WADe Project Report

DaVi: Ontology-Driven Semantic Data Visualization System

Authors

Nastasiu Stefan

Lazurca Samuel-Ionut

Repository

GitHub Repository

1. 1. Abstract

The DaVi (Data Visualization) project presents a web application designed to integrate, semantically enrich, and visualize heterogeneous datasets through Semantic Web technologies. Focusing on cybersecurity (NIST NVD) and media recommendation (MovieLens) domains, the system transforms raw JSON/CSV data into Resource Description Framework (RDF) graphs, stored in an Apache Jena Fuseki triplestore. A key innovation of DaVi is its **modular, ontology-driven architecture**: the frontend does not contain hardcoded logic for visualizations but instead queries the ontology to dynamically render components based on the semantic definition of data views. This report details the system's architecture, the custom knowledge models employed and the implementation of inverse relations for efficient querying.

2. 2. Introduction

In the context of the Web of Data, the separation between data storage and data presentation often leads to rigid systems that require significant refactoring when underlying data models change. DaVi addresses this by adhering to **Linked Data principles**, ensuring that all resources are identified by URIs and accessible via standard web protocols.

The project aims to provide a unified exploration interface for two distinct domains: software vulnerabilities (CVE, CWE, CPE) and movie recommendations. By leveraging a custom ontology hosted via **purl.org**, the system unifies these domains under a common visualization metamodel. The solution is deployed using a cloud-native approach, utilizing Google Cloud Platform for backend services and Vercel for frontend delivery, fulfilling the requirements for a modern, scalable Web Engineering project.

3. 3. Internal Data Structures & Knowledge Models

The core of DaVi is its data. We utilize large-scale datasets retrieved from **AcademicTorrents.com** to ensure reproducibility and access to persistent research data.

3.1 3.1. External Data Sources

- **NIST National Vulnerability Database (NIST):** Contains over 25,000 records of Common Vulnerabilities and Exposures (CVE), Common Weakness Enumerations (CWE), and Common Platform Enumerations (CPE).
- **MovieLens 20M (MVLN):** A stable benchmark dataset containing 20 million ratings, 465,000 tag applications, and genome scores describing movie attributes.

3.2 3.2. Ontology and Vocabulary Design

To manage this data, we developed a custom ontology suite, accessible via persistent URLs (PURL). The ontology is divided into three modules:

1. **DaVi Meta-Ontology (`davi-meta`):** Defines the visualization logic. It introduces concepts like `davi-meta:DataView`, `davi-meta:VisualizationOption`, and properties like `davi-meta:sparqlPath`. This layer allows the frontend to be agnostic of the domain data.
2. **NIST Domain Ontology (`davi-nist`):** Maps the NVD JSON structure to RDF, linking Vulnerabilities to Weaknesses and Software. It introduces concepts like `davi-nist:Vulnerability`, `davi-nist:Weakness`, and properties such as `davi-nist:affectsSoftware`.
3. **MovieLens Domain Ontology (`davi-mov`):** Models movies, users, ratings, and tag genomes. It introduces concepts like `davi-mov:GenomeTag` and properties like `davi-mov:relevanceScore`.

Key Architectural Decision: Inverse Relations

To optimize graph traversal, the ontology explicitly defines inverse properties using `owl:inverseOf` logic and SPARQL paths. For instance, while the raw data links a *Vulnerability* to *Software* (`davi-nist:affectsSoftware`), the ontology defines an inverse view property `davi-nist:affectedByVulnerability` with the path `^davi-nist:affectsSoftware`. This allows the system to instantly query "Which software is most vulnerable?" without expensive SPARQL joins, simply by following the inverse path.

3.3 3.3. RDF Transformation Pipelines

We developed a suite of Python-based ETL (Extract, Transform, Load) scripts to convert the raw data into Turtle (TTL) format. These scripts sanitize URIs, map dates to `xsd:date`, and link entities to the custom ontology.

- `cve_parser.py`: Extracts CVE details like names, descriptions, weaknesses, and references. Also extracts nested JSON metrics (CVSS scores) and creates `davi-nist:CVSSMetric` nodes. It uses batch processing to handle large datasets efficiently.

```
# CVE parser output example snippet
cve:CVE-2000-1208 a davi-nist:Vulnerability ;
schema:name "CVE-2000-1208" ;
  dcterms:identifier "CVE-2000-1208" ;
  schema:creativeWorkStatus "Deferred" ;
  schema:dateModified "2025-04-03T01:03:51.193000"^^xsd:dateTim
schema:datePublished "2002-08-12T04:00:00"^^xsd:dateTime ;
  schema:description "Format string vulnerability in startprint
  davi-nist:affectsSoftware cpe:5ECDDE72-C03D-40E5-955F-168E10A
      cpe:85A510C4-D66B-4AEA-876E-ACA639D8F6E4,
      cpe:CCC8529F-6BD1-4B01-BA9E-803ADFFEC05A,
      cpe:DD14A51A-1312-4D78-AE17-0233EB5C44FE,
      cpe:DFD3469F-CB35-451B-82F1-607AEEB3CA1C,
      cpe:F777EFC8-241B-449F-BA4A-E55B4B90C1FC ;
  davi-nist:hasCVSSMetric davi-nist:metric_CVE-2000-1208_v2 .
```

Figure 1: A CVE entry representing a software vulnerability with linked software and metrics.

- `cpe_parser.py`: Processes CPE entries and their vendors. Saves a map of CPE names to ids to link vulnerabilities to affected software in the CVE parser.

```
# CPE parser output example snippet
cpe:000007A8-79E3-4DF7-A00E-326188269361 a schema:SoftwareAppli
  rdfs:label "AG-Grid 19.1.3"@en ;
  dcterms:identifier "000007A8-79E3-4DF7-A00E-326188269361" ;
  schema:manufacturer vendor:ag-grid ;
  schema:name "ag-grid" ;
  schema:softwareVersion "19.1.3" ;
  davi-nist:cpe23 "cpe:2.3:a:ag-grid:ag-grid:19.1.3:*:*:*:*:*:*
```

Figure 2: A CPE entry representing a software application with vendor linkage.

- `cwe_parser.py`: Parses CWE entries, extracting hierarchical relationships using `skos:broader` and `skos:narrower` properties. It builds a comprehensive weakness taxonomy using rdflib to create the RDF graph.

```
# CWE parser output example snippet
cwe:1004 a skos:Concept,
      davi-nist:Weakness ;
  dcterms:identifier "CWE-1004" ;
  schema:description "The product uses a cookie to store sensit
  skos:broader cwe:732 ;
  skos:prefLabel "CWE-1004: Sensitive Cookie Without 'HttpOnly'
```

Figure 3: A CWE entry representing a software weakness with hierarchical relationships.

- `convert_movies_final.py`: Transforms CSVs about movies, users, ratings, and tag genomes into RDF, handling the massive volume of Rating objects by batching triples (schema.org)..

```
# MovieLens parser output example snippet
imdb:0114709 a schema:Movie ;
  schema:name "Toy Story (1995)" ;
  dcterms:identifier "1" ;
  schema:datePublished "1995-01-01"^^xsd:date ;
  schema:genre genre:adventure, genre:animation, genre:family,
  .
```

Figure 4: A MovieLens entry representing a movie with genres and identifiers.

4. 4. System Architecture

DaVi adopts a multi-layered architecture, strictly separating the data layer, the semantic mediation layer (API), and the presentation layer.

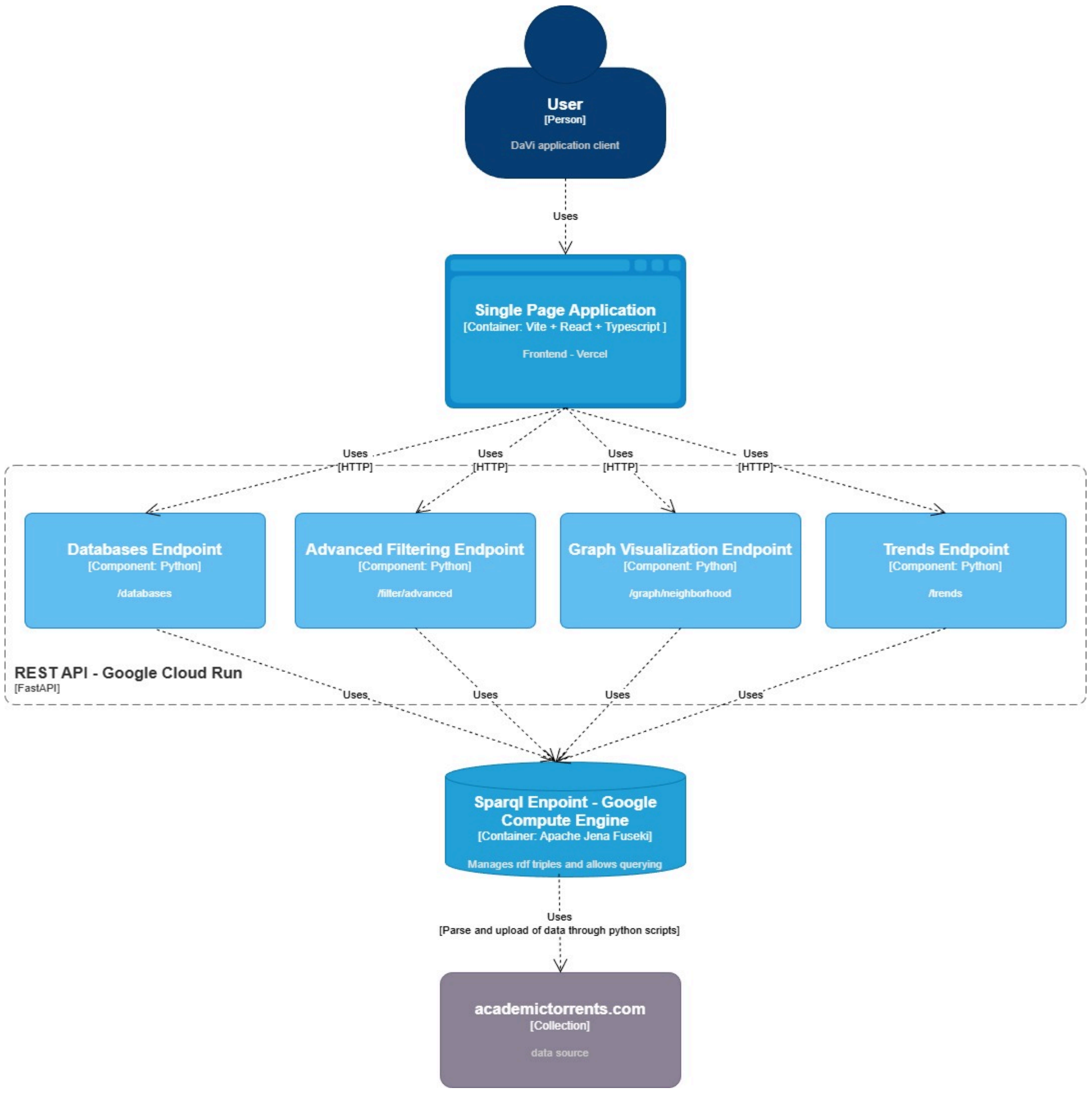

Figure 5: C4 Diagram illustrating the multi-layered architecture of the DaVi system.

4.1 4.1. Semantic Data Store: Apache Jena Fuseki

The persistence layer is powered by **Apache Jena Fuseki**. We utilize the **TDB2** storage engine for high-performance triple storage. In hope for efficient text search, we configured a **Lucene Text Index**.

Lucene Integration: Standard SPARQL `FILTER regex()` operations are computationally expensive (O(n)). By defining a `text:EntityMap` in the Fuseki configuration, we index properties like `schema:description` and `davi-mov:tagContent`. This allows the backend to use `text:query` for sub-millisecond keyword searches, essential for the "Intelligent Filter" feature. But we didn't see a big improvement on our data.

Statistics Computation is performed using **tdbstats** command-line tool to precompute graph statistics, enabling efficient query planning.

Unfortunately, having a big dataset poses challenges for indexing and query performance. Applications like DbPedia and Wikidata have bigger resources and more mature infrastructures, so in order for us to have a decent latency, we had to limit the dataset size drastically during development and testing phases.

4.2 4.2. Semantic Mediation Layer: FastAPI

The backend is developed using **Python FastAPI**. It acts as a semantic mediator, translating RESTful requests into SPARQL queries. It exposes an OpenAPI specification (Swagger UI) for easy consumption.

It is organized into routers that call service functions that implement the core business logic.

Complex Query Optimization: The API implements intelligent query builders that inspect the request structure. If a user filters by a textual field, the builder injects the `text:query` predicate. If the filter implies a hierarchy (e.g., "Find all weaknesses related to X"), it utilizes SPARQL 1.1 Property Paths (e.g., `skos:broader*`) to perform recursive transitive closures over the graph.

We tried to make the API seem more "intelligent" by making the functions generic enough to handle multiple datasets and views.

4.3 4.3. The Frontend Architecture

The frontend is a Single Page Application (SPA) built with **React**, **Vite**, and **TypeScript**. Unlike traditional dashboards, the DaVi frontend is **data-agnostic**.

It dynamically constructs the UI based on the ontology definitions retrieved at runtime. Tabs and dashboard panels are not using routes but are conditionally rendered components based on the selected dataset and view.

Ontology-Driven UI: When a user selects a dataset (e.g., NIST), the frontend queries the ontology for available `davi-meta:DataViews`. It inspects the `davi-meta:supportsVisualization` properties to determine which components to render. For example, if the ontology asserts that the *Publication Date* property supports `davi-meta:viz_timeline`, the frontend automatically instantiates a Line Chart component. This "Perfect Modularity" allows us to add new datasets (e.g., Biology or Finance) simply by updating the ontology, without deploying new frontend code.

- **GraphExplorer:** A force-directed graph visualization component (using `react-force-graph`) that allows users to traverse the RDF network interactively. Users can view the graph in 2D or 3D, zoom into nodes, and inspect properties. Clicking on a node fetches more related nodes so the graph can expand dynamically and the user can explore the data. Groups of nodes can be color-coded based on their RDF types for better visual distinction (e.g., Vulnerabilities vs. Software).
- **Analytics Builder:** Generates filter inputs based on the `davi-meta:isDimension` properties of the current view. The user can choose dimensions to filter and aggregate data dynamically.
- **Filter Panel:** Provides a user interface for applying multiple filters to the dataset, enhancing data exploration capabilities. This time in list form, allowing for complex boolean logic combinations.

5. 5. Linked Data

5.1 5.1. SPARQL & Linked Data Conformity

Every resource is identified by a dereferenceable URI. We use PURLs to ensure persistent access to ontology definitions and we tried to use as many known vocabularies as possible (Schema.org, Dublin Core, SKOS, OWL) and to use real links for subjects and objects as much as possible.

The backend API transforms RESTful requests into SPARQL queries, adhering to Linked Data principles (WADE). For example, to retrieve all vulnerabilities affecting a specific software, the API constructs a SPARQL query that navigates the RDF graph using the defined properties.

```
# Parameterized SPARQL Query Template for Advanced Analytics
SELECT ?groupKey {selection}
```

```
        WHERE {{
            {class_filter}
            ?s {dim_pred} ?dimVal .

            # Dynamic Label Resolution for the Group Key
            OPTIONAL {{ {group_node} rdfs:label | schema:name | skos:pref
            BIND(COALESCE(STR(?lbl), STR({group_node})) as ?groupKey)

            {metric_pattern}
            {"FILTER(BOUND(?metricRaw))" if metric else ""}
        }}
        GROUP BY ?groupKey
        ORDER BY DESC(?val)
        LIMIT {limit}
```

Figure 2: A backend query demonstrating dynamic SPARQL construction based on user-selected dimensions and metrics.

6. 6. Deployment

The deployment strategy follows a cloud-native approach, ensuring scalability and availability.

6.1 6.1. Backend & Triplestore (Google Cloud Platform)

The FastAPI application is containerized using Docker and deployed on **Google Cloud Platform (GCP)** using Cloud Run. This provides a serverless environment that auto-scales based on traffic. The Apache Jena Fuseki triplestore is hosted on a Compute Engine VM with 4 vCPUs and 16GB RAM to handle the RDF dataset.

6.2 6.2. Frontend (Vercel)

The React SPA is deployed via **Vercel**. The communication between the frontend and backend is secured using HTTPS, CORS and also it is API key-based.

7. References

NIST

nist-national-vulnerability-database-cve-cpe-cwe by US National Institute of Standards & Technology.

MVLN

MovieLens 20M Dataset.

schema.org

schema.org.

WADE

WADe Syllabus

GCP

Google Cloud Platform.