POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema
e Meccatronica

Corso di Laurea Magistrale in
Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Progetto ed implementazione di un'architettura in tempo reale per il supporto di videogiochi multiplayer

Relatore
Prof. Giovanni Malnati

Candidato
Carmelo Riolo

Aprile 2018

Titolo

Progetto ed implementazione di un'architettura in tempo reale per il supporto di videogiochi multiplayer

Autore

Carmelo Riolo

Relatore

Prof. Giovanni Malnati
Dipartimento di Automatica e Informatica
Politecnico di Torino

Sommario

Questa tesi è incentrata sullo sviluppo di una componente software a supporto di un videogioco per dispositivi mobili, il cui nome è Mak07. Questo è un gioco basato su una forte componente matematica e logica, il quale rientra nella categoria dei videogiochi rompicapo. L'idea di base è sfruttare la matematica per intrattenere il videogiocatore: partendo da una combinazione di sette numeri, questi deve applicare tutte le possibili operazioni aritmetiche al fine di ottenere come risultato finale il valore zero. Più brillante è la risoluzione dello schema, sia in termini di operazioni che di tempo impiegato, maggiore sarà il punteggio ottenuto dall'utente. Risolto una schema sarà possibile passare allo schema successivo, il tutto entro due minuti. Maggiore sarà il numero di schemi risolti, maggiore sarà l'esperienza ed i punti acquisiti dal giocatore. La prima versione prevedeva un'unica modalità di gioco, nella quale si aveva solo la possibilità di avviare una partita e risolvere una successione di schemi. Tuttavia, tale modello di gioco non va incontro a quelle che ad oggi risultano le aspettative dell'utenza, che vuole potere interagire e giocare con i propri amici o più in generale con altri avversari in rete. Da qui l'esigenza di realizzare un'infrastruttura tale da permettere una modalità multigiocatore, progettando ed implementando l'architettura lato server, che consenta l'identificazione di un potenziale avversario tra quelli attualmente presenti online, la gestione di sfide tra due giocatori e l'eventuale ricorso a bot per supportare quei momenti della giornata in cui non vi è un numero sufficiente di giocatori connessi.

Progetto dell'architettura

Dopo una prima fase di studio, si è scelto di implementare un'architettura di tipo REST ed utilizzare un framework per lo sviluppo di applicazioni su piattaforma Java quale Spring, per via della sua natura basata sull'accoppiamento

debole dei dati e del gran numero di librerie esistenti che permettono di estenderne le funzionalità. Per la memorizzazione e la persistenza dei dati è stato utilizzato un sistema per la gestione di basi di dati non relazionale, e la scelta è ricaduta su MongoDB. Tale scelta, rispetto ad un classico modello relazionale, è supportata da diversi motivi tra i quali principalmente le maggiori possibilità di scalabilità e l'assenza di uno schema rigido, che in un contesto come questo ha rappresentanto un grande vantaggio, in quanto ci si è trovati più volti a dovere modificare la struttura dei documenti in corso d'opera. Lo sviluppo ha seguito diverse fasi, che miravano a implementare le varie parti del sistema software. Inizialmente ci si è resi conto della necessità di implementare un meccanismo di autenticazione degli utenti, che inizialmente è stato basato solo su Facebook o Google, sfruttando il protocollo OAuth. In un secondo momento è stato implementato anche il classico modello di autenticazione tramite indirizzo email e password, inserendo tutte le funzionalità accessorie legate allo smarrimento e recupero della password. Una volta data all'utente la possibilità di registrarsi ed accedere al sistema, è seguito lo sviluppo di un meccanismo tale da permettere ai giocatori di sfidare i propri amici. Durante questa fase sono stati affrontati e risolti diversi problemi come la gestione della concorrenza e la garanzia del risultato finale di ogni sfida. Tale modello di sfida si basa su inviti scambiati tra giocatori, quindi presuppone l'attendere che l'avversario accetti la sfida prima di potere giocare e risolvere i vari schemi. Tale operazione, apparentemente banale, diventa critica adottando un DBMS come MongoDB che non gestisce le transazioni. Accanto a questo è stato realizzato un meccanismo di e-presence, tale da permettere ad un utente di giocare sul momento una data partita cercando un avversario tra quelli connessi al gioco. Ciò è stato ottenuto grazie all'utilizzo della tecnologia Websocket unita ad un broker di messaggi come RabbitMQ, con il risultato di potere cercare un avversario tra tutti quelli attualmente connessi, ossia che stiano eseguendo in quel dato momento il gioco e non siano impegnati in alcuna sfida. In assenza di utenti connessi, condizione non così rara soprattutto nelle prime fasi di avvio del prodotto, la sfida viene inoltrata ad un bot, ossia un finto giocatore, programmato in maniera tale da giocare la partita in modo congruo con il livello di capacità del suo concorrente umano. Infine, una volta in produzione è stato necessario analizzare e monitorare l'utenza registrata e le varie statistiche di gioco, per cui è stato realizzato anche un portale web dedito a questo scopo. L'ultima fase del lavoro ha riguardato la configurazione del server di produzione. In particolare, è stata utilizzata una macchina basata su sistema operativo Linux ed all'interno del quale, tramite la tecnologia Docker, sono stati configurati quattro diversi "container" contenenti server di gioco, database, broker di messaggi e pannello di amministrazione, come mostrato in figura.

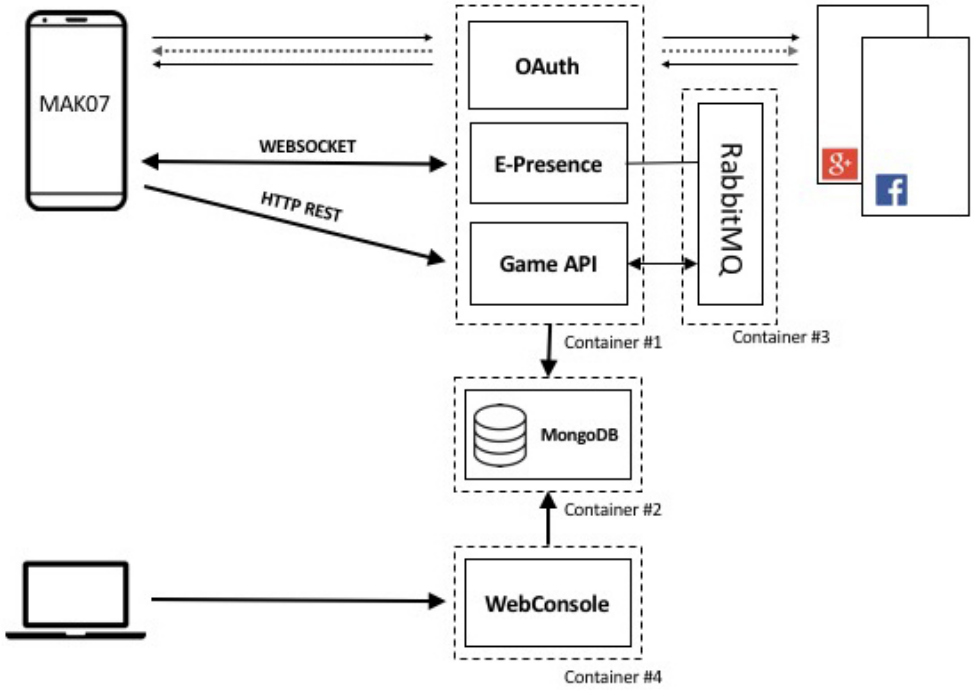


Figura 1: Architettura del sistema realizzato

Risultati e sviluppi futuri

In conclusione, tutti gli obiettivi preposti sono stati realizzati, e la nuova versione del gioco Mak07 è stata rilasciata in produzione a Dicembre 2017 ed ha registrato fino a questo momento oltre un migliaio di utenti. Tra gli obiettivi futuri vi è l'introduzione di una terza modalità di gioco, secondo uno schema a torneo, composto da diversi gironi, che prevede la partecipazione di un certo numero di giocatori che può variare da 8 a 16. Inoltre accanto a questa è possibile l'integrazione di un server di ricerca quale ElasticSearch, pienamente supportato dal framework Spring, al fine di migliorare il sistema di ricerca utenti, ed il riutilizzo delle tecnologie Websocket e RabbitMQ così da realizzare un meccanismo di messaggistica istantanea tra gli utenti. Infine potrebbe essere necessario in futuro predisporre più risorse, e scalare orizzontalmente, qualora continui a crescere il numero di utenti attivi e di conseguenza le richieste verso il server, motivo per cui sono state utilizzate tecnologie come Docker e MongoDB, che si prestano a realizzare ciò.

Ringraziamenti

Di seguito vorrei ringraziare tutte le persone che in un modo o nell'altro mi sono state accanto durante questi anni, anche se il primo e più importante ringraziamento va alla mia famiglia che mi ha sempre sostenuto ed ha sempre creduto in me, e senza la quale non sarei mai arrivato fino a qua.

Ai miei genitori perchè è solo grazie ai loro sacrifici che ho potuto realizzare tutto ciò.

A mia sorella Cristina e mio cognato Lino, in quanto l'averli vicino durante gli ultimi due anni mi ha permesso di sentirmi a casa ed in famiglia nonostante la lontananza.

A mia sorella Barbara e mio cognato Massimo, per avermi regalato una bellissima nipote.

A mia nipote Aurora, che mi regala sempre un sorriso ed il solo vederla mi riempie il cuore di gioia.

Ai miei amici, quelli di sempre, quelli che sono quasi dei fratelli, con cui sono cresciuto e ho condiviso gli anni migliori della mia vita.

Ai miei amici, colleghi ed anche coinquilini, con cui ho vissuto questi due anni di avventure a Torino, che hanno saputo rendere speciale e divertente anche la vita di tutti i giorni.

Ai miei amici di infazia, con cui sono cresciuto giocando sotto casa, ancora presenti dopo tantissimi anni.

Al Professore Giovanni Malnati ed a tutti i ragazzi di TonicMinds, che mi hanno accompagnato ed aiutato durante il mio percorso di tesi, con i quali ho condiviso dei bellissimi momenti sia dentro che fuori dal laboratorio.

Indice

Elenco delle figure

Parte I

La progettazione

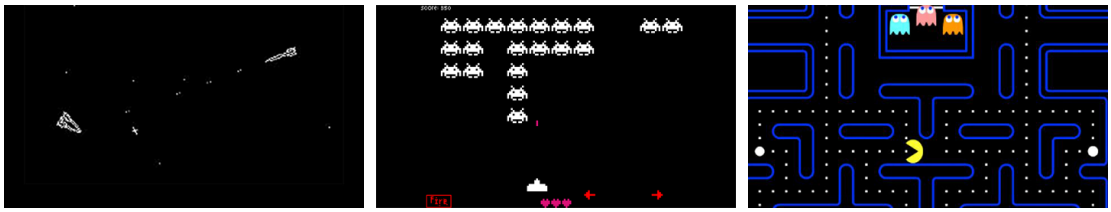
Capitolo 1

Introduzione

Obiettivo di questa tesi è stato lo sviluppo di un'infrastruttura a suppporto di videogiochi multiplayer. L'organizzazione dell'elaborato è divisa in maniera tale da descrivere inizialmente quello che è ad oggi il settore videoludico e quella che è stata la sua evoluzione nel tempo, per poi analizzare l'idea alla base del videogioco Mak07 e l'architettura realizzata a supporto di quest'ultimo. Segue successivamente l'analisi di tutte le moderne tecnologie utilizzate e le principali scelte effettuate nell'implementazione del progetto stesso.

1.1 Storia ed evoluzione dei videogiochi

I primi videogiochi risalgono agli anni cinquanta ed erano legati principalmente a progetti di ricerca all'interno delle università e grandi aziende. E' questo il caso di *Tennis for two*, realizzato nel 1958 da Willy Higinbotham, il quale lavorava come ingegnere al *Brookhaven National Laboratory* e realizzò un gioco che pilotato tramite pulsanti permetteva di simulare i rimbalzi di una pallina su un campo da tennis. Tuttavia il primo videogioco che la storia ricordi è *Spacewar*, realizzato nel 1962 da Steve Russel, studente del *Massachussets Institute of Technology (MIT)*. Ciò che differenzia i due ed è il motivo della maggiore fama del secondo rispetto al primo è l'elemento di sfida. Infatti, oltre a presentare un mondo dotato di regole fisiche, Spacewar non comprendeva una componente di intelligenza artificiale, ma permetteva a due giocatori umani di scontrarsi tra loro. Ogni giocatore pilotava un astronave e l'obiettivo era quello di colpire l'avversario con un missile e vincere il round. Steve Russell pensò anche alla possibilità di commercializzare il suo prodotto ma non riusciva ad immaginare nessuna persona disposta a giocarci, anche se anni dopo fu proprio tale commercializzazione che permise ai videogiochi di diventare un fenomeno di cultura, sempre più diffuso ed evoluto negli anni. Nel 1966 Ralph Baer, ingegnere in una azienda elettronica, inventò un congegno da collegare al televisore per cimentarsi in un gioco di ping-pong, prototipo poi acquisito dalla Magnavox, società che verrà a sua volta acquisita dalla Philips. Sempre la Magnavox nel 1972 realizzò *Odyssey*, primo esperimento, molto rudimentale, di console per videogiochi, che non ebbe molto successo. Contemporaneamente in tale anno nacqua la Atari, società produttrice di vi-



(a) Spacewar (b) Space-Invaders (c) Pac-man

deogiochi e hardware destinato al mondo videoludico, che subito spopolò grazie al videogioco *Pong*, copia del ping-pong di Ralph Baer, e successivamente entrò anche a casa delle persone con la versione domestica di nome *Home Pong*, la quale vendette più di 150.000 unità. Il 1978 vide la nascita di *Space Invaders*, il quale fu uno dei videogiochi più influenti della sua generazione, con un fatturato di oltre 500 milioni di dollari. Novità apportata da quest'ultimo fu l'introduzione di una narrazione ed una trama, in quanto il giocatore si trovava ad impersonare per la prima volta il ruolo dell'eroe e salvare la terra dall'invasione aliena. Quando si eliminavano tutti gli alieni, seguivano ondate sempre più veloci finché non si soccombeva all'invasione. Non vi era quindi una fine del gioco. Per ogni alieno colpito il punteggio aumentava, in tal modo il gioco invogliava a riprovare per superare il precedente record di punteggio. Altro gioco che segnò un'epoca fu *Pac-Man*, il cui omonimo protagonista era rappresentato da una creatura sferica gialla che attraverso diversi livelli, sempre più difficili, si trovava a dovere completare vari labirinti mangiando tutti i puntini presenti in essi, cercando di non farsi mangiare a sua volta dai fantasmi nemici. Tra la fine degli anni '70 e gli inizi degli '80 nacquero i primi personal computer a fare da concorrenti al settore delle console, come il *Commodore 64*, che offriva il vantaggio di poter eseguire anche altri programmi oltre ai giochi. In generale, con l'avanzare degli anni e dell'innovazione tecnologica la grafica e complessità dei giochi aumentò sempre più ed iniziarono a nascere modelli di console sempre più evoluti e nuove tipologie di videgiochi, sino ad arrivare alle console ed i videogiochi di ultima generazione.

1.2 Classificazione di videogiochi

Il primo tentativo di classificare diverse tipologie di videogiochi risale al 1984 quando Chris Crawford pubblicò il suo libro dal titolo *"The Art of Computer Game Design"*. All'interno di quest'ultimo, l'autore cercò di effettuare una prima classificazione dei videogiochi esistenti all'epoca, nonostante fosse comunque consapevole che questa sarebbe diventata obsoleta col passare del tempo a causa dei cambiamenti a cui tale settore è costantemente soggetto. Infatti con il passare degli anni grazie alla crescente innovazione tecnologica sono nati nuovi generi di videogiochi del tutto diversi da quelli che erano i *platform* degli anni '80. Infatti, a partire dagli anni '90 grazie all'avvento dei primi modelli di console domestiche moderne, basate su architetture a 32 e 64 bit,come la *Playstation*, arrivarano sullo schermo i primi videogiochi 3D, una assoluta novità rispetto ai platform 2D di qualche anno prima, con la possibili-

tà di utilizzare anche periferiche del tutto nuove. Inoltre, grazie alla diffusione di internet fu anche possibile iniziare a giocare *online* con i propri amici a distanza, o sfidare magari un giocatore dall'altra parte del mondo.
Ad oggi è possibili distinguere le seguenti macrocategorie di videogiochi, ognuno a sua volta con le sue sottocategorie: [2]

- **Avventura**. Una tra le prime categorie di videogiochi che è stata inventata, caratterizzata dall'esplorazione ed interazione con l'ambiente circostante, ed incentrata sulla narrazione piuttosto che sulle sfide.

- **Azione**. I videogiochi di azione sono quelli in cui il risultato dipende dall'abilità del giocatore, dalla coordinazione occhio-mano insieme all'abilità di gioco stessa, acquisita con l'esperienza. I giochi *platform* o di combattimento sono tra i più conosciuti sottogeneri di tale categoria.

- **Giochi di ruolo**. Conosciuti anche con la sigla GDR. In generale si focalizzano su uno o più personaggi che devono seguire una certa trama e che nel proseguo di questa acquisiscono abilità sempre maggiori in maniera tale da affrontare nemici sempre più forti. Una sottocategoria di tali giochi, attualmente molto diffusa è quella dei GDR online come i *"Massively Multiplayer Online Role Playing Game (MMORPG)"*.

- **Simulatori**. In generale tutti quei videogiochi che permettono di simulare accuratamente una esperienza reale, come i simulatori di volo, o come il famoso simulatore di vita di nome *The sims*.

- **Quiz**. La classica categoria alla quale appartengono i puzzle, gli enigmi ed i rompicapo.

- **Sport**. Appartengono a tale categoria tutti i videogiochi sportivi, dai videogiochi di calcio a quelli di basket, golf, etc.

- **Strategia**. I videogiochi di tale categoria richiedono una esperienza di gioco che in generale include un'accurata pianificazione e concentrazione per raggiungere la vittoria.

L'avvento di internet ha reso possibile interagire con altri giocatori online, ma è possibile condividere l'esperienza di gioco anche in altri modi, come ad esempio l'utilizzo di più *controller* collegati alla stessa *console*, dividendo lo schermo in due o quattro parti. Alcuni videogiochi permettono sia la modalità giocatore singolo sia la modalità multigiocatore, altri solamente l'una o l'altra. Tramite internet persone fisicamente distanti possono connettersi ad uno stesso server di gioco e partecipare alla stessa partita, anche se l'esperienza di gioco talvolta può essere influenzata negativamente da una bassa velocità di connessione. Molte tipologie di gioco, appartenenti alle macrocategorie precedentemente elencate, possono essere giocate in multiplayer.

1.3 Il fenomeno del “Mobile Gaming”

Viene definito *mobile game* un videogioco destinato a telefoni cellulari, smartphone e tablet, o un qualsiasi dispositivo di questo genere. Icona di questa categoria è il videogioco *Snake* lanciato da Nokia nel 1997 e preinstallato nella maggiore parte dei suoi dispositivi cellulari. E’ stato uno dei videogiochi più giocati di questa categoria, installato su più di 350 milioni di dispositivi. Ad oggi l’evoluzione dei telefoni cellulari in smartphone e la sempre maggiore diffusione di questi ha portato diversi cambiamenti anche all’interno del settore videoludico. Una maggiore potenza di elaborazione ha permesso la realizzazione di mobile game sempre più sofisticati, sia in termini di grafica che di funzionalità, attirando sempre più utenti. “Il mercato dei giochi per smartphone, ha realizzato una crescita considerevole nel corso di soli dodici anni, incrementando il proprio fatturato da 20 a 211 miliardi di dollari nel solo mercato americano” [4]. Tuttavia all’interno di tale contesto gli sviluppatori si trovano a lavorare su un dispositivo nato per comunicare e non per giocare. Quello che cambia è quindi anche il target, che non è il classico target di riferimento per i videogiochi, ma uno che include persone di differente età, sesso, classe sociale e cultura. Sono diverse anche le software house videoludiche, come Nintendo, che hanno dichiarato di volere sfruttare l’ampio bacino dei giochi mobile come una sorta di trampolino di lancio, una pubblicità per invogliare anche le persone più distanti al mondo videoludico a comprare tutti gli altri loro prodotti [9]. Inoltre la maggiore parte di questi titoli sono grauiti, o ad un costo di pochi euro, e permettono di essere giocati o completati senza nessun acquisto. Ma talvolta sono programmati in maniera tale da permettere acquisti esterni al fine di avvantaggiare l’utente durante la sua esperienza di gioco. Sebbene tale formula non sembri delle migliori, i risultati positivi in termini economici sono notevoli. In particolare è possibile riportare il successo di tale trend di videogiochi ai seguenti motivi: [7]

- essendo i dispositivi cellulari multifunzione, questo permette di incontrare il favore sia degli appassionati sia di chi li acquista per altre ragioni;
- potere giocare in qualsivoglia momento o situazione;
- nessuna limitazione territoriale, permettendo così agli sviluppatori di raggiungere mercati prima difficilmente penetrabili;
- modello *free-to-play* con acquisti opzionali;
- innovazione tecnologica continua di smartphone e tablet.

Secondo diverse stime entro il 2020 il *mobile gaming* rappresenterà poco più della metà del mercato dei giochi disponibili.

Capitolo 2

Requisiti del sistema

All'interno di questo capitolo è descritta l'architettura generale del sistema con particolare attenzione rivolta all'infrastruttura server, obiettivo di questo elaborato. In particolare, dopo avere introdotto l'idea alla base del videogioco Mak07 ed i vari componenti software del sistema nel suo complesso, saranno analizzati i requisiti tecnologici che un'infrastruttura a supporto di un videogioco multigiocatore deve rispettare. Requisiti che sono sempre stati tenuti in considerazione durante tutte le fasi di progettazione e sviluppo.

2.1 Mak07

Mak07 è il nome che è stato dato al videogioco oggetto di questa tesi. L'idea alla base di questo è sfruttare la matematica per intrattenere il videogiocatore, il quale partendo da uno schema di sette numeri deve applicare tutte le possibili operazioni aritmetiche, quindi sommare, sottrare, dividere e moltiplicare al fine di ottenere zero come risultato e così passare allo schema successivo. Più brillante è la risoluzione della schema, sia in termini di operazioni che di tempo impiegato, maggiore sarà il risultato ottenuto dall'utente. Risolto una schema sarà possibile passare allo schema successivo; il tutto entro due mi-

Figura 2.1: Mak07

nuti. Maggiore sarà il numero di schemi risolti, maggiore sarà l'esperienza ed i punti acquisiti dal giocatore. Ogni giocatore dispone di una propria pagina profilo in cui è possibile consultare l'esperienza guadagnata e tutte le statistiche di gioco. Il gioco si basa su una forte componente multiplayer che permette di sfidare altri avversari in rete, invitando direttamente i propri amici o cercando avversari casuali tra quelli connessi, così da accumulare punti ed avanzare in classifica.

2.2 Analisi dei requisiti

Ad oggi, sono sempre più i videogiochi che basano il loro successo sulla componente multigiocatore. In particolare, sempre più utenti vogliono disporre della possibilità di potere giocare remotamente con i loro amici, così da godere appieno dall'esperienza di gioco. Quindi è necessaria l'introduzione di una forte componente *social*. In generale, tutto ciò si traduce nella necessità di dovere pensare ad un sistema in grado di accogliere una certa mole di utenza, predisporre un meccanismo di registrazione ed autenticazione all'interno del sistema, e distinguere diversi livelli di autorizzazione, in maniera tale da discriminare le operazioni permesse e quelle non permesse. E' inoltre necessario tenere traccia delle operazioni dell'utente, in termini di partite giocate o schemi risolti, tempo di gioco, avversari sfidati o quant'altro in maniera da realizzare una sorta di profilazione dell'utenza. Trattandosi di un videogioco multigiocatore, in cui il risultato dipende dall'attività di più utenti, deve essere garantita la correttezza del risultato finale, evitando situazioni in cui venga registrato un risultato sbagliato, che oltre all'errore in se porta ad influenzare in maniera errata le statistiche di gioco. Il videogioco è stato pensato per un contesto mobile ed è quindi necessario un meccanismo di notifica, al fine di avvisare il giocatore sulla presenza di una nuova sfida o altri eventi all'interno del gioco. Accanto alla possibilità di sfidare i propri amici, tramite opportuni inviti, è necessario predisporre un meccanismo per tenere traccia dei giocatori attualmente connessi, così da potere realizzare un sistema di sfide casuali tra giocatori *online*. Sopratutto nelle prime fasi di esistenza del gioco, la presenza di utenti connessi può risultare scarsa o addirittura nulla, tale che un utente connesso potrebbe non trovare sfidanti con cui potere giocare. Si è quindi ritenuto importante l'introduzione di diversi *bot player* a cui è affidata la responsabilità di giocare una partita con l'utente qualora non vi siano giocatori disponibili. Sulla base di quanto detto, nella realizzazione dell'infrastruttura server a supporto dell'intero sistema, si è scelto di perseguire le seguenti caratteristiche:

- flessibilità: il sistema deve essere realizzato in maniera tale da potere essere modificato in corso d'opera ed essere adattato facilmente a nuovi scenari, quali l'aggiunta di nuove funzionalità, compatibili con quelle già esistenti, o la possibilità di modificare facilmente queste ultime senza stravolgere l'intera architettura realizzata;

- prestazioni: è necessario avere un sistema performante, in maniera tale da non rovinare l'esperienza di gioco dell'utente, che riesca a soddisfare molte richieste contemporaneamente da parte di utenti diversi, garantendo sempre la correttezza del risultato;

- scalabilità: tale sistema deve essere in grado di accogliere una grande mole di utenza. Tuttavia questa può trovarsi a diminuire o aumentare nel tempo. Nel secondo caso è necessario che il sistema sia scalabile così da potere predisporre più risorse e soddisfare un numero più elevato di richieste;

- sicurezza: tutte le informazioni relative ai videogiocatori e le loro attività all'interno del gioco sono memorizzate in maniera persistente all'interno del database. E' quindi necessario garantire la confidenzialità di queste tramite opportuni meccanismi di cifratura;

- qualità del codice: è opportuno seguire l'adozione di *best practice* e *pattern* consolidati nell'ingegneria del software così da realizzare codice manutenibile, testabile e scalabile.

2.3 Descrizione generale

L'architettura generale del sistema realizzato è tale per cui il videogioco in esecuzione su un dispositivo mobile, in ambiente Android o iOS, effettua la maggiore parte della comunicazione con il server di gioco tramite l'utilizzo del protocollo ReST, quindi tramite l'invocazione di metodi HTTP, contattando opportune API, come mostrato in Figura 2.2, la quale riassume l'architettura generale. Essendo al di fuori degli scopi di questo elaborato la descrizione circa la prima metà della Figura 2.2 sarà tralasciata. Le restanti comunicazioni invece avvengono tramite Websocket. Si è deciso di utilizzare il meccanismo delle Websocket, al fine di integrare il meccanismo delle sfide casuali tra giocatori connessi, in maniera tale da potere tenere traccia degli utenti effettivamente connessi al sistema, ossia quelli tali per cui l'applicazione è attualmente

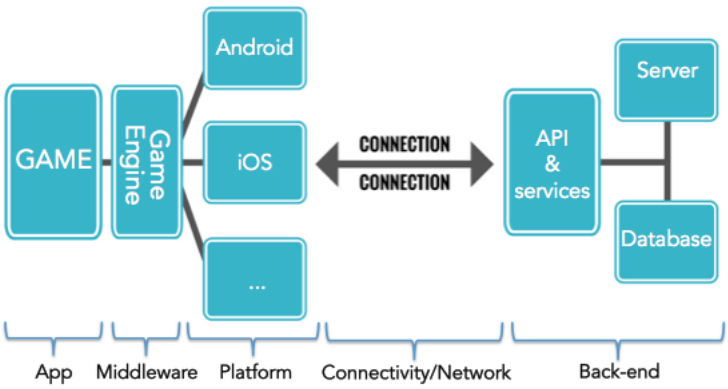


Figura 2.2: Architettura generale

in esecuzione ed in *foreground* all'interno del dispositivo mobile, ma ove non vi sia alcuna sfida in corso. Un utente può autenticarsi all'interno del sistema o tramite autenticazione *social*, appoggiandosi ad un *authorization server* di terze parti, in particolare Facebook o Google, oppure tramite la classica registrazione con email e password. In entrambi i casi, una volta autenticato l'utente avrà la possibilità di scegliere un *nickname* univoco all'interno del sistema. Tutte quelle che sono le validazioni dell'input utente sono eseguite sia lato client che lato server al fine di garantire la robustezza del sistema globale. All'utente è consentito l'accesso contemporaneo alla piattaforma di gioco da parte di più dispositivi, e le varie sessioni sono gestite separatamente, anche se opportuni controlli sono effettuati al fine da vietare alcune operazioni che potrebbero alterare l'esperienza di gioco, come la possibilità di giocare la stessa sfida in contemporanea su due dispositivi. Tutte le informazioni ed attività di gioco vengono memorizzate all'interno di un database non relazionale quale MongoDB. Il web server è rappresentato da Apache Tomcat, il quale è contenuto all'interno dell'applicazione Spring Boot che esegue il server di gioco. E' stato acquisito un certificato SSL ed utilizzato il protocollo HTTPS per garantire la confidenzialità della comunicazione. Per gestire la notifica di un particolare evento verso il dispositivo dell'utente si è scelto di appoggiarsi ad un servizio esterno, quale Google Firebase. Infine per la gestione ed inoltro dei messaggi tramite Websocket si è deciso di utilizzare RabbitMQ come broker di messaggi insieme al protocollo STOMP.

2.4 Altre componenti del sistema

Accanto al client mobile ed al server di gioco è stato successivamente realizzato anche un pannello di amministrazione, al fine di monitare l'andamento del gioco in termini di utenti registrati e sfide giocate, così come l'accesso alla classifica generale. Segue una breve panoramica sulle altre componenti del sistema, la cui importanza è secondaria relativamente agli scopi di questo elaborato.

2.4.1 Client mobile

Il gioco *mobile* è stato realizzato in maniera *cross-platform*, così da essere supportato sia dal sistema operativo *Android* sia da quello *iOS*. Ciò è stato possibile grazie all'utilizzo della librearia *libgdx*. Quest'ultima rappresenta un framework open-source per lo sviluppo di videogiochi in ambiente multipiattaforma basato sul linguaggio Java. Sono state tuttavia implementate porzioni di codice in linguaggio nativo al fine di supportare l'integrazione di alcune librerie, quali *Firebase*, per il supporto alle notifiche, e *GameAnalytics*, per tracciare le azioni eseguite dall'utente durante la fase di gioco.

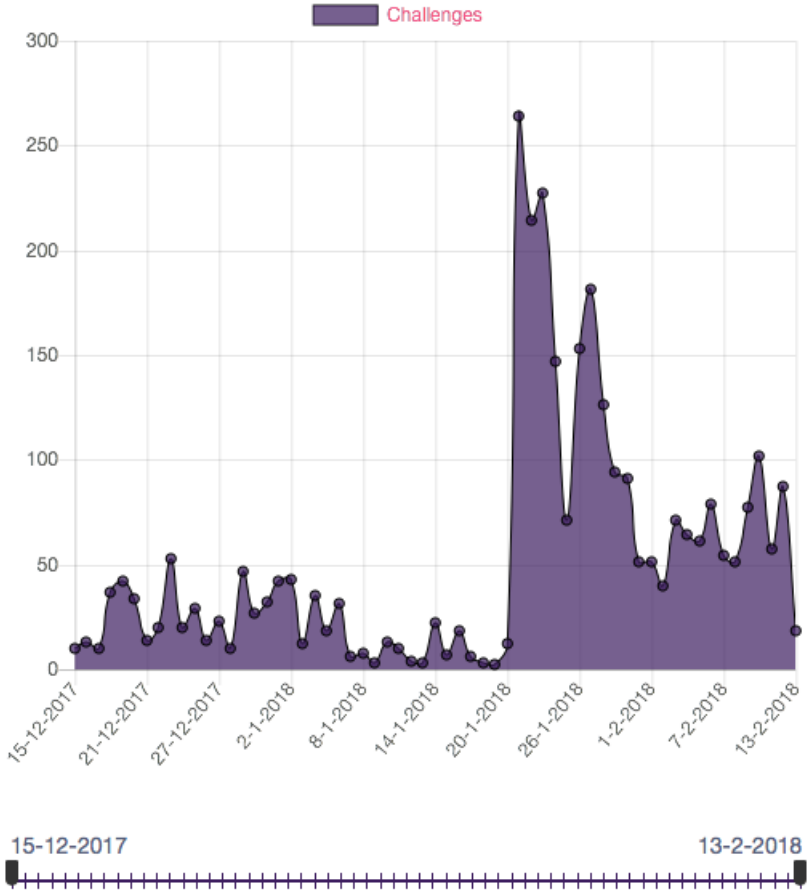


Figura 2.3: Grafico con *ChartJs*

2.4.2 Console di monitoraggio

Tale pannello di amministrazione ed analisi delle statistiche del gioco è stato realizzato come applicazione *full-stack*, che integra un web client sviluppato utilizzando il framework *AngularJS*, che si appoggia su un web server sviluppato in *Spring*. Quest'ultimo punta alla stessa instanza di database utilizzata dal server di gioco così da potere esporre in tempo reale le risorse relative ai dati del gioco. Infatti tale application server è eseguito in un quarto container, rispetto ai tre di cui al paragrafo precedente, sempre all'interno della stessa macchina fisica, ed è anche esso collegato al container in cui viene eseguita l'istanza di MongoDB. In particolare sono state implementate le seguenti funzionalità:

1. Conteggio di utenti, bot presenti, sfide e schemi risolti.
2. Grafico rappresentante l'andamento degli utenti iscritti.
3. Grafico rappresentante il numero di sfide giocate per giorno.
4. Classifica assoluta del gioco.
5. Possibilità di scaricare uno *stylesheet* contenente tutte le informazioni memorizzate sugli utenti.

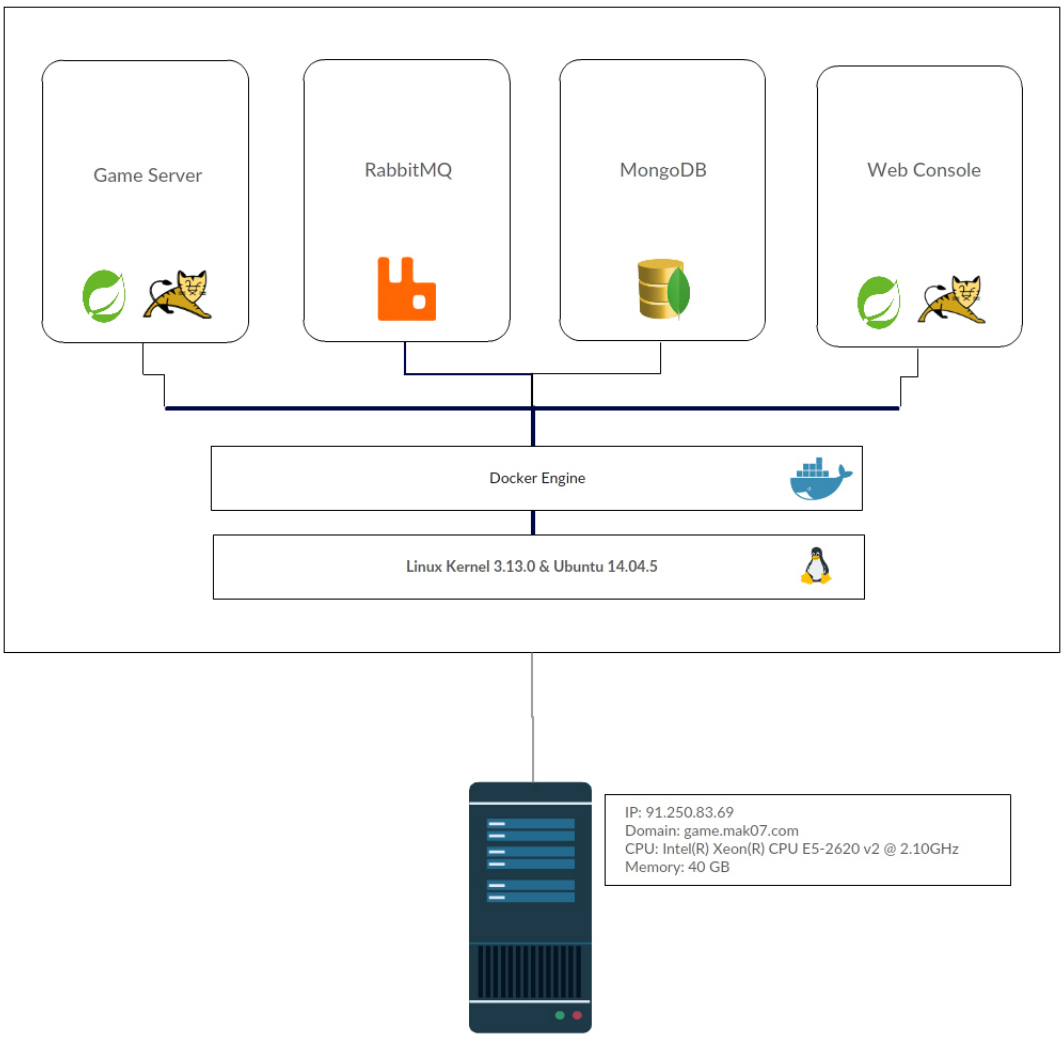


Figura 2.4: Architettura server

2.5 Architettura server

La Figura 2.4 mostra quella che è la struttura interna al server per il quale è stato registrato il dominio *game.mak07.com*. La macchina utilizzata possiede un processore Intel Xeon E5-2620 v2, caratterizzato dall'avere otto core fisici ad una frequenza base di 2.10 GHz. Per quanto riguarda la memoria invece la macchina è dotata di 40GB di RAM. Il sistema operativo installato è Ubuntu 14.04, il quale si basa su un kernel Linux 3.13.0. Al di sopra del sistema operativo viene gestita la virtualizzazione dei vari componenti del sistema a livello software tramite l'utilizzo dei *container*. Grazie a questi insieme alla tecnologia Docker è stato possibile realizzare un'architettura in cui è possibile scalare in orizzontale se necessario. Sono stati realizzati quattro diversi *container*:

1. Il primo *container* è quello contenente l'applicazione Spring Boot ed il server Apache Tomcat eseguito insieme a questa. Questo è il container contenente il server di gioco, il quale espone tutte le API contattabili dal client, che vanno a realizzare la logica dell'applicativo, insieme anche

agli *endpoint* di connessione per le websocket. La versione di Spring Boot utilizzata è la 1.5.7, la quale mantiene al suo interno di default, una versione di Apache Tomcat 8.5.20. Docker si occuperà di mappare tutte le connessioni TCP alla porta 443 verso la 8443, dove è in ascolto Tomcat. La porta TCP 443 è la porta utilizzata di default dal protocollo HTTPS. Inoltre tale *container* è collegato a quelli contenenti MongoDB e RabbitMQ. Al primo in maniera tale da fornire l'accesso alle informazioni mantenute nel database. L'accesso a questo è garantito da opportuni Java *driver* tramite i quali è possibile l'accesso in lettura e scrittura da Spring verso MongoDB. Il collegamento a RabbitMQ invece è necessario per potere inoltrare i messaggi ricevuti tramite Websocket dal client ed inviarli verso una data coda del broker.

2. Il secondo *container* mantiene al suo interno un'istanza del broker di messaggi, RabbitMQ, la cui immagine Docker è stata scaricata direttamente dal repository ufficiale di Docker, ossia Docker Hub. Esso comunica solamente con il server di gioco in maniera tale da accodare i messaggi ricevuti da questo, e viceversa per l'inoltro dei messaggi verso gli opportuni destinatari.

3. Il terzo *container* mantiene il database MongoDB, alla versione 2.6.11, in esecuzione sulla porta di default 27017. Anche tale immagine è stata scaricata da Docker Hub. Tale container è in comunicazione sia col server di gioco, sia con la console di monitoraggio, al fine di garantire l'accesso alle informazioni a questi. Grazie alla natura di MongoDB, ed al supporto di un orchestratore quale Docker, è possibile, quando necessario, scalare orizzontalmente partizionando i dati tra più *container* all'interno della stessa macchina o in più macchine, tramite il meccanismo dello *Sharding* e dei *Replica Set* per aumentare la ridondanza dei dati. Inoltre qualora fosse successivamente necessario una migrazione del database in una macchina distinta, è sufficiente eseguire il *commit* di tale container ed eseguire l'immagine generata all'interno della nuova macchina.

4. Ultimo *container* è quello contenente l'applicazione web realizzata al fine di monitorare le statistiche registrate dal server di gioco. Quest'ultimo è anch'esso connesso al *container* di MongoDB così da potere accedere in tempo reale alle informazioni registrate dal server di gioco, in maniera tale da consultare statistiche sempre aggiornate.

2.6 Considerazioni

All'interno di questo capitolo sono stati introdotti i requisiti tecnologici richiesti dall'applicazione e spiegato come essi abbiano influito all'interno delle diverse scelte progettuali. E' stato inoltre fatto cenno dell'idea alla base del videogioco Mak07, a supporto del quale tale infrastruttura è stata pensata e

realizzata, e brevemente visti anche gli altri componenti del sistema oltre al server di gioco realizzato, anche se maggiore attenzione per gli obiettivi di questo elaborato è stata dedicata a questo. Con questo capitolo si conclude anche la parte I dell'elaborato, che ha affrontato quello che è il contesto dei videogiochi e tutta una serie di analisi sulla struttura architetturale.

Parte II

Le tecnologie utilizzate

Capitolo 3

Spring

Spring è un framework open source per lo sviluppo di applicazioni su piattaforma Java, originariamente creato da Rod Johnson nel 2002. Tale framework fu creato come alternativa alle diverse tecnologie basate su Java *enterprise*. Infatti Spring fu successivamente riconosciuto all'interno della comunità Java quale valida alternativa al modello basato su *Enterprise Javabeans (EJB)*, in quanto il primo offriva un modello di programmazione più leggero e snello se paragonato al secondo. Novità proposta da Spring la quale fu il motivo del suo successo fu l'introduzione della *dependency injection (DI)* e della *aspect-oriented programming (AOP)*. Sulla scia del successo di Spring, successivamente anche in EJB furono introdotti i concetti di DI e AOP. Spring non deve la sua utilità solo alle possibilità di sviluppo lato server, ma qualsiasi applicazione Java può trarre beneficio dalle sue caratteristiche in termini di semplicità, testabilità e disaccoppiamento dei dati.

3.1 Accoppiamento dei dati

Pressoché tutte le applicazioni sono basate su un insieme di classi che interagiscono secondo una qualche logica al fine di raggiungere un obiettivo, ed ogni oggetto è responsabile di ottenere i riferimenti con gli oggetti con cui deve interagire, ossia le sue dipendenze. Questo pattern porta alla realizzazione di applicazioni strettamente accoppiate. In generale, la qualità del codice diminuisce quanto è maggiore tale accoppiamento, in quanto il risultato è un codice poco testabile, riutilizzabile e manutenibile. Spring permette di evitare l'accoppiamento stretto fra oggetti grazie alla DI, che è un particolare tipo di *Inversion of Control (IoC)*. L'Inversion of Control è un pattern di programmazione che realizza l'accoppiamento tra gli oggetti in fase di esecuzione invece che in fase di compilazione. Il controllo del flusso di sistema (Control Flow) è invertito rispetto alla programmazione tradizionale, dove la logica di tale flusso era demandata allo sviluppatore, invece con l'IoC è responsabilità del framework. Come già detto precedentemente, la DI è un particolare tipo di IoC, anche se talvolta questi due termini sono utilizzati come sinonimi, infatti la DI è una specifica implementazione dell'IoC che utilizza un *contextualized lookup*, come afferma anche Martin Fowler. [10]

3.2 Dependency Injection

L'idea alla base della DI è tale da demandare la responsabilità delll'inizializzazione degli oggetti e di fornire loro i riferimenti agli oggetti con cui collaborano ad un componente esterno, un *builder object*. Quindi, tutte le dipendenze vengono prima create da un quest'ultimo e poi successivamente iniettate nel programma principale sotto forma di proprietà degli oggetti. Approccio contrario alla normale creazione di applicazioni. L'approccio tradizionale infatti prevede che il punto di ingresso del programma crei tutte le dipendenze e che successivamente avvenga l'elaborazione. Per garantire un accoppiamento debole, ciascun oggetto dovrebbe conoscere le proprie dipendenze soltanto in termini di interfaccia, così da legare la classe all'interfaccia, piuttosto che all'implementazione. Successivamente, all'atto della creazione, verrà indicato l'oggetto da utilizzare. L'azione di creare associazioni tra gli oggetti prende il nome di *wiring* ed esistono diversi modi con cui questo può essere realizzato, in particolare:

- configurazione esplicita tramite un file XML;
- configurazione esplicita tramite l'utilizzo di classi Java;
- scoperta implicita dei bein e autowiring automatico.

Questi tre stili di configurazione che Spring mette a disposizione possono essere combinati tra loro. Esistono inoltre tre diversi modi con cui le dipendenze possono essere iniettate, ognuno con i propri vantaggi e svantaggi:

- **constructor injection:** le dipendenze sono fornite attraverso il costruttore della classe;
- **setter injection:** le dipendenze sono iniettate utilizzando un apposito metodo setter relativo ad una proprietà della classe;
- **interface injection:** la classe deve implementare un'interfaccia che definisce un metodo setter su ogni proprietà che deve essere iniettata.

Tra queste, la terza è molto invasiva e solitamente meno preferita rispetto alle prime due. L'inizializzazione tramite costruttore fornisce una chiara idea di come è composto l'oggetto stesso ed allo stesso tempo permette di nascondere qualsiasi campo immutabile, non fornendo il setter. Utilizzando invece il secondo metodo si ha la possibilità di istanziare un oggetto ed iniziare ad utilizzarlo iniettando le dipendenze in un secondo momento. Inoltre, l'iniezione basata sui setter consente di snellire i costruttori, utile nel caso in cui ci siano molti parametri.

3.3 Aspect Oriented Programming

La programmazione orientata agli oggetti, o OOP, ha rappresentato il paradigma di sviluppo software predominante degli ultimi anni e grazie alle novità

da essa introdotte è stato possibile realizzare codice modulare, manutenibile, scalabile e facile da riutilizzare. Tuttavia questo modello di sviluppo non si adatta alla modellazione di tutti i problemi che un sistema software deve risolvere , difatti ci sono dei comportamenti che non hanno un ruolo centrale nella business logic di un'applicazione, come la gestione della sicurezza oppure il *logging* delle attività, non facilmente isolabili in moduli distinti. Da qui l'introduzione di quella che prende il nome di programmazione orientata agli aspetti, o AOP. Questa si traduce in un paradigma di programmazione capace di descrivere aspetti e comportamenti trasversali dell'applicazione, in maniera tale da aumentare la modularità delle applicazioni. Ciò viene realizzato tramite l'utilizzo di entità esterne, che prendono il nome di *aspect*, le quali osservano il flusso di esecuzione del programma ed intervengono per modificarlo in determinati istanti, eseguendo operazioni opportune. Incapsulando tali operazioni all'interno degli aspetti le altri classi conterranno solo le loro funzionalità primarie, quindi il risultato è un codice più pulito, ed allo stesso tempo la logica relativa ai comportamenti trasversali non è ripetuta, generando così codice più manutenibile. Tali comportamenti trasversali che riguardano più punti di un'applicazione e vengono eseguiti dagli aspetti prendono il nome di *cross-cutting concern*. Nell'AOP il lavoro compiuto da un aspect è chiamato *advice*, e definisce sia il quando ed il dove di un aspect. Gli aspect di Spring supportano 5 tipi di advice:

- **before:** le funzionalità dell'advice vengono eseguite prima che il metodo sia invocato;
- **after:** le funzionalità dell'advice vengono eseguite dopo il completamento del metodo, indipendentemente dal risultato;
- **after-returning:** le funzionalità dell'advice vengono eseguite dopo il corretto completamento del metodo;
- **after-throwing:** le funzionalità dell'advice vengono eseguite se è sollevata un'eccezione all'interno del metodo;
- **around:** rappresenta un before più after. Alcune funzionalità vengono eseguite prima ed altre dopo l'invocazione del metodo.

3.4 Spring IoC Container

In Spring gli oggetti dell'applicazione risiedono all'interno di un *container*, il quale è responsabile della creazione di questi e di creare le associazioni tra loro, così come gestire il loro ciclo di vita, come mostrato in Figura 3.1. Tale container è il cuore del framework Spring ed utilizza la DI per gestire i componenti dell'applicazione. Tuttavia in Spring non esiste un singolo container, ma esistono diverse implementazioni, che possono essere ricondotte a due distinte tipologie. La prima è quella che prende il nome di *BeanFactory* (definita dall'interfaccia *org.springframework.beans.factory.BeanFactory*) e rappresenta la tipologia di container più semplice, la quale fornisce solo un supporto basilare

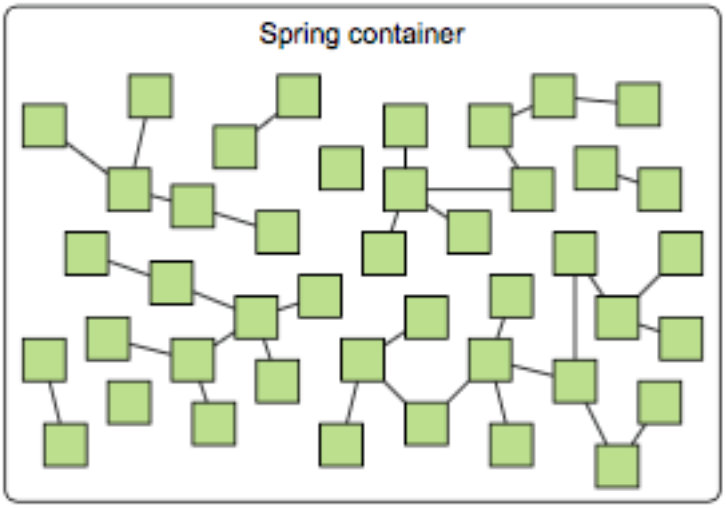


Figura 3.1: Spring container

alla DI. La seconda tipologia è quella che prende il nome di *ApplicationContext* (definita dall'interfaccia *org.springframework.context.ApplicationContext*), la quale è un estensione della prima, in quanto aggiunge funzionalità ulteriori, ed è adatta a scenari di complessità arbitraria. Spring permette l'utilizzo di diversi tipi di application context, i più utilizzati sono:

- FileSystemXmlApplicationContext, carica la definizione del contesto da uno o più file XML all'interno del filesystem.
- ClassPathXmlApplicationContext, carica la definizione del contesto da uno o più file XML presenti all'interno del classpath.
- XmlWebApplicationContext, carica la definizione del contesto da uno o più file XML contenuti all'interno di una web application.

3.4.1 Ciclo di vita dei bean

Il ciclo di vita di un bean in Spring è piuttosto elaborato in quanto esso attraversa diverse fasi tra la sua creazione e distruzione. Ogni step rappresenta una possibilità di modificare la gestione del bean stesso. In particolare, si distinguono le seguenti fasi, come illustrato anche in Figura 3.2.

1. Inizialmente il container carica le definizioni dei bean e quando gli viene chiesto un bean specifico lo instanzia.
2. Il bean viene popolato con le proprietà dichiarate, se una proprietà ha un riferimento ad un altro bean, questo viene creato ed inizializzato prima di creare il bean che lo referenzia.
3. Se il bean implementa l'interfaccia *BeanFactoryAware*, il metodo *setBeanFactory()* è invocato.

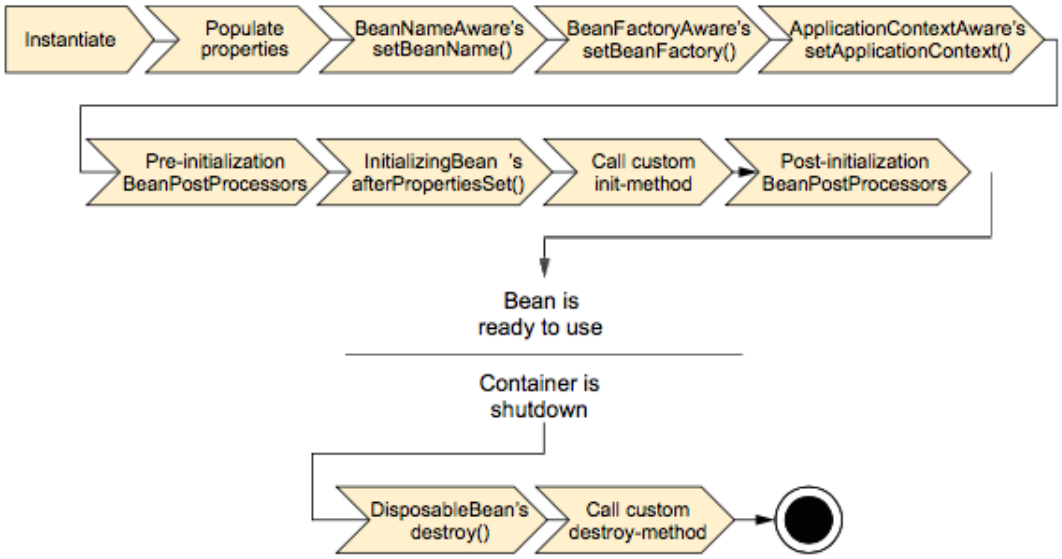


Figura 3.2: Ciclo di vita di un bean

4. Se il bean implementa l'interfaccia *ApplicationContextAware*, il metodo *setApplicationContext()* è invocato.

5. Se il bean implementa l'interfaccia *BeanPostProcessor*, il metodo *postProcessBeforeInitialization()* è invocato.

6. Se il bean implementa l'interfaccia *InitializingBean*, il metodo *afterPropertiesSet()* è invocato.

7. Se il bean implementa l'interfaccia *BeanPostProcessor*, il metodo *postProcessAfterInitialization()* è invocato.

8. A questo punto, il bean è pronto per essere utilizzato e rimarrà all'interno dell'application context fino a quando questo non sarà distrutto.

9. Se il bean implementa l'interfaccia *DisposableBean*, il metodo *destroy()* è invocato. Allo stesso modo, se il bean è stato dichiarato con un *destroy-methodo*, il metodo specificato sarà invocato.

3.4.2 Scope dei bean

Tutti i bean creati all'interno di uno Spring application context sono singleton di default. Questo comporta che indipendentemente dal numero di volte che tale bean è iniettato in altri bean, quella che sarà iniettata sarà sempre la stessa istanza. Il più delle volte tale modello va bene, ma in alcuni casi lavorare con classi mutabili che mantengono un certo stato al loro interno non va bene e non permette un riutilizzo sicuro. Tuttavia Spring definisce diversi tipi di scope per i bean, quali:

- **Singleton**: un'istanza del bean è creata per l'intera applicazione.

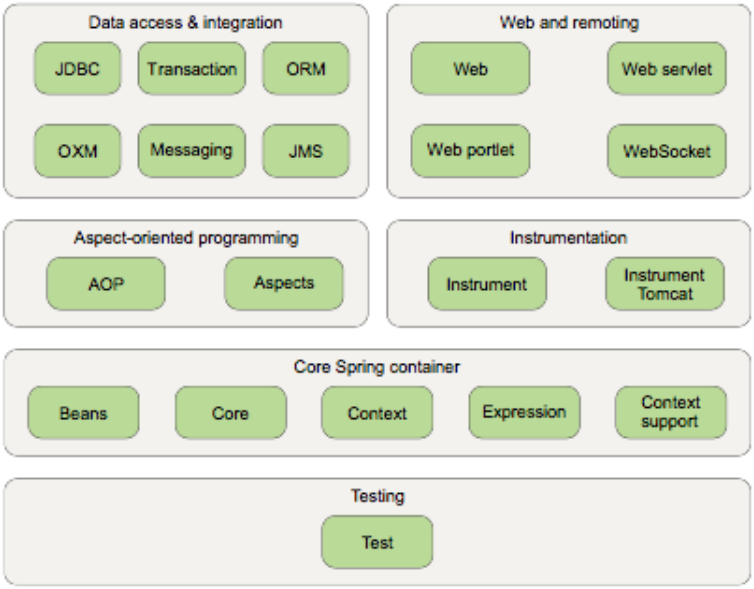


Figura 3.3: I moduli di spring

- **Prototype**: un'istanza del bean è creata ogni volta che il bean è iniettato o recuperato dallo Script application context.
- **Session**: un'istanza del bean è creata per ogni sessione all'interno di una web application.
- **Request**: un'istanza del bean è creata per ogni richiesta HTTP all'interno di una web application.

3.5 I moduli di Spring

All'interno della distribuzione di Spring 4.0 sono presenti 20 moduli distinti con tre file JAR per ogni modulo, relativi al binario, al codice sorgente ed alla documentazione JavaDoc. Questi moduli possono essere a loro volta raggruppati in sei categorie, come mostrato in Figura 3.3:

1. **Core Spring Container**: contiene lo Spring bean factory, che è il componente che permette di realizzare la DI. A partire da quest'ultimo, sono state realizzate diverse implementazioni dell'application context, ognuna delle quali consente di configurare Spring in modo differente. Sono contenuti inoltre tutta una serie di servizi enterprise come email, JNDI access, EJB integration e lo scheduling.
2. **Spring's AOP**: i moduli relativi a questa categoria sono quelli che permettono di realizzare l'Aspect Oriented Programming.
3. **Data access and integration**: questo modulo permette di realizzare una astrazione maggiore in maniera tale da semplificare l'accesso ai vari tipi di database, così da potere scrivere codice più pulito ed in quantità inferiore rispetto ad altri framework.

4. **Web e remoting**: consente di realizzare applicazioni web secondo il pattern *Model View Controller (MVC)*.

5. **Instrumentation**: include il supporto per aggiunta di agenti alla Java Virtual Machine.

6. **Testing**: fornisce una serie di implementazioni di mock object che consentono la scrittura di unit test funzionali a testare le singole componenti di un'applicazione.

3.6 Spring portfolio

Accanto a quelli che sono i moduli base che compongono il framework Spring, esistono tutta una serie di librerie basate su questo che permettono di estenderne le funzionalità. Di seguito sono descritte alcune tra le più importanti e maggiormente utilizzate nella realizzazione di questo elaborato.

Spring Boot

Spring di per sè semplifica molti dei compiti del programmatore, riducendo o addirittura eliminando una grande quantità di codice ridondato. Spring Boot è nato con l'obiettivo di semplificare ulteriormente Spring stesso. Questo impiega configurazione automatiche che eliminano quasi del tutto le configurazioni necessarie in Spring. Infatti Spring Boot permette di creare applicazioni basate su Spring già pronte per essere avviate.

Spring Data

Spring Data permette di lavorare facilmente con ogni tipo di database in Spring, offrendo un modello di programmazione semplificato per la persistenza dei dati, indipendentemente se questo sia un database a documenti come MongoDB, o a grafi come Neo4j, o un tradizionale database relazionale. Spring Data introduce un meccanismo di gestione automatica dei *repository* creando opportune implementazioni di queste al posto del programmatore.

Spring Social

Ad oggi sono sempre più le applicazioni che richiedono l'integrazione con i social network più diffusi, come Facebook e Twitter. Spring Social permette di ottenere facilmente ciò attraverso l'utilizzo di API ReST.

Spring WebMVC

Spring WebMVC è il framework di Spring per lo sviluppo di applicazioni web. Esso si basa sul pattern MVC e consente la realizzazione di applicazioni web flessibili in cui l'accoppiamento stretto tra oggetti è ridotto, così come avviene

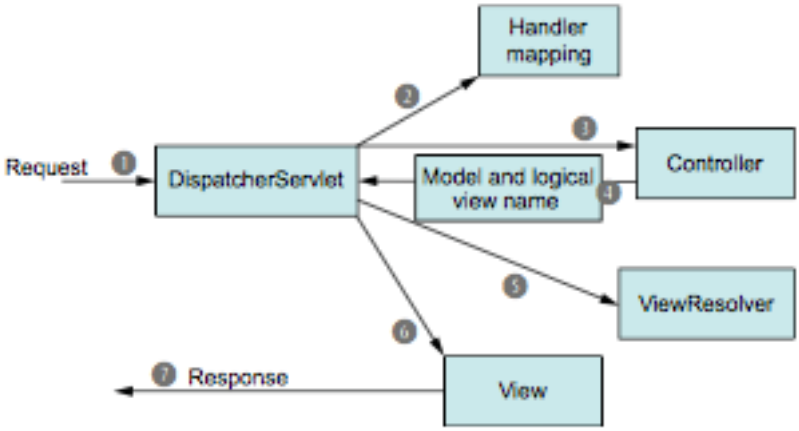


Figura 3.4: Dispatcher Servlet in Spring

all'interno di Spring stesso. Ogni qualvolta un utente clicca un link o esegue l'invio di un form all'interno del browser, l'effetto è quello di generare una richiesta. Quando questa lascia il browser, porta con se tutta una serie di informazioni, richieste dall'utente. Come minimo, questa porterà con se l'URL richiesta ma anche eventuali informazioni aggiuntive, come quelle presenti nel form popolato dall'utente. La richiesta viaggerà sino al *DispatcherServlet*, che rappresenta il *front controller* all'interno di Spring MVC. Il front controller ha il compito, una volta ricevuta una richiesta, di inoltrarla verso l'opportuno componente dell'applicazione che la processerà. In questo caso il DispatcherServlet inoltrerà la richiesta verso un *controller*, che è il componente all'interno di Spring che processerà la richiesta. Tuttavia, all'interno di una applicazione Spring possono coesistere più controller, ragion per cui è necessario un ulteriore supporto al DispatcherServlet per inoltrare correttamente la richiesta, il quale è fornito da uno o più *HandlerMapping*. Dopo aver consultato l'HandlerMapping il DispatcherServlet inoltrerà la richiesta verso il controller, che analizzerà ed elaborerà le informazioni ricevute. Un controller ben progettato, generalmente si serve di altri componenti, definiti *Service*, per eseguire la business logic opportuna. Il risultato delle operazioni eseguite da un controller spesso si traduce in una serie di informazioni che devono essere restituite all'utente e visualizzate nel browser. Queste informazioni prendono tipicamente il nome di *model*. Non è tuttavia consigliabile inviare queste all'utente in un formato poco *user-friendly*, ma è preferibile utilizzare una formattazione opportuna, come HTML. Di conseguenza il controller deve anche comunicare al DispatchServlet il nome logico della *view* da visualizzare. Il DispatcherServlet consulterà il *ViewResolver* per ottenere la view associata a quel nome logico. Se non viene configurato un ViewResolver specifico, Spring di default utilizza un *BeanNameViewResolver* che effettua la risoluzione della view cercando un bean il cui identificativo sia il nome logico della vista. Successivamente, i dati precedentemente preparati dal controller sono passati alla view da visualizzare e la rappresentazione grafica costruita. Quest'ultima è restituita al DispatcherServlet che la incapsulerà all'interno della

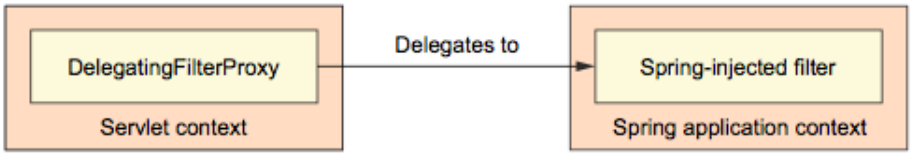


Figura 3.5: DelegatingFilterProxy

risposta da restituire all'utente. Il flusso appena descritto è osservabile in Figura 3.4. La più semplice configurazione di Spring MVC può essere ottenuta annotando una classe di configurazione con l'annotazione *@EnableWebMvc*.

Spring Security

Spring Security è il framework che permette di gestire la sicurezza dell'applicazione in maniera semplice, flessibile e potente. Questa è gestita sia a livello di richiesta HTTP sia a livello di invocazione di metodo, sfruttando i meccanismi di DI e AOP. Spring Security ha avuto il suo inizio come Acegi Security, un potente framework di sicurezza, ma che aveva il difetto di necessitare di una grande quantità di configurazione XML. Successivamente, Acegi Security con la versione 2.0 divenne Spring Security, portando ad una significativa riduzione della configurazione necessaria, per poi diminuire ancora con Spring Security 3.0. Per gestire la sicurezza sia a livello di richiesta web e per limitare l'accesso a livello di URL, Spring Security si appoggia a dei *servlet filters*. Mentre per garantire la sicurezza a livello di metodo si appoggia al meccanismo dell'AOP, controllando il flusso di esecuzione ed invocando opportuni aspetti al fine di assicurare che l'utente abbia l'autorizzazione necessaria per invocare un certo metodo. Al fine di utilizzare i servlet filters di Spring Security è sufficiente configurare solo uno di questi, ossia il *DelegatingFilterProxy*, il quale è un servlet filter speciale che di per sè non fa molto ma delega ad una implementazione dell'interfaccia *javax.servlet.Filter*, la quale è registrata come bean all'interno dell'application context di Spring, come illustrato in Figura 3.5. Relativamente a quella che è l'autenticazione dell'utente Spring Security offre la possibilità di configurare diversi meccanismi di autenticazione, tramite *in-memory* database, database relazionali, *LDAP directory servers*, ed in qualsiasi altro caso si presta ad essere modificato così da implementare configurazioni personalizzate.

3.7 Considerazioni

Le caratteristiche di tale framework, in particolare la presenza dell'IoC container e dell'utilizzo di DI e AOP, così da avere codice debolmente accoppiato, ergo scalabile e facilmente testabile, motivano la scelta del suo utilizzo. Questo è inoltre motivato dall'utilizzo: di Spring WebMVC, che ha permesso lo sviluppo di una applicazione web basata su architettura ReST; di Spring So-

cial che ha permesso una facile integrazione con Facebook e Google; di Spring Security che ha avuto particolare rilevanza per la sua impronta in termini di sicurezza, permettendo di gestire l'accesso ristretto alle risorse e l'autenticazione e autorizzazione degli utenti; di Spring Data che ha permesso di lavorare facilmente con il database MongoDB.

Capitolo 4

MongoDB

Nel seguente capitolo è analizzato il sistema di backing store adottato. E' inizialmente presentata una breve panoramica delle tecnologie esistenti, esponendo vantaggi e svantaggi di ognuna, così da esporre successivamente le ragioni che hanno portato alla scelta di utilizzo di un sistema non relazionale quale MongoDB insieme ad una descrizione di quest'ultimo.

4.1 I database relazionali

Esistono vari modelli di memorizzazione dei dati, tra i quali il modello relazionale, il quale è un modello logico di rappresentazione dei dati di un database implementato su sistemi di gestione di basi di dati (DBMS), detti perciò sistemi di gestione di basi di dati relazionali (RDBMS). Tale modello fu presentanto per la prima volta nel 1970 con l'articolo "A Relational Model of Data for Large Shared Data Banks" pubblicato da Edgar F. Codd. "Ancora oggi i RDBMS dominano il settore dei sistemi informativi, basti pensare all'enorme diffusione di DBMS come Oracle, MySQL e Microsoft SQL Server". [5]. Indipendentemente dalle varie differenze implementative ogni RDBMS è basato sul concetto fondamentale di *transazione*: un'unità logica di lavoro eseguita da un'applicazione, ovvero una sequenza di istruzioni che realizzano operazioni di scrittura o lettura su un database. E' stato standardizzato un linguaggio per eseguire tali istruzioni che prende il nome di *Structured Query Language (SQL)*. In un database relazionale l'articolazione dei dati consente di costruire modelli di raccolta dati complessi. L'elemento di partenza di ogni database relazionale è la *tabella*, che altro non è che un insieme di *righe* e *colonne*. Ogni colonna contiene un dato relativo all'entità memorizzata, mentre ogni riga rappresenta l'entità stessa. In termini di database ogni colonna è un *campo*, ogni riga un *record*. Un database può essere composto da più tabelle. Ciò che caratterizza fortemente un database relazionale è la presenza di legami fra le tabelle, di connessioni logiche, ossia di *relazioni*. Infatti l'assunto fondamentale del modello relazionale è che tutti i dati siano rappresentati come relazioni e manipolati con gli operatori dell'algegra relazionale o del calcolo relazionale, da cui appunto il nome. E' infine importante sottolineare che all'interno di un

DBMS le transazioni si basano su meccanismi che devono soddisfare sempre le seguenti proprietà:

- **Atomicity:** una transazione non può ammettere stati intermedi, o in altri termini la transazione è indivisibile nella sua esecuzione.
- **Consistency:** non devono essere violati i vincoli di integrità del database, ossia se il database si trova in uno stato consistente prima della transazione, dovrà trovarsi in un altro stato consistente anche dopo questa.
- **Isolation:** l'esecuzione di una transazione è independente dalle altre.
- **Durability:** una volta che una transazione abbia completato il suo lavoro, l'effetto di questo non viene più perso.

le quali prendono il nome di **ACID**.

4.1.1 Limiti dei database relazionali

I database relazionali hanno rappresentato lo standard per oltre quarant'anni, ma l'evoluzione tecnologica e l'aumentare della mole dei dati a cui le applicazioni sono soggette ha fatto emergere i limiti di tale modello. Basti pensare ai dati generati dai vari social network ed utenti connessi in rete, ma anche a quelli derivanti dai sensori propri dell'*Internet of Things*. Limiti, in particolare, in termini di *scalabilità*. Con questo termine si intende la capacità di un sistema di crescere o diminuire di scala in funzione delle necessità e delle disponibilità. Nel mondo dei database relazionali ciò può essere raggiunto solo tramite *scaling verticale*, ossia aumentare il numero di risorse a disposizione incrementando le risorse hardware della macchina, quindi un maggiore numero di processori, una maggiore quantità di memoria, hard disk e così via. Tuttavia questa soluzione rischia di essere molto costosa e dalle possibilità inevitabilmente limitate, rappresentando comunque un *single point of failure* per l'intero sistema. Soluzione migliore è quella dello *scaling orizzontale*, tale per cui la scalabilità non è ottenuta aumentando le prestazione ma bensì tramite diverse repliche divise su un cluster di macchine, senza che vi sia nessuna porzione di memoria condivisa fra i server. Uno dei maggiori limiti del modello relazionale è proprio la difficoltà di realizzazione della scalabilità orizzontale, che risulta essere sempre più importante nella gestione ed elaborazione dei dati. La rigidità del modello relazionale non si adatta bene alla dinamicità e flessibilità di cui si ha bisogno, infatti vi è la necessità di definire uno schema prima dell'inserimento dei dati. Alcuni casi d'uso impongono il bisogno di apportare modifiche allo schema (aggiungere o rimodellare i dati), questo processo richiede una significativa quantità di tempo durante il quale il database non può essere utilizzato. Inoltre, il più grande problema per gli sviluppatori invece è che le operazioni SQL non sono adatte per le strutture dati *object oriented* utilizzate nelle maggiore parte delle applicazioni odierne. Tutto ciò ha portato alla necessità di cercare nuove soluzioni, che andasserò aldilà del modello relazionale e dei suoi limiti.

4.2 I database non relazionali

L'introduzione dei database non relazionali, indicati con la sigla NoSQL, proprio perchè non seguono il modello SQL e non utilizzano l'omonimo linguaggio, ha portato una soluzione ai limiti indicati nel paragrafo precedente. Questi hanno registrato una crescita esponenziale nel loro sviluppo e successo grazie al sempre più crescente bisogno di scalare in orizzontale, obiettivo raggiunto proprio grazie alla loro semplicità. Infatti consentono di aggiungere nodi a caldo in maniera impercettibile dall'utente finale. Inoltre molti database NoSQL sono performanti per operazioni di lettura o scrittura sull intero data-set, tramite meccanismi di *lockless-design* che permettono di leggere e scrivere a velocità costante indipendentemente dal numero di utenti che stanno effettuando richieste sui dati. Essi non necessitano più di uno schema rigido, diversamente dal modello relazione, nasce quindi la possibilità di rimodellare la struttura in corso d'opera, senza avere alcun *downtime*. Tuttavia la semplicità che li caratterizza porta con sè anche delle mancanze. In particolare viene meno il vincolo sull'integrità dei dati, compito che ricade totalmente sull'applicativo che dialoga col database, che ovviamente dovrebbe essere testato in modo molto approfondito prima di essere messo in produzione. Inoltre la mancanza di uno standard universale come SQL si traduce nello sviluppo delle proprie APIs, per leggere o scrivere i dati, da parte di ogni database non relazionale, non vi è quindi una conformità, ma almeno vi è il lato positivo di avere abbandonato la complessità del linguaggio SQL. Prima di continuare l'analisi del mondo NoSQL si introduce il seguente teorema.

Il teorema CAP
Il teorema CAP enunciato dal professore Eric Brewer dell'università di Berkeley al *Principle of Distributed Computing (PODC)* descrive il compromesso coinvolto nel mondo dei sistemi distribuiti. Esso afferma che per un sistema informatico distribuito è impossibile garantire simultaneamente tutte e tre le seguenti proprietà:

- **Consistency:** capacità di avere una singola copia aggiornata dei dati, su tutti i nodi replica.
- **Availability:** capacità di un sistema di fornire sempre una risposta sia essa negativa o positiva ad una interrogazione.
- **Partition tolerance:** capacità di un sistema distribuito di continuare a funzionare anche in presenza di errori di rete, ergo arbitrarie perdite di messaggi.

Dopo aver enunciato il teorema CAP il professor Brewer propose le proprietà **BASE** in contrapposizione alle già esistenti **ACID**:

- **Basic Available:** indica la capacità di un sistema di garantire l'availability nei termini del teorema CAP, quindi ci sarà una risposta a qualsiasi richiesta.

- **Soft State:** indica che lo stato del sistema può cambiare nel tempo anche senza input, questo a causa del modello eventually. La consistenza dei dati diventa un problema da risolvere da parte dello sviluppatore e non deve essere gestita dal database. [1]

- **Eventually Consistent:** il sistema diventerà consistente secondo il teorema CAP, cioè tutti i nodi replica avranno la stessa copia dei dati, in un certo intervallo di tempo durante il quale il sistema non riceverà input dall'esterno.

Molti database NoSQL seguono le proprietà appena citate fornendo così la possibilità, a chi li utilizza, di avere un'architettura capace di scalare orizzontalmente.

4.2.1 Classificazione dei database non relazionali

Si distinguono le seguenti tipologie di database NoSQL:

- **Document Store:** consentono di manipolare dati semi-strutturati, ad ogni chiave è associato un documento che può contenere più coppie chiave-valore o anche documenti innestati (MongoDB, CouchDB).

- **Key-Value Store:** ogni elemento è memorizzato come una coppia chiave-valore (Riak, Voldemort).

- **Column-Oriented:** i dati sono memorizzati come colonne. Sono pensati per ottimizzare il calcolo degli aggregati (Cassandra, HBase).

- **Graph Store:** le relazioni tra i dati sono rappresentate attraverso grafi (Neo4j, HyperGraphDB).

4.3 Caratteristiche principali

MongoDB è un DBMS open source di tipo *document store*, che come gli altri database NoSQL propone un modello senza uno schema fisso. Esso è stato progettato per essere più efficiente sia dal punto di vista dello spazio di memorizzazione dei dati, sia dal punto di vista della velocità di ricerca. L'accesso ai dati non è offerto su API ReST basate su HTTP, ma tramite una soluzione proprietaria che realizza il CRUD su comunicazioni TCP/IP. L'architettura proposta da MongoDB è principalmente composta dai seguenti componenti:

- **database:** contenitore che raccoglie strutture dati chiamate collezioni. Tipicamente un'istanza di MongoDB server contiene più database;

- **collezioni:** rappresentano un insieme di documenti correlati tra loro, equivalenti alle tabelle dei database relazionali, sebbene esse non prevedano uno schema fisso;

- **documenti:** rappresentano l'unità elementare memorizzata in MongoDB. Questa è memorizzata in formato BSON (*Bynary* JSON), che è un'estensione del formato JSON (*Javascript Object Notation*), per la memorizzazione di oggetti binari. A differenza dei database relazionali i documenti di una stessa collezione possono avere campi diversi tra loro, oltre che gestire dati di natura diversa. Va inoltre detto che nonostante MongoDB, in quanto database NoSQL, non garantisce le proprietà ACID, esso comunque offre l'atomicità delle operazioni a livello di singolo documento.

4.4 Modello dei dati

Come giò detto i dati in MongoDB seguono uno schema flessibile che facilita il mapping tra documenti memorizzati nel database e oggetti utilizzati nell'applicazione, così da facilitare il lavoro dello sviluppatore. Per quanto riguarda invece le relazioni tra questi dati è possibile distinguere due modalità:

- **Riferimenti:** le relazioni tra i dati vengono rappresentate utilizzando collegamenti, appunto riferimenti, tra i vari oggetti del database. Tecnica che più si avvicina all'approccio dei database relazionali, in quanto i dati in questo modo vengono normalizzati.

- **Documenti innestati:** le relazioni tra i dati vengono rappresentate memorizzando i dati correlati in un unico documento. In questo caso i dati sono denormalizzati, sfruttando le proprietà intrinseche dei database NoSQL.

Il meccanismo dei riferimenti viene utilizzato per rappresentare relazioni molti a molti o grandi insiemi di dati organizzati gerarchicamente. Questo fornisce più flessibilità rispetto ai documenti innestati anche se lo svantaggio è che necessita di più operazioni per risolvere i riferimenti. Il secondo modello invece viene utilizzato solitamente per rappresentare relazioni uno a uno o uno a molti, dove il documento innestato rappresenterà la parte N della relazione. In questo caso sarà sufficiente una singola operazione per la lettura o aggiornamento di dati correlati.

4.5 Disponibilità dei dati e scalabilità

MongoDB fornisce diversi meccanismi al fine di aumentare la disponibilità temporale e spaziale del sistema.

4.5.1 Replica Set

Si definisce *replica set* un insieme di istanze di MongoDB, rappresentate dal processo *mongod*, che mantengono lo stesso insieme di dati. Tale soluzione

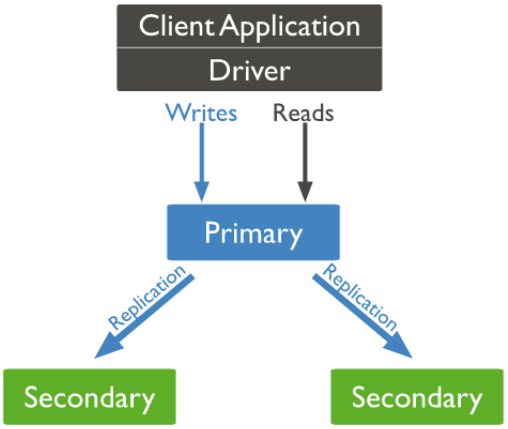


Figura 4.1: Replica Set

aumenta la ridondanza e la disponibilità dei dati, ed è la base per ogni sistema in produzione. Tra le diverse istanze che compongono il replica set, si distinguono nodi primari e nodi secondari, così come mostrato in Figura 4.1. Di default il nodo primario è configurato per rispondere a tutte le operazioni di lettura e scrittura da parte del client, mentre i nodi secondari sono trasparenti a quest'ultimo. Il nodo primario per ogni operazione ricevuta, registrerà le modifiche da apportare all'insieme di dati aggiornando un opportuno file di log. Tale file di log sarà replicato nei nodi secondari che applicheranno le stesse modifiche all'insieme di dati, così che tutti i nodi mantengano le informazioni allineate. Inoltre il sistema è tale che in caso di guasto del nodo primario, un nodo secondario venga eletto a primario. Ciò viene realizzato tramite un meccanismo di *heartbeat* ed eventualmente di un nodo arbitro, che è un server dove gira un istanza di MongoDB ove non è presente alcun dato. La replica delle operazioni tra nodo primario e secondari è effettuata di default in maniera asincrona, anche se è possibile forzare il sistemare ad effettuare copie sincrone, anche se questo comporterà un notevole degradamento delle prestazioni. In generale il meccanismo di copia asincrona non rappresenta un particolare problema, ma nel caso in cui i nodi siano configurati per permettere operazioni di lettura al client, quello che può succedere è che le informazioni non siano ancora allineate tra le varie repliche, e che vengano restituite al client informazioni non aggiornate. Ossia in tal caso viene meno la proprietà di consistenza forte.

4.5.2 Sharding

Lo *sharding* è il meccanismo utilizzato da MongoDB per gestire il *deploy* di sistemi che gestiscono un quantità di dati molto elevata sui quali vengono eseguite operazioni di calcolo intensive, che hanno un costo sia in termini di CPU che di I/O su disco. Tale tecnica è un esempio di come MongoDB, in quanto database non relazionale, permette di realizzare la scalabilità orizzontale. Infatti esso permette di partizionare i dati tra diverse macchine all'interno di un cluster di server, così da permettere alle proprie applicazioni di scalare

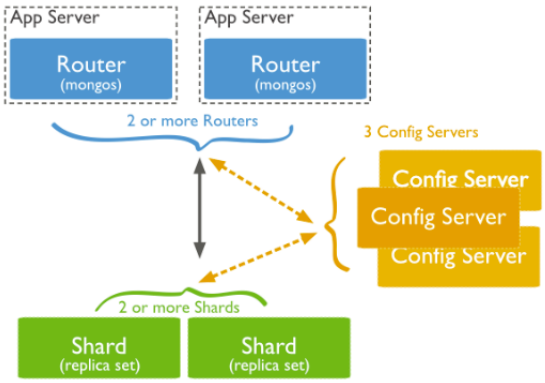


Figura 4.2: Sharded cluster

quando richiesto. I server appartenenti a tale cluster, dove sono mantenute le partizioni dei dati, prendono il nome di *shard*. Tale meccanismo è composto dai seguenti componenti, come mostrato in Figura 4.2:

- **shard**: le macchine fisiche che memorizzano i dati e formano il data set. Ogni shard può essere a sua volta un replica set;
- **query routers**: rappresentano l'interfaccia tra il client e lo *sharded cluster*. Un cluster ben progettato consiste di più router al fine di dividere il carico delle richieste provenienti dai client;
- **config server**: memorizzano i metadati necessari affinchè il cluster funzioni. Questi dati vengono consultati dai query router e consentono a questi ultimi di indirizzare le richieste allo shard corretto.

MongoDB effettua lo sharding dei dati a livello di collezione, dividendo queste tra i vari shard del cluster. Per dividere i documenti di una collezione, MongoDB partiziona le collezioni tramite una *shard key*, che è un campo indicizzato all'interno del documento. I valori di quest'indice vengono divisi in *chunk* che verranno distribuiti tra gli shard del cluster.

4.6 Considerazioni

Per via della sua natura non relazionale e delle sue funzionalità, MongoDB ha rappresentato la scelta migliore come sistema di backing store a supporto dell'infrastruttura di backend realizzata in questo elaborato. In quanto non avendo alcuno schema fisso è stato possibile modificare la struttura dei dati più volte in corso d'opera in maniera tale da adattarla alle varie esigenze. Sono state inoltrate utilizzate tecniche di modellazione dei dati, sia per riferimento sia tramite documenti innestati. Grazie alla sua semplicità esso ha consentito l'esecuzione del server applicativo in produzione inizialmente con una sola istanza ma con la possibilità di scalare orizzontalmente in seguito.

Capitolo 5

Docker

All'interno di questo capitolo sarà inizialmente spiegato cosa si intende per virtualizzazione software. Successivamente dopo avere effettuato un confronto su alcuni dei meccanismi ad oggi esistenti, verrà analizzata la tecnologia Docker e spiegati i motivi che hanno portato a sceglierlo come tecnologia di virtualizzazione in ambiente di produzione.

5.1 Il meccanismo della virtualizzazione

La virtualizzazione ha inizio negli anni sessanta con i mainframe, come metodo per dividere logicamente le risorse di sistema fornite dal mainframe tra le diverse applicazioni. Il sistema IBM 360 fu il primo sistema operativo che offriva a più utenti una copia personale del sistema operativo in esecuzione su un unico mainframe. Molte furono le aziende che successivamente si affacciarono al mondo della virtualizzazione, ma fu VMWare la prima azienda nel 1999 a lanciare sul mercato la prima tecnologia di virtualizzazione *VMware virtual platform* per architetture x86. Seguirono poi altre aziende quali IBM, Microsoft, RedHat e HP. Attraverso la virtualizzazione è possibile installare sistemi operativi su hardware virtuale. L'insieme delle componenti hardware virtuali prende il nome di macchina virtuale. Tale tecnica è applicabile sia su sistemi desktop che su sistemi server. Il vantaggio è che più macchine virtuali possono girare contemporaneamente sullo stesso sistema fisico condividendone le risorse. Le macchine virtuali presentano le seguenti caratteristiche, che offrono numerosi vantaggi: [13]

- **partizionamento** esecuzione di più sistemi operativi su una macchina fisica e suddivisione delle risorse di sistema tra le macchine virtuali;
- **isolamento:** isolamento di guasti e problemi di sicurezza a livello di hardware e protezione delle prestazioni grazie a controlli avanzati delle risorse;
- **incapsulamento:** salvataggio su file dell'intero stato di una macchina virtuale. Spostamento e copia delle macchine virtuali con estrema facilità, in modo analogo ai file;

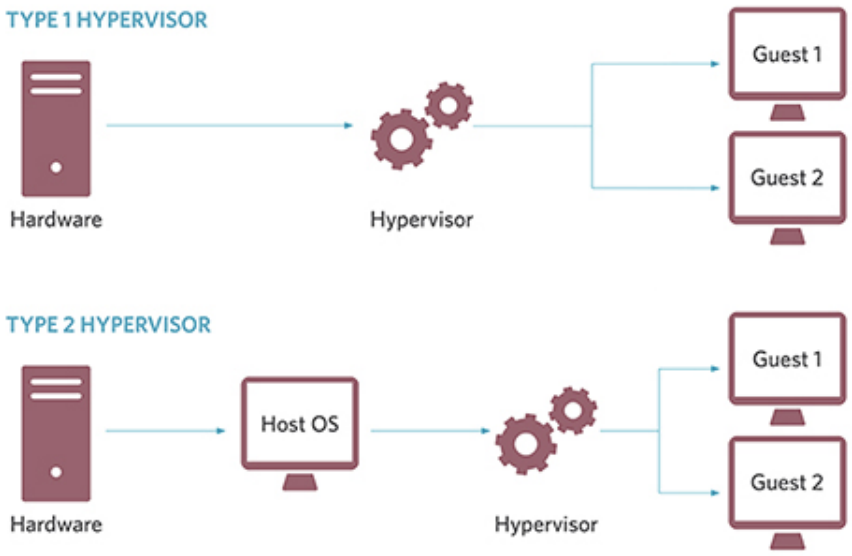


Figura 5.1: Tipi di hypervisor

- **indipendenza dell'hardware:** provisioning o migrazione delle macchine virtuali a qualsiasi server fisico.

5.1.1 Hypervisor e Container

Si distinguono due principali tecniche di virtualizzazione, quella basata su *hypervisor* e quella su *container*.
L'hypervisor, conosciuto anche come *virtual machine monitor*, è il componente principale di un sistema basato sulle macchine virtuali. Ha rappresentato per anni lo standard del mondo enterprise per la virtualizzazione di server applicativi. Un hypervisor consente di riprodurre interi stack, dall'hardware alle applicazioni, passando per il sistema operativo ed il suo kernel, garantendo ad ogni macchina virtuale un isolamento completo del resto del sistema, svolgendo anche attività di controllo sopra ogni sistema. La macchine virtuali sono eseguite su quello che prende il nome di sistema *host*, mentre il sistema operativo virtuale invece prende il nome di sistema *guest*, in quanto è ospitato dall'host. Un hypervisor è solitamente implementato come uno strato software al di sopra del sistema operativo, questo è il caso di tecnologie come *VMware vSphere* o *Microsoft Hyper-V*. Tuttavia esiste anche la possibilità che questo venga implementato direttamente all'interno del firmware di sistema. Da ciò segue la seguente classificazione, come mostrato in Figura 5.1: [12]

- **Type 1 Hypervisor**: eseguito direttamente sull'hardware di sistema senza alcun sistema operativo o software sottostante.

- **Type 2 Hypervisor**: eseguito al di sopra del sistema operativo. Tale tipo di hypervisor è solitamente chiamato *hosted*.

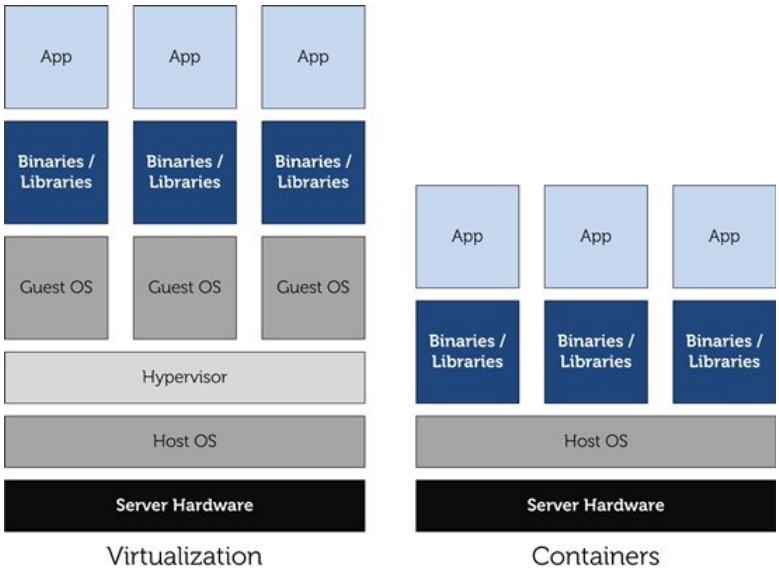


Figura 5.2: Hypervisor vs Container

Altra tecnologia è quella basata sui container. Se la tecnologia basata su hypervisor si basa sul concetto di virtualizzazione hardware, quella basata sui container si basa sul concetto di virtualizzazione software, realizzaziondo la virtualizzazione a livello di sistema operativo invece che a livello hardware. Tutto ciò garantendo lo stesso livello di isolamento dei processi e delle risorse degli hypervisor, ma allo stesso tempo richiedendo molto meno risorse di una macchina virtuale non dovendo emulare l'hardware ed eseguire un intero sistema operativo. *Linux Containers (LXC)* è un ambiente di virtualizzazione a container che permette di eseguire diversi ambienti Linux virtuali isolati tra loro, convidendo lo stesso kernel. Il supporto fornito dal kernel Linux ai LXC è quello dei *namespace* e dei *cgroups*. Un *namespace* è costituito da un insieme di processi che accedono a risorse kernel virtualizzate. Ogni container costituisce, per il sistema operativo, un gruppo di processi a se stanti. Tramite il meccanismo dei *cgroups* invece è possibile gestire limiti e priorità di utilizzo delle risorse, in termini di CPU, accesso alla memoria, accesso ai dischi ed alla rete, senza dovere utilizzare una vera e propria macchina virtuale.
La Figura 5.2 riassume le differenze tra un infrastruttura di virtualizzazione basata su hypervisor ed una basata su container.

5.2 Caratteristiche principali

Docker è una tecnologia *open source* per la costruzione, lo spostamento ed il rilascio di applicazioni basate su container. Esso è disponibile per tutte le maggiori piattaforme per eseguire container in ambienti Linux o Windows, fornendo un'implementazione che standardizza l'utilizzo dei container su tali piattaforme. Secondo la ditta di analisi industriale 451 Research, "Docker è uno strumento che può pacchettizzare un'applicazione e le sue dipendenze in un container virtuale che può essere eseguito su qualsiasi server Linux" [8].

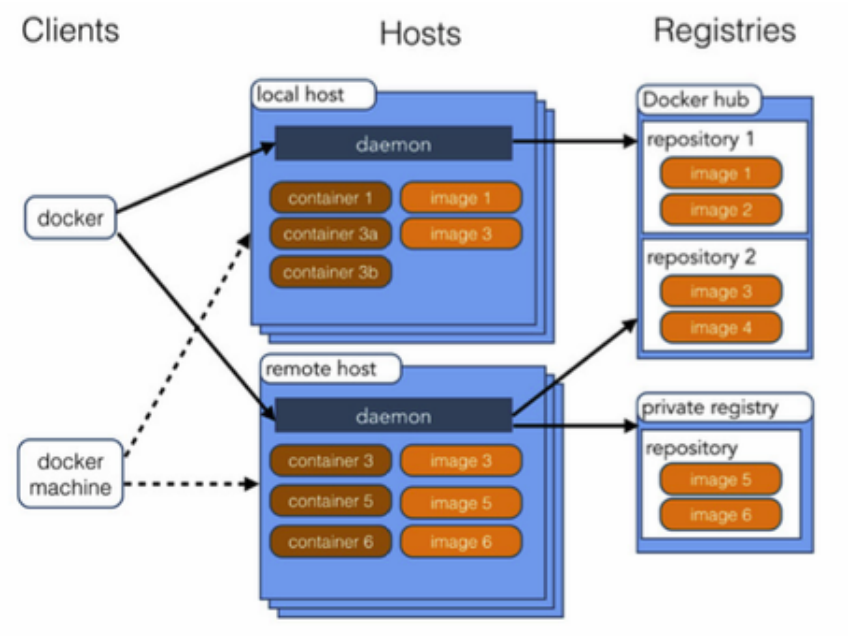


Figura 5.3: Architettura di Docker

Docker accede alle funzionalità di virtualizzazione del kernel Linux o direttamente utilizzando la libreria *libcontainer*, che è disponibile da Docker 0.9, o indirettamente attraverso *libvirt*, LXC o *systemd-nspawn*. In Figura 5.3 è mostrata l'architettura di Docker.

5.3 Componenti principali

I quattro componenti principali all'interno del sistema Docker sono:

- **Docker Engine**: è il core della piattaforma, processo demone, eseguito in background sulla macchina che deve ospitare i container. Fornisce l'accesso a tutte le funzionalità ed i servizi messi a disposizione da Docker. Esso standardizza la gestione dei container, infatti, grazie ad esso, è possibile sviluppare un'applicazione su una piattaforma per poi migrarla su un'altra. Ad oggi viene fornito in due diverse versioni, *Enterprise Edition* e *Community Edition*.

- **Docker Client**: interfaccia per le API esposte dal Docker Engine. Mette a disposizione una serie di comandi per accedere alle funzionalità fornite.

- **Docker images**: un'immagine è un insieme di file e parametri che definisce e configura un'applicazione da usare a runtime. Essa non ha uno stato ed è immutabile.

- **Docker Container**: un container è un'instanza in esecuzione di un'immagine il cui file system è costituito da uno strato R/W sovraposto agli strati immutabili dell'immagine.

5.4 Considerazioni

Tramite la virtualizzazione dei server è possibile ottimizzare l'utilizzo delle risorse server e ridurre il numero di server richiesti. Il risultato è il consolidamento dei server, che migliora l'efficienza e riduce i costi. Inoltre, la possibilità di ridurre ulteriormente le risorse necessarie tramite l'utilizzo di tecnologie basate sui *container*, come Docker, motiva la scelta del suo utilizzo.

Capitolo 6

RabbitMQ

6.1 Broker di messaggi

Nell'ambito dei sistemi distribuiti la logica con cui si collegano produttori a consumatori può essere molto complessa e spesso richiede l'uso di un apposito componente: il broker dei messaggi ed i relativi protocolli applicativi. Responsabilità di questo è la validazione, transformazione ed inoltro dei messaggi tra mittente e destinatari. Poichè mittente e destinatario non sono in comunicazione diretta, il broker introduce un livello di disaccoppiamento, rendendo potenzialmente più robusto il sistema. In particolare, un broker di messaggi può:

- inoltrare messaggi da uno a più destinatari;
- transformare messaggi da una rappresentazione ad un'altra;
- aggregare messaggi diversi o disaggregare singoli messaggi in sequenze di messaggi elementari.
- rendere persistenti i messaggi inviati;
- interagire con un *web service* per recuperare informazioni;
- reagire ad eventi o errori;
- provvedere all'inoltro dei messaggi tramite *topic* utilizzando il modello *publish-subscribe*.

Tutti i messaggi scambiati con un broker possono essere diretti ad una coda. A quel punto, è possibile distinguire due modelli di conversazione, come mostrato in Figura 6.1:

- *one-to-one*: il broker si incarica di inoltrare i messaggi giunti su una coda ad al più uno dei client iscritti a quella coda;
- *one-to-many*: il broker copia il messaggio pubblicato su un *topic* e lo invia a tutti gli iscritti (modello *publish-subscribe*).

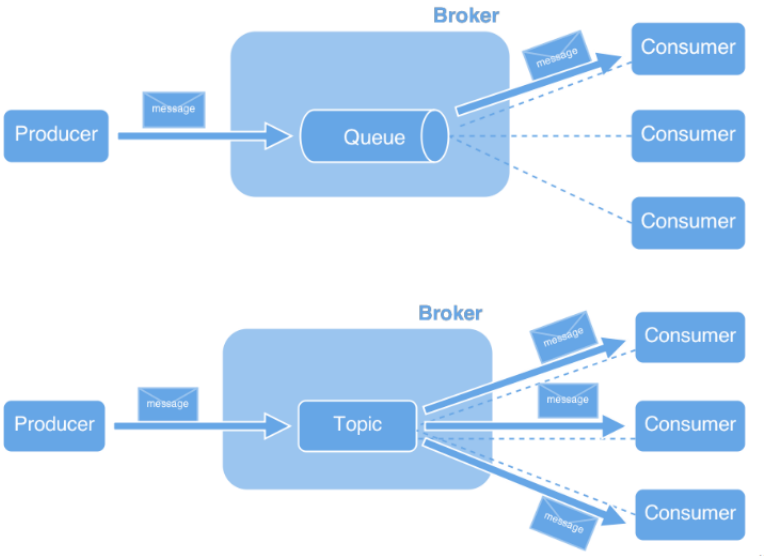


Figura 6.1: Modelli di conversazione

Un broker di messaggi è un componente che appartiene ad una infrastruttura più ampia, al supporto di invio e ricezione di messaggi tra sistemi distribuiti, che prende il nome di *Message-oriented middleware (MOM)*.

6.2 Il protocollo STOMP

Il protocollo *Simple Text Oriented Message Protocol (STOMP)* è un protocollo testuale ideato per lavorare all'interno di un MOM. Esso fornisce l'interoperabilità di comunicazione tra client e broker che supportano tale protocollo, indipendentemente dal linguaggio in cui questi siano stati sviluppati. Tale protocollo è simile al protocollo HTTP, e funziona su TCP. Ogni messaggio dispone di una linea di comando, zero o più *header*, una linea vuota, un corpo ed infine un carattere di terminazione. I messaggi che si originano sui client possono essere divisi in tre categorie:

- Gestione della connessione: CONNECT, DISCONNECT.
- Gestione di transazioni: BEGIN, COMMIT, ABORT.
- Gestione della conversazione: SEND, SUBSCRIBE, UNSUBSCRIBE, ACK, NACK.

I messaggi provenienti dal server possono riguardare:

- Gestione della connessione: CONNECTED, ERROR.
- Gestione della conversazione: MESSAGE, RECEIPT.

Figura 6.2: RabbitMQ

6.3 Architettura e design di RabbitMQ

RabbitMQ è un broker di messaggi che implementa diversi tipi di *messaging protocol*. E' stato uno dei primi broker *open source* ad introdurre un significativo numero di funzionalità, librerie, strumenti di sviluppo, e ben documentato. Esso è in grado di soddisfare diversi modelli di comunicazione. Il funzionamento base è mostrato in Figura 6.2. In particolare, vi è un componente, detto produttore, a cui è affidata la responsabilità di creazione ed invio dei messaggi verso il broker. Dall'altra parte i consumatori si connetteranno alla coda del broker, così da potere ricevere il messaggio. Un software può essere sia un produttore che un consumatore, o entrambi. I messaggi inviati al broker saranno memorizzati all'interno di una o più code, e lì resteranno finchè il consumatore non li avrà prelevati. I messaggi non sono tuttavia recapitati direttamente alla coda ma in realtà questi sono inviati ad un componente interno a RabbitMQ, che prende il nome di *exchange*. Responsabilità di quest'ultimo è quella di inoltrare i messaggi ricevuti verso code distinte, tramite l'utilizzo di *bindings* e *routing keys*. Un *binding* è sostanzialmente un collegamento tra una coda ed un *exchange*. Il flusso di un messaggio all'interno di RabbitMQ è

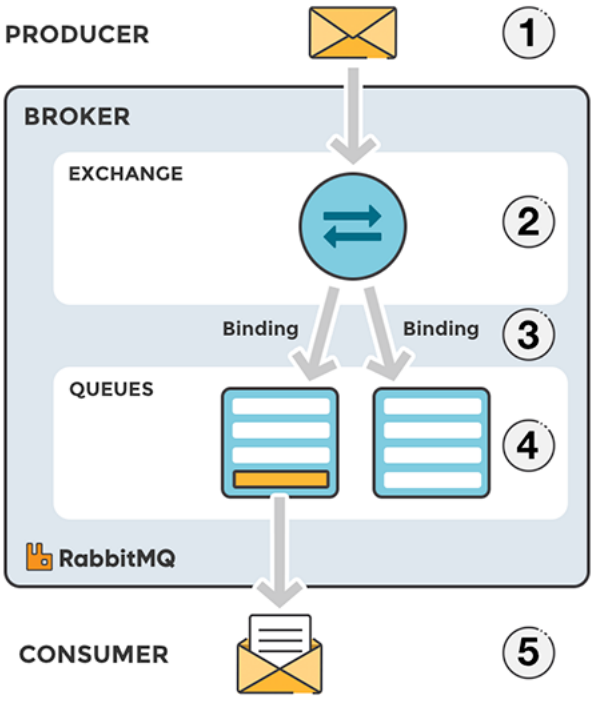


Figura 6.3: Flusso di un messaggio in RabbitMQ

il seguente, come mostrato anche in Figura 6.3: [6]

1. Il produttore pubblica un messaggio verso un *exchange*.
2. L'*exchange* riceve il messaggio ed è responsabile dell'inoltro di questo, prendendo in considerazione diversi fattori, come la *routing key*.
3. Sono creati i *bindings* verso le code ed il messaggio è inoltrato verso queste.
4. Il messaggio persiste all'interno della coda fino a che non viene prelevato dal consumatore.
5. Il consumatore preleva il messaggio.

Si distinguono tuttavia diversi tipi di *exchange*, ognuno dei quali utilizza meccanismi di inoltro diversi, in particolare:

- **Direct exchange:** effettua l'inoltro dei messaggi verso le code sulla base della *routing key*. In particolare, il messaggio è inoltrato verso le code la cui *binding key* corrisponde alla *routing key* del messaggio.
- **Fanout exchange:** effettua l'inoltro dei messaggi verso tutte le code a cui esso è collegato.
- **Topic exchange:** effettua l'inoltro sulla base di un *wildcard match* tra la *routing key* ed il *routing pattern* specificato nel *binding*.
- **Headers exchange:** effettua l'inoltro verso le code sulla base degli attributi specificati all'interno del messaggio.

E' possibile connettersi a RabbitMQ con uno specifico nome utente e password. A ciascun utente può essere assegnato un livello di privilegio all'interno di una data istanza di RabbitMQ. Inoltre ad utente può essere assegnato un livello di privilegio specifico per un dato *virtual host* o *vhost* in RabbitMQ. Un *vhost* rappresenta un namespace per *exchanges*, code e *bindings*.

6.4 Considerazioni

All'interno di questa capitolo è stata analizzata quella che è la natura di un broker di messaggi, quale RabbitMQ. L'utilizzo di quest'ultimo, insieme al protocollo STOMP, all'interno di questo elaborato è stato motivato dall'esigenza di avere un tale meccanismo a supporto della ricerca di un avversario casuale. Si rimanda alla sezione relativa all'implementazione per maggiori dettagli su come è stato gestito lo scambio dei messaggi al fine di raggiungere tale obiettivo.

Parte III

L’implementazione

Capitolo 7

La modellazione dei dati

Con questo capitolo inizia una nuova sezione dedicata all'implementazione del sistema di *backend*. In particolare in questo capitolo saranno inizialmente analizzati i modelli principali caratterizzanti l'applicazione, i quali riflettono le diverse entità in gioco, e le diverse relazioni tra loro. Nei capitoli successivi è gestita l'autenticazione di un utente all'interno del sistema. Infine diversi capitoli saranno incentrati sulla gestione delle sfide, con tutti i dettagli del caso.

7.1 La modellazione dei dati

All'interno del sistema esistono diverse entità, ognuna col proprio ruolo e responsabilità, a cui sono associate tutta una serie di informazioni. Ognuna di queste può essere in relazione con le altre, e si è scelto di utilizzare un approccio misto tra documenti innestati e riferimenti esterni, a seconda del specifico caso d'uso. In particolare, si distinguono i seguenti modelli.

7.1.1 Il modello UserPlayer

Tale documento mantiene tutte le informazioni relative ad un utente, sia quelle necessarie a gestire le credenziali di accesso al sistema sia quelle necessarie a mantenere le statistiche di gioco. Questi può essere registrato all'interno del sistema tramite autenticazione via social network o tramite classica registrazione con email e password. A seconda di quale venga utilizzata all'interno di tale documento saranno memorizzate informazioni differenti. Questo è un esempio di come il non avere uno schema rigido si è tradotto in un grande vantaggio per lo sviluppo. Infatti un utente proveniente da un social network porterà con se diverse informazioni come nome e cognome, che invece non saranno chiesti all'utente qualora effettui la registrazione classica con la propria email. Un utente all'interno del gioco disporrà di un proprio *nickname*, anche se questo sarà identificato sulla base del proprio *id*, univoco all'interno del sistema. Inoltre, un utente può avere diversi stati, quali:

- NEW: un utente appena registrato all'interno del sistema che non ha completato l'accesso al sistema inserendo il proprio *nickname* di gioco.

- NORMAL: un utente correttamente registrato all'interno del sistema.
- BANNED: un utente bannato sarà associato a tale stato.

Oltre agli stati precedenti esiste anche un quarto stato per il modello *User-Player*, ossia BOT, identificativo di un *bot player*. Per ciascun utente sono memorizzate le seguenti informazioni di gioco:

- *nickname*: nome univoco all'interno del gioco;
- *profilePhoto*: url foto profilo dell'utente. Questa può fare riferimento ad una foto all'interno di Facebook o Google oppure ad un avatar all'interno di Mak07;
- *xp*: l'esperienza dell'utente la quale aumenta per ogni partita giocata e schema risolto;
- *count_challenge*: il numero totale di sfide giocate dall'utente;
- *count_schemes*: il numero totale di schemi risolti dall'utente;
- *max_challenge_score*: il punteggio massimo raggiunto al termine di una sfida;
- *max_scheme_score*: il punteggio massimo ottenuto risolvendo un dato schema;
- *solved_time*: sommaria tempi risoluzione schemi. Indica il tempo di gioco totale di un utente.
- *max_scheme_count*: il numero massimo di schemi risolti all'interno di una partita;
- *max_consecutive_victories*: il numero massimo di vittorie consecutive raggiunto;
- *raw_consecutive_victories*: il numero di vittorie consecutive che l'utente sta mantenendo.

Per ogni utente sono inoltre memorizzati altri metadati come il ruolo, che sarà sempre pari a *ROLE_USER*, in quanto non sono previsti altri ruoli, ad esempio amministratore, all'interno del sistema. Sono infine presenti data di creazione e la data di ultima modifica del documento, entrambe memorizzate come *timestamp*. Di seguito in Figura 7.1 è mostrato il documento di un utente registrato tramite Facebook.

```
{
        "_id" : ObjectId("5a33e6994cedfd00016217b8"),
        "_class" : "backend.mak07.model.UserPlayer",
        "nickname" : "carmelo",
        "profilePhoto" : "https://graph.facebook.com/v2.8/1021478
           2392140037/picture?type=large",
        "lvl" : NumberInt(0),
        "xp" : NumberInt(617),
        "status" : "NORMAL",
        "count_challenge" : NumberInt(6),
        "count_schemes" : NumberInt(21),
        "max_challenge_score" : NumberInt(1553),
        "max_scheme_score" : NumberInt(672),
        "max_scheme_count" : NumberInt(5),
        "max_consecutive_victories" : NumberInt(3),
        "solved_time" : NumberInt(546),
        "raw_consecutive_victories" : NumberInt(0),
        "name" : "Carmelo Riolo",
        "firstName" : "Carmelo",
        "lastName" : "Riolo",
        "email" : "carmeloriolo93@gmail.com",
        "roles" : [
                {
                        "role" : "ROLE_USER"
                }
        ],
        "enabled" : true,
        "providerUserId" : "10214782392140037",
        "providerId" : "facebook",
        "createdAt" : NumberLong(1513350809628),
        "modifiedAt" : NumberLong(1515711370552)
}
```

Figura 7.1: Documento di un utente registrato con Facebook e memorizzato in MongoDB.

7.1.2 Il modello Challenge

Cuore del gioco sono le sfide tra utenti, le quali sono modellate dall'oggetto *Challenge*. Ad ogni utente possono essere associate più sfide mentre ad ogni sfida sono associati sempre due utenti, e questa relazione è realizzata mantenendo all'interno della sfida l'identificativo dei rispettivi giocatori all'interno dei campi *player1* e *player2*, dove il primo è sempre colui che ha dato inizio alla sfida ed il secondo colui che deve accettare. Qualora entrambi i giocatori abbiano terminato la partita, all'interno della sfida saranno memorizzati gli schemi risolti, oltre ad essere memorizzati in una collezione a parte, ed ogni

```
{
        "_id" : ObjectId("5a36165d4cedfd000162185a"),
        "_class" : "backend.mak07.model.Challenge",
        "player1" : "5a34dd334cedfd0001621805",
        "player2" : "5a34e52f4cedfd0001621807",
        "type" : "NORMAL",
        "seed" : NumberInt(1149238740),
        "created_at" : NumberLong(1513494109256),
        "modified_at" : NumberLong(1513494243846),
        "lastModifiedBy" : "5a34e52f4cedfd0001621807",
        "expireAt" : NumberLong(1513494109256),
        "state" : NumberInt(4),
        "expireAfter" : null,
        "firebaseIdPlayer1" : "fIIBPhiZz_8:APA91bFHm7-FXnb-aWRS5
           KnOcWNX856k_XrQycIjTYxqC2j8qYDstIv26ciIMTuVCyz5
           buTYp_JM45expUVRpKSBOAf1CX6imJj_ZbTCD5Ceicb_JIhJLlhhcD
           94SDaESo2t8z0DQAhe",
        "firebaseIdPlayer1_timestamp" : NumberLong(1513494119908)
           ,
        "firebaseIdPlayer2" : "d8QWNU9V-aY:APA91bE_YFwEh8JQ4
           rlNGIQ5hThytR6ceIU5EzrfWt4ycRZnP0qhfzqqKP4
           ffwLaAIfiSYmOKnhh1THMkWnywrftyPQhE7CgcXcMJaqZtuO82
           OyszF654X5EJIme2l_ZGe964yMW7lGF",
        "firebaseIdPlayer2_timestamp" : NumberLong(1513494120719)
        "schemesP1": [
        ...
        ]
        "schemesP2": [
        ...
        ]
}
```

Figura 7.2: Documento di una sfida completata e memorizzata in MongoDB.

sfida può quindi contenere zero o più schemi. Questi ultimi sono memorizzati all'interno della sfida separatamente per *player1* e per *player2*, rispettivamente in *schemesP1* e *schemesP2*. Non è detto tuttavia che questi campi siano entrambi presenti poichè magari un utente ha giocato la partita e risolto degli schemi, mentre l'altro ci ha ripensato ed ha abbandonato la partita prima di giocare. E' importante notare come per il caso dei giocatori sia stato utilizzato un modello di relazione tra entità mediante riferimenti esterni, mentre nel caso degli schemi risolti questi siano memorizzati come documento innestato all'interno del documento della sfida stessa. Ad ogni sfida è associato un tipo ed uno stato, in particolare si distinguono le seguenti tipologie di sfide:

- NORMAL: normale sfida in cui un giocatore sfida un altro tramite invito;

- RANDOM: la sfida è stata generata a seguito di ricerca di un avversario casuale ed un altro giocatore connesso è stato trovato ed ha accettato tale sfida;

- RANDOM_BOT: la sfida è stata generata a seguito di ricerca di un avversario casuale ma non è stato trovato alcun giocatore connesso, all'interno della finestra temporale di ricerca, ed è stato sfidato un *bot player*.

Relativamente agli stati che può avere ed attraversare una sfida questi sono diversi. Successivamente sarà analizzata la macchina a stati che descrive le varie transizioni da uno stato all'altro a fronte di un dato evento, e di come questa ha aiutato la realizzazione di un meccanismo di sfida in cui è garantita la correttezza del risultato a fronte di un accesso concorrente al documento di sfida, nonostante l'assenza del meccanismo delle transazioni all'interno di un database non relazionale quale MongoDB. In particolare, si distinguono i seguenti stati:

- Stato 0 (PENDING): la sfida è stata creata ed è in uno stato pendente in quanto lo sfidato non ha ancora accettato o rifiutato.

- Stato 1 (CREATED): lo sfidato ha accettato la sfida e la sfida è stata creata. I due giocatori possono entrambi giocare la partita.

- Stato 2 (SCORE_1): lo sfidante ha giocato la partita e risolto un certo numero di schemi che saranno memorizzati in *schemesP1*.

- Stato 3 (SCORE_2): lo sfidato ha giocato la partita e risolto un certo numero di schemi che saranno memorizzati in *schemesP2*.

- Stato 4 (COMPLETED): entrambi i giocatori hanno giocato la partita, la quale è ora terminata e non è più possibile eseguire ulteriori operazioni su questa.

- Stato -1 (ABANDONED_1): lo sfidante ha rinunciato a giocare la partita. Non è più possibile eseguire alcuna operazione sulla sfida.

- Stato -2 (ABANDONED_2): lo sfidato ha rinunciato a giocare la partita. Non è più possibile eseguire alcuna operazione sulla sfida.

- Stato -3 (DENIED): lo sfidato ha rifiutato l'invito a giocare la partita. Non è più possibile eseguire alcuna operazione sulla sfida.

- Stato -4 (CANCELED): lo sfidante ha annullato l'invito a giocare la partita prima che lo sfidante abbia accettato o rifiutato.

- Stato -5 (TIMEOUT): una sfida casuale non accettata entra in tale stato, oltre il quale non è più possibile accettare o rifiutare tale sfida. Inoltre una sfida in tale stato sarà eliminata anche dalla base dati dopo un certo periodo, in quanto non mantiene alcuna informazione utile.

```
{
        "_id" : ObjectId("5a3404c34cedfd00016217c8"),
        "_class" : "backend.mak07.model.Scheme",
        "code" : NumberInt(5535),
        "playerId" : "5a0eb56a273167a41d1a2be2",
        "challengeId" : "5a3404a84cedfd00016217c5",
        "maxPoints" : NumberInt(924),
        "score" : NumberInt(506),
        "bonus" : NumberInt(33),
        "xp" : NumberInt(54),
        "created_at" : NumberLong(1513358531581)
}
```

Figura 7.3: Documento di uno schema risolto e memorizzato in MongoDB.

Al fine di garantire un corretto *gameplay* per ambo i giocatori, e prevenire la possibilità per un giocatore di giocare la sfida contemporaneamente da più dispositivi, così da conoscere a priori gli schemi che saranno generati, all'interno della sfida sono memorizzate anche le informazioni, per entrambi i giocatori, sul *firebase identifier* del dispositivo che ha deciso di giocare la partita ed il timestamp di quando questo è avvenuto. Sulla base di tale informazioni il sistema bloccherà qualsiasi richiesta di accesso alla partita nell'arco dei due minuti di tempo di gioco, qualora sia presente un identificativo diverso dal dispositivo dal quale si sta tentando di giocare. In particolare queste informazioni sono memorizzate nei campi: *firebaseIdPlayer1*, *firebaseIdPlayer2*, *firebaseIdPlayer1_timestamp* e *firebaseIdPlayer2_timestamp*.
Infine sono presenti ulteriori metadati all'interno del documento in oggetto quali data di creazione, data di ultima modifica ed identificativo del giocatore che ha eseguito l'ultima operazione sulla sfida. E' stato inoltre utilizzato un indice di tipo *Time To Live (TTL)* all'interno di MongoDB, che andrà ad eliminare tutte quelle sfide che tre giorni dopo la data di creazione risultano essere ancora in uno stato pendente. In Figura 7.2 è mostrato il documento di una sfida correttamente completata, dove sono stati volutamente non inseriti gli schemi risolti, la cui struttura sarà analizzata nel paragrafo successivo.

7.1.3 Il modello Scheme

Tale entità modella la risoluzione di un dato schema, combinazione di sette numeri, da parte di un utente. Ad ogni schema sarà associato un identificativo univoco e farà riferimento solamente ad un utente ed una sfida, tramite i rispettivi identificativi. Tutte le relazioni presenti all'interno di tale documento sono espresse quindi mediante riferimenti esterni. Come mostrato in Figura 7.3, oltre alle informazioni appena citate ed ulteriori metadati, all'interno di uno schema sono presenti in particolare:

- *code*: codice della partita giocata, utilizzato per identificare la combinazione di sette numeri;
- *maxPoints*: numero massimo di punti raggiungibili con la migliore risoluzione di tale schema;
- *score*: punteggio ottenuto nella risoluzione dello schema;
- *bonus*: bonus ottenuto con la risoluzione dello schema a seconda del tempo impiegato;
- *xp*: punti esperienza guadagnati con la risoluzione dello schema.

Si è scelto di memorizzare tali informazioni sia all'interno del documento di una sfida, come documento innestato, per entrambi i giocatori, ma anche di memorizzarli in una collezione separata in maniera tale da avere la possibilità in futuro di potere effettuare operazioni di analisi sui vari schemi direttamente su questa.

7.2 Conclusioni

Sono stati visti finora i diversi i documenti che modellano le principali entità, anche se oltre queste ne esistono altre ancora, che si è scelto di non approfondire oltre, come per esempio quelle associate ai vari *token* utilizzati per implementare funzionalità accessorie, come *PasswordResetToken*, *VerificationToken*, *RememberMeToken*. Sviluppi futuri prevederanno molto probabilmente l'aggiunta di nuovi modelli o la modifica della struttura di quelli attualmente esistenti.

Capitolo 8

Autenticazione

Questo capitolo è dedicato a spiegare come è stato realizzato il meccanismo di autenticazione all'interno dell'infrastruttura di *backend*. Come è già stato detto più volte nel corso di questo elaborato, esiste un duplice metodo di autenticazione, via social network o via email. Si analizzerà come è stata realizzata una e come l'altra. Prima però è introdotto quello che è il protocollo OAuth, utilizzato per realizzare l'autenticazione tramite un *authorization server* esterno, come nel caso dell'autenticazione tramite social network.

8.1 Il protocollo OAuth 2.0

OAuth è uno standard per l'autorizzazione, nato nel 2007 come specifica, consolidatosi nel 2010 come *Request For Comments (RFC) 6749*, evoluto oggi in OAuth 2.0, pienamente supportato da Facebook, Google e Microsoft per i loro servizi online. [3] Il framework di autorizzazione OAuth 2.0 abilita un'applicazione di terze parti ad ottenere un accesso limitato ad un servizio HTTP, mantenuto da un *resource server*, tramite l'orchestrazione di una serie di interazioni di approvazione. L'RFC 6749 sostituisce completamente l'ormai obsoleto RFC 5849, relativo al protocollo OAuth 1.0. Nel modello di autenticazione tradizionale tra client e server, il client fa richiesta ad una risorsa ad accesso limitato sul server autenticandosi all'interno di quest'ultimo tramite opportune credenziali. Al fine di fornire accesso ad applicazioni di terze parti è necessario condividere le credenziali di accesso con queste. Ciò si traduce in una serie di problemi e limitazioni: [11]

- l'applicazione di terze parti deve memorizzare le credenziali di accesso per uso futuro;
- i server devono supportare autenticazione tramite password, nonostante tutte le debolezze legate all'uso di queste;
- non vi è la possibilità da parte del *resource server* di limitare l'accesso ad un sottoinsieme di risorse;

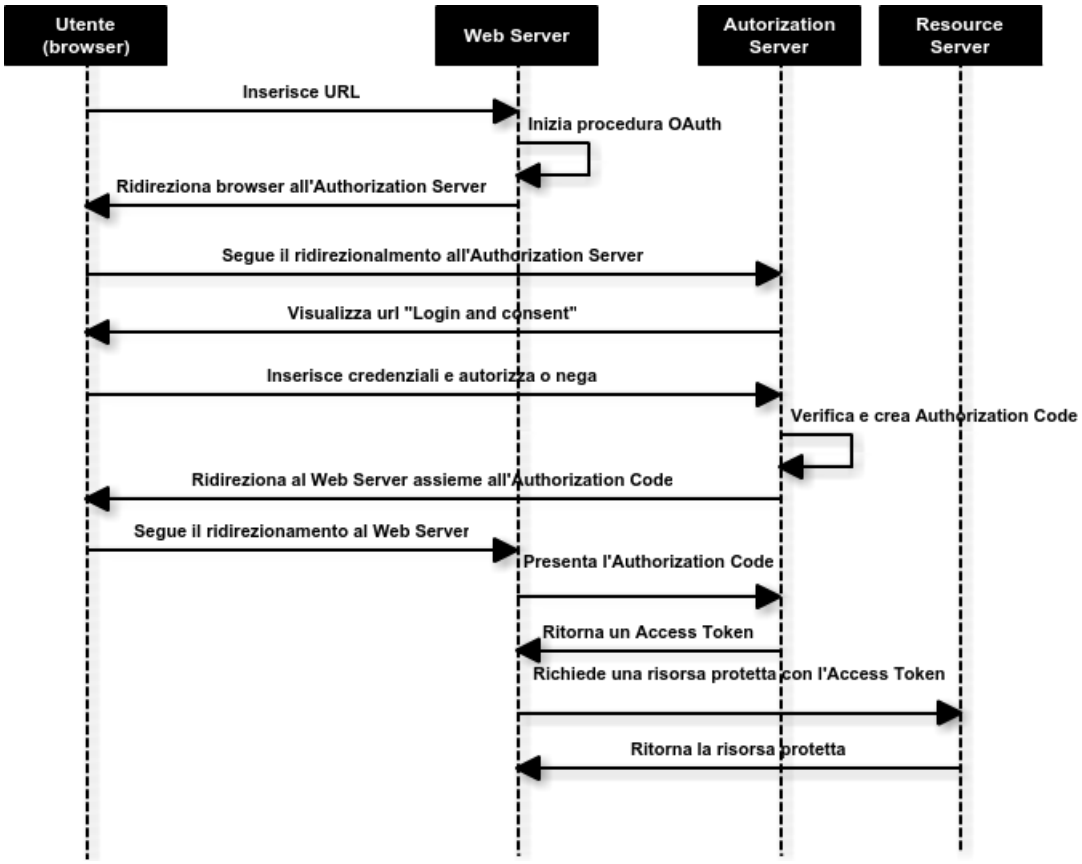


Figura 8.1: Il protocollo OAuth 2.0

- il *resource server* non può revocare l'accesso ad un'applicazione specifica, ma deve revocarlo a tutte invalidando tutte le password di terze parti;

- il compromettersi di un'applicazione di terze parti si traduce nel compromettersi della password dell'utente finale e di tutti i dati protetti da questa.

OAuth risolve questi problemi introducendo un livello aggiuntivo di autenticazione, disaccoppiando il client dal *resource server*. Invece di utilizzare le credenziali fornite dal *resource server* per accedere alle risorse, il client ottiene un *access token*. Questi sono forniti ai vari client da un *authorization server* a seguito dell'approvazione del *resource owner*. Tale protocollo è stato pensato per lavorare con il protocollo HTTP. I protagonisti principali del protocollo OAuth sono:

- *resource owner*: entità in grado di garantire l'accesso ad un insieme protetto di risorse. Quando questo è una persona, prende il nome di *end-user*;

- *resource server*: il server dove sono memorizzate le risorse protette, in grado di rispondere a richieste di accesso a tali risorse tramite l'utilizzo di *access token*;

- *client*: applicazione la quale fa richiesta di accesso alle risorse protette per conto del *resource owner* e della sua autorizzazione;
- *authorization server*: il server il quale fornisce l'*access token* al client dopo avere autenticato il *resource owner* ed avere ottenuto la sua autorizzazione.

L'*authorization server* può talvolta coincidere con il *resource server*, oppure essere un entità distinta. In particolare, si descrive di seguito il flusso OAuth generato da un'applicazione web, rappresentato in Figura 8.1:

1. L'utente inserisce l'url di un'applicazione web.
2. L'applicazione web ha la necessità di accedere alle risorse dell'utente sul *resource server* quindi redirige il browser dell'utente verso l'*authorization server*.
3. Sull'*authorization server* l'utente effettua l'autenticazione e fornisce l'autorizzazione di accesso alle sue risorse all'applicazione web.
4. L'*authorization server* genera un *authorization code* ed uno *shared secret* e li inserisce come parametri in una url di ridirezionamento che rimanda l'utente all'applicazione web.
5. L'applicazione web presenta l'*authorization code* all'*authorization server* il quale gli restituisce un *access token*.
6. L'applicazione web richiede al *resource server* una risorsa protetta presentando l'*access token*
7. Il *resource server* ritorna la risorsa protetta.

Talvolta la durata dell'*access token* tuttavia è temporanea. In questi casi quando viene restituito questo, viene anche restituito un *refresh token* il quale è da presentare all'*authorization server* per ottenere un nuovo *access token* valido.

8.2 Configurazione generale

All'interno dell'applicazione Spring al fine di realizzare i meccanismi di autenticazione preposti sono stati utilizzati diversi AuthenticationProvider. All'interno di Spring Security la responsabilità circa l'autenticazione è demandata ad un AuthenticationManager, la cui implementazione di default è rappresentata dalla classe ProviderManager. Quello che fa un AuthenticationManager è iterare tra le diverse implementazioni dell'interfaccia AuthenticationProvider che sono state registrate e configurate, ed alle quali è demandato il compito finale di autenticare l'utente, tramite il metodo *authenticate*.

```
@Override
protected void configure(HttpSecurity http) throws Exception
{
        http
        .anyRequest().authenticated()
        ...
        .and()
        .formLogin()
        .loginPage("/login")
        .usernameParameter("username")
        .passwordParameter("password")
        .successHandler(customDaoAuthenticationSuccessHandler())
        .failureHandler(customDaoAuthenticationFailureHandler())
        .and()
        ...
        .rememberMe()
        .rememberMeServices(
           persistentTokenBasedRememberMeServices())
        .key(PROVIDER_KEY)
        .and()
        ...
        });
}

@Bean
public DaoAuthenticationProvider daoAuthenticationProvider()
{
        CustomDaoAuthenticationProvider provider = new
           CustomDaoAuthenticationProvider(userPlayerService);
        provider.setPasswordEncoder(passwordEncoder());
        provider.setUserDetailsService(userDetailsService());
        return provider;
}
```

Figura 8.2: SecurityConfiguration.java

Sono stati configurati i seguenti AuthenticationProvider:

- DaoAuthenticationProvider: utilizzato per autenticare un utente tramite username e password. Le credenziali sono memorizzate in maniera persistente all'interno del database. In particolare, al fine di garantire la confidenzialità delle informazioni, relativamente alla password, è stata utilizzata la libreria BCrypt per realizzare la cifratura.

- SocialAuthenticationProvider: utilizzato per autenticare un utente tramite *authorization server* esterno, fornito da un social network.

- RememberMeAuthenticationProvider: l'autenticazione viene verificata tramite RememberMeToken, un utente precedentemente autenticato sarà provvisto di tale cookie che presentato all'interno della richiesta HTTP, permette di aggiornare la sessione e mantenere l'autenticazione all'interno del sistema.

La configurazione di Spring Security e la configurazione degli AuthenticationProvider appena citati è stata effettuata all'interno della classe *SecurityConfiguration.java*, in particolare all'interno del metodo *configure*, come mostrato in Figura 8.2, dove per questione di eccessiva verbosità alcune parti della configurazione sono state omesse. Tramite il metodo *formLogin* viene configurato un *endpoint* per l'autenticazione tramite nome utente e password, la quale si appoggia su un DaoAuthenticationProvider, il quale viene registrato come bean all'interno del framework Spring. Invece tramite il metodo *rememberMe* è possibile registrare e configurare il RememberMeAuthenticationProvider. Per quanto riguarda la registrazione del SocialAuthenticationProvider questa viene realizzata di default dalla dipendenze *spring-social-facebook* o *spring-social-google*, successivamente alla configurazione di *app secret* e *app id* relativi a Facebook o Google.

8.3 Autenticazione tramite social network

Si analizza all'interno di questo paragrafo il flusso OAuth 2.0 seguito al fine di effettuare l'autenticazione all'interno del sistema. Le dipendenze necessarie sono *spring-social-facebook* e *spring-social-google*. Questo meccanismo è utilizzato per ottenere l'*access token* così da contattare il *resource server* del provider, e successivamente registrare un nuovo utente all'interno del server di Mak07. L'utente creato sarà popolato con tutte le informazioni ottenute dal *resource server*, per le quali l'utente, *resource owner*, ha dato l'autorizzazione. Per semplicità, si analizza di seguito solo il caso di autenticazione via Facebook:

1. Un client di gioco che vuole effettuare l'autenticazione via Facebook contatta il server effettuando la seguente richiesta HTTP:
   POST https://game.mak07.com/auth/facebook.

2. Il server risponde con una risposta HTTP con stato pari a 302, redirezionandolo verso Facebook.

3. L'utente si presenta su Facebook con un *client id* ed uno *state*. Successivamente viene chiesto all'utente di autenticarsi tramite le proprie credenziali, e di autorizzare l'accesso alle risorse richieste dall'applicazione Mak07 registrata su Facebook. Se tutto è andato a buon fine al client viene restituito un *authorization code*.

4. Il client ricontatta il server game.mak07.com fornendo l'*authorization code* ottenuto nelle fasi precedenti. A questo punto è il server stesso che

contatta Facebook per verificare la validità di tale *authorization code* ed ottenere un *access token* in cambio.

5. L'*access token* è utilizzato per prelevare le informazioni relative all'utente da Facebook, quale *provider id*, nome, cognome, email e url della foto profilo. Queste sono utilizzate per riempire il documento UserPlayer che sarà inserito all'interno del database. Accanto a questo verrà creato un documento *UserConnection*, dove sono mantenute le informazioni circa *providerId*, *providerUserId* e *accessToken*. Ogni *UserConnection* è associata ad un dato *UserPlayer*, relazione realizzata tramite riferimento esterno.

6. Una volta terminato con successo tale flusso, il server game.mak07.com ridireziona il client verso l'url finale. Questa può essere arbitraria. Si è scelto di ritornare al dispositivo mobile un *deep link* per far ritornare l'utente all'interno del gioco così da potere completare la registrazione, qualora si tratti del primo accesso, o essere rimandato alla schermata principale del gioco.

8.4 Autenticazione tramite indirizzo email

Un'alternativa a quanto appena visto è l'autenticazione tramite indirizzo email e password, per la quale è stato necessario implementare anche un opportuno meccanismo di registrazione. I principali responsabili sono il *RegistrationController* ed il *SignupService*. Il controller espone le seguenti risorse:

- **POST /signup**. Risorsa utilizzata per registrare un nuovo utente all'interno del sistema. Il *body* della richiesta dovrà contenere un JSON con indirizzo email e password. Viene restituito errore qualora i dati inviati siano vuoti o non validi, la password ad esempio deve garantire un certo livello di robustezza. Oppure viene ritornato un errore se l'utente associato a quell'indirizzo email è già presente all'interno del sistema. Qualora i dati siano corretti un utente è creato e memorizzato all'interno del database, nonostante prima di potersi autenticare all'interno del sistema occorre che questo sia abilitato tramite email di conferma.

- **POST /resendLink**. Risorsa utilizzata per reinviare l'email di verifica account all'utente. Al fine di realizzare la verifica dell'utente un *VerificationToken* è generato e memorizzato in un opportuna collezione del database. A ciascun *VerificationToken* è associato uno stato che può essere CONFIRMED, NOT_CONFIRMED, o INVALID, rispettivamente se il token è stato utilizzato, non utilizzato o è diventato invalido, poichè è stato richiesto un nuovo token o il token stesso è scaduto.

- **GET /registrationConfirm**. Risorsa utilizzata per confermare un account ed abilitarlo. Riceve come parametro all'interno della url il token

associato all'utente da verificare ed abilitare. Viene ritornato un errore qualora l'utente non venga trovato, o sia già stato abilitato oppure il token specificato non sia più valido.

- **POST /login**. Risorsa utilizzata per autenticare un utente all'interno del sistema. All'interno del corpo della richiesta devono essere contenute le credenziali di accesso, ossia email e password.

- **POST /forgetPassword**. Tale risorsa realizza il meccanismo di recupero password. All'interno del *body* della richiesta viene passata l'email, e se questa è associata ad un utente presente all'interno del sistema viene generato un *PasswordResetToken* e memorizzato nel database. All'interno di tale token sarà presente un campo *code*, che sarà inviato all'utente via email, e sarà necessario per aggiornare la password.

- **GET /recovery**. Tale risorsa riceve come parametro il *code* relativo ad un *PasswordResetToken* e ne verifica la validità. Quest'ultimo può essere NOT_USED, USED o INVALID a seconda che sia stato usato o meno oppure sia diventato invalido, poichè scaduto, o invalidato da una nuova richiesta di recupero password.

- **POST /updatePassword**. Risorsa utilizzata per aggiornare la password dell'utente. Riceve all'interno del corpo della richiesta l'email associata all'utente, il *code* ricevuto via email e la nuova password, e se tutti questi sono validi la password è aggiornata.

8.5 Sessioni e persistenza

Una volta autenticato, ad un utente sarà associata una sessione. Questa si basa sul meccanismo dei *cookie*, in quanto una volta effettuato il login, via social o meno, il server risponderà ritornando al client due *cookie*, all'interno dell'header HTTP *Set-Cookie*. Questi *cookie* hanno il nome di JSESSIONID e Remember-Me-Token.
Il *JSESSIONID* è utilizzato dal server per identificare una certa sessione, ed ha di default una validità di due ore, oltre le quali la sessione non è più valida. Una richiesta con un JSESSIONID non valido o assente si traduce in una richiesta non autenticata, senza permessi sufficienti per accedere alle risorse protette.
Il Remember-Me-Token è utilizzato per realizzare la persistenza delle sessioni, infatti questo è memorizzato all'interno del database e qualora un utente si presenti con un JSESSIONID scaduto o non valido, ma con un Remember-Me-Token valido, viene generata una nuova coppia JSESSIONID e Remember-Me-Token, e restituita al client, memorizzando anche il nuovo Remember-Me-Token nel database. Questo meccanismo permette inoltre di riavviare il server senza che gli utenti debbano ogni volta riautenticarsi all'interno del sistema.

8.6 Conclusioni

All'interno di questo capitolo sono state analizzate le scelte adottate al fine di implementare un meccanismo di autenticazione per Mak07. In particolare grande sforzo è stato impiegato per integrare il flusso OAuth all'interno di un contesto *mobile*, in maniera tale da potere reindirizzare il client verso una *deep url*. Successivamente è stato necessario integrare all'interno di un contesto già operativo un altro meccanismo di autenticazione via username e password, e ciò è stato possibile solo dopo avere studiato approfonditamente il funzionamento di Spring Security, dei vari *Authentication provider* e della *Security Filter Chain*, così da rispettare tutte le linee guida del framework Spring. In conclusione è possibile affermare che il codice realizzato è tale da permettere facilmente l'integrazione futura di ulteriori meccanismi di autenticazione accanto a quelli esistenti, ad esempio tramite l'aggiunta di nuovi *authorization server* da contattare.

Capitolo 9

Gestione delle sfide

Questo capitolo è dedicato all'analisi degli algoritmi e delle scelte implementative che hanno permesso di realizzare le sfide tra giocatori. Come già detto più volte nel corso di questo elaborato sono state realizzate due modalità di sfida, anche se sostanzialmente la modalità di gioco è sempre la stessa, e quella che cambia è la ricerca e scelta dell'avversario.

9.1 Descrizione generale

Prima di descrivere il flusso generale di chiamate necessarie per creare e giocare una sfida si analizzano di seguito gli *endpoint* realizzati a supporto di ciò. Questi sono stati realizzati al fine di permettere a due utenti di giocare tra loro ma anche per fornire ad un utente l'accesso alle proprio sfide. Tutte le seguenti risorse sono protette, occorre quindi autenticarsi prima di potere effettuare la richiesta, altrimenti verrà restituito un codice di errore 403 *Forbidden*. Si distinguono:

A.1 **GET /api/challenges**. Ritorna all'utente che ha effettuato la richiesta, ossia il giocatore corrente, tutte le sfide attive, ossia che si trovino in uno stato dove è ancora possibile effettuare nuove transizioni, più quelle che sono state terminate nell'arco degli ultimi tre giorni.

A.2 **GET /api/challenges/id**. Ritorna una sfida identificata da un certo *id* o 404 *Not Found* se non esiste alcuna risorsa associata a quell'*id*.

A.3 **PUT /api/challenges/id**. Permette di modificare una sfida identificata da un certo *id*. In questo caso il client deve passare attraverso il *body* della richiesta un *IncomingChallengeDTO* contenente il nuovo stato della sfida ed eventuali schemi risolti se il nuovo stato è SCORE_1 o SCORE_2. Viene ritornato un errore, associato al codice 403 *Forbidden*, qualora non sia possibile aggiornare la sfida con il nuovo stato richiesto, poichè in conflitto con la macchina a stati, che sarà introdotta nel paragrafo successivo.

A.4 **PUT /api/challenges/id/play**. Registra l'inzio di una partita da parte di un giocatore. E' necessario passare all'interno del *body* della richie-

sta un *IncomingChallengeDTO* specificando il *firebase id* del giocatore. Questo viene utilizzato per associare l'inizio della partita ad un dato dispositivo.

A.5 **GET /api/challenges/id/schemes**. Ritorna tutti gli schemi risolti dal giocatore corrente per la sfida identificata da *id*.

A.6 **POST /api/challenges/player2Id**. Risorsa utilizzata per invitare un altro giocatore a prendere parte ad una sfida. Tale secondo giocatore assumerà il ruolo di *player2* all'interno del modello *Challenge*. Inoltre, a fronte di tale chiamata verrà contattato il server di Firebase e l'utente sfidato sarà notificato del fatto di avere ricevuto un invito a giocare.

A.7 **GET /api/players/id/challenges**. Ritorna tutte le sfide del giocatore associato all'identificativo *id*.

A.8 **GET /api/players/id/challenges/status**. Ritorna tutte le sfide, che abbiano stato pari a quello specificato in URL, del giocatore associato all'identificativo *id*.

Il flusso generale è tale per cui un utente che voglia sfidare un altro avversario, e di cui conosce il nickname, deve prima eseguire la chiamata **A.6**. Successivamente nel momento in cui vuole giocare la partita, deve comunicare al server l'inizio di questa tramite la chiamata **A.4**, ed infine tutte le successive modifiche avvengono invocando più volte la **A.3** e specificando il nuovo stato della sfida.

9.2 La macchina a stati

L'utilizzo di un modello non relazionale ha portato con se diversi vantaggi durante lo sviluppo. Tuttavia la più grande carenza per gli scopi dell'elaborato,

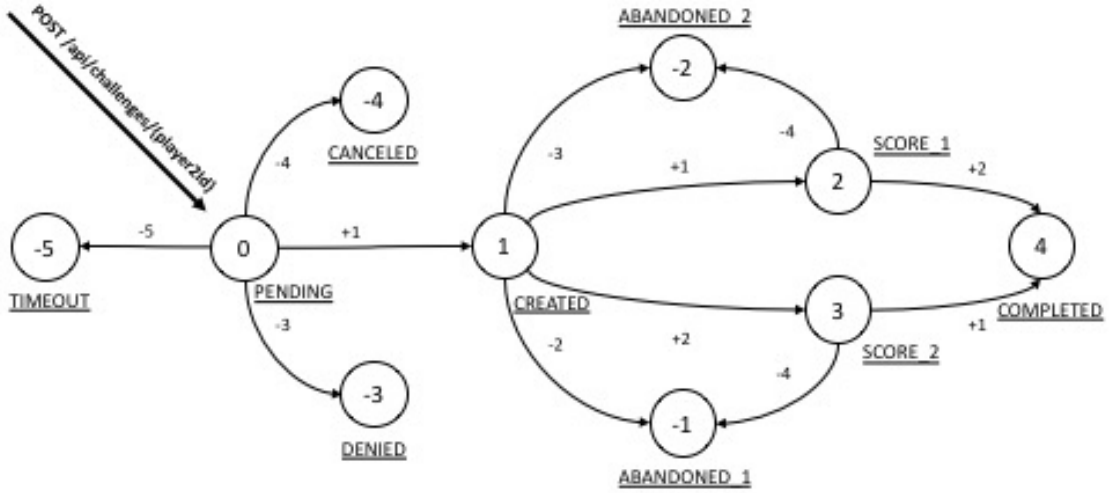


Figura 9.1: La macchina a stati finiti

ed in particolare modo per la parte di sviluppo dedicata all'implementazione delle sfide, è rappresentata dalla mancanza del meccanismo delle transazioni. Non potendo utilizzare un siffatto meccanismo è stato necessario trovare una soluzione alternativa per gestire le sfide tra giocatori, in cui l'accesso all'oggetto sfida avviene in maniera concorrente, sia in lettura che scrittura da parte dei due. Una cattiva o addirittura assente gestione della concorrenza si sarebbe tradotta in un risultato finale sbagliato. La soluzione adottata è stata quella di sfruttare le operazioni di MongoDB che permettono l'accesso atomico in lettura e scrittura al singolo documento, in particolare è stata utilizzata la funzione *findAndModify()* che garantisce l'atomicità nell'accesso e aggiornamento del documento. Se a seguito di una data *query* viene tornato un certo documento, che soddisfi determinati criteri, è garantito che nessuno possa modificarlo prima che sia correttamente aggiornato. Sulla base di ciò si è sviluppata la seguente macchina a stati finiti, rappresentata in Figura9.1, che rappresenta i possibili percorsi che un oggetto sfida può attraversare, da quando viene creata a quando è terminata. Come già introdotto in 7.1.2, una sfida può esistere e trovarsi in un determinato stato, che può evolversi nel tempo, a seguito di determinati eventi, legati alle azioni del giocatore. In particolare a seguito di **A.6** viene creata una nuova sfida nello stato 0. Tutti i futuri passaggi di stato avverrano a seguito di **A.3**. A partire dallo stato 0 le possibilità sono tre: il giocatore che ha creato la sfida la cancella, e la sfida passa allo stato -4; l'avversario non accetta la sfida che passa allo stato -3; l'avversario accetta la sfida che passa allo stato 1. Solo quest'ultimo stato può portare la sfida verso nuovi stati, mentre gli altri due rappresentano due foglie della macchina a stati. A partire dallo stato 1 sono molteplici le possibilità, i giocatori possono decidere di giocare la partita, così da andare in 2 o 3, o di abbandonare la partita, così da andare in -1 o -2. Se la sfida si trova allo stato 2, vuol dire che la partita è stata giocata dal *player1*, a questo punto può andare o in -2, in quanto l'avversario ha abbandonato, o in stato 4, dopo che anche quest'ultimo abbia giocato. Situazione analoga e speculare è quella della sfida allo stato 3, che può portare la sfida allo stato -1 o 4. In generale l'idea di identificare lo stato della sfida con un intero, ed utilizzare la *findAndModify()* di MongoDB ha permesso di realizzare un meccanismo per gestire l'accesso concorrente, tale per cui è garantito il corretto passaggio da uno stato all'altro. In Figura 9.2 è mostrata una porzione di codice invocata nel momento il cui il *player1* giocherà la partita ed invierà gli schemi risolti al server. Si può notare come il criterio di base è tale da andare ad eseguire la ricerca sulla base dell'*id* della sfida, ma questa deve trovarsi in uno stato 0 o 3, e successivamente eseguire l'incremento di una unità, ed è garantita l'atomicità di queste due operazioni all'interno della singola istruzione *findAndModify*, ossia nel momento in cui viene restituita la sfida questa sarà anche aggiornata come specificato, il tutto come un un'unica operazione indivisibile.

```
1  private Challenge p1Score(Challenge challenge) throws
       BadUpdateOperation
2  {
3
4          Query query = new Query();
5          Criteria criteria = new Criteria().andOperator(
6          Criteria.where("id").is(challenge.getId()),
7          new Criteria().orOperator(
8          Criteria.where("state").is(Challenge.
               getChallengeStateValue(Challenge.ChallengeState.
               CREATED)),
9          Criteria.where("state").is(Challenge.
               getChallengeStateValue(Challenge.ChallengeState.SCORE_
               2))
10         )
11         );
12         query.addCriteria(criteria);
13
14         Update update = new Update();
15         update.inc("state", 1);
16         update.set("schemesP1", challenge.getSchemesP1());
17         long now = System.currentTimeMillis();
18         update.set("lastModifiedBy", challenge.getPlayer1());
19         update.set("modified_at", now);
20         Challenge updatedChallenge = template.findAndModify(query
               ,
21         update,
22         FindAndModifyOptions.options().returnNew(true),
23         Challenge.class);
24
25         return updatedChallenge;
26 }
```

Figura 9.2: Snippet p1Score

9.3 Ricerca di un avversario casuale

Nella modalità basate su sfide tramite invito è necessario conoscere a priori il *nickname* dell'avversario. Secondo tale modello è inoltre necessario aspettare che l'avversario accetti, o rifiuti, l'invito a giocare la partita, e quindi il giocatore si trova in una condizione in cui deve aspettare un tempo indeterminato. Da ciò l'idea di implementare una seconda modalità che permette all'utente di giocare una partita in breve tempo, selezionando casualmente un avversario tra quelli attualmente connessi al sistema di gioco, tramite opportuno meccanismo di *e-presence*. Come è già stato detto diverse volte, ma è bene ribadirlo, un giocatore connesso è tale da avere attualmente in esecuzione il gioco al-

l'interno del proprio dispositivo mobile, o in altri termini in *foreground*, e non impegnato in alcuna sfida. Per realizzare ciò è stato realizzato uno studio volto ad inviduare quale fosse la tecnologia migliore che potesse permettere ciò. La scelta è ricaduta sull'uso della tecnologia Websocket insieme ad un broker di messaggi. La tecnologia Websocket fornisce canali di comunicazione bidirezionali attraverso una singola connessione TCP, facilitando la realizzazione di applicazioni che forniscono contenuti in tempo reale. Per garantire inoltra l'interoperabilità tra client, server e broker è stato utilizzato il protocollo STOMP come formato per la rappresentazione dei messaggi. Utilizzando le Websocket è possibile tracciare tutti gli utenti per i quali il gioco è attualmente in esecuzione, in quanto sono tutti quelli che avranno mandato un messaggio di avvio connessione, ma non di disconnessione. Un eventuale messaggio di disconnessione sarà ricevuto ogni qualvolta il gioco sarà chiuso, messo in *background*, o quando l'utente avrà iniziato una nuova sfida, così da non essere disturbato. Al termine della sfida l'utente segnalerà nuovamente la sua presenza tramite un nuovo messaggio di connessione. Di conseguenza un client che si connette e vuole giocare una partita contro un avversario casuale invierà, in ordine, i seguenti messaggi:

1. CONNECT.
2. SUBSCRIBE */user/queue/challenges*.
3. SEND */app/challenges* {"sender": *user_id*, "type": "WANT_TO_PLAY"}.

Analizzando il flusso appena elencato, inizialmente il client si connette alla websocket ed esegue la *subscribe* alla coda del broker. Successivamente nel momento in cui vuole giocare invia un messaggio di tipo WANT_TO_PLAY verso la destinazione */app/challenges*. Il server di gioco allora sceglierà un utente casuale tra quelli che hanno eseguito la *subscribe*, se possibile, e creerà un nuovo documento sfida, allo stato 0. Successivamente il broker avrà la responsabilità di inoltrare il messaggio di invito verso l'utente selezionato. Una volta recapitato il messaggio, le possibilità sono due:

- l'utente accetta l'invito a giocare la partita, presentatogli da uno sfidante casuale, e di conseguenza il client esegue un aggiornamento della sfida portandola allo stato 1, tramite **A.3**;
- l'utente rifiuta ed invia sulla websocket, alla destinazione */app/challenges/reply*, un messaggio del tipo:
  {"sender": *user_id*, "type": "RANDOM_CHALLENGE_DENY"}.
  La sfida che è stata rifiutata viene aggiornata allo stato -3 (DENIED).

Tuttavia, essendo un meccanismo realizzato con lo scopo di permettere ad un utente di giocare in tempi brevi, è stato introdotto anche un altro stato per la sfida, ossia -5 (TIMEOUT). E' stato implementato un *task* schedulato, eseguito ogni minuto, che controlla l'esistenza di sfide di tipo RANDOM allo stato 0. Se al momento in cui questo è eseguito, viene trovata una sfida che soddisfa tali condizioni, e che è stata creata da più di 30 secondi, allora viene aggiornata

allo stato -5, scelto un nuovo avversario casuale tra quelli connessi, creata una nuova sfida e mandato l'invito. Quindi nel caso peggiore chi ha lanciato la richiesta, qualora un utente non risponda in alcun modo all'invito a giocare, si troverà ad aspettare circa 2 minuti e 30 secondi per ogni invito. Nel caso in cui, siano già stati sfidati 3 utenti casuali e nessuno di questi abbia accettato a giocare, o la richiesta sia andata in timeout, viene selezionato un bot per giocare con l'utente. Il meccanismo dei bot sarà spiegato nel paragrafo successivo. In generale il meccanismo finora descritto funzionerà meglio, quanti più saranno gli utenti connessi, in quanto sarà maggiore la possibilità di trovare un utente connesso e di conseguenza di giocare la partita in tempi brevi. Anche quando un utente rifiuti, questo sarà subito elaborato e selezionato un nuovo avversario ed inoltrato l'invito. Se al momento della richiesta non vi sono altri utenti connessi, viene automaticamente creata la sfida contro un bot. Infine, al fine di rendere tale meccanismo il meno invasivo e fastidioso possibile, per gli utenti che vengono sfidati casualmente e devono ricevere tali inviti, è stata gestito anche il caso in cui se un utente rifiuta un invito casuale, per un certo tempo non riceve più alcun invito. Il rifiuto a giocare una partita contro un avversario casuale, viene modellato tramite il documento *rejection*. Ulteriore controllo è stato realizzato al fine di evitare che per ogni ricerca casuale venga sfidato più volte uno stesso utente, e ciò è stato realizzato mantenendo una storia degli utenti sfidati.

9.4 Il meccanismo dei bot

Qualora non sia stato trovato entro un certo tempo e dopo diversi tentativi un avversario contro cui giocare, oppure non vi sia alcun altro giocatore connesso, viene creata creata una sfida contro un *bot player*. Questo meccanismo è supportato da un secondo *task* schedulato ogni 30 secondi, che andrà a prelevare dal database tutte le sfide con tipo RANDOM_BOT e stato 0, e per ognuna di queste verrà scelto un bot casuale, tra quelli presenti nel sistema, che andrà a giocare la partita. Ogni bot è modellato come un utente vero e proprio, rappresentato dal documento *UserPlayer*, dove però il tipo specificato sarà appunto BOT. Di conseguenza esso è un entità vera e propria all'interno del gioco, con cui ogni utente può interagire, ed avrà le proprie statistiche che evolveranno per ogni sfida giocata. Ai fini di non interferire troppo nell'esperienza di gioco dell'utente, non è stata data ai bot la possibilità di accettare inviti a giocare esplicitamente ricevuti dagli utenti. I bot sono inoltre esclusi da tutte le classifiche di gioco. Ma come fa un bot a giocare una partita? Ciò è realizzato precalcolando per ogni possibile schema tutte le possibili operazioni che possono portare ad avere come risultato zero. Ossia un bot gioca conoscendo a priori le soluzioni dello schema, le quali possono essere diverse, e verranno selezionate dal bot in maniera casuale. Ognuna di queste soluzioni può portare ad un punteggio più alto o più basso, ed ordinando queste in maniera crescente, è possibile fare in modo che il bot scelga una soluzione che porti ad un punteggio maggiore o meno. Risolto uno schema, il bot genera un

Figura 9.3: Google Firebase

tempo casuale e passa al successivo, quando la somma dei tempi supera i due minuti, questo smette di giocare. Registrando i tempi di gioco dell'utente ed il numero di schemi risolti, è possibile ottenere il tempo medio impiegato per risolvere uno schema, e quindi fare giocare il bot in maniera compatibile con le capacità dell'utente, generando ogni volta un tempo casuale che stia nell'intorno del tempo medio dell'utente. All'interno del sistema sono attualmente presenti 44 bot diversi.

9.5 Gestione delle notifiche

Una volta invitato a partecipare ad una sfida il giocatore sarà informato di tale evento mediante un'opportuna notifica. Il meccanismo di notifica realizzato prevede l'uso di Google Firebase come mostrato in Figura 9.3. Al fine di supportare ciò è stato necessario memorizzare il *firebase identifier* di ciascun dispositivo associato all'utente, il quale viene memorizzato, o aggiornato, eseguendo una chiamata HTTP PUT verso */api/{firebaseId}* ed aggiornando tale identificativo ogni qualvolta l'utente esegue l'accesso all'interno dell'applicazione. Successivamente quando sarà necessario notificare l'utente sarà responsabilità del server Spring contattare le API di Firebase in maniera tale da inviare una notifica verso il dispositivo. E' stata realizzata l'interfaccia *FirebaseManagementService*, responsabile dell'interazione con Firebase, che dispone dei seguenti metodi:

1. *public void sendMessageChallenge(FirebaseMessageChallenge messageChallenge);*
2. *public void sendUpdateChallenge(FirebaseUpdateMessageChallenge updateMessageChallenge).*

In particolare il metodo *sendMessageChallenge* ha lo scopo di notificare all'utente che la sfida è stata accettata dall'avversario e creata, invece il metodo *sendUpdateChallenge* quello di notificare eventuali aggiornamenti relativi ad un cambiamento dello stato della sfida. Dei due soltanto il primo è utilizzato per comunicare con l'utente, il secondo invece è utilizzato dal client mobile per aggiornare in tempo reale lo stato dell'interfaccia utente. E' tralasciata la descrizione dei modelli *FirebaseMessageChallenge* e *FirebaseUpdateMessage-*

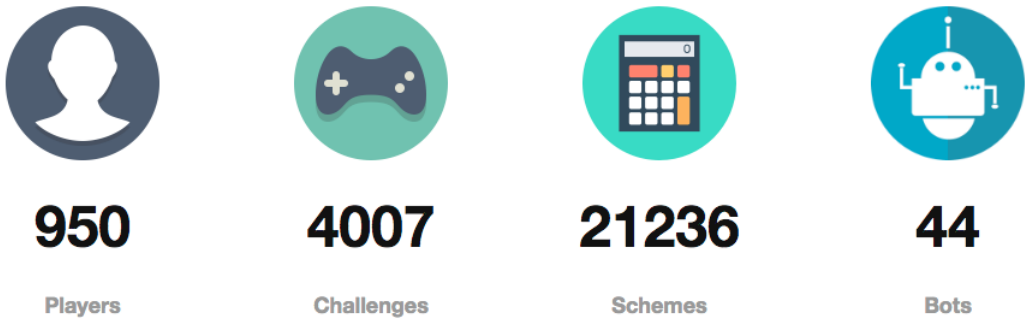


Figura 9.4: Contatori di gioco

Challenge in quanto questi mantengono solo diverse informazioni di contesto scambiate con Firebase al fine di consentire l'inoltro della notifica.

9.6 Funzionalità secondarie

Segue una breve introduzione circa alcune funzionalità che sono state sviluppate a supporto dell'architettura realizzata, anche se la loro importanza risulta essere di secondaria importanza rispetto ai principali meccanismi che permettono il funzionamento globale del sistema.

9.6.1 Monitoraggio risorse

E' possibile accedere al pannello di amministrazione realizzato per il gioco per avere una stima delle statistiche di gioco. Le seguenti risorse sono state definite:

- **GET /admin/count**. Ritorna il numero di utenti registrati, il numero di sfide giocate, il numero di schemi risolti ed il numero di bot presenti all'interno del sistema. Un'applicazione è mostrata in Figura 9.4.

- **GET /admin/challengesDates**. Ritorna una lista di elementi in cui viene specificato per ogni giorno il numero di sfide giocate. E' possibile specificare dei parametri aggiuntivi *from* e *to* in maniera tale da effettuare la ricerca limitatamente ad un certo periodo di tempo.

- **GET /admin/users**. Permette di scaricare uno *stylesheet* contenente le principali informazioni degli utenti registrati al sistema.

- **GET /admin/users/ranking**. Ritorna una classifica di utenti in ordine decrescente di *xp*. E' possibile specificare un parametro aggiuntivo *limit* per ottenere solo un numero limitato di risultati.

9.6.2 FollowsService e FriendshipService

Ulteriori due servizi sono stati realizzati al fine di aggiungere nuove funzionalità. In particolare è stato realizzata l'interfaccia *FollowsService* in maniera tale da permettere all'utente di mantenere dei preferiti, ossia amici da seguire e potere sfidare con maggiore semplicità. Tale interfaccia è a supporto delle seguenti risorse:

- **PUT /api/players/follow/{id}**. Permette di registrare un amico come preferito.
- **PUT /api/players/unfollow/{id}**. Annulla l'operazione precedente.

Altra interfaccia realizzata è quella che prende il nome di *FriendshipService*. Questo servizio ha la responsabilità di memorizzare per ciascuna coppia di utenti, in maniera non commutativa, tutte le informazioni circa la loro storia. Storia intesa come numero di partite giocate tra i due, numero di vittorie di ciascuno, numero di volte in cui uno dei due ha abbandonato la partita. In altri termini, tale servizio è mantenuto ai fini di realizzare una profilazione degli utenti, per capire chi ha giocato con chi, quante volte e l'andamento di queste partite. Un possibile utilizzo di tutti i documenti *Friendship* memorizzati potrebbe essere l'integrazione con un database come Neo4J, dove ogni elemento è rappresentanto come un nodo di un grafo e le relazioni tra i vari elementi sono rappresentate come collegamenti tra i vari nodi, così da avere una mappa di tutte le relazioni tra gli utenti. Successivamente sulla base di queste informazioni, sarebbe possibile suggerire nuovi avversari agli utenti in virtù della loro storia o dei loro amici, modificando l'algoritmo di ricerca casuale di un avversario, così da preferirne uno piuttosto che un'altro.

9.7 Conclusioni

Con questo capitolo si conclude l'analisi delle varie implementazioni realizzate. In particolare, all'interno di quest'ultimo sono stati illustrati il meccanismo a supporto delle sfide tra due giocatori e le diverse modalità di ricerca di un possibile avversario. Tutti i possibili casi sono stati presi in considerazione, e ciò ha portato anche all'introduzione di alcuni bot come già visto. Infine, essendo un'architettura che deve offrire supporto in tempo reale, è stato realizzato un meccanismo di notifica tramite Firebase, ed in questo capitolo è stato spiegato in breve il suo utilizzo. Quanto trattato in questo capitolo costituisce la componente fondamentale che ha permesso di estendere Mak07 a gioco *multiplayer* continuando a garantire un *gameplay* soddisfacente e corretto.

Capitolo 10

Conclusioni

Con questo capitolo si conclude la trattazione di questo elaborato. La quale è partita inizialmente da un'analisi del contesto videoludico, ripercorrendo la storia e le tappe principali dell'evoluzione dei videogiochi, distinguendo le varie categorie, sino ad arrivare ad introdurre il *mobile gaming* e l'idea alla base di Mak07. Sono stati successivamente introdotti tutti gli obiettivi da raggiungere, e tutti i requisti che si voleva soddisfare durante lo sviluppo. E' stata spiegata l'architettura generale del sistema, anche se tale elaborato è sempre stato rivolto solo ad una porzione di questa che è l'infrastruttura lato server. Prima di potere sviluppare qualsiasi cosa, è stata necessaria una fase di studio e scelta degli strumenti da utilizzare, diversi capitoli di questo elaborato sono stati dedicati ad un'analisi teorica di questi. Infine gli ultimi capitoli sono dedicati ai vari dettagli implementativi. In conclusione, tutti gli obiettivi che ci si era posti sono stati raggiunti e i requisiti sempre soddisfatti, ed ulteriori ne sono stati aggiunti in corso d'opera, quando si è sentita l'esigenza di introdurre nuove funzionalità o fare evolvere il sistema secondo determinate caratteristiche.

10.1 Sviluppi futuri

Il contesto all'interno del quale si colloca tale lavoro di tesi, è tale da essere sempre soggetto ad evoluzioni, anche solo per l'aggiunta di nuove funzionalità a disposizione degli utenti, o miglioramenti vari a quelle già esistenti. Sempre in termini di funzionalità, si potrebbero in futuro aggiungere nuove modalità di gioco, come la già citata modalità a torneo. Al fine di realizzare questa, il sistema già dispone di tutti gli strumenti necessari, occorre solo definire un algoritmo robusto ed efficiente per la raccolta dei giocatori in attesa, prendendo in considerazione tutti i possibili casi di errore, per poi andare a creare in automatico le sfide tra i giocatori. Tutto ciò prevede inoltre l'aggiunta di un nuovo tipo di sfida, ma nessuna modifica alla macchina a stati già esistente, in quanto è stato già anche codificato uno stato di timeout. Ulteriore funzionalità è una migliore ricerca globale degli utenti tramite l'integrazione di un server di ricerca come ElasticSearch, pienamento supportato dal framework Spring. Essendo inoltre già presenti la tecnologie WebSocket e RabbitMQ,

non è esclusa la realizzazione di un sistema di messaggistica istantanea tra i diversi giocatori. Se il numero di utenti crescerà a sufficienza, sarà necessario gestire la mole di connessioni predisponendo più risorse e di conseguenza aumentando la scalabilità del sistema.

Bibliografia

[1] *ACID o BASE, questo è il problema*. URL: https://www.spindox.it/it/blog/acid-base-problema.

[2] Adams e Ernest. “Fundamentals of Game Design (3rd ed.)” In: (2015).

[3] Fabrizio Cipriani. *OAuth2 e i suoi predecessori: breve storia dell’autenticazione online*. URL: http://www.sullaprogrammazione.com/oauth-2-e-i-suoi-predecessori-breve-storia-dellautenticazione-online.html.

[4] *Crescita esponenziale per il mercato mobile gaming*. 2016. URL: http://www.gamification.it/mobile-gaming/crescita-esponenziale-mercato-giochi-smartphone/.

[5] DB-Engine. *DB-Engines Ranking*. 2015. URL: http://db-%20engines.com/en/ranking.

[6] Lovisa Johansson. *RabbitMQ for beginners - What is RabbitMQ?* URL: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html.

[7] Dario Marchetti. *Il futuro dei videogiochi è mobile*. URL: https://www.wired.it/gadget/videogiochi/2014/02/06/futuro-videogiochi-mobile/.

[8] Katherine Noyes. *Docker: A ’Shipping Container’ for Linux Code*. 2013. URL: https://www.linux.com/news/docker-shipping-container-linux-code.

[9] Erin De Pasquale. *Il mobile gaming e il futuro dei videogiochi*. URL: http://vulcanostatale.it/2015/03/il-mobile-gaming-e-il-futuro-dei-videogiochi/.

[10] Pramod J. Sadalage e Martin Fowler. “NoSQL Distilled A Brief Guide to the Emerging World of Polyglot Persistence,” in: (2012).

[11] *The OAuth 2.0 Authorization Framework*. URL: https://tools.ietf.org/html/rfc6749.

[12] SearchServer Virtualization. *What is hypervisor?*
URL: http://www.searchservervirtualization.techtarget.com/definition/hypervisor.

[13] VMware. *Virtualizzazione VMware*. URL: https://www.vmware.com/it/solutions/virtualization.html.