Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

SVILUPPO DI UN SISTEMA DI RICERCA PER AI4EU

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Filippo Bartolucci

Correlatore:
Dott.
Saverio Giallorenzo

III Sessione
Anno Accademico 2019/2020

Indice

Elenco delle figure

Capitolo 1

Introduzione

Ad oggi l'utilizzo di tecnologie basate su intelligenza artificiale è realtà in quasi tutti gli ambiti della nostra vita quotidiana. Rispondono alle nostre domande con gli assistenti virtuali, controllano i dispositivi smart nella nostre case, ci consigliano contenuti secondo i nostri gusti sulle piattaforma online e tanto altro ancora. L'utilizzo di queste tecnologie crea nuove opportunità in tutti i settori che le adottano, dai contesti business a quelli consumer, e questo ha portato a una grande crescita della domanda per questo tipo di applicazioni.

AI4EU è un consorzio nato con lo scopo di creare la prima piattaforma europea on-demand dedicata all'intelligenza artificiale. Si propone come vetrina per gli utenti in cerca di innovazioni basate sull' intelligenza artificiale. La sua comunità è composta da una grande varietà di membri che lavorano insieme per sviluppare conoscenze, algoritmi e strumenti con l'obiettivo di rendere accessibile a tutti, esperti o meno, le soluzioni di questo mondo. La piattaforma è giovane, essendo nata solo nel 2019, ma è in crescita, con un numero di risorse disponibili sempre maggiore. A supportarla c'è anche il contributo di molte università ed enti europei e questo progetto di tesi si inserisce in questo contesto. Il catalogo di risorse offerto è vasto e include progetti di vario tipo, ma non ha un sistema che permetta di ricercarli. L'unico modo per conoscere queste risorse è scorrere il lungo elenco in cui sono raccolte. Questo è un grande ostacolo per la scoperta di questi strumenti da parte di quegli utenti-consumatori che hanno bisogno di soluzione per loro problemi e allo stesso tempo conoscono poco questo mondo.

L'obiettivo che questa tesi si pone è provare a sviluppare un sistema di ricerca che permetta ad un utente qualsiasi di navigare il catalogo delle risorse di AI4EU e trovare quella che riesce meglio a soddisfare le sue richieste. Arriveremo a fare ciò in più passaggi. Per prima cosa dovremo raccogliere tutti i dati delle risorse disponibili sulla piattaforma, poi ci occuperemo della trasformazione di questi dati in un formato utile al nostro scopo insieme al decidere come conservare e strutturare ricerche su di essi. Infine, valuteremo delle possibili soluzioni per il problema della scoperta e ricerca degli elementi del catalogo, presentando dei prototipi utili a testare tali soluzioni sull'audiance target.

Capitolo 2

Raccolta dati

In questo capitolo verranno trattate le prime attività svolte durante il lavoro per questa tesi, ovvero l'analisi e il reperimento dei dati relativi alle risorse del catalogo di AI4EU.

2.1 Le risorse di AI4EU

AI4EU ha un catalogo composto da risorse del mondo dell'intelligenza artificiale (Artificial Intelligence - AI) di vario tipo. Ci sono modelli per intelligenze artificiali, librerie, dataset, eseguibili e altro ancora. Scopo di questo catalogo è diventare il crocevia per lo scambio di risorse del mondo AI in Europa e connettere consumatori con i fornitori delle risorse AI. Tuttavia, la mancanza di un sistema di ricerca pregiudica la possibilità per un consumatore di trovare la risorsa che fa al caso suo.

Ciascuna delle risorse presenti del catalogo ha una serie di informazioni che la caratterizzano, come la licenza, il tipo di distribuzione, l'autore e altre ancora. Queste informazioni sono gli elementi che vogliamo utilizzare nello sviluppo del nostro sistema di ricerca. Purtroppo l'unico modo per ottenerle è attraverso le pagine web dedicate alle risorse presenti nel catalogo, in quanto AI4EU non fornisce nessun sistema per la loro consultazione. Questo è un problema perché i dati che ci servono non sono in un formato utilizzabile per i nostri scopi.

L'unica base dati a nostra disposizione è un'ontologia interna di AI4EU usata come data model per le risorse (Figura 2.1). Un'ontologia è una

descrizione formale dei concetti di un dominio di dati, ossia un modello che ci consente di rappresentare delle concetti di vario tipo e le loro relazioni. Un'ontologia è in grado di rappresentare qualsiasi tipo di dato, strutturato o meno, e la creazione di relazioni tra concetti ci permette di effettuare vari tipi di ragionamenti tra le informazioni.

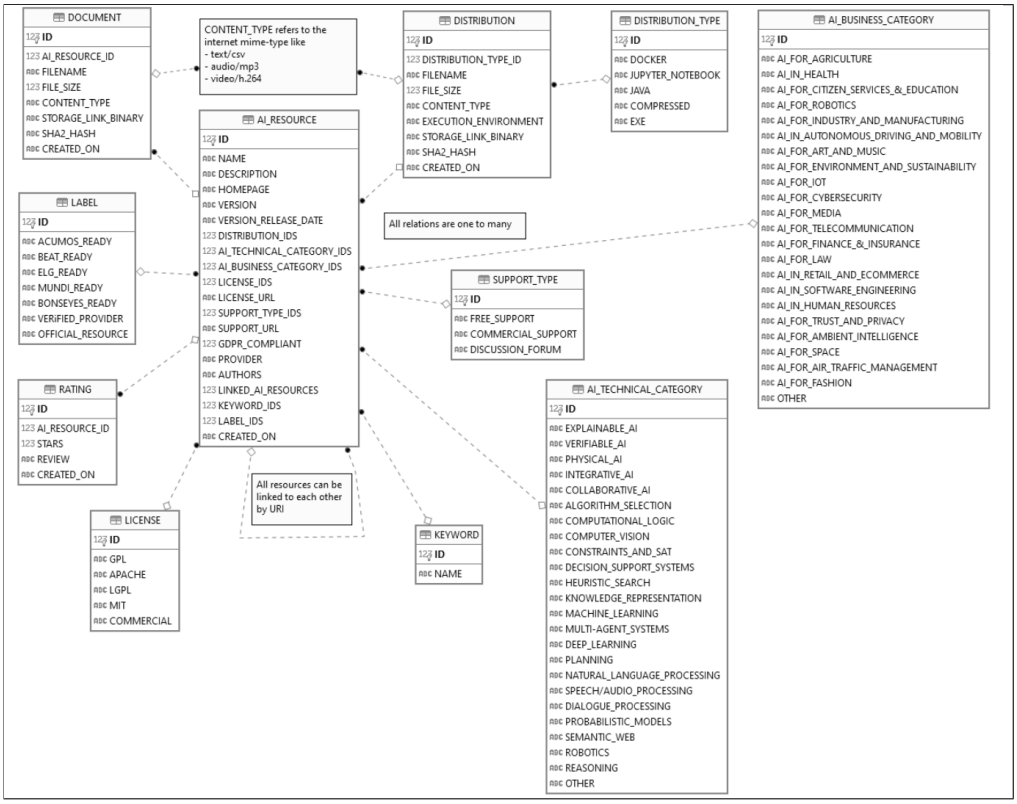

Figura 2.1: Data model risorse

Grazie a questa ontologia possiamo conoscere da vicino il tipo di informazioni che ogni risorsa può avere.

Il nostro primo obiettivo è costruire un set di dati, modellato sulla base del data model offerto dall'ontologia, su cui potremo poi fare le nostre operazioni di ricerca ragionando sui dati. Vogliamo che anche questi dati siano strutturati come l'ontologia per poter fare dei ragionamenti sulle risorse.

Per arrivare a ciò dovremmo accedere alle singole pagine delle risorse per trascrivere queste informazioni. Un processo manuale che richiede un grosso investimento di tempo e un'alta probabilità di commettere errori. Per risolvere, abbiamo sviluppato un web scraper automatico con lo scopo

di automatizzare questa operazione. Questo ci permetterà anche di ripetere le operazioni di raccolta dati quante volte vogliamo in poco tempo e con una probabilità ridotta di commettere errori di trascrizione.

2.2 Web Scraper

Il web scraping (dal verbo to scrape - raschiare) è una tecnica per l'estrazione di informazioni da pagine web. Con questa tecnica possiamo raccogliere dati non strutturati e convertirli in un qualsiasi formato strutturato (Figura 2.2).

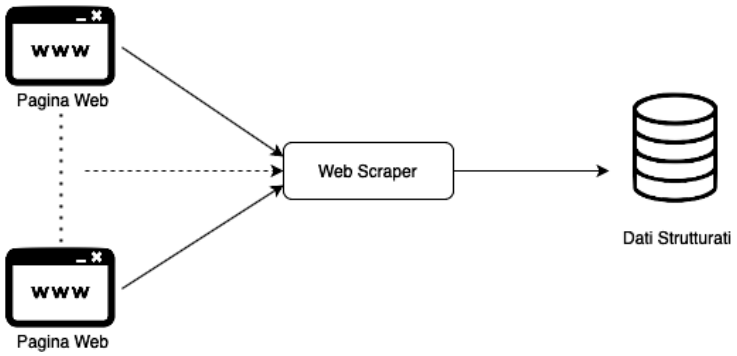

Figura 2.2: Risultato scraper

Non tutte le informazioni di una pagina web sono utili e uno scraper mira solo delle parti specifiche di ogni pagina che visita. L'attività di uno scraper è semplice: gli si forniscono gli indirizzi delle pagine che dovrebbe esaminare, il tool carica il codice html delle pagine, il codice viene esaminato seguendo una certa struttura e vengono estratti i dati che abbiamo richiesto.

Per realizzare il nostro scraper abbiamo usato JavaScript, NodeJS e Puppeteer [1], una libreria per NodeJS che fornisce API per controllo automatico di un web browser. Nelle prime righe di codice, lo scraper, dopo aver inizializzato una nuova istanza del browser, fa il login sulla homepage di AI4EU, in quanto l'elenco delle risorse è visibile solo dopo aver effettuato l'accesso.

Vengono lette da un file esterno le credenziali e con queste si compilano i campi necessari. Lo scraper riconosce gli elementi, box di testo e form della pagina HTML tramite selettori CSS, che conosciamo già perché trovati

ispezionando gli elementi della pagina di nostro interesse con gli strumenti da sviluppatore del browser.

Di seguito viene riportato uno snippet del codice in questione.

```
let login_data = JSON.parse( fs.readFileSync( 'login_data.json') );
  const browser = await puppeteer.launch( { headless: false } );
  const page = await browser.newPage();
  await page.goto('https://www.ai4eu.eu/user/oauth2', { waitUntil: 'networkidle0' });
  await page.type('#username', login_data.username);
  await page.type('#password', login_data.password);
  await Promise.all([
    page.click( "#loginForm > div.form__actions > button" ),
    page.waitForNavigation({ waitUntil: 'networkidle0' }),
  ]);
```

Ad accesso completato lo scraper può spostarsi nella pagina del catalogo e creare la lista delle risorse disponibili. Come prima, lo scraper si muove tra gli elementi della pagina leggendo i selettori CSS e riesce a trovare tutti i nomi delle risorse che ci interessano.

Nel riquadro sottostante il codice che si occupa di questo passaggio.

```
var tool_pages = new Array();
tool_pages = await page.evaluate( async ( tool_pages ) => {
   var findPages = () => {
      document.querySelectorAll( ".catalog-resource a" ).forEach( ( e ) => {
      tool_pages.push( String( e.getAttribute( "href" ) ) );
         });
         findNext();
      }
      var findNext = () => {
         const nextArrow = "#catalog-listing-js--ajax > nav > ul >
              li.pager__item.pager__item--next";
         if( document.querySelector( nextArrow ) !== null ){
              document.querySelector( nextArrow + " a" ).click();
              findPages();
         }
   }
   findPages();
   return tool_pages;
   }, tool_pages
);
```

Con la lista completata sappiamo quali risorse sono disponibili e si può iniziare la visita delle singole pagine. L'accesso avviene parallelamente e per

ognuna di queste vengono svolte le stesse operazioni, in quanto la struttura è sempre la stessa e a cambiare sono solo i contenuti. Grazie al data model fornito dall'ontologia sappiamo già che tipo di informazioni possiamo aspettarci di trovare e, in base a questo, ci orientiamo per i dati da raccogliere.

Nel box è riportato uno snippet del codice discusso:

```
for( tool_page_index in tool_page_chunk ){
   tools.push( await pages[ tool_page_index ].evaluate( async () => {
      const tool = new Object();
      tool.page = (window.location.href).split("resource/")[1]
      tool.name = document.querySelector( "#block-ai4eu-content > div > div:nth-child(1) >
           div > div > div > div.highlights-header__content > div.highlights-header__data >
           h2" ).textContent;
```

In questa fase, ogni risorsa è trattata come un oggetto JSON e le informazioni associate vengono salvate come sue proprietà (Figura 2.3). A termine della sua esecuzione, lo scraper restituisce un file con tutti i risultati trovati.

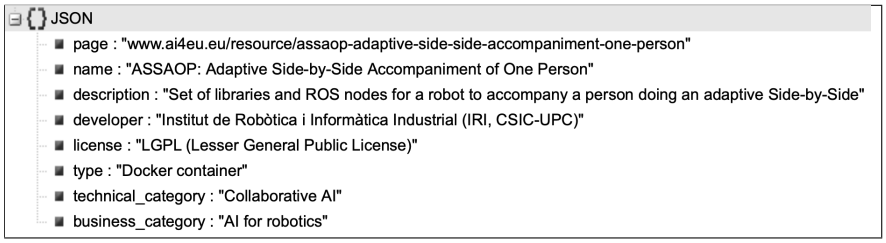

Figura 2.3: Risorsa in JSON

2.3 JSON

JSON, abbreviazione di JavaScript Object Notation [2], è un formato per la condivisione dei dati derivato da JavaScript, da cui prende la sintassi, ma ormai usato anche con molti altri linguaggi. Un oggetto JSON è formato da coppie chiave-valore racchiuse da parentesi graffe.

È un formato leggero e facilmente leggibile ed è utilizzato principalmente nel mondo web. Per questo è il formato più comodo da usare in questo momento.

```
{
   "name" : "Mario",
   "surname": "Rossi"
}
```

Purtroppo non è un formato ideale per un sistema di ricerca in quanto non è pensato per ottenere query efficienti.
Adesso il nostro obiettivo è trasformare questi dati da JSON allo stesso formato dell'ontologia per sfruttare tutti i vantaggi che ci possiamo avere da quel formato.

Capitolo 3

Database

All'interno di questo capitolo verrà presentato il lavoro svolto sul dataset appena ottenuto per trasformarlo in un formato più adatto ai nostri scopi.

3.1 Resource Description Framework

A questo punto, grazie al nostro scraper, abbiamo tutta una serie di informazioni per ogni risorsa del catalogo, ma sappiamo che JSON non è un formato adatto per i nostri scopi. Vogliamo trasformare tutti questi dati in RDF [3][4], acronimo di Resource Description Framework, ovvero lo stesso formato dell'ontologia interna di AI4EU. Così manteniamo anche una continuità tra i dati interni della piattaforma e i nostri risultati.

RDF è il linguaggio proposto dal W3C [5] per la descrizione di metadati associati ad una risorsa ed è oggi il formato alla base del Semantic Web. In RDF qualunque cosa può essere identificata attraverso un Universal Resource Identifier (URI) e qualunque cosa può dire qualcosa su qualunque cosa.
Il modello RDF è basato su tre elementi:

- Risorse, qualunque cosa possa essere identificata da un URI
- Proprietà, relazioni che legano risorse e valori
- Valori, dato primitivo che può anche contenere un URI di un'altra risorsa

Uno statement RDF è l'unità base di rappresentazione delle informazioni e si esprime attraverso una tripla semantica [6], composta, come suggerisce il

nome, da tre entità nella forma <soggetto-predicato-oggetto> (e.g. Mario ha 30 anni).

Uno statement RDF si può mostrare graficamente con grafo. La risorsa è rappresentata da un'ellisse, l'oggetto da un rettangolo, e la proprietà da un arco orientato, che collega risorse e oggetto (Figura 3.1).

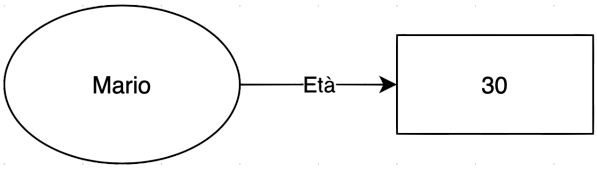

Figura 3.1: Grafo RDF

In forma testuale RDF ha due rappresentazioni principali: XML o Turtle. Quest'ultima è quella che useremo in questa tesi.

3.2 Turtle

La notazione Turtle [7] è usata per esprimere triple RDF in forma compatta ed evitando le ripetizioni. Questo è possibile perché Turtle adotta le seguenti semplificazioni:

- Uso di prefissi che permettono di evitare le ripetizioni di elementi comuni negli IRI.
- Soggetto in comune per più statement, il soggetto viene dichiarata una solo volta per più coppie di predicato-oggetto.
- Soggetto-predicato in comune con oggetti diversi.

Queste caratteristiche ci saranno utili perché avremo che ogni risorsa sarà soggetto di molti statement diversi.

A livello di sintassi, Turtle adotta convenzioni tradizionali: un URI di risorsa viene scritto tra parentesi angolari, i valori letterali tra virgolette e le triple terminano con un punto. Infine si usa una chiocciola per indicare una abbreviazione. Un ultimo vantaggio di Turtle è che ha un sintassi simile a SparQL il linguaggio di interrogazione per i dati RDF. L'esempio nella Figura 3.1 scritto in Turtle diventa: <Mario> <età> <30>.

3.3 Triple Store Database

Un Triple Store Database è un database speciale sviluppato per la gestione di triple RDF ed è quello che ci serve per salvare e recuperare i nostri statement. Ci permette l'interrogazione dei dati tramite il linguaggio SparQL. Per questa tesi ci siamo affidati a GraphDB Free, della società Ontotext, perché è quello che abbiamo trovato più semplice da utilizzare e con tutte le funzionalità di nostro interesse, prima fra tutte la possibilità di mappare oggetti JSON in statement RDF.

GraphDB infatti integra un tool, chiamato OntoRefine, che permette di convertire dati tabulari in RDF. Per fare ciò è sufficiente importare, nella sezione Tabular del pannello di controllo di GraphDB, il file JSON con tutte le nostre risorse e da questo verrà creata una tabella popolata dai tutti nostri dati (Figura 3.2).

All		_ - page	_ - name	_ - type	_ - license	_ - developer	_ - description	_ - business_ca	_ - technical_c
1.		edu2comapi	edu2comAPI	Docker container	ACADEMIC LICENSE	IIIA-CSIC	edu2com is a team formation algorithm for allocating teams of students to internship programs	AI for citizen services & education	Decision support systems
2.		covid-19-chest-ct-scan-dataset	Covid-19 chest CT-scan Dataset	Dataset	Creative Commons BY-NC-ND license	nehs	This is a sample of the Covid-19 chest CT-scan Dataset	AI in health	Machine Learning
3.		ai4eu-competences	ai4eu-competences	Executable	ASF 2.0	Fondazione Bruno Kessler	Tool to match text with ESCO competences.	AI for citizen services & education	Natural language processing
4.		lionforests-local-interpretation-random-forests	LionForests: Local Interpretation of Random Forests	Jupyter notebook	LICENSE	Intelligent Systems Lab - School of Informatics - Aristotle University of Thessaloniki	Building interpretable random forests!	Trusted and Privacy preserving AI	Explanable AI
5.		pomdp-ir-solver	POMDP-IR Solver	Executable	GPL V3	Instituto Superior Técnico	A decision-making framework for active perception with POMDPs.	AI for robotics	Planning

Figura 3.2: Tabella Dati JSON

Con i dati caricati possiamo lavorare per creare la mappatura da JSON a RDF. Bisogna decidere come e quali statement vogliamo formare perché dobbiamo trasformare le proprietà degli oggetti nella forma <soggetto-predicato-oggetto>, rispettandone il significato.

Dato che ogni risorsa ha più informazioni associate, scegliamo che le risorse saranno soggetto di tutti gli statement che le riguarderanno. Deciso questo, usiamo l'URL che identifica la risorsa su AI4EU come URI per il soggetto e aggiungiamo anche il prefisso base comune a tutti gli identificatori. Per formare la coppia predicato-oggetto, usiamo la coppia chiave-valore delle proprietà JSON.

Quindi, per ogni proprietà di un oggetto JSON, date in coppia chiave-valore, si crea uno statement RDF che segua la forma <oggetto-chiave-valore>. Prendiamo per esempio il seguente oggetto:

```
"page": "ai-drummer",
"name": "AI Drummer",
"description": "AI Drummer that responds in real-time to the playing of a Human
    pianist.",
"developer": "AASS, Orebro University",
"license": "MIT",
"type": "Executable",
"technical_category": "Computational logic",
"business_category": "AI for art and music"
```

Nella proprietà `"page"` dell'oggetto c'è salvato il path relativo rispetto all'indirizzo base delle risorse nel catalogo. Quindi si unisce questo valore con prefisso e si ottiene l'URI che verrà usato come soggetto per questa risorsa.

Come spiegato, per gli statement, si prende ogni coppia chiave-valore e si usano come predicato e oggetto. Usando come esempio la coppia `"license"` e `"MIT"` si ottiene la tripla composta da soggetto <"ai-drummer">, predicato<"license"> e oggetto <"MIT">.

Facendo così, riusciamo a mantenere la struttura e significato che avevamo già in JSON e otteniamo un risultato che ricalca quello della ontologia interna utilizzata in partenza.

Definita la mappatura, viene generata una query SparQL che si occupa di mettere in pratica la trasformazione dati.

3.4 SparQL

SparQL è il linguaggio di interrogazione per dati RDF. Il nome o la sintassi, per la condivisione di alcune keyword come `SELECT` e `WHERE`, potrebbero ricordare SQL, ma le idee dietro sono diverse.

La forma della sua sintassi è pensata per assomigliare a RDF, infatti una query SparQL consiste di un'insieme di triple nella forma <soggetto-predicato-

oggetto> in cui soggetto, predicato o oggetto possono essere scritti come variabili, distinguibili dalle costanti per il punto esclamativo messo all'inizio.

```
SELECT ?name
WHERE {
   ?x foaf:name ?name
}
```

Per risolvere una query viene fatto una match delle triple SparQL con le triple RDF salvate e si cercano delle soluzioni possibili per le variabili.

Generalmente una query inizia con la keyword SELECT, seguita da una o più variabili che vogliamo ci vengano restituite. Tra parentesi graffe che seguono il WHERE sono racchiuse le triple da usare per il match da cui si ottiene il valore delle variabili. Qualsiasi parte della tripla può essere soggetto o costante, nell'esempio sopra la proprietà è l'unica costante. I valori costanti vengono usati per il match con le stringhe presenti nel database e da queste si ricavano tutti i valori che possono assumere le variabili.

Riguardando ancora all'esempio, con quella query vogliamo che ci vengano restituiti tutti i possibili valori della variabile *?name* quando questa è l'oggetto di una qualsiasi tripla con predicato costante *foaf:name*.

La query che si occupa di mappare i dati JSON è formata da due parti:

- Nella prima si usa il metodo CONSTRUCT per definire la struttura del grafo RDF che la query dovrà restituire. Il costrutto CONSTRUCT restituisce un singolo grafo specificato dal template racchiuso tra parentesi graffe. Il grafo RDF risultante è formato prendendo ciascuna soluzione delle query tra parentesi graffe e sostituendole alle variabili nelle triple. Le triple risultanti vengono combinate in un singolo grafo RDF mediante unione.
- Nella seconda parte della query vengono assegnati i valori delle variabili nel grafo. Il metodo BIND lega un valore o il risultato di un'espressione a una variabile. Ad ognuna di queste viene fatta corrispondere una colonna della tabella generata al momento dell'importazione dei dati e da lì legge i valori da assegnare.

Di seguito viene riportato il codice completo della query SparQL.

```
BASE <https://www.ai4eu.eu/resource/>
PREFIX mapper: <http://www.ontotext.com/mapper/>
CONSTRUCT {
    ?s1 <name> ?o_name ;
        <type> ?o_type ;
        <license> ?o_license ;
        <developer> ?o_developer ;
        <description> ?o_description ;
        <business_category> ?o_business_category ;
        <technical_category> ?o_technical_category .
} WHERE {
    SERVICE <rdf-mapper:ontorefine:1684013736945> {
        BIND(IRI(mapper:encode_iri(?c_____page)) as ?s1)
        BIND(STR(?c_____name) as ?o_name)
        BIND(STR(?c_____type) as ?o_type)
        BIND(STR(?c_____license) as ?o_license)
        BIND(STR(?c_____developer) as ?o_developer)
        BIND(STR(?c_____description) as ?o_description)
        BIND(STR(?c_____business_category) as ?o_business_category)
        BIND(STR(?c_____technical_category) as ?o_technical_category)
    }
}
```

Eseguendo la query otteniamo un insieme di statement che descrivono tutte le informazioni che abbiamo a riguardo delle nostre risorse. Ora i dati sono a nostra disposizione e siamo quasi pronti a utilizzarli per le nostre ricerche.

Prima di poterli usare, dobbiamo creare una nuova repository sul nostro database e caricare il file con gli statement RDF. GraphDB viene eseguito in locale sullo stesso elaboratore in cui risiedono i dati e si accede al suo pannello di controllo tramite un browser.

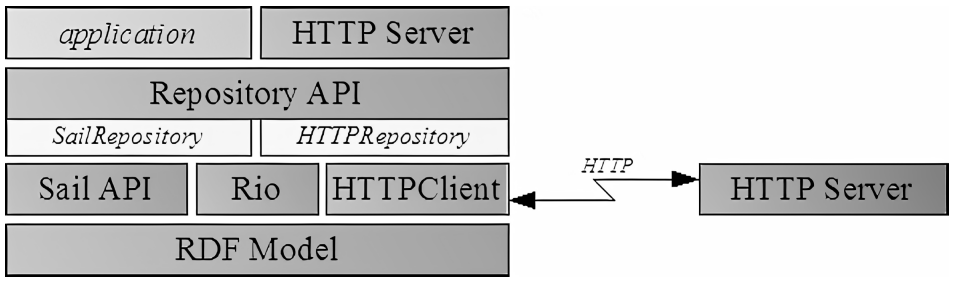

Figura 3.3: Architettura Database

Da questa pagina si può creare una repository, ovvero uno spazio dedi-

cato alla memorizzazione dei dati, dove andremo a caricare gli statement RDF. Questa repository ci servirà per iniziare la comunicazione con l'endpoint del database.

L'endpoint è il punto di ingresso delle comunicazioni del database e implementa i metodi che permettono di interagirci attraverso la rete. In client interagisce, in questo caso tramite API HTTP, con l'endpoint del database e può accedere ai dati conservati al suo interno. Nonostante un database sia composto da più elementi, l'endpoint permette di vederlo da fuori come fosse una cosa sola (Figura 3.3).

Configurato il database possiamo caricare tutti i dati a nostra disposizione e iniziare a fare query.

Capitolo 4

Sistemi di ricerca

In questo capitolo vedremo delle soluzione per realizzare dei sistemi di ricerca di vario livello partendo da quanto abbiamo ottenuto fino ad ora.

4.1 GraphDB+NodeJS

Il database scelto, GraphDB, permette di eseguire query sui dati sia dal suo interno, sia tramite HTTP utilizzandolo come endpoint. Per costruire il nostro sistema per la ricerca dei tool abbiamo bisogno di una piattaforma che ci permetta di sperimentare con il database, fare query e mostrare i risultati.

Per realizzarla abbiamo deciso di usare di nuovo NodeJS, stavolta per la creazione di un server e una pagina web, il primo servirà per comunicare le query al database, la seconda per visualizzare i risultati. Grazie ad un modulo, Graphdbjs[8], la comunicazione tra NodeJS e GraphDB è realizzabile senza particolari difficoltà.

Nel codice del nostro server Node aggiungiamo delle righe dedicate alla configurazione dell'endpoint di GraphDB e possiamo subito effettuare il login nella repository del nostro database senza problemi.

```
let graphDBEndpoint = new EnapsoGraphDBClient.Endpoint({
    baseURL: serverGDB.url,
    repository: serverGDB.repository,
    prefixes: DEFAULT_PREFIXES,
    transform: "toJSON",
});

graphDBEndpoint
.login(serverGDB.user, serverGDB.password)
.then((result) => {
   console.log(" - " + result.message)
   })
.catch((err) => {
   console.log(" - " + err.message);process.exit(1);
   });
```

Dobbiamo occuparci solo di come gestire la comunicazione tra server e database, perché le funzionalità di query e gestione dati sono già integrate tra le API di GraphDBjs.

Siccome vogliamo visualizzare le risorse cercate su una pagina web creiamo dei metodi REST sul server cosicché si possano fare richieste dalla pagina al server per ottenere i dati.

A questo punto possiamo concentrarci su come sviluppare i metodi di ricerca.

4.2 Ricerca diretta in SparQL

Il primo tipo di ricerca che proviamo è quello più semplice da implementare, ma che richiede maggiori conoscenze da parte dell'utente per essere usato. Avendo già server e database connessi, possiamo permettere ad un utente di cercare le risorse scrivendo direttamente query in SparQL.

A livello di codice c'è poco lavoro da svolgere, perché ci basterà leggere il testo della query da un form e inviarla al server che a sua volta dovrà mandarla al database. Dobbiamo solo occuparci del codice extra per poter visualizzare al meglio i risultati (Figura 4.1).

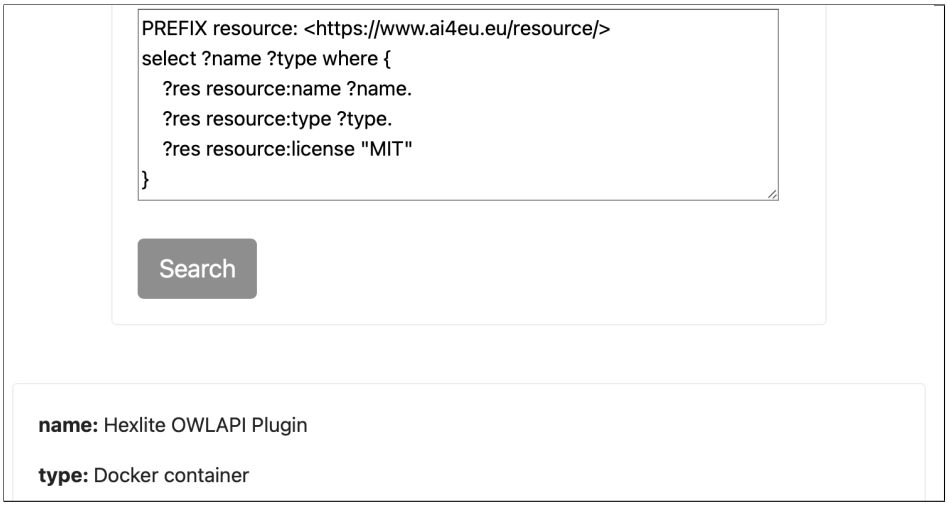

Figura 4.1: Ricerca con query

Sviluppando un sistema di ricerca con queste caratteristiche, non andiamo incontro a grosse difficoltà e non abbiamo bisogno di grandi risorse, ma non è una soluzione soddisfacente per la ricerca di risorse in una piattaforma. Perché, anche se un utente avanzato può sfruttare tutta la potenza del linguaggio, un utente meno esperto non riuscirà a scrivere una query perché non ha conoscenze di SparQL e dell'organizzazione dei dati nel database. La libertà data dalla potenza espressiva di SparQL non ci aiuta a risolvere il problema della scoperta e ricerca delle risorse.

Inoltre un sistema del genere è a rischio attacchi database injection per la facilità con cui si possono scrivere query di qualunque tipo. Sarebbe sensato aggiungere delle operazioni di sanificazione della query, prima che questa venga mandata al database, per assicurarci che non comprometta lo stato dei dati o del database

4.3 Ricerca per stringhe

Per rendere il sistema di ricerca accessibile dobbiamo nascondere le query SparlQL e usare sistemi che non richiedano conoscenze di come sono strutturati i dati. Queste operazioni non devono essere visibili per l'utente.

Passiamo quindi a sviluppare un sistema basato su string matching, immediato da usare per qualsiasi tipo di utente. Per realizzarlo partiamo

facendo delle modifiche al codice sviluppato finora. Bisogna pensare ad una query che ci permette di trovare tutti gli statement RDF che contengono nel loro oggetto la stringa che vogliamo usare per la ricerca. I metodi `FILTER` e `REGEX` sono ideali per questo scopo. Con il primo filtriamo i risultati trovati in funzione di una condizione verificabile, con il secondo possiamo usare le espressioni regolari. Utilizzandoli in combinazione possiamo filtrare variabili che contengono una certa stringa.

```
PREFIX resource: <https://www.ai4eu.eu/resource/>
SELECT ?tool ?name ?desc WHERE {
   ?tool resource:name ?name.
   ?tool resource:description ?desc.
   FILTER(REGEX(str(?desc),"Constraint Programming"))
}
```

Nell'esempio riportato sopra, usiamo `?tool resource:description ?desc.` per cercare una descrizione di risorsa poi filtriamo i risultati ottenuti con `FILTER(REGEX(str(?desc),"Constraint Programming"))` e otteniamo solo le risorse che hanno "Constraint Programming" come sottostringa nella loro descrizione. In questo esempio si lavora solo su una valore, ma si può lavorare nello stesso modo anche con tutti gli altri.

Il lavoro sulla query è quasi completo, ma abbiamo bisogno di aggiornare il codice del server NodeJS. Per irrobustire la sicurezza, ed evitare così il rischio di database injection, lasciamo che sia solo il server a poter formare la query con dei parametri passati dall'utente attraverso un form sulla pagina web. Volendo si può affinare la ricerca permettendo di usare keyword diverse per informazioni invece di cercare la stessa cosa in tutti gli statement.

Questa versione della ricerca è semplice da usare per un utente qualsiasi, ma ancora non ci soddisfa appieno per quanto riguarda la capacità di scoprire risorse che non si conoscono. Possiamo trovare gli elementi del catalogo, ma se non si conoscono le risorse o non si sa quali tool possono risolvere un dato problema questo sistema non è ancora sufficiente. Vogliamo che il consumatore, anche il meno informato, possa essere guidato attraverso le caratteristiche delle varie risorse per poter trovare quella che più si avvicina ai suoi interessi.

4.4 Ricerca Guidata

Abbiamo bisogno di un approccio ancora diverso, vogliamo guidare l'utente nella scoperta delle risorse in base alle sue necessità. L'utente non deve sapere quali risorse si occupano di quali problemi. Decidiamo di costruire un terzo sistema, più guidato rispetto agli altri guidato e basato sulle caratteristiche che conosciamo delle risorse. Deve esserci la possibilità di filtrare la lista in funzione delle proprietà che l'utente sceglie. Dobbiamo perciò raccogliere tutti i possibili valori che una certa proprietà può avere.

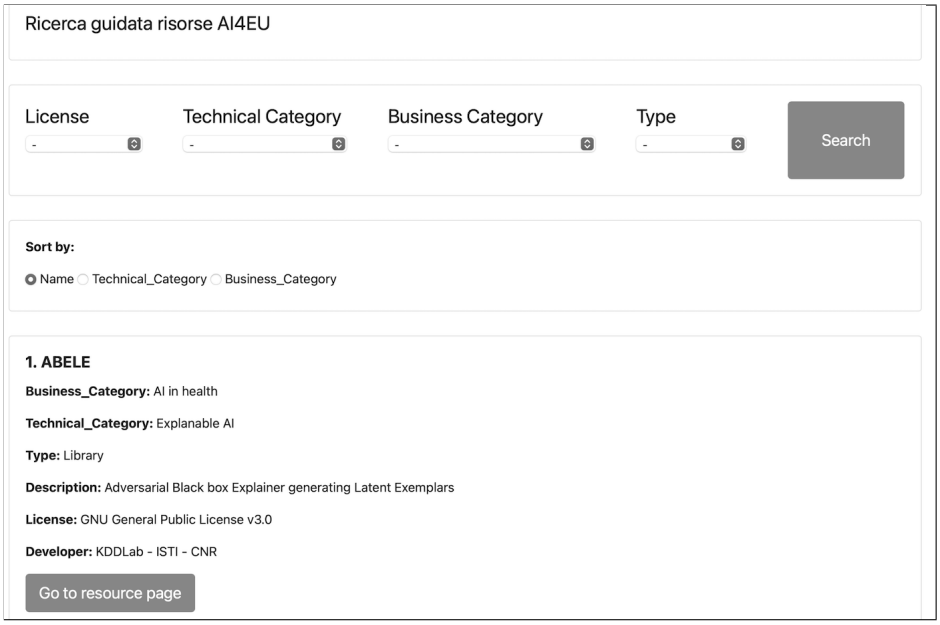

Figura 4.2: Ricerca Guidata

Ovviamente non vogliamo scrivere questa lista a mano quindi usiamo ancora SparQL per ottenere dinamicamente tutti i valori distinti che possono contraddistinguere le proprietà.

```
PREFIX resource: <https://www.ai4eu.eu/resource/>
       SELECT DISTINCT ?Technical_Category where {
          ?y resource:technical_category ?Technical_Category.
       }ORDER BY ASC(UCASE(str(?Technical_Category)))
```

Nell'esempio, l'esecuzione della query ci restituisce distintamente tutte le categorie tecniche citate nelle risorse, ordinate alfabeticamente. L'utilizzo di `DISTINCT` ci garantisce la loro unicità, mentre `ORDER BY ASC(UCASE(str(?Technical_Category)))`

si occuperà di ordinarli alfabeticamente. Faremo delle query simili, cambiando la proprietà cercata, per tutti gli altri predicati delle risorse.
Le informazioni che otteniamo con queste query andranno a popolare la pagina di ricerca. Tramite dei selettori per i vari dati potremo filtrare i risultati e cercare quindi risorse con determinate caratteristiche Ora che abbiamo tutte queste informazioni creiamo lato pagina web l'interfaccia di ricerca. Il server esegue le query per ottenere i valori che andranno a popolare i menù della ricerca.

Con questa implementazione la ricerca si semplifica ancora di più, ma non ci rende ancora pienamente soddisfatti, perché per filtrare i dati si usano informazioni tecniche che per un utente qualsiasi possono essere sconosciute. Prendiamo come esempio i tipi di licenza, ci sono molte varianti e non è facile capire quali permettono di vendere o quali no. Sfruttando le caratteristiche dell'ontologia possiamo semplificare ancora di più questa interazione.

Uno dei vantaggi che abbiamo è che, in un'ontologia, possiamo dire qualunque cosa su qualunque cosa e qualsiasi risorsa può essere usato come soggetto di uno statement. Quindi possiamo arricchire le informazioni che abbiamo a nostra disposizione per ottenere query ancora più specifiche. Usando ancora come esempio le licenze, possiamo arricchire le loro informazioni per ogni tipo aggiungendo se sono utilizzabili per scopi commerciali o no, oppure se viene fornita o meno una licenza. Questo arricchimento si esprime sempre tramite statement RDF, quindi, se vogliamo dire che la licenza MIT permette un utilizzo commerciale scriviamo: <MIT> <permission> <Commercial>. Con queste semplificazioni possiamo offrire all'utente la possibilità di scegliere tra caratteristiche che vuole e sarà poi la query a trovare risorse con la licenza giusta. Ciò non richiede più che si conoscano informazioni particolari sul tipo di licenza da usare.

Purtroppo, le informazioni sulle risorse che abbiamo sono limitate a quello che possiamo trovare nelle pagine dedicate nel catalogo. Non sappiamo nel dettaglio come funzionano e quindi le operazioni di arricchimento si fermano alle caratteristiche come la licenza. Esaminando più da vicino cosa fanno le risorse si potrebbero codificare statement che permetterebbero di rendere ancora più semplice la loro individuazione da parte degli utenti.

Capitolo 5

Conclusioni

Il lavoro svolto ci ha permesso di sviluppare tre modelli di sistemi di ricerca per la piattaforma AI4EU. Il primo semplice da implementare, ma difficile da usare per le competenze che richiedeva. Il secondo richiede più lavoro, è accessibile a tutti, ma non aiuta a scoprire nuove risorse. L'ultimo modello sviluppato ci ha permesso di creare una ricerca che non richiede conoscenze particolari da parte degli utenti, facile da usare e permette di trovare risultati senza conoscere le risorse. È chiaro che il risultato di questa tesi non può, e non vuole, essere esaustivo. Le informazioni sulle risorse a nostra disposizione sono limitate, ma ci hanno permesso di sperimentare e ottenere un risultato interessante, che può essere una base di partenza per la creazione di un sistema di ricerca per la piattaforma. Con un sempre maggiore numero di informazioni sui tool, si potrebbe affinare ancora di più la ricerca e migliorare la capacità di scoprire risorse seguendo l'esempio svolto per il terzo modello di ricerca. Il lavoro svolto potrebbe essere portato avanti andando a ottenere informazioni molto più specifiche sulle attività che svolgono le risorse. Questo permetterebbe anche nuovi sviluppo, invece di trovare delle singole risorse, si potrebbero combinare le risorse a disposizione per proporre nuove soluzioni più vicine alle problematiche che un utente cerca di risolvere.

Bibliografia

[1] *Puppetter*. URL: https://pptr.dev/. [Online; accessed: 10.12.2020].

[2] *Introducing JSON*. URL: https://www.json.org/json-en.html. [Online; accessed: 10.12.2020].

[3] Wikipedia contributors. *Resource Description Framework*. URL: en.wikipedia.org/wiki/Resource_Description_Framework. [Online; accessed: 11.12.2020].

[4] Open Data Support. *Introduction to RDF and SPARQL*. URL: www.europeandataportal.eu/sites/default/files/d2.1.2_training_module_1.3_introduction_to_rdf_sparql_en_edp.pdf. [Online; accessed: 19.12.2020].

[5] W3C. *RDF - Semantic Web Standards*. URL: https://www.w3.org/RDF/. [Online; accessed: 11.12.2020].

[6] Wikipedia Contributors. *Semantic Triple*. URL: https://en.wikipedia.org/wiki/Semantic_triple. [Online; accessed: 11.12.2020].

[7] W3C. *Turtle - Language Feature*. URL: https://www.w3.org/TR/turtle/#language-features. [Online; accessed: 12.12.2020].

[8] Ontotext-AD. *Ontotext-AD/graphdb.js*. URL: https://github.com/Ontotext-AD/graphdb.js. [Online; accessed: 18.12.2020].