Introduction to Semantic Systems: Exercise 3

Yahya Jabary, 11912007

In this exercise, we go through a practical semantic data integration scenario. We'll begin by constructing a knowledge graph from a given dataset. Next, we'll enrich this graph using SPARQL queries. Finally, we'll validate our knowledge graph with SHACL shapes to ensure data integrity and consistency.

1. Creating a Knowledge Graph

1.0. Film Ontology Extension

To integrate our instances into the preexisting ontology `film.ttl` [1] [2], provided by the "Semantic Systems Research of ISE research group" at TU Wien, we need to extend the ontology with new classes and properties. This extension will ensure that our data aligns with the existing structure.

The following ASCII diagram illustrates the necessary extensions using Markdown syntax. Note that two of the requested properties, `hasEditor` and `hasCast`, already exist in the ontology, so no changes are needed for them.

```
Film

- id (String)
- isAdultFilm (boolean)
- homepage (url)
- hasIMDBResource (IMDBResource)
- description (String)
- keyword (String)
- tagline (String)
- original_language (String)
- popularity (float)
- originalTitle (String)
- hasEditor (Editor)
- hasCast (Cast)
- hasSpokenLanguage (String)
- hasProductionCountry (String)

IMDBResource

- id (String)
- url (String)
- vote_average (float)
- vote_count (integer)

Actor/Director/ScriptWriter/Editor

- fullName (String) -> assignment error, already exists
- gender (String) -> assignment error, already exists

Genre

- id (Integer)

FilmStudio

- id (Integer)

Cast

- hasCastActor (Actor)
- hasCastCharacter (String)
```

1.1. OpenRefine Mappings

Next, we use OpenRefine (formerly known as Google Refine) to preprocess the data and convert the provided CSV files into RDF format. We'll join the tables to create a single RDF file that adheres to the extended ontology.

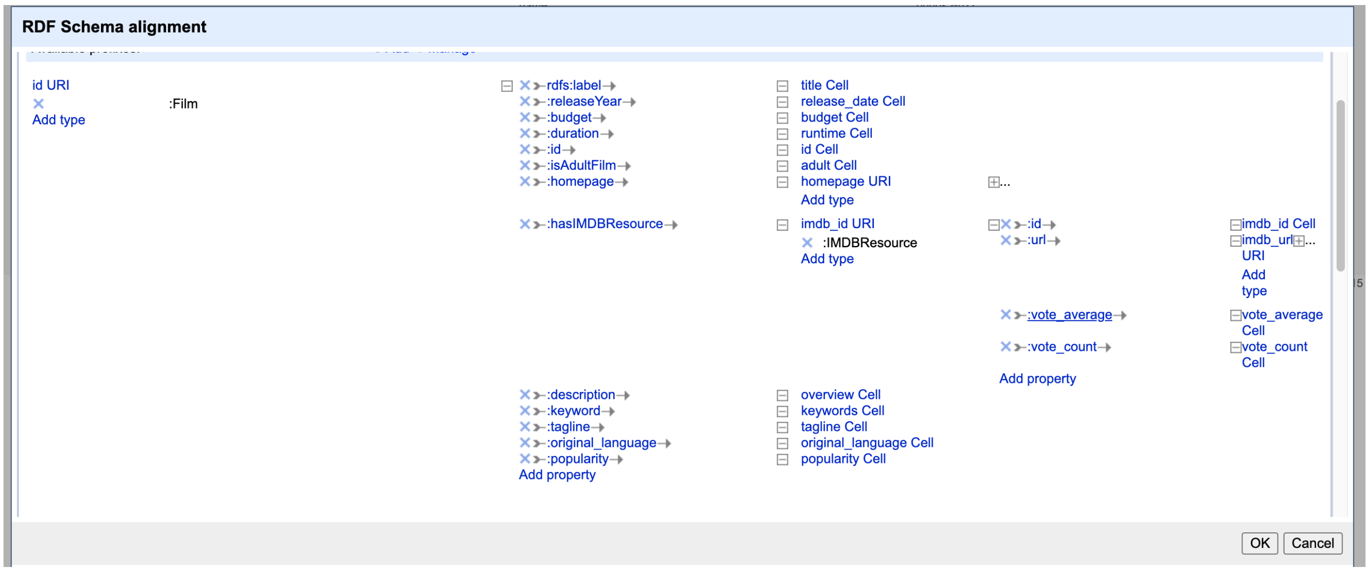
To ensure compatibility with the RDF extension [3], we downloaded OpenRefine v3.8.2 [4].

We created three separate projects. For each project, we performed basic preprocessing tasks such as removing whitespaces and parsing columns.

Next, we joined all tables on the `id` column, using a Google Refine Expression Language (GREL) instruction (e.g., `cell.cross("1000_keywords", "id").cells["keywords"].value[0]`) to merge the data. The following tables were joined to the `1000_movies_metadata.csv` table: `1000_keywords.csv` and `1000_links.csv`.

[1] Visualization: https://semantics.id/ns/example/film/webvowl/index.html
[2] Documentation: https://semantics.id/ns/example/film/
[3] OpenRefine RDF-Extension: https://github.com/fadmaa/grefine-rdf-extension
[4] OpenRefine v3.8.2: https://github.com/OpenRefine/OpenRefine/releases/tag/3.8.2 (Thanks to Valentin Bauer)

We then created an RDF schema for the data based on the RDF extension documentation and some older YouTube tutorials [5].


RDF Schema alignment

id URI — :Film — Add type

- :rdfs:label → title Cell
- :releaseYear → release_date Cell
- :budget → budget Cell
- :duration → runtime Cell
- :id → id Cell
- :isAdultFilm → adult Cell
- :homepage → homepage URI
- Add type
- :hasIMDBResource → imdb_id URI — :IMDBResource — Add type
  - :id → imdb_id Cell
  - :url → imdb_url URI — Add type
  - :vote_average → vote_average Cell
  - :vote_count → vote_count Cell
  - Add property
- :description → overview Cell
- :keyword → keywords Cell
- :tagline → tagline Cell
- :original_language → original_language Cell
- :popularity → popularity Cell
- Add property

OK Cancel

Figure 1: Grefine-RDF-extension Screenshot

When creating the RDF schema, some prefix and namespace preprocessing was necessary to ensure alignment with the existing (but very inconsistent) ontology. Specifically, we used `"film_" + (value + 10)` as the film ID to avoid conflicts with existing IDs, `value.substring(0, 4)` as the release year, and `"imdbresource_" + value` as the IMDB resource ID, to be then linked to the `IMDBResource` class through the `hasIMDBResource` property. This linking is the most sophisticated part of the RDF schema. The first entry in the CSV file demonstrates how instances from different classes are interconnected:

```
<http://semantics.id/ns/example#film_872> a :Film , owl:NamedIndividual ;
    rdfs:label "Toy Story" ;
    :releaseYear "1995"^^xsd:int ;
    :budget "30000000"^^xsd:double ;
    :duration "81"^^xsd:int ;
    :id "862" ;
    :isAdultFilm "False"^^xsd:boolean ;
    :homepage <http://toystory.disney.com/toy-story> .
    :hasIMDBResource <http://semantics.id/ns/example#imdbresource_tt0114709> ;
    :description "Led by Woody, Andy's toys live happily in his room until Andy's birthday brings Buzz Lightyear onto the scene. Af
    :keyword "[{'id': 931, 'name': 'jealousy'}, {'id': 4290, 'name': 'toy'}, {'id': 5202, 'name': 'boy'}, {'id': 6054, 'name': 'fri
    :original_language "en" ;
    :popularity "21.946943"^^xsd:double .

<http://semantics.id/ns/example#imdbresource_tt0114709> a :IMDBResource , owl:NamedIndividual ;
    :id "tt0114709" ;
    :url <https://www.imdb.com/title/tt0114709> ;
    :vote_average "7.7"^^xsd:double ;
    :vote_count "5415"^^xsd:int .
```

One of the core issues in the namespace was that some instances used the `example#` prefix, while others used the `film#` prefix. Also, some used titles and names as identifiers, while others used IDs. We had to manually adjust the RDF schema to ensure consistency and opted for the `film#` namespace, using IDs where no other information was available based on the existing ontology.

Finally, we exported the RDF data from OpenRefine and saved it as `final.ttl`.

We also attempted to reconcile the data with the preexisting `film.ttl` file, but because the film identifiers did not overlap (e.g., starting from 1 to 10 in the existing ontology with no similar names, titles, etc., in the new data), we were unable to. Additionally, the Tomcat server crashed when trying to reconcile the data, returning a `500` error and the failure message `Failed to initialize an instance of org.apache.velocity.runtime.log.Log4JLogChute with the current runtime configuration` on the `MOD-INF/controller.js` file.

Nonetheless, we were able to successfully export the RDF data from OpenRefine and save it as `final.ttl`.

1.2. RML Mappings

For the RDF mapping language (RML) part, we used the RMLMapper CLI tool to convert the JSON files to RDF. We downloaded the latest version of the RMLMapper (v7.2.0) from the official GitHub repository [6] to execute our code

[5]OpenRefine RDF Extension Tutorials: https://www.youtube.com/watch?v=80DN7tkNMdI, https://www.youtube.com/watch?v=kn-YeDXR4R8, https://www.youtube.com/watch?v=jf2jMDmaVis

[6]RMLMapper GitHub Repository: https://github.com/RMLio/rmlmapper-java

with Java 17 OpenJDK:

```
java -jar rmlmapper.jar -m final.rml.ttl -o final.ttl -s turtle
```

We initially experimented with YARRRML for its conciseness and syntactic sugar. However, we quickly discovered that the community support and documentation were lacking. We found the traditional Turtle-based syntax to be more reliable and readable, especially when defining namespaces and prefixes. The ability to easily copy and paste from the ontology made Turtle a more convenient choice for our project.

On a high level, we used subject maps like `<#FilmMapping>`, `<#GenreMapping>`, `<#FilmStudioMapping>`, `<#CastMapping>`, `<#ActorMapping>`, `<#DirectorMapping>`, and `<#ScriptWriterMapping>` for each class. We defined predicate-object maps for each property, such as `:originalTitle` and `:hasSpokenLanguage`. For URIs and IRIs, we used templates like `http://semantics.id/ns/example#film_{id}0000` for films and `http://semantics.id/ns/example#{name}` for actors, adding suffixes to avoid conflicts with the existing ontology or CSV data. We also manually validated that there are no identical IDs in the ontology and the CSV data we wanted to merge.

Actors are simply people listed in the `[*.cast]` array of the `1000_credits.json` file. Directors, scriptwriters, and editors are identified from the `[*.crew]` array with their respective job titles. For example, we used the JSONPath filter `[*].crew[?(@.job=='Director')]` to extract directors.

The key entities that were linked to other entities in this project included `Film` and its relationships with `hasFilmStudio`, `hasGenre`, `hasDirector`, `hasScriptWriter`, `hasEditor`, `hasActor`, and `hasCast`. Additionally, `Cast` was linked with `hasCastActor` and `hasCastCharacter`.

To establish these mappings, we utilized the `rr:parentTriplesMap` and `rr:joinCondition` properties to connect the entities. Specifically, we employed the `rr:objectMap` property, which was paired with either a `rr:template` and `rr:termType` property or a `rr:parentTriplesMap` property combined with a `rr:joinCondition` property to ensure the entities were correctly linked.

```
<#FilmMapping>
    # ... stuff omitted for brevity

    rr:predicateObjectMap [

    # ... stuff omitted for brevity

        rr:predicate :hasFilmStudio;
        rr:objectMap [
            rr:template "http://semantics.id/ns/example#{production_companies[*].name}";
            rr:termType rr:IRI
        ]
    ], [
        rr:predicate :hasGenre;
        rr:objectMap [
            rr:template "http://semantics.id/ns/example/film#genre_{genres[*].name}";
            rr:termType rr:IRI
        ]
    ], [
        rr:predicate :hasDirector;
        rr:objectMap [
            rr:parentTriplesMap <#DirectorMapping>;
                rr:joinCondition [
                rr:child "id";
                rr:parent "id"
            ]
        ]
    ], [
        rr:predicate :hasScriptWriter;
        rr:objectMap [
            rr:parentTriplesMap <#ScriptWriterMapping>;
            rr:joinCondition [
                rr:child "id";
                rr:parent "id"
            ]
        ]
    ], [
        rr:predicate :hasEditor;
        rr:objectMap [
            rr:parentTriplesMap <#EditorMapping>;
            rr:joinCondition [
                rr:child "id";
                rr:parent "id"
            ]
        ]
    ], [
        rr:predicate :hasActor;
        rr:objectMap [
            rr:parentTriplesMap <#ActorMapping>;
            rr:joinCondition [
                rr:child "id";
                rr:parent "id"
            ]
        ]
    ], [
        rr:predicate :hasCast;
        rr:objectMap [
            rr:template "http://semantics.id/ns/example#cast_{cast[*].credit_id}";
            rr:termType rr:IRI
        ]
    ].

<#CastMapping>
    # unique actor-role pairs for each film

    # ... stuff omitted for brevity

    rr:subjectMap [
        # no named individuals for reference in `film.ttl` we can use an arbitrary template
        rr:template "http://semantics.id/ns/example#cast_{credit_id}";
        rr:class :Cast ;
        rr:class owl:NamedIndividual
    ];
    rr:predicateObjectMap [
        rr:predicate :hasCastCharacter;
        rr:objectMap [ rml:reference "character" ]
    ], [
        rr:predicate :hasCastActor;
        rr:objectMap [
            rr:template "http://semantics.id/ns/example#{name}";
            rr:termType rr:IRI
        ]
    ].
```

Another issue we encountered was the inconsistency between the `example#` and `film#` namespaces. This discrepancy required manual adjustments to ensure uniformity across the dataset.

Additionally, the `gender` property was represented as integers (0-2) instead of using a more descriptive enumeration. This required further data cleaning to align with the expected format.

We also conducted manual tests to verify that individuals could hold multiple roles, such as being both a screenwriter and a director simultaneously.

An example demonstrating the mapping of entities is shown below:

```
<http://semantics.id/ns/example#film_9800000> a :Film, owl:NamedIndividual;
    :hasActor <http://semantics.id/ns/example#Vinnie%20Jones>;
    :hasFilmStudio <http://semantics.id/ns/example#Twentieth%20Century%20Fox%20Film%20Corporation>;
    :hasGenre :genre_Action, :genre_Drama, :genre_Western;
    :hasProductionCountry "United States of America";
    :hasSpokenLanguage "English";
    :originalTitle "The Ox-Bow Incident" .

<http://semantics.id/ns/example#Vinnie%20Jones> a :Actor, owl:NamedIndividual;
    :fullName "Vinnie Jones";
    :gender "2" .

<http://semantics.id/ns/example#Twentieth%20Century%20Fox%20Film%20Corporation> a
    :FilmStudio, owl:NamedIndividual;
    :id 306;
    rdfs:label "Twentieth Century Fox Film Corporation" .

<http://semantics.id/ns/example#cast_52fe4217c3a36847f800360f> a :Cast, owl:NamedIndividual;
    :hasCastActor <http://semantics.id/ns/example#Vinnie%20Jones>;
    :hasCastCharacter "Big Chris" .
```

1.3. Merging

Finally, we combined all the files, including the extended ontology, the OpenRefine RDF export, and the RML export, into a single `merged.ttl` file for further processing. We used the `ttl-merge` tool to merge the files and `turtlefmt` to format the resulting TTL file.

```
# fix encoding issues
for f in ./*.ttl; do
    iconv -f utf-8 -t utf-8 -c "$f" > "${f}.clean"
    mv "${f}.clean" "$f"
done

# merge
# ignore weird double/float type missmatch warnings, due to openrefine export
npm install -g ttl-merge
ttl-merge -i ./*.ttl > merged.ttl

# fmt
pip install turtlefmt
turtlefmt merged.ttl
```

2. SPARQL Enrichment

We used GraphDB as our triple store to execute the SPARQL queries.

2.1. Construct Queries

We crafted four SPARQL queries to enrich our knowledge graph with additional insights. These queries focused on identifying frequently collaborating actors, directors with preferred actors, directors specializing in specific genres, and films from the same era.

The first query's goal was to find actors who often worked together in multiple films. It selected actors who appeared together in at least two films and introduced a new property, `frequentCostarWith`, to capture this relationship.

The second query aimed to pinpoint directors who frequently collaborated with the same actor across multiple films. It identified directors who worked with the same actor in at least two films and created a new property, `hasPreferredActor`, to denote this connection.

The third query focused on directors who specialized in specific genres. It selected directors who directed films in the same genre at least twice and added a new property, `specializesInGenre`, to represent this specialization.

The fourth query sought to identify films released around the same time, specifically within a two-year span. It selected films released within two years of each other and introduced a new property, `fromSameEra`, to indicate this temporal relationship.

The use-case for these queries was to provide recommendations for potential collaborations between actors and directors, identify genre preferences and suggest films from the same era for thematic programming or analysis.

```
PREFIX film: <http://semantics.id/ns/example/film#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

# insert frequently collaborating actors and director preferences
INSERT {
    ?actor1 film:frequentCostarWith ?actor2 .

    ?director film:hasPreferredActor ?actor .

    ?director film:specializesInGenre ?genre .

    ?film1 film:fromSameEra ?film2 .
}
WHERE {
    {
        # find actors who worked together in at least 2 films
        SELECT ?actor1 ?actor2 (COUNT(DISTINCT ?film) as ?collaborations)
        WHERE {
            ?film film:hasActor ?actor1 .
            ?film film:hasActor ?actor2 .
            FILTER(?actor1 != ?actor2)
        }
        GROUP BY ?actor1 ?actor2
        HAVING (?collaborations >= 2)
    }
    UNION
    {
        # find directors who worked with same actor in multiple films
        SELECT ?director ?actor (COUNT(DISTINCT ?film) as ?collaborations)
        WHERE {
            ?film film:hasDirector ?director .
            ?film film:hasActor ?actor .
        }
        GROUP BY ?director ?actor
        HAVING (?collaborations >= 2)
    }
    UNION
    {
        # find directors who frequently work in specific genres
        SELECT ?director ?genre (COUNT(DISTINCT ?film) as ?genreFilms)
        WHERE {
            ?film film:hasDirector ?director .
            ?film film:hasGenre ?genre .
        }
        GROUP BY ?director ?genre
        HAVING (?genreFilms >= 2)
    }
    UNION
    {
        # find films from the same era, with a maximum difference of 2 years
        SELECT ?film1 ?film2
        WHERE {
            ?film1 film:releaseYear ?year1 .
            ?film2 film:releaseYear ?year2 .
            FILTER(?film1 != ?film2 && abs(?year1 - ?year2) <= 2)
            FILTER NOT EXISTS { ?film1 film:fromSameEra ?film2 }
        }
    }
}
```

2.2. Mandatory Update Query

Finally, to ensure the originality of our submission, we replaced the name of the director `Christopher Nolan` with `YahyaJabary_11912007`, who happens to be our favorite director.

```
PREFIX : <http://semantics.id/ns/example/film#>

DELETE {
    ?d :fullName "Christopher Nolan"
}
INSERT {
    ?d :fullName "YahyaJabary_11912007"
}
WHERE {
    ?d a :Director ;
       :fullName "Christopher Nolan"
}
```

2.3. Export data

The data stored in the GraphDB repository `repo` was exported as a Turtle file using the following bash command:

```
curl -X GET -H "Accept:application/x-turtle" "http://localhost:7200/repositories/repo/statements?infer=false" > export.ttl
```

3. SHACL Shapes

In a last step the data was validated against SHACL shapes. We used two different tools for this purpose: `pyshacl` and `shacl` from Apache Jena.

```
pip install pyshacl
pyshacl -s shapes.ttl -f human merged.ttl

brew install jena
shacl validate --shapes shapes.ttl --data merged.ttl
```

We encountered numerous data type warnings because OpenRefine did not provide us with enough granularity to define the types accurately. Additionally, it turns out that the input data is not entirely consistent with the ontology. Many instances have missing properties or incorrect values.

Therefore, we decided not to use the SHACL shapes for data type, cardinality or value range validation. Instead, we focused on the structural integrity of the data and that the relationships between entities are correctly defined.

```
14:00:39 WARN  riot            :: [line: 54163, col: 55] Lexical form 'False' not valid for datatype XSD boolean
14:00:39 WARN  riot            :: [line: 54268, col: 54] Lexical form 'Beware Of Frost Bites' not valid for datatype XSD double
...
14:00:39 WARN  riot            :: [line: 54012, col: 50] Lexical form '/ff9qCepilowshEtG2GYWwzt2bs4.jpg' not valid for datatype XSD
14:00:39 WARN  riot            :: [line: 54014, col: 55] Lexical form '- Written by Ørnås' not valid for datatype XSD boolean
14:00:39 WARN  riot            :: [line: 54145, col: 50] Lexical form '/zV8bHuSL6WXoD6FWogP9j4x80bL.jpg' not valid for datatype XSD
14:00:39 WARN  riot            :: [line: 54147, col: 55] Lexical form 'Rune Balot goes to a casino connected to the October corpora
14:00:39 WARN  riot            :: [line: 54257, col: 50] Lexical form '/zaSf5OG7V8X8gqFvly88zDdRm46.jpg' not valid for datatype XSD
14:00:39 WARN  riot            :: [line: 54259, col: 55] Lexical form 'Avalanche Sharks tells the story of a bikini contest that tu
...
```

Therefore, most of the SHACL shapes focused on ensuring the correct linking between entities, as we lacked sufficient information to validate other properties. However, we did create five additional shapes to utilize advanced SHACL features like `sh:sparql`.

These shapes were designed to ensure some degree of path traversal, thus leveraging the actual benefits of using graph data and SHACL shapes over traditional 3NF databases and data validation strategies by taking advantage of fast graph-based retrieval.

The first constraint serves to ensure that genre labels are unique per language, by checking the `rdfs:label` property of the `film:hasGenre` relationship via the `sh:uniqueLang` constraint within the `:Film` class.

```
film:UniqueGenreLabelShape
    a sh:NodeShape ;
    sh:targetClass film:Film ;
    sh:property [
        sh:path (film:hasGenre rdfs:label) ;
        sh:uniqueLang true ;
        sh:message "genre labels must be unique per language" ;
    ] .
```

The second constraint ensures that all crew members of a film comply with the validation rules defined in the `film:PersonShape` shape. This shape is used to validate the properties of `film:hasDirector`, `film:hasScriptWriter`, `film:hasActor`, and `film:hasEditor` relationships within the `:Film` class.

```
film:FilmCrewShape
    a sh:NodeShape ;
    sh:targetClass film:Film ;
    sh:property [
        sh:path [
            sh:alternativePath (
                film:hasDirector
                film:hasScriptWriter
                film:hasActor
                film:hasEditor
            )
        ] ;
        sh:node film:PersonShape ;
        sh:message "all crew members must comply with person validation rules" ;
    ] .
```

Next, we defined a shape to ensure that cast members are unique per film. This shape uses the `sh:uniqueLang` constraint to validate the `film:hasCastActor` and `film:hasCastCharacter` properties within the `:Cast` class (per lanugage and per film). Additionally, the `sh:maxCount` constraint is used to ensure that each cast member is unique and that there is only one character per cast member.

```
film:UniqueCastShape
    a sh:NodeShape ;
    sh:targetClass film:Cast ;
    sh:property [
        sh:path film:hasCastActor ;
        sh:uniqueLang true ;
        sh:maxCount 1 ;
        sh:property [
            sh:path film:hasCastCharacter ;
            sh:uniqueLang true ;
            sh:maxCount 1
        ] ;
        sh:message "cast members must be unique" ;
    ] .
```

The fourth constraint ensures that the spoken language of a film is one of the original languages of the film. This shape uses the `sh:in` constraint to validate the `film:original_language` property of the `film:hasSpokenLanguage` relationship within the `:Film` class.

```
film:LanguageConsistencyShape
    a sh:NodeShape ;
    sh:targetClass film:Film ;
    sh:property [
        sh:path film:hasSpokenLanguage ;
        sh:property [
            sh:path film:original_language ;
            sh:in ( film:hasSpokenLanguage )
        ] ;
        sh:message "spoken language must be one of the film's original languages" ;
    ] .
```

Finally, the most sophisticated shape uses a SPARQL query to ensure that if a director acts in a film, they must also be listed as an actor. This shape leverages the `sh:sparql` constraint to validate the `film:hasDirector` and `film:hasCast` relationships within the `:Film` class. It retrieves all films where the director is also listed as an actor but not as an actor in the cast.

```
film:DirectorActorShape
    a sh:NodeShape ;
    sh:targetClass film:Film ;
    sh:property [
        sh:path film:hasDirector ;
        sh:name "Director Acting Check" ;
        sh:message "If director acts in film, they must also be listed as actor" ;
        sh:sparql [
            sh:select """
                PREFIX film: <http://semantics.id/ns/example/film#>
                SELECT $this
                WHERE {
                    $this film:hasDirector ?person .
                    $this film:hasCast ?cast .
                    ?cast film:hasCastActor ?person .
                    FILTER NOT EXISTS {
                        $this film:hasActor ?person
                    }
                }
            """ ;
        ] ;
        sh:message "if director acts in film, they must also be listed as actor" ;
    ] .
```

In conclusion, the SHACL shapes were designed to ensure the structural integrity of the data and the relationships between entities, leveraging the graph-based nature of the data to provide more advanced validation capabilities.