

Advanced Techniques and Tools for Software Development

SISTEMA CLIENT-SERVER PER LA RICERCA
DEL CAMMINO MINIMO IN UN GRAFO
CON TOPOLOGIA A GRIGLIA

TOMMASO PUCCHETTI, ALEX FOGLIA, FRANCESCO SECCI

Anno Accademico 2017-2018

Tommaso Puccetti, Alex Foglia, Francesco Secci: *Sistema client-server per la ricerca del cammino minimo in un grafo con topologia a griglia*, Corso di Laurea Magistrale in Informatica, © Anno Accademico 2017-2018

INDICE

1	Sistema implementato	9
1.1	Panoramica ad alto livello	9
2	Tecniche e framework utilizzati	13
2.1	Code versioning	13
2.2	Dependencies and build tool	13
2.3	Continuos integration	14
2.4	Frameworks	14
2.4.1	Spring Boot	14
2.4.2	Framework di test	15
2.4.3	Http Client	16
2.4.4	GUI	16
2.4.5	End to end test	16
3	Design e scelte implementative	17
3.1	Server	17
3.1.1	Rappresentazione delle griglie	18
3.1.2	REST Controller	19
3.1.3	Web Controller	19
3.1.4	Algoritmo di ricerca del cammino minimo	22
3.2	Client	23
3.2.1	Interfaccia utente	24
4	Problemi incontrati	27
4.1	Integration test su Mac	27
4.2	Build Mac del Server su Travis	27
4.3	End to end test	27
4.4	Deployment su heroku	28
4.5	Utilizzo di due differenti database	28
4.6	Mutation testing con PitClipse	29
4.7	Testing dell'interfaccia grafica	29

ELENCO DELLE TABELLE

Tabella 1	Matrice A	10
Tabella 2	Cammino minimo relativo ad A	10

ELENCO DELLE FIGURE

Figura 1	Schema UML-Like	11
Figura 2	Home page della web-app	21
Figura 3	Form per l'inserimento di una nuova griglia	21
Figura 4	Lista delle griglie salvate nel database	22
Figura 5	Form per il cancellamento di una griglia	22
Figura 6	Interfaccia utente	24
Figura 7	Interfaccia utente	24
Figura 8	Interfaccia utente	25
Figura 9	Interfaccia utente	25

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 $A \in \mathbb{N}^{n \times n}$ 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 la griglia che A rappresenta.

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 cammino 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 un database MongoDB contenente le matrici che rappresentano particolari griglie.

Il Server deve poter essere in grado di gestire tre tipi di richieste pervenibili attraverso la sua interfaccia *REST*: recuperare tutte le griglie salvate nel 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 sia un normale browser, oppure il Client java 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.

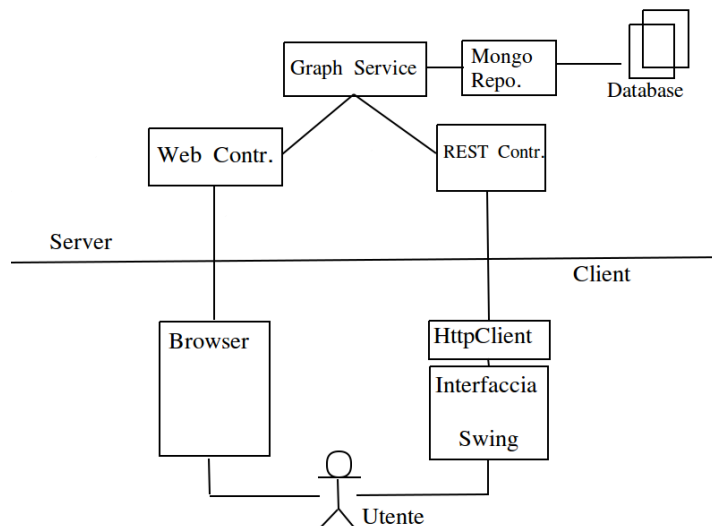


Figura 1: Schema che mostra il funzionamento del sistema implementato così come visto dall'utente

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*. La repository remota utilizzata è disponibile per la consultazione al seguente URL:

- <https://github.com/alexfoglia1/attsw-server>

L'utilizzo di *git* si è rivelato particolarmente utile per il lavoro in gruppo, grazie al meccanismo delle *pull requests*. Il workflow che abbiamo seguito prevedeva di aprire un ramo ogni qualvolta un membro del gruppo volesse implementare 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 sia gli unit test che gli integration test ed è proprio quest'ultima build quella che effettuiamo anche in locale. Dal momento che il codice del progetto si articola in 3 moduli (Client, Server, End2End), il comando di build viene eseguito nella directory del parent pom che accomuna i sopracitati moduli.

CONTINUOS INTEGRATION

Le repository elencate in 2.1 sono collegate al server di Continuous 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. Per ricreare l'ambiente *Travis* viene configurato in modo tale da usare il servizio Docker, ma è solo durante la build che viene lanciato il container Docker di MongoDB, in quanto il profilo failsafe, che deve eseguire gli integration test, è stato appositamente configurato per lanciare il container Docker legandone il relativo plugin alla fase *pre-integration-test*. 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 sui server di *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 offre inoltre una libreria che implementa il design pattern *Depencency Injection*, reso necessario dal meccanismo di *Inversion of Control* sottostante una qualsiasi

web-app Java. 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'istanziamento degli oggetti a run-time.

Per questo motivo, Spring Boot, contestualmente al lancio dell'applicativo carica un particolare *application-context* (contesto applicativo) il quale è l'insieme di tutti i bean che dovranno essere iniettati negli oggetti da essi dipendenti. Ogni bean iniettato è necessariamente un oggetto *singleton*. 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.

Scelta del database

Spring Boot possiede librerie per gestire un ampio insieme di database; noi abbiamo scelto MongoDB in quanto funzionale alle necessità dell'applicativo che intendevamo realizzare. MongoDB è un database NoSQL, questo implica che le entità presenti al suo interno non devono rispettare alcun vincolo di integrità relazionale, e questo ha particolarmente senso dal momento che nel database sono salvate istanze di griglie relazionalmente slegate l'una dall'altra. MongoDB è infine molto semplice da utilizzare ed integrare in una web-app.

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 sul database. Per testare questa interfaccia, che è necessariamente un'interfaccia HTML, abbiamo utilizzato il framework HtmlUnit. Per mockare le dipendenze delle classi durante il processo di test unitario abbiamo utilizzato il framework Mockito sia per il Client che per il Server.

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).

End to end test

Gli end to end test sono stati implementati usando Selenium per simulare la comunicazione browser-server; mentre Cucumber è il framework utilizzato per implementare gli end to end test secondo una tecnica BDD.

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 utente 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 all'utente 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.

IGridService viene estesa da una classe concreta (*ConcreteGridService*) che implementa direttamente l'algoritmo di ricerca del cammino minimo,

delegando tutte le operazioni relative alla lettura e scrittura dal e sul database all'interfaccia `IMongoRepository<DatabaseGrid,String>`, estensione della generica interfaccia di Spring Boot `MongoRepository<T,S>`. Questa interfaccia definisce la mappatura tra oggetti `DatabaseGrid` salvati sul database e le stringhe ID che li identificano, mentre l'implementazione dei metodi di lettura e scrittura è definita dal bean iniettato dal framework a run-time (il quale è disponibile di default).

L'intero sviluppo del sistema client-server ha seguito la metodologia test-driven.

Nei classici modelli di sviluppo software a cascata, o a V, prima di poter scrivere il codice è necessario definire i requisiti funzionali e non funzionali del sistema da implementare, e i successivi test hanno lo scopo di verificare che il comportamento del sistema sia coerente rispetto ai requisiti derivati dall'analisi effettuata a priori. Solo dopo aver verificato che un certo sistema soddisfi le proprietà imposte dai requisiti può seguire il processo di *validazione*. In tale fase, lo scopo è quello di validare un sistema sulla base delle funzionalità attese dal cliente/committente. Lo sviluppo test-driven accoppia le fasi di analisi dei requisiti e implementazione del codice, infatti scrivere i test prima del codice significa definire il requisito sotto forma di codice Java, e poi realizzare qualcosa che possa soddisfare il requisito appena definito. In quest'ottica, i test case che abbiamo scritto rappresentano altresì la definizione di *tutti* i requisiti funzionali del sistema implementato.

Rappresentazione delle griglie

Un oggetto griglia salvato nel database, viene visto dall'interfaccia `IGrid-Service` 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 n che rappresenta il numero massimo di righe e colonne che devono essere lette dalla matrice wrappata per estrapolarne la griglia.

Nonostante questa scelta sembra perdere di generalità, in quanto considera solamente matrici quadrate, in realtà se volessimo rappresentare una griglia che non sia in generale quadrata, sarebbe sufficiente porre a 0 le righe o le colonne che non si vuole facciano parte del grafo.

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/` → restituisce al client una lista JSON di stringhe rappresentanti ciascun ID delle griglie salvate nel database.
- `/api/grid[i]` → restituisce al client l'oggetto griglia avente `id='i'` (serializzato come oggetto JSON);
- `/api/path[SOURCE]TO[SINK]IN[ID]` → 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.

La sintassi di un nodo, affinché sia interpretabile dal server, deve rispettare la seguente espressione regolare:

$$[0-9]^+_[0-9]^+$$

La prima cifra rappresenta l'indice di riga del nodo che si vuole considerare, mentre la seconda rappresenta l'indice di colonna.

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:

- `/` → dalla root si accede a una pagina mediante la quale il client sceglie un particolare servizio del Web Controller;
- `/viewdb` → viene renderizzata una pagina HTML nella quale sono riportate tutte le griglie presenti nel database;

- /addtable → viene renderizzata una pagina HTML che prevede l'immissione tramite un form di una griglia all'interno del database;
- /remtable → come addtable, ma il form richiede solo l'inserimento dell'id della griglia che si intende cancellare.

Manage Database

- [View contents](#)
- [Add table](#)
- [Remove table](#)

Figura 2: Home page della web-app

Input

Size:

Content:

[Go back](#)

Figura 3: Form per l'inserimento di una nuova griglia

Grids

ID	N
5a82fb5d656a730004a97fee	3
Go back	

Figura 4: Lista delle griglie salvate nel database

Input

Id to remove:

[Go back](#)

Figura 5: Form per il cancellamento di una griglia

Algoritmo di ricerca del cammino minimo

Per implementare la ricerca del cammino minimo in una griglia rappresentata dalla matrice A , la classe concreta che implementa `IGridService` (`ConcreteGridService`) opera nel modo seguente:

- Per ogni nodo i,j di A tale che $A[i][j] > 0$ aggiungi il nodo i,j a una lista *nodes* di stringhe nella forma ' i_j ';
- Per ogni nodo n di *nodes*:
 - Controlla quali vicini ha n (controllando gli indici di riga e colonna come specificato nel capitolo 1)

- Se n ha un vicino n' , allora aggiungi a una lista *edges* di vettori di stringhe, l'arco $n \rightarrow n'$, dove ciascun arco e è un vettore di dimensione 2 tale per cui $e[0] = n$ ed $e[1] = n'$.

A questo punto il grafo è completamente rappresentato dalla lista dei suoi nodi e dalla lista dei suoi archi, ed è possibile applicarvi il seguente algoritmo di visita per ricercare il cammino minimo tra due nodi source e sink.

```

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 corrente:
                se nodi_visitati non contiene v:
                    coda.add(corrente);
                    nodi_visitati(v,true);
                    nodi_precedenti.add(v,corrente);
            fine se
        fine per
    fine se
    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.

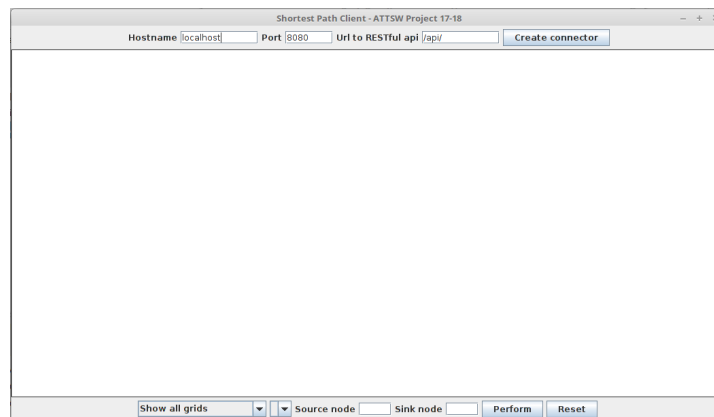


Figura 6: Interfaccia del Client all'avvio del sistema

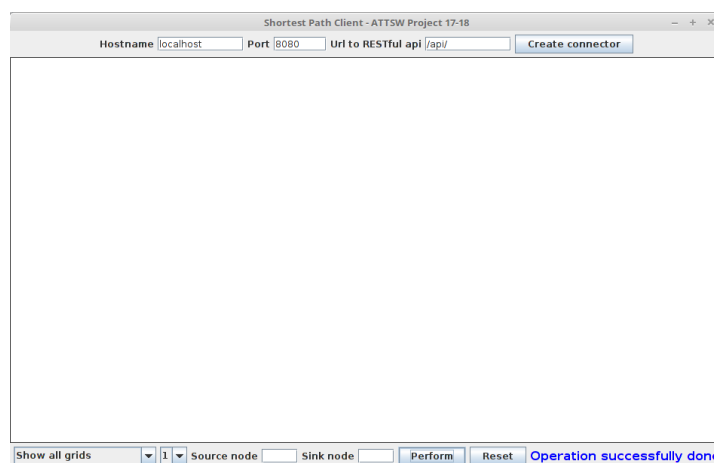


Figura 7: Richiedi tutte le griglie al Server

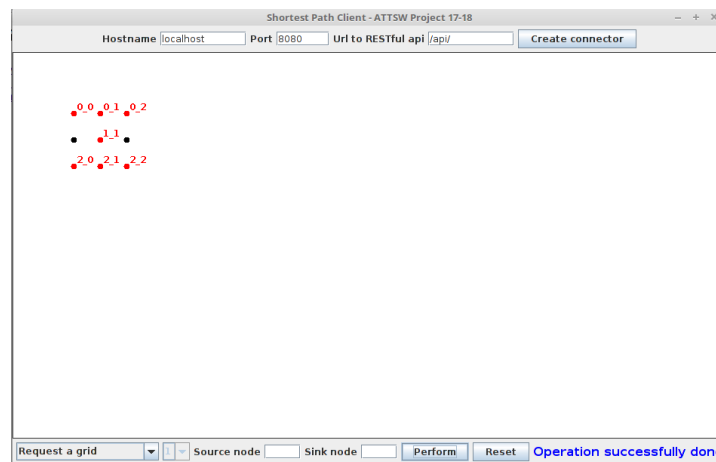


Figura 8: Richiedi griglia con id=1

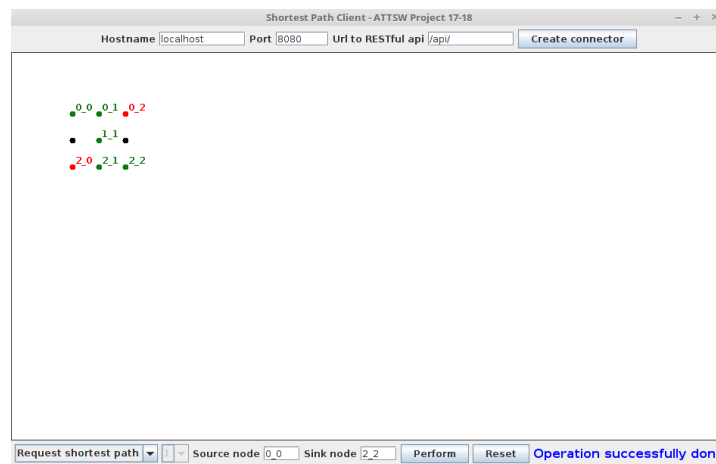


Figura 9: Richiedi il cammino minimo nella griglia 1 da 0,0 a 2,2

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. Infine, i nodi indicizzati dalla matrice ricevuta dal server che non fanno parte della griglia, sono visualizzati come punti di colore nero.

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.

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 DEL SERVER SU TRAVIS

Come riportato nella documentazione di Travis, quest'ultimo non supporta l'utilizzo di Docker sulle macchine remote quando impostate per utilizzare Mac OS; pertanto siamo stati costretti a disabilitare la build su Mac per quanto riguarda il server.

END TO END TEST

Per realizzare gli end to end test abbiamo scritto un Junit test case che utilizzasse un server non mockato. Tuttavia questa particolare circostanza impone che prima del lancio del test venga eseguito il server in locale, cosa non implementabile in automatico su Travis. Per risolvere questa problematica abbiamo escluso la classe riguardante gli end to end test dalla build di maven, ma l'abbiamo lasciata a disposizione per il suo lancio direttamente da eclipse. Al fine di verificare che gli end to end

test passassero anche quando l'applicazione Server non risiede esclusivamente sulla stessa macchina locale in cui si lancia il Client (localhost) abbiamo eseguito il deploy dell'applicazione su heroku (all'indirizzo <http://attsw-server.herokuapp.com>) in modo tale da poter inserire questo URL negli end to end test al posto del semplice localhost.

UTILIZZO DI DUE DIFFERENTI DATABASE

Abbiamo provato anche ad integrare un database MySQL all'interno del Server, tuttavia in corso d'opera sono sorti numerosi problemi le cui soluzioni avevano complicato troppo la struttura delle classi. I principali problemi incontrati sono stati:

- Incongruenza di tipi: il tipo java intero viene mappato su un database MySQL come un long a 64 bit, mentre non esiste il tipo di dato matrice. Questo fatto ci aveva costretto a creare una seconda classe, compatibile con MySQL, che rappresentasse un'istanza di una griglia all'interno del database MySQL.
- Differenti repository Spring: l'interfaccia IGridService vede esclusivamente DatabaseGrid, oggetti direttamente mappati all'interno del database Mongo tramite l'interfaccia nativa IMongoRepository. Questa interfaccia ha un suo equivalente per i database MySQL (CrudRepository) ma l'utilizzo dell'una e dell'altra interfaccia (IMongoRepository, CrudRepository) per le operazioni con il database aveva notevolmente complicato la struttura di IGridService, aggiungendovi una dipendenza esplicita da un'interfaccia *Implementor*, da noi realizzata, la quale astrae IMongoRepository e CrudRepository.
- Testing del database: non siamo riusciti a testare le operazioni sul database quando effettuate su MySQL, infatti, sebbene a runtime gli oggetti venivano correttamente scritti sul database e letti dal database, durante i test le operazioni di scrittura mediante repository non producevano alcuna istanza all'interno del database. L'unica soluzione che siamo riusciti a trovare è stata quella di sostituire l'utilizzo della repository MySQL con l'utilizzo diretto di una query sul database; soluzione giudicata come fortemente sbagliata in quanto non utilizzava gli strumenti di Spring Boot.

Per risolvere questi problemi abbiamo fatto un refactoring completo del codice su una nuova repository git. Approfittando di ciò abbiamo ripulito

non solamente il codice dall'uso di MySQL, ma abbiamo avuto la possibilità di realizzare meglio, grazie all'esperienza maturata in precedenza, anche le altre funzioni già realizzate sulla vecchia repository.

MUTATION TESTING CON PITCLIPSE

Per effettuare mutation testing sugli unit-test abbiamo utilizzato il framework PitClipse, tuttavia quest'ultimo quando genera i mutanti del SUT, lo fa mutando anche classi che non fanno parte del SUT propriamente inteso per gli unit test. Questo problema ha fatto sì che esistessero mutanti sopravvissuti nei report di PitClipse, ma tutti i mutanti sopravvissuti sono mutanti di classi che non sono *under-test* nel test case in questione.

TESTING DELL'INTERFACCIA GRAFICA

Per verificare che quando doveva essere disegnato un punto nel pannello centrale del Client (GUIpanel), questo venisse effettivamente disegnato, abbiamo voluto verificare che venisse chiamato il metodo `fillOval` coi giusti argomenti su un oggetto `Graphics`. Mockito tuttavia permette di effettuare asserzioni sull'invocazione di un metodo solo su oggetti mockati o oggetti spiati. A run-time, l'oggetto `Graphics` usato da un `JPanel` per disegnare il proprio contenuto, è istanza di una particolare sottoclasse finale di `Graphics`, quindi quest'ultimo non può essere né mockato né spiato. Per risolvere questa problematica abbiamo creato una classe `Wrapper` di `Graphics` che estende `Graphics`, spiato un'istanza di quest'ultima, e nei test abbiamo invocato il metodo `paintComponent(Graphics)` (metodo delegato al disegno del pannello) passandovi l'istanza spiata. In questo modo abbiamo potuto verificare che venissero invocati correttamente i metodi `fillOval` all'interno della classe `GUIpanel`.

Infine, dover scrivere test sull'interfaccia grafica ci ha costretto a modificare l'interfaccia delle classi delegate alla renderizzazione grafica dei contenuti ricevuti dal REST controller del server: sono stati creati dei metodi non-privati per poter accedere alle informazioni necessarie al fine di poter scrivere asserzioni coerenti con il comportamento atteso dal sistema.

END TO END TESTING

Gli End to End test sono stati motivo di un importante cambiamento nella struttura del progetto. Inizialmente avevamo previsto di realizzare Client e Server come due progetti separati ed inserire gli EndToEnd come test di integrazione del Client. Questo si prestava poco bene al meccanismo di build automation (di fatto tali test erano esclusi dalla build) dunque abbiamo deciso di rifattorizzare l'intero progetto; creando un parent-pom avente come moduli il Server, il Client, e gli End to End test.

Al fine di poter popolare e resettare il database abbiamo creato due nuovi end point Rest, che ci consentono di effettuare facilmente il testing dell'applicazione nel caso in cui fossero presenti griglie nel database senza dover fare procedure strane per popolarlo; ed altresì ci consentono di ripulire facilmente il database al termine di ogni test.

Per testare la comunicazione mediante GUI del nostro Client con il Server abbiamo usato, come per i test dell'interfaccia grafica, la libreria AssertJ, mentre per testare la comunicazione Browser con il server abbiamo utilizzato congiuntamente Selenium e Cucumber.