Relazione
SparqlAppAnalyzer

Un sistema per l'analisi e costruzione di grafi relativi all'analisi di endpoint

di:
Lavinia De Divitiis

Esame:

Security and Knowledge Management

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Sommario

1. Introduzione

Lo scopo di questo elaborato è stato quello di realizzare un'applicazione che sia in grado, dato un endpoint, di analizzarlo tramite l'uso di query SPARQL e di rappresentare il grafo relativo all'analisi realizzata sull'endpoint.

L'applicazione permette all'utente di inserire un nuovo endpoint da analizzare e di scegliere un endpoint dall'elenco per visualizzarne il grafo.

Dato il grafo, l'utente ha la possibilità di spostare nello spazio nodi singoli o l'intero grafo. Inoltre, è possibile anche poter effettuare lo zoom.

2. Tecnologie utilizzate

2.1. Lato Backend

Per lo sviluppo della mia applicazione lato backend, ho utilizzato come IDE Eclipse.

Come linguaggio di programmazione ho utilizzato Java con il framework Spring, inoltre per quanto riguarda le query SPARQL eseguite dall'applicazione ho utilizzato il framework Apache Jena.

Per quanto riguarda il database ho utilizzato un database MySQL.

2.2. Lato Frontend

Lato frontend, ho utilizzato un editor di testo, Sublime Text.

Come linguaggio di programmazione ho utilizzato Javascript con la libreria JQuery.

Per quanto riguarda la parte di grafica ho utilizzato: HTML, CSS e Bootstrap, in particolare ho utilizzato Pingendo Bootstrap.

Per la costruzione del grafo ho utilizzato la libreria d3.js.

3. Sviluppo

3.1. Lato Backend

Per quanto riguarda lo sviluppo dell'applicazione lato backend, come prima cosa, ho definito il file "application.properties" dove ho definito i parametri di configurazione del database, precedentemente creato. I parametri sono:

```
url=jdbc:mysql://localhost:3306/db_sparql
username=root
password=root
```

Successivamente ho creato il *Model*:

- Endpoint
- ClassStructure
- PredicateStructure
- Relations

Endpoint rappresenta la classe che modella l'entità endpoint, che contiene i seguenti attributi:

- Id: identificativo univoco
- Stato: indica lo stato di avanzamento dell'analisi
- Uri: indentifica la uri dell'endpoint

ClassStructure rappresenta la classe che modella l'entità classe, che contiene i seguenti attributi:

- Id: identificativo univoco
- Stato: indica lo stato di avanzamento dell'analisi
- Uri: indentifica la uri della classe
- Endpoint: indentifica la uri dell'endpoint a cui appartiene

PredicateStructure rappresenta la classe che modella l'entità predicato, che contiene i seguenti attributi:

- Id: identificativo univoco
- Stato: indica lo stato di avanzamento dell'analisi
- Uri: indentifica la uri del predicato
- Classe: indentifica la uri della classe a cui appartiene

Relations rappresenta la relazione tra una classe e un oggetto legati da un predicato, che contiene i seguenti attributi:

- Id: identificativo univoco
- Endpoint: indentifica a quale endpoint appartiene la relazione
- Classe: indentifica quale classe appartiene a questa relazione
- Predicato: identifica il predicato che mette in relazione la classe con l'oggetto
- Uri: identifica la uri dell'oggetto
- Cardinalità: indica la cardinalità della relazione

Tutte queste classi sono state annotate con @Entity, che specifica che la classe Java è un'entità. Inoltre, ho utilizzato l'annotazione @Table che specifica la tabella primaria per l'entità annotata.

Endpoint in *ClassStructure*, classe in *PredicateStructure* ed endpoint, classe e predicato in *Relations* sono stati annotati con @ManyToOne, per definire la relazione molti a uno che contraddistingue la loro relazione.

Infine, l'attribuito id definito nelle classi precedenti, è stato annotato con @Id che specifica la chiave primaria di un'entità e @GeneratedValue che prevede la specifica delle strategie di generazione dei valori delle chiavi primarie.

Per ogni classe del modello ho creato le seguenti interfacce. Queste interfacce rappresentano il *Repository* (DAO):

- EndpointRepository
- ClassStructureRepository
- PredicateStructureRepository
- RelationsRepository

Queste interfacce sono tutte state annotate con l'annotazione di Spring @Repository, indica che una classe con questa annotazione è un "Repository".

Nelle interfacce sono presenti solo i metodi creati da me, ovvero i metodi che non riguardano le operazioni standard (CRUD operations), perché le operazioni CRUD sono implementate di default da Spring.

I metodi creati da me sono tutti annotati con @Query(value= “query JPQL”), per poter effettuare le query JPQL sul database.

Dopodichè ho creato il *Controller,* nello specifico *EndpointController*:

Questa classe è stata annotata con @Controller, questo significa che questa classe è un controller, ed è stata annotata anche con @RequestMapping(path = "/app"), ciò significa che l'URL inizia con “/app” (dopo il path dell'applicazione).

Il servizio REST di tipo POST è stato annotato con @PostMapping, annotazione per la mappatura delle richieste HTTP POST, mentre i servizi REST di tipo GET con @GetMapping, annotazione per la mappatura delle richieste HTTP GET.

EndpointController: è la classe in cui sono specificate le varie richieste REST che possono essere invocate:

- **/add_endpoint**: Aggiunge un endpoint al database se non è già presente.

	SERVER ADDRESS
POST	http://localhost:8080/app/add_endpoint

Per invocare la richiesta POST in questione, è necessario specificare nel Body della richiesta la chiave “endpointUri” e come value l’uri dell’endpoint da analizzare.

Se l’endpoint è già presente nel database, il sistema ritornerà il messaggio: “Endpoint gia salvato” come mostrato nella *Figura1*, altrimenti mostrerà “Endpoint Salvato”. Un esempio è mostrato nella figura sottostante, realizzata con Postman.

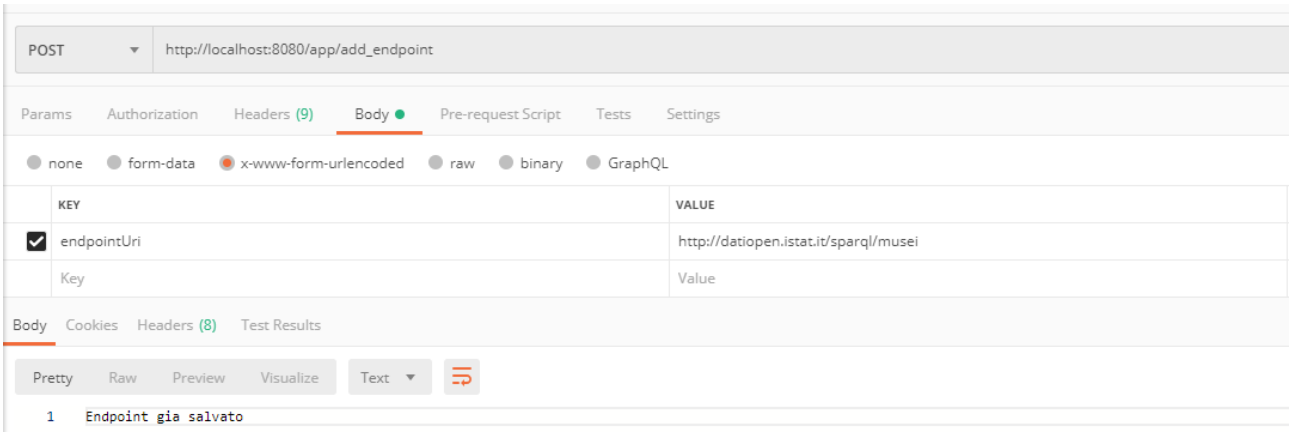

Figura 1: Esempio richiesta POST con Postman

- **/start**: fa partire l’analisi dell’endpoint dall’inizio o dal punto in cui si era fermata.

	SERVER ADDRESS
POST	http://localhost:8080/app/start

Questa chiama POST è simile a quella precedente, anche qui viene passato l’endpoint nel body della richiesta.

- **/all_endpoints**: permette di ottenere tutti gli endpoint salvati nel database in ordine di completamento dell’analisi, ovvero prima gli endpoint con l’analisi completata e poi gli altri.

	SERVER ADDRESS
GET	http://localhost:8080/app/all_endpoints

Di seguito è mostrato un esempio di risultato:


```
[{
      "id":1,
      "status":"ANALISI COMPLETATA",
      "endpointUri":"http://datiopen.istat.it/sparql/musei"

},
   {
      "id":14,
      "status":"ANALIZZARE CLASSI",
      "endpointUri":"http://lod.seoul.go.kr/sparql"

},
   {
      "id":40,
      "status":"ANALIZZARE PREDICATI",
      "endpointUri":" http://opendata.comune.arezzo.it/services/lod/sparql "

}]
```


- **/endpoint**: permette di ottenere il grafo relativo ad uno specifico endpoint. Se l'analisi dell'endpoint è terminata allora restituisce il grafo.

	SERVER ADDRESS
GET	http://localhost:8080/app/endpoint

Nella richiesta è necessario specificare il parametro id dell'endpoint di cui desidero il grafo.

Un esempio è mostrato nella figura sottostante, realizzata con Postman (*Figura2)*.

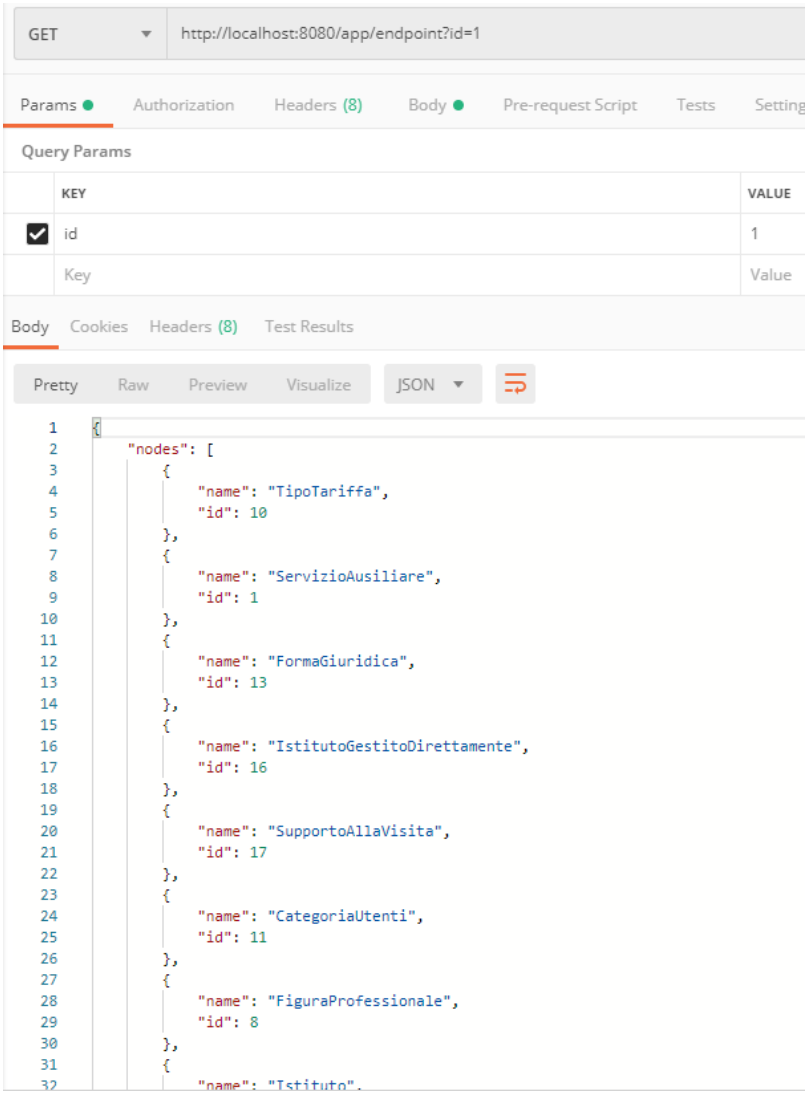

Figura 2: Esempio richiesta GET con Postman

Come si può vedere dalla *Figura2*, il servizio REST restituisce un JSON che raffigura la struttura dell'endpoint analizzato. In particolare, come si può vedere di seguito, il JSON restituito è strutturato così:

Nodes: insieme dei nodi del grafo

Name: nome del nodo

Id: identificativo del nodo

Links: insieme degli archi del grafo

cardCO: indica la cardinalità tra soggetto e oggetto

cardOC: indica la cardinalità tra oggetto e soggetto

source: indica da che nodo parte l'arco

type: indica il nome del predicato che comparirà sull'arco

target: indica in che nodo termina l'arco

```
{
    "nodes": [
        {
            "name": "ServizioAusiliare",
            "id": 1
        },
        {
            "name": "FormaGiuridica",
            "id": 13
        },
        {
            "name": "TitoloBeni",
            "id": 9
        },
        . . .

        {
            "name": "ServizioWeb",
            "id": 20
        }
    ],
    "links": [
{
            "cardCO": "molti",
            "cardOC": "molti",
            "source": 19,
            "type": "siAvvaleDi",
            "target": 8
        },
        {
            "cardCO": "uno",
            "cardOC": "molti",
            "source": 19,
            "type": "haFormaGiuridicaGestore",
            "target": 13
        },
        {
            "cardCO": "molti",
            "cardOC": "molti",
            "source": 5,
            "type": "offre",
            "target": 3
        },
        . . .
        {
            "cardCO": "uno",
            "cardOC": "molti",
            "source": 19,
            "type": "haTematicaSecondaria",
            "target": 15
        }
    ]
}
```

Questi servizi REST utilizzano dei *Service*:

- *ClassFinderService*: permette di eseguire la query SPARQL che trova tutte le classi di un endpoint, filtrando sui prefissi “http://www.w3.org/2002/07/owl” e “http://www.w3.org/2000/01/rdf-schema”.
- *PredicateFinderService* e *AsyncServices*: è il servizio che permette di trovare tutti i predicati legati ad una classe.
- *ObjectFinderService* e *ObjectAsyncServices*: è il servizio che permette di trovare tutti gli oggetti legati ad una classe e ad un predicato della classe. Gli oggetti possono essere *individuals* o classi. Nel caso in cui siano degli *individuals* vengono salvati sul database come “String” o “Integer”, rispettivamente per frasi/parole o numeri.
- *CardinalityFinderService* e *CardinalityAsyncServices*: è il servizio che permette di trovare la cardinalità tra due classi dato un predicato.

Nello specifico, in questi Service sono eseguite le query SPARQL di cui parlerò in dettaglio nel capitolo “*Analisi Endpoint*”.

Tutti i servizi, tranne *ClassFinderService,* sono eseguiti in parallelo tramite l’uso di thread. Inoltre, sono tutti annotati con @Service, che indica che la classe con questa annotazione è un "Servizio".

Inoltre, il metodo delle classi che hanno nel nome “Async” è annotato con @Async, annotazione che identifica un metodo come candidato all'esecuzione asincrona.

In generale, per fare l’inject dell'istanza della classe è necessario dichiarare la classe e annotarla con @Autowired.

Infine, ho creato il main:

SparqlAppApplication: contiene il metodo main della mia applicazione e serve per avviarla. È annotata con @SpringBootApplication, che è utilizzato per abilitare queste tre caratteristiche:

- @EnableAutoConfiguration: abilitare il meccanismo di autoconfigurazione di Spring Boot
- @ComponentScan: abilitare @ComponentScan sul pacchetto in cui si trova l'applicazione
- @Configurazione: consente di registrare ulteriori beans nel contesto o di importare classi di configurazione aggiuntive

3.2. Analisi Endpoint

L’analisi dell’endpoint si suddivide in varie fasi.

L’analisi inizia eseguendo la seguente query SPARQL sull’endpoint ottenuto:

```
SELECT DISTINCT ?class WHERE {

?s a ?class.

FILTER ( !strstarts(str(?class), "http://www.w3.org/2002/07/owl") ).

FILTER ( !strstarts(str(?class), "http://www.w3.org/2000/01/rdf-schema") ).

}
```

Questa query ha il compito di recuperare tutte le classi relative all’endpoint da analizzare, filtrando sui prefissi “http://www.w3.org/2002/07/owl” e “http://www.w3.org/2000/01/rdf-schema”.

Una volta ottenute tutte le classi, si passa alla fase in cui per ogni classe si trovano tutti i predicati. Per trovare i predicati data una classe si esegue la seguente query SPARQL:

SELECT DISTINCT ?p WHERE { ?s a <Uri della classe>. ?s ?p ?o. }

Questa operazione è eseguita in parallelo attraverso l'uso di thread che fanno query SPARQL in parallelo.

Successivamente data la classe e i suoi predicati, si passa alla fase in cui, vengono trovati gli oggetti associati. Anch'esso eseguito in parallelo. Come prima cosa viene eseguita la seguente query che permette di trovare la classe degli oggetti che derivano dalla classe e dal predicato della classe specificati nella query:

SELECT DISTINCT ?otype WHERE { ?s a <Uri della classe>.

?s <Uri del predicato della classe> ?o.

?o a ?otype. });

Gli oggetti possono essere classi oppure individuals. Se gli oggetti sono individuals, vengono salvati come "String" o "Integer" a seconda che siano rispettivamente parole/frasi o numeri. Di seguito è mostrata la query SPARQL nel caso in cui gli oggetti siano degli individuals:

SELECT DISTINCT?o WHERE { ?s a <Uri della classe>.

?s <Uri del predicato> ?o. }

Nella fase finale vengono individuate le cardinalità tra le classi legate da un predicato, sempre tramite query SPARQL eseguite in parallelo. Di seguito sono mostrate le query SPARQL per trovare la cardinalità:

Trova la cardinalità tra classe1 – predicato – classe2:

SELECT ?s (count(*) as ?conto) WHERE { ?s a <Uri della classe1>.

?s <Uri del predicato relativo alla classe1> ?o.

?o a <Uri della classe2 in relazione con classe1 tramite il predicato> } GROUP BY ?s

Trova la cardinalità tra classe2 – predicato – classe1:

SELECT ?o (count(*) as ?conto) WHERE{ ?s a <Uri della classe1>.

?s < Uri del predicato relativo alla classe1> ?o.

?o a < Uri della classe2 in relazione con classe1 tramite il predicato> } GROUP BY ?o

Le relazioni vengono salvate sul database come "*uno*", nel caso in cui la query restituisca solo un risultato o "*molti*" nel caso in cui la query restituisca più di un risultato.

Durante ogni fase viene aggiornato lo stato dell'endpoint, delle classi e dei predicati, in modo tale che, nel caso in cui il sistema abbia problemi o l'endpoint non sia funzionante, l'applicazione possa ripartire dal punto dell'analisi in cui si era fermato.

Inoltre, il sistema è programmato in modo da rieseguire la query SPARQL in caso di mancata risposta. La query viene eseguita due volte, se dopo i due tentativi non si ha risposta allora il sistema mostrerà un messaggio di errore.

3.3. Lato Frontend

Come prima cosa, ho creato il file index.html, in cui ho creato due script.

Uno, è utilizzato per eseguire la chiamata GET al backend per prendere l'elenco degli endpoint.

L'altro, ha il compito, una volta che l'utente ha cliccato sul bottone "Invia" del form, di effettuare la chiamata al servizio REST di tipo POST: "/add_endpoint" e quindi salvare l'endpoint sul database se non è stato salvato in precedenza.

Successivamente, lo script ha il compito di aggiornare l'elenco degli endpoint effettuando una chiamata REST GET "/all_endpoints".

Infine, effettua una chiamata POST "/start" per far partire l'analisi. L'analisi parte dall'inizio se l'endpoint non è stato analizzato in precedenza, altrimenti l'analisi riparte dal punto in cui si era fermata.

Nel caso in cui l'analisi produca un errore verrà visualizzata una finestra pop-up con un messaggio di errore.

Per realizzare entrambi gli script ho utilizzato JQuery.

Infine, ho aggiunto la grafica usando Bootstrap, HTML e CSS ed ho ottenuto la seguente interfaccia *Figura 3*.

Nello specifico, come si può vedere dalla *Figura 3*, la pagina comprende un form, in cui l'utente può inviare l'endpoint al backend per farlo analizzare.

Se l'utente non scrive nulla e clicca su invio verrà visualizzata una finestra pop-up con il seguente messaggio di errore "Please Fill All Fields".

Inoltre, in basso, è presente l'elenco degli endpoint, dove gli endpoint sono mostrati in ordine di completamento, ovvero gli endpoint già completati compaiono per primi nell'elenco. Sempre osservando l'elenco, di seguito ad ogni endpoint è visualizzato lo stato della sua analisi.

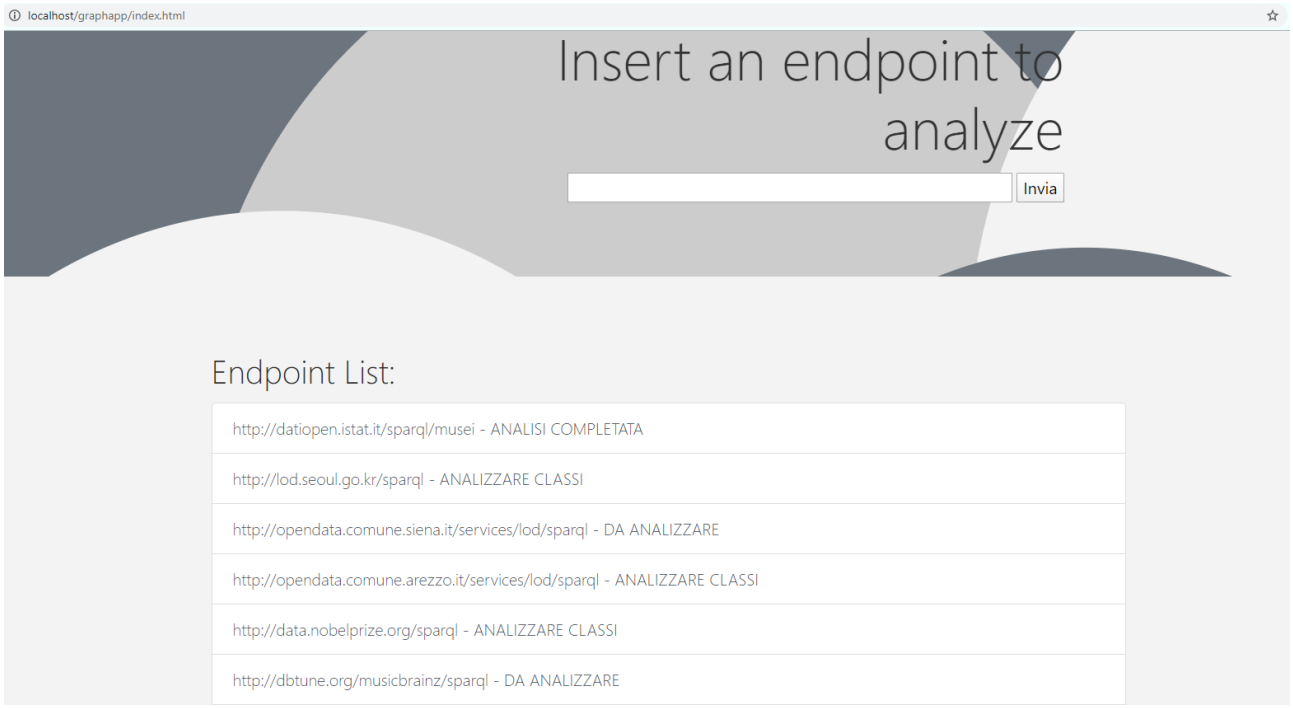

Figura 3: index.html

Cliccando su un endpoint dell'elenco, l'utente viene reindirizzato sulla pagina graph-view.html.

In cui se l'analisi dell'endpoint è completa, verrà visualizzato il grafo, altrimenti comparirà una finestra pop-up con il seguente messaggio di errore "Analisi non completata".

Per quanto riguarda i dettagli implementativi, per realizzare il grafo ho utilizzato la libreria d3.js.

Inoltre, ho realizzato uno script che ha il compito di prendere l'id presente nella url ed eseguire la chiamata GET al backend per ottenere il grafo associato all'endpoint con quel dato id.

Ho aggiunto la possibilità di effettuare lo zoom sul grafo ed infine, ho realizzato la parte grafica della pagina utilizzando Bootstrap, HTML e CSS ed ho ottenuto la seguente interfaccia *Figura 4*.

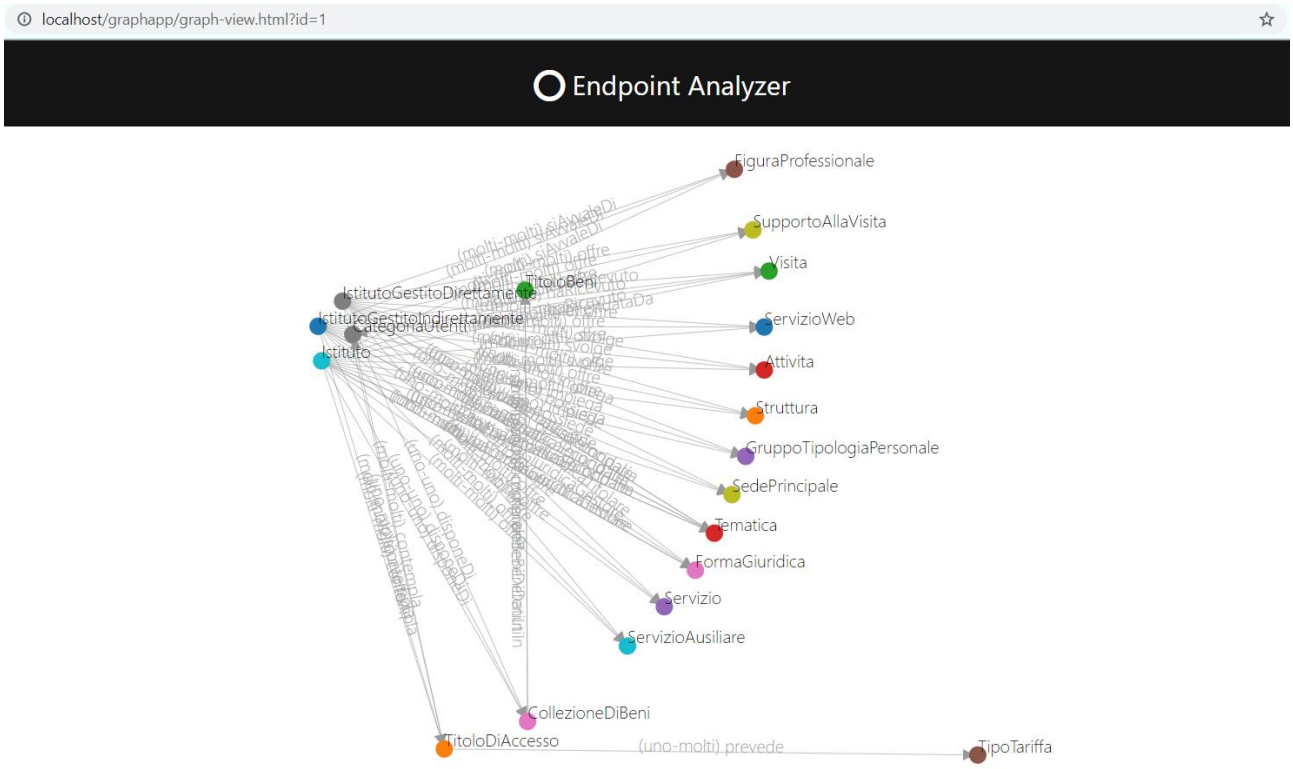

Figura 4: graph-view.html

4. Analisi di sicurezza della WebApp

Sono passata ad analizzare la sicurezza della mia webapp. Ho controllato che principalmente SQL injection ed XSS non fossero rilevati sugli endpoint della mia webapp.

Mi sono avvalsa del tool: OWASP ZAP.

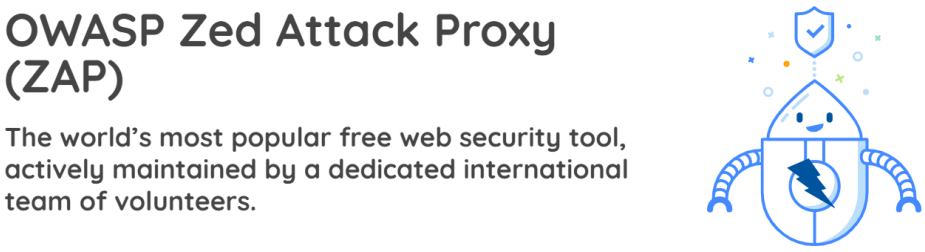

Figura 5: owasp zap

Di seguito sono mostrati gli endpoint analizzati:

4.1. Endpoint FrontEnd

Per quanto riguarda l'analisi degli endpoint lato frontend, sono stati analizzati i seguenti endpoint:

- http://localhost/graphapp/
- http://localhost/graphapp/graph-view.html?id=1

In entrambi non è stata rilevata dal sistema alcuna vulnerabilità di tipo XSS e SQL injection.

4.2. Endpoint BackEnd

Per quanto riguarda l'analisi degli endpoint lato backend, sono stati analizzati i seguenti endpoint:

- http://localhost:8080/app/add_endpoint
- http://localhost:8080/app/start
- http://localhost:8080/app/all_endpoints
- http://localhost:8080/app/endpoint?id=1

Il sistema non ha rilevato alcuna vulnerabilità di tipo XSS e SQL injection.

Inoltre, ho effettuato degli attacchi manuali mettendo le seguenti stringhe nei vari input del sistema:

1. '
2. “>
3. "><script>alert(1)</script>
4. %3Cscript%3Ealert(1)%3C%2Fscript%3E

Sia concatenandoli con valori plausibili, ad esempio un id valido o un url valido a seconda del parametro richiesto dall'endpoint, sia non concatenandoli.

Nello specifico, gli ultimi tre servono per iniettare e perciò eseguire del possibile codice Javascript malevolo, mentre il primo comando serve per troncare una query SQL ed eventualmente poter iniettare codice malevolo SQL.

5. Conclusione

In conclusione, in questo elaborato ho sviluppato una WebApp, sia lato frontend che backend.

Lo scopo di questo elaborato è stato quello di permettere ad un utente di poter eseguire l'analisi di un endpoint e di visualizzare il grafo raffigurante la sua struttura.

All'utente è stata fornita un'interfaccia che permette di inserire un nuovo endpoint da analizzare, di visualizzare l'elenco degli endpoint inseriti ed infine, di poter visualizzare lo stato di avanzamento dell'analisi per ogni endpoint. L'utente inoltre, può visualizzare il grafo raffigurante la struttura di uno specifico endpoint. È possibile anche interagire con il grafo ottenuto, spostando i singoli nodi o l'intero grafo ed effettuando lo zoom su di esso.

Una volta ricevuto l'endpoint da analizzare, il sistema parte con l'analisi eseguendo varie query SPARQL su di esso per poter estrarre i dati relativi alla struttura dell'endpoint.

Il sistema è inoltre in grado di gestire situazioni in cui ci siano delle improvvise interruzioni dell'analisi provocate da cause esterne, questo è possibile grazie al costante salvataggio dello stato dell'analisi. Il salvataggio dello stato dell'analisi, inoltre, permette al sistema di gestire situazioni in cui un endpoint non sia disponibile.