Relazione Assignment 3

Gruppo:

Oleg Konchenkov – E-mail: oleg.konchenkov@studio.unibo.it

Elia Pasqualini – E-mail: elia.pasqualini@studio.unibo.it

Giovanni Poggi – E-mail: giovanni.poggi2@studio.unibo.it

Analisi del problema:

Il gruppo ha realizzato due diversi progetti assegnati dal docente. Il primo aveva l'obiettivo della realizzazione di un programma ad attori ispirati alla rivisitazione del gioco Mastermind. Il programma consiste in N attori (N fornito esternamente dalla GUI) che svolgono il ruolo di Player del gioco ed un Attore incaricato di svolgere il ruolo dell'Arbitro.

Ogni player ad inizio gioco sceglie a caso un numero M di cifre (M fornito esternamente dalla GUI) che rappresenterà il proprio codice segreto che gli altri Player dovranno scoprire. L'obiettivo di ogni Player sarà quello di scoprire il numero segreto altrui e vincere il gioco indovinando tutti i codici degli altri. La dinamica del gioco procederà per turnazione, decisa e coordinata dall'Arbitro.

Ad ogni turno, i Player dovranno eseguire un tentativo (Guess) per indovinare il numero di un altro Player. Il criterio con cui il Player ad ogni turno deciderà l'Attore a cui fare il Guess non sono specificati. Il player che riceverà il Guess invia una risposta codificata in 2 numeri (Numero delle cifre indovinare al posto giusto e numero di cifre indovinate ma non al posto giusto). Tale risposta NON verrà inviata solo all'attore che ha fatto il Guess ma a tutti i Player. Ad ogni turno, i Player eseguiranno il proprio tentativo secondo un ordine ben preciso determinato dall'Arbitro casualmente (l'ordine cambierà ad ogni turno).

Ogni Player ha un tempo limitato per eseguire il proprio tentativo (Guess Timeout), misurato dall'Arbitro, scaduto il quale il Player salterà il proprio turno ed il suo tentativo verrà scartato. Quando un Player, dopo il suo tentativo, riterrà di aver scoperto tutti i numeri degli altri Attori, potrà inviare all'arbitro l'elenco dei Codici Segreti degli altri Player. L'Arbitro verificherà la risposta chiedendo a tutti i Player in gioco, in caso affermativo la partita terminerà eleggendo il vincitore. In caso contrario, il Player verrà eliminato dal gioco. Il programma dovrà essere dotato

di una GUI con cui si potranno scegliere i parametri citati ad inizio paragrafo, visualizzare le informazioni sullo stato del gioco, far iniziare o terminare una partita.
Il gruppo ha inoltre implementato il punto Facoltativo richiesto dal docente, ovvero l'inserimento di un Player umano che possa partecipare al gioco tramite GUI. E' possibile considerare una variazione del gioco la limitazione dell'invio della risposta ad un Guess al solo Player che ha fatto il Guess. Questo primo esercizio imponeva come vincoli l'utilizzo di un approccio ad Attori ed una programmazione NON distribuita.
Il secondo programma richiesto forniva, come base, un sorgente di un programma che gestiva il gioco di un Puzzle semplificato per bambini, pensato per un solo giocatore. Scelta un'immagine di dimensioni W x H, il puzzle viene rappresentato da una matrice di N x M tessere di dimensioni W / M e H / N. L'immagine ad inizio gioco verrà visualizzata con le tessere disposte casualmente.
Il giocatore, tramite un Client Web, potrà scambiare la posizione di due tessere selezionandole in sequenza. Lo scopo del gruppo era la realizzazione di una versione aperta collaborativa, quindi Multi-Player, che consenta ad un insieme dinamico ed eterogeneo di utenti in rete di partecipare alla risoluzione del Puzzle.
Ogni Utente che parteciperà alla sessione di gioco dovrà visualizzare sul Browser la posizione degli altri Utenti (Puntatore e Scelta dei Pezzi) e l'evolversi dello stato del Puzzle considerando tutte le azioni degli Utenti in tempo reale.
Quando il Puzzle sarà risolto, la sessione finirà e tutti i partecipanti dovranno veder apparire a schermo un messaggio corrispondente ("Risolto!").
Il sistema distribuito dovrà essere progettato a servizi, in modo che risulti come Entry Point un Puzzle Service con una Web API funzionale alla partecipazione dinamica degli Utenti nel Gioco. Gli Utenti in linea di principio potrebbero utilizzare Client eterogenei. Inoltre non sono stati forniti dal docente vincoli sulle specifiche tecnologie e modelli di programmazione da utilizzare.

Soluzione proposta:

Il gruppo è partito dall'implementazione del primo Esercizio utilizzando una programmazione ad **Attori** basata su Akka. **Akka** è un toolkit Open Source, disponibile sotto licenza Apache 2 Licence che consente di costruire applicazioni concorrenti e distribuite sulla JVM. Le applicazioni costruite con la piattaforma Akka si propongono di essere scalabili, elastiche e

responsive, in accordo con il Reactive Manifesto. Le applicazioni che seguono il modello Reactive mirano ad essere il più tolleranti ai fallimenti possibile ed in particolare a gestirli con eleganza nel momento in cui si verificano. Il modello ad attori è un framework utile per ragionare sulla concorrenza come base teorica per l'implementazione di sistemi concorrenti. Lo scopo del modello è risolvere i problemi di concorrenza legati alla memoria condivisa, eliminandola completamente. Si basa su un'entità, chiamata Attore, che comunica attraverso l'invio di messaggi. Quando un attore riceve un messaggio può inviare messaggi ad attori di cui conosce il nome, creare nuovi attori o decidere il comportamento da utilizzare per processare il prossimo messaggio ricevuto. Nel modello Akka creare attori e inviare messaggi è indipendente dalla locazione fisica dell'attore (Location Transparency). Questo rende il modello adatto a implementare sistemi distribuiti senza avere conoscenza delle relazioni all'interno della rete di computer.

Nell'**Implementazione** realizzata dal gruppo, oltre alla creazione degli attori suggeriti come Player e Arbitro, si è optato per un ulteriore attore, denominato TimeOutActor, che si occupa della gestione del TimeOut.
L'attore TimeOut, possiede una Receiver che, in base al messaggio che gli arriverà, si occuperà della gestione di tre differenti richieste:

1. Se riceverà il messaggio di "StartTimer" di occuperò di far partire il Timer, inviando eventualmente il Timeout.
2. Se riceverà il messaggio di "ResetTimer" resetterà il Timer.
3. Se riceverà, invece, un messaggio di "StopTimer", gestirà l'interruzione dell'esecuzione del Timer.

L'attore Arbitro presenta il metodo CreateReceive, che si occuperà di tutta la gestione del Gioco. L'Arbitro è il primo attore creato, non appena iniziata la partita, e si occuperà della creazione di N Attori in base alla scelta dell'utente sulla GUI (effettuando anche un controllo sull'eventuale Player Umano). Altre funzioni dell'Arbitro:

1. Si occuperà di terminare la partita in corso (Tramite StopMsg) stoppando tutti gli attori in gioco.
2. Gestirà l'eventuale tentativo del Player Umano, tramite HumanPlayerAttempt.
3. Farà partire la partita non appena tutti i Player (Umano e non) avranno generato i loro codici.
4. Gestirà i tentativi degli attori NON Umani.

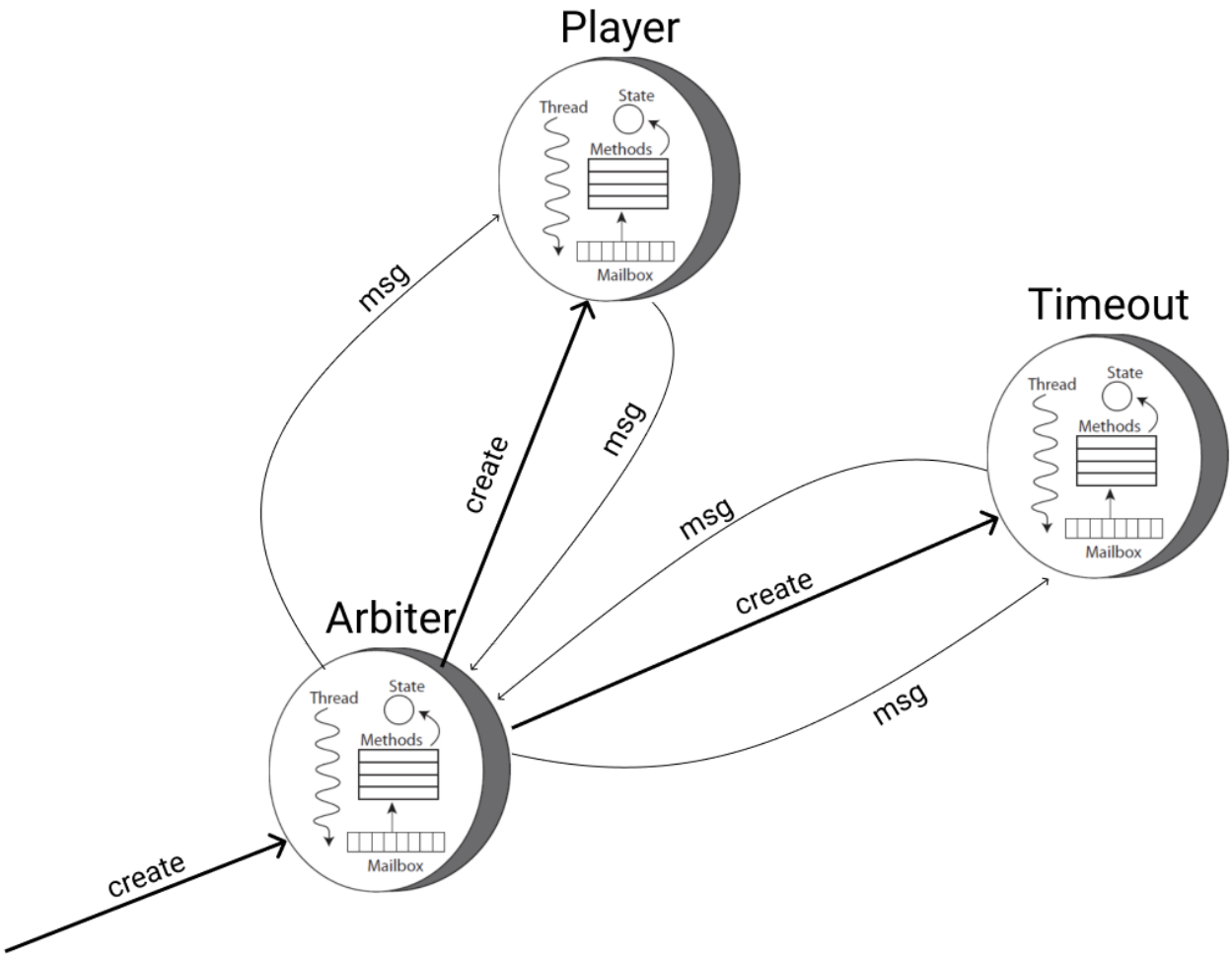
5. Si occuperà della gestione totale del Timeout tramite messaggio “Timeout”.
6. Gestirà l’eventuale vincita da parte di un Attore (Umano o non) facendo terminare il gioco e eleggendo il vincitore.
7. Infine si occuperà della creazione casuale dei Turni cambiandoli in maniera randomica ad ogni Turno.

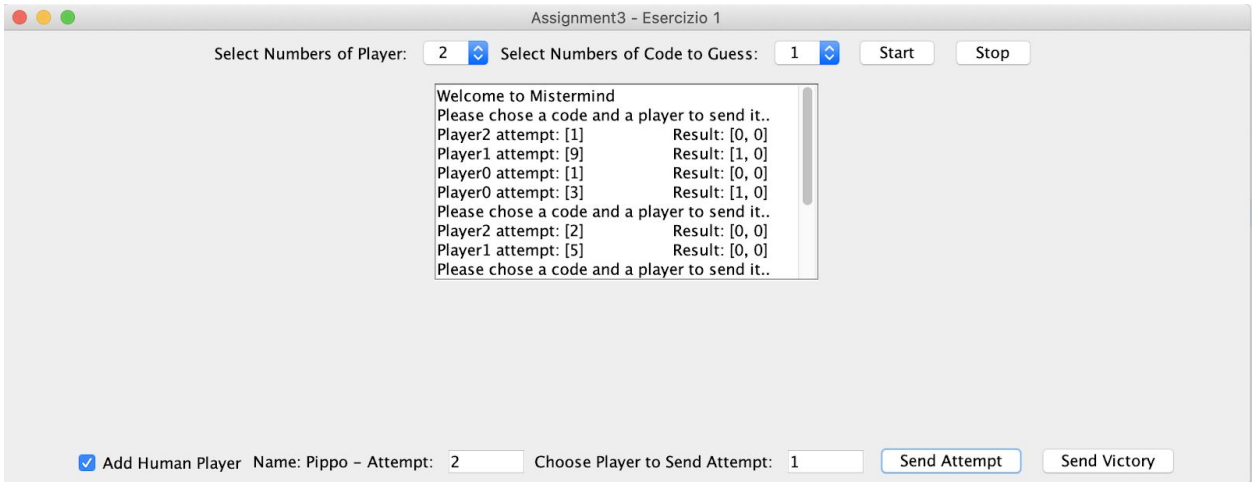
Nell’implementazione dell’attore Player, invece, si è optato per una Receive che esegue le seguenti operazioni:

1. Creazione del suo codice di gioco in caso sia un Player non Umano.
2. Creazione del codice di gioco nel caso sia Umano, indicato tramite GUI.
3. Si occuperà dell’invio del tentativo di indovinare il codice segreto di altri Player.
4. Gestirà la ricezione da parte di un altro Player di un tentativo di indovinare il proprio codice segreto, tramite SendAttemptToPlayer.
5. Si occuperà di tentare di indovinare il codice segreto degli altri Player, tramite TryAttempt.
6. Gestirà la terminazione del Turno ed, in caso, inviando il tentativo di dichiarazione della vittoria all’Arbitro, tramite SendResponseToPlayer.
7. Si occuperà di stoppare la sua esecuzione in caso il gioco sia terminato.
8. Se riceverà un messaggio di Timeout, terminerà il proprio turno non inviando il suo tentativo per indovinare il codice segreto.
9. In caso nessun messaggio corrisponda a quelli gestiti, verrà visualizzato un messaggio di errore.

Il Player Umano svolgerà le stesse funziona sopra descritte tramite una GUI ad esso dedicata, tramite la classe ViewFrame.

Di seguito viene mostrata una foto che riassume le iterazioni fra gli attori e uno screen della GUI.


Player
Thread
State
Methods
Mailbox
msg
create
msg
Timeout
Thread
State
Methods
Mailbox
msg
create
Arbiter
Thread
State
Methods
Mailbox
msg
create


Assignment3 - Esercizio 1
Select Numbers of Player: 2
Select Numbers of Code to Guess: 1
Start
Stop
Welcome to Mistermind
Please chose a code and a player to send it..
Player2 attempt: [1] Result: [0, 0]
Player1 attempt: [9] Result: [1, 0]
Player0 attempt: [1] Result: [0, 0]
Player0 attempt: [3] Result: [1, 0]
Please chose a code and a player to send it..
Player2 attempt: [2] Result: [0, 0]
Player1 attempt: [5] Result: [0, 0]
Please chose a code and a player to send it..
Add Human Player
Name: Pippo – Attempt: 2
Choose Player to Send Attempt: 1
Send Attempt
Send Victory

Successivamente, il gruppo si è concentrato sul secondo Esercizio, in ambiente distribuito. Per garantire la realizzazione di un programma distribuito nella rete, si è optato per le seguenti tecnologie: **Docker, Maven, VertX, Service Discovery, RestAPI, API Gateway, Service Proxy, SockJS, Redis, Circuit Breaker, WebSocket, WebToken, MongoDb e MySQL.**

Microservizi:

Innanzitutto, abbiamo realizzato il progetto basandoci sui **Microservizi**. I microservizi sono un approccio per sviluppare e organizzare l'architettura dei software secondo cui quest'ultimi sono composti di servizi indipendenti di piccole dimensioni che comunicano tra loro tramite API ben definite. Le architetture dei microservizi permettono di scalare e sviluppare le applicazioni in modo più rapido e semplice, permettendo di promuovere l'innovazione e accelerare il time-to-market di nuove funzionalità.

Docker:

Per la realizzazione di questi Microservizi, si è optato per la creazione di un ambiente Virtuale utilizzando **Docker.** Docker è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Docker raccoglie il software in unità standardizzate chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime. Con Docker, è possibile distribuire e ricalibrare le risorse per un'applicazione in qualsiasi ambiente, tenendo sempre sotto controllo il codice eseguito. Docker fornisce un sistema operativo per container. Così come la macchina virtuale virtualizza i server hardware (ovvero elimina la necessità di gestirli direttamente), i container virtualizzano il sistema operativo di un server. Docker è installato su ogni server e fornisce semplici comandi con cui creare, avviare o interrompere i container.

Maven:

Come strumento di Build, al posto di Gradle, si è optato per l'utilizzo di **Maven**. Maven è uno strumento di build automation utilizzato prevalentemente nella gestione di progetti Java. Maven si differenzia dagli altri strumenti di Build per quanto concerne la compilazione del codice. Con questo strumento infatti non è più necessaria la compilazione totale del codice, ma viene fatto uso di una struttura di progetto standardizzata su template definita archetype. Il vantaggio

derivante dall'utilizzo di questo tool è da subito evidente: se generalmente per sviluppare un software sono necessarie numerose fasi, con la build automation l'intero processo viene automatizzato, riducendo il carico di lavoro del programmatore e diminuendo le possibilità di errore da parte dello stesso. Alcune delle fasi fondamentali che vengono automatizzate dal tool sono la compilazione in codice binario, il packaging dei binari, l'esecuzione di test per garantire il funzionamento del software, il deployment sui sistemi ed infine la documentazione relativa al progetto portato a termine. Esistono poi alcuni tratti comuni ai prodotti di build automation, a partire dal build file, che contiene tutte le operazioni svolte. Una volta deciso l'Ambiente ed il Build, il gruppo si è dedicato alla creazione della struttura base del progetto.

Vertx:

Per prima cosa si è optato per il Framework **Vert.X**. 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. 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. Un'applicazione Vert.X consiste di uno o più componenti chiamati Verticle, questi, sono parti di codice che il motore di Vert.x esegue. Ogni verticle viene eseguita in maniera concorrente rispetto alle altre e non c'è uno stato che viene condiviso. Quindi si riesce a creare applicazioni multi-threaded senza dover gestire problematiche di concorrenza come la sincronizzazione o i lock tra thread.

Vertx Eventbus:

L'**Event Bus** è il cuore di Vert.X. Tutte le comunicazioni tra Verticle avvengono tramite questo componente, attraverso lo scambio di messaggi. I Verticle possono inviare o mettersi in ascolto di 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. Affinché comunichino fra loro, non siamo obbligati a rilasciare i Verticles sulla stessa JVM. E' sufficiente 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.

Hazelcast:

Visto che l'Event Bus è distribuito, Vert.X offre la possibilità di distribuire i Verticles: tale 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.

Service Discovery:

Per la creazione di un'infrastruttura per pubblicare e scoprire le varie risorse, abbiamo utilizzato **Service Discovery**. Questo componente fornisce risorse come proxy di servizio, endpoint HTTP e tanto altro. 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. Pertanto, le applicazioni possono reagire all'arrivo e alla partenza dei servizi. Successivamente, si è pensato che in un'architettura di questo tipo, alcuni requisiti diventano cruciali, per esempio la scalabilità, il modo in cui vengono rappresentate le funzioni e i dati forniti.

REST:

Tutto questo deve essere reso più semplice possibile, questa è la ragione per cui è nato **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. 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.

API Gateway:

Come punto di accesso al server side, si è pensato alla creazione di una **API Gateway**.

Le API gateway sono lo strumento che ci permette di intermediare i servizi (o microservizi) da esporre in maniera strutturata verso l'esterno. 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. 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. 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. 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. Quando si compone un'applicazione Vert.X, potresti voler isolare una funzionalità da qualche parte e renderla disponibile per il resto dell'applicazione.

ServiceProxy:

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.

SockJS:

Per l'utilizzo delle WebSocket, il gruppo ha pensato all'utilizzo di **SockJS** per sfruttare la sua implementazione sopra l'Event Bus di Vert.X. 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.

Redis:

Per lo Storage dei Record dei Microservizi, il gruppo ha optato per l'utilizzo di **Redis**. Redis è uno store di strutture dati chiave-valore in memoria rapido e open source. Redis offre una serie di strutture dati in memoria molto versatili, che permettono di creare un'ampia gamma di applicazioni personalizzate. I principali casi d'uso per Redis sono il caching, la gestione di sessioni, servizi pub/sub e graduatorie. Si tratta dello store chiave-valore più usato. Dispone di licenza BSD, è scritto in C, il suo codice è ottimizzato e supporta diverse sintassi di sviluppo. Redis è un acronimo e sta per REmote DIctionary Server. Data la velocità e la semplicità di utilizzo, Redis è una scelta molto comune per applicazioni Web, videogiochi, tecnologie pubblicitarie, IoT e app per dispositivi mobili che necessitano di elevate prestazioni.

CircuitBreaker:

Per la gestione degli errori, il gruppo ha optato per il Pattern **Circuit Breaker** che supporta 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.

WebSocket:

Come citato in precedenza, per il protocollo di messaggistica si è optato per le **WebSocket**. 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. 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).

WebToken:

Per le trasmissioni sicure degli oggetti Json, il gruppo ha optato per i **WebToken**. JsonWebToken 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 l'Header, il Payload e la Signature. 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.

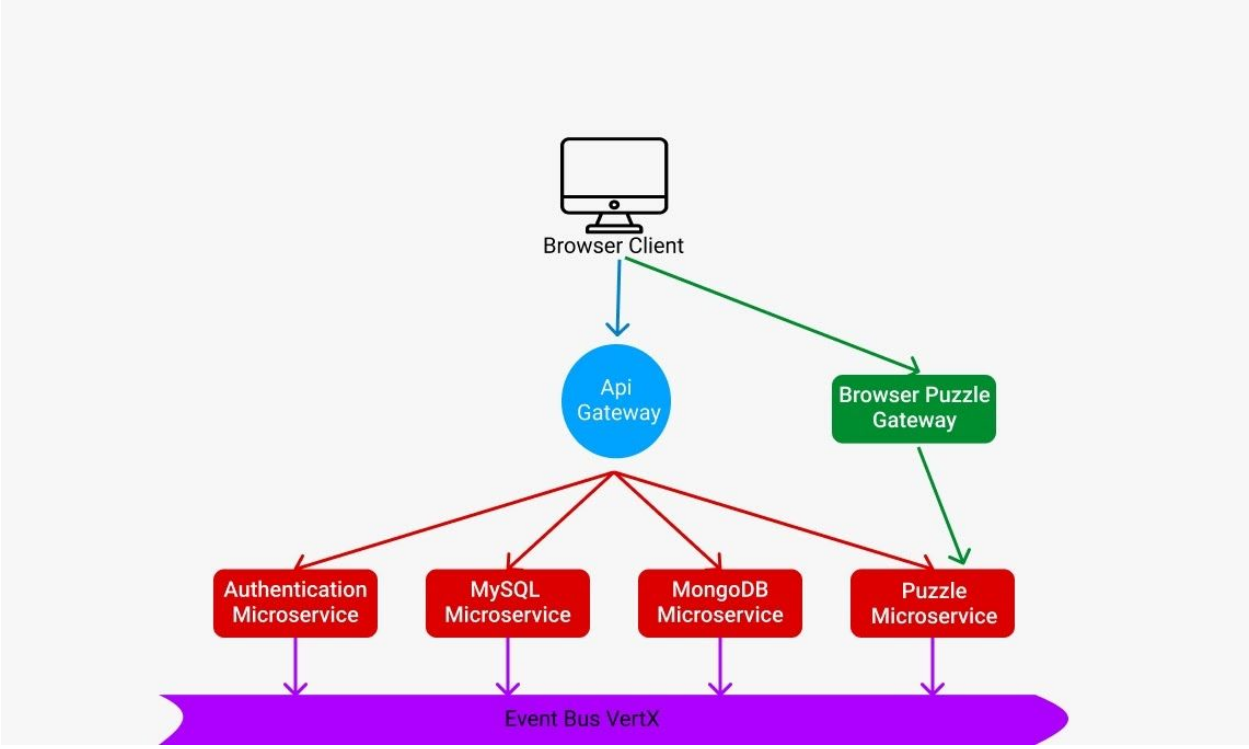
Nel processo di autenticazione quando l'utente accede correttamente utilizzando le proprie credenziali, gli verrà restituito un JWT che dovrà poi includere nell'intestazione di autorizzazione utilizzando lo schema Bearer. Questo può essere, in alcuni casi, un meccanismo di autorizzazione stateless.

Storage (MongoDB MySQL):

Per la memorizzazione dei dati degli utenti, si è optato per la realizzazione di due diversi Database (Per sopportare eventuali fault) **MongoDB** e **MySQL**. 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. 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. Un prodotto software completo, efficiente ed affidabile è MySQL. Un DBMS, come MySQL, 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.

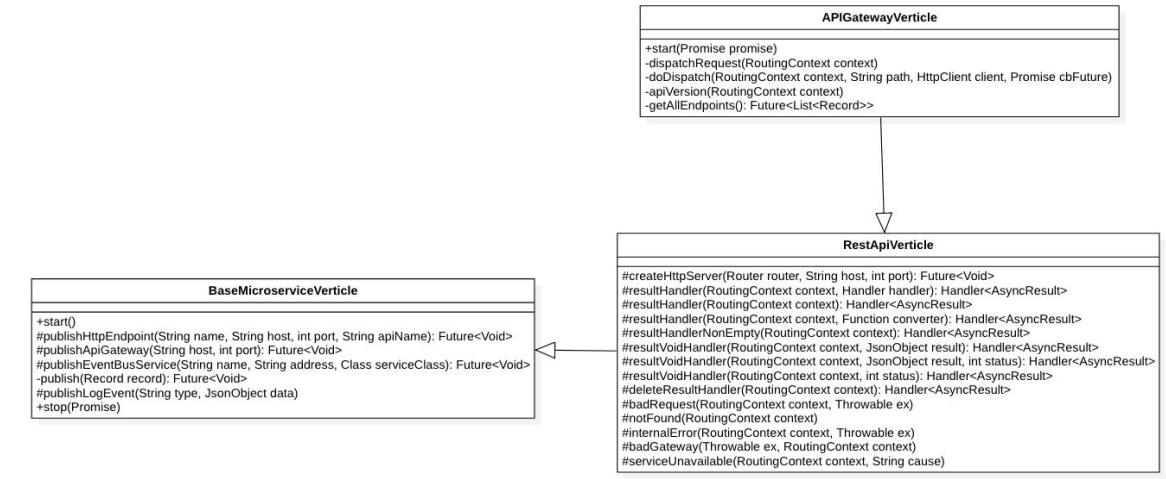
Implementazione del Puzzle:

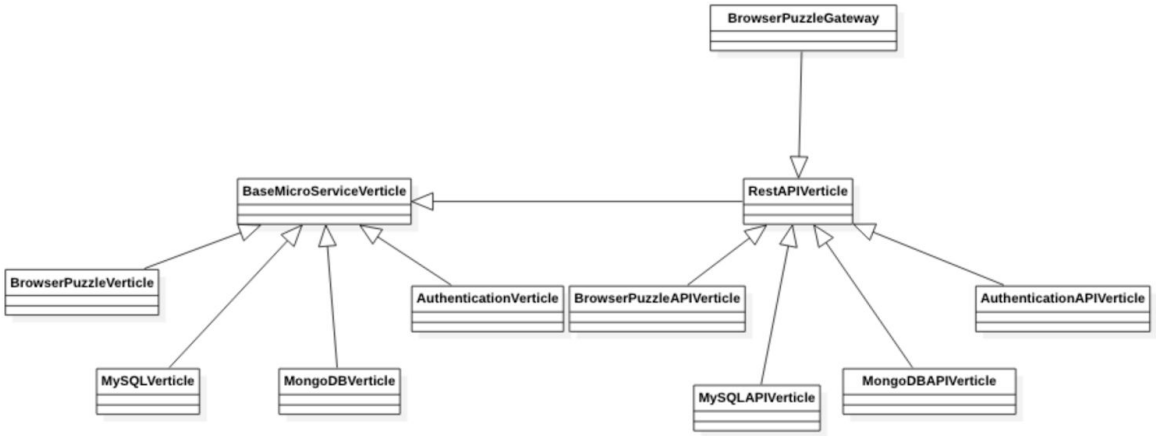
Nell'**Implementazione** realizzata dal gruppo, come citato precedentemente, per la realizzazione di questo progetto si è optato per un'architettura a Microservizi. La struttura finale realizzata consiste in un insieme di Microservizi che possono comunicare tra di loro, tramite scambio di messaggi asincrono sull'eventbus di Vert.X, tramite RPC con l'utilizzo di ServiceProxy (sempre dell'eventbus) oppure tramite Rest. 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 sull'eventbus di Vert.X.



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.

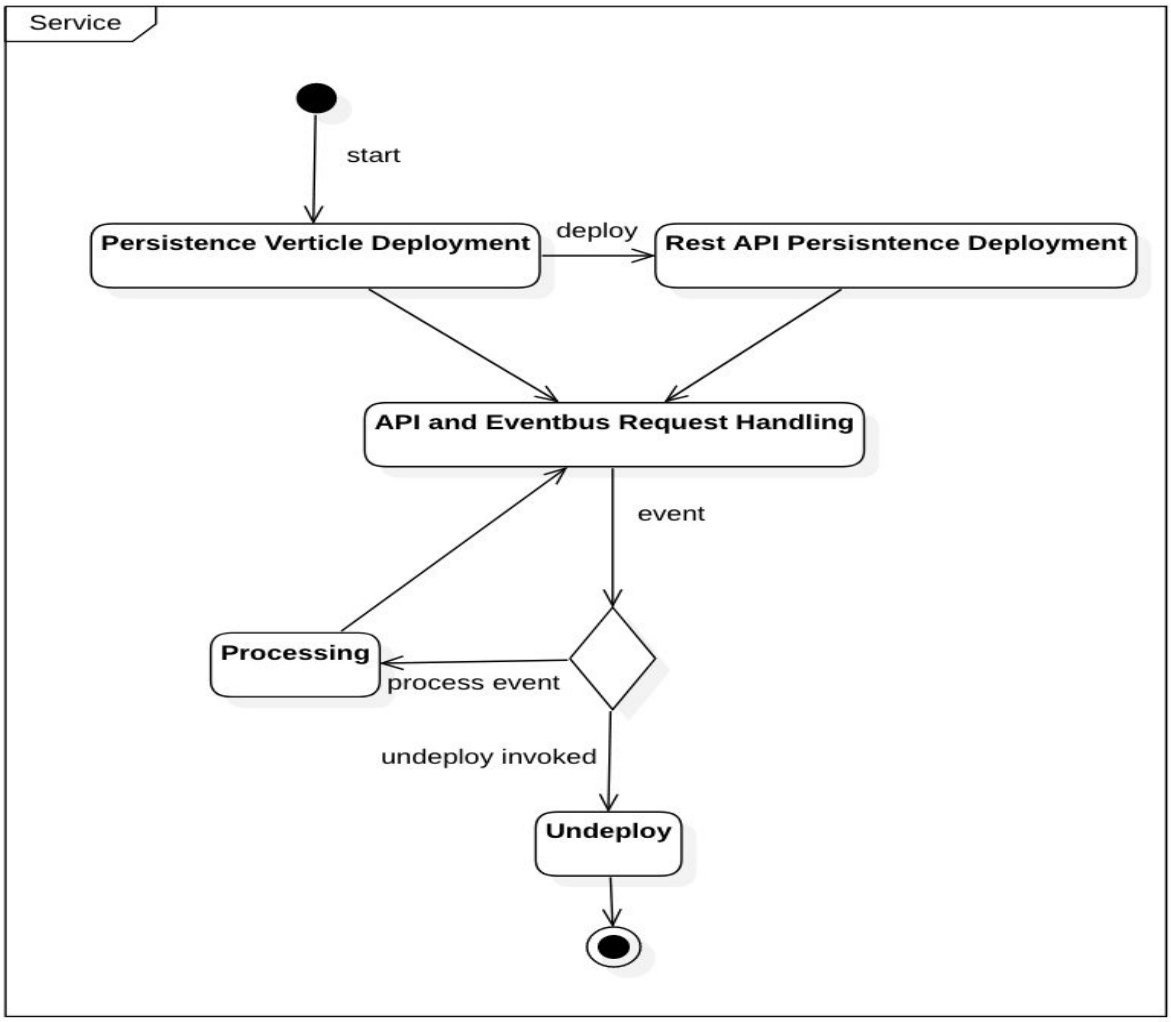
Come citato in precedenza, l'"APIGateway" è un Microservizio che si occupa di ricevere le richieste API dai vari Client, verificare se all'interno del "ServiceDiscovery" sia presente un Microservizio con il target specificato, invocare e processare la richiesta. Se quest'ultima è presente, il Gateway 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.





Comportamento dei Microservizi:

Tutti i Microservizi hanno un proprio flusso di esecuzione basato sull'architettura ad EventLoop e possono interagire tra di loro tramite scambio di messaggi, tramite chiamate RPC (Service Proxy) sull'eventbus o richieste REST. Il Diagramma delle Attività qui sotto, 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.



Microservizio di Autenticazione:

Alla ricezione di un evento che richiede 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, ritornerà 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.

Microservizio del Puzzle:

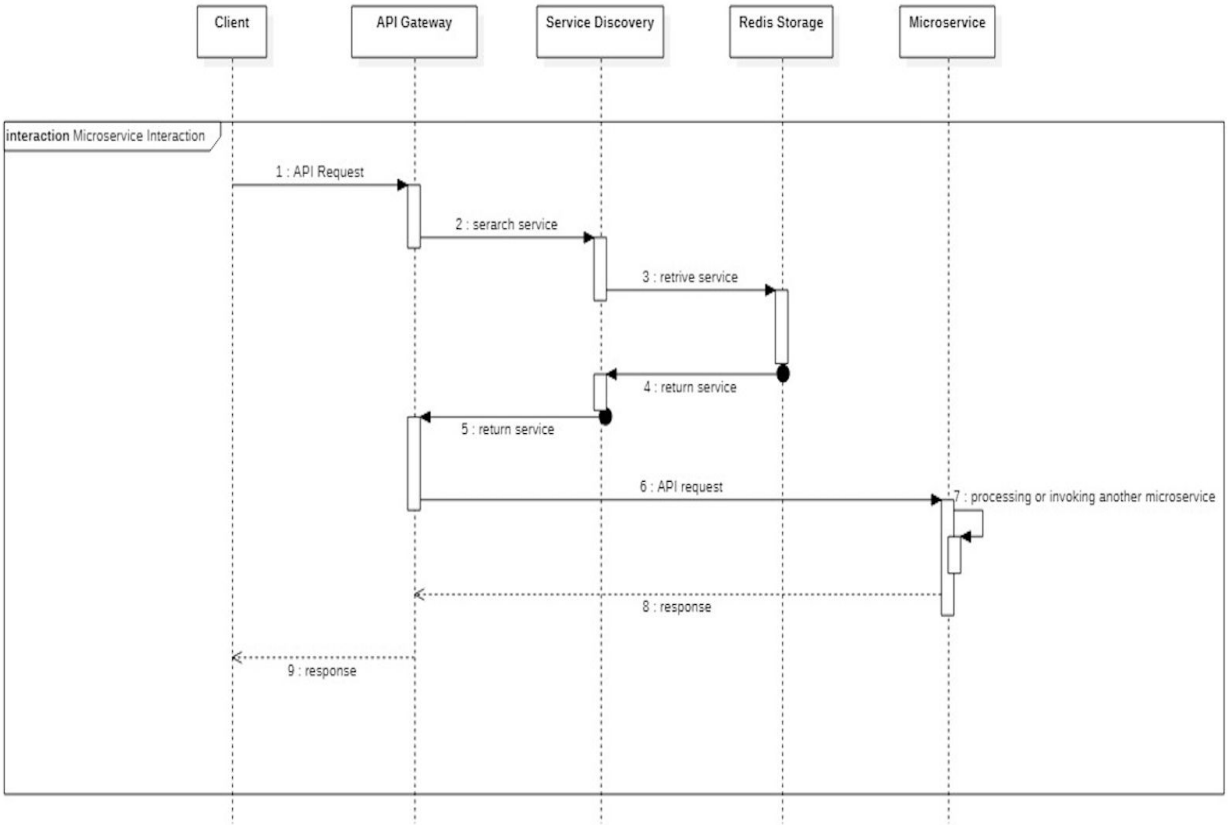
Questo Microservizio si occupa della gestione di tutte le API relative ad una sessione di gioco.
In primo luogo vi è presente una API che permette di creare una partita a partire dall'URL di un immagine in formato ".jpg", il numero delle righe e delle colonne in cui andare a dividere l'immagine e lo username dell'utente che creerà la partita. Alla chiamata di questa API viene recuperata l'immagine dal web e viene divisa in sotto-immagini che poi verranno codificate in "*Base64*" e messe all'interno di un JSON che verrà ritornato al chiamante, il tutto verrà eseguito in maniera Asincrona. In aggiunta, verrà fornita una API che permetterà ad un nuovo giocatore di partecipare ad una sessione di gioco. Questa richiederà l'username dell'utente in modo da poterlo aggiungere e notificare agli altri utenti del gioco connessi, in modo da far vedere le sue mosse (sfruttando un messaggio sull'Eventbus di Vert.X). Inoltre questa API ritornerà al chiamante un JSON con all'interno le immagini precedentemente divise (Operazione effettuata alla creazione della partita) e la loro posizione attuale.
Infine, vi è l'API che permette ad un utente di effettuare lo swap di due pezzi dell'immagine per cercare di ricomporre l'imamgine originale. Questa prende in ingresso la posizione delle due caselle e il server si occupa di modificare la posizione all'interno della propria struttura dati e di comunicare a tutti gli utenti connessi di aggiornare le posizioni delle proprie caselle tramite un messaggio sull'eventbus.

Microservizio di Storage:

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 ed il login, 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 MongoDB o MySQL.

Interazione dei Microservizi:

Per quanto riguarda l'interazione tra i componenti del sistema, si è optato per l'utilizzo della Service Discovery di Vert.X e dell'utilizzo di Redis come database distribuito per mantenere memorizzati i record di ciascun Microservizio. Al momento del deploy di ogni Microservizio, quest'ultimo pubblicherà il suo record presso la Service Discovery che lo andrà a memorizzare su Redis. Ciò consentirà di poter recuperare successivamente questi record e far comunicare/interagire i Microservizi tra di loro. Successivamente il Gateway effettuerà un'operazione di parsing per capire quale Microservizio andare ad invocare in base al path espresso nell'URI. Poi si recupereranno tutti i record di tipo http-endpoint presenti nella Service Discovery (storage Redis) e si verificherà se tra questi è presente un servizio con lo stesso nome di quello desiderato. Se la ricerca va a buon fine il Gateway recupera il record e si trasforma a sua volta in un client HTTP ed effettuerà una richiesta all'API del Microservizio target che andrà a processare la richiesta e a ritornare una risposta asincrona. Se invece non verrà trovato alcun record che possa gestire tale richiesta, verrà ritornato un messaggio di errore al Client con scritto che il servizio al momento non è disponibile.



I microservizi possono comunicare tra di loro in tre modi diversi:

- **RPC (Remote Procedure Call)**: Ciascun Microservizio, oltre ad esporre un'interfaccia API che può essere invocata tramite richieste HTTP, permette anche di invocare queste API come metodi da altri Microservizi. Ciò è reso possibile grazie alla registrazione di un Record di tipo event-bus-proxy presso la Service Discovery. Successivamente un Microservizio potrà andare a recuperare il record nello stesso modo in cui si recupera un record di tipo http-endpoint e utilizzare quel record per invocare i metodi di quel Microservizio.

- **Messaggio sull'Eventbus (Vert.X)**: Grazie all'utilizzo di Hazelcast, che permette di creare dei Cluster e si occupa della gestione delle connessioni TCP tra tutti i componenti del Cluster, sarà possibile creare un eventbus distribuito sul quale possono essere inviati messaggi tra i Microservizi.

- **HTTP request (REST)**: Allo stesso modo del Gateway, si andrà a recuperare il Record del Microservizio con il quale si desidererà comunicare dalla Service Discovery e si effettuerà una richiesta HTTP a quest'ultimo.

Design Pattern Utilizzati:

Il gruppo ha optato per l'utilizzo dei pattern **SOA, REST, APIGateway, CircuitBreaker**:

- **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. La chiave sta nella totale assenza di business logic sul client SOA, il quale è totalmente agnostico rispetto alla piattaforma di implementazione, riguardo ai protocolli, al binding, al tipo di dati, alle policy con cui il servizio produrrà l'informazione richiesta. Tutto questo a beneficio dell'indipendenza dei servizi, che possono essere chiamati per eseguire i propri compiti in un modo standard, senza che il servizio abbia conoscenza dell'applicazione chiamante e senza che l'applicazione abbia conoscenza, o necessiti di averne, del servizio che effettivamente eseguirà l'operazione.

- **REST** (Representational State Transfer) è uno stile architetturale che si basa su HTTP. Il funzionamento prevede una struttura degli URL ben definita che identifica univocamente una risorsa o un insieme di risorse e l'utilizzo dei verbi HTTP specifici per il recupero di informazioni (GET), per la modifica (POST, PUT, PATCH, DELETE). Un numero qualsiasi di connettori (client, server, cache, tunnel ecc.) può mediare la richiesta alle risorse. Ogni connettore interviene senza conoscere la "storia passata" delle altre richieste. Proprio per questo motivo l'architettura REST si definisce stateless. Di conseguenza, un'applicazione può interagire con una risorsa conoscendo due cose: l'identificatore della risorsa e l'azione richiesta. Non serve sapere se ci sono proxy, gateway, firewall, tunnel, altri meccanismi intermedi tra essa ed il server in cui è presente l'informazione necessaria. L'applicazione comunque deve conoscere il formato dell'informazione restituita, ovvero la sua rappresentazione. Tipicamente è un documento HTML, XML o JSON, potrebbero essere anche immagini o altri contenuti.

- **APIGateway** è un pattern che introduce un singolo punto di ingresso per tutti i client. E' simile al pattern Facade della programmazione ad oggetti, ma in questo caso, è parte di un sistema distribuito. L'APIGateway pattern è spesso conosciuto come "backend for frontend" (BFF) perché viene implementato pensando ai servizi di cui il client ha bisogno. Quindi l'APIGateway si trova tra le app client ed i microservizi. Funziona come proxy inverso, instradando le richieste dai client ai servizi. Può anche fornire funzionalità trasversali aggiuntive, come l'autenticazione, la terminazione SSL e la cache.

- **Circuit Breaker** è un design pattern che viene utilizzato per rilevare guasti e incapsula la logica di impedire che un guasto si ripeta costantemente. Infatti, quando il numero di guasti consecutivi supera una soglia, il Circuit Breaker scatta e per la durata di un periodo di timeout tutti i tentativi di invocare il servizio remoto falliranno immediatamente. Allo scadere del timeout, l'interruttore automatico consente il passaggio di un numero limitato di richieste di test. Se tali richieste hanno esito positivo, il Circuit Breaker riprende a funzionare normalmente. Altrimenti, se si verifica un errore, il periodo di timeout ricomincia.

Istruzioni per il Deployment del Puzzle:

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*
- *mvn test (nel caso si voglia fare il testing del sistema)*

A questo punto sarà sufficiente aprire il browser e collegarsi al seguente indirizzo:
http: //localhost:8080.
Dopo aver effettuato l’accesso la schermata sarà simile a quella mostrata qui di seguito.