Advanced Techniques and Tools for Software Development

RICERCA DEL CAMMINO MINIMO IN UN GRAFO CON TOPOLOGIA A GRIGLIA

TOMMASO PUCCETTI, ALEX FOGLIA, FRANCESCO SECCI

Anno Accademico 2017-2018

INDICE

ELENCO DELLE TABELLE

ELENCO DELLE FIGURE

1

SISTEMA IMPLEMENTATO

PANORAMICA AD ALTO LIVELLO

Il sistema che abbiamo implementato è una *web-app* la cui funzione è quella di ricercare il cammino minimo in un grafo con topologia a griglia, dati due particolari nodi, uno di partenza e uno di arrivo.
Una *griglia* è rappresentata attraverso una matrice $A \in \mathbb{N}^{n x n}$ siffatta:

$$A = [a_{i,j}]$$

Dove $a_{i,j} > 0 \iff \exists$ un nodo nella riga i e colonna j della griglia rappresentata da A e n è il numero massimo di righe e colonne ammesso per una particolare griglia.
Occorre precisare che in questa particolare topologia di grafo, ogni nodo i, j può avere al più 4 vicini:

- $i + 1, j$
- $i - 1, j$
- $i, j + 1$
- $i, j - 1$

Pertanto il cammino minimo potrà attraversare archi solo in direzione orizzontale o verticale.

Tabella 1: Matrice A

1	0	1	0	0
1	1	1	0	0
1	0	0	0	1
1	1	1	1	1
0	0	0	0	1

Tabella 2: Cammino minimo relativo ad A

1	0	1	0	0
1	1	1	0	0
1	0	0	0	1
1	1	1	1	1
0	0	0	0	1

A titolo di esempio, si consideri la matrice in tabella 1. Supponiamo di voler cercare il camminimo minimo che parta dal nodo $0, 0$ e che termini al nodo $4, 4$ (nella fattispecie, esso è unico). Si ottiene il risultato esposto in tabella 2.

L'implementazione del suddetto sistema si articola in due componenti:

1. Server
2. Client

Il Server, realizzato attraverso l'utilizzo del framework *Spring Boot*, gestisce due distinti database contenenti le matrici che rappresentano particolari griglie; un database è di tipo *no-sql* (MongoDB), l'altro è invece un database *sql* (MySQL).
Il Server deve poter essere in grado di gestire tre tipi di richieste pervenibili attraverso la sua interfaccia *REST*: recuperare tutte le griglie salvate in un database, recuperarne una in particolare, oppure trovare un eventuale cammino minimo in una particolare griglia.
Per quanto riguarda le operazioni *CRUD* sul database, queste sono realizzate mediante un *Web Controller* protetto da username e password. Occorre infine precisare che il server non è consapevole di colloquiare con una particolare istanza di un client, ciò comporta che le richieste vengano ugualmente servite sia nel caso in cui il client usi un normale browser, oppure il software che abbiamo realizzato.
Il Client da noi implementato è un' applicazione Java che utilizza la libreria *Swing* per fornire all'utente un'interfaccia grafica *user-friendly*. Il Client é capace di eseguire le tre possibili richieste *HTTP* inoltrabili all'interfaccia *REST* del Server.

2

TECNICHE E FRAMEWORK UTILIZZATI

CODE VERSIONING

Per tenere traccia della storia dei codici di Server e Client abbiamo utilizzato il software *git* creando una repository locale connessa a una repository remota su *github*. Le repository remote utilizzate sono disponibili per la consultazione ai seguenti URL:

- Server - `https://github.com/francescosecci/sp-server-attsw`
- Client - `https://github.com/francescosecci/sp-client-attsw`

L'utilizzo di *git* si è rivelato particolarmente utile per il lavoro in gruppo, grazie al meccanismo delle *pull requests*; grazie alle quali abbiamo aperto un ramo ogni qualvolta un membro del gruppo ha implementato una particolare *feature* all'interno dei software. Abbiamo dunque realizzato sul master tutte le interfacce di cui gli applicativi faranno uso e creato un ramo per ogni loro singola implementazione. Quando una particolare implementazione veniva terminata, allora procedevamo al *merge* del ramo sul master risolvendo gli eventuali conflitti generati, mediante l'interfaccia di *github*.
Non abbiamo sollevato alcuna *issue* perchè abbiamo lavorato per la maggior parte del tempo nel medesimo luogo fisico.
Git è stato utilizzato sia da terminale, che tramite il plugin per eclipse *e-git* che ci ha semplificato alcune operazioni.

DEPENDENCIES AND BUILD TOOL

Per gestire le dipendenze e realizzare la build del codice, abbiamo utilizzato il software *Apache Maven*, in particolare ne abbiamo utilizzato la sua

integrazione nativa in eclipse.
Dal momento che occorre fornire evidenze su metriche quali *code coverage* e qualità del codice abbiamo realizzato i seguenti profili Maven all'interno del POM con lo scopo di eseguire differenti processi di build in relazione ai particolari plugin che era necessario legare a ciascuna fase del ciclo di vita di Maven.

```
clean verify -Pjacoco coveralls:report
clean verify -Pjacoco sonar:sonar
clean verify -Pfailsafe
```

Il profilo jacoco serve a generare i report sugli unit test e ad inviarli rispettivamente ai server di *Coveralls* e di *SonarQube* (*SonarCloud*); mentre il profilo failsafe esegue gli integration test. Questi profili sono utlizzati sia nel Client che nel Server.

CONTINUOS INTEGRATION

Le repository elencate in 2.1 sono collegate al server di Continuos Integration *Travis* attraverso il file *.travis.yml*. Questo file rappresenta lo script che *Travis* deve eseguire sulle macchine remote al fine di ricreare un ambiente in cui è possibile lanciare le stesse build che effettuiamo anche in locale. Quando *Travis* finisce di ricreare l'ambiente, allora esegue la build come specificato sempre nel file *.travis.yml*. Dal momento che ciascuna build viene eseguita usando i profili Jacoco e Failsafe definiti nel POM, questa genera i report su *Coveralls* e *SonarCloud* sia nel caso in cui la build venga fatta in locale che da *Travis*.

FRAMEWORKS

Spring Boot

Nel capitolo 1 si menziona Spring Boot quale framework utilizzato per realizzare il Server. Spring Boot è un particolare framework che facilita il programmatore nella realizzazione di una *web-app* evitando allo stesso di utilizzare direttamente le Servlet Java. Spring Boot implementa inoltre il meccanismo di *Inversion of Control* che rende necessaria la *Depencency Injection*. La *Dependency Injection* è necessaria poiché springboot esegue la web-app all'interno di un web container il quale si trova a sua volta all'interno di un server, es. Tomcat. Infatti, proprio perché la web-app viene lanciata all'interno di un web container, non possiamo avere il

totale controllo sull'istanziazione degli oggetti a run-time.
Le principali funzionalità messe a disposizione da Spring Boot che abbiamo utilizzato sono le repository per i database e i controller per interfacciare la web-app verso l'esterno.
Per realizzare il Server, al posto di eclipse, è stato utilizzato Spring Tool Suite, il quale ne rappresenta una versione modificata ad hoc.

Database utilizzati

Il Server utilizza due differenti database: MongoDB e MySQL. Spring Boot permette di passare facilmente dall'utilizzare l'uno o l'altro grazie al meccanismo dei *Profiles* che verrà spiegato nel dettaglio in seguito.

Template engine

Spring Boot, per renderizzare le pagine html che un client può richiedere, utilizza l'engine *Thymeleaf*.

Framework di test

Grazie alla sua facile integrazione con eclipse, gli unit test e gli integration test vengono eseguiti in locale utilizzando il framework *Junit*.
Per testare la corretta realizzazione dell'interfaccia utente di cui fa uso il client abbiamo utilizzato il framework *AssertJ*, che ci ha permesso di scrivere asserzioni sulle componenti grafiche realizzate.
L'interfaccia del Client non è la sola interfaccia utente realizzata, infatti, come già specificato nel capitolo 1, il Server predispone un'interfaccia web dedicata alle operazioni CRUD sui database. Per testare questa interfaccia, che è necessariamente un'interfaccia HTML, abbiamo utilizzato il framework HtmlUnit.

Http Client

Per implementare le richieste HTTP del Client abbiamo scelto il framework *Apache HttpClient* che mette a disposizione le classi necessarie per effettuare richieste HTTP su un particolare server (es. GET oppure POST) evitando di scrivere direttamente codice che usi le classiche Socket.

GUI

Il framework scelto per la realizzazione dell'interfaccia grafica è *Swing*. Questo framework ci ha permesso di integrare Apache Http Client con una semplice interfaccia utente composta da una singola finestra (JFrame).

3

DESIGN E SCELTE IMPLEMENTATIVE

SERVER

Come già specificato nei precedenti capitoli, il Server deve esporre verso l'esterno due interfacce: l'interfaccia REST e l'interfaccia Web. Lo scopo della prima è quello di fornire un'interfaccia che consenta a un client di poter:

- Ricevere la lista di tutte le griglie presenti sul Server;
- Visualizzare una particolare griglia tra quelle disponibili;
- Richiedere un cammino minimo fra due nodi di una particolare griglia.

Per l'interfaccia Web, invece, è previsto che questa debba permettere a un client di:

- Inserire una griglia nel database;
- Visualizzare tutte le griglie salvate nel database;
- Eliminare una griglia dal database.

Entrambe le interfacce fanno uso di un'interfaccia interna all'applicazione: l'interfaccia *IGridService*. Questa interfaccia ha il compito di fornire le operazioni necessarie per svolgere la funzione prevista dal sistema, ossia chi usa IGridService non vede i dettagli implementativi riguardanti come le operazioni di lettura e scrittura sul database vengono effettivamente realizzate. IGridService fornisce inoltre l'operazione di ricerca del cammino minimo, i cui dettagli verranno spiegati in seguito.

Rappresentazione delle griglie

Un oggetto griglia salvato nel database, viene visto dall'interfaccia IGridService come un'istanza della classe *DatabaseGrid*. Questa classe è un wrapper di una matrice di interi, la quale rappresenta il grafo. La classe DatabaseGrid aggiunge inoltre informazioni su una griglia associando ad essa un intero (id) e un intero che rappresenta la dimensione massima di righe e colonne che la matrice può avere (n). Nonostante questa scelta sembra perdere di generalità, in quanto considera solamente matrici quadrate, in realtà se volessimo rappresentare una griglia che non è in generale quadrata, sarebbe sufficiente porre a o le righe o le colonne che non si vuole facciano parte del grafo.
Nel caso in cui si stia usando il Database Mongo, tramite l'interfaccia MongoRepository di Spring Boot, possiamo inserire e leggere oggetti DatabaseGrid dal database Mongo; mentre nel caso in cui si stia usando il Database MySql, la sua repository non può mappare oggetti DatabaseGrid a particolari istanze interne al database perchè il campo matrice non è un tipo di dato primitivo per un database SQL. Per ovviare a questa problematica abbiamo creato la classe SqlGrid che sostiuisce il campo matrice con un campo stringa che rappresenta la matrice stessa. Esistono infine due metodi simmetrici all'interno delle classi SqlGrid e DatabaseGrid che consentono di convertire un'istanza di una classe in un'istanza dell'altra, preservando l'isomorfismo tra i due grafi che esse rappresentano.

REST Controller

Il REST Controller che abbiamo realizzato è una classe, opportunamente annotata secondo la struttura di Spring Boot, che a run-time serve determinate richieste HTTP su tre particolari URL:

- /api/ $\rightarrow$ restituisce al client la lista JSON di tutte le griglie presenti nel database;
- /api/grid[i] $\rightarrow$ restituisce al client l'oggetto griglia avente id=i (serializzato come oggetto JSON);
- /api/path[SOURCE]TO[SINK]IN[ID] $\rightarrow$ restituisce al client una lista JSON di stringhe rappresentanti i nodi che fanno parte del cammino minimo nel grafo con id=ID partendo dal nodo SOURCE

e finendo nel nodo SINK. Se il cammino non esiste, viene restituita una lista vuota.

Web Controller

Al Web Controller sono delegate le operazioni di aggiunta e rimozione di griglie all'interno del database. Tali operazioni sono concesse solo previa autenticazione con username e password. Per implementare il login abbiamo utilizzato la classe WebSecurityConfig, che comunica al framework di abilitare il sistema di login nativo di Spring Boot, escludendo da esso gli URL relativi all'interfaccia REST e permettendo l'accesso a qualsiasi altro URL solamente se identificati con nome utente "user" e password "password".
Gli URL serviti dal Web Controller sono:

- / $\rightarrow$ dalla root si accede a una pagina mediante la quale il client sceglie un particolare servizio del Web Controlller;
- /viewdb $\rightarrow$ viene renderizzata una pagina HTML nella quale sono riportate tutte le griglie presenti nel database;
- /addtable $\rightarrow$ viene renderizzata una pagina HTML che prevede l'immissione tramite un form di una griglia all'interno del database;
- /remtable $\rightarrow$ come addtable, ma il form richiede solo l'inserimento dell'id della griglia che si intende cancellare.

Algoritmo di ricerca del cammino minimo

Per implementare la ricerca del cammino minimo abbiamo realizzato la classe Graph. Questa classe rappresenta in generale qualsiasi tipologia di grafo, di cui la griglia rappresenta un caso particolare. L'implementazione concreta dell'interfaccia IGridService, quando viene interrogata sul camminimo minimo, crea un'istanza della classe Graph partendo dalla matrice che rappresenta la griglia selezionata dal client. La classe Graph è composta da una collezione di nodi e una collezione di archi, i primi sono rappresentati da stringhe e i secondi da vettori di lunghezza 2, tali per cui in posizione 0 e in posizione 1 vi sono rispettivamente il nodo uscente e il nodo entrante del particolare arco che stiamo considerando. Dal punto di vista algoritmico dunque la ricerca del cammino minimo si traduce nella visita di un grafo, secondo il seguente algoritmo.

```
Algoritmo cammino_minimo(source,sink):
        crea nodi_visitati;
        crea nodi_precedenti;
        crea lista_cammino_minimo;
        crea coda;
        Nodo corrente=source;
        coda.add(corrente);
        nodi_visitati.add(corrente,true);
        fintantoche(coda non vuota)
                corrente=coda.remove();
                se corrente==sink allora
                        esci;
                altrimenti
                        per ogni nodo v vicino di current:
                                se nodi_visitati non contiene v:
                                        coda.add(corrente);
                                        nodi_visitati(v,true);
                                        nodi_precedenti.add(v,corrente);
                                fine se
                        fine per
                fine se
        fine fintantoche
        se corrente != sink allora:
                restituisci lista vuota;
        altrimenti
                per Nodo n=sink, se n non nullo, n=nodi_precedenti.get(n):
                        lista_cammino_minimo.add(n);
                fine per
                restituisci reverse(lista_cammino_minimo);
        fine se
fine algoritmo
```

CLIENT

Il Client utilizza l'interfaccia IClient per colloquiare con l'interfaccia REST del Server. Questa classe utilizza al suo interno un'altra interfaccia, IRestServiceClient. Quest'ultima utilizza il framework Apache Http Client col fine di eseguire richieste HTTP sui particolari URL dell'interfaccia REST del Server che abbiamo realizzato. L'interfaccia IClient permette all'utente di:

- ricevere la lista di tutte le griglie presenti sul database del Server;
- ricevere una particolare griglia;
- ricevere un particolare cammino minimo all'interno di una griglia.

Gli oggetti ricevuti sono stringhe JSON nel primo e nel terzo caso, mentre quando si riceve una particolare griglia, se ne riceve la serializzazione JSON. Una griglia serializzata tramite JSON viene poi caricata in memoria come un'istanza della classe GridFromServer; classe del tutto equivalente alla classe DatabaseGrid che abbiamo discusso in 3.1.1.

Interfaccia utente

L'applicazione mette a disposizione dell'utente la seguente interfaccia grafica per fare uso di un'implementazione concreta di IClient.

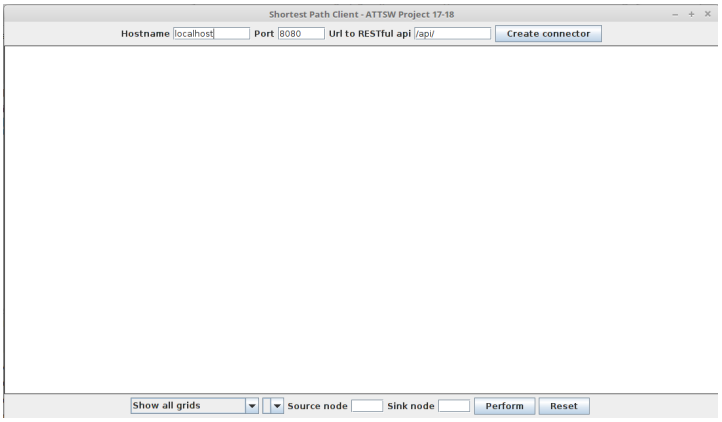

Figura 1: Interfaccia del Client all'avvio del sistema

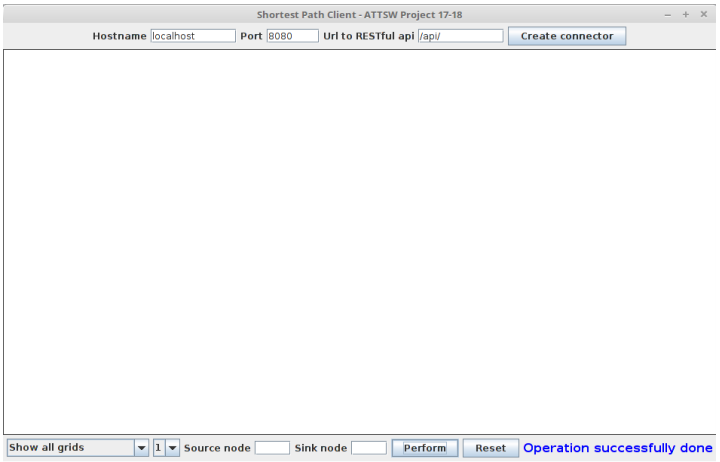

Figura 2: Richiedi tutte le griglie al Server

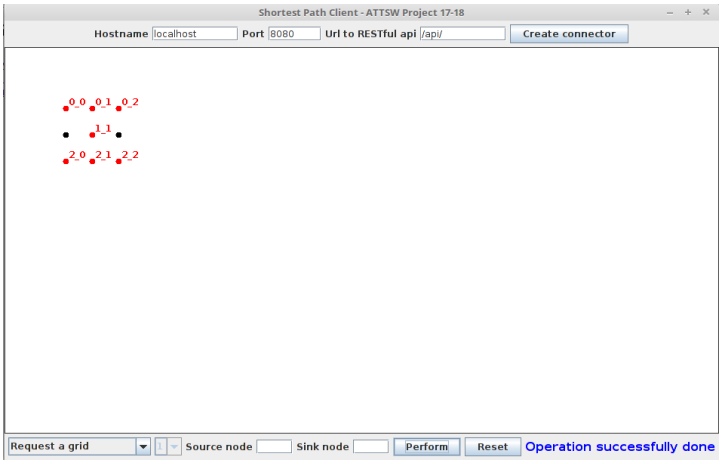

Figura 3: Richiedi griglia con id=1

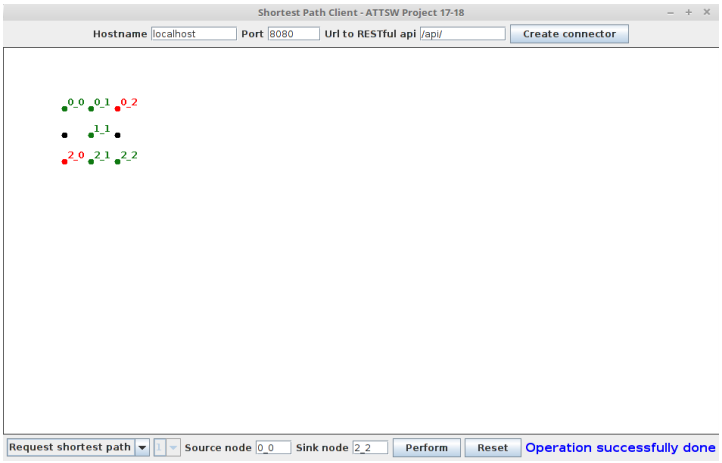

Figura 4: Richiedi il cammino minimo nella griglia 1 da 0,0 a 2,2

Implementazione dell'interfaccia

L'interfaccia fa uso del framework Swing. Essa è gestita da una classe GUI che ha al suo interno un'istanza di JFrame, una classe di Swing che rappresenta una finestra come comunemente intesa dall'utente. Come visto in precedenza, il frame è diviso in tre sezioni:

- NORTH
- CENTER
- SOUTH

A ciascuna di questa sezione corrisponde un JPanel. JPanel è una classe di Swing che funziona come container di componenti come pulsanti, barre di inserimento testuale, menù, ecc. La classe JPanel può essere estesa col fine di riscrivere il suo metodo *paintComponent(Graphics g)*, il metodo delegato al disegno del container. Questo è quanto avviene per il JPanel corrispondente al centro della finestra; infatti quest'ultimo è un'istanza della classe GUIpanel e rappresenta il punto centrale di tutta l'interfaccia grafica realizzata.
Un GUIpanel è un oggetto che ha al suo interno una matrice di punti indicizzati (ciascun punto è istanza della classe Point del medesimo framework) che possono essere creati e disegnati con differenti colori. Questa struttura si adatta al nostro sistema in quanto per visualizzare una griglia, oppure un cammino al suo interno, è sufficiente colorare i punti che fanno parte della griglia rispettivamente di colore rosso e di colore verde. I punti che non fanno parte della griglia e non sono indicizzati dalla matrice ricevuta dal server verranno disegnati come punti *hidden* ossia dello stesso colore dello sfondo del container, cosicchè possano apparire all'utente come nascosti.

4

PROBLEMI INCONTRATI

In questo capitolo parleremo dei problemi che abbiamo incontrato sia durante la stesura del codice che durante l'utilizzo delle tecniche oggetto di questo corso.

USO DI DUE DIFFERENTI DATABASE

Uno dei problemi che abbiamo riscontrato è stato utilizzare due differenti Database: MongoDB e MySQL. Questa decisione implementativa ci ha messo di fronte a un primo problema: come poter separare l'implementazione dei repository Mongo e MySQL senza dover creare necessariamente due differenti interfacce IGridService. Questo problema è derivato dalla necessità di avere una singola interfaccia che possa interagire con due differenti database, dove per differenti si intende una differenza in questo caso anche molto sostanziale in quanto MongoDB e MySQL fanno parte di due paradigmi opposti.
Permettere di avere una sola interfaccia che possa evolversi parallelamente rispetto a come si evolve l'implementazione di una parte dei suoi metodi è l'intento del design pattern *bridge* che è stato applicato delegando a un oggetto IServiceImplementor l'esecuzione delle operazioni CRUD sui database; esistono poi due implementazioni concrete di IServiceImplementor:

- MongoImplementor
- MySqlImplementor

A questo punto tuttavia si è presentato un nuovo problema relativo alla scelta di quale dei due utilizzare quando Spring Boot deve iniettare le dipendenze all'interno delle classi. Questo problema è stato facilmente risolto col meccanismo dei Profiles. Sono stati creati due differenti profili ("mongo") e ("mysql") ciascuno dei quali è progettato per utilizzare rispettivamente MongoImplementor e MySqlImplementor.
Questi profili sono utilizzati dai test mediante l'annotazione *@ActiveProfiles()* mentre il file Application.properties specifica qual'è il profilo che dovrà essere utilizzato dall'applicazione a runtime.

INTEGRATION TEST SU MAC

Abbiamo avuto un problema relativo all'esecuzione degli Integration Test dell'interfaccia grafica del client su Mac OS X. Il problema è stato risolto includendo nel POM un modulo della libreria AssertJ che era richiesto a run-time dal sistema Mac. Per essere poi sicuri che successive modifiche continuassero a non creare problemi su Mac OS, ne abbiamo abilitato la relativa build su travis grazie al meccanismo delle build *matrix*.

BUILD MAC SU TRAVIS

L'ultimo problema che abbiamo avuto è stato la configurazione del server MySQL e MongoDB su Travis quando veniva lanciata la build su Mac OS X. Abbiamo provveduto a risolvere questo problema configurando Travis in modo tale da aggiornare mediante il comando