



Ontologies and the Semantic Web – CSE488

Submitted by:

Amr Essam Kamal	19P5641
Omar Tarek Mohamed	19P2772
Shehab Adel Ramadan	19P4512
Omar Salah abd Elkader	19P4606
Mohamed Reda Mohamed	19P4160

Problem Description

Overview

Creating a Semantic Web project involves designing an ontology, populating it with data, querying the ontology, manipulating it using tools like Jena, and finally building a Java application to interact with the ontology. In this project, we focus on modeling a movie ontology, populating it with relevant data, querying it using SPARQL, manipulating it using Jena, and building a Java application for movie search based on various criteria such as actors, directors, and genres.

Objectives

The primary objectives of this project are as follows:

1. **Modeling the Ontology:** Define classes, properties, and relationships to represent movies, persons (actors, directors, writers), genres, and other relevant concepts using Protégé.
2. **Populating the Ontology:** Populate the ontology with sample data for movies and persons.
3. **Querying the Ontology:** Write SPARQL queries to retrieve specific information from the ontology such as listing actors, directors, writers, movies by genre, etc.
4. **Manipulating the Ontology:** Use Jena to manipulate the ontology, perform inference, and execute queries programmatically.
5. **Java Application Development:** Develop a Java application that interacts with the ontology, allowing users to search for movies based on various criteria.

Ontology Analysis

1. Entities

This ontology defines several key classes and individuals within the domain of movies. Here's a breakdown:

Classes

Person: Represents people involved in movies, with subclasses for specific roles like Actor, Director, and Writer.

Movie: Represents movies, containing information like title, year, genre, and people involved.

Genre: Represents movie genres like Action, Comedy, Crime, etc.

Individuals

People: Includes specific individuals like Edgar Wright, Quentin Tarantino, Uma Thurman, etc., each categorized under their respective roles (Actor, Director, Writer).

Movies: Includes specific movie titles like Pulp Fiction, Shaun of the Dead, The Shawshank Redemption, etc., each with its attributes.

Genres: Lists individual genres like Action, Comedy, Thriller, etc.

Relationships between entities:

Movies are connected to People through properties like hasActor, hasDirector, hasWriter. This indicates the people involved in each movie.

People are connected to Movies through inverse properties like isActorOf, isDirectorOf, isWriterOf. This allows for navigating from a person to the movies they participated in.

Movies are associated with Genres using the hasGenre property, specifying the genre(s) of a movie.

2. Relations (Properties)

This ontology utilizes both object properties and data properties to establish relationships between entities:

Object Properties

These connect instances of classes together.

hasActor, hasDirector, hasWriter: Links a Movie to the People involved.

isActorOf, isDirectorOf, isWriterOf: Inverse properties, linking a Person to the Movies they participated in.

Data Properties

These assign data value to entities.

title, year, country, language: Describe characteristics of a Movie.

name, age, nationality, hasGenderType: Describe attributes of a Person.

3. Logic

The ontology employs OWL (Web Ontology Language) to define its structure and logic.

Restrictions

These define constraints on properties.

owl: someValuesFrom: Ensures a movie has at least one actor, director, writer, and genre.

owl: minQualifiedCardinality: Ensures each person is associated with at least one movie in each role (actor, director, writer).

Axioms

Explicit statements that define relationships and constraints.

owl: disjointWith: States that the classes "Movie" and "Person" are distinct and cannot have overlapping instances.

Inference Rules

OWL allows for reasoning and inferring new knowledge based on existing axioms and restrictions.

Overall, this ontology provides a well-structured representation of movie-related data, using OWL to define classes, properties, and logical constraints. This enables the organization and retrieval of information about movies, genres, and individuals involved in filmmaking.

Class hierarchy

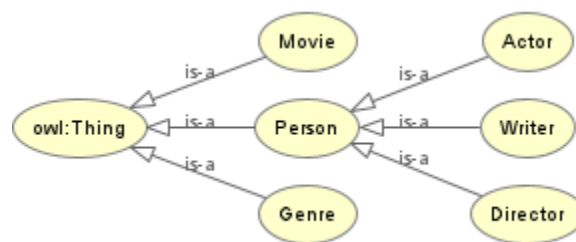


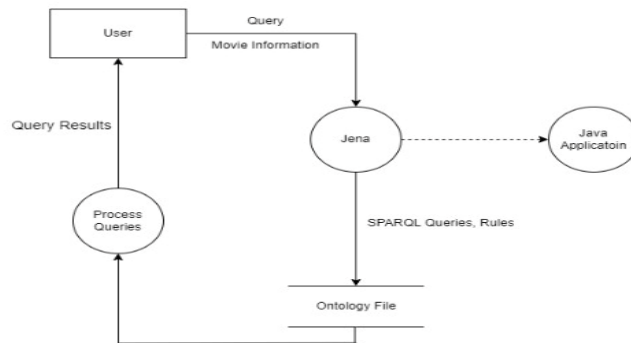
Figure 1: asserted hierarchy using OWLviz

Ontology visualization

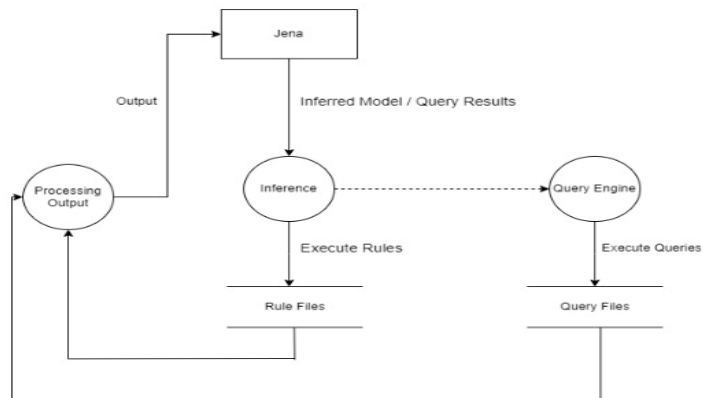
[Click here to see the full visualization \(RDF GRAPH\)](#)

Data Flow Diagram (DFD)

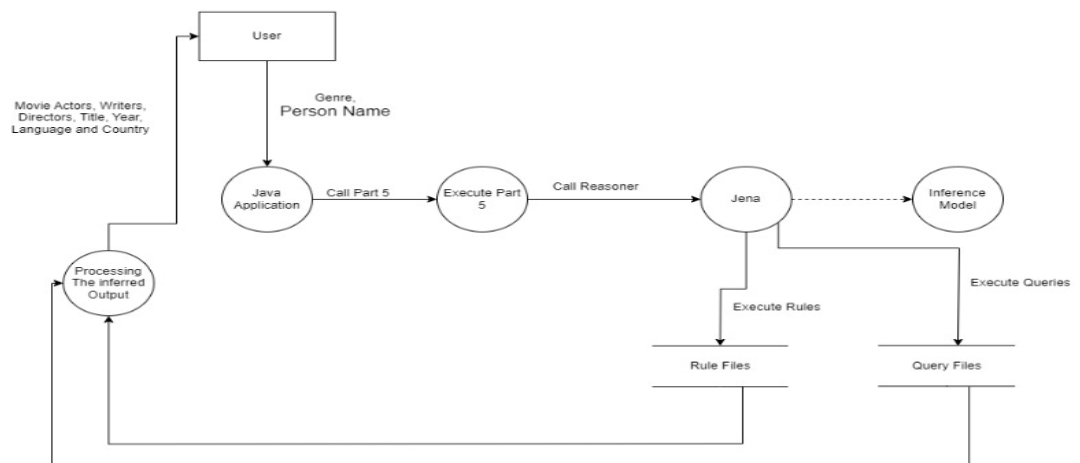
Level 0: Context Diagram



Level 1: System (Jena) Decomposition



Level 2: Java Application Decomposition



Part I & II:

[.Owl file link on drive:](https://drive.google.com/file/d/1HHtH6Y9q4xbcv42M1ecy9n3aU2HkQz/view?usp=sharing)

<https://drive.google.com/file/d/1HHtH6Y9q4xbcv42M1ecy9n3aU2HkQz/view?usp=sharing>

Part III: Querying the ontology

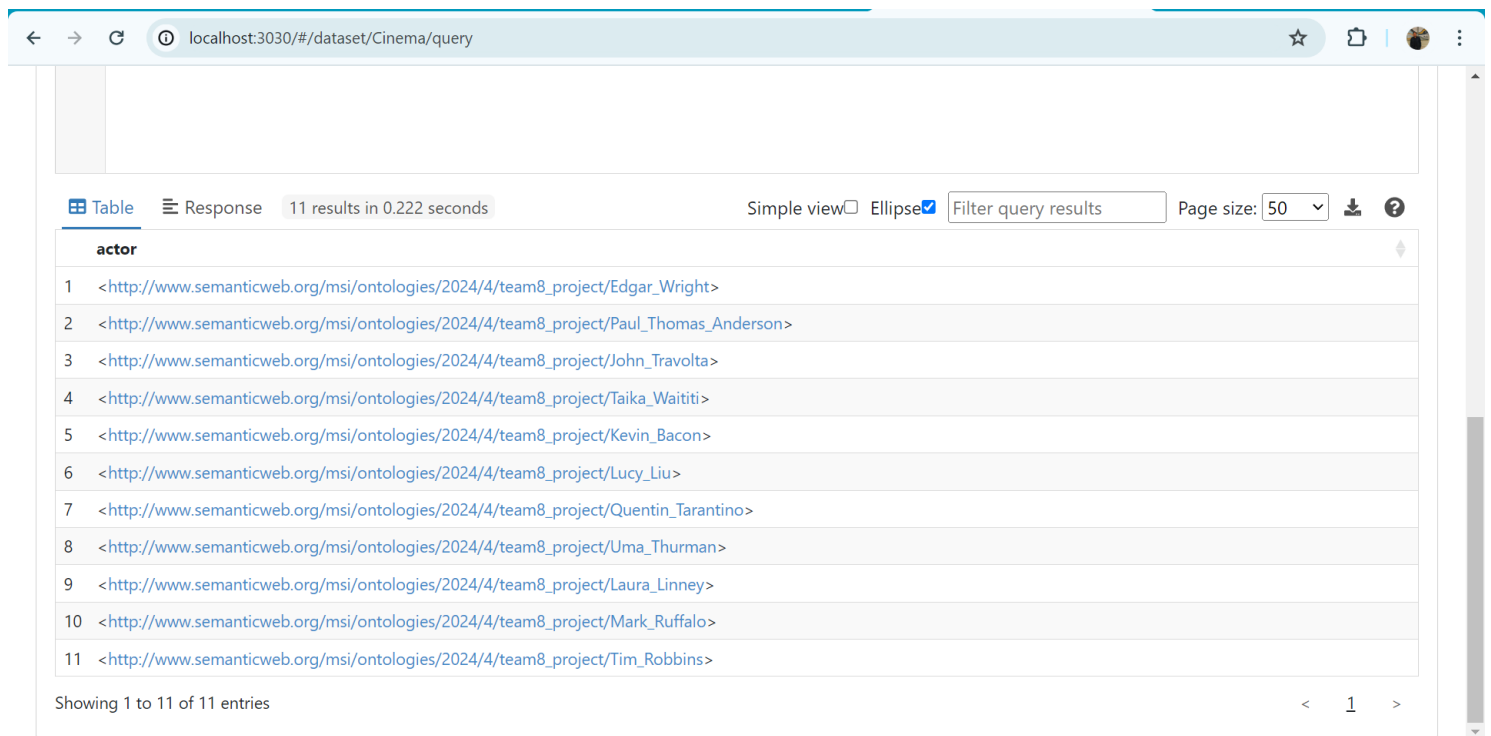
1.0 List the instances of the class Actor

Sparql:

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>

```
SELECT ?actor
WHERE {
  ?actor rdf:type :Actor .
}
```



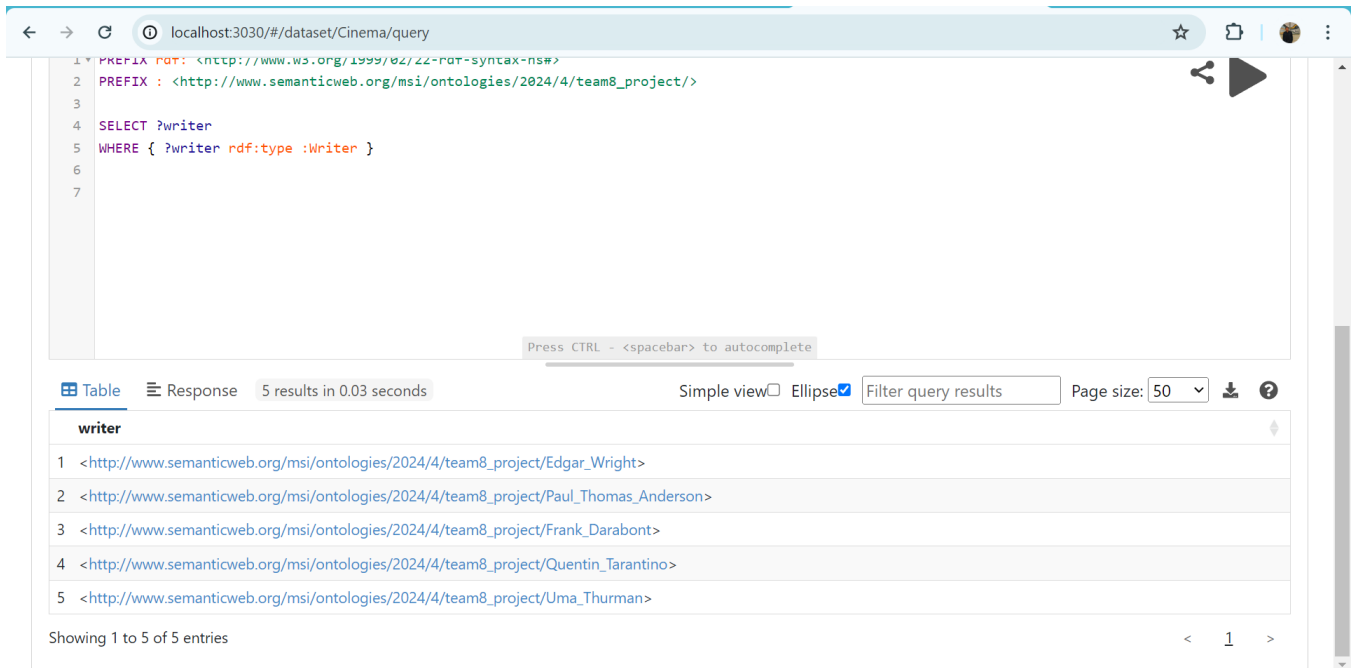
actor
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Edgar_Wright>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Paul_Thomas_Anderson>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/John_Travolta>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Taika_Waititi>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Kevin_Bacon>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Lucy_Liu>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Uma_Thurman>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Laura_Linney>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Mark_Ruffalo>
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Tim_Robbins>

Figure 2: Testing on jena first sparql

2.0 List the instances of the class Writer

Sparql query:

```
SELECT ?writer
WHERE { ?writer rdf:type :Writer }
```



localhost:3030/#/dataset/Cinema/query

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
3
4 SELECT ?writer
5 WHERE { ?writer rdf:type :Writer }
6
7
```

Press CTRL - <spacebar> to autocomplete

Table Response 5 results in 0.03 seconds Simple view Ellipse Filter query results Page size: 50

writer
1 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Edgar_Wright>
2 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Paul_Thomas_Anderson>
3 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Frank_Darabont>
4 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino>
5 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Uma_Thurman>

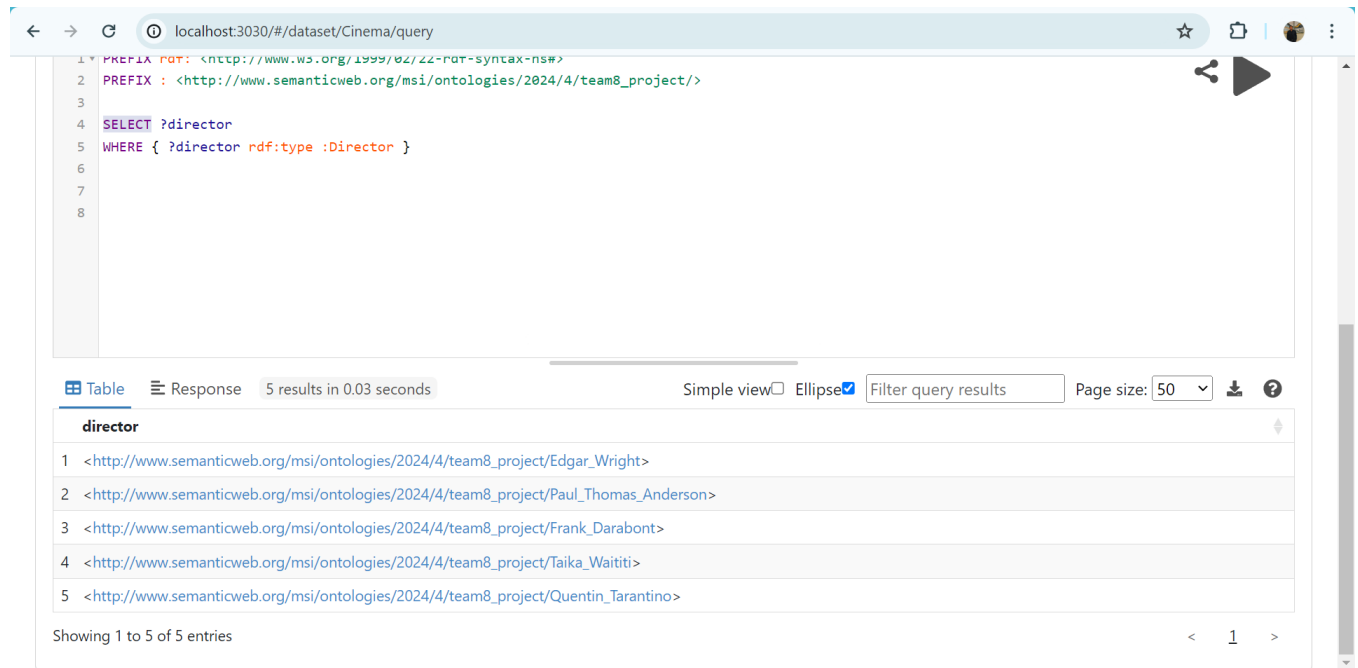
Showing 1 to 5 of 5 entries

Figure 3: Testing on jena second SPARQL query

3.0 List the instances of the class Director

Sparql query:

```
SELECT ?director
WHERE { ?director rdf:type :Director }
```



The screenshot shows a web browser at the URL `localhost:3030/#/dataset/Cinema/query`. The SPARQL query editor contains the following query:

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
3
4 SELECT ?director
5 WHERE { ?director rdf:type :Director }
6
7
8
```

Below the query editor, the results are displayed in a table view. The table has a header row with the column name `director`. There are 5 results, each represented by an IRI:

director
1 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Edgar_Wright>
2 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Paul_Thomas_Anderson>
3 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Frank_Darabont>
4 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Taika_Waititi>
5 <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino>

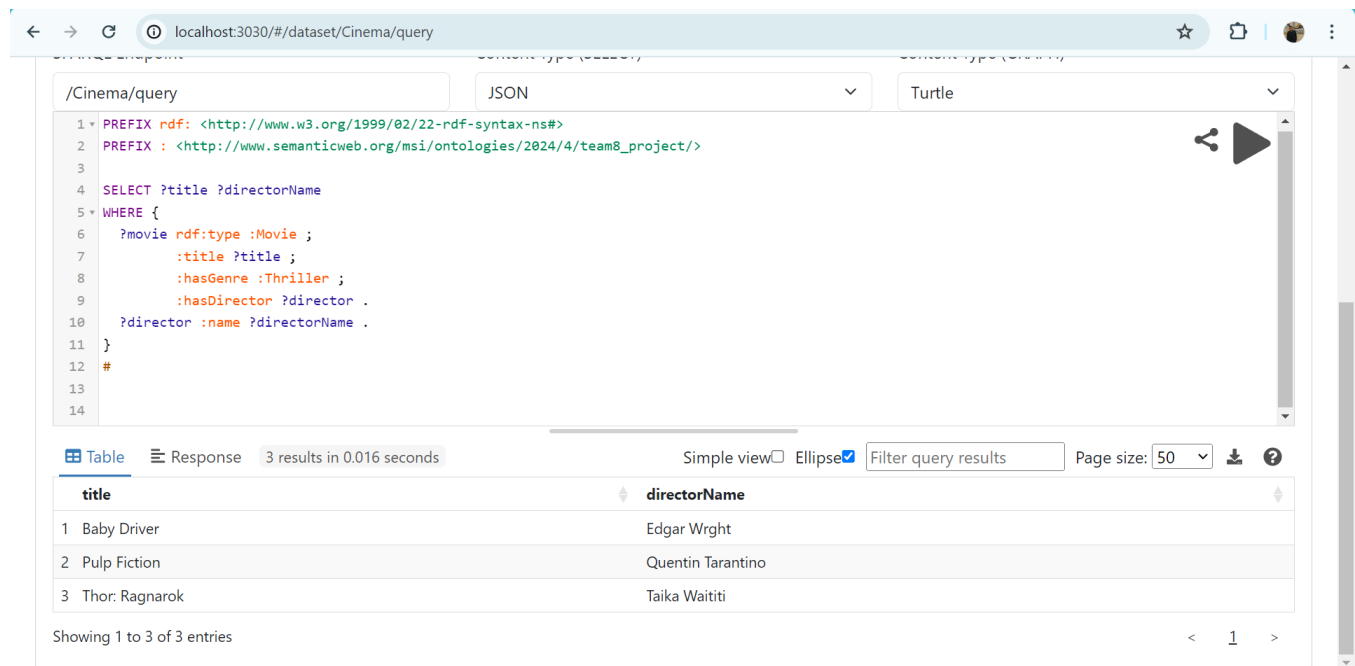
At the bottom of the results, it says "Showing 1 to 5 of 5 entries".

Figure 4: Testing on jena third SPARQL query

4.0 List the name of all Thriller movies and their directors

SPARQL query:

```
SELECT ?title ?directorName
WHERE {
  ?movie rdf:type :Movie ;
    :title ?title ;
    :hasGenre :Thriller ;
    :hasDirector ?director .
  ?director :name ?directorName .
}
```



The screenshot shows a web browser at the URL `localhost:3030/#/dataset/Cinema/query`. The interface includes a text area for the SPARQL query, a dropdown for the result format (set to JSON), and a dropdown for the result language (set to Turtle). The query is as follows:

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
3
4 SELECT ?title ?directorName
5 WHERE {
6   ?movie rdf:type :Movie ;
7     :title ?title ;
8     :hasGenre :Thriller ;
9     :hasDirector ?director .
10  ?director :name ?directorName .
11 }
12 #
13
14
```

Below the query editor, the results are displayed in a table view. The table has two columns: `title` and `directorName`. It shows 3 results in 0.016 seconds. The results are:

title	directorName
1 Baby Driver	Edgar Wrght
2 Pulp Fiction	Quentin Tarantino
3 Thor: Ragnarok	Taika Waititi

At the bottom, it indicates "Showing 1 to 3 of 3 entries" and includes navigation controls.

Figure 5: Testing on jena the fourth SPARQL query

5.0 List the name of all Crime Thriller movies

SPARQL query:

```
SELECT ?title
WHERE {
  ?movie rdf:type :Movie ;
         :title ?title ;
         :hasGenre :Crime, :Thriller .
}
```

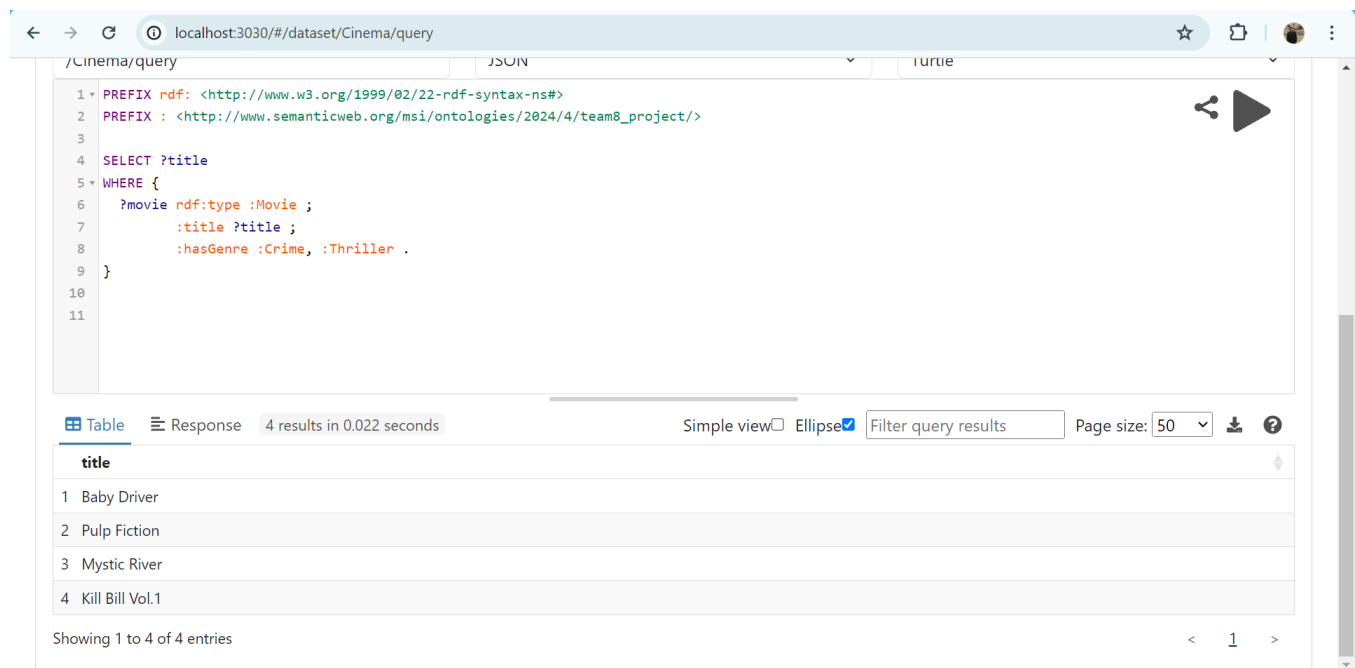


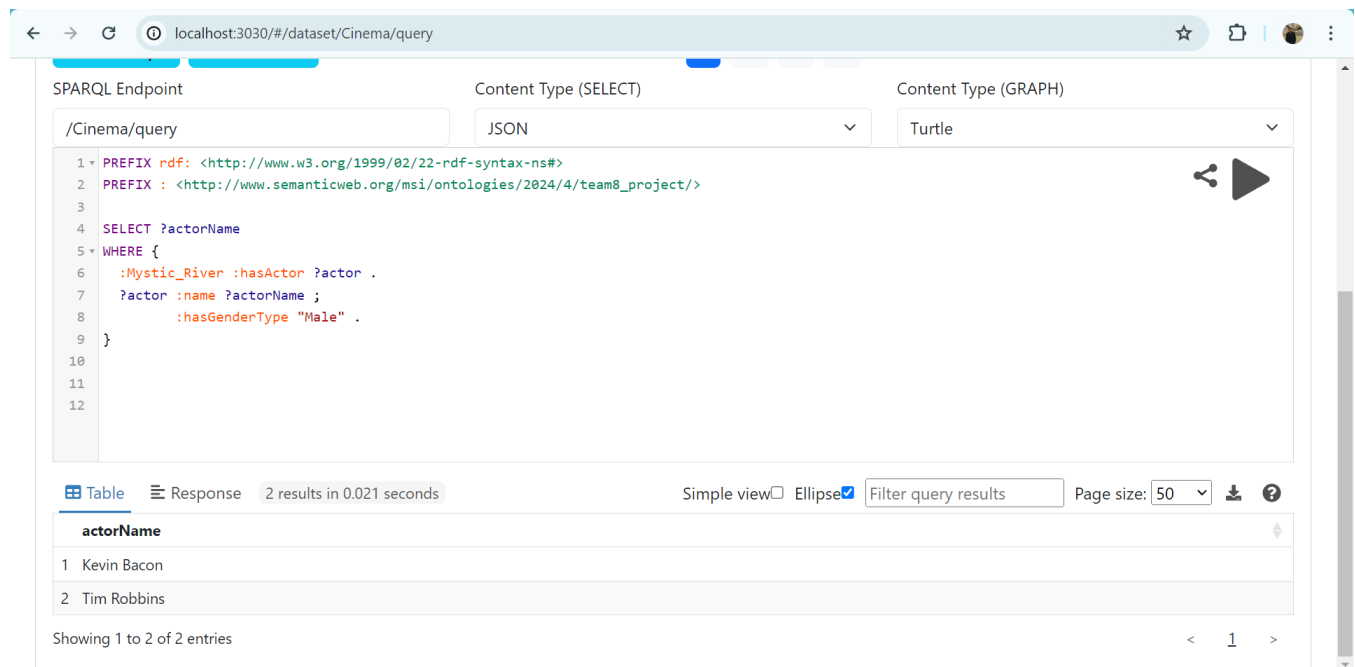
Figure 6: Testing on jena the fifth SPARQL query

6.0 List the male actors in a specific film (replace "Movie Name" with the actual movie title)

SPARQL query:

Note: We try it for every movie for example: Mystic_River

```
SELECT ?actorName
WHERE {
  :Mystic_River :hasActor ?actor .
  ?actor :name ?actorName ;
         :hasGenderType "Male" .
}
```



The screenshot shows a web browser at the URL `localhost:3030/#/dataset/Cinema/query`. The interface includes a SPARQL Endpoint field with `/Cinema/query`, a Content Type (SELECT) dropdown set to `JSON`, and a Content Type (GRAPH) dropdown set to `Turtle`. The SPARQL query is entered in a text area and is as follows:

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
3
4 SELECT ?actorName
5 WHERE {
6   :Mystic_River :hasActor ?actor .
7   ?actor :name ?actorName ;
8         :hasGenderType "Male" .
9 }
10
11
12
```

Below the query area, the results are displayed in a table view. The table has a header `actorName` and two rows of results:

	actorName
1	Kevin Bacon
2	Tim Robbins

At the bottom, it indicates "Showing 1 to 2 of 2 entries".

Figure 7: Testing on jena with Mystic_River movie

7.0 How many movies have both "Action" and "Thriller" as genres

SPARQL query:

```
SELECT (COUNT(?movie) AS ?count)
WHERE {
  ?movie rdf:type :Movie ;
         :hasGenre :Action, :Thriller .
}
```

The screenshot shows a web browser at `localhost:3030/#/dataset/Cinema/query`. The interface includes tabs for "Selection of triples" and "Selection of classes", and format buttons for "rdf", "rdfs", "owl", and "xsd". The "SPARQL Endpoint" is `/Cinema/query`. The "Content Type (SELECT)" is set to "JSON" and "Content Type (GRAPH)" is set to "Turtle". The query is as follows:

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
3
4 SELECT (COUNT(?movie) AS ?count)
5 WHERE {
6   ?movie rdf:type :Movie ;
7         :hasGenre :Action, :Thriller .
8 }
9
10
11
```

Below the query editor, the results are displayed in a table view. The table has a header "count" and one row with the value "3". The status bar indicates "1 result in 0.042 seconds".

count
3

Showing 1 to 1 of 1 entries

Figure 8: Testing on Jena that we have 3 Action Thriller movies

8.0 List all the movies written by a specific writer (replace "Writer Name" with the actual writer name):

Note: We try it for every writer for example: Frank_Darabont

SPARQL query:

```
SELECT ?title
WHERE {
  ?movie rdf:type :Movie ;
    :title ?title ;
    :hasWriter :Frank_Darabont .
}
```

The screenshot shows a web browser at `localhost:3030/#/dataset/cinema/query`. The interface includes tabs for "Selection of triples" and "Selection of classes", and buttons for "rdf", "rdfs", "owl", and "xsd". The "SPARQL Endpoint" is set to `/cinema/`. The "Content Type (SELECT)" is set to "JSON", and the "Content Type (GRAPH)" is set to "Turtle". The SPARQL query is entered in a text area:

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
3
4 SELECT ?title
5 WHERE {
6   ?movie rdf:type :Movie ;
7     :title ?title ;
8     :hasWriter :Frank_Darabont .
9 }
10
11
12
13
```

Below the query area, there are tabs for "Table" and "Response". The "Table" tab is selected, showing a table with one entry:

title
1 The Shawshank Redemption

At the bottom, it says "Showing 1 to 1 of 1 entries".

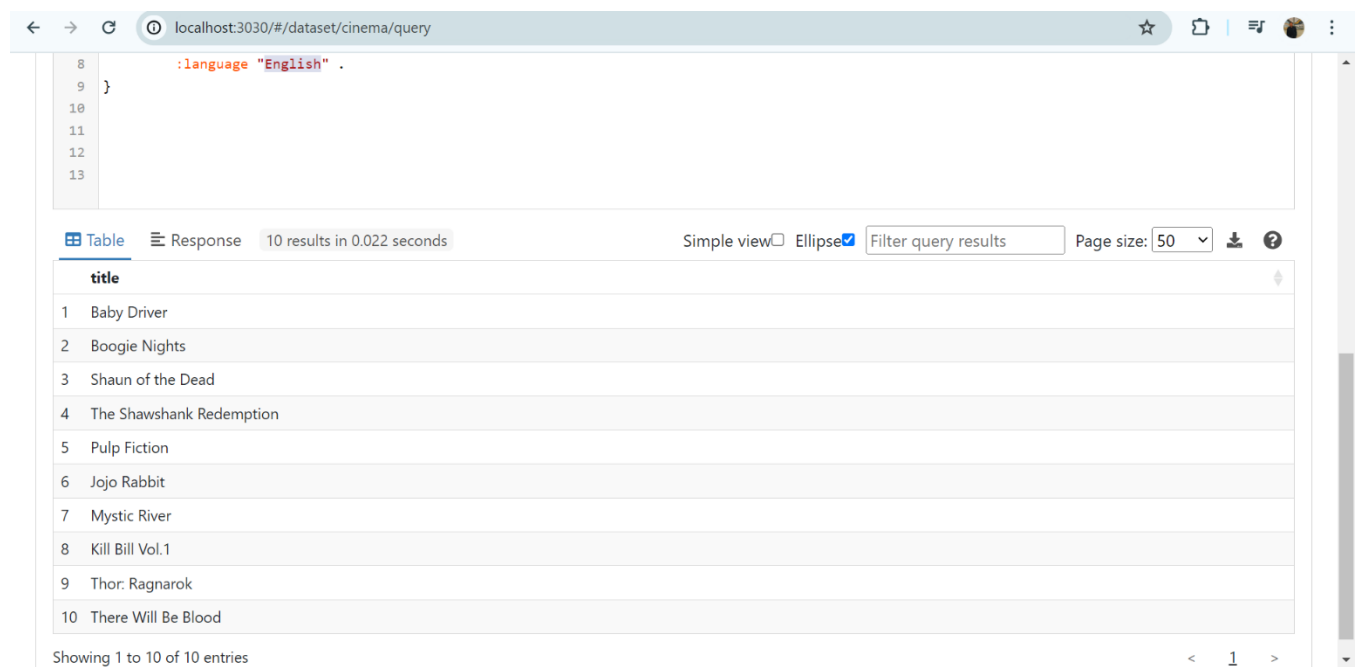
Figure 9: Testing on jena with Frank_Darabont as an example

9.0 Find movies with a certain language (replace "Language" with the actual language):

Note: We try it for every language for example: English

SPARQL query:

```
SELECT ?title
WHERE {
  ?movie rdf:type :Movie ;
    :title ?title ;
    :language "English" .
}
```



The screenshot shows a web browser at the URL `localhost:3030/#/dataset/cinema/query`. The SPARQL query editor contains the following query:

```
8      :language "English" .
9    }
10
11
12
13
```

Below the query editor, the interface shows "10 results in 0.022 seconds". The results are displayed in a table with the following columns: "title".

	title
1	Baby Driver
2	Boogie Nights
3	Shaun of the Dead
4	The Shawshank Redemption
5	Pulp Fiction
6	Jojo Rabbit
7	Mystic River
8	Kill Bill Vol.1
9	Thor: Ragnarok
10	There Will Be Blood

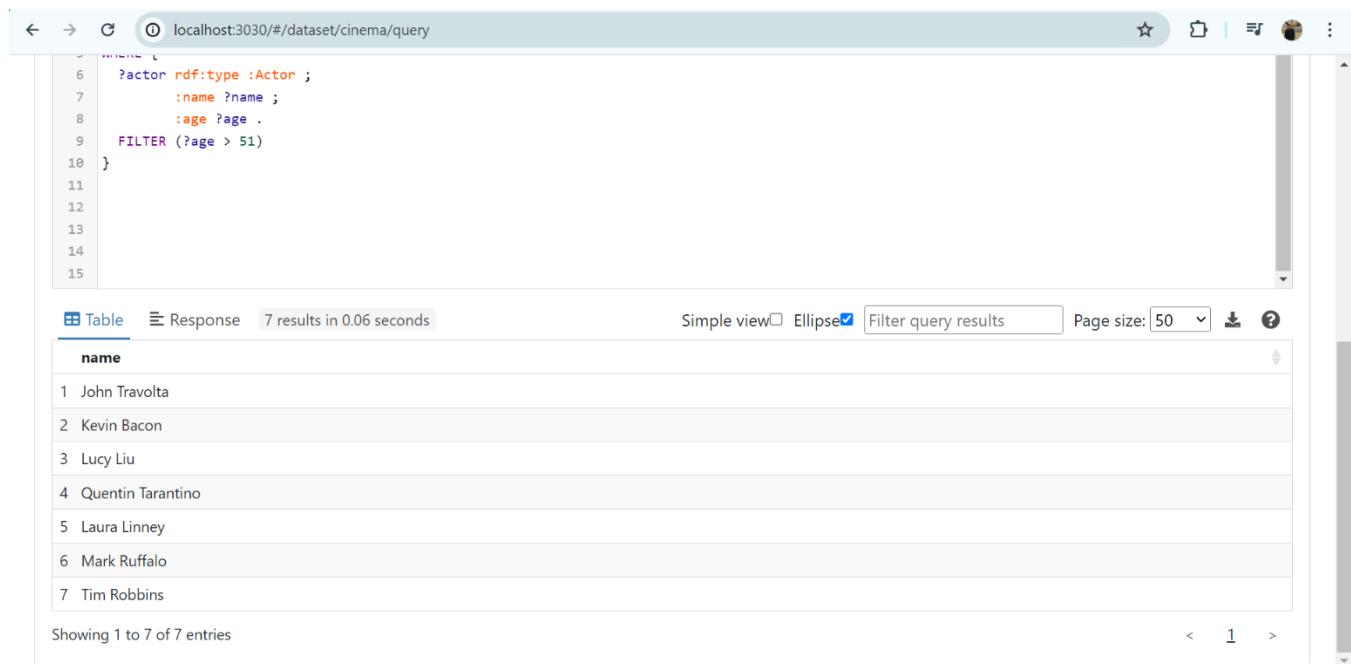
At the bottom, it says "Showing 1 to 10 of 10 entries".

Figure 10: Testing on jena to get all english language movies

10.0 List the name of Actors older than 51 years:

SPARQL query:

```
SELECT ?name
WHERE {
  ?actor rdf:type :Actor ;
    :name ?name ;
    :age ?age .
  FILTER (?age > 51)
}
```



localhost:3030/#/dataset/cinema/query

```
6  ?actor rdf:type :Actor ;
7    :name ?name ;
8    :age ?age .
9  FILTER (?age > 51)
10 }
11
12
13
14
15
```

Table Response 7 results in 0.06 seconds Simple view ☐ Ellipse ☒ Filter query results Page size: 50

	name
1	John Travolta
2	Kevin Bacon
3	Lucy Liu
4	Quentin Tarantino
5	Laura Linney
6	Mark Ruffalo
7	Tim Robbins

Showing 1 to 7 of 7 entries < 1 >

Figure 11: Testing on jena to get name of Actors older than 51 years

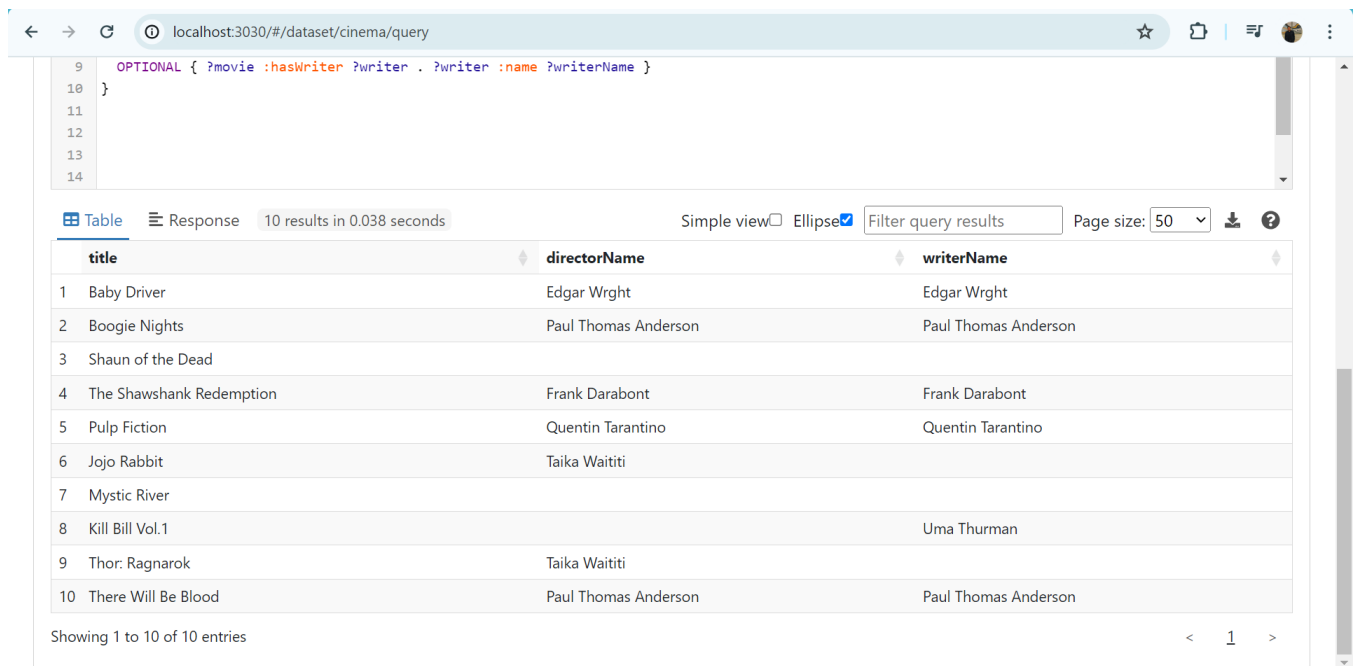
Proposed SPARQL Queries

1.0 Query with 2 Optional Graph Patterns

SPARQL query:

This query retrieves movies and optionally their directors and writers

```
SELECT ?title ?directorName ?writerName
WHERE {
  ?movie rdf:type :Movie ;
  :title ?title .
  OPTIONAL { ?movie :hasDirector ?director . ?director :name ?directorName }
  OPTIONAL { ?movie :hasWriter ?writer . ?writer :name ?writerName }
}
```



localhost:3030/#/dataset/cinema/query

```
9  OPTIONAL { ?movie :hasWriter ?writer . ?writer :name ?writerName }
10 }
11
12
13
14
```

Table Response 10 results in 0.038 seconds Simple view Ellipse Filter query results Page size: 50

	title	directorName	writerName
1	Baby Driver	Edgar Wrght	Edgar Wrght
2	Boogie Nights	Paul Thomas Anderson	Paul Thomas Anderson
3	Shaun of the Dead		
4	The Shawshank Redemption	Frank Darabont	Frank Darabont
5	Pulp Fiction	Quentin Tarantino	Quentin Tarantino
6	Jojo Rabbit	Taika Waititi	
7	Mystic River		
8	Kill Bill Vol.1		Uma Thurman
9	Thor: Ragnarok	Taika Waititi	
10	There Will Be Blood	Paul Thomas Anderson	Paul Thomas Anderson

Showing 1 to 10 of 10 entries < 1 >

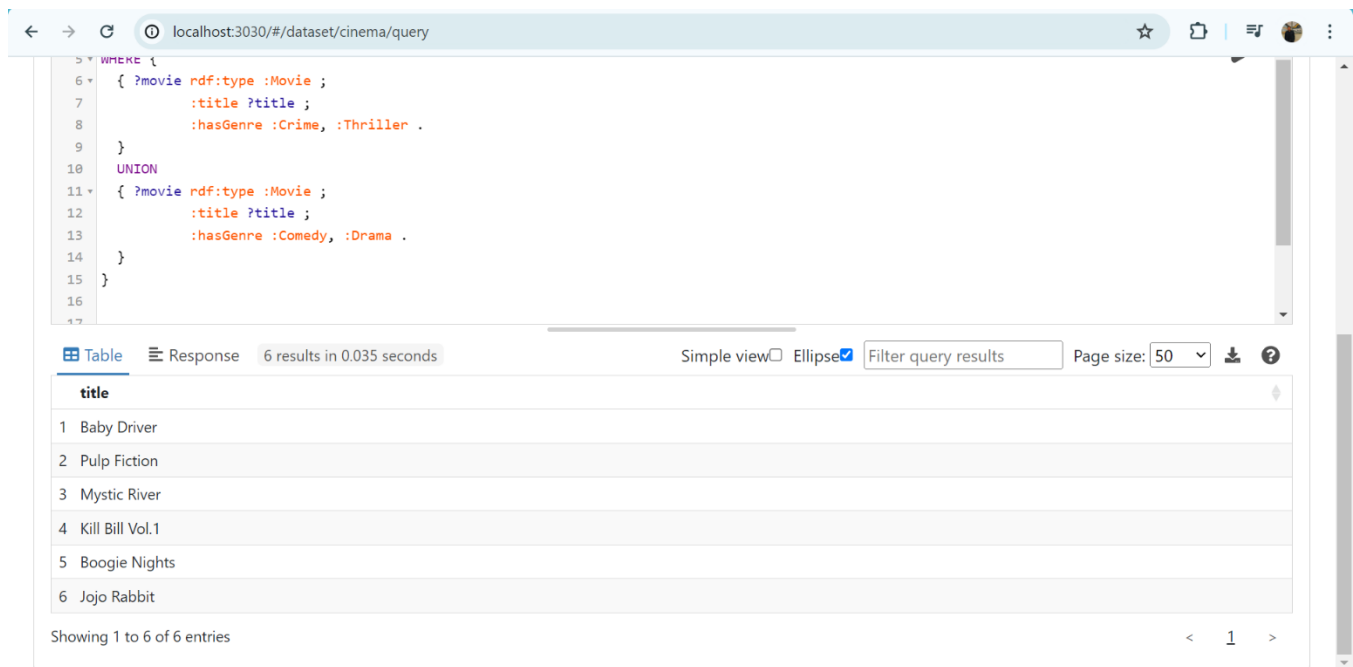
Figure 12: Testing the query on jena that retrieves movies and optionally their directors and writers

2.0 Query with 2 alternatives and conjunctions:

This query finds movies that are either Crime Thriller or Comedy Drama:

SPARQL query:

```
SELECT ?title
WHERE {
  { ?movie rdf:type :Movie ;
    :title ?title ;
    :hasGenre :Crime, :Thriller .
  }
  UNION
  { ?movie rdf:type :Movie ;
    :title ?title ;
    :hasGenre :Comedy, :Drama .
  }
}
```



The screenshot shows a web browser window with the URL `localhost:3030/#/dataset/cinema/query`. The browser displays a SPARQL query editor with the following query:

```
5 WHERE {
6   { ?movie rdf:type :Movie ;
7     :title ?title ;
8     :hasGenre :Crime, :Thriller .
9   }
10  UNION
11  { ?movie rdf:type :Movie ;
12    :title ?title ;
13    :hasGenre :Comedy, :Drama .
14  }
15 }
16
17
```

Below the query editor, the interface shows the results in a table view. The table has one column labeled `title` and contains 6 rows of results:

title
1 Baby Driver
2 Pulp Fiction
3 Mystic River
4 Kill Bill Vol.1
5 Boogie Nights
6 Jojo Rabbit

At the bottom of the results section, it says "Showing 1 to 6 of 6 entries". The interface also includes a "Simple view" checkbox, an "Ellipse" checkbox (checked), a "Filter query results" input field, and a "Page size" dropdown set to 50.

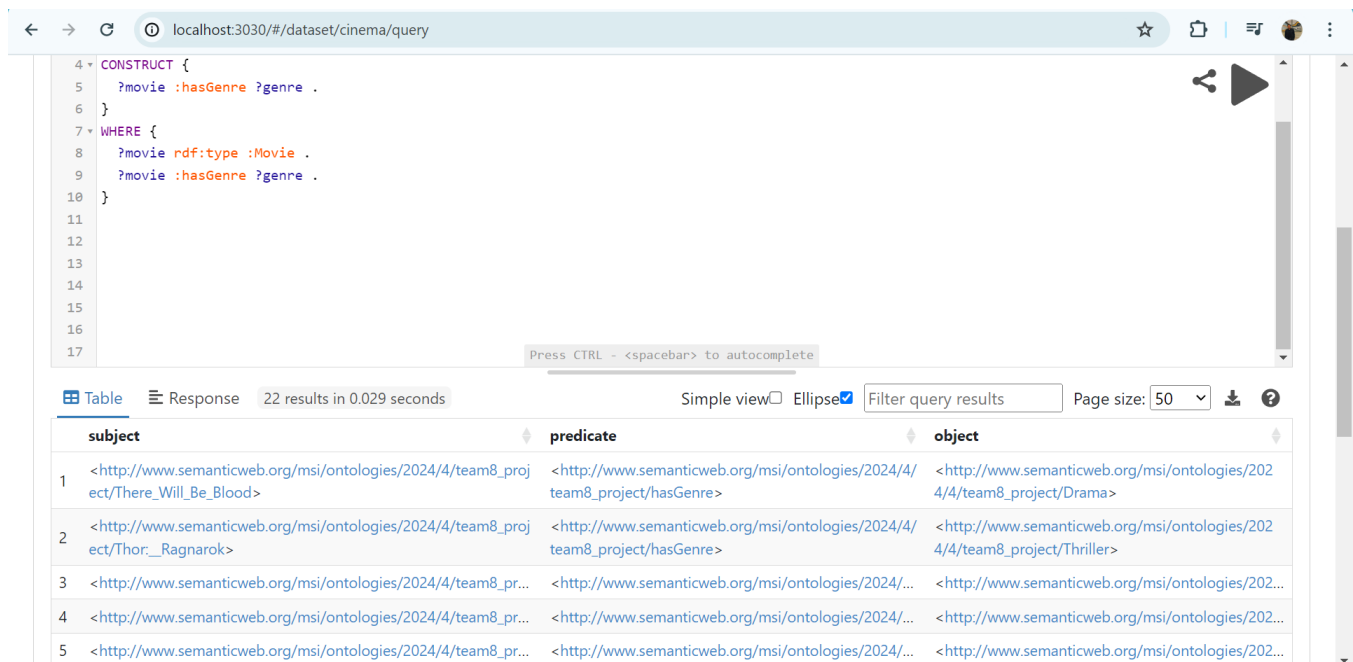
Figure 13: Testing on jena with 2 alternatives and conjunctions

3.0 CONSTRUCT Query:

This query constructs triples about movies and their genres:

SPARQL query

```
CONSTRUCT {  
  ?movie :hasGenre ?genre .  
}  
WHERE {  
  ?movie rdf:type :Movie .  
  ?movie :hasGenre ?genre .  
}
```



localhost:3030/#/dataset/cinema/query

```
4 CONSTRUCT {  
5   ?movie :hasGenre ?genre .  
6 }  
7 WHERE {  
8   ?movie rdf:type :Movie .  
9   ?movie :hasGenre ?genre .  
10 }  
11  
12  
13  
14  
15  
16  
17
```

Press CTRL - <spacebar> to autocomplete

Table Response 22 results in 0.029 seconds Simple view Ellipse Filter query results Page size: 50

	subject	predicate	object
1	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/There_Will_Be_Blood>	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/hasGenre>	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Drama>
2	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Thor_Ragnarok>	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/hasGenre>	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Thriller>
3	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_pr...	<http://www.semanticweb.org/msi/ontologies/2024/...	<http://www.semanticweb.org/msi/ontologies/202...
4	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_pr...	<http://www.semanticweb.org/msi/ontologies/2024/...	<http://www.semanticweb.org/msi/ontologies/202...
5	<http://www.semanticweb.org/msi/ontologies/2024/4/team8_pr...	<http://www.semanticweb.org/msi/ontologies/2024/...	<http://www.semanticweb.org/msi/ontologies/202...

Figure 14: Testing the query with construction

4.0 ASK Query:

This query checks if there are any movies from the year 2003

SPARQL query

```
ASK {  
  ?movie rdf:type :Movie ;  
  :year 2003 .  
}
```

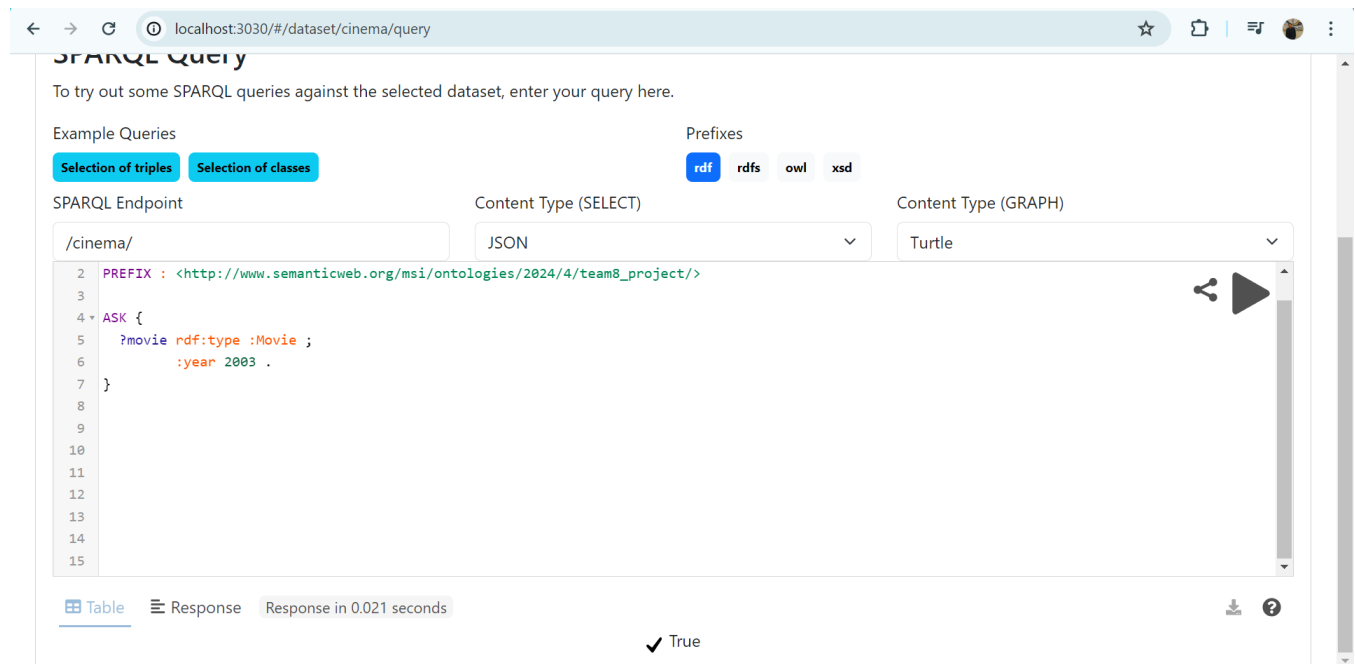


Figure 15: Testing in jena and the answer is "True"

5.0 DESCRIBE Query:

This query describes the resource "Quentin_Tarantino":

SPARQL query

DESCRIBE :Quentin_Tarantino

The screenshot shows the Apache Jena Fuseki UI interface. The browser address bar indicates the URL is `localhost:3030/#/dataset/cinema/query`. The query editor contains the following SPARQL query:

```
2 PREFIX : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
3
4 DESCRIBE :Quentin_Tarantino
5
6
7
8
9
10
11
12
```

Below the query editor, the results are displayed in a table view. The table has three columns: **subject**, **predicate**, and **object**. The results show 11 rows of data, with the first row being the most detailed:

	subject	predicate	object
1	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/nationality></code>	<code>"American"^^<http://www.w3.org/2001/XMLSchema#string></code>
2	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/name></code>	<code>"Quentin Tarantino"^^<http://www.w3.org/2001/XMLSchema#str...></code>
3	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/sex></code>	<code>"Male"^^<http://www.w3.org/2001/XMLSchema#string></code>
4	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/age></code>	<code>"53"^^<http://www.w3.org/2001/XMLSchema#integer></code>
5	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/occupation></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/occupation></code>
6	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/occupation></code>	<code><http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/occupation></code>

Figure 16: Testing on jena to describe all things about Quentin Tarantino

Part IV: Manipulating the ontology using Jena

1. Create a java program (Jena1.java) that loads the ontology and displays all the Persons (without using queries, without inference).

code:

```
import rdflib
from rdflib.namespace import OWL, RDF, RDFS
from SPARQLWrapper import SPARQLWrapper, JSON
from rdflib import Graph, Namespace, URIRef

# Load the ontology
g = Graph()
g.parse(r"C:\Users\Dell\Desktop\Cinema.rdf")

# Define the URIs for the subclasses
director_uri = URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Director")
writer_uri = URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Writer")
actor_uri = URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Actor")

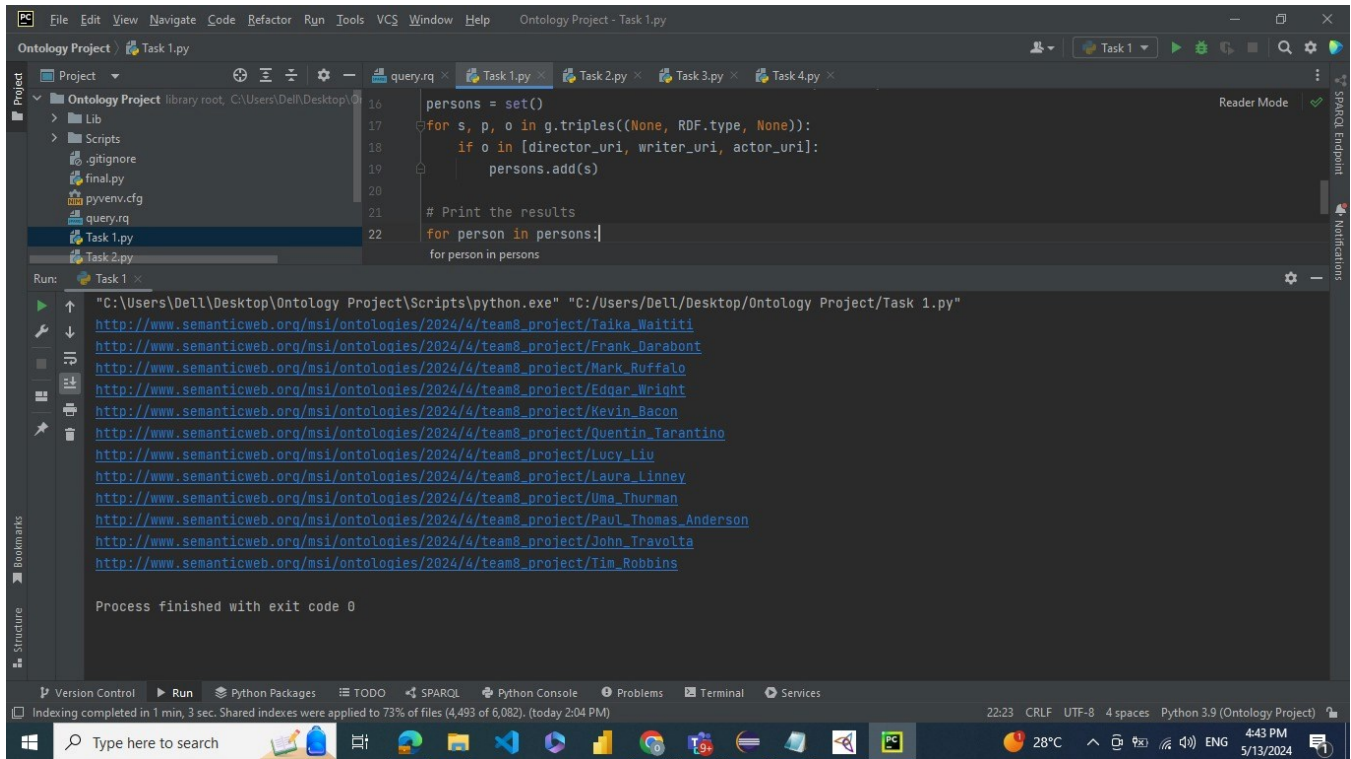
# Find individuals that are instances of Director, Writer, or Actor
persons = set()
for s, p, o in g.triples((None, RDF.type, None)):
    if o in [director_uri, writer_uri, actor_uri]:
        persons.add(s)

# Print the results
for person in persons:
    print(person)
```

Output:

```
"C:\Users\Dell\Desktop\Ontology Project\Scripts\python.exe" "C:/Users/Dell/Desktop/Ontology
Project/Task 1.py"
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Laura_Linney
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Taika_Waititi
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Uma_Thurman
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Kevin_Bacon
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Tim_Robbins
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Edgar_Wright
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Frank_Darabont
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Lucy_Liu
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Paul_Thomas_Anderson
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/John_Travolta
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Mark_Ruffalo
```

Screenshot



The screenshot shows an IDE window titled "Ontology Project - Task 1.py". The editor displays a Python script that iterates over a set of URIs and prints them. The output window shows the execution of the script, displaying a list of URIs and the message "Process finished with exit code 0".

```
16 persons = set()
17 for s, p, o in g.triples((None, RDF.type, None)):
18     if o in [director_uri, writer_uri, actor_uri]:
19         persons.add(s)
20
21 # Print the results
22 for person in persons:
    for person in persons
```

Run: Task 1

```
"C:\Users\Dell\Desktop\Ontology Project\Scripts\python.exe" "C:/Users/Dell/Desktop/Ontology Project/Task 1.py"
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Taika_Waititi
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Frank_Darabont
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Mark_Ruffalo
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Edgar_Wright
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Kevin_Bacon
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Lucy_Liu
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Laura_Linney
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Uma_Thurman
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Paul_Thomas_Anderson
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/John_Travolta
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Tim_Robbins

Process finished with exit code 0
```

Version Control Run Python Packages TODO SPARQL Python Console Problems Terminal Services

Indexing completed in 1 min, 3 sec. Shared indexes were applied to 73% of files (4,493 of 6,082). (today 2:04 PM)

22:23 CRLF UTF-8 4 spaces Python 3.9 (Ontology Project)

Type here to search 28°C 4:43 PM 5/13/2024

Figure 17: Screenshot of the output of task 1

2. Create a java program (Jena2.java) that loads the ontology and displays all the Persons (using a query, without inference). Create the used query in text file under the data folder.

Code:

```
from rdflib import Graph

# Load the ontology
g = Graph()
g.parse(r"C:\Users\Dell\Desktop\Cinema.rdf")

# Load the SPARQL query
with open(r"C:\Users\Dell\Desktop\Ontology Project\query.rq", "r") as f:
    query_string = f.read()

# Execute the query
results = g.query(query_string)

# Use a set to store unique results
unique_results = set()

# Store the results in the set
for row in results:
    unique_results.add(row.person)

# Print the unique results
for person in unique_results:
    print(person)
```

Output:

```
"C:\Users\Dell\Desktop\Ontology Project\Scripts\python.exe" "C:/Users/Dell/Desktop/Ontology
Project/Task 2.py"
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Lucy_Liu
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Edgar_Wright
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Paul_Thomas_Anderson
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Tim_Robbins
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Taika_Waititi
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Quentin_Tarantino
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Uma_Thurman
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Mark_Ruffalo
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Kevin_Bacon
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Frank_Darabont
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Laura_Linney
http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/John_Travolta
```

Query:

```
SELECT ?person
WHERE {
  ?person rdf:type/rdfs:subClassOf*
  <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Person>
}
```

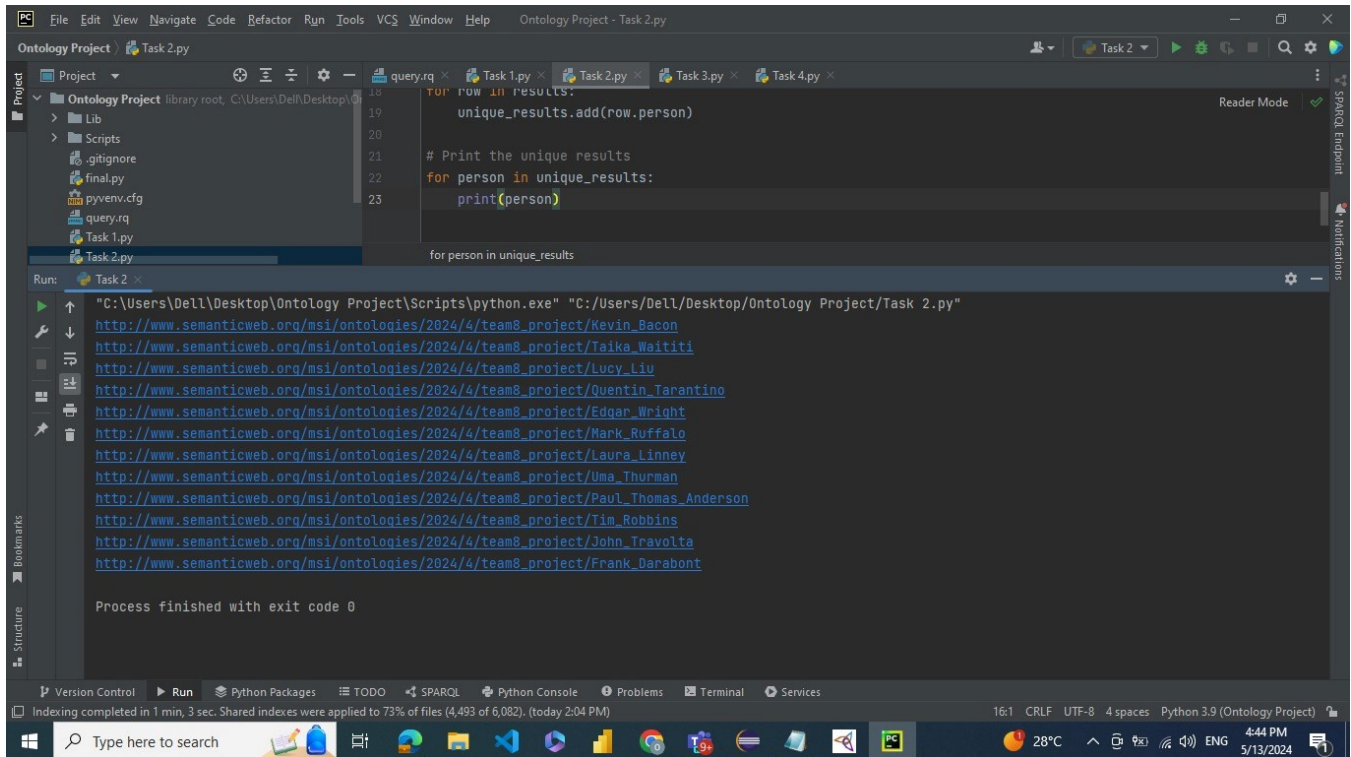


Figure 18: Screenshot of output of task 2

3. Create a java program (Jena3.java) that loads the ontology and displays all the Actors (without using queries, using inference). To load the inferred model, use the `JenaEngine.readInferencedModelFromRuleFile` method and use owl rules

Code

```
package pack;
import tools.JenaEngine;

import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.Property;
import org.apache.jena.rdf.model.ResIterator;
import org.apache.jena.rdf.model.Resource;

public class Task3 {

    private Model model;
    private String file;
    private String namespace;

    Task3(String path){
        this.namespace = "";
        this.file = path;
        this.model = JenaEngine.readModel(path);
        if(model != null ){
            namespace = model.getNsPrefixURI("");
        }
    }

    public void readActor(){
        Model ourmodel = JenaEngine.readInferencedModelFromRuleFile(model,
"Ontology/rule3.txt");
        Property pname = model.getProperty(namespace + "name");
        ResIterator iter =ourmodel.listResourcesWithProperty(pname);
        for (; iter.hasNext();) {
            Resource i = iter.next();
            JenaEngine.readRsDataType(ourmodel, namespace, i, "name");
        }
    }
}
```

rule

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

@prefix : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/> .

[rule: (?per rdf:type :Actor) ->(?per rdf:type A)]

Output

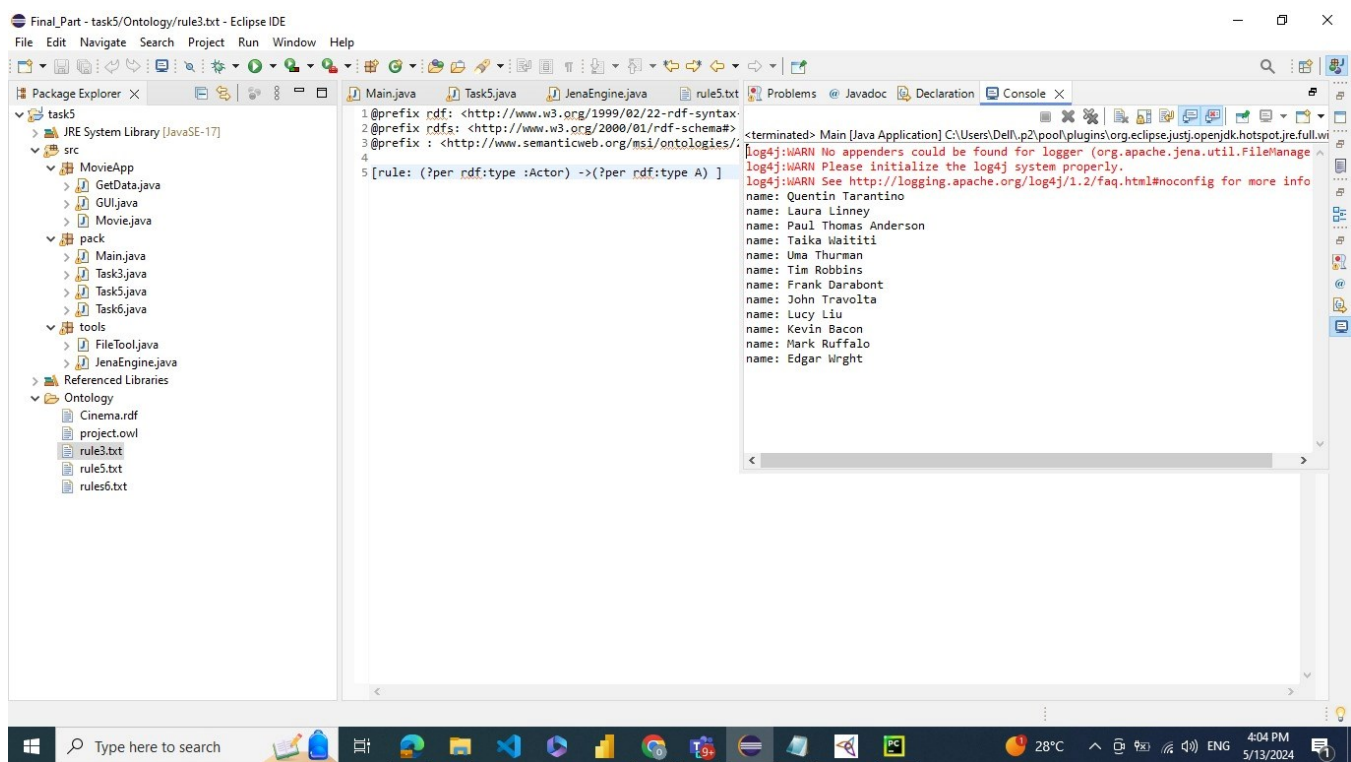


Figure 19: Screenshot of the output task 3

**4. Create a java program (Jena4.java) that: a. Reads a name of a movie
b. If it doesn't exist displays an error message c. Else, display its
year, country, genres and actors**

Code

```
from rdflib import Graph, URIRef
from rdflib.namespace import OWL, RDF, RDFS

def get_movie_info(movie_title):
    # Load the ontology
    g = Graph()
    g.parse(r"C:\Users\Dell\Desktop\Cinema.rdf")

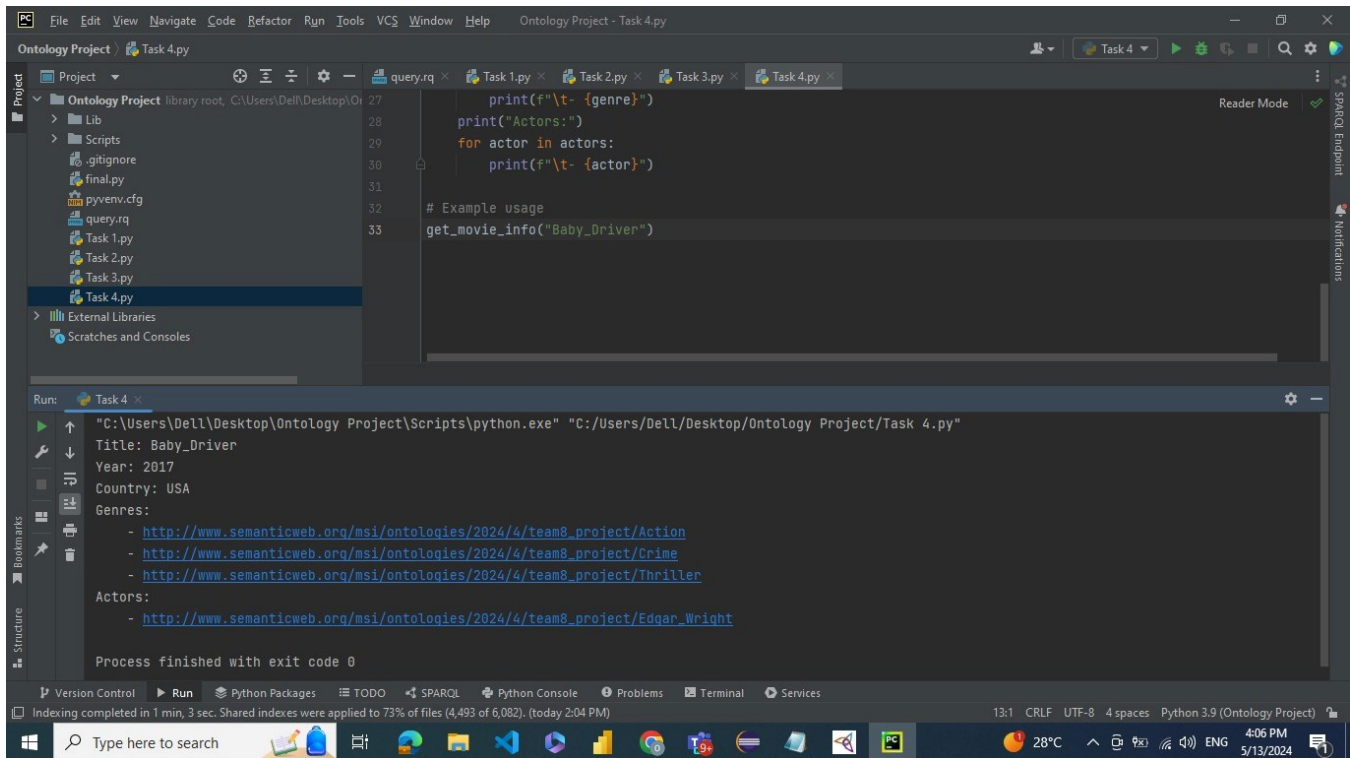
    # Find the movie
    movie = URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/" + movie_title)
    if (movie, RDF.type,
        URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Movie")) not in g:
        print("Movie not found!")
        return

    # Get movie details
    year = g.value(movie,
        URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/year"))
    country = g.value(movie,
        URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/country"))
    genres = g.objects(movie,
        URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/hasGenre"))
    actors = g.objects(movie,
        URIRef("http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/hasActor"))

    # Print movie information
    print(f"Title: {movie_title}")
    print(f"Year: {year}")
    print(f"Country: {country}")
    print("Genres:")
    for genre in genres:
        print(f"\t- {genre}")
    print("Actors:")
    for actor in actors:
        print(f"\t- {actor}")

# Example usage
get_movie_info("")
```

Output



The screenshot shows an IDE window titled "Ontology Project - Task 4.py". The editor displays a Python script with the following code:

```
27 print(f"\t- {genre}")
28 print("Actors:")
29 for actor in actors:
30     print(f"\t- {actor}")
31
32 # Example usage
33 get_movie_info("Baby_Driver")
```

The "Run" panel at the bottom shows the execution output for "Task 4":

```
"C:\Users\Dell\Desktop\Ontology Project\Scripts\python.exe" "C:\Users\Dell\Desktop\Ontology Project\Task 4.py"
Title: Baby_Driver
Year: 2017
Country: USA
Genres:
- http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Action
- http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Crime
- http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Thriller
Actors:
- http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/Edgar_Wright
Process finished with exit code 0
```

The status bar at the bottom indicates "Indexing completed in 1 min, 3 sec. Shared indexes were applied to 73% of files (4,493 of 6,082). (today 2:04 PM)". The system tray shows the time as 4:06 PM on 5/13/2024.

Figure 20: Screenshot of Output task 4

5. Create a java program (Jena5.java) that displays all persons that are actors and directors. Do this using a rule that defines a new class ActorDirector. The rule file must be saved in the data folder.

Code

```
package pack;
import tools.JenaEngine;

import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.Property;
import org.apache.jena.rdf.model.ResIterator;
import org.apache.jena.rdf.model.Resource;
public class Task5 {

    private Model model;
    private String file;
    private String namespace;

    Task5(String path){
        this.namespace = "";
        this.file = path;
        this.model = JenaEngine.readModel(path);
        if(model != null ){
            namespace = model.getNsPrefixURI("");
        }
    }

    public void readActorDirector(){
        //this.model = JenaEngine.readInferencedModelFromRuleFile(model,
        "Ontology/owlrules.txt");
        this.model = JenaEngine.readInferencedModelFromRuleFile(model,
        "Ontology/rule5.txt");

        String query = "PREFIX :
        <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>"
            + "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
            + "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>"
            + "SELECT ?person "
            + "WHERE {"
            + "?person rdf:type :ActorDirector. " + "}"
        System.out.println(JenaEngine.executeQuery(model, query));
    }

}
```

rule

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

@prefix : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/> .

[rule1: (?per rdf:type :Actor), (?per rdf:type :Director) -> (?per rdf:type :ActorDirector)]

Output

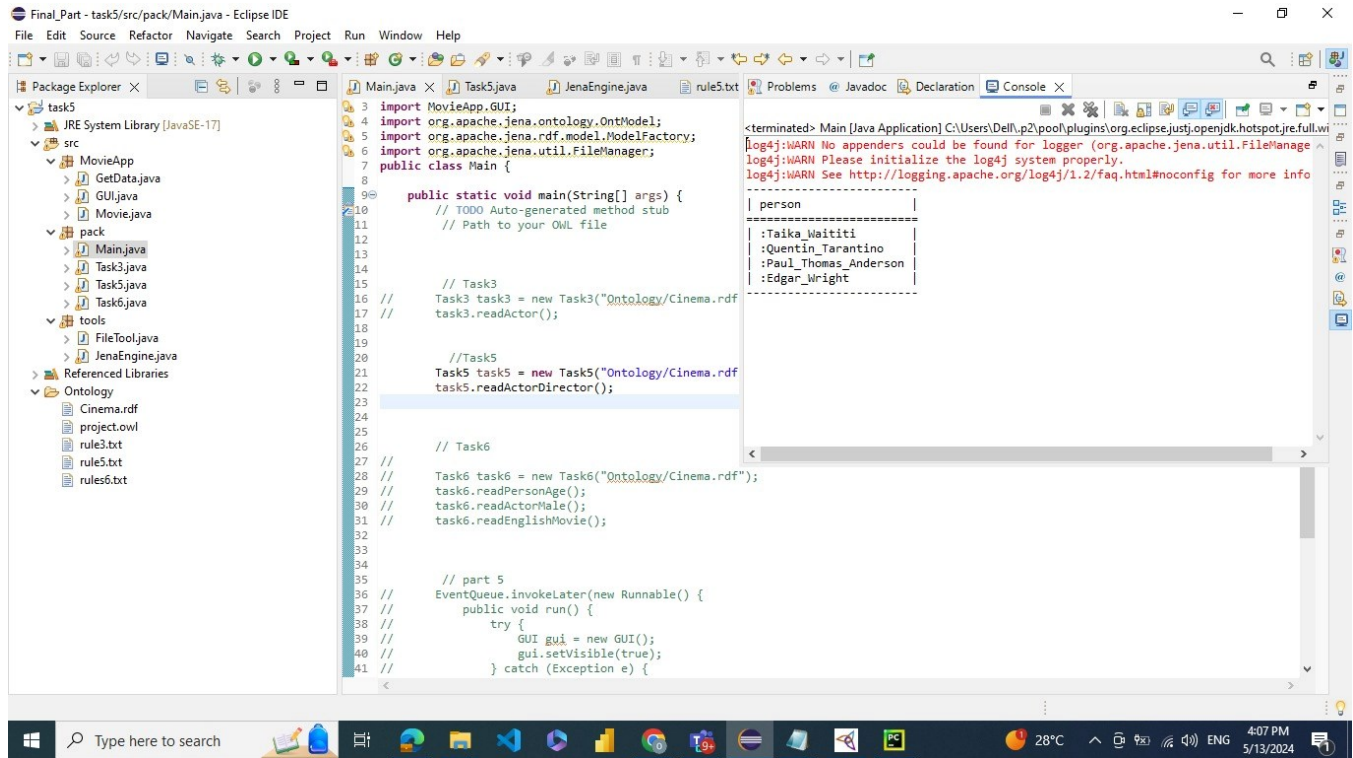


Figure 21: Screenshot of the output task 5

6. Specify 3 different rules and implement them in a java program (Jena6.java)

Code

```
package pack;
import tools.JenaEngine;

import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.Property;
import org.apache.jena.rdf.model.ResIterator;
import org.apache.jena.rdf.model.Resource;
public class Task6 {

    private Model model;
    private String file;

    Task6(String path) {
        this.file = path;
        this.model = JenaEngine.readModel(file);

        //this.model = JenaEngine.readInferencedModelFromRuleFile(model,
        //    "data/owlrules.txt");
        this.model = JenaEngine.readInferencedModelFromRuleFile(model,
            "Ontology/rules6.txt");
    }

    public void readPersonAge() {
        String query = "PREFIX :
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
+ "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
+ "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>"
+ "SELECT ?person ?age "
+ "WHERE {"
+ "?person rdf:type :PersonAge. ?person :age ?age." + "}";
        System.out.println(JenaEngine.executeQuery(model, query));
    }

    public void readActorMale() {
        String query = "PREFIX :
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
+ "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
+ "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>"
+ "SELECT ?person ?gender "
+ "WHERE {"
+ "?person rdf:type :ActorMale. ?person :hasGenderType ?gender."
+ "}";
        System.out.println(JenaEngine.executeQuery(model, query));
    }
}
```

```

public void readEnglishMovie() {
    String query = "PREFIX :
<http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/>
    + "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
    + "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>"
    + "SELECT ?movie ?year ?language "
    + "WHERE {"
    + "?movie rdf:type :EnglishMovie. ?movie :year ?year. ?movie :language
?language ."
    + "}";
    System.out.println(JenaEngine.executeQuery(model, query));
}
}

```

rules

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://www.semanticweb.org/msi/ontologies/2024/4/team8_project/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

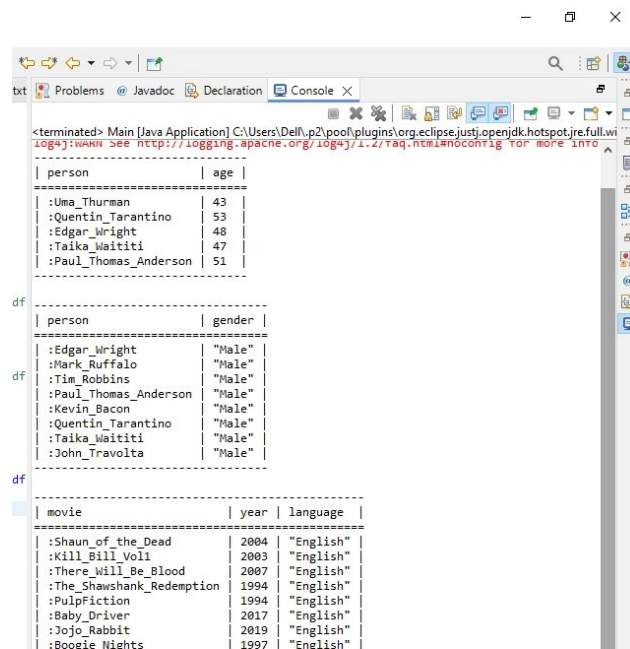
```

```

[rule1: (?per rdf:type :Actor) (?per :age ?age) lessThan(?age, 55)-> (?per rdf:type :PersonAge)]
[rule2: (?per rdf:type :Actor) (?per :hasGenderType "Male")-> (?per rdf:type :ActorMale)]
[rule3: (?per rdf:type :Movie) (?per :language "English") -> (?per rdf:type :EnglishMovie)]

```

Output



The screenshot shows the Eclipse IDE console with three tables of query results. The first table lists actors and their ages. The second table lists actors and their genders. The third table lists movies, their years, and their languages.

person	age
:Uma_Thurman	43
:Quentin_Tarantino	53
:Edgar_Wright	48
:Taika_Waititi	47
:Paul_Thomas_Anderson	51

person	gender
:Edgar_Wright	"Male"
:Mark_Ruffalo	"Male"
:Tim_Robbins	"Male"
:Paul_Thomas_Anderson	"Male"
:Kevin_Bacon	"Male"
:Quentin_Tarantino	"Male"
:Taika_Waititi	"Male"
:John_Travolta	"Male"

movie	year	language
:Shaun_of_the_Dead	2004	"English"
:Kill_Bill_Vol1	2003	"English"
:There_Will_Be_Blood	2007	"English"
:The_Shawshank_Redemption	1994	"English"
:Pulpfiction	1994	"English"
:Baby_Driver	2017	"English"
:Jojo_Rabbit	2019	"English"
:Boogie_Nights	1997	"English"

Figure 22: Screenshot of the output task 6

Part V: Java application

Overview

This Java application allows users to search for movies based on genre and individuals involved in the film industry (actors, directors, writers). Here's a breakdown of its components:

1. Movie.java:

This class defines the structure of a Movie object.

It stores information such as:

- title
- year
- language
- nation (country of origin)

It provides getters and setters to access and modify these attributes.

2. GetData.java:

This class interacts with an RDF knowledge base (Ontology/Cinema.rdf) using the Jena library.

- The getFilmData method retrieves information from the knowledge base based on two inputs:
- name: The name of a genre, actor, director, or writer.
- property: The relationship to search for (e.g., "hasGenre", "isActorOf").
- It queries the knowledge base to find movies related to the given name and property.
- It returns an ArrayList of Movie objects populated with data from the knowledge base.

3. GUI.java:

This class creates the user interface using Swing components.

It includes:

- Text areas for entering genre and person's name.
- Buttons for searching and clearing data.
- Four tables to display the search results:
 - a. Film list based on genre.
 - b. Films as an actor.
 - c. Films as a director.
 - d. Films as a writer.
- The initTable method sets up the column names for the tables.
- The setData method populates the tables with data retrieved from the GetData class.

- The clearData method clears the contents of the tables.

How the Application Works:

1. User Input: The user enters a genre in the genre text area and/or a person's name in the actor text area.
2. Search: When the user clicks the "search" button:
 - The GUI class calls the getFilmData method from the GetData class.
 - The getFilmData method queries the knowledge base for relevant movies.
 - The results are returned to the GUI class as an ArrayList of Movie objects.
3. Display Results: The GUI class uses the setData method to populate the appropriate tables with the retrieved movie data.
4. Clear: The "clear" button clears the user input and the contents of the tables.

In summary: This application provides a graphical interface for querying a semantic knowledge base about movies. Users can search based on genre and the roles of individuals in the film industry. The results are presented in an organized tabular format.

Code

Movie.java Code:

```
package MovieApp;

public class Movie {

    private String title;
    private String year;
    private String language;
    private String nation;

    public Movie(String title) {
        super();
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getYear() {
        return year;
    }

    public void setYear(String year) {
        this.year = year;
    }

    public String getLanguage() {
        return language;
    }

    public void setLanguage(String language) {
        this.language = language;
    }

    public String getNation() {
        return nation;
    }

    public void setNation(String nation) {
        this.nation = nation;
    }

}
```

GetData.java Code:

```
package MovieApp;

import java.util.ArrayList;

import tools.JenaEngine;

import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.Property;
import org.apache.jena.rdf.model.Resource;
import org.apache.jena.rdf.model.Statement;
import org.apache.jena.rdf.model.StmtIterator;

public class GetData {
    private Model model;
    private String namespace;
    private String file;

    public GetData() {
        namespace = "";
        file = "Ontology/Cinema.rdf";
        setModel();
        if (model != null) {
            namespace = model.getNsPrefixURI("");
        }
    }

    public void setModel() {
        this.model = JenaEngine.readModel("Ontology/Cinema.rdf");
    }

    public ArrayList getFilmData(String name, String property){
        Resource rs = model.getResource(namespace + name);
        Property p = model.getProperty(namespace + property);
        Property ptitle = model.getProperty(namespace + "title");
        Property pyear = model.getProperty(namespace + "year");
        Property plang = model.getProperty(namespace + "language");
        Property pnation = model.getProperty(namespace + "country");
        ArrayList result = new ArrayList();

        if ((rs != null) && (p != null)) {
            StmtIterator it = rs.listProperties(p);
            while (it.hasNext()) {
                Statement s = it.next();
                Resource re = s.getResource();
                String title = re.getProperty(ptitle).getString();
                Movie temp = new Movie(title);
                temp.setLanguage(re.getProperty(plang).getString());
            }
        }
    }
}
```

```

        temp.setYear(re.getProperty(pyear).getString());
        temp.setNation(re.getProperty(pnation).getString());
        result.add(temp);
    }
    return result;
} else {
    return result;
}
}
}

```

GUI Code:

```

package MovieApp;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.util.ArrayList;
import java.util.Vector;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.table.DefaultTableModel;

public class GUI extends JFrame {

    private JTextArea genre;
    private JTextArea actortext;

    private GetData data;
    private DefaultTableModel filmTableModel;
    private DefaultTableModel actorTableModel;
    private DefaultTableModel directorTableModel;
    private DefaultTableModel writerTableModel;

    private JTable filmTable;
    private JTable actorTable;
    private JTable directorTable;
    private JTable writerTable;

    private Vector filmVectorData;
    private Vector actorVectorData;
    private Vector directorVectorData;

```

```

private Vector writerVectorData;
private Vector filmVectorColName;

public GUI() {
    super();
    getContentPane().setLayout(null);
    setBounds(100, 100, 800, 500);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    data = new GetData();

    //clear data in the input and table
    final JButton buttonClear = new JButton();
    buttonClear.addActionListener(new ActionListener() {
        public void actionPerformed(final ActionEvent e) {
            genre.setText(null);
            actortext.setText(null);
            clearData();
        }
    });
    buttonClear.setText("clear");
    buttonClear.setBounds(500, 390, 106, 28);
    getContentPane().add(buttonClear);

    //initial vectors
    filmVectorColName = new Vector();
    filmVectorData = new Vector();
    actorVectorData = new Vector();
    directorVectorData = new Vector();
    writerVectorData = new Vector();

    initTable();

    filmTableModel = new DefaultTableModel();
    filmTableModel.setDataVector(filmVectorData,filmVectorColName);
    filmTable = new JTable(filmTableModel);

    actorTableModel = new DefaultTableModel();
    actorTableModel.setDataVector(actorVectorData,filmVectorColName);
    actorTable = new JTable(actorTableModel);

    directorTableModel = new DefaultTableModel();
    directorTableModel.setDataVector(directorVectorData,filmVectorColName);
    directorTable = new JTable(directorTableModel);

    writerTableModel = new DefaultTableModel();
    writerTableModel.setDataVector(writerVectorData,filmVectorColName);
    writerTable = new JTable(writerTableModel);

    //button search

```

```

final JButton buttonSearch = new JButton();
buttonSearch.addActionListener(new ActionListener() {
    public void actionPerformed(final ActionEvent e) {
        clearData();

        String str = genre.getText();
        ArrayList filmlist = data.getFilmData(str, "hasGenre");
        setData(filmlist, filmVectorData);
        filmTable.invalidate();
        filmTable.updateUI();

        String pname = actortext.getText();
        ArrayList actfilmlist = data.getFilmData(pname, "isActorOf");
        setData(actfilmlist, actorVectorData);
        actorTable.invalidate();
        actorTable.updateUI();

        ArrayList direfilmlist = data.getFilmData(pname, "isDirectorOf");
        setData(direfilmlist, directorVectorData);
        directorTable.invalidate();
        directorTable.updateUI();

        ArrayList writerfilmlist = data.getFilmData(pname, "isWriterOf");
        setData(writerfilmlist, writerVectorData);
        writerTable.invalidate();
        writerTable.updateUI();
    }
});

//add table and text into frame
buttonSearch.setText("search");
buttonSearch.setBounds(100, 390, 106, 28);
getContentPane().add(buttonSearch);

final JScrollPane scrollPane = new JScrollPane();
scrollPane.setBounds(50, 46, 100, 30);
getContentPane().add(scrollPane);

genre = new JTextArea();
scrollPane.setViewportViewView(genre);

// final JScrollPane scrollPane_1 = new JScrollPane(filmTable);
// filmTable.setFillViewportHeight(true);
// scrollPane_1.setBounds(400, 30, 350, 70);
// getContentPane().add(scrollPane_1);

final JScrollPane scrollPane_input = new JScrollPane();
scrollPane_input.setBounds(50, 150, 150, 30);
getContentPane().add(scrollPane_input);

```

```

actortext = new JTextArea();
scrollPane_input.setViewportView(actortext);

final JScrollPane scrollPane_output1 = new JScrollPane(actorTable);
actorTable.setFillViewportHeight(true);
scrollPane_output1.setBounds(400, 115, 350, 70);
getContentPane().add(scrollPane_output1);

//

final JScrollPane scrollPane_output2 = new JScrollPane(directorTable);
directorTable.setFillViewportHeight(true);
scrollPane_output2.setBounds(400, 205, 350, 70);
getContentPane().add(scrollPane_output2);

final JScrollPane scrollPane_output3 = new JScrollPane(writerTable);
writerTable.setFillViewportHeight(true);
scrollPane_output3.setBounds(400, 295, 350, 70);
getContentPane().add(scrollPane_output3);

final JLabel label = new JLabel();
label.setText("Input genre");
label.setBounds(50, 22, 80, 18);
getContentPane().add(label);

//
//
//
//
final JLabel label_1 = new JLabel();
label_1.setText("Filmlist");
label_1.setBounds(400, 13, 66, 18);
getContentPane().add(label_1);

final JLabel labeinput = new JLabel();
labeinput.setText("Input person name");
labeinput.setBounds(50, 130, 150, 18);
getContentPane().add(labeinput);

final JLabel labeoutput1 = new JLabel();
labeoutput1.setText("As actor");
labeoutput1.setBounds(400, 100, 150, 10);
getContentPane().add(labeoutput1);

final JLabel labeoutput2 = new JLabel();
labeoutput2.setText("As director");
labeoutput2.setBounds(400, 190, 150, 18);
getContentPane().add(labeoutput2);

final JLabel labeoutput3 = new JLabel();
labeoutput3.setText("As writer");
labeoutput3.setBounds(400, 278, 150, 18);

```



```

        getContentPane().add(labeoutput3);
    }

    public void initTable(){
        this.filmVectorColName.addElement("Title");
        this.filmVectorColName.addElement("Year");
        this.filmVectorColName.addElement("Language");
        this.filmVectorColName.addElement("Nationality");
    }

    //add data into table
    public void setData(ArrayList list, Vector data){
        for(int i = 0; i < list.size() ; i++){
            Vector vector = new Vector();
            Movie movie = (Movie) list.get(i);
            vector.addElement(movie.getTitle());
            vector.addElement(movie.getYear());
            vector.addElement(movie.getLanguage());
            vector.addElement(movie.getNation());
            data.addElement(vector);
        }
    }

    public void clearData(){
        this.filmVectorData.clear();
        this.directorVectorData.clear();
        this.actorVectorData.clear();
        this.writerVectorData.clear();
    }

}

```

App Screenshots

The screenshot shows a web application window with a light gray background. On the left, there are two input fields: 'Input genre' with the value 'Action' and 'Input person name' with the value 'Edgar_Wright'. At the bottom left is a 'search' button and at the bottom right is a 'clear' button. On the right side, there are three tables under the headings 'As actor', 'As director', and 'As writer'. Each table has columns for Title, Year, Language, and Nationality. The 'As actor' table shows one entry: 'Shaun of the ...' from 2004, English, British. The 'As director' table shows one entry: 'Shaun of the ...' from 2004, English, British. The 'As writer' table shows one entry: 'Shaun of the ...' from 2004, English, British.

As actor			
Title	Year	Language	Nationality
Shaun of the ...	2004	English	British

As director			
Title	Year	Language	Nationality
Shaun of the ...	2004	English	British

As writer			
Title	Year	Language	Nationality
Shaun of the ...	2004	English	British

Figure 23: Edgar Write roles in all actions movies

The screenshot shows the same web application window. The 'Input genre' field now contains 'Crime' and the 'Input person name' field contains 'Paul_Thomas_Anderson'. The 'search' and 'clear' buttons remain at the bottom. The tables on the right are updated: 'As actor' shows two entries ('Boogie Nights' 1997 and 'There Will Be...' 2007, both English, USA); 'As director' shows two entries ('There Will Be...' 2007 and 'Boogie Nights' 1997, both English, USA); 'As writer' shows two entries ('Boogie Nights' 1997 and 'There Will Be...' 2007, both English, USA).

As actor			
Title	Year	Language	Nationality
Boogie Nights	1997	English	USA
There Will Be...	2007	English	USA

As director			
Title	Year	Language	Nationality
There Will Be...	2007	English	USA
Boogie Nights	1997	English	USA

As writer			
Title	Year	Language	Nationality
Boogie Nights	1997	English	USA
There Will Be...	2007	English	USA

Figure 24: Paul Thomas roles in all crime movies

The screenshot shows the same web application window. The 'Input genre' field now contains 'Comedy' and the 'Input person name' field contains 'Taika_Waititi'. The 'search' and 'clear' buttons remain at the bottom. The tables on the right are updated: 'As actor' shows one entry ('Jojo Rabbit' 2019, English, Nazi Germany); 'As director' shows two entries ('Thor: Ragnar...' 2017, English, USA and 'Jojo Rabbit' 2019, English, Nazi Germany); the 'As writer' table is currently empty.

As actor			
Title	Year	Language	Nationality
Jojo Rabbit	2019	English	Nazi Germany

As director			
Title	Year	Language	Nationality
Thor: Ragnar...	2017	English	USA
Jojo Rabbit	2019	English	Nazi Germany

As writer			
Title	Year	Language	Nationality

Figure 25: Taiki Waititi roles in Comedy movies

Github Link

<https://github.com/amrressam148/Cinema-Ontology>