Web o Dispositivi Mobili (Android o iOS), in cui tutti gli utenti collegati possono eseguire l'accesso, assumere dei ruoli precisi all'interno della chat, inviare e leggere messaggi e partecipare alle conversazioni. In particolare, si andranno ad utilizzare i protocolli HTTP, WebSocket, Vert.X e Java. Altre caratteristiche sono state aggiunte in seguito durante la progettazione per aumentare l'affidabilità, l'availability e l'efficienza del servizio.

Contents

List of Figures

1 Goal/Requirements

Come già accennato, il servizio di chat consente ad un gruppo di utenti di scambiare in tempo reale dei messaggi di testo leggibili da tutti gli appartenenti al gruppo. Ogni messaggio, reca ovviamente l'indicazione di chi lo ha generato, il contenuto, il timestamp e viene visualizzato insieme agli altri in un apposito campo testuale a disposizione di ogni utente. Le due principali architetture più utilizzate per la realizzazione di questo servizio sono:

- **Ad anello**: In questo tipo di architettura non esiste un elemento centrale, i client si conoscono tra di loro formando un gruppo con topologia ad anello. Ciò presuppone che siano a disposizione dei client gli indirizzi degli altri, dunque persiste una conoscenza reciproca. In ambiente Internet questo è visto come una breccia dal punto di vista della sicurezza e comporta meccanismi per la gestione dell'anello, introducendo pesanti ritardi nella comunicazione in presenza di un numero elevato di client, a causa dei tempi di propagazione di ogni messaggio.

- **Con server centrale**: rappresenta la tipica modalità client/server, in cui ogni client non deve conoscere gli altri ma solo il server presso cui effettuare la connessione. Ovviamente, tale server costituisce il punto critico dell'architettura per quanto riguarda i possibili guasti, che determinerebbero la non disponibilità del servizio, e può rappresentare (come nel caso dell'anello) un collo di bottiglia. D'altro lato, la presenza di un centro, consente una facile realizzazione dell'atomicità nella consegna dei messaggi.

L'architettura scelta per questo progetto è stata la seconda, cercando di porre rimedio ai problemi sopra presentati attraverso la creazione di un sistema di replicazione a copie, in grado di bilanciare il carico di lavoro ed evitare eventuali congestioni.

Riassumendo, gli obiettivi dello sviluppo di questo progetto sono:

- Realizzazione di una chat in real-time multipiattaforma tramite l'utilizzo delle WebSocket.

- Realizzazione del software basato su un'architettura a Microservizi.

- Comprensione delle tecnologie WebSocket e Vert.X, per la realizzazione di un centro di messaggistica che possa smistare i messaggi da differenti nodi della rete.

- Sviluppo di un'applicazione Java, con supporto Maven ed implementazione / comprensione del framework Vert.X.

- Realizzazione di un server implementato tramite l'architettura Event-Loop di Vert.X per riuscire ad ottenere un'architettura modulare e scalabile.

- Sviluppo di un Client Web che possa connettersi alla Chat Globale e gestire l'invio e la lettura dei messaggi.

- Sviluppo di Client Mobile per permettere all'utente di collegarsi alla Chat Globale ed inviare/ricevere messaggi.
- Autorizzazione ed accesso alla piattaforma tramite l'utilizzo di Token.
- Sviluppo di uno storage che possa garantire la persistenza dei dati inseriti.
- Gestione dei ruoli dei vari utenti connessi alla chat.
- Visualizzazione di dati statistici per il ruolo di "Admin".
- Possibilità di iniziare e concludere una Chat Privata scegliendo tra gli utenti connessi alla Chat Globale.
- Realizzazione di alcuni test per capire se l'architettura funzionerà a dovere.

Qui di seguito, presentiamo un diagramma semplificato dei Casi d'Uso del progetto realizzato:

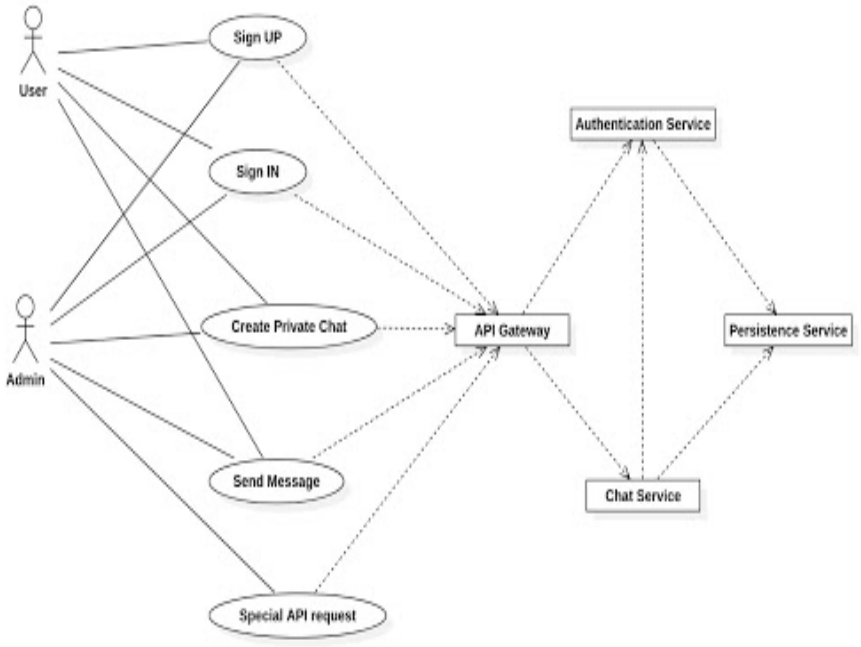

Figure 1: Diagramma dei Casi d'Uso

1.1 Scenarios

La **Figura 2** descrive la struttura base di una chat. Un utente, attraverso l'applicazione client, richiede al server di registrarlo specificando un nickname univoco di sua scelta,

una email ed una password o, in alternativa, potrebbe semplicemente eseguire l'accesso come ospite (Inserendo semplicemente uno Username). In genere, la connessione viene rifiutata se quel nickname è già in uso. Per i dispositivi Client Mobile (Da **Figura 3** a **Figura 6**), comparirà all'utente una schermata che gli permetterà di effettuare il Log In o la Registrazione, successivamente potrà visualizzare la Chat Globale ed interagire con essa.

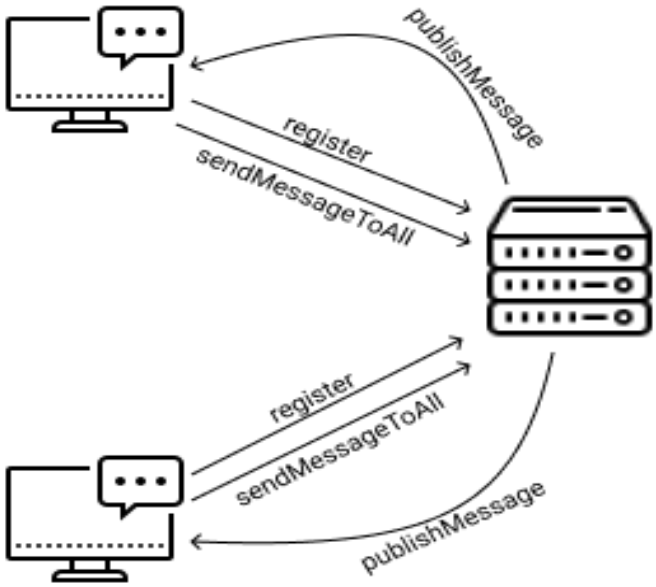

Figure 2: Client-Server Interaction

Una volta connesso, l'utente può iniziare ad inviare messaggi ed il server si preoccuperà di inviarli a tutti i dispositivi Client connessi. Per il recapito dei messaggi, si è optato per la registrazione di un evento che viene scatenato dall'arrivo di un messaggio nel server. Quest'ultimo si occuperà di notificare l'avvenuto cambiamento a tutti i client connessi all'eventbus o alle WebSocket ed aggiornarli.

Per quanto riguarda invece la lista degli utenti collegati online, si tende in genere a procedere all'aggiornamento dei client secondo il meccanismo delle Callback, dato che l'evento di registrazione o cancellazione di un client è molto meno frequente di quello di invio dei messaggi. Ogni volta in cui un utente si connetterà da uno dei tre Client sviluppati dal gruppo (Browser Web, Android e IOS), sarà dunque il server a preoccuparsi di notificare l'evento ai client già collegati, trasmettendo il nickname del nuovo arrivato. Ogni Client riceve solo i messaggi inviati dagli altri Client dal momento della sua registrazione in poi. Inoltre, all'interno della Chat Globale, risulterà possibile creare Chat Private tra utenti semplicemente cliccando sul nome dell'utente con cui si vuol chattare.

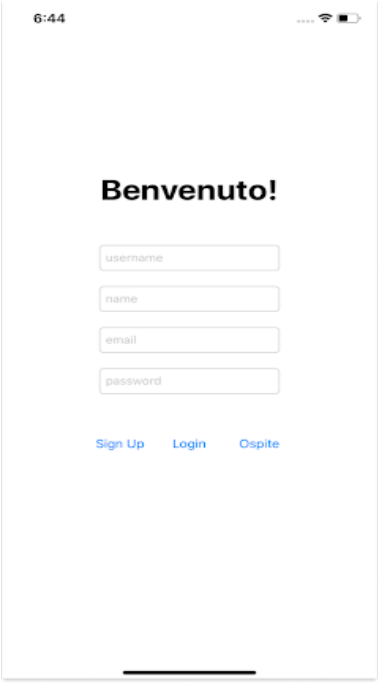

Figure 3: Login da Dispositivo iOS

Figure 4: Visione Chat da Dispositivo iOS

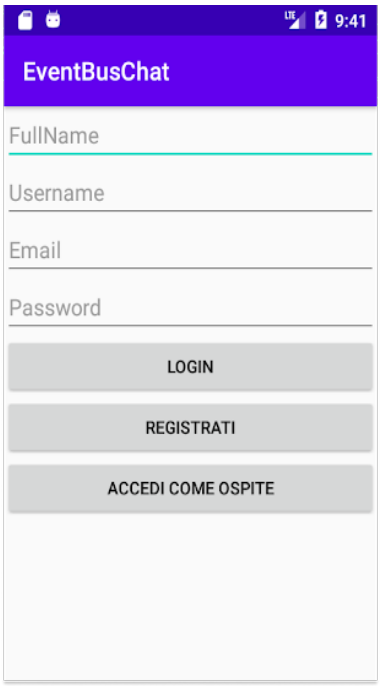

Figure 5: Login da Dispositivo Android

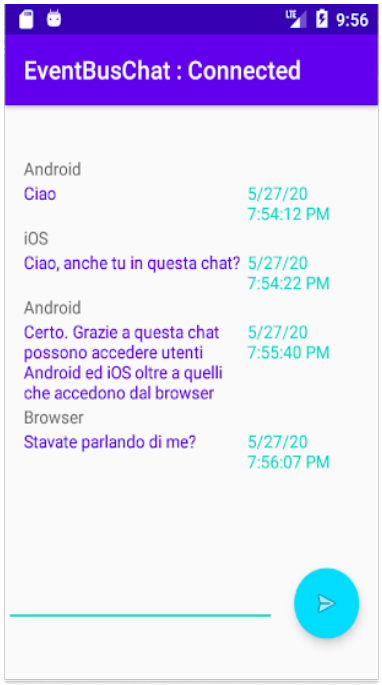

Figure 6: Visione Chat da Dispositivo Android

Figure 7: Login da client web

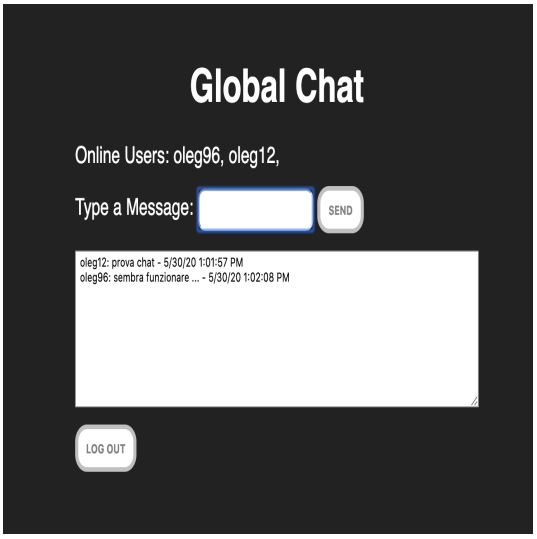

Figure 8: Visione Chat da Dispositivo web (user)

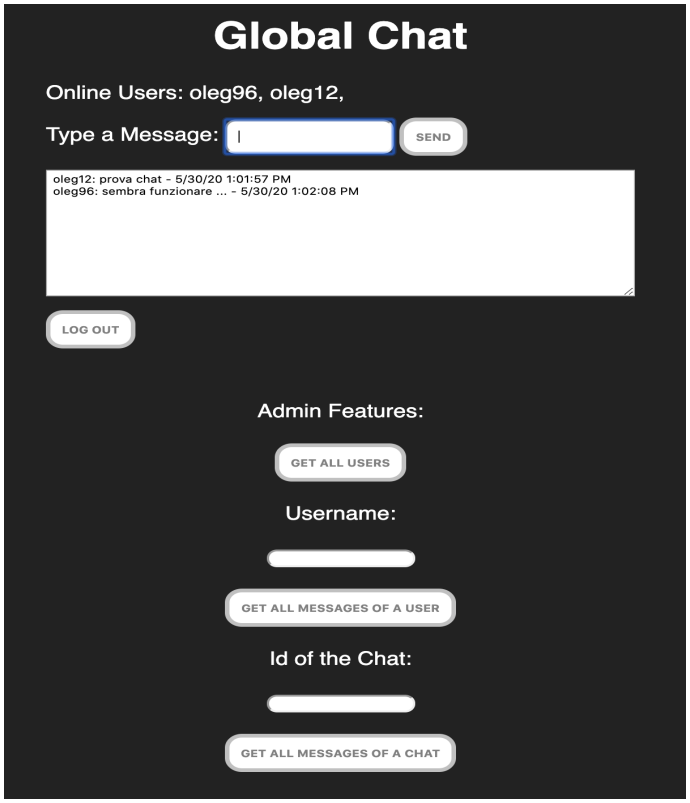

Figure 9: Visione Chat da Dispositivo web (admin)

1.2 Self-assessment policy

Verranno ora esaminate le caratteristiche peculiari del sistema progettato, mirate ad aumentare la disponibilità e le prestazioni del servizio. Il gruppo, inizialmente, per valutare la corretta esecuzione dell'applicazione si è basato sul corretto funzionamento ed interazione dei vari Client creati. In seguito, dato che il progetto è stato sviluppato in linguaggio Java, è stato utilizzato il Framework jUnit 4 per l'esecuzione e la realizzazione dei test. Per verificare il corretto funzionamento dei Microservizi, sono stati creati dei test, tali da cercare coinvolgere tutti i principali componenti. Questi hanno l'obiettivo di verificare il corretto funzionamento delle API REST dei singoli Microservizi e della loro interazione tramite le chiamate a procedura attraverso il servizio di proxy offerto dall'eventbus di Vert.X. La correttezza dei risultati di progetto è quindi verificata e garantita dall'esito positivo dei test. I test portati avanti sono trattati in maniera più approfondita nel **Capitolo 5**. La qualità del software è inoltre garantita dalla scelta dell'architettura a Microservizi che offre maggior modularità, scalabilità e manutenibilità dell'applicazione. Inoltre, sono stati utilizzati paradigmi di OOP per la realizzazione di un codice pulito ed estendibile, supportato da una documentazione in Javadoc il più completa possibile. Infine, per realizzare un software che possa essere eseguito su qualsiasi macchina abbiamo optato per Docker (Analizzato in seguito) come ambiente di esecuzione, supportato da una build dei vari microservizi automatizzati grazie a Maven e da un file di docker-compose per il loro deployment.

2 Requirements Analysis

Come specificato nella sezione precedente, il gruppo ha utilizzato le tecnologie Maven, VertX, SockJS, Redis, Web Socket, MongoDB/MySQL per i Database che immagazzinano dati e Docker. Analizziamo ed osserviamo queste tecnologie qui di seguito:

2.1 Microservizi

I microservizi sono un approccio architetturale alla realizzazione di applicazioni. Quello che distingue l'architettura basata su microservizi dagli approcci monolitici tradizionali è la suddivisione dell'app nelle sue funzioni di base. Ciascuna funzione, denominata servizio, può essere compilata e implementata in modo indipendente. Pertanto, i singoli servizi possono funzionare, o meno, senza compromettere gli altri. Pensa alla tua ultima visita al sito di un retailer online. Probabilmente hai usato la barra di ricerca del sito per sfogliare i prodotti. La ricerca costituisce un servizio. Potresti avere anche ricevuto suggerimenti per i prodotti correlati, che sono basati su un database di preferenze dei clienti. Anche questo è un servizio.

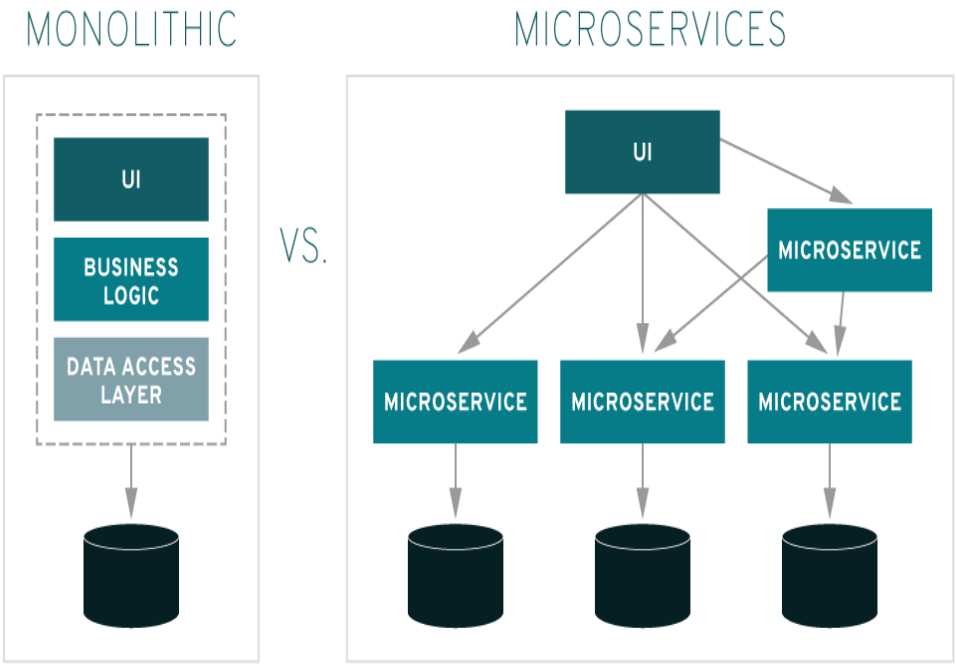

Figure 10: Monolithic vs microserrvice

Pertanto, un microservizio è una funzione di base di un'applicazione, che viene eseguita indipendentemente dagli altri servizi. Tuttavia, l'architettura basata sui microservizi non comporta solo il basso accoppiamento tra le funzioni di base di un'app, ma propone

una ristrutturazione dei team di sviluppo e del framework comunicativo tra i servizi. Tale approccio offre la possibilità di gestire criticità inevitabili, supporta la scalabilità dinamica e agevola l'integrazione di nuove caratteristiche. Per eseguire il deployment dei microservizi e sfruttare i vantaggi di questo approccio, occorre adattare gli elementi di base di un'architettura SOA (Service-Oriented Architecture).

Suddividere un'app nelle relative funzioni base ed evitare le insidie dell'approccio monolitico possono sembrare concetti familiari, perché lo stile architetturale dei microservizi è simile a quello dell'architettura SOA (Service-Oriented Architecture), uno stile di progettazione del software ormai consolidato. Ai primordi dello sviluppo applicativo, anche un cambiamento minimo a un'app esistente imponeva un aggiornamento completo e un ciclo di controllo qualità (QA, Quality Assurance) a sé, che rischiava di rallentare il lavoro di vari team secondari. Tale approccio viene spesso definito "monolitico", perché il codice sorgente dell'intera app veniva compilato in una singola unità di deployment (ad esempio con estensione .war o .ear). Se gli aggiornamenti a una parte dell'app provocavano errori, era necessario disconnettere tutto, fare un passo indietro e correggere. Questo approccio è ancora applicabile alle piccole applicazioni, ma le aziende in crescita non possono permettersi tempi di inattività. Qui entra in gioco l'architettura SOA (Service-Oriented Architecture), in cui le app sono strutturate in servizi riutilizzabili che comunicano fra loro tramite un Enterprise Service Bus (ESB). In questa architettura i singoli servizi sono incentrati su uno specifico processo aziendale e seguono un protocollo di comunicazione, tra cui SOAP, ActiveMQ o Apache Thrift, per essere condivisi attraverso l'ESB. Nel complesso, questa suite di servizi integrati attraverso un ESB costituisce un'applicazione. Inoltre, questo metodo permette di compilare, testare e modificare più servizi contemporaneamente, svincolando i team IT dai cicli di sviluppo monolitici. Tuttavia, poiché l'ESB rappresenta un singolo punto di errore per l'intero sistema, potrebbe comportare un ostacolo per l'intera organizzazione.

2.1.1 Dall'architettura SOA ai microservizi

Qual'è la differenza? I microservizi possono comunicare tra di loro, in genere in modalità stateless, permettendo di realizzare app con una maggiore tolleranza di errore e meno dipendenti da un singolo ESB. Inoltre, comunicano tramite interfacce di programmazione delle applicazioni (API) indipendenti dal linguaggio e ciò consente ai team di sviluppo di scegliere i propri strumenti. Considerando l'evoluzione di SOA, i microservizi non costituiscono una novità assoluta, ma ultimamente sono diventati più appetibili grazie ai progressi delle tecnologie di containerizzazione. Oggi i container Linux permettono di eseguire più parti di un'app in modo indipendente, sullo stesso hardware, con un controllo decisamente superiore sui singoli componenti e cicli di vita. In combinazione con le API e i team DevOps, i microservizi containerizzati rappresentano la base delle applicazioni cloud-native.

2.1.2 I vantaggi di un'architettura basata sui microservizi

Basandosi su un'architettura distribuita, i microservizi consentono di ottenere sviluppo e routine più efficienti. La possibilità di sviluppare più microservizi in contemporanea consente a più sviluppatori di lavorare simultaneamente alla stessa app, riducendo le tempistiche di sviluppo.

2.1.3 Time-to-market più rapido

Consentendo di abbreviare i cicli di sviluppo, un'architettura basata su microservizi supporta deployment e aggiornamenti più agili.

2.1.4 Scalabilità superiore

Man mano che la domanda per determinati servizi aumenta, i microservizi possono essere distribuiti su più server e infrastrutture, in base alle esigenze aziendali.

2.1.5 Resilienza

Ciascun servizio, se costruito correttamente, è indipendente e non influisce sugli altri servizi nell'infrastruttura. Di conseguenza, l'eventuale errore di un componente non determina il blocco dell'intera app, come avviene con il modello monolitico.

2.1.6 Semplicità di deployment

Poiché le app basate su microservizi sono più piccole e modulari delle tradizionali applicazioni monolitiche, tutti i problemi associati a tali deployment vengono automaticamente eliminati. Benché questo approccio richieda un coordinamento superiore, i vantaggi che ne derivano sono determinanti.

2.1.7 Accessibilità

Poiché le app più grandi vengono suddivise in parti più piccole, per gli sviluppatori è molto più semplice comprendere, aggiornare e migliorare tali componenti, e questo permette di accelerare i cicli di sviluppo, soprattutto in combinazione con le metodologie di sviluppo Agile.

2.1.8 Maggiore apertura

Grazie alle API indipendenti dal linguaggio, gli sviluppatori sono liberi di scegliere il linguaggio e la tecnologia ottimali per la funzione da creare.

Oltre a queste, un'architettura basata su microservizi presenta due criticità principali: la complessità e la produttività:

- Compilazione: occorre dedicare tempo all'identificazione delle dipendenze fra i servizi. A causa di tali dipendenze, quando si esegue una compilazione può essere

necessario eseguirne anche molte altre. È fondamentale considerare anche gli effetti prodotti dai microservizi sui tuoi dati.

- Test: i test di integrazione, così come i test end-to-end, possono diventare molto difficili, ma anche più importanti che mai. Occorre ricordarsi che un errore in una parte dell'architettura potrebbe causare un errore in un componente a vari passi di distanza, a seconda del modo in cui i servizi sono strutturati per supportarsi a vicenda.

- Gestione delle versioni: l'aggiornamento a una nuova versione potrebbe compromettere la retrocompatibilità. Il problema può essere gestito attraverso la logica condizionale, ma in breve tempo può diventare difficile da gestire. Disporre di più versioni live per client diversi può essere un'alternativa, la manutenzione e la gestione possono essere molto più laboriose.

- Deployment: durante la configurazione iniziale, anche questa è una fase critica. Per semplificare il deployment, occorre innanzitutto investire notevolmente in soluzioni di automazione. La complessità dei microservizi renderebbe il deployment manuale estremamente difficile. Pensa al modo e all'ordine in cui eseguire il roll-out dei servizi.

- Registrazione: nei sistemi distribuiti, per ricollegare tutti i vari componenti sono necessari registri centralizzati, senza i quali gestirli in modo scalabile risulterebbe impossibile.

- Monitoraggio: è fondamentale per i team operativi avere una visibilità centralizzata del sistema, al fine di identificare l'origine dei problemi.

- Debugging: il debugging remoto non è una scelta praticabile con decine o centinaia di servizi. Al momento non esiste un'unica soluzione legata al debugging.

- Connettività: occorre valutare quale metodo di rilevamento dei servizi scegliere, se centralizzato o integrato.

2.2 Docker

La tecnologia Docker utilizza il kernel di Linux e le sue funzionalità, come Cgroups e namespace, 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. Gli strumenti per la creazione di container, come Docker, consentono il deployment a partire da un'immagine. Ciò semplifica la condivisione di un'applicazione o di un insieme di servizi, con tutte le loro dipendenze, nei vari ambienti. Docker automatizza anche la distribuzione dell'applicazione (o dei processi che compongono un'applicazione) all'interno dell'ambiente containerizzato. Gli strumenti sviluppati partendo dai container Linux, responsabili dell'unicità e della semplicità di utilizzo di Docker, offrono agli utenti

accesso alle applicazioni, la capacità di eseguire un deployment rapido, e il controllo sulla distribuzione di nuove versioni. 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). 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, per un ambiente IT più efficiente.

2.2.1 Iniziare con docker-compose

2.2.2 Osservazioni

Compose è uno strumento per la definizione e l'esecuzione di applicazioni Docker multi-contenitore. Con Compose, si utilizza un file Compose per configurare i servizi dell'applicazione. Quindi, utilizzando un singolo comando, puoi creare e avviare tutti i servizi dalla tua configurazione. L'uso di Compose è fondamentalmente un processo in tre fasi.

- Definisci l'ambiente della tua app con un Dockerfile modo che possa essere riprodotto ovunque.
- Definisci i servizi che compongono la tua app in docker-compose.yml modo che possano essere eseguiti insieme in un ambiente isolato.
- Infine, esegui docker-compose up e Compose inizierà ed eseguirà l'intera app.

2.2.3 Creare una semplice applicazione

Supponiamo di avere un'applicazione python usando redis come backend. Dopo aver scritto Dockerfile , crea un file docker-compose.yml questo modo:

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
    depends_on:
     - redis
  redis:
    image: redis
```

Quindi eseguire docker-compose up configurerà l'intera applicazione include: app python e redis.

- version: '2': è la versione della sintassi del file di composizione docker.

- services: è una sezione che descrive i servizi da eseguire.
- web: e redis: sono i nomi dei servizi da avviare, i loro contenuti descrivono come la
- depends-on: implica una dipendenza del web verso i redis e quindi la docker-compose avvia prima il contenitore redis e quindi il contenitore web.

Tuttavia, la docker-compose non attende che il contenitore redis sia pronto prima di avviare il contenitore web. Per ottenere ciò è necessario utilizzare uno script che ritardi l'avvio del server delle applicazioni o qualsiasi altra cosa fino a quando il contenitore redis può eseguire richieste. Una sezione di volumi e reti può essere aggiunta pure. L'utilizzo della sezione dei volumi consente il volume disconnesso che può essere indipendente dalla sezione Servizi di composizione docker. La sezione delle reti ha un risultato simile. La sezione dei servizi di redis dovrebbe essere aggiustata in questo modo:

```
redis:
  image: redis
  volumes:
    - redis-data:/code
  networks:
    -back-tier
```

Quindi, aggiungere le seguenti sezioni nella parte inferiore del file di versione 2 di composizione docker.

```
volumes:
  # Named volume
  redis-data:
    driver: local
networks:
  back-tier:
    driver: bridge
```

- redis-data: fornisce un'etichetta accessibile dalla sezione dei servizi. driver:local imposta il volume sul file system locale.
- back-tier: imposta l'etichetta della sezione delle reti per essere accessibile nella sezione servizi come ponte.

2.2.4 Docker Compose: Hello World

Un docker-compose.yml molto semplice assomiglia a questo:

```
version: '2'
services:
  hello_world:
    image: ubuntu
    command: [/bin/echo, 'Hello world']
```

Questo file sta facendo in modo che ci sia un servizio hello-world , che è inizializzato da ubuntu:latest immagine e che, quando è in esecuzione, esegue semplicemente echo 'Hello world'. Se si è nella folder della folder (che contiene questo file docker-compose.yml), si potrà eseguire la docker-compose up e si dovrebbe vedere:

```
Creating folder_hello_world_1
Attaching to folder_hello_world_1
hello_world_1 | Hello world
folder_hello_world_1 exited with code 0
```

Questo ha creato il contenitore dall'immagine di ubuntu e ha eseguito il comando specificato su docker-compose.yml. Docker-Compose utilizza il nome della cartella come nome del progetto per prefisso contenitori e reti. Per impostare un altro nome di progetto, puoi chiamare docker-compose –project-name NAME up—down—... oppure suppy un file chiamato .env accanto al tuo docker-compose.yml e scrivi COMPOSE_PROJECT_NAME=name in esso. È meglio evitare nomi di progetto lunghi con trattini (-) perché la finestra mobile componi i bahaves in modo strano con questo tipo di nomi. Nota: docker-compose consente di eseguire più contenitori finestra mobile su un singolo host. Se si desidera eseguire più contenitori su più di un nodo, fare riferimento a soluzione come swarm / kubernetes.

2.3 Maven

Apache Maven è un potente strumento per la gestione dei progetti per le piattaforme Java in termini di compilazione del codice, distribuzione, documentazione e collaborazione del team di sviluppo. I vantaggi principali di Maven sono i seguenti:

- Standardizzazione della struttura del progetto e del processo di compilazione.
- Test ed esportazione automatizzati.
- Gestione e download automatico delle librerie necessarie al progetto con risoluzione delle eventuali dipendenze transitive.
- Facilità di utilizzo dei Plugin anche per ampliare precedenti funzionalità.
- Creazione automatica di un semplice sito per la documentazione del progetto.

I principali componenti di Maven sono:

- pom.xml: file xml di configurazione, contiene tutte le informazioni sul progetto (dipendenze, test, documentazione, etc...).
- Goal: singola funzione che può essere eseguita sul progetto.
- Plug-in: goal riutilizzabili in tutti i progetti.
- Repository: directory strutturata destinata alla gestione delle librerie. Può essere locale o remota.

2.3.1 Pom.xml

POM sta per Project Object Model ed ogni singolo progetto è descritto attraverso questo file di configurazione, senza il quale Maven non può fare niente. Il POM è un file XML e guida l'esecuzione in Maven definendo in modo chiaro l'identità e la struttura di un progetto in ogni suo aspetto. Tutto il progetto è descritto nel POM: informazioni generali, dipendenze, processo di compilazione e fasi secondarie come la generazione di documentazione.

Un POM, di base, è composto da un tag root <project> che conterrà i tag:

- <modelVersion>: dichiara a quale versione di POM questo progetto è conforme (es. 4.0.0).
- <groupId>: ID del gruppo del progetto.
- <artifactId>: ID (dell'artefatto) del progetto.
- <version>: la versione del progetto.
- <packaging>: tipo di archivio che vogliamo esportare (jar, war o ear, come vedremo troveremo in una cartella chiamata "target" l'archivio esportato).

I parametri groupId, artifactId, version e packaging identificheranno univocamente un progetto. Se packaging non è specificato nel POM, assumerà "jar" a valore di default. Ecco un esempio di pom.xml minimale con packaging non specificato:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

Come abbiamo detto, la gestione ed il download automatico delle librerie necessarie per il progetto è uno dei punti di forza di Maven e non è cosa da poco. Supponiamo pertanto di voler includere nel nostro progetto la libreria pippo. Maven di default si collegherà al repository remoto e scaricherà lui tutti i pacchetti di cui necessita, comprese le dipendenze degli stessi. Per inserire questa dipendenza è sufficiente aggiungere nel file pom.xml un tag <dependencies> e dentro questo mettiamo quanto segue:

```
<dependency>
  <groupId>pippo</groupId>
  <artifactId>pippo</artifactId>
  <version>10.9.13</version>
</dependency>
```

Per ogni libreria di cui si necessita (in qualsiasi versione) basterà andare nel sito principale Maven, ricercare la libreria di cui abbiamo bisogno e copia-incollare la stringa <dependency>. Per ogni dipendenza è inoltre possibile definire uno scope:

- Compile: Di default, le dipendenze sono disponibili in tutti i classpath del progetto
- Provided: E' simile a compile, prevede che a runtime le dipendenze siano rese disponibili dall'ambiente di esecuzione (per esempio le JavaEE APIs per un'applicazione enterprise)
- Runtime: Le dipendenze sono richieste solo in esecuzione
- Test: Le dipendenze sono richieste solo per la compilazione e l'esecuzione dei test
- System – Le dipendenze non vengono recuperate tramite repository, vengono esplicitamente dichiarate nella posizione locale

In pratica basterà aggiungere dentro <dependency> il tag <scope>, il cui valore è lo scope desiderato di quella libreria:

```
<dependency>
  <groupId>pippo</groupId>
  <artifactId>pippo</artifactId>
  <version>10.9.13</version>
  <scope>test</scope>
 </dependency>
```

2.3.2 Plugin & Goal

Un goal è una singola funzione che può essere eseguita sul progetto. I Goal possono essere sia specifici per il progetto dove sono inclusi, sia riusabili. I Plugin sono goal riutilizzabili in tutti i progetti. Maven ha una serie di plugin built-in disponibili, i principali sono i seguenti:

- Clean: permette di cancellare i compilati dal progetto
- Compiler: permette di compilare i file sorgenti
- Deploy: permette di depositare il pacchetto generato nel repository remoto
- Install: permette di depositare il pacchetto generato nel repository locale
- Site: permette di generare la documentazione del progetto
- Archetype: permette di generare la struttura di un progetto a partire da un template

Ciascun plugin mette a disposizione dei goal specifici o più di un goal (ad esempio, il plugin di compilazione (compiler) prevede un goal per la compilazione del codice sorgente e un altro per la compilazione dei casi di test). Altri plugin possono essere realizzati qualora sia necessario estendere ulteriormente le capacità di Maven a causa di particolari esigenze. Maven è un tool da riga di comando e i comandi hanno la seguente struttura:

```
mvn <nome-plugin>:[<nome_goal>] [{<parametro>}]
```

Un'altra tipologia di comandi Maven è:

```
mvn [phase]
```

2.3.3 Repository

Il Repository è una directory strutturata destinata alla gestione delle librerie. Maven ne crea uno in locale sotto la home dell'utente al fine di evitare accessi alla rete inutili. Di default, il repository si trova sotto Linux in:

```
/home/${user}/.m2/repository
```

e sotto Windows in:

```
C:\Documents and Settings\${user}\.m2\repository
```

Nel processo di gestione del progetto, Maven verifica in locale l'esistenza della libreria desiderata ed, in caso negativo, interroga un repository remoto. Se nel POM non si specifica alcun repository remoto, eseguendo il Build del progetto, verrà ereditato il Repository di default che corrisponde a `http://repo1.maven.org/maven2/`. Ovviamente, risulta possibile specificare altri repository in cui cercare librerie e dipendenze. Per integrarli all'interno del progetto sarà sufficiente aggiungere nel pom.xml la sezione repositories.

```
<repositories>
  <repository>
    <id>id_nuovo_repository</id>
    <url>http://nuovo_repository</url>
  </repository>
</repositories>
```

2.3.4 Altre Informazioni Utili

- Maven permette inoltre di creare un progetto a partire da un template prestabilito, chiamato archetype. Esso stabilisce la struttura base delle cartelle e dei file che devono essere creati.

- prima volta in cui compiliamo il progetto, Maven scarica tutte le librerie dipendenti necessarie alla compilazione. Questa fase è lunga ed onerosa però le prestazioni non saranno sempre le stesse, le compilazioni successive saranno molto più veloci perchè Maven non avrà più bisogno di scaricare dal repository remoto tutte le librerie necessarie. Quando Maven avrà terminato il processo, visualizzeremo nella cartella target l'esito delle compilazioni, compreso il file jar finale.

Maven permette la suddivisione del programma in moduli. Così facendo, eseguendo un comando Maven sul progetto principale, questo sarà eseguito anche sui suoi moduli.

2.4 Vertx

Vert.x è una piattaforma su cui è possibile rilasciare ed eseguire le proprie applicazioni. È simile, per certi aspetti, a Node.JS ma con una grande differenza: gira in una JVM, è scalabile, concorrente, non bloccante, distribuito e poliglotta. Sviluppato originariamente con il nome di Node.x da Tim Fox, uno sviluppatore di VMware, ha una storia relativamente breve: la prima versione è datata 2011. Per evitare problemi legali con Node.js, il nome è stato modificato nell'attuale Vert.x mantenendo lo stesso significato: non a caso "vertex" è sinonimo di "nodo" in matematica. Attualmente il progetto è mantenuto dalla Eclipse Foundation dopo essere fuoriuscito da WMware nell'agosto 2013. Vert.x utilizza come strato di I/O la libreria Netty: essendo semplicemente un toolkit, si può decidere se utilizzarlo direttamente per lo sviluppo delle proprie applicazioni o integrarlo in applicazioni o framework esistenti. Le funzionalità del core occupano poco spazio sia in termini di codice che di memoria.

2.4.1 Verticles

Un'applicazione Vert.x consiste di uno o più componenti chiamati Verticle. Questi pezzi di codice sono il motore che Vert.x esegue. Ogni verticle viene eseguito in maniera concorrente rispetto agli altri e non c'è uno stato condiviso tra di essi. Tradotto in parole semplici, si riesce a creare applicazioni multi-threaded senza dover gestire problematiche di concorrenza come la sincronizzazione o i lock tra thread.

2.4.2 Tipologie di verticle

Leggendo la documentazione ufficiale possiamo trovare 3 tipi di Verticle:

- Standard verticles: le più semplici, vengono eseguite utilizzando il thread dell'event-loop.
- Worker verticles: vengono eseguiti in thread separati presi da un pool ed ogni istanza non è mai eseguita in concorrenza con altre verticles.
- Multi-threaded worker verticles: sono anch'essi eseguiti su thread del pool ma vengono eseguiti in maniera concorrente con gli altri verticles.

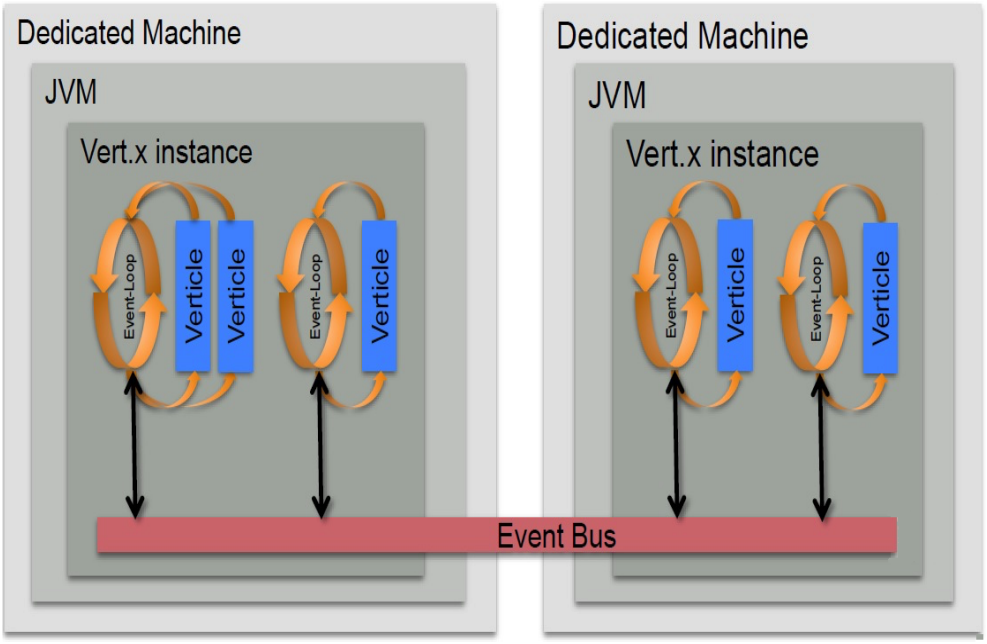

Figure 11: Vertx architecture

2.4.3 Event Bus

L'Event Bus è il cuore di Vert.x. Tutte le comunicazioni tra verticle avvengono tramite questo componente attraverso lo scambio di messaggi. Le verticle possono inviare o

ricevere eventi attraverso il bus con le modalità punto-punto, pub-sub o request-replay. Altra caratteristica importante è che l'event bus è distribuito di default. In pratica, affinché comunichino fra loro, non siamo obbligati a rilasciare le verticles sulla stessa JVM, basta creare un Event Handler e registrarsi su uno specifico indirizzo per utilizzare il bus di eventi. La base dell'implementazione è il pattern Multi Reactor che prevede di creare event loop multipli. Il fatto che una verticle abbia un solo handler collegato ad un indirizzo del bus fa sì che non possa essere eseguito in maniera concorrente rispetto agli altri event loop. Inoltre il numero di event loop viene definito in base al numero di CPU a disposizione, permettendo di scalare molto facilmente in caso di necessità perché basta aumentare il numero di CPU a disposizione.

2.4.4 Blocco o non blocco

Se una verticle deve eseguire molte operazioni di I/O per molto tempo, la coda degli eventi rimane bloccata finché le operazioni non sono terminate. Si presenta pertanto il problema che nessun altro messaggio può essere instradato negli altri verticle. Come abbiamo visto sopra, esistono differenti tipi di verticle. Utilizzando una verticle di tipo worker, possiamo evitare il blocco dell'applicazione mentre attende che una verticle finisca il suo compito. C'è anche la possibilità di eseguire codice bloccante nell'Event Loop senza bloccare tutto.

2.4.5 Poliglotta

Una peculiarità del toolkit è la possibilità di utilizzare linguaggi diversi per creare i singoli Verticle. Attualmente il supporto riguarda i linguaggi:

- Java
- Javascript
- Groovy
- Ruby
- Ceylon
- Scala
- Kotlin

2.4.6 Un esempio

L'implementazione risulta identica per ogni linguaggio ed anche le API sviluppate mantengono per tutti i linguaggi la stessa filosofia. Qui sotto potete vedere una semplice e banale implementazione in linguaggio Java:

```
import io.vertx.core.AbstractVerticle;
public class Server extends AbstractVerticle {
    public void start() {
     vertx.createHttpServer().requestHandler(req -> {
       req.response()
       .putHeader("content-type", "text/plain")
      .end("Hello from Vert.x!");
     }).listen(8080);
  }
}
```

e JavaScript (Utilizzati dentro i sorgenti del progetto):

```
vertx.createHttpServer()
.requestHandler(function (req) {
  req.response()
  .putHeader("content-type", "text/plain")
  .end("Hello from Vert.x!");
}).listen(8080);
```

Si tratta dello stesso server che stampa il classico messaggio di benvenuto. Come possiamo vedere la sintassi è la stessa e questo permette agli sviluppatori di concentrarsi sulle funzionalità e sulla qualità delle del codice, non dovendo imparare uno specifico linguaggio.

2.4.7 Altre componenti del core

Il toolkit è stato pensato per dare alcune facilitazioni ai programmatori. Vediamole in seguito:

JSON
Per quanto riguarda JSON, sono presenti due classi, JsonObject e JsonArray. La prima rappresenta un oggetto come una mappa con chiavi e valori e supporto dei valori nulli, mentre la seconda è una sequenza di valori eventualmente nulli.

Buffers
I buffers sono una sequenza di zero o più byte che possono essere letti o scritti e che si espandono man mano che si scrivono dati al suo interno.

TCP server e client
Vert.x permette di scrivere server o client TCP non bloccanti, ad esempio per implementare un server web o delle API REST, oppure per interrogare servizi esterni. Ovviamente c'è anche il supporto per SSL e HTTP/2.

Logging
C'è la possibilità di utilizzare il logger preferito: da Java Log a Log4J, a SLF4J a LogBack.

Host name resolution
Sono presenti delle classi che permettono di risolvere indirizzi IP non usando le funzionalità di Java e quindi rendendo il tutto non bloccante.

Estensioni aggiuntive
Oltre alla parte core, il toolkit ha una serie di funzionalità aggiuntive che permettono di migliorare i comportamenti base o aggiungere funzionalità, rispettando sempre quella che è la filosofia alla base dello sviluppo di Vert.x.

2.4.8 Vert.x-Web

È stato sviluppato sulla parte core del toolkit prendendo i principi di Express da Node.js e di Sinatra da Ruby. Si possono sviluppare sia applicativi classici server-side sia RESTful e sia applicazioni in real-time con tecnologia push. Alcune caratteristiche di Vert.x-Web sono:

- Routing basato su path o metodi
- Path risolti con regular expression
- Estrazioni di parametri
- Content negotiation
- Gestione dei cookie
- Multipart form e file
- Sessioni
- CORS
- Autenticazione (Basic, JWT)
- Template

2.4.9 Vertx Web Client

Facilita la gestione di chiamate HTTP, con supporto di Json, Request, Response, e form. Integra la possibilità di gestione del polling e di HTTP/2.

2.4.10 Vert.x Data Access

Si tratta di una serie di client creati per l'accesso ai vari DataStore, sempre in maniera asincrona:

- Mongo
- JDBC
- Redis
- SQL Common
- MySQL/PostgreSQL

2.4.11 Integration

C'è anche una serie di librerie aggiuntive per la gestione di SMTP, JCA Adapter, Rabbit MQ, Kafka, AMQP Bridge. Rimandiamo in ogni caso al sito ufficiale per l'elenco delle altre "estensioni" che coprono aspetti di DevOps, custering, sviluppo di microservizi con funzionalità di discovery e configurazione distribuita. Oltre a questi, ci sono "estensioni" che si occupano del testing dei componenti, di servizi come Proxy, SockJs, gRPC e del cloud con Openshift.

Il gruppo vuole inoltre soffermare l'attenzione sulla Concorrenza e la Scalabilità del Framework VertX:

Concorrenza e scalabilità (verticale)
Come detto precedentemente, una verticle gira in un thread suo e non espone direttamente lo stato agli altri. Per riuscire a scalare esiste un flag di Vert.x che indica al sistema quante istanze di verticle verranno instanziate all'avvio. In sostanza, indica quanti thread concorrenti creare per ogni verticle di cui si effettua il deployment. Vert.x si occuperà di utilizzare tutti i thread a disposizione in base al carico, permettendo di far fronte a carichi di richieste crescenti in base alle capacità della macchina in cui gira. Nel caso specifico di una risorsa di rete come potrebbe essere un web server, Vert.x crea un singolo server e usa un load balancer interno per suddividere le varie richieste.

Concorrenza e scalabilità (orizzontale)
Dato che l'Event Bus risulta distribuito, Vert.x offre la possibilità di distribuire le Verticles. Questa possibilità è chiamata Clustering e risulta essere trasparente al codice sviluppato. L'implementazione base è fornita da hazelcast ed è utilizzato per trovare i Verticles. Ovviamente essendo un semplice toolkit, nulla vieta di agganciare il proprio Cluster Manager. Un vantaggio di questa soluzione è che, senza dover gestire nulla di complicato, nel caso di fallimento di un nodo la nostra Verticle viene spostata su un altro nodo del cluster attivo, evitando così il fallimento dell'applicazione.

2.5 Service Discovery

Questo componente fornisce un'infrastruttura per pubblicare e scoprire varie risorse, come proxy di servizio, endpoint HTTP, origini dati ... Queste risorse sono chiamate servizi. Un servizio è una funzionalità rilevabile. Può essere qualificato per tipo, metadati e posizione. Quindi un servizio può essere un database, un proxy di servizio, un endpoint HTTP e qualsiasi altra risorsa che puoi immaginare non appena puoi descriverlo, scoprirlo e interagire con esso. Non deve essere un'entità vert.x, ma può essere qualsiasi cosa. Ogni servizio è descritto da un record.

Il rilevamento del servizio implementa le interazioni definite nel calcolo orientato al servizio. E in una certa misura, fornisce anche le interazioni informatiche dinamiche orientate ai servizi. Pertanto, le applicazioni possono reagire all'arrivo e alla partenza dei servizi.

Un fornitore di servizi può:

- pubblicare un record di servizio

- annullare la pubblicazione di un record pubblicato
- aggiorna lo stato di un servizio pubblicato (inattivo, fuori servizio ...)

Un consumatore di servizi può:

- servizi di ricerca
- associare a un servizio selezionato (ottiene un riferimento di servizio) e utilizzarlo
- rilasciare il servizio una volta che il consumatore ha finito con esso
- ascoltare l'arrivo, la partenza e la modifica dei servizi.

Il consumatore dovrebbe 1) cercare un record di servizio corrispondente alle proprie necessità, 2) recuperare i riferimenti di servizio che danno accesso al servizio, 3) ottenere un oggetto di servizio per accedere al servizio, 4) rilasciare l'oggetto di servizio una volta fatto. Il processo può essere semplificato utilizzando il tipo di servizio in cui è possibile recuperare direttamente l'oggetto del servizio se si conosce da quale tipo è (client JDBC, client Http ...). Come indicato sopra, le informazioni centrali condivise dai fornitori e dai consumatori sono registrazioni. Fornitori e consumatori devono creare la propria istanza di ServiceDiscovery. Queste istanze stanno collaborando in background (struttura distribuita) per mantenere sincronizzato l'insieme di servizi. Il servizio di rilevamento supporta i bridge per importare ed esportare servizi da / verso altre tecnologie di rilevamento.

2.5.1 Utilizzo del Service Discovery

Per utilizzare il rilevamento del servizio Vert.x, aggiungere la seguente dipendenza alla sezione delle dipendenze del descrittore di build (pom.xml):

```
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-service-discovery</artifactId>
    <version>3.9.1</version>
</dependency>
```

Concetti generali

Il meccanismo di scoperta si basa su alcuni concetti spiegati in questa sezione.

Record di servizio

Un record di servizio è un oggetto che descrive un servizio pubblicato da un fornitore di servizi. Contiene un nome, alcuni metadati, un oggetto location (che descrive dove si trova il servizio). Questo record è l'unico oggetto condiviso dal provider (dopo averlo pubblicato) e dal consumatore (recuperarlo durante una ricerca). I metadati e persino il formato della posizione dipendono dal tipo di servizio (vedi sotto). Un record viene pubblicato quando il provider è pronto per essere utilizzato e ritirato quando il provider del servizio è in arresto.

Fornitore di servizi ed editore
Un fornitore di servizi è un'entità che fornisce un servizio. L'editore è responsabile della pubblicazione di un record che descrive il fornitore. Può essere una singola entità (un provider che pubblica se stessa) o un'entità diversa.

Consumatore di servizi
I consumatori di servizi cercano servizi nella scoperta del servizio. Ogni ricerca recupera 0..n Record. Da questi record, un consumatore può recuperare un riferimento di servizio, che rappresenta l'associazione tra il consumatore e il fornitore. Questo riferimento consente al consumatore di recuperare l'oggetto del servizio (per utilizzare il servizio) e di rilasciare il servizio. È importante rilasciare riferimenti di servizio per pulire gli oggetti e aggiornare gli utilizzi del servizio.

Oggetto di servizio
L'oggetto servizio è l'oggetto che dà accesso a un servizio. Può presentarsi in varie forme, ad esempio un proxy, un client e potrebbe anche non esistere per alcuni tipi di servizi. La natura dell'oggetto di servizio dipende dal tipo di servizio. Si noti che a causa della natura poliglotta di Vert.x, l'oggetto del servizio può differire se lo si recupera da Java, Groovy o un'altra lingua.

Tipi di servizio
I servizi sono solo risorse e ci sono molti tipi diversi di servizi. Possono essere servizi funzionali, database, API REST e così via. Il rilevamento del servizio Vert.x ha il concetto di tipi di servizio per gestire questa eterogeneità. Ogni tipo definisce:

- come si trova il servizio (URI, indirizzo bus evento, IP / DNS ...) - posizione
- la natura dell'oggetto di servizio (proxy di servizio, client HTTP, consumer di messaggi ...) - client

Alcuni tipi di servizi sono implementati e forniti dal componente di individuazione del servizio, ma è possibile aggiungerne di propri.

Eventi di servizio
Ogni volta che un fornitore di servizi viene pubblicato o ritirato, un evento viene generato sul bus dell'evento. Questo evento contiene il record che è stato modificato. Inoltre, per tenere traccia di chi sta utilizzando chi, ogni volta che un riferimento viene recuperato con getReference o rilasciato con rilascio, gli eventi vengono emessi sul bus degli eventi per tenere traccia degli utilizzi del servizio.

Backend
Il rilevamento del servizio utilizza una struttura di dati distribuiti Vert.x per archiviare i record. Pertanto, tutti i membri del cluster hanno accesso a tutti i record. Questa è

l'implementazione di backend predefinita. È possibile implementare il proprio implementando SPI ServiceDiscoveryBackend. Ad esempio, forniamo un'implementazione basata su Redis. Si noti che il rilevamento non richiede il clustering Vert.x. Nella modalità a nodo singolo, la struttura è locale. Può essere popolato con ServiceImporter's. Da 3.5.0, è possibile utilizzare una struttura locale anche in modalità cluster impostando la proprietà di sistema 'vertx-service-discovery-backend-local su true (o la variabile d'ambiente VERTX-SERVICE-DISCOVERY-BACKEND-LOCAL su true).

Creazione di un'istanza di individuazione del servizio
Editori e consumatori devono creare la propria istanza di ServiceDiscovery per utilizzare l'infrastruttura di rilevazione:

```
ServiceDiscovery discovery = ServiceDiscovery.create(vertx);

// Customize the configuration
discovery = ServiceDiscovery.create(vertx,
    new ServiceDiscoveryOptions()
        .setAnnounceAddress("service-announce")
        .setName("my-name"));

// Do something...

discovery.close();
```

Per impostazione predefinita, l'indirizzo di annuncio (l'indirizzo del bus degli eventi su cui vengono inviati gli eventi di servizio è: vertx.discovery .announce. È inoltre possibile configurare un nome utilizzato per l'utilizzo del servizio (vedere la sezione sull'uso del servizio). Quando non è più necessario l'oggetto di rilevamento del servizio, non dimenticare di chiuderlo. Chiude i diversi importatori ed esportatori di rilevamento configurati e rilascia i riferimenti di servizio. È necessario evitare di condividere l'istanza di individuazione del servizio, pertanto l'utilizzo del servizio rappresenterebbe gli "usi" corretti.

Publishing services Una volta che hai un'istanza di rilevamento del servizio, puoi pubblicare servizi. Il processo è il seguente:

- creare un record per un determinato fornitore di servizi
- pubblica questo record
- conservare il record pubblicato utilizzato per annullare la pubblicazione di un servizio o modificarlo.

Per creare record, è possibile utilizzare la classe Record o utilizzare metodi convenienti dai tipi di servizio.

```
Record record = new Record()
    .setType("eventbus-service-proxy")
    .setLocation(new JsonObject().put("endpoint", "the-service-address"))
    .setName("my-service")
    .setMetadata(new JsonObject().put("some-label", "some-value"));

discovery.publish(record, ar -> {
  if (ar.succeeded()) {
    // publication succeeded
    Record publishedRecord = ar.result();
  } else {
```

```
    // publication failed
  }
});

// Record creation from a type
record = HttpEndpoint.createRecord("some-rest-api", "localhost", 8080, "/api");
discovery.publish(record, ar -> {
  if (ar.succeeded()) {
    // publication succeeded
    Record publishedRecord = ar.result();
  } else {
    // publication failed
  }
});
```

È importante mantenere un riferimento sui record restituiti, poiché questo record è stato esteso da un ID di registrazione.

Tipi di servizi
Detto sopra, la scoperta del servizio ha il concetto del tipo di servizio per gestire l'eterogeneità dei diversi servizi. Questi tipi sono forniti per impostazione predefinita:

- HttpEndpoint: per l'API REST, l'oggetto del servizio è un HttpClient configurato sull'host e sulla porta (la posizione è l'URL).
- EventBusService: per i proxy di servizio, l'oggetto del servizio è un proxy. Il suo tipo è l'interfaccia proxy (la posizione è l'indirizzo).
- MessageSource: per le origini dei messaggi (editore), l'oggetto del servizio è un MessageConsumer (la posizione è l'indirizzo).
- JDBCDataSource: per le origini dati JDBC, l'oggetto del servizio è un client JDBCC (la configurazione del client viene calcolata dalla posizione, dai metadati e dalla configurazione del consumatore).
- RedisDataSource: per le origini dati Redis, l'oggetto del servizio è un RedisClient (la configurazione del client viene calcolata dalla posizione, dai metadati e dalla configurazione del consumatore).
- MongoDataSource: per le origini dati Mongo, l'oggetto del servizio è MongoClient (la configurazione del client viene calcolata dalla posizione, dai metadati e dalla configurazione del consumatore).

Questa sezione fornisce dettagli sui tipi di servizio in generale e descrive come utilizzare i tipi di servizio predefiniti.

Servizi senza tipo
Alcuni record potrebbero non avere alcun tipo (ServiceType.UNKNOWN). Non è possibile recuperare un riferimento per questi record, ma è possibile creare i dettagli della connessione dalla posizione e dai metadati del Record. L'uso di questi servizi non genera eventi di utilizzo del servizio.

2.6 TCPBridge to vertx eventbus

Per utilizzare questa dipendenza di VertX per il TCP Bridge dell'EventBus, bisogna aggiungere alle dipendenze Maven (pom.xml):

```
<dependency>
 <groupId>io.vertx</groupId>
 <artifactId>vertx-tcp-eventbus-bridge</artifactId>
 <version>3.9.1</version>
</dependency>
```

L'EventBus di TCP Bridge è situato sopra TCP, ciò significa che le applicazioni possono creare socket TCP per interagire con un'istanza remota VertX via Event Bus. Il caso d'uso principale per il bridge TCP rispetto al bridge SockJS è per le applicazioni che sono più vincolate alle risorse e che devono essere leggere poiché l'intero HTTP WebSocket viene sostituito con normali socket TCP. Resta ovviamente utile anche per le applicazioni che non hanno vincoli di risorse rigidi: il protocollo è abbastanza semplice da fornire in modo efficiente un'interfaccia di integrazione con applicazioni non JVM. Il protocollo è stato mantenuto il più semplice possibile e le comunicazioni utilizzano i frame in entrambi i modi. La struttura di un Frame è simile alla seguente:

```
<Length: uInt32><{
  type: String,
  address: String,
  (replyAddress: String)?,
  headers: JsonObject,
  body: JsonObject
}: JsonObject>
```

Il messaggio è costituito da un documento JSON che può essere stato minimizzato o meno. Il messaggio deve essere preceduto da un grande endian 32 bit positivo intero (4 byte) che indica l'intera lunghezza del documento JSON, in byte. Il tipo di messaggio può essere il seguente per i messaggi inviati dal client TCP:

- Send: per inviare un messaggio ad un indirizzo.
- Publish: per pubblicare un messaggio ad un indirizzo.
- Register: per iscriverti ai messaggi inviati o pubblicati ad un indirizzo.
- Unsubscribe: per annullare l'iscrizione ai messaggi inviati o pubblicati ad un indirizzo.
- Ping: per inviare una richiesta ping al bridge.

Si noti che il campo replyAddress è facoltativo e può essere utilizzato solo per un messaggio di invio. Un messaggio con quel campo dovrebbe eventualmente ricevere un messaggio dal server il cui campo indirizzo sarà quello del valore replyAddress originale. Il server invia messaggi al client e possono essere del seguente tipo:

- Message: per i messaggi inviati o pubblicati ad un indirizzo.
- Err: per indicare un errore (il corpo deve contenere i dettagli dell'errore).
- Pong: per rispondere alla richiesta di ping inviata dal client.

Un client NodeJS di esempio è disponibile nell'origine del progetto. Questo client utilizza la stessa API della controparte SockJS, pertanto dovrebbe facilitare il passaggio tra le implementazioni TCP e SockJS. Un esempio su come iniziare con TCPbridge potrebbe essere:

```
TcpEventBusBridge bridge = TcpEventBusBridge.create(
    vertx,
    new BridgeOptions()
        .addInboundPermitted(new PermittedOptions().setAddress("in"))
        .addOutboundPermitted(new PermittedOptions().setAddress("out")));

bridge.listen(7000, res -> {
  if (res.succeeded()) {
    // succeed...
  } else {
    // fail...
  }
});
```

2.7 RestAPI

Uno degli argomenti caldi dell'ultimo anno è stato il Cloud Computing: sistemi distribuiti ed eterogenei possono collaborare nella realizzazione di una qualche funzionalità di business. In questo contesto assume sempre maggiore importanza il concetto di servizio che, in molti ma non in tutti i casi, si traduce tecnologicamente in quella che chiamiamo Web Services. Naturalmente associamo i Web Service a tecnologie ben precise: SOAP, XML, HTTP e via dicendo. In ogni caso, in tale ottica, restano fondamentali alcuni punti fissi per chiunque intenda fornire un servizio:

- Il proprio servizio può essere acceduto da moltissime applicazioni, anche sconosciute, che ne vogliono utilizzare le funzionalità
- La funzionalità realizzata non dovrà mai dipendere da quella del servizio che ne usufruirà

In una architettura di questo tipo alcuni requisiti diventano cruciali: per esempio la scalabilità, il modo in cui vengono rappresentate le funzioni e i dati forniti. Tutto questo deve essere reso più semplice possibile, questa è la ragione per cui è nato REST.

2.7.1 Che cosa è REST?

Representational State Transfer (REST) è uno stile di architettura software per sistemi ipermediali distribuiti. Un sistema conforme ai design principle REST è detto RESTful. Il Web è l'esempio più importante e più famoso di un sistema distribuito che realizza i principi REST. Il padre di REST, Roy Fielding, introdusse per la prima volta l'espressione Representational State Transfer nella sua tesi di dottorato per indicare le caratteristiche delle parti meglio progettate del Web. Il termine REST è diventato più comune negli ultimi anni con l'esplosione del Web 2.0 e il proliferare di servizi web. Con la popolarità, il termine REST ha assunto anche significati diversi e scorretti fino a diventare in molti casi una buzzword. In poche parole, un'architettura RESTful è di tipo client/server. I client fanno richieste ai server che a loro volta processano le richieste

e restituiscono una risposta. Richieste e risposte contengono delle rappresentazioni di risorse. Una risorsa è un concetto significativo per il dominio di riferimento (Ad esempio l'articolo di un blog o il profilo utente di un social network) che può essere identificato attraverso un indirizzo. Una rappresentazione è una descrizione dello stato corrente o voluto di una risorsa.

2.7.2 I vincoli dell'architettura

Lo stile architetturale REST descrive 6 vincoli principali che devono essere rispettati dal sistema affinché possa essere definito RESTful. Poiché REST è uno stile e non una tecnologia e nemmeno una lista di raccomandazioni tecniche, i dettagli su come questi vincoli debbano essere implementati sono lasciati ai singoli sviluppatori. Ad esempio, l'Application Layer sul quale costruire il sistema non deve essere per forza HTTP, anche se HTTP è quasi sempre la scelta più naturale. Il formato delle rappresentazioni delle risorse spesso è XML o JSON, ma potrebbe essere anche PDF o un documento Microsoft Word. REST non è uno stile architetturale solo per le web application, è molto più generale. Se i vincoli REST sono rispettati, l'architettura software di un robot (server) controllato da uno o più computer (client) dove le risorse sono l'hardware del robot (sensori e motori) può essere a tutti gli effetti considerata RESTful. L'esperienza ha mostrato che sistemi RESTful hanno delle desiderabili proprietà emergenti, ad esempio la scalabilità e la modificabilità. Anche la semplicità è una caratteristica desiderata in un'architettura RESTful.

2.8 API Gateway

Strumento indispensabile per moltissimi sviluppatori, le Application Programming Interface sono la chiave d'accesso per realizzare architetture scalabili e manutenibili. È il modo con cui ci integriamo abitualmente ad applicazioni e piattaforme di vario tipo e con cui permettiamo a terze parti di espandere le funzionalità dei nostri software. Per molte aziende le api sono diventate un asset aziendale, con un valore tale da poter essere vendute. Ad esempio, pensa a Google maps. Integrare all'interno di un sito web la mappa di Google, e geolocalizzare la tua attività, è semplicissimo proprio grazie alle API, attraverso le quali lo sviluppatore può interagire con il programma. Qui di seguito, cercheremo di capire come valorizzare le nostre API e gestirle in maniera efficiente, utilizzando uno strumento aziendale: l'API Gateway.

2.8.1 Cos'è l'API gateway?

L'API gateway è lo strumento che ci permette di intermediare i servizi (o microservizi) da esporre in maniera strutturata verso l'esterno. Non c'è nulla di fantascientifico in un API-gateway: si tratta di applicazioni web molto sofisticate che agiscono da proxy rispetto alle interrogazioni dei client, monitorando e gestendo il traffico di chiamate. I migliori prodotti sul mercato sono disponibili in versione Saas (software come servizio) o "on premise", generalmente open source. Attraverso l'API-gateway è possibile esporre a

terze parti le proprie API sviluppate internamente (magari solo una parte, o un accorpamento di più set diversi) gestendo in maniera efficiente l'autenticazione e monitorandone l'utilizzo. Lo sviluppatore che desidera utilizzare le API può accedere ad un'area riservata per consultare la documentazione, oltre che a scaricare i contratti in formato Open API swagger o API blueprint. Ciascuna chiamata è collegata ad un account e quindi possiamo misurare il numero di chiamate fatte, inserire dei limiti per evitare abusi oppure fatturare in base all'utilizzo.

2.8.2 L'API gateway all'interno dell'IT

L'API gateway è importante anche nel caso in cui le API servono internamente all'IT. Infatti nella maggior parte delle situazioni, anche all'interno dell'IT abbiamo una moltitudine di servizi chiamati "punto-punto". Ogni servizio implementa il proprio sistema di logging e l'interazione è generalmente un sistema punto-punto. Ciascun applicativo interagisce con tutti gli altri, rendendo difficile il monitoraggio e la condivisione di documentazione. Anche in questo contesto l'API gateway è una soluzione utile perché permette di:

- centralizzare il punto di ingresso per le chiamate.
- monitorare le risorse utilizzate.
- gestire in maniera efficiente i log.
- applicare politiche di throttling efficienti.

Inoltre, in alcuni scenari fuori standard è un utile paracadute. Ad esempio possiamo:

- proteggere un servizio che è aperto a tutti.
- includere un sistema di tracing su un servizio che ne è sprovvisto.
- aggiungere campi fissi su alcune chiamate, astraendo l'utilizzo al chiamante.

2.8.3 Quali sono le caratteristiche di un API-gateway?

Sistema di monitoring e analisi
L'API gateway deve essere in grado di registrare ogni singola chiamata, evidenziare ciò che è andato in errore. Inoltre, le attività monitorate devono essere segmentate per applicativo o per insieme di applicativi. Infine, non necessaria, ma utile, la possibilità di evidenziare tramite grafici l'utilizzo delle risorse.

API Developer Portal
Uno degli obiettivi dell'API gateway è permettere ad ogni utilizzatore di essere indipendente ed autonomo nell'utilizzo delle API. Questo non significa soltanto riuscire ad effettuare le chiamate ma soprattutto mettere a disposizione la documentazione e gli strumenti per capire come funzionano i servizi. Deve essere possibile caricare i contratti

swagger (Open API) o in altro formato ma soprattutto pubblicare la documentazione e le informazioni utili per gli sviluppatori.

DevOps Oriented
A prescindere dai vincoli tecnologici, il sistema scelto deve essere facile da rilasciare (deployment) e da gestire. Ormai tutti gli applicativi installabili on-premise supportano Docker, il che li rende molto facili da gestire. Tuttavia, in base alla propria esperienza e all'infrastruttura che si ha in azienda è importante sincerarsi che il prodotto scelto sia compatibile.

Quota e limiti sulle chiamate
Il sistema API gateway deve essere in grado di impostare limiti sulle chiamate (es. max 1000 richieste per ora per API key). L'ideale sarebbe avere la possibilità di applicare tali limitazioni per user/API key o per singolo endpoint. Inoltre, nel caso in cui si voglia monetizzare o si pensi di volerlo fare in futuro, la possibilità di fatturare le chiamate.

Autenticazione
Questo è il punto più delicato di tutto il processo. La criticità nasce dal requisito che tutti i microservizi hanno necessità di conoscere l'identità del chiamante, così come l'API gateway. I microservizi, infatti, devono sapere chi chiama in modo da profilare i dati di conseguenza (ad esempio per dare a Mario Rossi l'elenco dei suoi clienti). Anche l'API gateway ha bisogno di sapere che una determinata API key è collegata ad uno specifico utente per conteggiare le chiamate e far comparire all'interno del suo profilo utente le statistiche di utilizzo. Per questo occorre appoggiarsi ad un protocollo di autenticazione standard (es. jwt o oauh2) ed un identity server dedicato. Il passo successivo è far svolgere all'identity server il ruolo di identity broker, che consente di propagare i token di accesso verso gli applicativi, tracciando le chiamate in maniera ottimale. Questa configurazione non è fuori dagli schemi nella maggior parte dei casi, ma è importante che la soluzione scelta sia compatibile con questa configurazione, anche con un effort lato configurazione.

Mock API
In fase di sviluppo è importante avere una piattaforma che permetta di caricare file swagger (interfacce OpenAPI) o API blueprints per costruire servizi mock (un sistema per simulare le API). Questo non è un requisito per l'utilizzo a regime, ma aiuta ogni qual volta dobbiamo creare applicativi nuovi. Possiamo, ad esempio, creare un utente "dev" su cui carichiamo i servizi fake. Così sblocchiamo il fornitore che può procedere con lo sviluppo dell'applicativo intanto che i nostri sviluppatori interni realizzano i servizi.

Notifiche ed eventi
Avere un sistema perfettamente funzionante ma isolato non è il massimo. Ci piacerebbe avere la possibilità di ricevere notifiche al verificarsi di determinati eventi. Il modo in cui ci aspettiamo di avere questa feature è tramite webhook. Questo meccanismo, letteralmente uncino del web, ci permette di sviluppare un nostro applicativo e integrarci con

i sistemi pre-esistenti. Ad esempio possiamo inserire una riga di log sul nostro sistema ELK al verificarsi dell'errore o inserire un ticket su CRM quando un utilizzatore sfora la quota per operazioni commerciali.

Trasformazioni
Non è il motivo per cui va adottato un API-gateway, ma poter alterare richieste e risposte dei servizi per adattarle tra i vari formati è un vantaggio da tenere in considerazioni. In ogni caso non aspettatevi di poter fare cose troppo elaborate, il caso d'uso tipico è inserire parametri costanti, aggiungere header, convertire i payload da json (JavaScript Object Notation) a xml.

Un esempio pratico
Dopo questa veloce panoramica sulle API gateway proviamo a mettere insieme un esempio pratico per valutare un caso d'uso reale. Ad esempio:

- API gateway: tyk
- Identity server: redhat keycloak
- App backend: Utilizziamo un cms headless, cockpit! Si tratta di un cms molto semplice che ci permette di mettere in luce diversi casi d'uso.
- docker: utilizzato per far girare in locale tutti gli applicativi

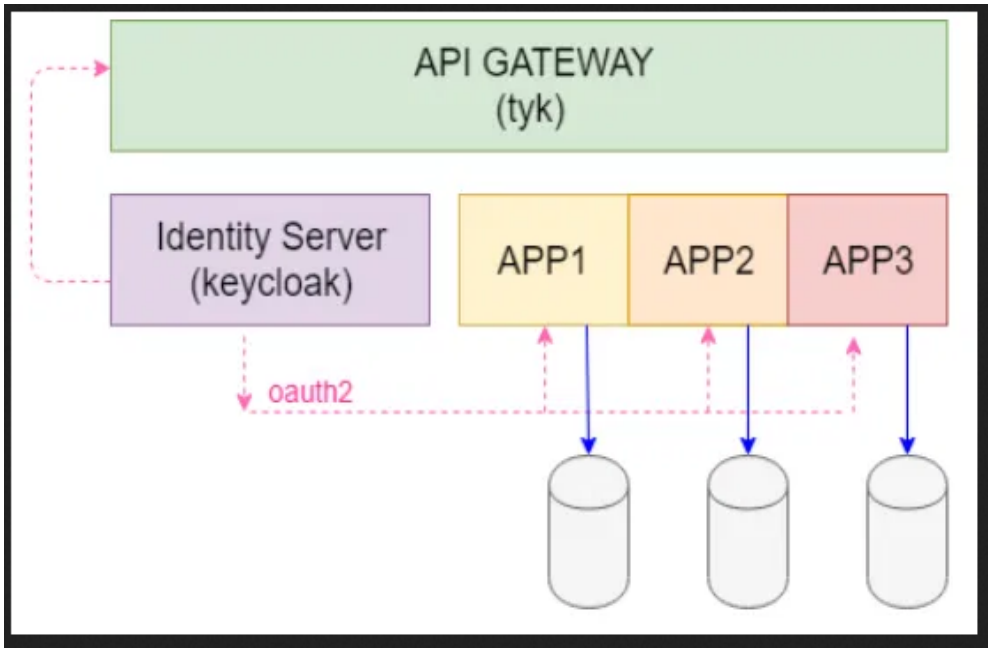

Figure 12: API Gateway example

2.9 Service Proxy

Un proxy è un "server intermediario", un computer che si posiziona tra un client (utente che naviga) ed un sito/pagina web che vogliamo visitare (ospitati in un server), facendo da tramite tra i due. Il procedimento è il seguente:

- L'utente (client) si collega al proxy e gli invia le richieste.
- Il proxy si collega al server ospitante il sito web e gli inoltra la richiesta dell'utente.
- Ricevuta la risposta, il proxy manda la risposta al client.

In pratica, non siamo più connessi direttamente al server del sito che visitiamo ma passiamo attraverso questo filtro chiamato proxy sia in entrata che in uscita. L'utilizzo di un server proxy, nei casi sopra citati, influenza positivamente la sicurezza informatica dell'utente internet. L'indirizzo IP del computer o della rete da cui l'utente naviga non comparirà mai direttamente nel corso della navigazione, risulterà visibile soltanto quello associato al server proxy. Inoltre un proxy velocizza la navigazione in quanto i siti visitati recentemente da qualunque utente, vengono memorizzati nel proxy che in questo modo evita di scaricarli nuovamente e possono essere ricevuti alla massima velocità permessa dal proprio router/modem.

2.9.1 Service Proxy sull'EventBus

Quando componi un'applicazione vert.x, potresti voler isolare una funzionalità da qualche parte e renderla disponibile per il resto dell'applicazione. Questo è lo scopo principale dei Service Proxy. Consentono di esporre un servizio sul bus degli eventi, quindi essere utilizzato da un qualsiasi componente vert.x non appena verrà a conoscenza dell'indirizzo su cui è stato pubblicato il servizio. Questa operazione è ottenuta dai Service Proxy di vert.x. Tuttavia, i client di servizio vert.x, formano i Service Proxy base di vert.x, non possono essere utilizzati dal browser. Vert.x SockJS Service Proxy genera client di servizio che è possibile utilizzare dal browser. Questi client si basano sul bridge SockJS che propaga gli eventi dal bus degli eventi vert.x a SockJS e viceversa.

2.9.2 Utilizzo di Vert.x SockJS Service Proxy

Per utilizzare Vert.x SockJS Service Proxy, aggiungere le seguenti dipendenze alla sezione delle dipendenze del descrittore di build nel pom.xml di Maven:

```
<dependency>
 <groupId>io.vertx</groupId>
 <artifactId>vertx-sockjs-service-proxy</artifactId>
 <version>3.9.1</version>
</dependency>
<dependency>
 <groupId>io.vertx</groupId>
 <artifactId>vertx-service-proxy</artifactId>
 <version>${vertx.version}</version>
</dependency>
```

Tenere presente che, poiché il meccanismo del Service Proxy si basa sulla generazione del codice, le modifiche all'interfaccia del servizio richiedono la ricompilazione delle fonti per rigenerare il codice.

2.9.3 Utilizzo del servizio da un browser

E' possibile utilizzare il servizio direttamente dal browser utilizzando un proxy basato su SockJS. Innanzitutto, è necessario configurare il bridge SockJS per consentire al proxy di comunicare con il servizio:

```
SomeDatabaseService service = new SomeDatabaseServiceImpl();
ProxyHelper.registerService(SomeDatabaseService.class, vertx, service,
    "database-service-address");


Router router = Router.router(vertx);
// Allow events for the designated addresses in/out of the event bus bridge
SockJSBridgeOptions opts = new SockJSBridgeOptions()
    .addInboundPermitted(new PermittedOptions()
        .setAddress("database-service-address"))
    .addOutboundPermitted(new PermittedOptions()
        .setAddress("database-service-address"));

// Create the event bus bridge and add it to the router.
router.mountSubRouter("/eventbus", SockJSHandler.create(vertx).bridge(opts));

vertx.createHttpServer().requestHandler(router).listen(8080);
```

Una volta configurato il bridge SockJS, altre applicazioni sviluppate in JavaScript possono interagire direttamente con il tuo servizio. Durante la compilazione del servizio, viene generato un modulo proxy JS, che viene chiamato come segue: nome-modulo-js / nome-simple-server-server + -proxy.js. Ad esempio, se la tua interfaccia si chiama MyService, il modulo proxy si chiamerà my-service-proxy.js. Questo proxy è utilizzabile dal tuo browser. Il proxy generato è un modulo JavaScript compatibile con CommonJS, AMD e Webpack. È quindi necessario creare un'istanza del proxy con il bridge EventBus e l'indirizzo EventBus del servizio:

```
<script src="http://cdn.sockjs.org/sockjs-0.3.4.min.js"></script>
<script
        src="https://cdnjs.cloudflare.com/ajax/libs/vertx/3.4.2/vertx-eventbus.min.js">
</script>
<!-- This is your generated service proxy -->
<script src="vertx-database-js/some_database_service-proxy.js"></script>
<script>
 var eb = new EventBus('http://localhost:8080/eventbus');
 eb.onopen = function() {
   var svc = new SomeDatabaseService(eb, "database-service-address");
   // use the service
 };
</script>
```

2.10 SockJS

SockJS è una libreria JavaScript del browser che fornisce un oggetto simile alle WebSocket. SockJS offre un'API Javascript coerente, cross-browser, che crea un canale di comunicazione tra domini, full duplex e bassa latenza tra il browser e il server web. SockJS tenta di utilizzare prima di tutto i WebSocket nativi. In caso non ci riesca, può utilizzare una varietà di protocolli di trasporto specifici del browser e li presenta attraverso astrazioni simili a WebSocket. SockJS è progettato per funzionare con tutti i browser moderni e in ambienti che non supportano il protocollo WebSocket, ad esempio dietro proxy aziendali restrittivi. Il client SockJS richiede una controparte del server:

- SockJS-node è un server SockJS per Node.js.

2.10.1 Filosofia

- Le API seguono strettamente le API Websocket HTML5.
- Tutti i trasporti devono supportare connessioni interdominio predefinite. È possibile e consigliato ospitare un server SockJS in un server diverso rispetto al sito Web principale.
- Esiste il supporto per almeno un protocollo di streaming per ogni browser principale.
- I trasporti in streaming dovrebbero funzionare su più domini e dovrebbero supportare i cookie (per sessioni adesive basate su cookie).
- I trasporti polling vengono utilizzati come fallback per vecchi browser e host dietro proxy restrittivi.
- Lo stabilimento di connessione dovrebbe essere veloce e leggero.
- Nessun Flash all'interno (non è necessario aprire la porta 843, che non funziona tramite proxy, non è necessario ospitare "crossdomain.xml" e non è necessario attendere 3 secondi per rilevare i problemi)

2.11 Redis

2.11.1 Che cos'è Redis?

Redis è un datastore in memoria rapido, open source e di tipo chiave-valore per l'utilizzo con database, caching, broker di messaggistica e code. Il progetto è nato quando ci fu la necessità di migliorare la scalabilità di un servizio. Redis fornisce ad oggi tempi di risposta inferiori al millisecondo, consentendo milioni di richieste al secondo per applicazioni in tempo reale nei campi di videogiochi, tecnologie pubblicitarie, servizi finanziari, settore sanitario e IoT. Redis è una scelta popolare per caching, gestione di sessioni, videogiochi, graduatorie, analisi in tempo reale, geospazialità, ride-hailing, chat/messaggistica, streaming multimediale e app pub/sub.

2.11.2 Come funziona Redis?

Tutti i dati Redis risiedono in-memory, a differenza dei database che memorizzano i dati su disco o SSD. Eliminando l'esigenza di accedere ai dischi, i datastore in-memory come Redis evitano i ritardi dovuti ai tempi di ricerca e sono in grado di accedere ai dati in pochi microsecondi. Redis presenta versatili strutture dei dati, elevata disponibilità, geospazialità, transazioni, persistenza su disco e supporto di cluster, facilitando la creazione in tempo reale di app in scala Internet.

2.11.3 Vantaggi di Redis

Datastore in memoria
Tutti i dati di Redis si trovano nella memoria principale del server, a differenza di quanto avviene nei database come PostgreSQL, Cassandra, MongoDB e altri, che memorizzano la maggior parte dei dati su disco o su SSD. Nei database tradizionali basati su disco, i dati vengono trasferiti da e verso lo storage per ogni operazione, mentre i datastore in memoria come Redis non prevedono questa operazione. Pertanto, questo consente loro di supportare una quantità di operazioni di ordine di grandezza superiore e tempi di risposta più rapidi. Il risultato è costituito da prestazioni incredibilmente elevate con tempi medi di lettura e scrittura inferiori al millisecondo e supporto per milioni di operazioni al secondo.

Strutture dati flessibili
A differenza dei semplicistici datastore chiave-valore che offrono strutture dati limitate, Redis offre strutture dati adatte a diverse applicazioni. I tipi di dati Redis includono:

- **String** (stringhe): testo o dati binari di dimensioni fino a 512 MB
- **List** (elenchi): una raccolta di stringhe nell'ordine in cui erano state aggiunte
- **Set**: una raccolta non ordinata di stringhe con capacità di intersezione, unione e differenza di altri tipi set
- **Sorted Set** (set memorizzati): set ordinati per un valore
- **Hash**: una struttura dati per lo storage di un elenco di campi e valori
- **Bitmap**: un tipo di dato che offre operazioni a livello di bit
- **HyperLogLog**: una struttura dati probabilistica per la stima di voci uniche in un dataset

Semplicità e intuitività
Redis semplifica la creazione di codice perché permette di memorizzare, visualizzare e utilizzare i dati nelle applicazioni impiegando un numero inferiore di righe. Ad esempio, se i dati di un'applicazione sono memorizzati in mappe hash e si desidera memorizzare tali dati in un datastore, è possibile utilizzare la struttura dei dati hash in Redis. Per eseguire un'attività simile su un datastore senza strutture dati hash, sarebbero necessarie molte linee di codice aggiuntive per eseguirne la conversione da un formato all'altro. Redis include strutture dati native e diverse opzioni che permettono di gestire e interagire con i dati. Per gli sviluppatori Redis sono disponibili più di cento client open source. Tra i linguaggi supportati sono presenti Java, Python, PHP, C, C++, C#, JavaScript, Node.js, Ruby, R, Go e molti altri.

Replica e persistenza
Redis impiega un'architettura primary-replica e supporta la replica asincrona, in cui i dati vengono replicati su diversi server appositi. In questo modo è possibile ottenere migliori prestazioni in lettura poiché le richieste vengono suddivise tra i diversi server ed il ripristino in caso di interruzione del server principale. Per la persistenza, Redis supporta backup point-in-time (copia su disco del dataset Redis).

Disponibilità e scalabilità elevate
Redis offre un'architettura primary-replica in un singolo nodo principale o una topologia a cluster. In questo modo è possibile creare soluzioni dotate di elevata disponibilità e prestazioni costanti e affidabili. Le dimensioni di un cluster possono essere modificate in vari modi, secondo modelli di scalabilità orizzontale e verticale. Le risorse di un cluster potranno variare di pari passo con la domanda.

Estensibilità
Redis è un progetto open source sostenuto da una community molto attiva. Non esistono vincoli di fornitore o di tecnologia; Redis è basato su standard aperti, supporta formati di dati aperti e funziona con diversi client.

2.11.4 Casi d'uso di Redis comuni

Caching
Redis è la soluzione ideale per implementare caching in memoria ad elevata disponibilità e ridurre la latenza in accesso ai dati, potenziare il throughput e alleggerire il carico da applicazioni e database relazionali o NoSQL. Il servizio è in grado di servire elementi richiesti con maggiore frequenza con tempi di risposta inferiori al millisecondo e di calibrare le risorse in base al carico senza dover investire in costosi back-end. Il caching di Redis è ideale in casi d'uso quali la memorizzazione di risultati di query di database, sessioni persistenti, pagine Web e oggetti utilizzati di frequente, quali immagini, file e metadata.

Chat, messaggistica e code
Redis supporta la messaggistica pub/sub con criteri di ricerca e diversi tipi di strutture dati, tra cui elenchi e hash. In questo modo è in grado di offrire prestazioni elevate per chat, flussi di commenti in tempo reale, feed da social network e l'intercomunicazione di server. La struttura dati List di Redis semplifica l'implementazione di code semplici da gestire. List consente atomicità nelle operazioni e funzionalità di blocco, quindi può essere utilizzata per diverse applicazioni che richiedono affidabilità nella gestione di messaggi o elenchi circolari.

Classifiche per videogiochi
Redis è un servizio molto utilizzato tra gli sviluppatori di videogiochi per la creazione di classifiche in tempo reale. È sufficiente utilizzare la struttura di dati Sorted Set di Redis, che distingue gli elementi univoci mantenendo un elenco ordinato in base ai pun-

teggi degli utenti. In questo modo, la classifica viene aggiornata simultaneamente alla variazione dei punteggi dei giocatori. Sorted Set può essere utilizzata anche per gestire dati di serie temporali utilizzando timestamp come punteggio.

Memorizzazione
Redis, in quanto datastore in memoria dotato di disponibilità e persistenza elevate, è una soluzione molto utilizzata tra gli sviluppatori a cui occorre memorizzare e gestire dati di sessione per applicazioni su Internet. Redis offre latenza nell'ordine dei millisecondi, scalabilità e resilienza, tutti elementi fondamentali per gestire dati di sessione quali profili utente, credenziali, stati di sessione e personalizzazioni specifiche per ciascun utente.

Streaming di contenuti multimediali
Redis offre un datastore in memoria rapido ideale per lo streaming in tempo reale. Il servizio può essere utilizzato per memorizzare metadati di profili utente e cronologie di visualizzazione, token e informazioni di autenticazione per milioni di utenti e file manifest con cui permettere alle reti per la distribuzione di contenuti di trasmettere video a milioni di utenti contemporaneamente, su computer e dispositivi mobili.

Dati geospaziali
Redis offre operatori e strutture dati in memoria dedicati per gestire dati geospaziali reali su vasta scala e con la massima rapidità. Grazie a comandi quali GEOADD, GEODIST, GEORADIUS e GEORADIUSBYMEMBER, memorizzazione, elaborazione e analisi di dati geospaziali in Redis è semplice e veloce. Il servizio può anche essere utilizzato per aggiungere alle applicazioni caratteristiche basate sulla posizione, quali tempi di percorrenza, distanza percorsa e punti di interesse.

Machine Learning
Le moderne applicazioni basate sui dati necessitano di apprendimento automatico per automatizzare le decisioni ed elaborare in modo rapido grandi quantità di dati di vario genere e con varie frequenze di aggiornamento. In casi d'uso quali servizi finanziari, rilevamento di attività fraudolente nel settore ludico, inoltro di offerte in tempo reale per ad tech e creazione di corrispondenze in app di condivisione o appuntamenti, la possibilità di elaborare dati in tempo reale e prendere decisioni in pochi millisecondi è un fattore critico. Redis offre un datastore in memoria dotato di eccezionale velocità per creare, addestrare e distribuire modelli di apprendimento automatico in tempi ridotti.

Analisi in tempo reale
Redis può essere utilizzato con soluzioni di streaming quali Apache Kafka e Amazon Kinesis come datastore in memoria per acquisire, elaborare e analizzare dati in tempo reale con latenza inferiore al millisecondo. Si tratta di una soluzione ideale nei casi d'uso di analisi in tempo reale, ad esempio per social media, targeting di inserzioni, personalizzazione e IoT.

2.12 Circuit breaker

Il Pattern Circuit Breaker aiuta la gestione degli errori il cui ripristino potrebbe richiedere una quantità variabile di tempo in fase di connessione a una risorsa o ad un servizio remoto. In un ambiente distribuito le chiamate alle risorse remote e ai servizi possono non andare a buon fine a causa di errori temporanei, ad esempio connessioni di rete lente, timeout oppure indisponibilità o sovraccarico delle risorse. Questi errori in genere si risolvono autonomamente dopo un breve periodo di tempo. Un'applicazione cloud affidabile deve essere preparata per gestirli tramite una strategia, ad esempio un modello di ripetizione dei tentativi. Tuttavia, possono anche verificarsi situazioni in cui gli errori sono causati da eventi imprevisti, per cui la risoluzione potrebbe richiedere molto più tempo. Questi errori possono variare, in base alla gravità, dalla perdita parziale della connettività alla totale interruzione di un servizio. In queste situazioni potrebbe essere inutile ripetere continuamente un'operazione che ha scarse possibilità di successo, è importante invece che l'applicazione accetti rapidamente l'esito negativo dell'operazione e gestisca il problema in modo appropriato. Se un servizio è molto occupato, è possibile che un errore in una parte del sistema generi errori a cascata. Un'operazione che richiama un servizio potrebbe, ad esempio, essere configurata per l'implementazione di un timeout e restituire quindi un messaggio di errore se il servizio non risponde entro l'intervallo specificato. Questa strategia potrebbe, tuttavia, causare il blocco di un elevato numero di richieste simultanee alla stessa operazione fino alla scadenza del periodo di timeout. Le richieste bloccate potrebbero tenere in sospeso risorse di sistema critiche quali la memoria, i thread, le connessioni ai database e così via. Di conseguenza, queste risorse potrebbero esaurirsi e causare errori in altre parti del sistema, che devono usare le stesse risorse. In questi casi, sarebbe preferibile che l'operazione restituisca immediatamente un esito negativo e tentasse di richiamare il servizio solo se è probabile che questo venga eseguito correttamente. L'impostazione di un timeout più breve potrebbe risolvere il problema ma non deve essere troppo breve, altrimenti avremmo un esito negativo dell'operazione anche quando la richiesta al servizio dovrebbe dare esito positivo.

Quando usare Circuit Breaker:

Per impedire che un'applicazione tenti di richiamare un servizio remoto o accedere a una risorsa condivisa se è molto probabile che questa operazione abbia esito negativo.

Quando NON usare Circuit Breaker:

- Per gestire l'accesso a risorse private locali in un'applicazione, come una struttura di dati in memoria. In questo ambiente, l'uso di un interruttore aggiungerebbe un overhead al sistema.
- Come soluzione alternativa per la gestione delle eccezioni nella logica di business delle applicazioni.

Esempio:
In un'applicazione Web diverse pagine vengono popolate con dati recuperati da un servizio esterno. Se il sistema implementa una memorizzazione nella cache minima, la maggior parte dei riscontri per queste pagine causerà un round trip al servizio. Le connessioni dall'applicazione Web al servizio potrebbero essere configurate con un periodo di timeout (in genere 60 secondi), per cui se il servizio non risponde entro questo intervallo di tempo la logica in ogni pagina Web assumerà che il servizio non è disponibile e genererà un'eccezione. Se, tuttavia, il servizio ha esito negativo e il sistema è molto occupato, gli utenti potrebbero essere costretti ad attendere fino a 60 secondi prima che si verifichi un'eccezione. Alla fine, risorse come memoria, connessioni e thread potrebbero esaurirsi e impedire ad altri utenti di connettersi al sistema, anche se non stanno accedendo a pagine che recuperano dati dal servizio. Il ridimensionamento del sistema mediante l'aggiunta di ulteriori server Web e l'implementazione del bilanciamento del carico potrebbe ritardare quando le risorse si esauriscono, ma non risolverà il problema perché le richieste utente continueranno a non rispondere e tutti i server Web alla fine potrebbero ancora esaurire le risorse. Il wrapping della logica che si connette al servizio e recupera i dati in un interruttore può essere utile per risolvere questo problema e gestire l'esito negativo del servizio in modo più efficace. Le richieste utente continueranno ad avere esito negativo, ma in tempi più rapidi, e le risorse non verranno bloccate. La classe CircuitBreaker conserva le informazioni sugli stati di un interruttore in un oggetto che implementa l'interfaccia ICircuitBreakerStateStore mostrata nel codice che segue:

```
interface ICircuitBreakerStateStore
{
  CircuitBreakerStateEnum State { get; }

  Exception LastException { get; }

  DateTime LastStateChangedDateUtc { get; }

  void Trip(Exception ex);

  void Reset();

  void HalfOpen();

  bool IsClosed { get; }
}
```

La proprietà State indica lo stato corrente dell'interruttore, che sarà Aperto, Semiaperto o Chiuso in base a quanto definito dall'enumerazione CircuitBreakerStateEnum. Il metodo IsClosed deve essere true se l'interruttore è chiuso, ma false se è aperto o semiaperto. Il metodo Trip imposta lo stato dell'interruttore su Aperto e registra l'eccezione che ha causato il cambiamento di stato, insieme alla data e all'ora in cui si è verificata l'eccezione. Le proprietà LastException e LastStateChangedDateUtc restituiscono queste informazioni. Il metodo Reset chiude l'interruttore, mentre il metodo HalfOpen lo imposta su Semiaperto. La classe InMemoryCircuitBreakerStateStore nell'esempio contiene un'implementazione dell'interfaccia ICircuitBreakerStateStore. La classe CircuitBreaker crea un'istanza di questa classe per conservare lo stato dell'interruttore. Il metodo ExecuteAction nella classe CircuitBreaker esegue il wrapping di un'operazione specificata come delegato Action. Se l'interruttore è chiuso, ExecuteAction richiama il delegato Action. Se l'operazione non riesce, un gestore di eccezioni chiama TrackEx-

ception, che imposta lo stato dell'interruttore su Aperto. Il codice di esempio seguente illustra questo flusso.

```
public class CircuitBreaker
{
  private readonly ICircuitBreakerStateStore stateStore =
    CircuitBreakerStateStoreFactory.GetCircuitBreakerStateStore();

  private readonly object halfOpenSyncObject = new object ();
  ...
  public bool IsClosed { get { return stateStore.IsClosed; } }

  public bool IsOpen { get { return !IsClosed; } }

  public void ExecuteAction(Action action)
  {
    ...
    if (IsOpen)
    {
      // The circuit breaker is Open.
      ... (see code sample below for details)
    }

    // The circuit breaker is Closed, execute the action.
    try
    {
      action();
    }
    catch (Exception ex)
    {
      // If an exception still occurs here, simply
      // retrip the breaker immediately.
      this.TrackException(ex);

      // Throw the exception so that the caller can tell
      // the type of exception that was thrown.
      throw;
    }
  }

  private void TrackException(Exception ex)
  {
    // For simplicity in this example, open the circuit breaker on
    // the first exception.
    // In reality this would be more complex. A certain type of
    // exception, such as one
    // that indicates a service is offline, might trip the circuit
    // breaker immediately.
    // Alternatively it might count exceptions locally or across
    // multiple instances and
    // use this value over time, or the exception/success ratio
    // based on the exception
    // types, to open the circuit breaker.
    this.stateStore.Trip(ex);
  }
}
```

L'esempio seguente illustra il codice (omesso nell'esempio precedente) che viene eseguito se l'interruttore non è chiuso. Verifica innanzitutto se l'interruttore è stato aperto per un intervallo di tempo più lungo di quello specificato dal campo OpenToHalfOpenWaitTime locale nella classe CircuitBreaker. In questo caso, il metodo ExecuteAction imposta l'interruttore su Semiaperto, quindi tenta di eseguire l'operazione specificata dal delegato Action. Se l'operazione ha esito positivo, l'interruttore viene reimpostato sullo stato Chiuso. Se l'operazione non riesce, viene riattivato lo stato Aperto e l'ora in cui si è verificata l'eccezione viene aggiornata, in modo che l'interruttore attenderà per un ulteriore intervallo di tempo prima di provare ad eseguire di nuovo l'operazione. Se l'interruttore è stato aperto solo per un breve periodo di tempo, inferiore al valore OpenToHalfOpenWaitTime, il metodo ExecuteAction genera semplicemente un'eccezione CircuitBreakerOpenException e restituisce l'errore che ha causato la transizione dell'interruttore allo stato Aperto. Usa inoltre un blocco per impedire che

l'interruttore tenti di eseguire chiamate simultanee all'operazione mentre è Semiaperto. Un tentativo simultaneo di chiamata all'operazione verrà gestito come se l'interruttore fosse Aperto e avrà esito negativo con un'eccezione, come descritto più avanti.

```
...
    if (IsOpen)
    {
      // The circuit breaker is Open. Check if the Open timeout has expired.
      // If it has, set the state to HalfOpen. Another approach might be to
      // check for the HalfOpen state that had be set by some other operation.
      if (stateStore.LastStateChangedDateUtc + OpenToHalfOpenWaitTime < DateTime.UtcNow)
      {
        // The Open timeout has expired. Allow one operation to execute.
        //Note that, in this example, the circuit breaker is set to HalfOpen
        //after being in the Open state for some period of time. An
        //alternative would be to set this using some other approach
        //such as a timer, test method, manually, and so on, and check
        //the state here to determine how to handle execution
        // of the action. Limit the number of threads to be executed
        //when the breaker is HalfOpen. An alternative would be to
        //use a more complex approach to determine which threads
        //or how many are allowed to execute, or to execute a simple test
        // method instead.
        bool lockTaken = false;
        try
        {
          Monitor.TryEnter(halfOpenSyncObject, ref lockTaken);
          if (lockTaken)
          {
            // Set the circuit breaker state to HalfOpen.
            stateStore.HalfOpen();

            // Attempt the operation.
            action();

            // If this action succeeds, reset the state and allow other
            //operations. In reality, instead of immediately returning
            //to the Closed state, a counter here would record the
            //number of successful operations and return the circuit
            //breaker to the Closed state only after a specified number
            //succeed.
            this.stateStore.Reset();
            return;
          }
        }
        catch (Exception ex)
        {
          // If there's still an exception, trip the breaker again immediately.
          this.stateStore.Trip(ex);

          // Throw the exception so that the caller knows which exception occurred.
          throw;
        }
        finally
        {
          if (lockTaken)
          {
            Monitor.Exit(halfOpenSyncObject);
          }
        }
      }
      // The Open timeout hasn't yet expired. Throw a CircuitBreakerOpen
      //exception to inform the caller that the call was not actually attempted,
      // and return the most recent exception received.
      throw new CircuitBreakerOpenException(stateStore.LastException);
    }
    ...
```

Per usare un oggetto CircuitBreaker per proteggere un'operazione, un'applicazione crea un'istanza della classe CircuitBreaker e richiama il metodo ExecuteAction, specificando l'operazione da eseguire come parametro. L'applicazione deve essere pronta a intercettare l'eccezione CircuitBreakerOpenException se l'operazione ha esito negativo perché l'interruttore è aperto. Il codice seguente mostra un esempio:

```
var breaker = new CircuitBreaker();

try
```

```
{
  breaker.ExecuteAction(() =>
  {
    // Operation protected by the circuit breaker.
    ...
  });
}
catch (CircuitBreakerOpenException ex)
{
  // Perform some different action when the breaker is open.
  // Last exception details are in the inner exception.
  ...
}
catch (Exception ex)
{
  ...
}
```

2.13 Web Socket

I WebSocket sono un protocollo di messaggistica che permette una comunicazione asincrona e full-duplex su connessione TCP. I WebSockets non sono connessioni HTTP anche se usano l'HTTP per avviare la connessione. Un sistema full-duplex permette la comunicazione in entrambe le direzioni in maniera contemporanea, tipico esempio di questo tipo di comunicazione è quella telefonica dove gli interlocutori possono parlare ed essere ascoltati allo stesso tempo. I WebSocket sono stati inizialmente proposti come parte della specifica HTML5 , che promette di portare la facilità di sviluppo e di efficienza della rete alle applicazioni web moderne , interattive , ma è stato successivamente spostato in un documento standard separato per mantenere la specifica focalizzata solo su WebSocket (RFC 6455 e WebSocket API JavaScript).

2.13.1 Come funziona un WebSocket ?

Ogni connessione WebSocket inizia la sua vita come una richiesta HTTP. Tale richiesta HTTP è molto simile a tutte le altre richieste salvo che nell'intestazione viene specificata un'operazione di tipo Upgrade che indica che il client vuole aggiornare la connessione ad un protocollo diverso, in questo caso a WebSocket. Ad aggiornamento avvenuto viene stabilita la connessione WebSocket tra client e server sfruttando la stessa connessione sottostante usata durante la fase iniziale della comunicazione (handshake) e la comunicazione in entrambe le direzioni può cominciare.

Piccolo esempio di handshake lato client:

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

Lato server, in risposta ad una richiesta del tipo visto sopra, abbiamo una cosa del genere:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=
```

Il server applica un'operazione nota (sia al cliente che al server) al valore della chiave Sec–WebSocket–Key presente nell'header della chiamata per generare il valore della Sec–WebSocket–Accept. Il client fa la stessa operazione sempre sul valore della stessa chiave Sec–WebSocket–Key, e, se il valore ricevuto dal server coincide con il valore calcolato dal client la connessione può considerarsi come stabilita con successo e client e server possono cominciare a comunicare. Tramite i WS è possibile scambiare sia messaggi di testo (codificati come UTF–8) sia messaggi in formato binario. Un end-pint WebSocket è rappresentato da URI nel seguente formato:

```
ws://host:port/path?query
wss://host:port/path?query
```

Lo schema ws rappresenta le comunicazioni in chiaro mentre lo schema wss rappresenta le comunicazioni criptate. La porta è un dato facoltativo, si consideri che per le comunicazioni in chiaro si usa la porta 80 mentre per le comunicazione criptate la 443, analogamente al protocollo l'http. La componente path rappresenta il path del server dove si trova l'end-point e la parte query è opzionale. I browser moderni implementano il protocollo WebSocket e forniscono una API JavaScript per la connessione agli end-point, inviare messaggi, e assegnare i metodi di callback per gli eventi websocket (quali: connessioni aperte, messaggi ricevute, connessioni chiuse, ecc.).

2.13.2 Vantaggi nell'uso delle WebSockets

II WebSocket sono più efficienti e performanti rispetto ad altre soluzioni, come il polling.

- Richiedono meno banda e riducono la latenza
- Semplificano le architetture applicative in real-time
- I WebSocket non richiedono intestazione per inviare messaggi riducendo quindi la larghezza di banda richiesta .

2.13.3 A cosa mi possono servire i WebSocket?

Alcuni dei possibili casi d'uso di WebSocket sono :

- applicazioni di chat
- giochi multiplayer
- Stock trading o applicazioni finanziarie
- Editing cooperativo di documenti
- Applicazioni di social networking

2.13.4 WebSockets in Java

La JSR 356 è la specifica di riferimento dei WebSocket per la piattaforma Java che permette di creare, configurare e distribuire endpoint Websocket in una web application, nonché implementare client remoti per poter accedere a tali endpoint da qualsiasi applicazione java. L'API Java per WebSocket si compone dei seguenti package.

- javax.websocket.server che contiene le annotazioni, le classi e le interfacce per creare e configurare gli endpoint del server.
- javax.websocket che contiene annotazioni, classi, interfacce, e le eccezioni che sono comuni a client e server endpoint.

Gli Endpoint WebSocket sono istanze della classe javax.websocket.Endpoint. L'API Java per WebSocket consente di creare gli endpoint sia in maniera programmatica sia mediante l'uso di annotation. Per creare un endpoint in maniera programmatica, si estende la classe Endpoint e si fa l'override dei vari metodi che gestiscono il ciclo di vita di un server websocket. Per creare un endpoint mediante annotazioni, basta decorare una classe Java e alcuni dei suoi metodi con le annotazioni specifiche. Dopo aver creato l'endpoint, questo viene distribuito esponendo un URI specifico che i client remoti possono utilizzare per connettersi ad esso. L'API WebSocket è una libreria puramente event driven .

2.13.5 Un esempio

Le comunicazioni via websocket sono naturalmente indipendenti dalle varie implementazioni e consente la comunicazione tra entità eterogenee implementate in un qualsiasi linguaggio di programmazione. Come esempio vogliamo mostrare un caso del genere in cui abbiamo un server endpoint implementato in java e un client endpoint implementato in javascript che permette di inviare e ricevere messaggi da una comunissima pagina html.

WebSocket Java Server Endpoint:

```
package myfirstws;

import java.io.IOException;
&amp;nbsp;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
&amp;nbsp;
/**
 * @ServerEndpoint da un nome all'end point
 * Questo pu  essere acceduto via ws://localhost:8080/myfirstws/echo
 * "localhost"    l'indirizzo dell'host dove    deployato il server ws,
 * "myfirstws"    il nome del package
 * ed "echo"    l'indirizzo specifico di questo endpoint
 */
@ServerEndpoint("/echo")
public class EchoServer {
    /**
     * @OnOpen questo metodo ci permette di intercettare la creazione
     *di una nuova sessione.
     * La classe session permette di inviare messaggi ai client connessi.
     * Nel metodo onOpen, faremo sapere all'utente che le operazioni
     *di handskake  sono state completate con successo ed    quindi
```

```
     *possibile iniziare le comunicazioni.
     */
    @OnOpen
    public void onOpen(Session session){
        System.out.println(session.getId() + " ha aperto una connessione");
        try {
            session.getBasicRemote().sendText("Connessione Stabilita!");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
 
    /**
     * Quando un client invia un messaggio al server questo metodo
     *intercetter  tale messaggio e compier  le azioni di conseguenza.
     *In questo caso l'azione    rimandare una eco del messaggi indietro.
     */
    @OnMessage
    public void onMessage(String message, Session session){
        System.out.println("Ricevuto messaggio da: " + session.getId() + ": " + message);
        try {
            session.getBasicRemote().sendText(message);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
 
    /**
     * Metodo che intercetta la chiusura di una connessione da parte di un client
     *
     * Nota: non si possono inviare messaggi al client da questo metodo
     */
    @OnClose
    public void onClose(Session session){
        System.out.println("Session " +session.getId()+" terminata");
    }
}
```

WebSocket Javascript client

```
<!DOCTYPE html>
    <html>
        <head>
            <title>Echo Web Socket</title>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width">
        </head>
        <body>
            <div>
                <input type="text" id="messageinput"/>
            </div>
            <div>
                <button type="button" onclick="openSocket();" >Open</button>
                <button type="button" onclick="send();" >Send</button>
                <button type="button" onclick="closeSocket();" >Close</button>
            </div>
            <!-- la risposta del server viene scritta qu  -->
            <div id="messages"></div>

            <!-- Script che utilizza i WebSocket -->
            <script type="text/javascript">

                var webSocket;
                var messages = document.getElementById("messages");


                function openSocket(){
                    // Assicura che sia aperta un unica connessione
                    if(webSocket !== undefined && webSocket.readyState
                    !== WebSocket.CLOSED){
                        writeResponse("Connessione WebSocket gi   stabilita");
                         return;
                    }
                    // Creiamo una nuova istanza websocket
                    webSocket = new WebSocket("ws://localhost:8080/EchoChamber/echo");

                    /**
                     * Facciamo il bind delle funzioni con gli eventi dei websocket
                     */
                    webSocket.onopen = function(event){
                        // quando viene aperta la connessione inviamo un
                        //messaggio di OKal server scrivo il messaggio nella
                        //textbox e lo invio
```

```
                    writeResponse("OK");
                    send();
                };

                webSocket.onmessage = function(event){
                    writeResponse(event.data);
                };

                webSocket.onclose = function(event){
                    writeResponse("Connection closed");
                };
            }

            /**
             * Invia il contenuto della text input al server
             */
            function send(){
                var text = document.getElementById("messageinput").value;
                webSocket.send(text);
            }

            function closeSocket(){
                webSocket.close();
            }

            function writeResponse(text){
                messages.innerHTML += "<br/>" + text;
            }

        </script>

    </body>
</html>
```

2.14 WebToken

JWT è uno standard (RFC 7519) che definisce un modo compatto e autonomo per la trasmissione sicura di informazioni come oggetto Json. Queste informazioni possono essere verificate e attendibili perché firmate digitalmente. Lo scenario più comune per l'utilizzo di JWT è quello dell'autenticazione/autorizzazione. Una volta effettuato l'accesso con le sue credenziali da parte dell'utente, ad ogni richiesta successiva si includerà nell'header il JsonWebToken, consentendogli di accedere a route, servizi e risorse consentiti con quel token. Generalmente i JsonWebToken sono composti da 3 parti separate da punti (.). Esse sono:

- Header: composto da due parti, descrive rispettivamente il tipo di token e l'algoritmo di firma utilizzato
- Payload: contiene i reclami, ovvero dichiarazioni su un'entità (in genere, l'utente) e alcuni dati aggiuntivi
- Signature: per generare questa ultima parte è necessario ottenere l'intestazione codificata, il payload codificato, un "segreto" e firmare tutto insieme utilizzando l'algoritmo specificato precedentemente nel campo intestazione

La firma viene utilizzata per verificare che il messaggio non sia stato modificato durante la trasmissione e che il mittente del JWT sia veramente chi dice di essere. E' infatti possibile risalire ai dati dell'utente analizzando il payload del JWT ricevuto. Nel processo di autenticazione quando l'utente accede correttamente utilizzando le proprie credenziali, gli verrà restituito un JWT che dovrà poi includere nell'intestazione di autorizzazione

utilizzando lo schema Bearer. Questo può essere, in alcuni casi, un meccanismo di autorizzazione stateless. Il diagramma seguente mostra come ottenere e utilizzare una JWT per accedere alle risorse ed alle API:

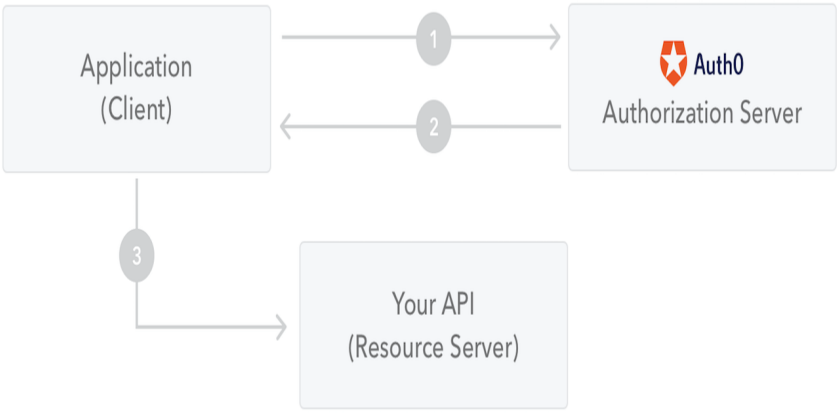

Figure 13: JWT example

2.14.1 Vantaggi

- JSON è meno dettagliato di XML e quando viene decodificato la sua dimensione è ridotta. Ottimo per essere trasmesso in ambienti come HTML e HTTP
- I parser JSON sono comuni nella maggior parte dei linguaggi di programmazione perchè si associano direttamente agli oggetti. Ciò semplifica il lavoro con JWT rispetto alle asserzioni SAML.
- JWT viene utilizzato su scala internet. Questo evidenzia la facilità di elaborazione lato client del JWT su più piattaforme, in particolare quelle mobile.

2.15 MongoDB

Le aziende e le organizzazioni sono alla ricerca di nuovi metodi per gestire il flusso di dati e sono attratte da strumenti e sistemi di gestione del database alternativi che sono diversi dai tradizionali sistemi di database relazionali. Uno di questi sono certamente i database NoSql, che vengono utilizzati per gestire al meglio i dati generati dagli utenti, i dati sulla posizione geografica, i dati generati dall'Internet of Things (IoT o Internet delle cose), i grafici sociali, dati che nel mondo reale sono aumentati esponenzialmente. Usando il database NoSQL possiamo archiviare e gestire documenti, valori-chiave, dati basati su grafici in modo facile e veloce. Possiamo facilmente evitare complesse operazioni di join SQL.

2.15.1 Cos'è MongoDB?

MongoDB è un sistema di gestione di database (DBMS) open source orientato ai documenti e che supporta varie forme di dati. È una delle numerose tecnologie di database non relazionali nate a metà degli anni 2000 sotto l'etichetta NoSQL per l'uso in applicazioni big data e altri processi di elaborazione che coinvolgono dati che non si adattano bene a un modello relazionale rigido. MongoDB è stato creato da Dwight Merriman ed Eliot Horowitz, quando avevano riscontrato problemi di sviluppo e scalabilità con i tradizionali approcci di database relazionali durante la creazione di applicazioni web presso DoubleClick, una società di pubblicità online ora di proprietà di Google. Il nome del database deriva dalla parola gigantesca per rappresentare l'idea di supportare grandi quantità di dati. Essere uno strumento NoSQL significa che non utilizza le solite righe e colonne che si associano così tanto alla gestione del database relazionale. È un'architettura che si basa su raccolte e documenti.

2.15.2 Architettura MongoDB: come sono organizzati i dati?

Per comprendere al meglio come MongoDB lavora è bene avere chiaro i concetti su cui esso si basa. Innanzitutto, con RDBMS (Relational Database Management Systems) abbiamo database, tabelle, righe e colonne. Sono i classici database relazionali. Ma nei database NoSQL, come MongoDB, i dati sono archiviati in strutture chiamate "collezioni". Dette anche raccolte, le collezioni contengono insiemi di documenti e funzionano come l'equivalente delle tabelle del database relazionale.

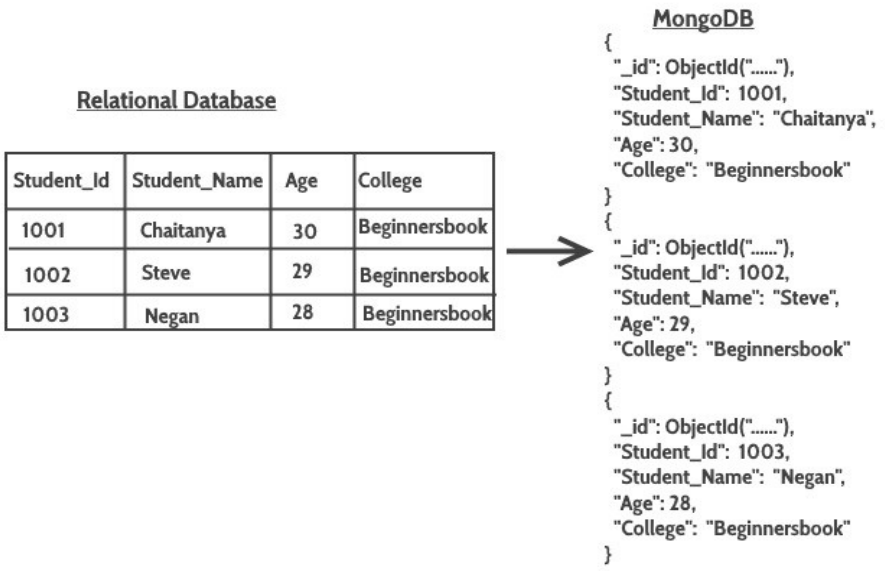

Figure 14: Esempio MongoDB

I campi nei documenti sono simili alle colonne di un database relazionale e i valori in essi contenuti possono essere una varietà di tipi di dati, inclusi altri documenti, secondo il manuale dell'utente MongoDB. Un record in MongoDB è un documento, che è una struttura di dati composta da coppie di campi e valori. I documenti, che devono anche includere una chiave primaria come identificatore univoco, sono l'unità di base dei dati in MongoDB. Sono rappresentate dalle righe nei database relazionali. La dimensione massima di un documento è di 16 MB, ma MongoDB utilizza GridFS per contenere documenti più grandi. Infine si ha il Database: in parole semplici, può essere chiamato il contenitore fisico per i dati. Ciascuno dei database ha il proprio set di file sul file system con più database esistenti su un singolo server MongoDB.

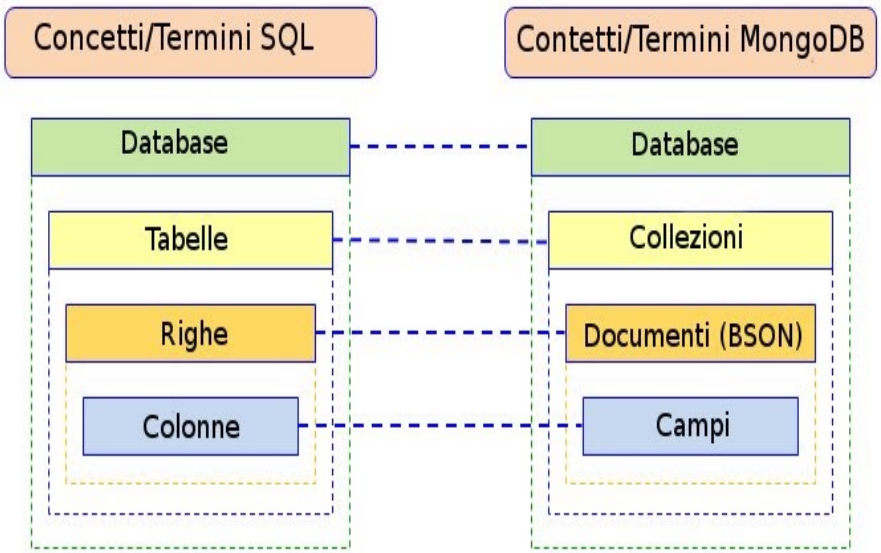

Figure 15: Esempio termini Sql e noSql

2.15.3 Scalabilità dei dati

La scalabilità è una caratteristica di un sistema che descrive la sua capacità di funzionare con un carico di lavoro maggiore. Un sistema che si adatta bene sarà in grado di mantenere e aumentare il suo livello di prestazioni in caso di maggiori esigenze operative. I database relazionali solitamente utilizzano un approccio di ridimensionamento dei dati di tipo verticale. Ciò significa che si collegano ad un singolo server e man mano che vengono gestiti più dati, viene aggiunto più spazio di archiviazione o aggiornata ad una CPU più potente con una maggiore RAM per gestire il carico di lavoro. Il server rimane uno, al limite diventa più grande. Al contrario MongoDB utilizza una forma di ridimensionamento orizzontale chiamata sharding, che permette di distribuire i dati su più macchine e facilitare operazioni ad alta produttività con grandi serie di dati, invece

di affidarsi a un singolo server ad alta velocità e alta capacità. In questo modo è possibile fornire agli amministratori di sistema la possibilità di aumentare la capacità al momento richiesto, definendo solamente quante macchine possono essere connesse correttamente.

2.15.4 Alta disponibilità dei dati

Oltre al ridimensionamento orizzontale MongoDB garantisce un'alta disponibilità dei dati. Questo avviene tramite il processo di sincronizzazione dei dati tra più server (detto replica), in modo da avere più copie dei dati su diversi server di database. Tramite la replica è possibile proteggere un database dalla perdita di un singolo server, di ripristinarlo da guasti hardware e interruzioni del servizio. Con copie aggiuntive dei dati, è possibile inoltre dedicarne uno al ripristino di emergenza, ai report o al backup. In MongoDB si ha un set di repliche formate da due o più nodi (almeno 3 nodi di solito). Uno di questi è il nodo primario e i nodi rimanenti sono secondari. Tutti i dati vengono replicati dal nodo primario a quelli secondari, per i motivi accennati poco fa. Al momento del guasto del server o della manutenzione, viene eletto un nuovo nodo primario. Il nodo guasto, si unisce nuovamente al set di repliche e funziona come nodo secondario. Viene mostrato un diagramma tipico della replica di MongoDB in cui l'applicazione client interagisce sempre con il nodo primario e il nodo primario replica quindi i dati sui nodi secondari.

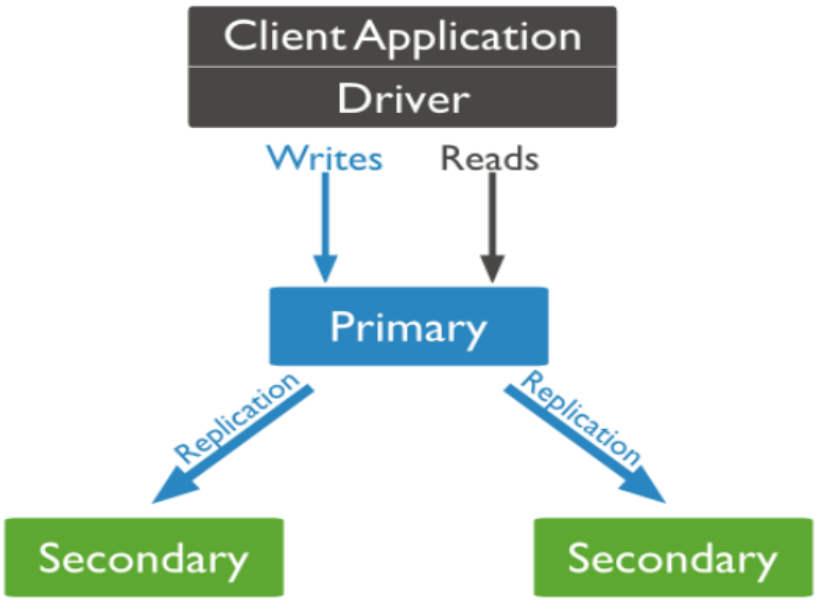

Figure 16: Esempio funzionamento replica

2.15.5 MongoDB Query Language (MQL)

In MongoDB, i documenti sono rappresentati da file BSON, un'estensione dei file JSON (JavaScript Object Notation) che aiuta a fornire tipi di dati aggiuntivi, campi ordinati e maggiore efficienza per la codifica e la decodifica in lingue diverse. JavaScript Object Notation (JSON) è uno standard leggibile da una macchina che facilita lo scambio di dati e insieme a XML è il formato principale per lo scambio di dati utilizzato sul web moderno. Si vuole ricordare anche che un file JSON può essere facilmente analizzato, con poca o nessuna trasformazione, direttamente da JavaScript e dai linguaggi di programmazione più diffusi. Quindi per MongoDB risulta semplice archiviare e rappresentare questi formati. Una volta creata una collezione e salvati all'interno dei documenti, è possibile sfruttare MongoDB per estrarre informazioni tramite un'interfaccia JavaScript interattiva, lo shell Mongo. Esso consente agli utenti di eseguire query e aggiornare i dati e condurre operazioni amministrative, come l'inserimento di codice JavaScript. Tuttavia, i comandi MongoDB possono diventare piuttosto veloci, quindi funzionalità come il completamento automatico delle query per far risparmiare tempo all'utente. La shell è un componente standard delle distribuzioni open source di MongoDB, e una volta installato il database NoSql, permette agli utenti di collegarsi dallo shell stesso alle loro istanze MongoDB in esecuzione.

2.16 MySQL

Uno dei più grandi contributi che i sistemi informatici offrono al genere umano è la memorizzazione di dati in maniera persistente. Quotidianamente, immense quantità di informazioni vengono affidate a tecnologie che ne garantiscono la conservazione duratura ed un recupero efficiente che ne permetta l'analisi. Da anni, questo ruolo viene interpretato molto bene da un prodotto software completo, efficiente ed affidabile: MySQL. Nel seguito chiariremo sin da subito che cos'è esattamente, a cosa serve e come utilizzarlo, illustrandone anche le principali caratteristiche e potenzialità.

2.16.1 Database e DBMS

I concetti centrali in tema di memorizzazione di dati sono due: database e DMBS. Il primo indica un sistema di file finalizzato a memorizzare informazioni a supporto di un qualsivoglia software. La struttura interna di un database deve rispettare una certa architettura di immagazzinamento dei dati per poterne permettere il corretto salvataggio, il rispetto dei tipi impiegati e soprattutto agevolarne il recupero, un'operazione generalmente molto onerosa. Un DBMS è un servizio software, realizzato in genere come server in esecuzione continua, che gestisce uno o più database. I programmi che dovranno interagire quindi con una base di dati non potranno farlo direttamente, ma dovranno dialogare con il DBMS. Esso sarà l'unico ad accedere fisicamente alle informazioni. Quanto detto implica che il DBMS è il componente che si occupa di tutte le politiche di accesso, gestione, sicurezza ed ottimizzazione dei database.

2.16.2 Database relazionali e RDBMS

I DBMS esistenti non sono tutti della stessa tipologia. Al giorno d'oggi, ad esempio, si parla molto di DBMS NoSQL, nati per venire incontro alle esigenze dei più recenti servizi Web. Eppure un filone molto nutrito di DBMS, cui si deve il funzionamento della maggior parte dei prodotti informatici esistenti oggi, è quello dei cosiddetti RDBMS (Relational DBMS), ispirati dalla Teoria Relazionale. Questa nacque nel 1970 ad opera del britannico Edgar f. Codd. Nonostante i 40 anni abbondanti trascorsi, i suoi principi si dimostrano tuttora attuali. Un database relazionale è costituito da tabelle, ognuna delle quali è composta da righe identificate da un codice univoco denominato chiave. Le tabelle che compongono il database non sono del tutto indipendenti tra loro ma relazionate da legami logici. La figura seguente mostra un esempio di database:

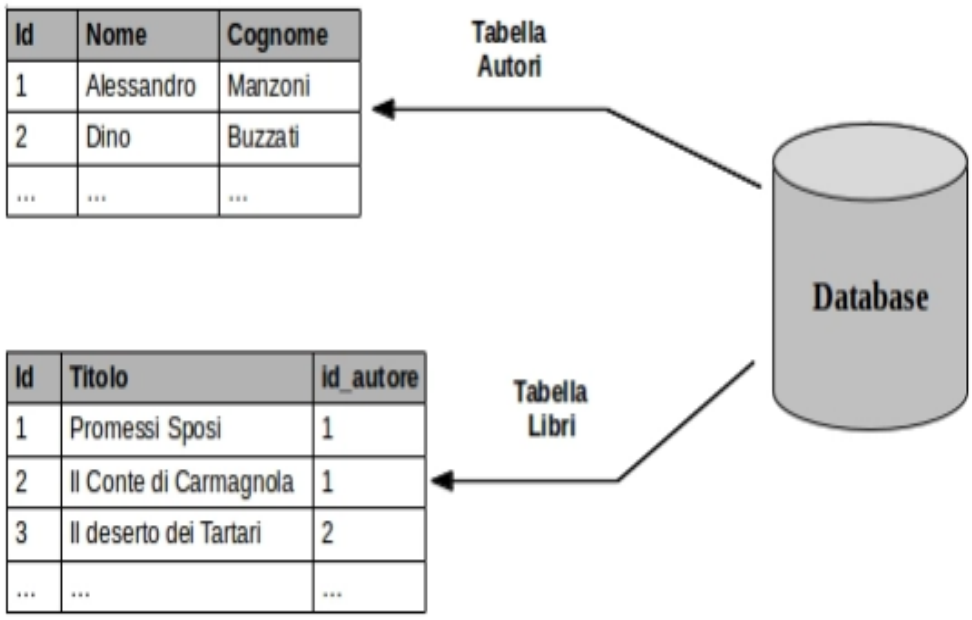

Figure 17: Esempio database relazionele

Il database ha due tabelle: Autori e Libri. Ognuna di esse ha una colonna di nome id contenente l'identificativo univoco, la chiave primaria, che permette di riconoscere una riga senza possibilità di confusione. I valori nel campo id_autore della tabella Libri permettono di associare un autore alle sue opere. Ad esempio, "I Promessi Sposi" e "Il Conte di Carmagnola" hanno nel campo id_autore il valore 1, cioè la chiave primaria che nell'altra tabella permette di riconoscere il loro autore, Alessandro Manzoni. Questo legame creato tramite chiavi prende il nome di relazione.

Figure 18: Esempio tabella relazionale

Esistono vari tipi di relazioni. Quello appena descritto è un esempio di relazione uno-a-molti, in quanto ad un autore possono corrispondere più libri. Nel nostro esempio si è assunto che un libro possa avere un solo autore, ma sfruttando relazioni di tipo differente si potranno rappresentare situazioni più vicine alla realtà. Protagonista importante è il linguaggio SQL (Structured Query Language). Si tratta di un formalismo che permette di indicare al DBMS quali operazioni svolgere sui database che gestisce. Tramite SQL si può attivare qualsiasi tipo di operazione, sia sui dati che sulla struttura interna del database, sebbene le principali (e più frequenti) operazioni ricadono in una delle seguenti quattro tipologie: inserimento, lettura, modifica e cancellazione di dati, tipicamente indicate con l'acronimo CRUD (Create-Read-Update-Delete).

2.16.3 MySQL

MySQL è un RDBMS open source e libero, e rappresenta una delle tecnologie più note e diffuse nel mondo dell'IT. MySQL nacque nel 1996 per opera dell'azienda svedese Tcx, basato su un DBMS relazionale preesistente, chiamato mSQL. Il progetto venne distribuito in modalità open source per favorirne la crescita. Dal 1996 ad oggi, MySQL si è affermato molto velocemente prestando le sue capacità a moltissimi software e siti Internet. I motivi di tale successo risiedono nella sua capacità di mantenere fede agli impegni presi sin dall'inizio:

- alta efficienza nonostante le moli di dati affidate;

- integrazione di tutte le funzionalità che offrono i migliori DBMS: indici, trigger e stored procedure ne sono un esempio, e saranno approfonditi nel corso della guida;
- altissima capacità di integrazione con i principali linguaggi di programmazione, ambienti di sviluppo e suite di programmi da ufficio.

3 Design

3.1 Structure

Come citato precedentemente, per la realizzazione di questo progetto si è optato per un'architettura a Microservizi. Dalla **Figura 19** si può osservare la struttura finale realizzata che consiste in un insieme di Microservizi che possono comunicare tra di loro, sia tramite scambio di messaggi asincrono sull'eventbus di Vert.X, sia tramite RPC con l'utilizzo di ServiceProxy (sempre dell'eventbus). I vari Microservizi non vengono invocati direttamente dai client, vi sono dei Gateway che si occupano di ridirigere le richieste ad essi se queste sono lecite e se il servizio è disponibile. L'API Gateway si occupa di ridirigere tutte le richieste API ai vari Microservizi, gli altri servono per registrare i Client tramite l'utilizzo di SockJS o del BridgeTCP sull'eventbus di Vert.X.

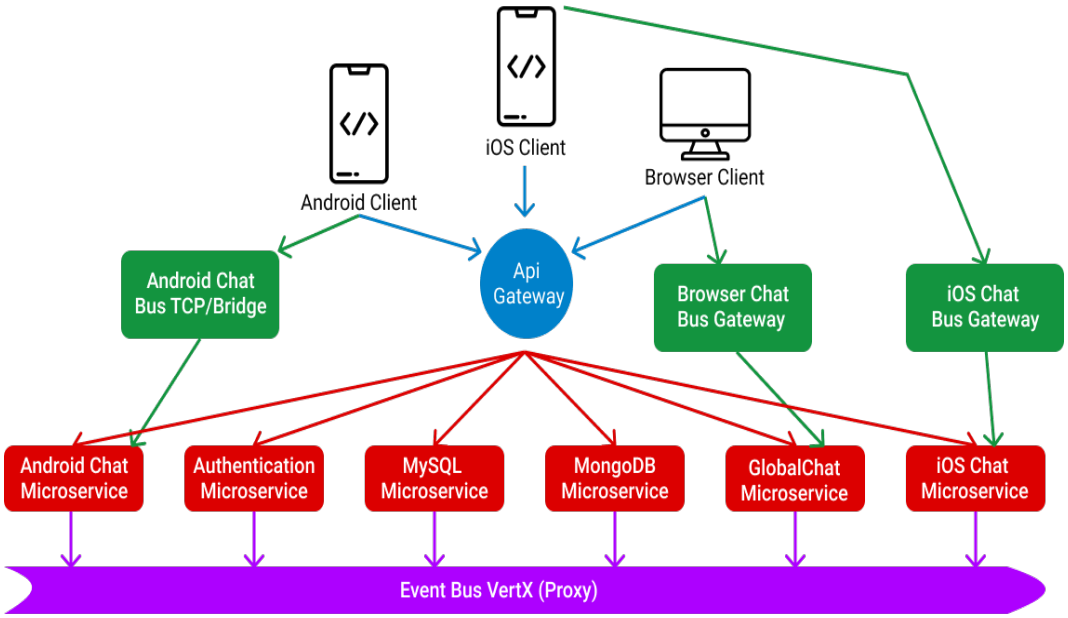

Figure 19: Architettura generale dei Microservizi

La **Figura 20** rappresenta il Diagramma delle Classi completo dei vari Microservizi. Si è scelto di racchiudere all'interno delle Classi "BaseMicroserviceVerticle" e "RestAPIVerticle" le funzionalità in comune di tutti i Microservizi che andranno ad estendere queste classi.

"BaseMicroserviceVerticle" contiene i metodi di registrazione dei vari tipi di servizi, la loro pubblicazione e la rimozione dal "ServiceDiscovery".

"RestAPIVerticle" raccoglie tutti i tipi di "ResultHandler" che possono essere utilizzati dai Microservizi, la creazione di un "HttpServer" e la gestione dei vari tipi di codice di

stato HTTP da inserire nella risposta.

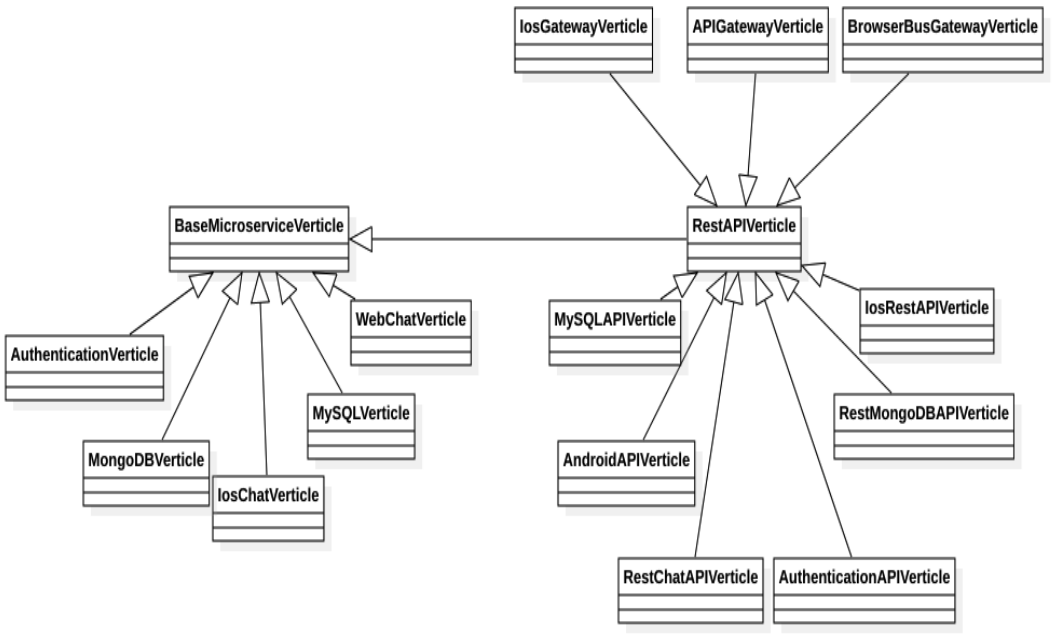

Figure 20: Diagramma delle Classi "globale"

Nella **Figura 21** vengono rappresentati i metodi implementati dalle Classi precedentemente trattate. Come citato in precedenza, l'"APIGateway" è un Microservizio che si occupa di ricevere le richieste API dai vari Client, verificare se all'interno del "ServiceDiscovery" sia presente un Microservizio con il target specificato, invocare e processare la richiesta. Se quest'ultima è presente, il Gateway inoltrerà la richiesta trasformandosi a sua volta in un Client ed effettuando una richiesta HTTP al servizio. Una volta processata la richiesta, verrà mandata la risposta al Gateway che, a sua volta, la rimanderà al Client che aveva inizialmente effettuato la richiesta.

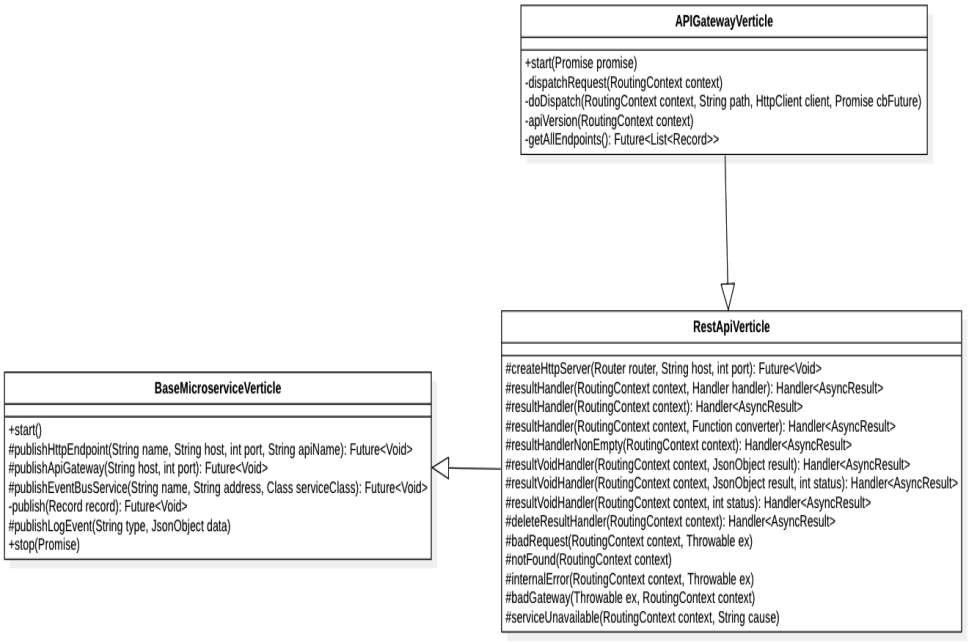

Figure 21: Diagramma delle Classi "common"

Nella **Figura 22**, viene presentato il Microservizio "android-chat-microservice" che si occupa della gestione di tutte le richieste API da parte dei Client mobile Android. Per questioni di tempo e scelte progettuali, l'unica API attualmente implementata è "sendGlobalMessage", la quale si occuperà di salvare il messaggio ricevuto dal Client in uno storage (MongoDB o MySQL) e di inoltrare il messaggio a tutti gli altri Client collegati sull'eventbus del server. Per connettere i Client Android all'eventbus di Vert.X, è stato utilizzato un TCPBridge che è stato trattato nel **Capitolo 2**. Per la trasmissione di un nuovo messaggio dal Client al Server, è sufficiente effettuare una publish sull'eventbus all'indirizzo in cui è registrato il Client.

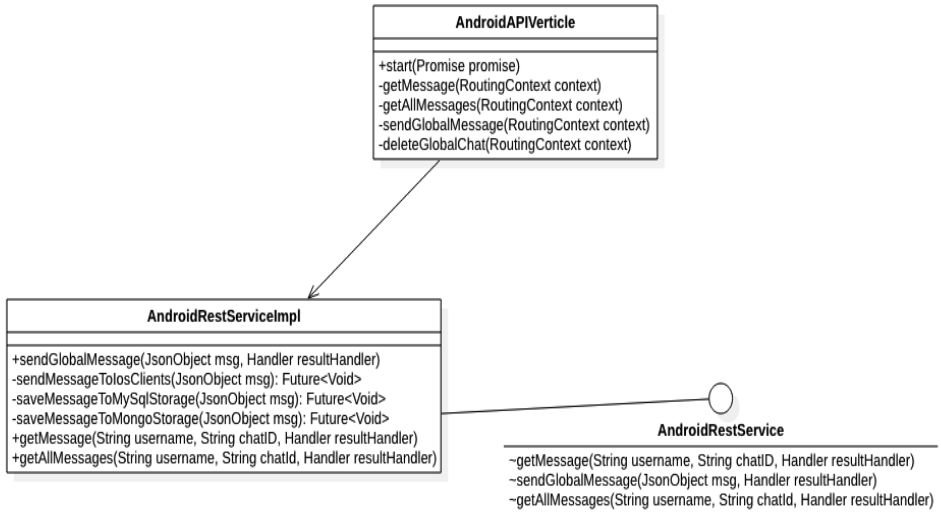

Figure 22: Diagramma delle Classi "andrord-chat-microservice"

Il microservizio "authentication-microservice" (**Figura 23**) gestisce tutte le richieste API che i client effettuano durante il processo di Autenticazione. Grazie a questo, è possibile registrare il proprio account. Successivamente, inserendo le credenziali dell'account creato, verrà effettuato il login e restituito un token che servirà per svolgere eventuali richieste successive. Questo microservizio è in grado di verificare se un dato Token sia valido e consistente. Inoltre, è possibile analizzare il token, risalendo in quel determinato momento al ruolo che ricopre l'account loggato nel client. Per poter svolgere tutte queste operazioni, sono state utilizzate alcune classi di utility come JsonWebToken e SecurityUtils.

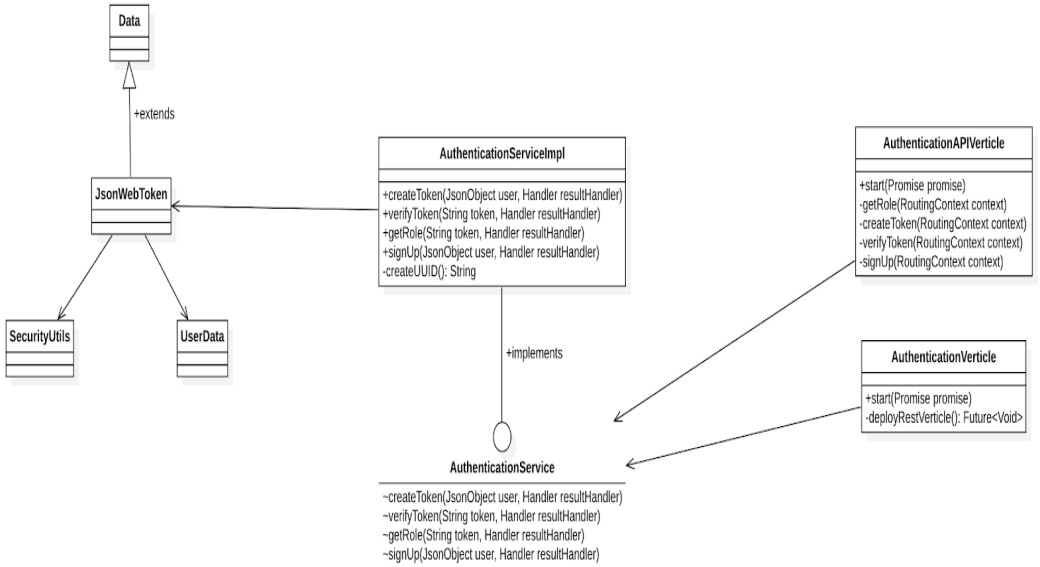

Figure 23: Diagramma delle Classi "authentication-microservice"

Il microservizio "ios-chat-microservice" (**Figura 24**) si occupa della gestione di tutte le richieste API da parte dei Client mobile iOS. Per questioni di tempo e scelte progettuali, attualmente l'unica API che è possibile invocare è "sendGlobalMessage". Questa API si occupa di salvare il messaggio ricevuto dal client in uno storage (MongoDB o MySQL) e di inoltrare il messaggio a tutti gli altri client collegati sull'eventbus del server. Per permettere un aggiornamento in tempo reale della chat, è stato adottato nel Client un approccio a WebSocket gestito dalla libreria "StarScream" lato Swift e dalla libreria "SockJ" lato Java Vert.X. Al momento della connessione, il Client registra una WebSocket presso il server sulla quale si andrà ad inoltrare un messaggio ogni qualvolta questo verrà ricevuto. Alla disconnessione del Client, questa socket verrà rimossa dalla lista di socket aperte. Tutte queste procedure vengono gestite dal servizio di "IosGatewayVerticle".

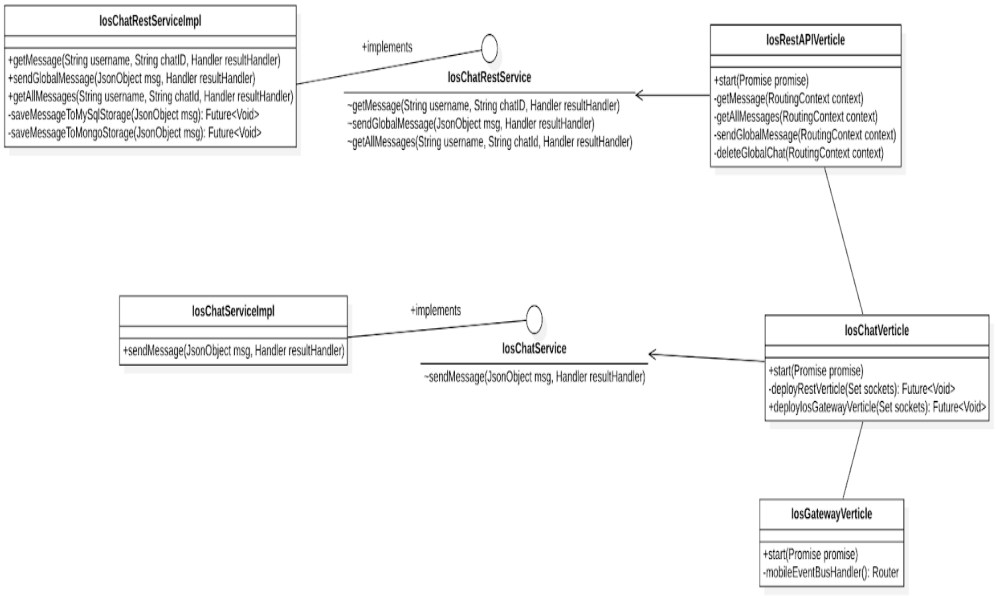

Figure 24: Diagramma delle Classi "ios-chat-microservice"

Il microservizio "web-chat-microservice" (**Figura 25**) svolge la stessa funzione dei due microservizi Client mobile trattati precedentemente, l'unica differenza sostanziale consiste nei metodi API implementati. Questi mostrano il corretto funzionamento e le possibili interazioni dei vari microservizi. L'API "sendGlobalMessage" svolge la stessa funzione dei microservizi precedenti. Le API "addOnlineUser" e "removeOnlineUser" si occupano rispettivamente di aggiungere ad un Set un utente che si è collegato al server (salvato per Username) e rimuoverlo in caso di disconnessione. Questo Set verrà poi utilizzato lato Client per mostrare a ciascun Utente quanti e quali altri utenti sono attualmente Online ed, in caso abbiamo effettuato precedentemente il login, permetterà tramite Click sull'Username la creazione di una Chat Privata. La Chat Privata viene gestita dalla chiamata API "createPrivateChat" che cambia l'indirizzo dell'evenbus sul quale i due Client andranno a comunicare in modo da non permettere la ricezione dei messaggi di altre chat. Infine, l'API "getAllMessages", ha la funzione di ritornare tutti i messaggi inviati all'interno di una specifica chat, identificata tramite un "id". Questa API può essere invocata soltanto da un utente registrato come amministratore. Per quanto riguarda l'aggiornamento dei messaggi in tempo reale, lato Client (Javascript) si è utilizzata la libreria SockJS e lato Server (Java) Vert.X. Questa implementazione permette di registrare un Client Javascript sull'eventbus di Vert.X e, di conseguenza, registrare un handler per la ricezione dei messaggi pubblicati sul bus dal server.

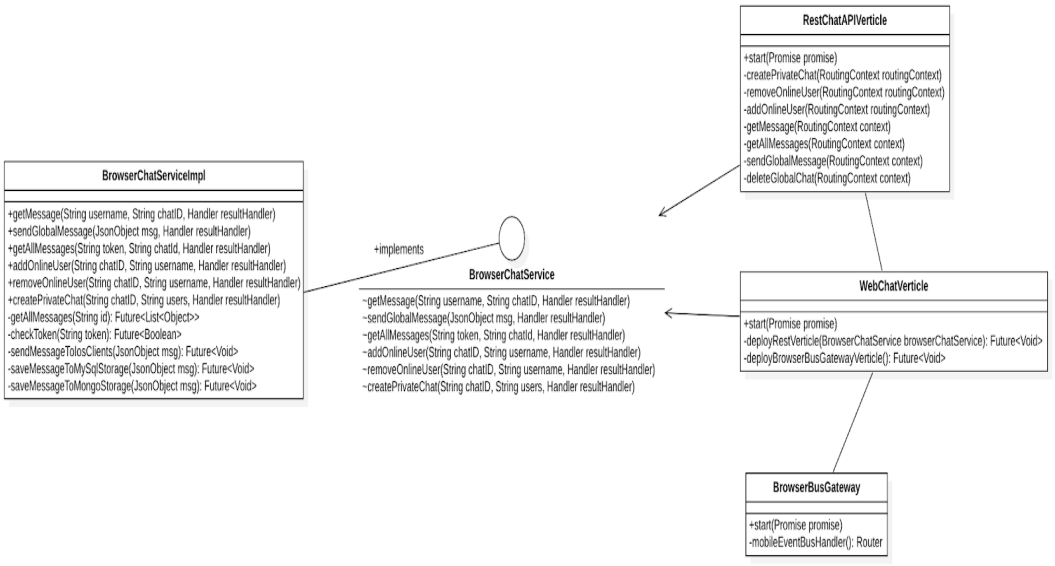

Figure 25: Diagramma delle Classi “web-chat-microservice”

Il Microservizio “mysql” (**Figura 26**), insieme a MongoDB, si occupa di tutte quelle operazioni che riguardano la persistenza dei dati. Le API messe a disposizione da MySQL vengono utilizzate dai vari Microservizi per lo storage dei dati relativi agli Utenti (Users) ed i Messaggi delle Chat (Messages). Tramite i metodi di Create, si andranno a creare la tabella per la memorizzazione degli Utenti registrati al sistema, la tabella per la memorizzazione dei messaggi inviati nella Chat Globale e le tabelle relative alle varie chat aperte (Private) che verranno create quando gli Utenti Online decideranno di inviare messaggi tra di loro privatamente. I metodi Insert, Get ed Update sono resi disponibili dal Microservizio tramite API per consentire ad altri Microservizi di interagire con esso gestendo i dati del database. I Messaggi, verranno ricevuti e/o inviati al Microservizio richiedente dopo averli gestiti e adattati tramite classi dedicate o Json Object.

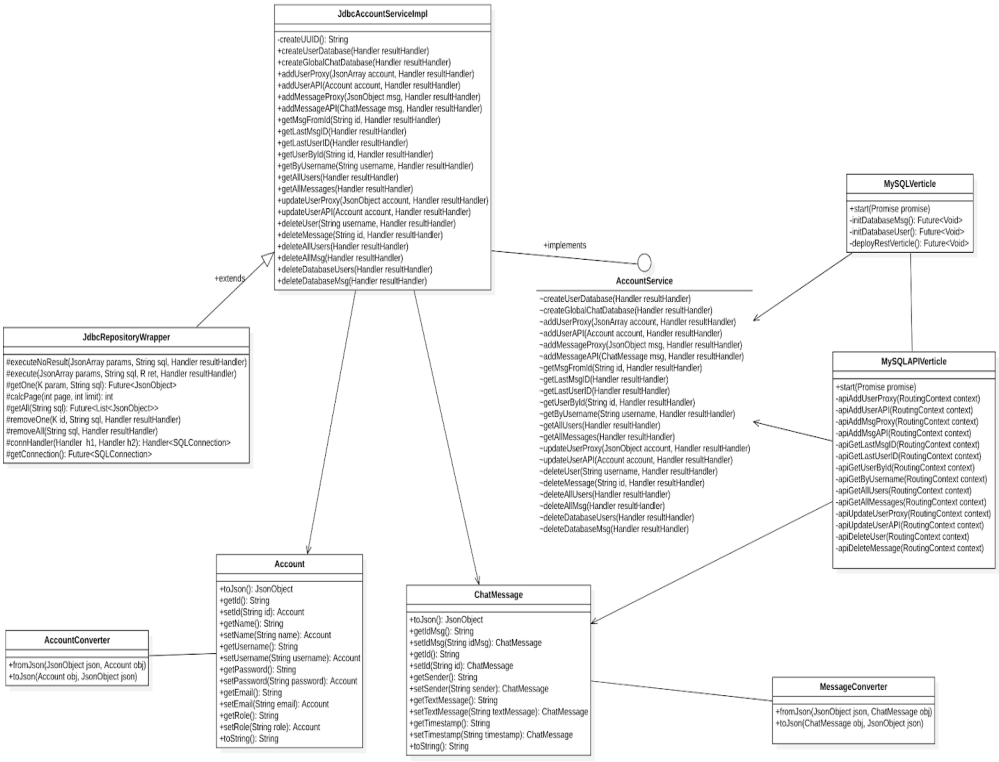

Figure 26: Diagramma delle Classi "mysql-microservice"

Il Microservizio "mongoDB" (**Figura 27**), insieme a MySQL, si occupa di tutte quelle operazioni che riguardano la persistenza dei dati. Questo Database, a differenza di MySQL, è di tipo Documentale e non Relazionale e viene utilizzato come doppione per garantire la persistenza e la disponibilità dei dati, anche in caso MySQL non sia disponibile. Le API messe a disposizione da MondoDB vengono utilizzate dai vari Microservizi per lo storage dei dati relativi agli Utenti (Users) ed i Messaggi delle Chat (Messages). Tramite i metodi di Create, si andranno a creare la collection per la memorizzazione degli Utenti registrati al sistema, la collection per la memorizzazione dei messaggi inviati nella Chat Globale e le collections relative alle varie chat aperte (Private) che verranno create quando gli Utenti Online decideranno di inviare messaggi tra di loro privatamente. I metodi Insert, Get e Update sono resi disponibili dal Microservizio tramite API per consentire ad altri microservizi di interagire con esso gestendo i dati del database. I messaggi, verranno ricevuti o inviati al microservizio richiedente dopo averli gestiti e adattati tramite classi dedicate o Json Object.

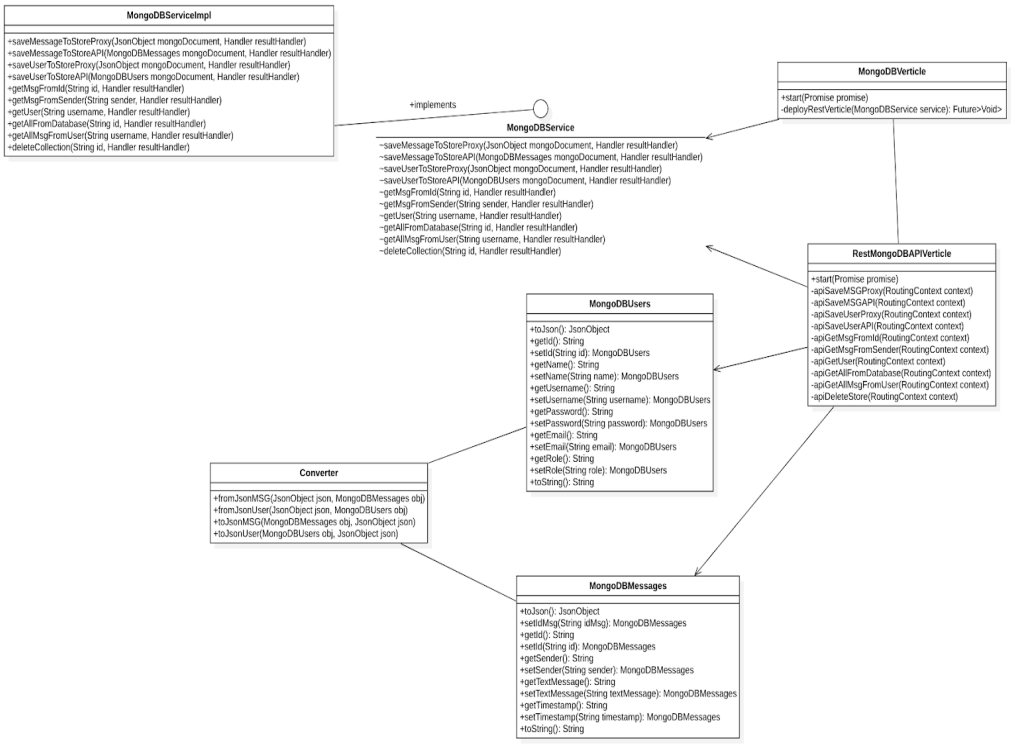

Figure 27: Diagramma delle Classi "mongodb-microservice"

3.1.1 Wrong Design

Per questo progetto, il gruppo aveva organizzato una struttura del software Monolitica e basata su Gradle. Successivamente, ci si è accorti di dover rifare totalmente da zero il Design e l'Architettura del Software dato che la struttura monolitica non era sufficientemente scalabile e difficilmente modificabile/aggiornabile. Oltretutto, ad ogni modifica, bisognava ricompilare completamente il monolite e questa operazione risultava troppo onerosa in termini di tempo per essere compilato ed eseguito. Di conseguenza, si è ovviato al problema ricreando e basando il progetto su un'architettura a Microservizi. Inoltre, si è optato per una migrazione da Gradle a Maven, per lo studio di una nuova tecnologia di gestione delle dipendenze.

3.2 Behaviour

Come introdotto in precedenza, tutti i Microservizio hanno un proprio flusso di esecuzione basato sull'architettura ad EventLoop e possono interagire tra di loro tramite scambio di messaggi o tramite chiamate RPC (Service Proxy) sull'eventbus. Il Diagramma delle Attività in **Figura 28** rappresenta il comportamento generico di un Microservizio. Quando viene effettuata la chiamata del metodo "start()" viene effettuato il

deploy della propria verticle, viene registrato il servizio presso la ServiceDiscovery ed, in base al tipo di servizio, viene eseguito il deploy di una RestAPIVerticle che si occuperà della gestione di tutte le chiamate API al Microservizio. Dopo aver terminato tutta la parte di inizializzazione e deployment, il Microservizio si metterà in attesa di possibili eventi provenienti dall'eventbus o dalle richieste API che verranno processate in maniera asincrona.

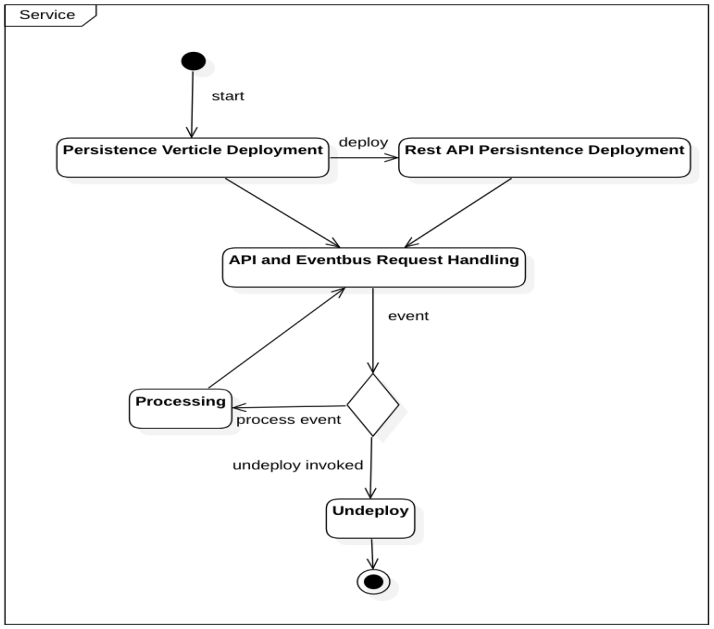

Figure 28: Diagramma delle Attività "Service"

Il diagramma di attività, in **Figura 29**, rappresenta in modo astratto le possibili interazioni e comportamenti dei microservizi a fronte dei possibili eventi che possono essere scatenati all'interno del sistema.

Authentication Microservice
Alla ricezione di un evento che richiedere di verificare se il Token inviato come messaggio sia valido o quale sia il tipo di ruolo dell'utente proprietario del token, questo Microservizio esegue una computazione asincrona interna ed, al termine dell'operazione, ritorna il risultato tramite un Handler<AsyncResult<T>> al chiamante.

Nel caso in cui arrivi un evento di tipo "registra un nuovo utente" oppure "esegui il login di un utente", il Microservizio dovrà interagire con uno dei Microservizi di storage. Nel primo caso, si dovrà creare l'hash della password ed inviare tutti i dati dell'utente al Microservizio di storage che andrà a salvare il risultato e lo ritornerà al servizio chiamante. Nel secondo caso, invece, il Microservizio di autenticazione andrà ad interrogare il Microservizio di storage per verificare che l'utente che sta richiedendo l'accesso sia presente nel database e che le due password coincidano. Se questa operazione va a buon

fine, il microservizio andrà a creare un token e lo invierà come risposta all'utente.

Chat Microservice
Il seguente microservizio ha un comportamento molto simile a quello di "andoid-chat-microservice" ed "ios-chat-microservice", dunque, onde evitare delle ripetizioni, verrà trattato soltanto una volta in questo paragrafo. La funzione condivisa dai tre microservizi è quella della gestione dei messaggi che arrivano dai client. All'arrivo di un nuovo messaggio, tramite una richiesta HTTP REST al server, vengono scatenate diverse azioni (Elencate di seguito in ordine):

1. Viene pubblicato il messaggio dell'eventbus di vert.X sull'indirizzo della chat globale target del messaggio ricevuto dal server in modo da trasmetterlo a tutti i client connessi.

2. Viene inviato il messaggio tramite WebSocket a tutti i client iOS connessi al server.

3. Viene invocato il metodo "saveMessage()" dei servizi di persistenza (MongoDB e mySQL).

4. Se tutte queste operazioni andranno a buon fine, verrà ritornato un messaggio di successo, altrimenti, in caso di fallimento di uno dei passaggi, il fallimento.

Altre funzionalità che, come detto in precedenza, non sono ancora implementate nei microservizi iOS e Android, prevedono la possibilità di aggiungere un utente online o rimuoverlo da una collection interna al microservizio. Un'altra operazione eseguibile è la creazione di una chat privata. Essa prevede la notifica all'utente target di modifica dell'indirizzo del bus su cui si andranno a ricevere i messaggi. Questa operazione non richiede interazione con altri Microservizi. Infine, la richiesta di uno o di tutti i messaggi di una relativa chat da parte di un amministratore:

1. Richiesta di autenticazione dell'utente, presso il Microservizio di Autenticazione, che verificherà se il token dell'utente sia valido e se il suo gruppo risulta essere "admin".

2. Se tale verifica andrà a buon fine, si farà richiesta al Microservizio di storage del messaggio o dei messaggi richiesti.

3. Se queste operazioni andranno a buon fine, si ritornerà una risposta con all'interno ciò che l'amministratore ha richiesto.

Storage Microservice
Questo microservizio si occuperà della gestione di tutti i dati che verranno generati durante l'esecuzione. Oltre a mettere a disposizione delle API, come descritto in precedenza, per la creazione di nuovi utenti, il login, il salvataggio e la lettura dei nuovi messaggi inviati e le varie operazioni che l'utente con il ruolo di Admin vorrà effettuare sui dati salvati, questo Microservizio si occuperà di processare tutte le richieste (INSERT,

UPDATE, GET, DELETE) che arriveranno da altri Microservizi e che richiederanno operazioni all'interno dei database Mongo DB o MySQL.

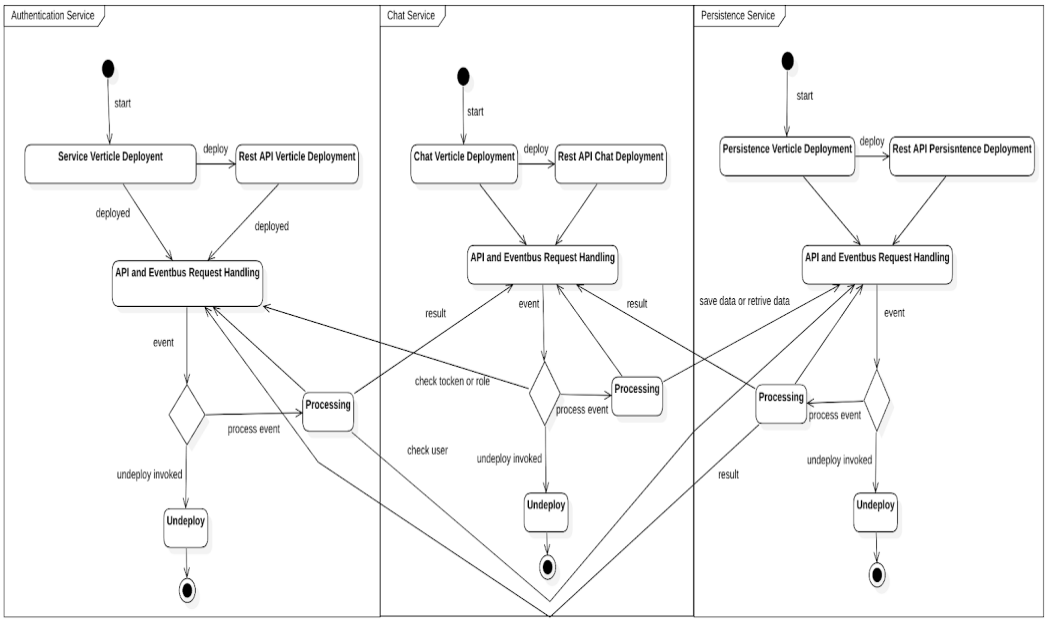

Figure 29: Diagramma delle Attività globale

3.3 Interaction

Le interazioni dell'applicazione si basano sul Client User che dovrà interagire con tutto il sistema realizzato. Come prima cosa, si andrà ad effettuare il Login o la Registrazione al servizio. Una volta interrogato l'Authentication Service per la validazione dei dati inseriti e l'assegnazione dei Token, si passeranno i dati al Chat Service che si occuperà di effettuare tutte le chiamate del caso al Persistence Service per la memorizzazione/gestione/convalida effettiva dei dati inseriti dall'utente. Successivamente il software ritornerà un feedback all'utente in attesa che visualizzerà la Chat Globale o un messaggio di errore.

Una volta visualizzata la User Interface della Chat, l'applicazione stabilirà una connessione all'eventbus del server tramite l'utilizzo di Websocket o di un bridgeTCP (nel caso del Client Android) per poter ricevere i messaggi in tempo reale. Questa operazione verrà effettuata sia per gli utenti che accederanno come ospiti che quelli che effettuano il login.

Dalla schermata principale, se l'utente deciderà di inviare un messaggio, la richiesta andrà direttamente all'APIGateway che la inoltrerà al Chat Service. Quest'ultimo si occuperà della gestione di invio del messaggio a tutti i Client connessi (Web, Android, iOS) e dell'inoltro del messaggio ad un servizio di persistenza che si occuperà del suo salvataggio.

Quando un Utente con il ruolo di "Admin" vorrà usufruire delle sue funzionalità aggiuntive di statistica o semplice visualizzazione dei dati del sistema, la richiesta verrà effettuata all'API Gateway e poi al Chat Service. Questa richiesta verrà convalidata dall'Authentication Service e successivamente eseguita interrogando il Persistence Service per ottenere i dati richiesti. Infine l'Utente vedrà visualizzato a schermo, se avente i permessi necessari, il risultato della sua richiesta effettuata.

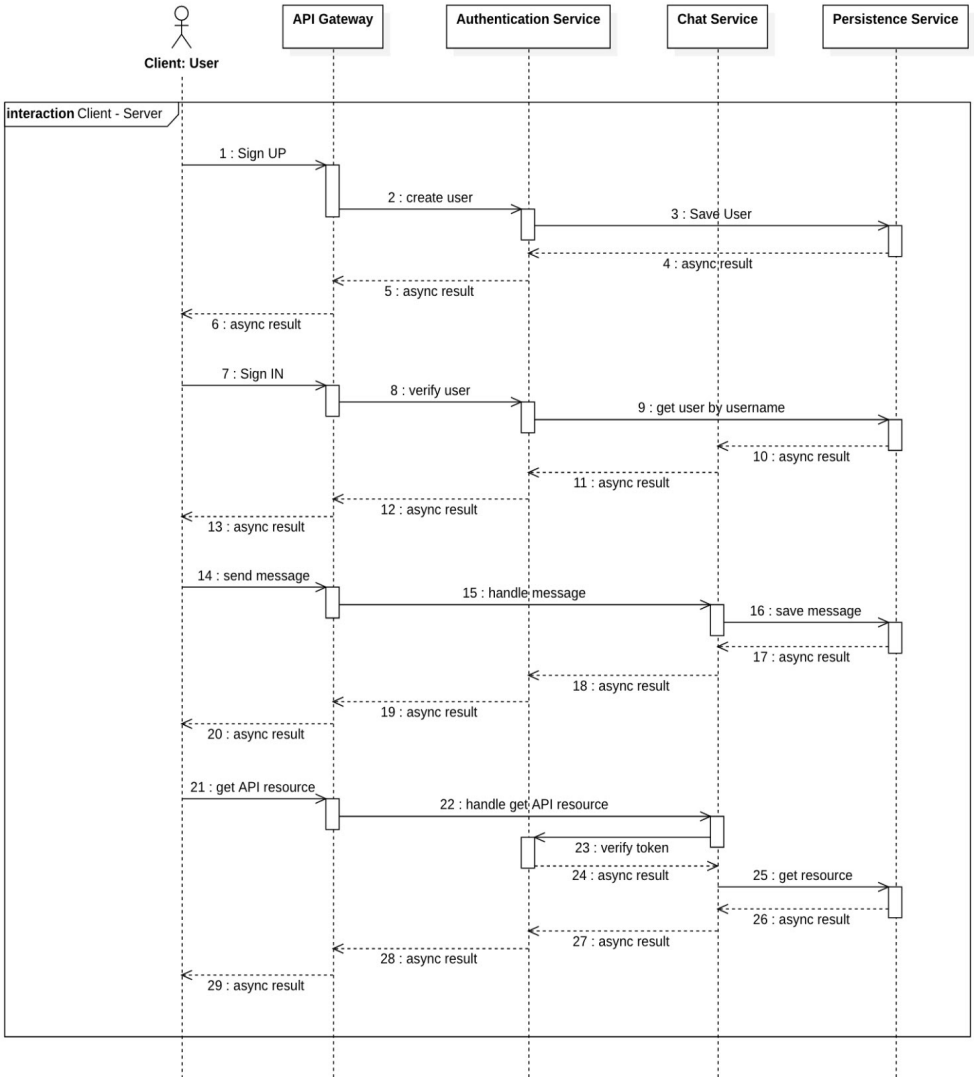

Figure 30: Diagramma di Sequenza

4 Implementation Details

Il gruppo ha deciso di strutturare il progetto con un approccio incrementale, suddiviso in macro argomenti e lavorando in gruppo a tutti i vari step della realizzazione del Software:

- Si è partiti con un'attenta analisi dettagliata del problema e si sono individuate le tecnologie più efficienti per la realizzazione del progetto, così da avere cognizione sui vari passi da svolgere durante l'esecuzione.
- Si è proseguiti definendo i dettagli implementativi per una corretta stesura del codice analizzando ciascun componente e concordando le classi e le interfacce da implementare.
- Si è inoltre deciso di individuare i test da implementare successivamente, a progetto concluso, per la verifica effettiva dei requisiti di efficienza richiesti.
- Dopo aver equamente distribuito tutto il carico di lavoro tra i vari membri del gruppo ed aver terminato la creazione del Software, si è concluso testando tutte le varie parti ed eseguendo molteplici prove da diversi client (Mobile e PC). Sono stati effettuati anche diversi stress test per analizzare le prestazioni del sistema e del protocollo utilizzato quando le richieste al server diventavano onerose.

4.1 APIGateway

Il Gateway API è l'unico punto di ingresso del client front-end. Ogni richiesta proveniente dal Client, arriverà prima al Gateway che ricoprirà anche la funzione di semplice proxy inverso ed è responsabile degli errori di bilanciamento del carico. Grazie al rilevamento automatico dei servizi, non dobbiamo preoccuparci delle modifiche alla posizione di questi. Inoltre, se il protocollo del Microservizio interno non è compatibile con il Web, si possono adattare i protocolli nel Gateway. Esistono anche alcuni svantaggi, poiché il Gateway ha obblighi importanti, deve essere disponibile e far fronte anche a richieste elevate. Si deve, perciò, prestare attenzione alle sue prestazioni in quanto potrebbe diventare il collo di bottiglia dell'applicazione. In questo modello di Microservizio, il Gateway è un singolo componente api-gateway ed ha solo una verticle (APIGatewayVerticle). Questa semplice implementazione del Gateway utilizza un modello HTTP-HTTP. In cui il Gateway riceve richieste dall'esterno e le invia agli endpoint corrispondenti.

4.2 Persistenza

Data la possibile ed ingente richiesta di memorizzazione di messaggi ed utenti, per la persistenza dei dati si è optato per la realizzazione di due diversi Database (MongoDB e MySQL). Il Software si occupa di memorizzare in entrambi gli storage tutte le informazioni relative al servizio in modo da garantire la disponibilità dei dati anche in caso della indisponibilità di uno dei due database.

4.3 Design Pattern Utilizzati

Il gruppo ha optato per l'utilizzo dei pattern **SOA**,**REST**, **APIGateway**, **Circuit-Breaker**.
SOA è un pattern che descrive architetture, implementazioni comuni e le loro aree di applicazione per aiutare nella pianificazione, implementazione, funzionamento, gestione e manutenzione continue di sistemi complessi.
REST è un modello architettonico per rendere il web più efficace.

4.4 Documentazione del Software e leggibilità del Codice

Nella realizzazione del Software si è cercato di mantenere il più possibile il codice autoesplicativo, tramite l'utilizzo di metodi ed attributi aventi nomi significativi e facilmente comprensibili. Si sono seguiti i principali pattern di programmazione cercando inoltre di modularizzare adeguatamente le entità. I passaggi più delicati sono stati trattati all'interno del codice sorgente stesso, tramite un sistema di commenti il più pulito ed efficace possibile.

5 Self-assessment/Validation

Nella parte di Testing, il gruppo ha cercato il più possibile di testare e creare degli appositi test da poter eseguire sul software. Ovviamente, vista la possibilità dell'elevato numero di entità Client in gioco, non è stato possibile testare in modo adeguato il sistema (Es. più di mille Client attivi che scrivono messaggi contemporaneamente o più di mille ruoli diversi che in contemporanea eseguono azioni di ricerca/aggiornamento dei dati mentre altri Client utilizzano la Chat). Per le procedure di testing effettuate in fase di implementazione, sono stati utilizzati un paio di Client connessi alla rete per testare l'invio molteplice di messaggi ed azioni all'interno della Chat. Inoltre si è optato per creare dei Test appositi per la parte dei singoli Microservizi per testare se il codice dal gruppo creato fosse funzionale o meno. Qui di seguito un paio di esempi di test:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running sd.microservices.test.CircuitBreakerTest
May 24, 2020 5:07:44 PM sd.microservices.test.CircuitBreakerTest
INFO: Testing circuit breaker proper working..
May 24, 2020 5:07:44 PM sd.microservices.test.CircuitBreakerTest
INFO: Sending a correct request to a working microservice..
May 24, 2020 5:07:44 PM sd.microservices.test.CircuitBreakerTest
INFO: EXPECTED: 200     RESULT: 200
May 24, 2020 5:07:44 PM sd.microservices.test.CircuitBreakerTest
INFO: Triggering circuit closure..
May 24, 2020 5:07:45 PM sd.microservices.test.CircuitBreakerTest
INFO: Sending a request while circuit is closed..
May 24, 2020 5:07:45 PM sd.microservices.test.CircuitBreakerTest
INFO: Waiting for circuit to reopen..
May 24, 2020 5:07:45 PM sd.microservices.test.CircuitBreakerTest
INFO: EXPECTED: 502     RESULT: 502
May 24, 2020 5:07:55 PM sd.microservices.test.CircuitBreakerTest
INFO: sending a request after the circuit reopened..
May 24, 2020 5:07:55 PM sd.microservices.test.CircuitBreakerTest
INFO: EXPECTED: 200     RESULT: OK
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 12.308 sec
Running sd.microservices.test.GatewayTest
May 24, 2020 5:07:56 PM sd.microservices.test.GatewayTest
INFO: Testing Gateway redirection given a wrong path..
May 24, 2020 5:07:56 PM sd.microservices.test.GatewayTest
INFO: EXPECTED: 404     RESULT: 404
May 24, 2020 5:07:56 PM sd.microservices.test.GatewayTest
INFO: Testing Gateway redirection given a correct path..
May 24, 2020 5:07:56 PM sd.microservices.test.GatewayTest
INFO: EXPECTED: 200     RESULT: 200
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.198 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

Figure 31: Risultati Test su API Gateway

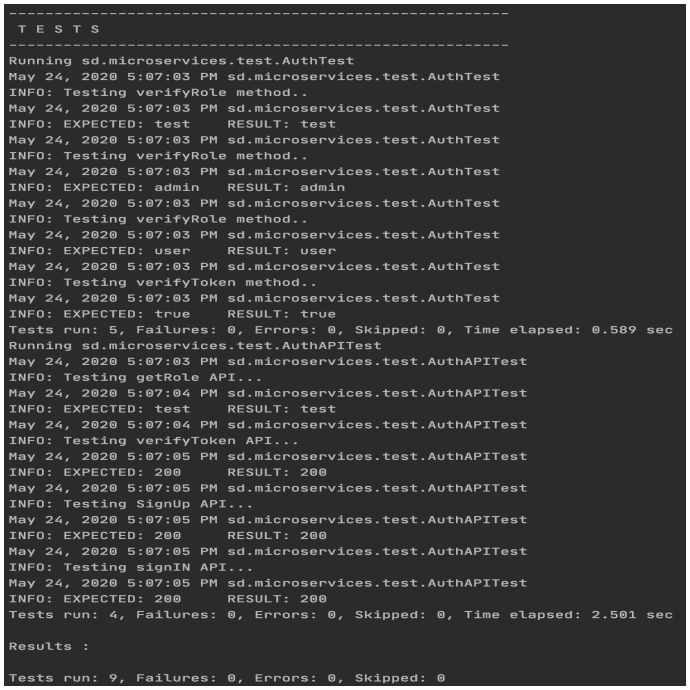

Figure 32: Risultati Test su Microservizio Authentication

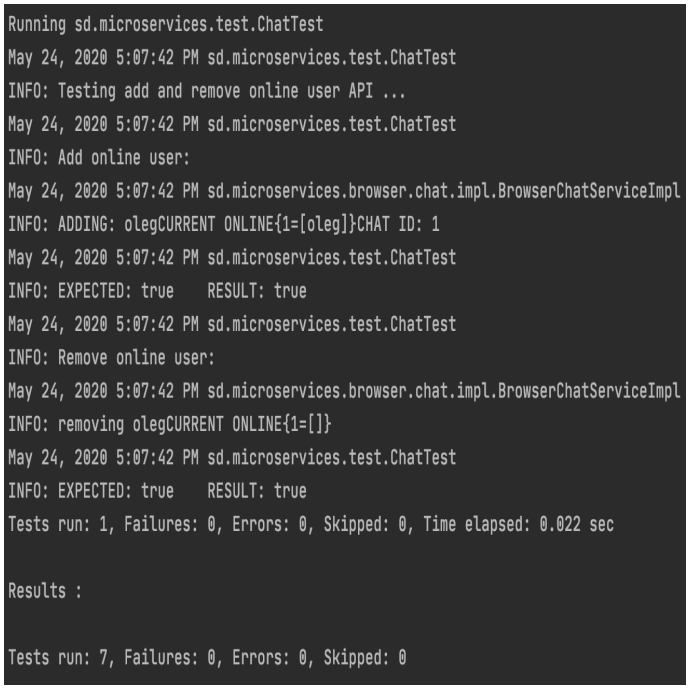

Figure 33: Risultati Test su Microservizio Web Chat Parte 1

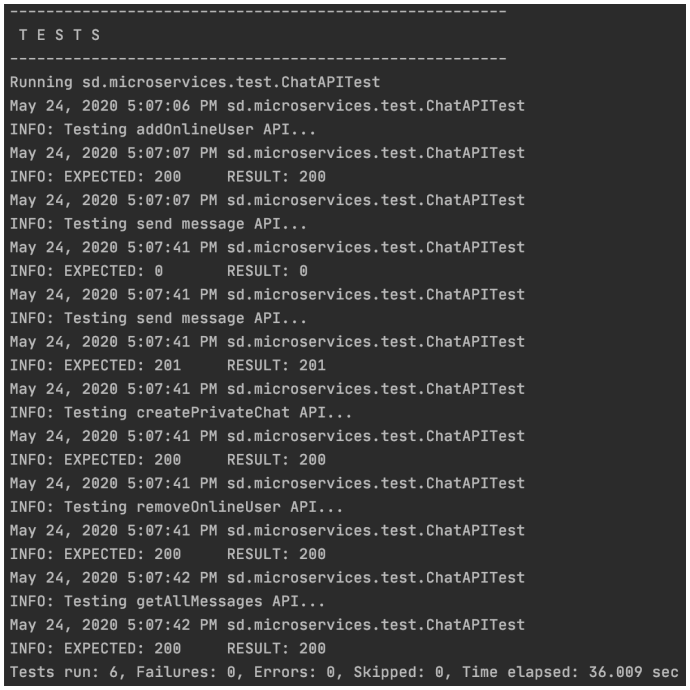

Figure 34: Risultati Test su Microservizio Web Chat Parte 2

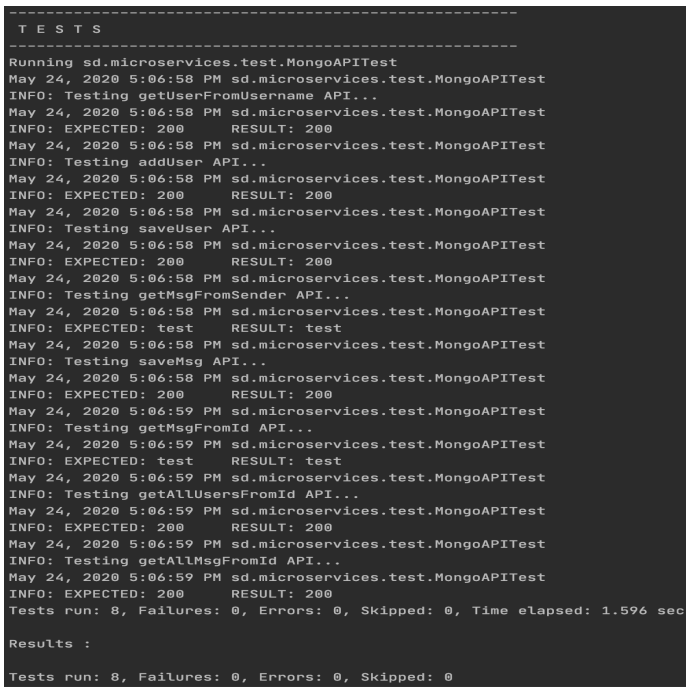

Figure 35: Risultati Test su Microservizio MongoDB

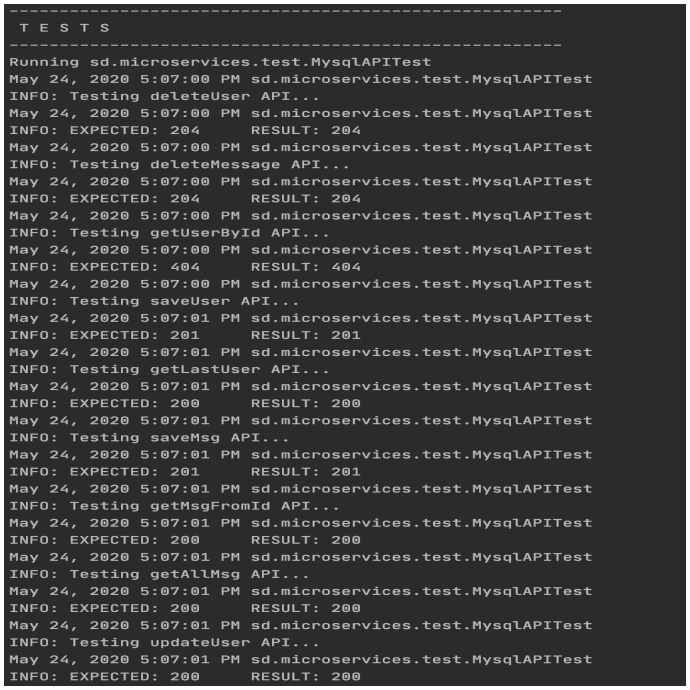

Figure 36: Risultati Test su Microservizio MySQL Parte 1

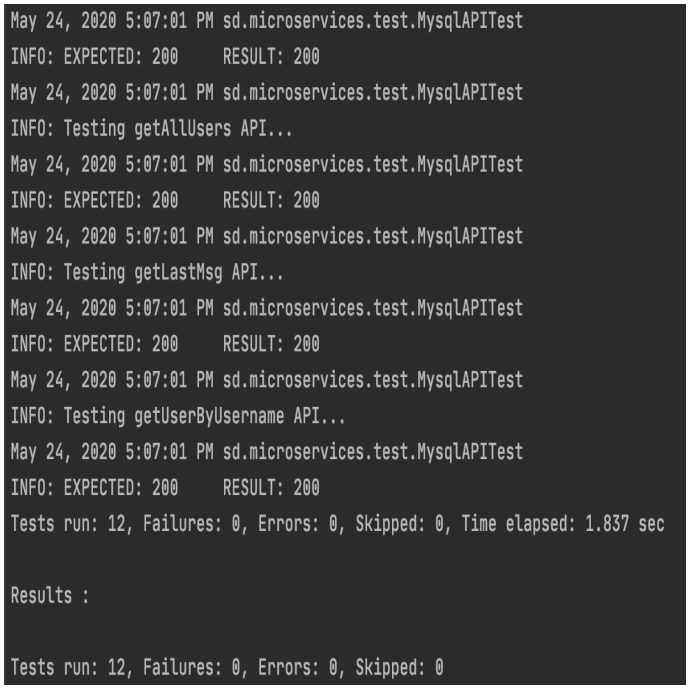

Figure 37: Risultati Test su Microservizio MySQL Parte 2

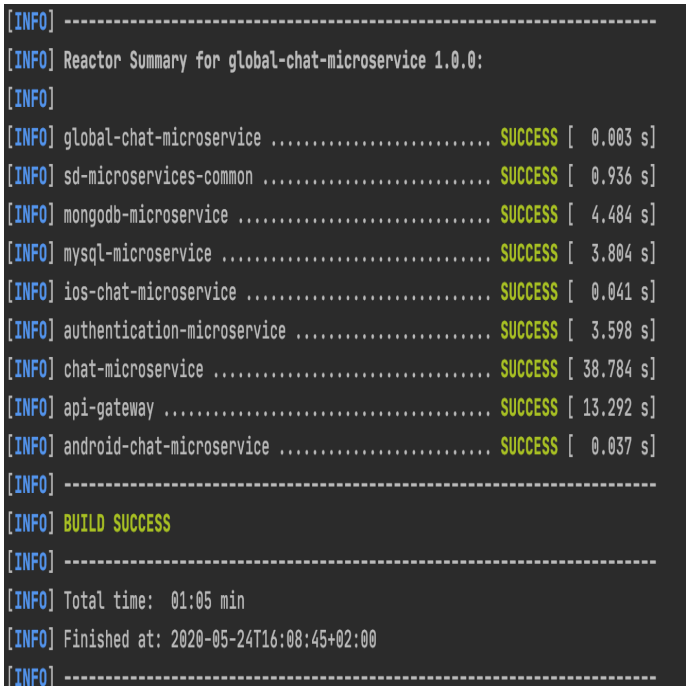

Figure 38: Risultato Finale di Testing

6 Deployment Instructions

Se si vuole installare ed eseguire il software, sarà necessario installare sul proprio PC Maven e l'applicazione Docker

Maven necessità di una JDK di 1.7+ ed uno spazio su disco di almeno 500 MB che dipenderà dalle applicazioni che si andranno ad eseguire. Per installare Maven su Windows, basterà andare al seguente Link: `http://maven.apache.org` e cliccare su "Download" nella sezione "Get Maven". Il link reindirizzerà ad una serie di pacchetti ".zip". Una volta scelto e scaricato il giusto file compatibile con il proprio sistema operativo, dovrete estrarlo ed impostare la variabile d'ambiente "M2_HOME". Andate in "System Properties" →"Advanced"→ "Environment Variables"→ "New" scrivete il nome della variabile ed il percorso in cui avete salvato Maven sul vostro pc. Successivamente rinominate la variabile d'ambiente "PATH" ed inserite lì il percorso del file "bin" di Maven. Infine verificate l'effettiva installazione di Maven aprendo una Shell e digitando "mvn –version".

Docker necessiterà di un Sistema Operativo Windows 10 o MAC, 4 GB di RAM e la virtualizzazione attiva nel vostro BIOS. Per scaricarlo basterà andare sul sito `https://hub.docker.com/editions/community/docker-ce-desktop-windows/` scaricare il file ".exe" ed eseguirlo sul proprio PC.

Dopo aver installato questi programmi, si potrà eseguire il software aprendo un terminale, posizionandosi nella directory del progetto "global-chat-microservice" e digitare i seguenti comandi:

- mvn clean install -Dmaven.test.skip=true
- cd docker
- ./build.sh
- docker-compose up

Per eseguire i test aprite poi un altro terminale nella directory "global-chat-microservice" e lanciate il seguente comando:

- mvn test

A questo punto, basterà aprire il browser e collegarsi al seguente indirizzo: `http://localhost:8080` oppure eseguite uno dei due Client Mobile realizzati (Android o iOS).

7 Usage Examples

Il sistema può essere utilizzato eseguendo sia ogni componente all'interno della stessa macchina (nei casi di testing) e sia eseguendo il Client Web ed i Client Mobile sulla stessa macchina (tramite IDE) o su dispositivi fisici (come nel mondo reale). In questo caso sarà necessario indicare ai client l'indirizzo IP e la porta utilizzati dal server per la comunicazione.
Per lanciare ed eseguire il back-end, si guardi il **Punto 6**.

8 Conclusions

Il progetto finale realizzato rispecchia accuratamente gli obiettivi ed i requisiti imposti dal gruppo ad inizio lavoro:

- Si è realizzata un'architettura efficiente e ben strutturata prendendo come scopo primario quello di strutturare al meglio la parte del Server per la gestione delle varie operazioni da eseguire.
- I servizi realizzati riescono a coordinarsi ed eseguire la corretta gestione dei messaggi, anche sotto sforzo come nei test eseguiti.
- I test implementati ed eseguiti costituiscono una garanzia della correttezza dell'elaborato realizzato. Si è notato che, grazie a questi, i moduli che compongono il sistema sono stati creati tutti correttamente ed il Software reagisce abbastanza bene anche sotto stress.

8.1 Future Works

Purtroppo, date le tempistiche di realizzazione del Software, il gruppo si è accorto di non esser riuscito a realizzare in tempo alcune delle funzionalità che aveva ideato ad inizio progetto.
Le future implementazioni e funzioni che potrebbero essere aggiunte sono:

- Implementazione della Chat Privata lato Client Mobile ed aggiunta della Lista degli Utenti Online.
- Implementazione di un ulteriore Client Java basato su GUI Swing.
- Implementazione di ulteriori funzionalità relative ai Ruoli (Moderatore, Editor, etc..).
- Implementazione e modifica server per richiesta di Chat Private con un ristre gruppo di utenti.
- Realizzazione di un piccolo Bot nella Chat Globale per la gestione dei messaggi e relative operazioni su di essi (es. Possibilità di minigiochi con il Bot).
- Miglioramento della sicurezza della Chat relativa ai vari accessi degli Utenti.

8.2 What did we learned

La realizzazione di questo Software è stato un importante banco di prova per il Team di sviluppo. In primo luogo ci ha permesso di capire ed analizzare a fondo come viene realizzato un Software distribuito all'interno della rete ed i vari collegamenti via Socket con i Client di diversi dispositivi con differenti tecnologie.
In secondo luogo il gruppo ha appreso nuove tecniche di programmazione con differenti

Framework e Servizi (Vert.X, Docker, WebSocket) che prima del corso non erano stati affrontati così nel dettaglio e sottovalutati. Per la prima volta ci si è trovati di fronte ad un progetto potenzialmente molto ampio ed oneroso, quindi si sono riscontrate molteplici difficoltà iniziali dovute alla mole di lavoro da eseguire ed alla organizzazione dei vari Microservizi.
Infine ci si è dovuti confrontare con la parte di Testing del sistema. I problemi iniziali di questa parte sono stati quelli di capire come riuscire a testare agevolmente ed in maniera approfondita un progetto di questo tipo, distribuito e potenzialmente sovraccarico di utenti che tentano di inviare messaggi nella chat o richiedono operazioni quali la registrazione/accesso ed operazioni relative ai vari ruoli che essi possono assumere. Di conseguenza abbiamo tentato di testare il più possibile la correttezza del codice e qualche esempio di operazioni onerose da parte del server quali il collegamento multiplo di molteplici utenti o l'invio multiplo di messaggi.
Possiamo concludere che la realizzazione di questo progetto ci è servita per apprendere e confrontarsi al meglio con questo tipo di tecnologia che, a nostro parere, potrebbe tornare utile in futuro all'interno del nostro settore.

9 References

1. https://vertx.io/docs/
2. https://docs.docker.com/
3. https://docs.mongodb.com/
4. https://dev.mysql.com/doc/
5. https://maven.apache.org/guides/index.html
6. http://www.sczyh30.com/vertx-blueprint-microservice/
7. https://github.com/sockjs
8. https://github.com/daltoniam/Starscream
9. https://www.html.it/pag/63253/docker-e-i-container-2/
10. https://www.ilsoftware.it/articoli.asp?tag=Docker-cos-e-e-come-funziona-la-containe
    18547
11. https://riptutorial.com/it/docker
12. https://www.ionos.it/digitalguide/server/configurazione/tutorial-di-docker/
13. https://docs.microsoft.com/en-us/dotnet/core/docker/build-container?
    tabs=windows
14. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-
    multi-container-applications-docker-compose
15. https://vertx.io/docs/vertx-sockjs-service-proxy/java/
16. https://vertx.io/docs/vertx-tcp-eventbus-bridge/java/
17. https://en.wikipedia.org/wiki/Proxy_server
18. https://vertx.io/docs/vertx-service-discovery/java/
19. https://www.sintraconsulting.it/api-gateway-definizione-caratteristiche-esempio/
20. https://docs.microsoft.com/bs-cyrl-ba/azure/architecture/patterns/circuit-breaker
21. https://docs.docker.com/docker-for-windows/install/
22. https://it.siteground.com/tutorial/php-mysql/mysql/
23. https://www.mongodb.com/it/what-is-mongodb
24. https://www.redhat.com/it/topics/microservices/what-are-microservices

25. https://www.net-informatica.it/blog/server-proxy/

26. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

27. https://hazelcast.com/

28. https://codingjam.it/maven-sveliamo-un-pom-di-segreti/

29. https://en.wikipedia.org/wiki/Hazelcast

30. https://aws.amazon.com/it/elasticache/what-is-redis/

31. https://aws.amazon.com/it/redis/

32. https://www.javaboss.it/primi-passi-con-vert-x-parte-1/