ISTITUTO TECNICO SETTORE TECNOLOGICO JFKennedy

ITST J.F. KENNEDY

PORDENONE

Specializzazione Informatica

Tesina Esame di Stato

Lupus in Tabula

Limiti della Prima Forma Normale

Candidato
Edoardo Morassutto
Quinta C IA

Anno scolastico 2015-2016

Sommario

Lupus in Tabula è un gioco di ruolo complesso nel quale i giocatori interpretano dei personaggi e devono raccontare una storia cercando di far vincere la propria squadra. Il gioco, spesso noto anche come Mafia[9], si basa sul riuscire a dedurre il ruolo degli altri giocatori per capire da chi è composta la propria squadra.

In questo documento è stato analizzato il problema di modellare i dati delle partite e di creare un server che permettesse a molti utenti di giocare in contemporanea. Durante lo sviluppo ci sono state delle difficoltà riguardo la memorizzazione di alcuni dati delle partite che sono stati in parte risolti.

Il documento è stato diviso in due parti, la prima è un'analisi completa dell'applicazione trattando nel dettaglio ogni aspetto sia progettuale che implementativo, la seconda invece, dopo un'introduzione alle forme normali, tratta delle problematiche incontrate e di alcune delle possibili soluzioni.

Durante lo sviluppo di un database *relazionale* è bene cercare di rispettare le *forme normali*. Una forma normale è un'insieme di regole che evitano che in una base di dati ci sia ridondanza o inconsistenza.

In questa trattazione si esaminerà uno scenario in cui un semplice database relazionale in prima forma normale non è sufficiente a modellare il problema e si rende necessaria una diversa implementazione.

L'applicazione è stata completata ed è disponibile su `https://lupus.serben.tk`, tutto il sorgente è pubblico ed accessibile su `https://github.com/lupus-dev/lupus`.

Indice

Parte I

Lupus in Tabula

Capitolo 1

Introduzione generale

1.1 Descrizione del gioco

Lupus in Tabula è un gioco di ruolo per, normalmente, 7 o più giocatori. Il narratore, che giocando senza computer, è un giocatore speciale che racconta una storia. La storia viene sviluppata da tutti i giocatori che, compiendo delle azioni, comportano dei cambiamenti nella vicenda, per esempio uccidendo dei personaggi o facendoli risorgere.

I giocatori sono divisi in due o più gruppi, esistono sempre le fazioni dei *lupi* e dei *contadini* alle quali può aggiungersi quella dei *criceti mannari* ed altre. Quando la partita finisce solo una di queste squadre vince.

Lo scopo della squadra dei *lupi* è quello di sbranare tutti gli altri giocatori, mentre quello dei *contadini* è di individuare ed uccidere tutti i *lupi*.

La partita si alterna di giorni e di notti, durante ogni notte tutta la squadra dei lupi deve scegliere un bersaglio (solitamente un contadino) che quella notte verrà sbranato. Durante il giorno, invece, ogni giocatore deve votare chi, secondo lui, mandare al rogo. Questo è l'unico modo per i contadini di uccidere i vari lupi che si aggirano nel villaggio. A questa votazione partecipano anche i lupi che cercheranno ovviamente di non farsi votare.

Per rendere il gioco più dinamico vengono inseriti tra i giocatori alcuni ruoli speciali. Per esempio un *veggente* è un contadino che, durante la notte, può scegliere un giocatore vivo e, tramite consultazione della palla di cristallo, sapere se ha un'anima buona oppure cattiva. Ogni ruolo infatti, oltre ad essere di una certa squadra, ha anche delle caratteristiche extra come il *mana*, che può essere buono oppure cattivo. Normalmente (ma non necessariamente) i giocatori nella squadra dei lupi hanno mana cattivo mentre quelli nella squadra dei contadini ce l'hanno buono.

Ogni giocatore, in generale, conosce solo il proprio ruolo e deve dedurre quello degli altri giocatori; ci possono essere dei casi in cui qualche giocatore conosce il ruolo esatto di qualcun altro. Per esempio ogni lupo conosce esattamente quali sono i suoi compagni di squadra.

Quanto un giocatore muore in una partita normale è il narratore ad annunciarlo, in questa implementazione è presente un giornale nel quale viene scritto il nome dei giocatori che sono morti.

Affinché una partita termini correttamente deve verificarsi almeno una delle seguenti condizioni:

- Tutti i giocatori sono morti
- Almeno una fazione ha vinto
  - Lupi: Il numero di lupi è maggiore o uguale al numero di contadini

– Contadini: Il numero di lupi è zero

1.2 Ruoli

Ad ogni giocatore viene assegnato dal sistema uno ed un solo ruolo, quindi il giocatore saprà anche la sua fazione e il suo mana.

Ogni ruolo, oltre al nome ha anche alcune proprietà aggiuntive:

- Mana: indica se il giocatore agirà come malintenzionato oppure come una brava persona
- Nome identificativo: nome unico che identifica il ruolo
- Priorità: serve per stabilire un ordine nell'esecuzione delle azioni
- Debug: alcuni ruoli sono accessibili solo dagli sviluppatori e sono marcati come *debug*
- Enabled: È possibile disattivare alcuni ruoli tramite questa proprietà
- Chat: alcuni ruoli hanno dei canali di comunicazione dedicati
- Gen*: parametri aggiuntivi per la generazione automatica dei ruoli, come numero minimo e massimo di occorrenze del ruolo, probabilità, ecc...

1.2.1 Ruoli comuni

I ruoli che normalmente vengono usati in una partita di Lupus in Tabula sono:

ID	Lupo	Priorità	100
Nome	Lupo	Debug	no
Mana	Cattivo	Enabled	sì
Squadra	Lupi	Chat	Lupi

Durante la notte i *lupi* votano chi eliminare, se almeno il 50%+1 dei *lupi* vivi votano la stessa persona, questa è una candidata a morire

ID	Guardia	Priorità	10000
Nome	Guardia	Debug	no
Mana	Buono	Enabled	si
Squadra	Contadini	Chat	

Durante la notte la *guardia* può scegliere di proteggere una persona, se i *lupi* quella notte decidessero di ucciderla, questa non muore

ID	Medium	Priorità	150
Nome	Medium	Debug	no
Mana	Buono	Enabled	si
Squadra	Contadini	Chat	

Il *medium* durante la notte può scegliere di guardare un giocatore morto, lui saprà se quel giocatore aveva un mana buono o cattivo

ID	Paparazzo	Priorità	1000
Nome	Paparazzo	Debug	no
Mana	Buono	Enabled	si
Squadra	Contadini	Chat	

Il *paparazzo* durante la notte sceglie una persona da pedinare, vengono riportati sul giornale della mattina seguente tutti i giocatori che hanno visitato il personaggio *paparazzato*

ID	Criceto mannaro	Priorità	10000
Nome	Criceto mannaro	Debug	no
Mana	Cattivo	Enabled	si
Squadra	Criceti	Chat	

É un giocatore normale, senza poteri speciali. Se la partita termina e lui è ancora vivo allora vince solo lui e non la sua fazione

ID	Assassino	Priorità	10
Nome	Assassino	Debug	no
Mana	Cattivo	Enabled	si
Squadra	Contadini	Chat	

L'*assassino* una sola volta nella partita può scegliere una persona e ucciderla

ID	Massone	Priorità	10000
Nome	Massone	Debug	no
Mana	Buono	Enabled	si
Squadra	Contadini	Chat	Massoni

I *massoni* non hanno poteri però hanno una chat dedicata e quindi si conoscono tra loro

ID	Contadino	Priorità	10000
Nome	Contadino	Debug	no
Mana	Buono	Enabled	si
Squadra	Contadini	Chat	

I *contadini* non hanno poteri...

ID	Pastore	Priorità	50
Nome	Pastore	Debug	sì
Mana	Buono	Enabled	si
Squadra	Contadini	Chat	

I *pastori* possono scegliere di sacrificare delle pecore per salvare dei giocatori dalle grinfie dei lupi

ID	Sindaco	Priorità	10000
Nome	Sindaco	Debug	no
Mana	Buono	Enabled	si
Squadra	Contadini	Chat	

Il *sindaco* è un contadino che non può essere messo al rogo nella votazione diurna

Capitolo 2

Analisi

2.1 Analisi del problema

È necessario sviluppare un sistema robusto che gestisce gli utenti e le partite di Lupus in Tabula.

Il sistema deve essere in grado di sopportare un numero crescente di giocatori e un numero molto elevato di partite, deve essere in grado di scalare ottimalmente con le richieste. La base di dati deve essere sviluppata in modo da garantire un funzionamento efficiente e una robustezza dei dati.

I dati identificativi degli utenti devono essere conservati in modo sicuro, è necessario proteggere le credenziali di accesso tramite moderni sistemi di *hashing*.

È necessario fare molta attenzione ai privilegi degli utenti, proteggendo le risorse che non sono accessibili. Per esempio è opportuno evitare che gli utenti possano vedere, entrare o modificare le partite alle quali gli è stato negato il permesso.

Le chiavi interne del database non sono visibili agli utenti, delle chiavi alternative non numeriche sono visualizzate al loro posto. Per esempio le partite non vengono identificate (lato utente) da un intero progressivo ma da un *nome breve*, una sequenza di caratteri che rispetta la seguente `regex`: `[a-zA-Z][a-zA-Z0-9]{0,9}`. Deve essere lungo da 1 a 10 caratteri `ASCII`, il primo carattere deve essere una lettera e deve essere unico nel suo contesto.

2.2 Struttura dei componenti

Ogni utente registrato nel sistema può giocare alle partite create dagli altri giocatori oppure può crearne di nuove. Per creare delle partite l'utente deve creare delle stanze, dei raggruppamenti di zero, una o più partite.

Una stanza appartiene ad uno ed un solo utente che ha poteri amministrativi su questa, solo lui può creare una nuova partita in quella stanza. Ogni stanza viene identificata da un *nome breve* univoco e non modificabile ed ha una breve descrizione. In una stanza ci può essere al più una parita in corso.

Quando un utente possiede una stanza libera (senza partite in corso), può decidere di creare una nuova partita, questa viene identificata da un *nome breve* univoco all'interno della stanza ma non globalmente e non modificabile. La partita ha anche una breve descrizione.

Gli altri utenti possono entrare nella partita solo se hanno i permessi per farlo, ogni partita infatti condivide i permessi della stanza a cui appartiene, per esempio se la stanza è protetta da delle liste di accesso (`ACL`), solo gli utenti specificati possono accedervi.

Una partita è composta da una serie di giocatori, degli utenti a cui è stato assegnato un ruolo (eventualmente non definito). Appena un giocatore entra nella partita (e questa non è iniziata) gli viene assegnato un ruolo non definito (`unknown`). Appena il numero di giocatori raggiunge il valore specificato dall'amministratore vengono generati i ruoli esatti dei giocatori. I ruoli possono essere generati in due modi: manualmente secondo precise impostazioni dell'amministratore oppure automaticamente specificando solo quali ruoli usare.

Ogni utente può giocare anche a diverse partite in contemporanea, in accordo con i limiti imposti dal suo livello. Ogni utente infatti ha un livello che limita le sue possibilità di gioco, come il numero di partite contemporanee, il numero di stanze pubbliche e private.

Il livello di un giocatore è sempre crescente, si può venire declassati solo dall'amministratore del server. Il livello viene stabilito tramite un valore detto *karma* che rappresenta un'indicazione del tempo di gioco dell'utente. Si guadagnano punti karma giocando partite, vincendo partite o invitando altri giocatori. Quando il karma raggiunge un valore sufficientemente alto viene aumentato il livello dell'utente.

Per poter usare le funzionalità in beta è necessario possedere un livello sufficientemente alto.

Il livello di un utente è visibile nella pagina dell'utente e in ogni partita viene evidenziato. I possibili livelli sono:

	Nome	**Karma**	**Partite parallele**	**Stanze**	**Stanze private**	**BETA**
1	Neofita	0	3	1	0	no
2	Principiante	25	5	1	0	no
3	Gamer	50	5	3	1	no
4	Esperto	100	5	5	3	no
5	Maestro	200	7	5	5	no
6	ProGamer	500	10	10	5	no
7	Stratega	2000	15	10	10	si
8	Generale	5000	50	100	10	si
9	Guru	10000	100	100	100	si
10	GameMaster	∞	1000	1000	1000	si

Tabella 2.1: Livelli degli utenti

Giocando è anche possibile sbloccare degli obiettivi che arricchiscono il profilo del giocatore, un badge verrà infatti visualizzato nella pagina dell'utente appena compie un'azione memorabile. Il valore di *difficoltà* serve per ordinare gli obiettivi, non necessariamente indica la difficoltà.

Codice	**Nome**	**Descrizione**	**Difficoltà**
AtLeast5Games	Appena iniziato	Gioca almeno 5 partite	10
AtLeast20Games	Iniziamo a ragionare...	Gioca almeno 20 partite	11
AtLeast50Games	Esperto del mestiere	Gioca almeno 50 partite	12
AtLeast100Games	Saggio del villaggio	Gioca almeno 100 partite	13
AtLeast5Wins	Primi successi	Vinci almeno 5 partite	20
AtLeast20Wins	Hai capito le regole	Vinci almeno 20 partite	21
AtLeast50Wins	Pericolo pubblico	Vinci almeno 50 partite	22
AtLeast100Wins	Stratega professionista	Vinci almeno 100 partite	23

Tabella 2.2: Obiettivi sbloccabili

2.3 Progettazione della base di dati

Come analizzato nella Parte II [5.3] il database è stato diviso in due parti, una relazionale e una `NoSQL`. In merito alla parte relazionale è qui proposto uno schema Entità-Relazione.

Alcuni attributi sono stati mantenuti dal passaggio dalla prima versione del Database (senza NoSQL) a quella più recente. In particolare tutti gli attributi che non rispettavano la Prima Forma Normale sono stati lasciati ed ora sono facoltativi (se lasciati a `NULL` viene usata la versione in NoSQL).

Tutto ciò sia per retrocompatibilità che per garantire il funzionamento della piattaforma anche in eventuali servizi di hosting che non supportano NoSQL.

2.3.1 Schema concettuale

Nello schema in figura 2.1 sono riportate tutte le entità e le relazioni che compongono la base di dati. In ogni entità sono anche elencati tutti gli attributi che la compongono, comprese le chiavi primarie ed eventuali vincoli di unicità.

I campi marcati con una `x` sono i campi interessati dalla ristrutturazione tramite NoSQL. Fare riferimento alla descrizione presente nella Parte II, in particolare [5.3].

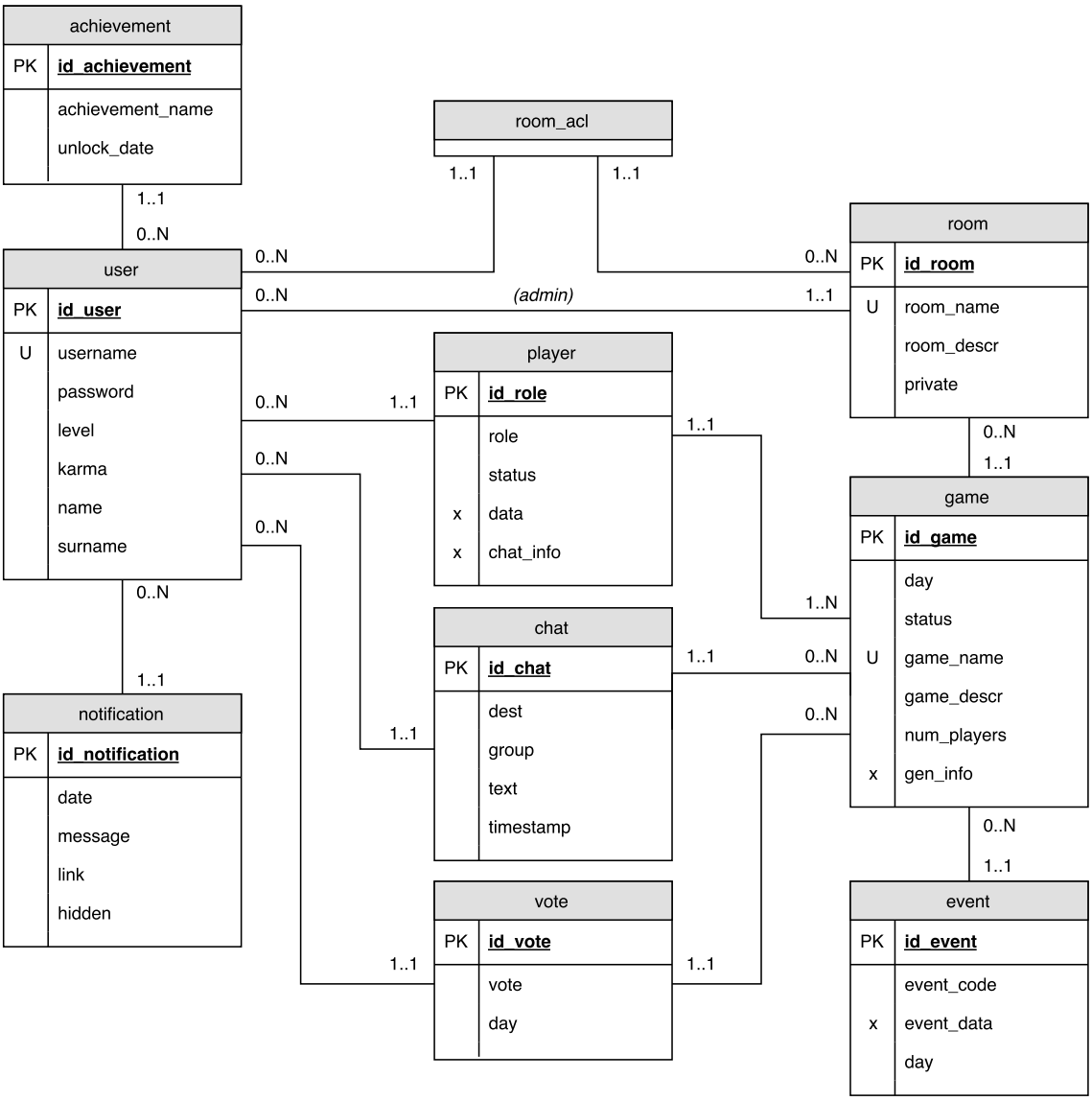

Figura 2.1: Diagramma Entità-Relazione della base di dati

2.3.2 Schema logico

Vengono qui riportate, in modo sintetico le definizioni di tutte le tabelle che compongono la base di dati. Le chiavi primarie sono sottolineate, le chiavi esterne sono in *corsivo* con il relativo riferimento; i campi ristrutturati con NoSQL hanno un asterisco (*).

- `user(id_user, username, password, level, karma, name, surname)`
- `room(id_room, id_admin`$^{\texttt{user.id\_user}}$`, room_name, room_descr, private)`
- `game(id_game, id_room`$^{\texttt{room.id\_room}}$`, day, status, game_name, game_descr, num_players, gen_info`*`)`

- player(id_role, *id_game*[game.id_game], *id_user*[user.id_user], role, status, data*, chat_info*)
- chat(id_chat, *id_game*[game.id_game], *id_user_from*[user.id_user], dest, group, text, timestamp)
- vote(id_vote, *id_game*[game.id_game], *id_user*[user.id_user], vote, day)
- event(id_event, *id_game*[game.id_game], event_code, event_data*, day)
- notification(id_notification, *id_user*[user.id_user], date, message, link, hidden)
- achievement(id_achievement, *id_user*[user.id_user], achievement_name, unlock_date)
- room_acl(*id_room*[room.id_room], *id_user*[user.id_user])

2.3.3 Descrizione delle entità

Vengono qui riportate le descrizioni dettagliate di tutte le tabelle presenti nella base di dati. Anche gli attributi sono elencati ed analizzati, con particolare riferimento alle varie caratteristiche di questi. Nella varie tabelle che seguono vengono usate delle abbreviazioni per snellire la trattazione. In particolare riguardo i vincoli le varie sigle significano:

- **P**: chiave primaria
- **E**: chiave esterna
- **U**: vincolo di unicità
- **X**: attributo ristrutturato nel NoSQL
- **N**: attributo opzionale (`NULL`)

È possibile trovare anche la combinazione di queste lettere.

Nella colonna relativa al dominio viene indicato il tipo di dato dell'attributo, se maiuscolo è da riferirsi ad uno dei tipi predefiniti. In caso di chiavi esterne in questa colonna viene indicato l'attributo di "chiave primaria" collegato.

Tabella `user`

Questa tabella contiene le informazioni personali di ogni utente, compreso username e password. Ogni utente possiede un livello numerico intero e un livello di karma, sempre intero. Nella tabella sono anche memorizzati il nome e il cognome dell'utente.

Per la gestione della password è stato adottato un metodo particolare per garantire il funzionamento anche su sistemi datati o privi della libreria `crypto`.

Se il server PHP supporta le funzioni `password_hash` e `password_verify` allora vengono usate quelle, altrimenti viene usata la funzione non sicura SHA-1.

La funzione `password_hash` cifra la password utilizzando l'algoritmo `CRYPT_BLOWFISH` che viene ripetuto 10 volte e viene utilizzato un *salt* crittograficamente sicuro. Questo algoritmo è sufficientemente lento da evitare attacchi di tipo *brute-force* e l'utilizzo del salt evita che le password vengano ricavate facilmente attraverso tabelle di lookup.

La funzione `password_verify` effettua l'hash della password inviata in chiaro utilizzando gli stessi parametri di `password_hash` ed effettua un confronto tra le due stringhe non vulnerabile a degli attacchi *timed-based*.

Attributo	Vincoli	Dominio	Descrizione
id_user	P	INT	Identificativo dell'utente
username	U	VARCHAR(10)	Username dell'utente
password		VARCHAR(255)	Hash della password dell'utente
level		INT	Numero del livello dell'utente, fare riferimento alla tabella [2.1]
karma		INT	Punti karma dell'utente
name		VARCHAR(50)	Nome dell'utente
surname		VARCHAR(50)	Cognome dell'utente

Tabella 2.3: Attributi della tabella `user`

Tabella `room`

Ogni stanza ha un identificativo univoco `id_room`, nascosto al pubblico che identifica la stanza all'interno del database. Al pubblico la stanza è identificata con un nome `room_name` il quale è unico. Ogni stanza ha anche una descrizione `room_descr` che rappresenta il titolo della stanza.

Ogni stanza contiene l'identificativo dell'utente amministratore della stanza `id_admin`.

Il campo `private` indica se la stanza è pubblica o privata, secondo le specifiche in [2.4.5].

Attributo	Vincoli	Dominio	Descrizione
id_room	P	INT	Identificativo della stanza
id_admin	E	user.id_user	Identificativo dell'utente a cui appartiene la stanza
room_name	U	VARCHAR(10)	Nome identificativo della stanza
room_descr		VARCHAR(50)	Descrizione della stanza
private		INT	Grado di visibilità della stanza

Tabella 2.4: Attributi della tabella `room`

Tabella `game`

Ogni partita è identificata all'interno del database con un identificativo unico `id_game`. Ogni partita è identificata all'esterno con un *nome breve* unico all'interno della stessa stanza (`game_name`), inoltre ha una descrizione che rappresenta il titolo della partita `game_descr`.

Ogni partita è contenuta all'interno della stanza con id `id_room`. I campi `day` e `status` sono informazioni utili della partita, l'istante temporale del gioco e lo stato della partita.

Il numero di giocatori è memorizzato nel campo `num_players` per eseguire velocemente alcune query. Le informazioni per generare la partita vengono memorizzate nel database NoSQL, in precedenza erano memorizzate come JSON nel campo `gen_info`.

Attributo	Vincoli	Dominio	Descrizione
id_game	P	INT	Identificativo della partita
id_room	E	room.id_room	Identificativo della stanza associata
day		INT	Giorno in cui si trova la partita. Fare riferimento a 2.4.1
status		INT	Stato in cui si trova la partita. Fare riferimento a 2.4.2
game_name	U	VARCHAR(10)	Nome breve della partita. Deve essere unico all'interno della stanza
game_descr		VARCHAR(50)	Descrizione della partita
num_players		INT	Numero di giocatori nella partita
gen_info	XN	VARCHAR(500)	Informazioni per generare la partita

Tabella 2.5: Attributi della tabella `game`

Tabella `player`

In questa tabella sono memorizzati i ruoli dei giocatori interni ad una partita.

Ogni riga è identificata da un campo `id_role` non pubblico. Ogni ruolo è interno alla partita `id_game` e relativo all'utente `id_user`. Il ruolo dell'utente è memorizzato nella stringa `role` la quale identifica un ruolo nella cartella dei ruoli. Per approfondire il significato di questa stringa fare riferimento a [3.5].

Il campo `status` indica lo stato dell'utente (vivo/morto/espulso). Alcuni ruoli potrebbero richiedere di memorizzare delle informazioni aggiuntive, il campo `data` (che viene memorizzato nel NoSQL) contiene dei dati salvati da un ruolo. Un giocatore ha anche memorizzato alcune informazioni sulle sue chat, per sapere quali sono i messaggi che non ha ancora letto.

Attributo	Vincoli	Dominio	Descrizione
id_role	P	INT	Identificativo del giocatore
id_game	E	game.id_game	Identificativo della partita associata
id_user	E	user.id_user	Identificativo dell'utente associato
role		VARCHAR(50)	Nome identificativo del ruolo corrispondente, fare riferimento a [3.5].
status		INT	Stato dell'utente nella partita, fare riferimento a [2.4.6]
data	XN	VARCHAR(500)	Dati aggiuntivi del ruolo
chat_info	XN	VARCHAR(500)	Informazioni sulle chat del giocatore

Tabella 2.6: Attributi della tabella `player`

Tabella `chat`

In questa tabella vengono memorizzati i messaggi delle varie chat. Ogni messaggio è identificato all'interno del database da `id_chat` ed è relativo ad una singola partita (`id_game`).

Il mettente del messaggio è `id_user_from` cioè l'identificativo dell'utente che ha inviato il messaggio. Il destinatario `dest` è un numero intero che può rappresentare vari identificativi, in base a `group` il destinatario può essere un utente, il pubblico, una chat privata, ecc...

Il testo del messaggio è `text` il quale non può essere più lungo di 200 caratteri e viene eseguito l'*escape* sia per il database che per il JavaScript.

Attributo	Vincoli	Dominio	Descrizione
id_chat	P	INT	Identificativo del messaggio
id_game	E	game.id_game	Partita associata
id_user_from	E	user.id_user	Utente mittente del messaggio
dest		INT	Destinatario del messaggio
group		INT	Gruppo di destinazione del messaggio
text		VARCHAR(200)	Testo del messaggio
timestamp		TIMESTAMP	Data e ora di invio del messaggio

Tabella 2.7: Attributi della tabella `chat`

Tabella `vote`

In questa tabella vengono memorizzati i voti degli utenti. Ogni voto è identificato tramite il campo `id_vote`, il quale rimane nascosto all'esterno del database. Ogni voto è specifico della partita `id_game` nel momento `day` e appartiene all'utente `id_user`.

Il voto è `vote` il quale può essere l'identificativo di un utente, un voto *nullo* o altro.

Attributo	Vincoli	Dominio	Descrizione
id_vote	P	INT	Identificativo della votazione
id_game	E	game.id_game	Identificativo della partita associata
id_user	E	user.id_user	Identificativo del giocatore associato
vote		INT	Voto dell'utente
day		INT	Giorno della partita in cui vale la votazione

Tabella 2.8: Attributi della tabella `vote`

Tabella `event`

In questa tabella vengono memorizzati gli eventi delle partite. Ogni evento è identificato dal campo `id_event` e appartiene alla partita `id_game` ed è specifico del giorno `day` secondo le specifiche in [2.4.1].

Il tipo di evento che si è verificato è memorizzato in `event_code` secondo i codici descritti in [2.4.4]. Il campo dei dati dell'evento è memorizzato nel database NoSQL e in precedenza era salvato come JSON in `event_data`.

Attributo	Vincoli	Dominio	Descrizione
id_event	P	INT	Identificativo dell'evento
id_game	E	game.id_event	Partita associata all'evento
event_code		INT	Codice dell'evento, fare riferimento a [2.4.4]
event_data	XN	VARCHAR(500)	Dati dell'evento
day		INT	Giorno della partita

Tabella 2.9: Attributi della tabella `event`

Tabella `notification`

In questa tabella sono memorizzate tutte le notifiche di un utente. Le notifiche cancellate non vengono rimosse ma semplicemente marcate come lette. Ogni notifica viene identificata dal campo `id_notification` ed appartiene all'utente `id_user`.

Il testo del messaggio si trova nel campo `message` ed avrà un collegamento `link`, il quale è il percorso assoluto senza `basePath` (vedi capitolo [3.3]). La notifica viene nascosta tramite il parametro `hidden`.

Nel testo della notifica è possibile inserire dell'HTML per aggiungere dello stile.

Attributo	Vincoli	Dominio	Descrizione
id_notification	P	INT	Identificativo della notifica
id_user	E	user.id_user	Utente a cui appartiene la notifica
date		TIMESTAMP	Data di generazione della notifica
message		VARCHAR(200)	Testo della notifica
link		VARCHAR(200)	Collegamento alla notifica
hidden		TINYINT	Booleano che indica se la notifica è stata nascosta

Tabella 2.10: Attributi della tabella `notification`

Tabella `achievement`

Gli obiettivi sbloccati dagli utenti sono memorizzati in questa tabella. Ogni obiettivo sbloccato viene identificato dal campo `id_achievement`. Per riconoscere l'obiettivo viene usato il campo `achievement_name` che contiene il nome identificativo dell'obiettivo, come descritto nella tabella [2.2].

Attributo	Vincoli	Dominio	Descrizione
id_achievement	P	INT	Identificativo dell'obiettivo sbloccato
id_user	E	user.id_user	Utente che ha sbloccato l'obiettivo
achievement_name		VARCHAR(50)	Nome identificativo dell'obiettivo
unlock_date		TIMESTAMP	Data di sblocco dell'obiettivo

Tabella 2.11: Attributi della tabella `achievement`

Tabella `room_acl`

Le stanze che sono state impostate private con controllo degli accessi utilizzano questa tabella per sapere quali sono gli utenti autorizzati. La tabella contiene due campi che formano la chiave primaria: `id_room` e `id_user`.

Attributo	Vincoli	Dominio	Descrizione
id_room	PE	room.id_room	Identificativo della stanza
id_user	PE	user.id_user	Identificativo dell'utente

Tabella 2.12: Attributi della tabella `room_acl`

2.4 Codici utilizzati

Nel gioco, per alleggerire la base di dati e per snellire il tutto sono state usate delle convenzioni riguardo alcuni codici. Per esempio per evitare di memorizzare un'intera stringa come `Giorno 4` è stato scelto di memorizzare l'intero 6, mentre per la `Notte 1` il valore 1.

Questi codici sono qui riportati e descritti e sono modificabili sono andando ad intervenire nella apposite classi dedicate alla sola memorizzazione di queste costanti. Questi valori non sono stati usati semplicemente nel codice ma viene sempre fatto riferimento alle costanti specificate.

2.4.1 Tempo del gioco

Come appena preannunciato il tempo di una partita viene semplicemente memorizzato come numero intero secondo la seguente convenzione:

$$day \begin{cases} \text{se } day = 0 \Rightarrow \text{arrivo al villaggio} \\ \text{se } day \text{ è } pari \Rightarrow \text{giorno } \dfrac{day}{2} + 1 \\ \text{se } day \text{ è } dispari \Rightarrow \text{notte } \dfrac{day}{2} + 1 \end{cases}$$

Il valore di *day* viene memorizzato nella tabella game in quanto si tratta del giorno della partita. Sono qui riportati degli esempi dei valori che può assumere *day*:

day	**valore**
0	Arrivo al villaggio
1	Notte 1
2	Giorno 2
3	Notte 2
4	Giorno 3
5	Notte 3
6	Giorno 4
7	Notte 4

2.4.2 Stato della partita

Ogni partita deve trovarsi in uno stato preciso che indica le condizioni del gioco. In particolare è necessario sapere se la partita è già iniziata, se è finita, se ha vinto una certa squadra, se è stata interrotta ecc...

Per rendere più chiaro il valore di questi codici i vari stati delle partite sono stati divisi in alcune fasi: Pre-Partita, In-Partita, Terminata-OK, Terminata-Errore.

Fase Pre-Partita

Questa fase è presente solo prima dell'avvio della partita e serve per salvare ed aggiustare i dettagli della partita, come il numero di giocatori, di ruoli, la descrizione, ecc...

Questa fase è identificata da codici nell'intervallo:

$$0 \leq x < 100$$

I codici riconosciuti sono:

- **0**: `Setup` Impostazione della partita

Fase In-Partita

Questa fase è presente poco prima dell'inizio della partita e durante tutto il corso della partita.

Viene identificata da codici nell'intervallo:

$$100 \leq x < 200$$

I codici riconosciuti sono:

- **100**: `NotStarted` La partita è attiva e i giocatori possono iniziare ad entrare
- **101**: `Running` La partita è in corso e i giocatori non possono più entrare

Fase terminata in modo corretto

Questa fase si verifica quando la partita termina in modo corretto e viene scelta una squadra vincitrice.

I codici che la identificano sono compresi nell'intervallo:

$$200 \leq x < 300$$

I codici riconosciuti sono:

- **200 + y**: `Win`y La partita è terminata e ha vinto la squadra y ($0 \leq y < 99$)
- **299**: `DeadWin` La partita è terminata perché tutti i giocatori sono morti

Fase terminata in modo inaspettato

Questa fase si verifica quando la partita viene interrotta prima della sua normale fine. In questo caso non viene designato alcun vincitore.

I codici che identificano questa fase sono compresi dall'intervallo:

$$300 \leq x < 400$$

I codici riconosciuti sono:

- **300**: `TermByAdmin` L'amministratore della stanza ha fatto terminare la partita
- **301**: `TermBySolitude` Un numero eccessivo di giocatori hanno abbandonato la partita
- **302**: `TermByVote` É stato raggiunto il *quorum* per terminare la partita
- **303**: `TermByBug` Un errore interno del server ha fatto terminare la partita per preservare l'integrità del server
- **304**: `TermByGameMaster` Il GameMaster ha deciso di terminare la partita

2.4.3 Codici di risposta delle API

Le funzioni delle API ritorneranno un codice numerico identificativo del messaggio di errore/successo che la relativa funzione ha avuto.

I codici sono divisi in gruppi di decine: le unità sono un numero progressivo che identifica la sezione caratterizzata dalle altre cifre del codice.

#	Abbreviazione	Descrizione
1	`NotLoggedIn`	L'utente non è connesso e non può accedere a questa risorsa
2	`FatalError`	Un errore grave interno al server può aver compromesso la partita
3	`AccessDenied`	L'utente non ha i permessi per accedere alla risorsa richiesta
4	`NotFound`	La risorsa richiesta non è stata trovata
5	`MissingParameter`	Non è stato specificato un parametro richiesto
6	`MalformedParameter`	Uno dei parametri non ha un formato corretto o è invalido
7	`Done`	La richiesta è stata completata con successo
8	`Fail`	La richiesta non è stata completata a causa di un errore
9	`Found`	La risorsa è stata trovata
101	`LoginAlreadyDone`	Utente già connesso
130	`JoinDoneGameStarted`	L'utente è entrato nella partita e questa è iniziata
131	`JoinDoneGameWaiting`	L'utente è entrato nella partita ma questa è in attesa
132	`JoinFailedAlreadyIn`	L'utente era già entrato nella partita
133	`JoinFailedGameClose`	Non sono permessi ingressi
134	`JoinFailedGameFull`	L'utente non è entrato nella partita perché è piena
135	`JoinFailedGamesEnded`	La partità era finita, il giocatore non è entrato
143	`NewGameAlreadyRunning`	Una partita nella stanza è ancora in corso
144	`NewGameAlreadyExists`	Esiste già una partita con questo nome nella stanza
152	`StartNotInSetup`	La partita non è in fase di setup
153	`StartWithoutLupus`	La partita non contiene lupi
160	`VoteDoneNextDay`	La votazione è stata effettuata e la partita è avanzata
161	`VoteDoneWaiting`	La votazione è stata effettuata e la partita è in attesa
162	`VoteDoneGameEnd`	La votazione è stata effettuata e la partita è terminata
164	`VoteGameNotRunning`	La partita non è in corso
166	`VoteNotValid`	Il voto dell'utente non è valido
167	`VoteNotNeeded`	L'utente non deve votare o ha già votato
172	`NewRoomRoomsEnded`	L'utente ha finito il numero di stanze che può creare
173	`NewRoomPrivateRoomsEnded`	L'utente ha finito il numero di stanze private che può creare
174	`NewRoomAlreadyExists`	Esiste già una stanza con questo nome
200	`CheckRoomNameAccepted`	Il nome della stanza è corretto ed accettabile
203	`CheckRoomNameExisting`	Il nome della stanza è già stato usato
210	`CheckRoomDescrAccepted`	La descrizione della stanza è accettabile
220	`CheckGameNameAccepted`	Il nome della partita è corretto ed accettabile
224	`CheckRoomNameExisting`	Il nome della partita è già stato usato in questa stanza
230	`CheckGameDescrAccepted`	La descrizione della partita è accettabile
242	`SetupNotInSetup`	La partita non è in fase di setup
252	`SignupAlreadyExists`	Non è possibile registrare un utente con questo `username`
263	`ChatUserNotInGame`	L'utente non è nella partita
264	`ChatInvalidUser`	Il destinatario del messaggio non è valido
270	`GameTerminated`	La partita è stata terminata dall'amministratore
272	`GameTermNotRunning`	La partita non era in corso
282	`PlayerKickNotValidState`	La partita non è in uno stato valido per espellere giocatori
290	`ACLAlreadyPresent`	La regola ACL era già presente nell'elenco
291	`ACLCannotRemoveAdmin`	Non è possibile rimuovere l'admin della stanza dall'ACL

Non tutte le decine sono state usate in quanto, durante lo sviluppo, alcuni codici sono stati rimossi.

2.4.4 Codici degli eventi

Gli eventi che si verificano nella partita vengono memorizzati per essere visualizzati nel giornale del villaggio. Questi eventi vengono distinti in base ad un codice numerico che identifica il tipo di

evento. Ad ogni tipologia di evento corrispondono alcuni attributi specifici, come per esempio lo username del giocatore ucciso o dell'assassino o una lista degli username dei giocatori visti.

- **0**: `GameStart` La partita è appena iniziata

  Dati registrati dell'evento:
  `players` Vettore con gli `username` dei giocatori nella partita
  `start` Timestamp dell'ora dell'inizio della partita

- **1**: `Death` Un giocatore è stato ucciso o è stato trovato morto

  Dati registrati dell'evento:
  `dead` Username del giocatore morto
  `cause` Causa della morte
  `actor` Causante della morte (killer)

- **2**: `MediumAction` Un medium ha guardato un morto

  Dati registrati dell'evento:
  `medium` Username del medium
  `seen` Username del giocatore guardato
  `mana` Mana del giocatore guardato

- **3**: `VeggenteAction` Un veggente ha guardato un vivo

  Dati registrati dell'evento:
  `medium` Username del veggente
  `seen` Username del giocatore guardato
  `mana` Mana del giocatore guardato

- **4**: `PaparazzoAction` Un paparazzo ha fotografato un giocatore

  Dati registrati dell'evento:
  `paparazzo` Username del paparazzo
  `seen` Username del giocatore guardato
  `visitors` Lista dei giocatori che hanno fatto visita al giocatore

- **5**: `BecchinoAction` Un becchino ha resuscitato un giocatore

  Dati registrati dell'evento:
  `becchino` Username del becchino
  `dead` Username del giocatore resuscitato

- **6**: `PlayerKicked` Un giocatore è stato espulso dalla partita

  Dati registrati dell'evento:
  `kicked` Username del giocatore espulso

2.4.5 Visibilità della stanza

Una stanza può essere resa privata dall'amministratore. Esistono 3 livelli di visibilità di una stanza:

0. **Pubblica**: la stanza è accessibile a chiunque ed è elencata nell'elenco delle partite
1. **Solo link**: la stanza è accessibile a chiunque possegga il link, non viene elencata
2. **Privata**: solo gli utenti specificati dall'amministratore della stanza possono vederla e solo loro la vedono negli elenchi

2.4.6 Stato dei giocatori

Un giocatore in una partita può trovarsi in uno di questi stati:

0. Vivo: il giocatore è ancora vivo
1. Morto: in questa partita il giocatore è morto
2. Kicked: il giocatore è stato espulso dalla partita

Capitolo 3

Implementazione

3.1 Tecnologie utilizzate

Essendo questa un'applicazione scritta utilizzando il linguaggio `PHP` è necessario che sia installato un server web per la ricezione delle richieste `HTTP`. Per semplificare lo sviluppo di alcune parti del programma sono state usate delle tecnologie intermedie. Per esempio per lo sviluppo della parte grafica, quindi il codice `CSS`, è stato usato il linguaggio `SASS`, un'estensione del CSS che aggiunge delle utili funzionalità come la possibilità di usare variabili o ereditarietà delle classi di stile.

I fogli di stile hanno quindi bisogno di essere compilati nei comuni file `CSS` attraverso il programma `scss`, che quindi è una dipendenza per lo sviluppo. Nel repository del progetto vengono sempre mantenuti anche i file `.css` già compilati per alleggerire la fare di installazione.

A causa di alcune funzioni utilizzate la versione minima richiesta di PHP è la 5.4.0, quella consigliata è la 5.5.0. Alcune funzionalità come l'hashing sicuro della password richiedono che il server PHP supporti la funzione `crypt` che potrebbe essere necessario abilitare manualmente.

Utilizzando PHP in versione minore della 5.5.0 si rende impossibile l'utilizzo di queste funzionalità:

- L'hashing sicuro delle password attraverso `password_hash` e `password_verify`

È possibile utilizzare anche PHP 7 se il sistema lo supporta.

Come web server è possibile utilizzare `apache2` oppure `nginx` con `php-fpm`. All'interno del repository sono presenti alcuni esempi di configurazione per i due web-server. È possibile utilizzare anche un qualunque altro server web a patto che supporti l'interazione con `PHP` (che attraverso `cgi`) e il *rewrite* degli url. In `apache2` potrebbe essere necessario abilitare il relativo modulo (`mod_rewrite`) ed utilizzarlo all'interno della configurazione (`apache2.conf` o relativo file in `sites-available`) oppure tramite il file `.htaccess` presente come esempio. `nginx` non necessità di moduli esterni per il rewrite degli indirizzi ma è necessario installare `php-fpm` a parte per usare il `PHP`.

Come `DBMS` è stato usato `MySQL` attraverso la libreria `PDO` di PHP. Potrebbe essere necessario abilitare questa estensione all'interno del file `php.ini`. Teoricamente dovrebbe essere possibile usare qualunque DBMS supportato da PDO ma a causa delle diversità tra i vari dialietti SQL, l'unico suppportato è MySQL. Una qualunque versione di MySQL dovrebbe andare bene, non sono state utilizzate funzionalità recenti. All'interno del repository è presente un file `.sql` per importare le varie tabelle che compongono il database.

Oltre al semplice DB relazionale è stato utilizzato anche un database NoSQL. Tra le varie possibilità è stato scelto `MongoDB`, dato che i dati memorizzati sono dei documenti `JSON`. È necessario

installare nel sistema anche un server di MongoDB, anche se questa dipendenza è opzionale. Il sistema è infatti in grado di funzionare anche senza, a patto di infrangere la prima forma normale. Per un approfondimento fare riferimento a [5.3].

L'interfaccia del PHP per mongo proviene da una libreria esterna, installata tramite `composer`, un *package manager* per le dipendenze in PHP. È quindi necessario "installare" composer, processo molto facile che prevede di scaricare il file `phar` da `https://getcomposer.org/download/`, inserirlo nella root del progetto ed eseguire nel terminale `php composer.phar update`. La versione corretta delle librerie verrà scaricata ed inserita in `vendor/`.

3.2 Configurazione

L'applicazione viene configurata attraverso il file `config/config.ini`, il quale viene ricaricato ad ogni caricamento di una pagina. All'interno di questo file ini sono presenti diverse sezioni per configurare le varie parti che compongono l'applicazione.

3.2.1 Sezione `database`

In questa sezione verranno configurati i vari parametri relativi alla connessione e all'uso dei vari database. In particolare verranno specificate le varie stringhe di connessione ai DBMS e le credenziali da usare.

- `string`: stringa di PDO sa usare per connettersi al database MySQL. Normalmente viene usata una stringa simile a `mysql:host=localhost;dbname=lupus`
- `username`: username dell'utente del database
- `password`: password dell'utente specificato
- `mongo_string`: stringa di connessione al database di `MongoDB`, può essere vuota nel caso non si possa usare un database mongo. Un valore tipico può essere: `mongodb://localhost:27017`
- `mongo_fallback`: se il database di mongo non è disponibile è opportuno impostare questa opzione a `true` per utilizzare MySQL come fallback. Fare riferimento a [5.3].

3.2.2 Sezione `log`

L'applicazione dispone di un sistema di logging degli eventi del server, utile per il debug ma anche per il controllo dell'integrità in fase di produzione.

- `level`: livello di output del log, maggiore è il livello, più dati sono stampati. Un valore pari a 0 mostrerà solo errori molto gravi, mentre un valore di 4 stamperà molto output. Il valore 4 è utile solo per il debug di brevi periodi in quanto genera file molto grandi.
- `path`: percorso relativo alla root dell'applicazione del file di log da usare, il file deve essere scrivibile altrimenti l'applicazione potrebbe non funzionare correttamente.

3.2.3 Sezione `webapp`

L'applicazione web ha bisogno di alcune informazioni aggiuntive per funzionare correttamente. In particolare se questa non viene hostata in un sottodominio dedicato ma in una sottodirectory, per esempio `http://example.com/lupus` è necessario specificare in questa sezione la parte di percorso da aggiungere agli url.

- `basedir`: parte dell'url che deve essere aggiunta. Per esempio, nel caso di hosting in `http://example.com/lupus` è necessario impostare `/lupus`. Nel caso in cui l'applicazione sia hostata in un sottodominio dedicato, per esempio `http://lupus.example.com` è opportuno lasciare vuota questa opzione. **Attenzione**: è anche necessario specificare anche lo

stesso valore all'interno di `js/default.js` ed eventualmente modificare il percorso da usare per le API.

3.2.4 Sezione `game`

È possibile personalizzare alcuni parametri del gioco, come il numero minimo di giocatori o il numero massimo. Queste impostazioni non vengono controllate, è opportuno inserire valori coerenti. Inserire per esempio un numero minimo di giocatori troppo basso può creare partite non valide. Ammettere troppi giocatori potrebbe sovraccaricare il server.

- `min_players`: numero minimo di giocatori all'interno di una partita
- `max_players`: numero massimo di giocatori all'interno di una partita
- `lupus_cutoff`: numero di giocatori minimo per scattare da `lupus_low` lupi a `lupus_hi`
- `lupus_low`: numero di lupi se il numero di giocatori è minore di `lupus_cutoff`.
- `lupus_hi`: numero di lupi se il numero di giocatori è maggiore o uguale di `lupus_cutoff`.

3.3 Pagine web

L'applicazione è utilizzabile attraverso delle pagine web generate dal PHP. Queste pagine sono tutte fornite da un unico script, `wrapper.php`, il quale, in base all'url, decide quale file renderizzare.

Nel caso in cui l'applicazione venga esposta in una sottodirectory del server web è necessario configurare la variabile `basedir` nel file di configurazione. Dall'url è necessario rimuovere o aggiungere quella parte per ottenere l'effettiva pagina richiesta. Lo stesso vale anche per le API, le quali di default vengono esposte in `$basedir/api` ma è possibile modificare questo comportamento andando ad agire nel file `js/default.js`.

Per riconoscere una pagina l'url viene testato su ognuna delle possibilità, in ordine, la prima che viene riconosciuta viene usata. Se l'indirzzo non viene riconosciuto viene effettuato un redirect alla homepage.

- `/index` Home page dell'applicazione
- `/login` Pagina per effettuare il login applicazione
- `/signup` Pagina di registrazione
- `/game` Elenco delle partite giocate dall'utente
- `/game/(room)/_new` Pagina per creare una nuova partita nella stanza
- `/game/(room)/(game)` Pagina relativa ad una partita specifica
- `/game/(room)/(game)/admin` Pagina di amministrazione di una partita
- `/room` Elenco delle stanze dell'utente
- `/room/(room)` Pagina di informazioni si una specifica stanza
- `/room/_new` Pagina per creare una nuova stanza
- `/join` Elenco delle partite in cui l'utente può entrare
- `/user` Pagina dell'utente connesso
- `/user/(username)` Pagina di un utente in particolare

3.4 Webservice REST

Tutto ciò che è possibile fare attraverso le pagine web è possibile anche attraverso le API RESTful, così facendo è possibile scrivere una app nativa per dispositivi mobili. L'interfaccia è molto semplice e rispetta le caratteristiche di REST. Le risorse sono identificate tramite URL, i dati sono inviati tramite GET o POST a seconda del tipo di richiesta.

- `/login` Effettua il login tramite `username/password` e salva nella *sessione* l'identificativo dell'utente. Devono essere specificati i parametri `username` e `password` tramite `POST`

- `/login/(username)` Sostituisce il metodo precedente. Effettua il login dell'utente specificato. La password va specificata in `POST` come per `/login`. Se viene specificato anche lo `username` in `POST` viene ignorato.
- `/logout` Se l'utente è connesso lo disconnette cancellando la sessione
- `/user/(username)` Mostra le informazioni dell'utente specificato
- `/me` Scorciatoia per `/user/(username)` con lo `username` dell'utente
- `/room/(room_name)` Mostra le informazioni della stanza specificata
- `/room/(room_name)/add_acl` Permette all'utente `username` in `POST` di accedere alla stanza
- `/room/(room_name)/remove_acl` Nega all'utente `id_user` in `POST` di accedere alla stanza
- `/room/(room_name)/autocompletion` Specificando il parametro `q` in `GET` viene restituita una lista di username che assomigliano a `q`
- `/game/(room_name)/(game_name)` Mostra le informazioni della partita specificata
- `/game/(room_name)/(game_name)/vote` Effettua la votazione di un utente. Il voto deve essere l'`username` dell'utente votato nel parametro `vote` in `POST`.
- `/game/(room_name)/(game_name)/join` Prova ad entrare nella partita specificata. Se si ha raggiunto il numero di giocatori, la partita inizia
- `/game/(room_name)/(game_name)/start` Avvia la partita e la porta allo stato `NotStarted` per far entrare i giocatori.
- `/game/(room_name)/(game_name)/admin/term` Termina la partita e passa allo stato `TermByAdmin`.
- `/game/(room_name)/(game_name)/admin/kick` Espelle un giocatore dalla partita, lo username del giocatore deve essere passato in `POST`.
- `/new_room/(room_name)` Crea una nuova stanza appartenente all'utente. Deve venire specificato il parametro `descr` in `POST`. Può essere specificato il parametro `private` per rendere la stanza privata
- `/new_game/(room_name)/(game_name)` Crea una nuova partita nella stanza specificata. Devono venire specificati i parametri `descr` e `num_players` in `POST`.
- `/notification/dismiss` Nasconde la notifica con l'`id_notification` impostato in `POST`.
- `/notification/update` Restituisce le ultime notifiche dell'utente dalla data `since`. È possibile ottenere le notifiche nascoste impostando `hidden` a 1. Di default vengono ritornate le ultime 5 notifiche, è possibile ottenerne un numero diverso impostando `limit`

3.5 Polimorfismo dei ruoli

Lupus in Tabula è un gioco intrinsecamente pieno di polimorfismo, i ruoli ne sono un esempio tipico. Ogni giocatore in una partita possiede un ruolo, possibilmente diverso da quello degli altri giocatori. Ogni tipo di ruolo dispone di proprietà e caratteristiche diverse ed è necessario che il tutto venga gestito in modo molto flessibile per consentire un'aggiunta semplice di nuovi ruoli e funzionalità.

Alla base della gestione dei vari ruoli c'è una classe base `Role` che si occupa di istanziare correttamente le classi derivate e la chiamata di eventuali metodi specifici. È necessario prestare una particolare attenzione alle convenzioni usate per gestire i vari ruoli, per esempio il nome del file sorgente è molto importate per la corretta individuazione del ruolo corrispondente.

Ogni ruolo deve personalizzare alcune proprietà statiche derivate dalla classe padre `Role` per gestire correttamente alcune parti vitali della partita. Per esempio ogni ruolo dispone di una priorità che gestisce l'ordine in cui vengono eseguite le azioni durante la notte.

Le proprietà statiche presenti all'interno della classe base `Role` che è necessario personalizzare sono:

- `$role_name` Il nome breve del ruolo, deve essere unico e viene usato per identificarlo. Deve essere composto solo da lettere minuscole dell'alfabeto inglese. Questa proprietà deve anche

coincidere con il nome della classe (con la sola iniziale maiuscola) e del file, il quale è nel formato `class.Ruolo.php`
- `$name` Nome del ruolo, deve essere `$role_name` con l'iniziale maiuscola
- `$debug` Se viene impostato a `true` il ruolo è disponibile sono per gli utenti che hanno accesso alle funzionalità in beta
- `$enabled` Se viene impostato a `true` il ruolo non è utilizzabile
- `$priority` Valore numerico che indica la priorità di esecuzione delle azioni durante la notte. Un ruolo con un indice di priorità inferiore viene eseguito prima
- `$team_name` Nome della fazione a cui appartiene il ruolo. Deve essere una delle costanti definite in `RoleTeam`
- `$mana` Valore che indica se il ruolo è buono o cattivo. Deve essere una delle costanti definite in `Mana`
- `$chat_groups` Vettore che elenca i codici delle chat disponibili per il ruolo. Per esempio i lupi hanno accesso ad una chat privata. I valori in questo vettore devono essere delle costanti definite in `ChatGroup`
- `$gen_probability` Valore indicativo che definisce un livello di probabilità della generazione del ruolo in caso di generazione automatica dei ruoli. Non è necessario che sia minore di 1
- `$gen_number` Numero di copie dello stesso ruolo da generare inseme, per esempio i massoni che vengono generati in coppia avranno questo valore impostato a 2

Oltre a queste proprietà statiche che descrivono i vari ruoli è necessaria una parte di codice per l'esecuzione delle varie azioni specifiche di ogni ruolo. La classe base `Role` dispone di diversi metodi, non solo astratti, da sovrascrivere per personalizzare il comportamento di ogni specifico ruolo. Questi metodi sono qui elencati e descritti, quelli con l'asterisco sono astratti:

- `splash()`* Questa funzione ritorna una stringa da usare come messaggio con il nome del ruolo, per esempio per il ruolo Lupo ritornerà *Sei un lupo*
- `needVoteNight()` Questo metodo di controllo serve per verificare se il giocatore con questo ruolo deve ancora effettuare la votazione notturna. Il valore di ritorno è un vettore che contiene un breve messaggio `HTML` e una lista degli username votabili. Nella lista degli utenti votabili potrebbero essere presenti delle stringhe che non sono veri username ma dei *segnaposto* come, per esempio, *(nessuno)* per effettuare una votazione nulla. Se la funzione ritorna false il giocatore non deve votare
- `performActionNight()` Viene effettuata l'azione notturna corrispondente al ruolo. Se questa funzione dovesse ritornare `false` la partita verrebbe interrotta
- `checkVoteNight($username)` Controlla se `$username` è un valore di votazione valido, se la funzione ritorna false il voto viene annullato
- `needVoteDay()` L'equivalente di `needVoteNight()` per le azioni diurne
- `performActionDay()` L'equivalente di `performActionNight()` per le azioni diurne
- `checkVoteDay()` L'equivalente di `checkVoteNight()` per le azioni diurne

Per il funzionamento di alcuni ruoli è anche necessario l'utilizzo di funzioni che permettono il monitoraggio di alcuni parametri della partita. Per esempio tramite le funzioni `visit` e `unvisit` è possibile memorizzare che un certo utente ha fatto visita ad un altro, informazione utile per il *paparazzo*. La funzione `kill` invece si occupa di *uccidere* un giocatore facendo tutti i controlli del caso, per esempio non è possibile uccidere un giocatore morto o un giocatore protetto. È infatti presente un complesso sistema di protezione che si occupa di evitare che alcuni giocatori muoiano per mano di altri. Per esempio è possibile proteggere un giocatore specifico, un intero ruolo o tutti i giocatori da un altro giocatore, un altro ruolo o anche da tutti i giocatori della partita. Questo viene fatto memorizzando una lista di protezioni, le quali vengono identificate da un codice, `@username` per un utente specifico, `#group` per un gruppo, oppure `*` per ogni giocatore.

All'interno della base di dati, in particolare nella tabella `player` viene memorizzata l'associazione `user` → `role`. In particolare come chiave esterna del ruolo viene usata la proprietà `$role_name`. Quando la partita non è ancora iniziata e i ruoli non sono ancora stati assegnati ai giocatori viene usato il valore di controllo `unknown`.

Per istanziare correttamente i ruoli e per ottenere le proprietà statiche richieste viene utilizzata una funzionalità del linguaggio `PHP` che permette di usare il contenuto di una variabile come nome di classe o di metodo. In particolare cercando di accedere a dei campi statici di una stringa (la quale naturalmente non ne ha) si accede in realtà ai campi di una classe che ha come nome il contenuto della stringa. È necessario però essere certi che il valore della stringa corrisponda esattamente al nome della classe del ruolo e che questa classe derivi da `Role`. Per questi controlli vengono utilizzate le funzioni standard `class_exists` e `class_parents`.

Parte II

Limiti della 1NF

Capitolo 4

Introduzione alle forme normali

4.1 Cenni introduttivi

Le *forme normali* sono delle caratteristiche dei database relazionali utili per eliminare delle possibili ridondanze e inconsistenze. La normalizzazione di una base di dati è quindi il processo che si occupa di rendere delle tabelle in forma normale. Esistono numerose forme normali, dalle più semplici alle più complesse.

Il concetto di *dipendenza funzionale* è necessario per comprendere le definizioni delle forme normali dalla terza in poi. Considerando A e B degli insiemi di attributi, la dipendenza funzionale $A \to B$ indica che, conoscendo A, è possibile stabilire il valore di B. Per esempio, in una tabella contentente dei dati anagrafici, conoscendo il *codice fiscale* (chiave primaria della tabella) è possibile ricavare il nome e il cognome; si ha quindi la dipendenza funzionale $(CF) \to (nome, cognome)$.

4.2 Prima forma normale

La Prima Forma Normale (1NF) venne ideata da Edgar Codd nel 1971 con una definizione molto semplice che venne poi ampliata e studiata da Hugh Darwen e Chris Date.

La definizione iniziale di Codd è stata:

> Una relazione è in prima forma normale se ha la proprietà che nessuno dei suoi domini ha elementi che sono degli insiemi
>
> *«A relation is in first normal form if it has the property that none of its domains has elements which are themselves sets»* [2]

Più semplicemente nessuno degli attributi di una tabella deve essere scomponibile in parti più semplici, ogni attributo deve quindi essere atomico. Questa definizione non è sufficientemente chiara, pur escludendo la possibilità di inserire degli insiemi in una singola colonna rischia di escludere anche dei tipi di dato fondamentali. Per esempio una stringa può essere considerato un'insieme di caratteri, un numero decimale può essere considerato l'unione di un numero intero e della sua parte decimale. Secondo Christopher Date quindi il concetto di atomicità non ha significato (*«the notion of atomicity has no absolute meaning»* [3, p. 112]).

Secondo Date la prima forma normale andrebbe riscritta, questa dovrebbe imporre le seguenti 5 condizioni [3, p. 127-128]:

1. Tra le righe non deve importare l'ordine
2. Tra le colonne non deve importare l'ordine
3. Non ci devono essere delle righe duplicate
4. Ogni cella della tabelle deve contenere un solo valore del dominio
5. Tutte le colonne sono regolari, non ci sono campi nascosti nelle righe

Nella trattazione successiva riguardo le problematiche riscontrate di fa riferimento solo alla definizione di Codd riguardo all'atomicità, la definizione di Date non verrà considerata.

Per esempio, la seguente relazione non si trova in 1NF, in quanto l'ultimo campo ha come dominio un insieme di numeri di telefono.

ID	Nome	Cognome	Numeri di telefono
1	Edoardo	Morassutto	333-3141592 338-1472583
2	Mario	Rossi	124-5214225
3	Fabio	Bianchi	123-1231231

Una possibile soluzione al problema è quella di spezzare la tabella in due, una per memorizzare i dati delle persone (nome e cognome) e una per memorizzare i numeri di telefono.

ID	Nome	Cognome
1	Edoardo	Morassutto
2	Mario	Rossi
3	Fabio	Bianchi

ID	Numero di telefono
1	333-3141592
1	338-1472583
2	124-5214225
3	123-1231231

4.3 Seconda forma normale

La Seconda Forma Normale (2NF) fu anch'essa stilata da E. Codd nel 1971 [2]. La 2NF è un'estensione della prima forma normale; per essere in seconda forma normale, quindi, la tabella deve anche essere in 1NF.

Secondo la definizione, ogni attributo non primo[1] deve dipendere dall'intera chiave candidata e non da parte di essa. Per esempio, in una relazione, $R(\underline{A, B}, C, D, E)$ dove le dipendenze funzionali sono $(A, B) \to (C, D, E)$ e $(A) \to (C)$, quindi A e B compongono la chiave. La relazione non è in seconda forma normale in quanto l'attributo C dipende solo da A, quindi da parte della chiave.

Questo è un esempio pratico [5] di tabella che non è in seconda forma normale, la tabella memorizza il numero di reti di ogni giocatore in un determinato anno, in più viene memorizzato anche il luogo di nascita di ogni giocatore.

[1] Un attributo si dice primo quando fa parte di una chiave candidata

nome	**luogo**	**anno**	**reti**
Rossi Mario	Firenze	1998/1999	3
Rossi Mario	Firenze	1999/2000	4
Conti Bruno	Roma	1998/1999	6

Tabella 4.1: Tabella campionati

Per risolvere il problema e rendere la tabella in (almeno) seconda forma normale è possibile dividere la relazione in due, creando una tabella per i giocatori e una per i campionati. In particolare, dato che il luogo dipende solo dal nome del giocatore è possibile memorizzare questa informazione una volta sola in una tabella separata.

nome	**luogo**
Rossi Mario	Firenze
Conti Bruno	Roma

nome	**anno**	**reti**
Rossi Mario	1998/1999	3
Rossi Mario	1999/2000	4
Conti Bruno	1998/1999	6

Tabella 4.2: Soluzione: tabella giocatori e tabella campionati

4.4 Terza forma normale

Tra i requisiti che una tabella deve avere per essere in Terza Forma Normale è l'essere in 1NF e in 2NF. Una volta accertato ciò è possibile controllare, ed eventualmente correggere, per la 3NF. Anche questa forma normale è stata inizialmente ideata da Codd nel 1972 in [2] ed ha una definizione alquanto complessa: per ogni dipendenza funzionale $A \to B$, o A è superchiave[2] o B è primo.

Questa definizione è una semplificazione di quella fornita inizialmente da Codd, la quale si esprime in termini più tecnici, affinché una relazione sia in terza forma normale è necessario che ogni attributo non-primo sia non-transitivamente dipendente da ogni chiave (*«every non-prime attribute of R is non-transitively dependent on every key of R»* [2]).

Secondo Janssen [6] la terza forma normale esiste, oltre ad evitare la ridondanza, anche per rendere alcune query più efficienti riducendo in più lo spezio richiesto per la memorizzazione. Si può quindi stabilire che la terza è la forma normale minima consigliata per un qualunque database relazionale, si ha così una buona garanzia di non-ridondanza unita, in generale, a delle buone prestazioni.

4.5 Forma normale di Boyce-Codd

Questa forma normale, spesso chiamata anche 3.5NF, è un'estensione della terza forma normale. Più propriamente questa forma normale è una restrizione delle condizioni imposte dalla terza, in particolare, oltre al requisito delle 2NF, è necessario che per ogni dipendenza funzionale $A \to B$, A sia superchiave. Questa limitazione, seppur minima rende alcune relazioni impossibili da ristrutturare, in [1] sono mostrati alcuni validi esempi di tabelle di questo tipo.

[2] Un insieme di attributi si dice superchiave se contiene interamente la chiave

Piccola nota storica: seppur questa forma normale si chiami di *Boyce-Codd*, Chris Date fa notare che già tre anni prima della formulazione dei due informatici, Ian Heath, in una pubblicazione del 1971 [4], mostrò la stessa definizione. Secondo Date quindi la 3.5NF andrebbe chiamata Forma Normale di Heath ma ciò non viene fatto per la *Legge dell'eponimia di Stigler*[3]

4.6 Altre forme normali

Oltre alle quattro principali forme normali ne esistono altre, spesso ignorate e non approfondite in ambito scolastico. Per esempio, dopo la Forma Normale di Boyce-Codd, si può parlare di Quarta Forma Normale introducendo il concetto di *Dipendenze Multivalore*, poi di Quinta e di Sesta. Queste forme normali sono relativamente poco utili e aggiungono un livello notevole di complessità nella base di dati affinché vengano rispettate sempre. Per quasi tutti i casi di database relazionali non è necessario rispettare queste forme normali, limitarsi alla terza o a quella di Boyce-Codd è sufficiente.

[3] Questa legge asserisce che: *«A una scoperta scientifica non si dà mai il nome del suo autore»* [8]

Capitolo 5

Problematiche con la 1NF

5.1 Contesto del problema

In questo capitolo si farà spesso riferimento all'applicazione descritta nella Parte I, con particolare riferimento alla sua base di dati.

Come descritto, uno dei problemi principali di Lupus in Tabula è la gestione dei ruoli. I ruoli sono molti e possono aumentare e diminuire molto rapidamente, in più ogni ruolo è intrinsecamente diverso, questi infatti condividono alcune proprietà ma differiscono in altre. Per esempio alcuni ruoli necessitano di memorizzare da qualche parte delle informazioni, spesso molto diverse da quelle memorizzate da qualche altro ruolo. Spesso queste informazioni non sono ben strutturate o a volte lo sono troppo, basti pensare ad un ruolo che deve memorizzare delle informazioni come una lista di giocatori o, peggio ancora, un albero di dati.

I ruoli non sono l'unico caso di problematica nella base di dati, ci sono altre sezioni dell'applicazione che soffrono di problemi simili. La parte di generazione di una partita per esempio deve memorizzare alcune informazioni per generare i ruoli dei giocatori. Queste informazioni sono mantenute in un oggetto composto da alcune liste che, per esempio, indicano per ogni ruolo possibile il numero di giocatori da generare.

Un'altra problematica è la memorizzazione di alcune informazioni riguardo ai messaggi letti nelle chat. Ogni giocatore ha accesso ad un numero variabile di chat nelle quali possono arrivare dei messaggi che l'utente può leggere. Per sapere quali messaggi deve ancora leggere è necessario sapere fino a dove è arrivato con la lettura. È quindi necessario memorizzare queste informazioni in qualche posto, la problematica è molto simile a quella per la generazione della partita, ogni possibile chat deve sapere qual'è l'ultimo messaggio letto (o il relativo `timestamp`).

Infine la quarta parte che soffre di questo problema riguarda gli eventi della partita. Durante il gioco accadono degli eventi che possono essere di molti tipi diversi. Per esempio l'inizio della partita genera un evento con la data di inizio, la morte di un giocatore ne genera un altro con lo username del giocatore morto, l'azione di un veggente ne genera un altro con i dati della visione. Man mano che il numero di ruoli aumenta il numero di eventi possibili cresce. La problematica è quindi identica a quella relativa alla memorizzazione delle informazioni dei ruoli.

Tutti queste problematiche sono risolvibili con la stessa strategia dato che, in generale, sono raggruppabili in uno stesso problema comune: la memorizzazione di dati complessi e con una struttura variabile.

5.2 Prime soluzioni parziali

La soluzione a questo problema non è unica, ne vengono esposte quattro, non in ordine di correttezza ma in ordine di semplicità. La prima è molto semplice, sia da progettare che da implementare, passando poi per la seconda soluzione più complessa da pensare e da implementare. La terza invece è molto semplice da capire ma molto complessa da realizzare per passare poi alla soluzione proposta, relativamente facile da capire, un po' ostica da implementare ma nel complesso abbastanza efficiente.

5.2.1 Infrangere la prima forma normale

Tutti i dati problematici da memorizzare si possono ridurre, senza perdita di informazione, in una stringa *serializzata*. La serializzazione è quindi il processo che, dati in input dei dati strutturati, ne cambia il formato in qualcosa di memorizzabile o trasmissibile. Esistono diversi metodi che *serializzano* degli oggetti, i due più famosi sono la serializzazione `XML` e la serializzazione `JSON`.

La prima converte l'oggetto in un documento `XML`, una stringa composta da tag che rappresentano i dati in modo gerarchico in un albero. In modo molto simile anche il `JSON` modella i dati in una sorta di albero, senza però l'utilizzo di tag ma usando dei simboli come parentesi quadre e graffe. Il principale vantaggio del secondo metodo, a discapito di una migliore strutturazione dei dati, è la semplicità. Un documento `JSON` è molto più conciso (molti meno byte) e più leggero (più facile da decodificare) rispetto all'equivalente `XML`.

La prima soluzione è quindi serializzare i dati *problematici* in `JSON` da memorizzare in colonne dedicate sotto forma di stringhe. Questo però infrange la Prima Forma Normale, esisterebbero infatti degli attributi che non sono atomici dato che sono l'aggregazione di dati semplici.

Dal punto di vista pratico questa soluzione è facilmente realizzabile, ogni volta che è necessario scrivere in quella colonna è sufficiente serializzare i dati (per esempio con `json_encode`) e ogni volta che vanno letti basta deserializzarli (per esempio con `json_decode`).

Il principale problema di questa soluzione, che deriva infatti dalla Prima Forma Normale, è il rischio di inconsistenza. Non è possibile infatti realizzare delle relazioni e tutelare i dati con i vincoli di chiave esterna. È quindi necessario prestare la massima attenzione nella manipolazione dei dati all'interno del database, aggiungendo tutti i controlli del caso lato applicazione.

Questa soluzione è stata utilizzata nelle prime versioni di Lupus in Tabula e solo recentemente è stata modificata a favore della soluzione proposta [5.3].

5.2.2 Una tabella per tipo

Il modo più immediato per risolvere il problema della mancanza di relazioni della prima soluzione si può risolvere creando una tabella per ogni tipo di ruolo. Questa soluzione può sembrare accettabile, permetterebbe alla base di dati di rimanere almeno in Prima Forma Normale e garantirebbe l'integrità referenziale.

Il principale svantaggio di questa soluzione si ha nella realizzazione. Per i ruoli più semplici basta una tabella per ruolo, i più complessi però richiedono anche due o più tabelle messe tra loro in relazione. Il numero di tabelle crescerebbe quindi con il numero di ruoli, andando a complicare ed appesantire notevolmente la base di dati.

Lo sviluppo dell'applicazione ne risentirebbe anche di più, è necessario implementare delle interfacce alle nuove tabelle per riuscire ad estrarre efficacemente i dati. Ognuna di queste interfacce sarebbe diversa dalle altre, spesso anche complicate da scrivere e da gestire.

Queste problematiche si applicano anche agli altri dati da memorizzare, gli eventi per esempio richiedono una tabella per tipologia. Le informazioni per la generazione di una partita richiedono una colonna in più per ogni ruolo per memorizzare il numero di giocatori.

Questa soluzione non è stata implementata per la sua eccessiva complessità rispetto alla prima soluzione o alla soluzione proposta.

5.2.3 Tabelle key-value

Una soluzione completamente diversa dalle precedenti che prende spunto da alcune tipologie di database NoSQL è la seguente; mantenere un numero molto limitato di tabelle con due colonne principali, `key` e `value`.

Tutti i dati vengono destrutturati in una serie di parti, per esempio una relativa alle relazioni con i giocatori, una con le relazioni con le partite, una con gli eventi, ecc... Ognuna di queste parti viene ancora scomposta in elementi minimi, associazioni chiave-valore di informazioni. Per esempio è possibile destrutturare l'informazione *L'assassino è user1* nella coppia chiave-valore $assassino \rightarrow user1$ in una tabella associata con `game`.

Così facendo la struttura delle tabelle rimarrebbe costante con l'aggiunta di ruoli o di funzionalità nuove. Progettando efficientemente queste tabelle speciali è possibile rendere questa soluzione sufficientemente flessibile richiedendo rare modifiche alla struttura della base di dati.

Il principale svantaggio di questa soluzione, oltre all'enorme complessità in fase di progettazione, è la non strutturazione dei dati memorizzati. Questi sono infatti frammentari e, pur risolvendo in parte il problema dei vincoli di integrità referenziale, rende pressoché impossibile la verifica dell'effettiva consistenza dell'informazione memorizzata. Nel caso mancasse una riga in una tabella non sarebbe ovvio riuscire a trovare e risolvere il problema. In più questa soluzione andrebbe anche ad appesantire la base di dati che si vede diverse tabelle con un numero molto elevato di record memorizzati.

5.3 Soluzione proposta

La soluzione proposta è alquanto inconsueta, viene infatti implementato un sistema eterogeneo tra database relazionali e NoSQL. Molte delle informazioni da memorizzare sono molto strutturate e i database relazionali sono perfetti per la loro memorizzazione, altri invece non si candidano per essere salvati in un RDBMS. Questi verranno quindi memorizzati in una base di dati esterna a MySQL.

Tra i principali vantaggi di memorizzare i dati in questo modo è la separazione dei dati strutturati da quelli non strutturati. Il database relazionale non dovrà gestire dati complessi, viene infatti mantenuta la prima forma normale. Ovviamente questa soluzione non permette di garantire l'integrità referenziale, non c'è infatti modo di creare un'associazione tra le due basi di dati.

Sarà quindi necessario avere installate due basi di dati: MySQL e MongoDB. Questi due DBMS non devono necessariamente essere installati nella stessa macchina server, è infatti possibile distribuire il carico in due o più macchine dato che i due database sono indipendenti.

Naturalmente per evitare problemi di inconsistenza è necessario accertarsi che le transazioni comprendano entrambe le basi di dati, non deve accadere infatti che, per esempio, vengano aggiornati i dati solo nel database relazionale o solo in quello NoSQL. Per accertarsi di ciò è necessario che il codice sia sottoposto a rigidi test.

La dipendenza aggiuntiva di MongoDB può essere un problema per i sistemi datati o limitati. I requisiti di sistema dell'applicazione aumentano dato che è necessario avviare anche un secondo DBMS a fianco di MySQL, è possibile evitare ciò utilizzando un *fallback*. Nel file di configurazione dell'applicazione è possibile specificare che, nel caso in cui MongoDB non fosse disponibile, di usare solo MySQL come base di dati. Il funzionamento di questo metodo è molto semplice, viene

utilizzata la prima soluzione [5.2.1] cioè i dati da memorizzare come oggetto in MongoDB vengono memorizzati come `JSON` in MySQL.

L'efficienza di questa soluzione nasce da fatto che MongoDB è pensato proprio per memorizzare dati di questo genere, oggetti senza una struttura costante definita. Sarebbe anche possibile usare solo MongoDB come database per l'intera applicazione ma ci sarebbero notevoli svantaggi: l'applicazione non sarebbe più così portabile, non sarebbe possibile installarla in hosting gratuito per esempio su Altervista. La notevole struttura della maggior parte dei dati dell'applicazione scoraggia l'utilizzo di solo una base di dati non strutturata come un NoSQL.

Un ulteriore punto di forza della coesistenza RDBMS-NoSQL è lo scenario dei *big* dell'informatica. Attualmente alcune tra le più importanti compagnie come Google, Facebook, Digg e Amazon investono molto nello sviluppo di NoSQL[7] e alcuni dei servizi da loro offerti permettono la coesistenza dei due mondi.

Bibliografia

[1] *3NF table not meeting BCNF (Boyce–Codd normal form)*. URL: https://en.wikipedia.org/wiki/Boyce%E2%80%93Codd_normal_form#3NF_table_not_meeting_BCNF_.28Boyce.E2.80.93Codd_normal_form.29.

[2] E. F. Codd. *Further normalization of the database relational model*. Courant Institute: Prentice-Hall, Oct 1972. ISBN: 013196741X.

[3] C. Date. *Date on Database: Writings 2000-2006*. Springer-Verlag, 2007.

[4] I. Heath. «Unacceptable File Operations in a Relational Database.» 1971.

[5] C. Iacobelli et al. *Eprogram Informatica*. Juvenilia Scuola, 2014. ISBN: 9788874853939.

[6] Cory Janssen. *What is Third Normal Form?* 2014. URL: https://www.techopedia.com/definition/22561/third-normal-form-3nf.

[7] Herman Mehling. *How NoSQL and Relational Database Storage Can Coexist*. URL: http://www.devx.com/dbzone/Article/45636.

[8] S. Stigler. *Statistics on the Table: The History of Statistical Concepts and Methods*. 1999. ISBN: 0674836014.

[9] *The Original Mafia Rules*. URL: http://web.archive.org/web/19990302082118/http://members.theglobe.com/mafia_rules/.