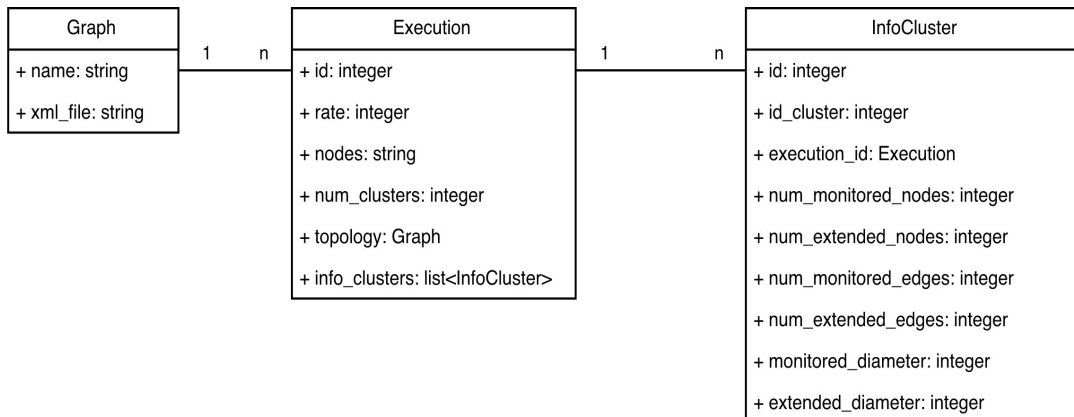


Documentazione Interfaccia REST

Class Diagram



C

come si evince dal Class Diagram di cui sopra le entità prese in considerazione sono 3. Graph, Execution e InfoCluster. Graph rappresenta la topologia caricata dall'utente, tale oggetto è composto da due campi di tipo string che sono rispettivamente il nome e il contenuto del file in formato xml. Execution, invece, rappresenta l'esecuzione dello script python lanciato per ogni topologia caricata dall'utente. Questo oggetto presenta come campi: un id integer, il rate, sempre integer, che indica la percentuale random di nodi selezionati dall'utente su cui applicare lo script python, una stringa nodes che indica le singole interfacce selezionate manualmente dall'utente su cui applicare lo script python, un oggetto topology che rappresenta il Graph caricato e due campi num_clusters e info_clusters che rappresentano rispettivamente il numero di cluster generati al termine dell'esecuzione, di tipo integer, e le info specifiche per ogni cluster; quest'ultimo è una lista di oggetti InfoCluster. InfoCluster invece contiene un id integer, un execution_id, sempre integer, che si riferisce all'id dell'esecuzione in esame e altri 7 campi di tipo integer che sono derivati dal file di output in formato json generato al termine dell'esecuzione dello script. La relazione tra l'oggetto Graph ed Execution è 1:N, ossia per ogni oggetto graph esistono più oggetti execution e per ogni oggetto execution esiste un unico oggetto graph. La relazione tra l'oggetto Execution e InfoCluster è 1:N perchè per ogni esecuzione posso generare da 1 a N clusters.

MySQL

Per la creazione del database è stato scelto MySQL. Il link per effettuare il download è :<https://www.mysql.com/it/products/community/>. Tra le varie edizioni disponibili è stata scelta MySQL Community Edition perchè a differenza delle altre presenti, è sottoposta a licenza GPL, ed è quindi gratuitamente scaricabile. Mette comunque a disposizione tutte le componenti principali: il server ma anche MySQL Cluster e MySQL Fabric.

Un'installazione di MySQL Community Edition per Windows si compone di un certo numero di strumenti:

- 1)MySQL Server: il vero e proprio servizio per la gestione dei database;
- 2)MySQL for Excel: plugin per Excel per la gestione di tabelle MySQL tramite fogli di calcolo;
- 3)MySQL Notifier: una piccola utility che permette di monitorare i servizi MySQL dal proprio desktop Windows;
- 4)MySQL Workbench: strumento unico di accesso ai dati e di monitoraggio;
- 5)MySQL For Visual Studio: un plugin di Visual Studio che permette ai programmatori .NET di lavorare ai propri database senza abbandonare l'IDE di sviluppo;
- 6)Connectors: driver per l'integrazione di database MySQL nei propri programmi.

Tuttavia gli unici strumenti utilizzati nel caso in esame sono MySql Workbench, per la creazione del DB e delle tabelle e MySql Server.

Le tabelle create sono 3 rispettivamente graphs, executions e infoclusters:

```
CREATE TABLE `graphs` (  
  `name` varchar(255) NOT NULL,  
  `xml_file` blob,  
  PRIMARY KEY (`name`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

La tabella graph presenta un campo name di tipo varchar che è primary key e un campo xml_file di tipo blob che contiene il contenuto del file in formato xml.

```
CREATE TABLE `executions` (  
  `id` int(11) NOT NULL,  
  `rate` int(11) DEFAULT NULL,  
  `num_clusters` int(11) DEFAULT NULL,  
  `nodes` varchar(255) DEFAULT NULL,  
  `topology` varchar(255) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `topology` (`topology`),  
  CONSTRAINT `executions_ibfk_1` FOREIGN KEY (`topology`) REFERENCES `graphs` (`name`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

La tabella executions, invece, presenta un campo id di tipo int che è primary key e un campo topology di tipo varchar su cui è applicato un vincolo di chiave esterna con il campo name della tabella graphs.

```
CREATE TABLE `infoclusters` (
  `id` int(11) DEFAULT NULL,
  `num_monitored_nodes` int(11) DEFAULT NULL,
  `num_extended_nodes` int(11) DEFAULT NULL,
  `num_monitored_edges` int(11) DEFAULT NULL,
  `num_extended_edges` int(11) DEFAULT NULL,
  `monitored_diameter` int(11) DEFAULT NULL,
  `extended_diameter` int(11) DEFAULT NULL,
  `execution_id` int(11) DEFAULT NULL,
  `id_cluster` int(11) NOT NULL,
  `idx` int(11) DEFAULT NULL,
  PRIMARY KEY (`id_cluster`),
  KEY `execution_id` (`execution_id`),
  CONSTRAINT `infoclusters_ibfk_1` FOREIGN KEY (`execution_id`) REFERENCES `executions` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

La tabella infoclusters invece presenta un campo id di tipo int che è primary key e un campo execution_id di tipo int su cui è applicato un vincolo di chiave esterna con il campo id della tabella executions.

Di seguito lo schema della tabella graphs.

Column	Type	Default Value	Nullable	Character Set	Collation	Privileges
name	varchar(255)		NO	utf8	utf8_general_ci	select,insert,update,references
xml_file	blob		YES			select,insert,update,references

Di seguito lo schema della tabella executions.

Column	Type	Default Value	Nullable	Character Set	Collation	Privileges
id	int(11)		NO			select,insert,update,references
rate	int(11)		YES			select,insert,update,references
num_clusters	int(11)		YES			select,insert,update,references
nodes	varchar(255)		YES	utf8	utf8_general_ci	select,insert,update,references
topology	varchar(255)		YES	utf8	utf8_general_ci	select,insert,update,references

Di seguito lo schema della tabella infoclusters.

Column	Type	Default Value	Nullable	Character Set	Collation	Privileges
id	int(11)		YES			select,insert,update,references
num_monitored_nodes	int(11)		YES			select,insert,update,references
num_extended_nodes	int(11)		YES			select,insert,update,references
num_monitored_edges	int(11)		YES			select,insert,update,references
num_extended_edges	int(11)		YES			select,insert,update,references
monitored_diameter	int(11)		YES			select,insert,update,references
extended_diameter	int(11)		YES			select,insert,update,references
execution_id	int(11)		YES			select,insert,update,references
id_cluster	int(11)		NO			select,insert,update,references
idx	int(11)		YES			select,insert,update,references

REST Service Design

Siccome l'architettura usata è REST, sappiamo che un concetto importante in REST è l'esistenza di risorse (fonti di informazioni), a cui si può accedere tramite un identificatore globale (un URI). Per utilizzare le risorse, le componenti di una rete (componenti client e server) comunicano attraverso una interfaccia standard (HTTP) e si scambiano rappresentazioni di queste risorse (il documento che trasmette le informazioni).

Le risorse sono gli elementi fondamentali su cui si basano i Web Service RESTful, a differenza dei Web Service SOAP-oriented che sono basati sul concetto di chiamata remota.

Per risorsa si intende un qualsiasi oggetto su cui è possibile effettuare operazioni. Per fare un parallelo con la programmazione ad oggetti possiamo dire che una risorsa può essere assimilata ad una istanza di una classe. Ciascuna risorsa deve essere identificata univocamente ed il meccanismo più naturale per individuare una risorsa è dato dal concetto di URI. Per cui nel caso in esame la risorsa principale è Graph. Questa presenta al suo interno la sottorisorsa Execution che, a sua volta, presenta al suo interno la sottorisorsa InfoCluster.

Le URI individuate sono le seguenti :

- | | |
|-----------------------------------------|-----------------------------------------|
| 1)GET /graphs | 200 OK |
| 2)GET /graphs/{name} | 200 OK/404 Not Found |
| 3)GET /graphs/{name}/executions/{id} | 200 OK/404 Not Found |
| 4)GET /graphs/{name}/executions | 200 OK |
| 5)POST /graphs | 201 Created/404 Not Found/409 Conflict |
| 6)POST /graphs/{name}/executions | 201 Created/404 Not Found /409 Conflict |
| 7)DELETE /graphs/{name} | 200 OK |
| 8)DELETE /graphs/{name}/executions/{id} | 200 OK |

La 1) ritorna tutti i grafi caricati dall'utente presenti sul server. In caso di successo ritorna 200 OK. La 2) ritorna il grafo in base al nome. Anche questa in caso di successo ritorna 200 OK altrimenti 404 not found. La 3) ritorna una execution identificata dal suo id ed associata ad un graph il cui nome è passato come parametro. Questa in caso di successo ritorna 200 OK altrimenti 404 not found. La 4) ritorna tutte le excutions disponibili per un dato graph. In caso di successo ritorna 200 OK. La 5) consente di creare un graph, per cui in caso di esito positivo ritorna 201 Created, altrimenti 404 Not found oppure 409 Conflict se la risorsa con lo stesso id è già presente sul server. Stessa cosa per la 6) solo che la post in esame in questo caso consente la creazione di una execution. Le ultime due operazioni sono due delete. La 7) consente di eliminare un graph dato il nome, invece la 8) consente di eliminare una execution associata ad un graph dato l'id dell'execution stessa. In caso di successo dell'operazione ritorna 200 OK. Tutte le operazioni sopra elencate possono ritornare 500 Internal Server Error in caso di errori presenti durante l'eleborazione della risorsa lato server.

Implementazione

I tools e framework usati per lo sviluppo dell'applicazione lato server e client sono :

- 1) Eclipse IDE for Java EE Developers (Neon Packages) : tool usato dagli sviluppatori Java per creare applicazioni Java EE e Web application. Integra Maven attraverso un apposito plug-in. Il link per effettuare il download è :<https://www.eclipse.org/downloads/packages/release/Neon/3>.

2)Apache Tomcat 7: è un web server (nella forma di contenitore servlet) open source sviluppato dalla Apache Software Foundation. Implementa le specifiche JavaServer Pages (JSP) e servlet, fornendo quindi una piattaforma software per l'esecuzione di applicazioni Web sviluppate in linguaggio Java. Il link per effettuare il download è :<https://tomcat.apache.org/download-70.cgi>

3)Integrazione di networkx in jython per eseguire script in python lato java. Scaricare networkx :<https://networkx.github.io/documentation/networkx-1.10/download.html>, scaricare jython 2.7.0 :<http://www.jython.org/downloads.html> e python 2.7.0 al link :<https://www.python.org/download/releases/2.7/>, estrarre i rispettivi contenuti in 3 cartelle differenti. Aggiungere python 2.7.0 e jython 2.7.0 alle variabili d'ambiente (in Windows). Su Windows eseguire il comando (jython -m pip install -U pip) per installare pip (il gestore dei pacchetti di python). Nella cartella source di networkx eliminare il file networkx/generators/atlas.py. Nel file networkx/readwrite/gml.py, rimuovere tutti i riferimenti a lib2to3 (dovuto ad un bug presente in Jython, lib2to3 non è disponibile). Alle righe 44-46, commentare tutti gli imports from lib2to3 e alla riga 75 cambiare :

```
rtp_fix_unicode = RefactoringTool(['lib2to3.fixes.fix_unicode'],  
                                  {'print_function': True})
```

in:

```
rtp_fix_unicode = None
```

Alla riga 145, nel try-except statement, rimuovere ParseError e TokenError. A questo punto è necessario tornare nella cartella source ed eseguire il comando: jython/pip install . A questo punto abbiamo integrato networkx in jython, quindi basta semplicemente caricare il file.jar in eclipse nel build path del progetto maven.

4)Jython versione 2.7.0. Prendere il file.jar presente nella cartella jython 2.7.0 e aggiungerlo al Build Path del progetto. Tasto destro sul progetto maven -> Build Path -> Configure Build Path -> Libraries-> Add External JARs-> Apply->OK.

5)Hibernate: è una piattaforma middleware open source usata per lo sviluppo di applicazioni Java, attraverso l'appoggio al relativo framework, che fornisce un servizio di Object-relational mapping (ORM) ovvero gestisce la persistenza dei dati sul database attraverso la rappresentazione e il mantenimento su database relazionale di un sistema di oggetti Java. Nell'ambito dello sviluppo di applicazioni web, tale strato software si frappone tra il livello logico di business o di elaborazione e quello di persistenza dei dati sul database (Data Access Layer).

6) Jersey: Jersey RESTful Web Services framework è un framework open source per lo sviluppo di RESTful Web Services in Java. Fornisce supporto per le API JAX-RS e funge da implementazione di riferimento JAX-RS (JSR 311 e JSR 339). I seguenti componenti fanno parte di Jersey: Core Server: per creare servizi RESTful basati su annotazione (jersey-core, jersey-server, jsr311-api), Core Client: per comunicare con i servizi REST (jersey-client), JAXB support , JSON support e il modulo di integrazione per Spring e Guice.

7) Jackson: un framework usato per la deserializzazione di una stringa in formato json in una classe java e per la serializzazione di una classe java in una stringa json.

8) Apache Maven: Apache Maven è un tool di gestione di progetti software basati su Java e build automation. Per funzionalità è simile ad Apache Ant, ma basato su concetti differenti. Maven usa un costrutto conosciuto come Project Object Model (POM); un file XML che descrive le dipendenze fra il progetto e le varie versioni di librerie necessarie nonché le dipendenze fra di esse. In questo modo si separano le librerie dalla directory di progetto utilizzando questo file descrittivo per definirne le relazioni. Altro vantaggio di Maven è il download automatico di librerie Java e plug-in Maven dai vari repository definiti scaricandoli in locale o in un repository centralizzato lato sviluppo. Questo permette di recuperare in modo uniforme i vari file JAR e di poter spostare il progetto indipendentemente da un ambiente all'altro avendo la sicurezza di utilizzare sempre le stesse versioni delle librerie

Dipendenze incluse nel pom.xml del progetto tesiclustering:

```
<dependencies>
  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.8</version>
  </dependency>
  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
    <version>1.8</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.30</version>
  </dependency>
  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.8</version>
  </dependency>
</dependencies>
```

```

<dependency>
  <groupId>com.sun.jersey.contribs</groupId>
  <artifactId>jersey-multipart</artifactId>
  <version>1.8</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.4</version>
</dependency>
<dependency>
  <groupId>commons-lang</groupId>
  <artifactId>commons-lang</artifactId>
  <version>2.6</version>
</dependency>
<dependency>
  <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.10</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>2.9.4</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.9.4</version>
</dependency>
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20171018</version>
</dependency>
</dependencies>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.3.5.Final</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-tools</artifactId>
  <version>4.3.5.Final</version>
</dependency>
<dependency>
  <groupId>dom4j</groupId>
  <artifactId>dom4j</artifactId>
  <version>1.6.1</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.6</version>
</dependency>
<dependency>
  <groupId>asm</groupId>
  <artifactId>asm</artifactId>
  <version>3.3.1</version>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
</dependencies>

```

9) Per il Front-End della Web Application ho usato Bootstrap, Ajax, JQuery. Bootstrap è un framework che contiene modelli in HTML CSS e JS per la creazione di pagine web dinamiche. Il link per effettuare il download è: <https://v4-alpha.getbootstrap.com/>. Basta estrarre il contenuto in una cartella specifica e importare le sottocartelle css e js sotto la cartella webapp del progetto Maven. Per utilizzarlo all'interno della pagina web includere i file nel tag head della pagina html.

```

<link href="css/bootstrap.min.css" rel="stylesheet" type="text/css"/>
<script type="text/javascript" src="js/bootstrap.min.js"></script>

```

Ajax, acronimo di Asynchronous JavaScript and XML, è una tecnica di sviluppo software per la realizzazione di applicazioni web interattive. Lo sviluppo di applicazioni HTML con AJAX si basa su uno scambio di dati in background fra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente. JQuery è una libreria JavaScript per applicazioni web, che consente di semplificare e uniformare la manipolazione degli elementi del DOM di una pagina HTML. Il link per effettuare il download è: <https://jquery.com/>. Scaricare la versione 3.3.1 e aggiungere il file jquery-3.3.1.js nella cartella webapp del progetto Maven. Per utilizzarlo all'interno della pagina web includere i file nel tag head della pagina html.

```

<script src="jquery-3.3.1.js"></script>.

```


Struttura del Servizio

Una volta estratto il progetto tesiclustering in una cartella è necessario importarlo in Eclipse. I passi sono i seguenti. File-> Import-> Existing Maven Project->Next-> Selezionare la cartella in cui è contenuto il progetto ->Finish. A questo punto è necessario aspettare alcuni minuti utili per il Building del progetto e il download di tutte le dipendenze Maven necessarie per l'applicazione. La struttura del service è la seguente:

```
it > tesiclustering_v1 [Workspace25-02-2018 master]
├── src/main/java
│   ├── it.polito.tesiclustering.controller
│   │   ├── ExecutionController.java
│   │   ├── GraphController.java
│   │   └── UserController.java
│   ├── it.polito.tesiclustering.dao
│   │   ├── ExecutionDao.java
│   │   ├── GraphDao.java
│   │   └── UserDao.java
│   ├── it.polito.tesiclustering.infrastructure
│   ├── it.polito.tesiclustering.model
│   │   ├── Execution.java
│   │   ├── ExecutionDTO.java
│   │   ├── Graph.java
│   │   ├── InfoCluster.java
│   │   ├── InfoClusterDTO.java
│   │   ├── Node.java
│   │   ├── Result.java
│   │   └── User.java
│   ├── it.polito.tesiclustering.model.exception
│   │   ├── DaoException.java
│   │   ├── ExecutionAlreadyCreatedException.java
│   │   ├── GraphAlreadyCreatedException.java
│   │   └── UserAlreadyCreatedException.java
│   ├── it.polito.tesiclustering.model.serializers
│   ├── it.polito.tesiclustering.service
│   │   ├── ExecutionService.java
│   │   ├── GraphService.java
│   │   ├── ResultFactory.java
│   │   ├── UserService.java
│   │   ├── Bics.graphml
│   │   ├── Geant2012.graphml
│   │   └── Singaren.graphml
├── src/main/resources
└── src/test/java
```

Il primo package in esame è `it.polito.tesiclustering.model`, in cui ci sono le risorse e le sottorisorse REST ossia l'Excution Model, il Graph Model e l'InfoCluster Model. Il Model rappresenta un object o un JAVA POJO che porta con sè delle informazioni. In aggiunta nel package in esame ci sono due classi DTO necessarie per evitare il problema dell'infinite recursive loop di Jackson, e quindi della ridondanza delle informazioni nel json. L'obiettivo è quello di inserire nel json, e quindi inviare al client, meno informazioni rispetto a quelle gestite lato DB. I models quindi presentano gli stessi campi delle entità sul DB più i relativi metodi getter and setter, costruttore e metodo `toString()`. Il secondo package è quello delle classi DAO, `it.polito.tesiclustering.dao`, in cui ci sono rispettivamente l'ExecutionDAO e il GraphDAO per l'accesso CRUD alle tabelle. Uno degli aspetti fondamentali quando si costruisce un'applicazione

è la persistenza dei dati, per realizzare la quale, nel mondo Java, esistono una varietà numerosa di API e framework, come JDBC o Hibernate.

Il framework utilizzato in questo progetto è Hibernate. Per rendere più facile l'accesso al DB, che in nostro caso è MySQL, è stato utilizzato il pattern DAO. Il DAO (Data Access Object) è un pattern architetturale per la gestione della persistenza: si tratta fondamentalmente di una classe con relativi metodi che rappresenta un'entità tabellare di un RDBMS, usata principalmente in applicazioni per stratificare e isolare l'accesso ad una tabella tramite query (poste all'interno dei metodi della classe) ovvero al data layer da parte della business logic creando un maggiore livello di astrazione ed una più facile manutenibilità. I metodi del DAO con le rispettive query dentro verranno così richiamati dalle classi della business logic. Il vantaggio relativo all'uso del DAO è dunque il mantenimento di una rigida separazione tra le componenti di un'applicazione, le quali potrebbero essere il "Modello" e il "Controllo" in un'applicazione basata sul paradigma MVC. HibernateUtil.java presente nel package it.polito.tesiclustering.infrastructure è la classe che consente di costruire la sessione di persistenza a partire dai file .hbm.xml. Questa istanzia l'oggetto SessionFactory responsabile dell'apertura della sessione verso il DB. In particolare in it.polito.tesiclustering.resources è presente il file hibernate.cfg.xml, che specifica tutto il necessario per identificare ed accedere ad un database, quindi driver JDBC, URI, username, password, dialetto SQL ecc. Inoltre il file hibernate.cfg.xml identifica altri file XML di configurazione che, a loro volta, dichiarano il mapping tra classi Java Bean e specifiche tabelle presenti nel database. Nel nostro caso questi file sono rispettivamente:

```
<mapping resource="graph.hbm.xml" />

<mapping resource="execution.hbm.xml" />

<mapping resource="infocluster.hbm.xml" />
```

Tutta la logica applicativa è contenuta nei service presenti nel package it.polito.tesiclustering.service. In particolare le classi sono ExecutionService e GraphService. Il service, quindi, incapsula tutta la business logic dell'applicazione. Per la risorsa graph sono implementati i seguenti metodi:

- 1) `getGraphByName()` : riceve come parametro il nome del grafo da caricare e restituisce il grafo corrispondente se tale risorsa è presente sul DB, altrimenti restituisce null. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.
- 2) `getGraphs()` : restituisce la lista dei grafi presenti sul DB. Se non ci sono grafi restituisce una lista vuota. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.
- 3) `createGraph()` : riceve come parametro l'oggetto Graph, se il nome del graph che si vuole creare è già presente sul DB viene lanciata una `GraphAlreadyCreatedException()`, altrimenti viene creato il grafo a partire dalla topologia. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.
- 4) `deleteGraph()` : riceve come parametro il nome del grafo che si desidera cancellare e procede nell'operazione. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.

Per la risorsa execution sono implementati i seguenti metodi:

- 1) `getExecutionByTopologyAndId()` : riceve come parametro il nome del grafo per il quale è stata generata l'execution e l'id dell'execution stessa. Ritorna l'execution richiesta se presente. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.
- 2) `getExecutionsByTopology()` : riceve come parametro il nome del grafo per il quale sono state generate le executions e restituisce la lista delle executions presenti sul DB. Se non ci sono executions restituisce una lista vuota. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.
- 3) `createExecution()` : riceve come parametro l'oggetto `Execution` e l'oggetto `Graph`, se l'execution con lo stesso id è già presente sul DB viene lanciata una `ExecutionAlreadyCreatedException()`, altrimenti viene lanciato lo script python che genererà i risultati che verranno integrati nell'oggetto execution creato. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.
- 4) `deleteExecution()` : riceve come parametro il nome del grafo cui si riferisce l'esecuzione e l'id dell'esecuzione stessa che si desidera cancellare e procede nell'operazione. Eventuali errori durante il recupero dell'informazione dal DB provocano una `DAOException`.

I controllers, presenti nel package `it.polito.tesicustering.controller`, gestiscono le richieste e le risposte HTTP secondo il paradigma architetturale REST. Nel `GraphController` sono gestite le operazioni:

- 1) `POST /graphs` : in caso di successo ritorna il grafo creato e una `Response 201 Created`, altrimenti se il grafo non è creato con successo ritorna `404 Graph not found`. Se il grafo è già presente ritorna `409 Conflict` e in caso di `DaoException` invece ritorna `500 Internal Server Error`.
- 2) `GET /graphs/{name}` : in caso di successo ritorna un graph e una `Response 200 OK`, altrimenti se il grafo non è presente ritorna `404 Graph not found`. In caso di `DaoException` invece ritorna `500 Internal Server Error`.
- 3) `GET /graphs` : in caso di successo ritorna la lista di grafi e una `Response 200 OK`, altrimenti in caso di `DaoException` invece ritorna `500 Internal Server Error`.
- 4) `DELETE /graphs/{name}` : ritorna `200 Ok` in caso di successo.

Mentre nell'`ExecutionController` :

- 5) `POST /graphs/{name}/executions` : in caso di successo ritorna l'execution creata e una `Response 201 Created`, altrimenti se l'execution non è stata creata con successo o il grafo non è presente ritorna `404 Execution not found`. Se l'Execution con lo stesso id è già presente ritorna `409 Conflict` e in caso di `DaoException` invece ritorna `500 Internal Server Error`.
- 6) `GET /graphs/{name}/executions/{id}` : in caso di successo ritorna una execution e una `Response`

200 OK, altrimenti se l'execution non è presente ritorna 404 Execution not found. In caso di DaoException invece ritorna 500 Internal Server Error.

7)GET /graphs/{name}/executions:in caso di successo ritorna la lista di executions e una Response 200 OK, altrimenti in caso di DaoException invece ritorna 500 Internal Server Error.

8)DELETE /graphs/{name}/executions/{id}:ritorna 200 Ok in caso di successo.

Interfaccia Client-Side

La WebApplication lato client è strutturata in questo modo. Ci sono essenzialmente 3 viste. Index.jsp, graphs.jsp e executions.jsp.

Clustering

Load the topology and calculate the clusters

Upload the topology

Choose a single topology that contains information about a specific ISP's network.

Scegli file

Nessun file selezionato

Load The file!

Random Selection

Insert a single value ranging from 10 to 100.

Submit

View all graphs

Exact Selection

Insert a list of interfaces.
(example=[[33,34,"in"],[33,34,"out"],[34,33,"in"],[34,33,"out"]]]).

Submit

View all executions

Clustering

View all the topology graphs and decide if delete one of these.

Graphs List

- [Bellcanada.graphml](#)
- [Belnet2006.graphml](#)
- [Ibm.graphml](#)
- [Uran.graphml](#)

Delete a graph by Name

Delete a topology graph from the list below by name.

Name Topology:

Delete

Back Home

Clustering

View all the executions of a graph and decide if delete one of these.

Executions List

Insert the name of graph to show its executions

Graph Name :

- 14 Num clusters: 4
- 15 Num clusters: 4
- 16 Num clusters: 3

Delete an execution by Id

Delete an execution from the list below by id.

Id :

Le funzioni ajax-jquery richiamate nelle viste sono definite e implementate nel main.js presente sotto la cartella WEB-INF insieme ai file.jsp. I file css and js invece sono sotto le rispettive cartelle css e js presenti sotto la cartella webapp. Il file web.xml, invece, è il web deployment descriptor per le web application, risiede sotto la cartella WEB-INF. Questo descrive come fare il deploy di una web application in un servlet container come Tomcat. Infatti al suo interno è specificato il servlet e il servlet mapping.

```
<servlet>
  <servlet-name>jersey-servlet</servlet-name>
  <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>com.sun.jersey.config.property.packages</param-name>
    <param-value>it.polito.tesiclustering</param-value>
  </init-param>
  <init-param>
    <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
    <param-value>>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>jersey-servlet</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Per l'applicazione client il procedimento è lo stesso. Una volta estratto il progetto client in una cartella è necessario importarlo in Eclipse. I passi sono i seguenti. File-> Import-> Existing Maven Project->Next-> Selezionare la cartella in cui è contenuto il progetto ->Finish. A questo punto è necessario aspettare alcuni minuti utili per il Building del progetto e il download di tutte le dipendenze Maven necessarie per l'application.

La struttura del client è la seguente:

```
✓ > client [Clientrepository master ↓1]
  ✓ src/main/java
    ✓ it.polito.client
      > ClusteringClient.java
      > HttpClient.java
        Aarnet.graphml
        Abilene.graphml
        Bics.graphml
        Geant2012.graphml
        Ibm.graphml
        Kdl.graphml
        Uran.graphml
      > it.polito.infrastructure
    ✓ it.polito.model
      > Execution.java
      > ExecutionDTO.java
      > ExecutionSerializer.java
      > Graph.java
      > GraphSerializer.java
      > InfoCluster.java
      > InfoClusterDTO.java
      > Node.java
      > Result.java
      > User.java
    src/test/java
  ✓ > src/main/test/clientTest
    ✓ > clientTest
      > > ClientTest.java
```

Nel package `it.polito.model` sono definiti i models. Gli stessi presenti lato server. Nel package `it.polito.client`, invece, ci sono due classi: rispettivamente `HttpClient.java` in cui sono definiti i metodi che poi sono richiamati nel `ClusteringClient.java`. Questo è essenzialmente un interprete di comandi basato su riga di comando. E' presente un menu e in base al comando inserito è possibile effettuare un'operazione diversa. Ad esempio `storegraph` permette di caricare un file il cui nome è passato come primo argomento da command line. `Listgraphs`, invece, permette di vedere tutti i grafi caricati sul server, `showgraph` permette di vedere un grafo in particolare il cui nome è passato come primo argomento da command line, `deletograph`, invece, permette di cancellare un grafo il cui nome è passato come primo argomento da command line. `Deleteexecution` invece richiede due argomenti, il nome del grafo e l'id dell'execution che si intende cancellare. `Listexecutions` invece permette di visionare tutte le executions presenti sul DB per quel determinato grafo passato come primo argomento da command line, `showexecution`, invece, richiede sempre due argomenti rispettivamente il nome del grafo e l'id dell'execution specifica. In ultimo ci sono `excuterate` e `executenodes` che necessitano di due argomenti: il primo è il grafo su cui si vuole eseguire lo script python e il secondo è, nel caso di `executerate`, un intero che indica la percentuale random di nodi selezionati, mentre per `executenodes`, una stringa in cui sono indicate esplicitamente le interfacce dei nodi da selezionare. Di seguito sono elencate le operazioni con i relativi output.

Menu

Per terminare usa Ctrl+z o digita Invio.

```
> menu
```

Comandi disponibili:

- storegraph
- listgraphs
- showgraph
- deletegraph
- deleteexecution
- executerate
- executenodes
- listexecutions
- showexecution
- quit
- menu

Per terminare usa Ctrl+z o digita Invio.

```
> |
```

Delete a graph

Per terminare usa Ctrl+z o digita Invio.

```
> deletegraph Geant2012.graphml
```

Cancellazione grafo in corso...

Output from Server

```
{ "result": "success" }
```

Per terminare usa Ctrl+z o digita Invio.

```
> |
```

Post a graph

```
> storegraph IBM.graphml
Invio grafo in corso...
Convert xml file inserted to string
<?xml version="1.0" encoding="utf-8"?><graphml xmlns='

Set name and xml_file in java object
Set name and xml_file in java object
Start serialization
End serialization
Output from Server ....

Per terminare usa Ctrl+z o digita Invio.
>
```

List of all graphs

```
Per terminare usa Ctrl+z o digita Invio.
> listgraphs
Ricerca grafi presenti sul server...
[{"name":"Bellcanada.graphml","xml_file":"<?xml version=\"1.0\" encoding=\"utf-8\"?>
Per terminare usa Ctrl+z o digita Invio.
> |
```

Show a graph

```
> showgraph IBM.graphml
Ricerca grafi disponibili sul server...
{"name":"IBM.graphml","xml_file":"<?xml version=\"1.0\" encoding=\"utf-8\"?><graphml
....
```

Delete an execution

```
Per terminare usa Ctrl+z o digita Invio.
> deleteexecution Belnet2006.graphml 10
Cancellazione esecuzione in corso...
Output from Server ....

{ "result": "success" }
Per terminare usa Ctrl+z o digita Invio.
> |
```

Show an execution

```
> showexecution Bellcanada.graphml 13
Mostra il risultato di una esecuzione.
Topologia: Bellcanada.graphml
ID: 13
{"id":13,"nodes":null,"num_clusters":2,"info_clusters":[{"id":0,"id_cluster":25,"monitored_diameter":3,
Per terminare usa Ctrl+z o digita Invio.
> |
```


List of all executions

```
> listexecutions Bellcanada.graphml
Ricerca esecuzioni disponibili sul server...
[{"id":13,"nodes":null,"num_clusters":2,"info_clusters":[{"id":0,"id_cluster":25,"monitored_diameter":3,
Per terminare usa Ctrl+z o digita Invio.
>
```

Execute a rate

```
> executerate Ibm.graphml 20
Invio execution in corso...
Output from Server ....

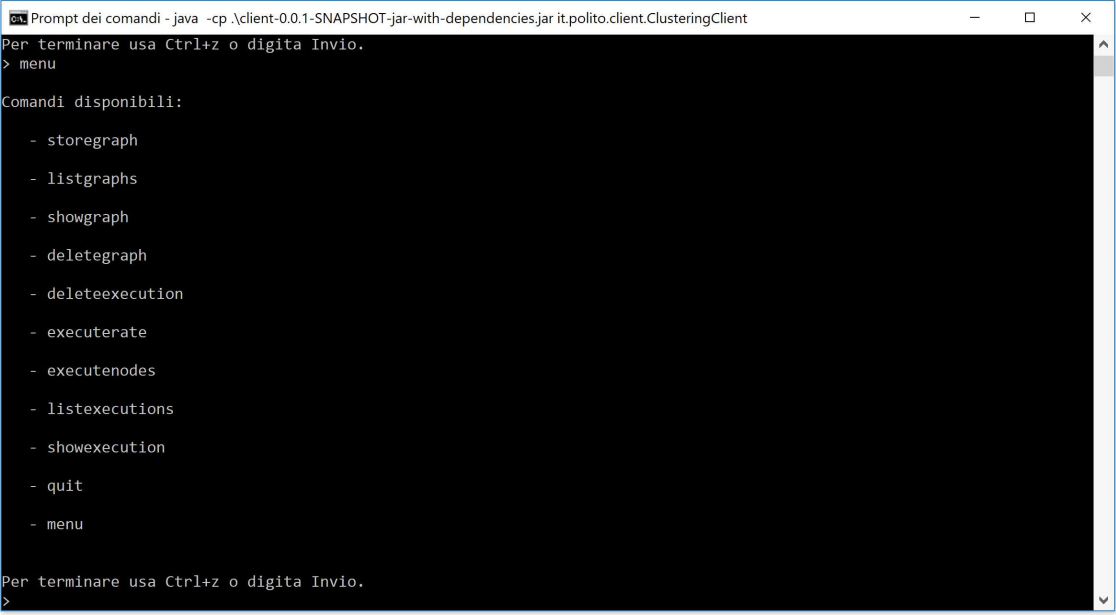
{"id":15,"nodes":null,"num_clusters":4,"info_clusters":[{"id":0,"id_cluster":31,"monitored_diameter":1,
Per terminare usa Ctrl+z o digita Invio.
> |
```

Execute nodes

```
> executenodes Ibm.graphml [[33,34,"in"],[33,34,"out"],[34,33,"in"],[34,33,"out"]]
Invio execution in corso...
Output from Server ....

{"id":16,"nodes":"[[33,34,\"in\"],[33,34,\"out\"],[34,33,\"in\"],[34,33,\"out\"]]", "num_clusters":3,"info_clusters":|
Per terminare usa Ctrl+z o digita Invio.
>
```

Sotto Windows, ad esempio, è possibile anche lanciare l'applicazione client dal prompt dei comandi. Basta semplicemente usare l'assembly plugin di Maven per generare il file .jar.



```
c:\> java -cp .\client-0.0.1-SNAPSHOT-jar-with-dependencies.jar it.polito.client.ClusteringClient
Per terminare usa Ctrl+z o digita Invio.
> menu

Comandi disponibili:

- storegraph
- listgraphs
- showgraph
- deletegraph
- deleteexecution
- executerate
- executenodes
- listexecutions
- showexecution
- quit
- menu

Per terminare usa Ctrl+z o digita Invio.
>
```

I passi da seguire per generare un executable jar con Maven sono i seguenti:

Per creare un jar eseguibile, non abbiamo bisogno di ulteriori dipendenze nel pom.xml. Abbiamo solo bisogno di creare un progetto Maven Java e avere almeno una classe con il metodo principale main(). Apache Maven Assembly Plugin consente agli utenti di aggregare l'output del progetto insieme alle sue dipendenze, moduli, documentazione e altri file in un unico pacchetto eseguibile. Bisogna semplicemente aggiungere il plugin sotto nel pom.xml:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <archive>
          <manifest>
            <mainClass>
              it.polito.client.ClusteringClient
            </mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Dobbiamo fornire le informazioni sulla classe principale, mainClass, nel nostro caso è it.polito.client.ClusteringClient. In questo modo il plug-in Maven Assembly copia automaticamente tutte le dipendenze richieste in un unico file jar. Nella parte descriptorRefs del codice di configurazione bisogna fornire il nome che verrà aggiunto al nome del progetto. L'output nel nostro caso verrà denominato client-0.0.1-SNAPSHOT-jar-with-dependencies.jar. Quindi per eseguirlo basta andare nella cartella in cui è contenuto il file jar e lanciare il comando:

```
java -cp .\client-0.0.1-SNAPSHOT-jar-with-dependencies.jar it.polito.client.ClusteringClient .
```