GENERAL ARCHITECTURE

The **ELAN (Esoteric Language Explorer)** application helps users discover and learn about unique programming languages. It allows users to find detailed information about different esoteric languages by selecting specific search options. The app is designed to make exploring these unusual languages easy and enjoyable, with a user-friendly interface. Additionally, it provides useful details about tools and characteristics related to these languages. The system can also recommend similar esoteric languages and display a comparison table of their characteristics.

1. Software Architecture Overview

The web application follows a **client-server architecture** with a **RESTful API** backend. The system consists of:

- **Frontend (Client Side)**: Built with **HTML, CSS, and vanilla JavaScript**, responsible for rendering the user interface and handling user interactions.
- **Backend (Server Side)**: A **Flask-based REST API** written in **Python**, which processes client requests and serves data.
- **Ontology Storage**: Apache **Jena Fuseki** is used to store and query the esoteric programming language data locally during development and later deployed to the cloud.
- **Communication**: The frontend communicates with the backend via **AJAX requests (fetch API)**, and the backend retrieves data from the ontology using **SPARQL queries**.

---

2. Main Modules

Frontend (Client-Side)

- **User Interface (UI)**
  - Displays a list of esoteric programming languages.
  - Allows filtering by **designer** and **year**.
  - Provides a **search bar** for finding specific languages.
- **Client-side Logic (JavaScript)**
  - Handles fetching data from the backend via **REST API calls**.
  - Dynamically updates the DOM to display filtered search results.

Backend (Server-Side)

- **Flask API**
  - Exposes RESTful endpoints (e.g., `/languages/details`, `/languages/{name}`, `/languages/designers`).

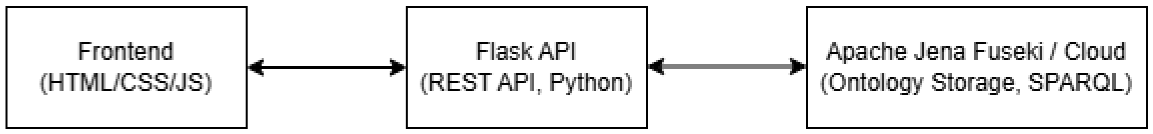
    - Processes frontend requests and interacts with the ontology.
- **Ontology Processing**
    - Uses Apache Jena Fuseki to store esoteric language data.
    - Executes **SPARQL queries** to fetch relevant information.
- **Data Handling**
    - Formats retrieved ontology data as **JSON** responses for the frontend.

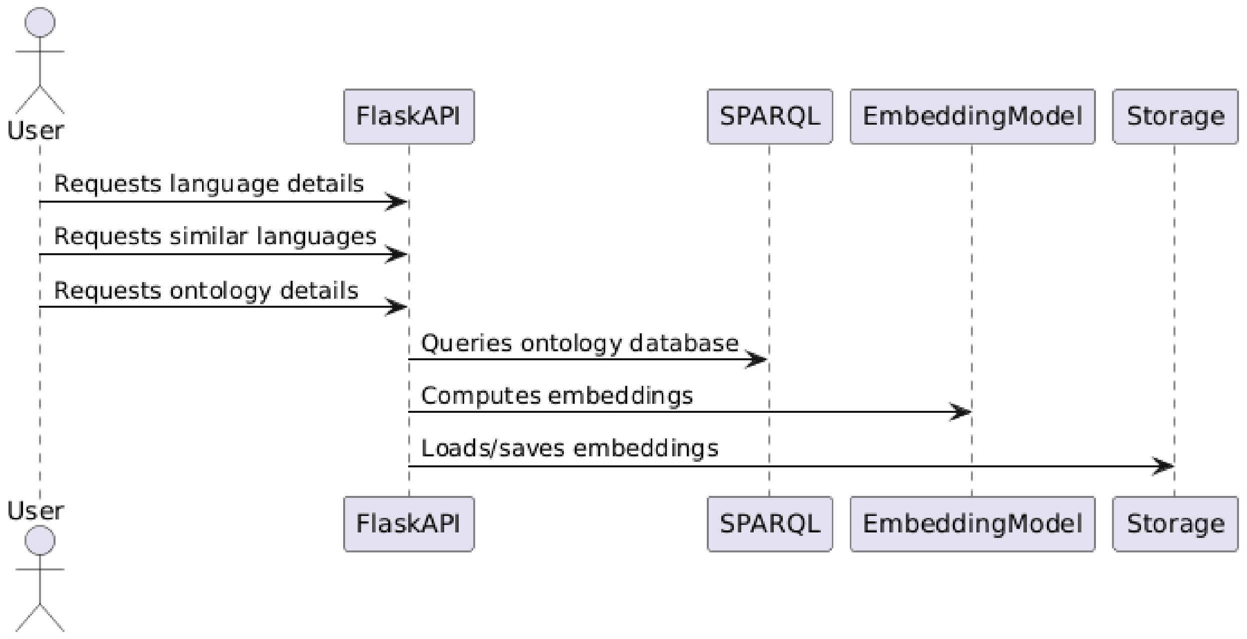
Ontology Storage - Initially hosted **locally** with Apache Jena Fuseki, then **deployed to the cloud**.

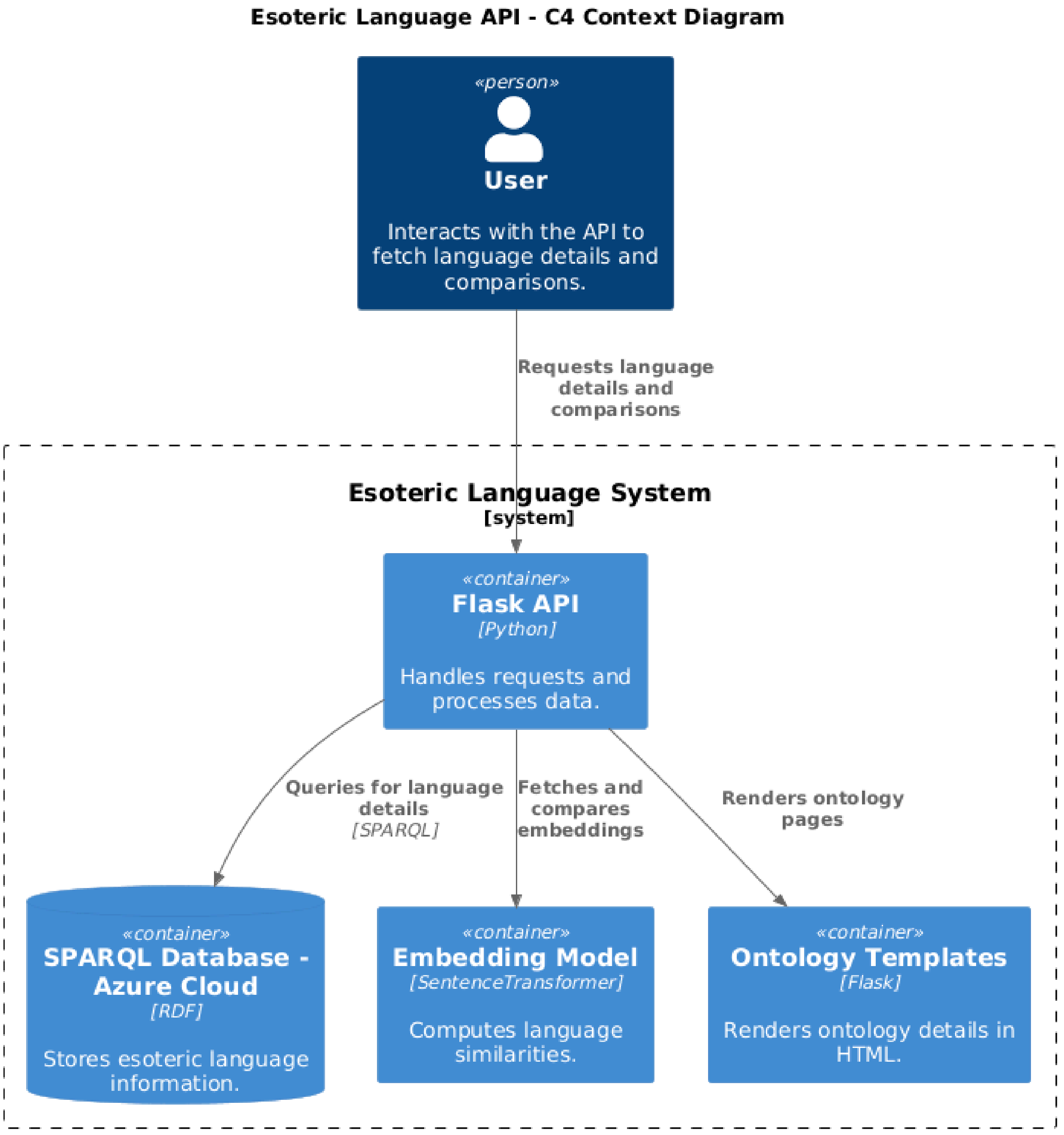
- **Apache Jena Fuseki**
    - Stores and serves the ontology containing esoteric language metadata.
    - Enables querying using **SPARQL**.

---

3. Software Architecture Diagram






Esoteric Language API - C4 Context Diagram
«person»
User
Interacts with the API to fetch language details and comparisons.
Requests language details and comparisons
Esoteric Language System
[system]
«container»
Flask API
[Python]
Handles requests and processes data.
Queries for language details
[SPARQL]
Fetches and compares embeddings
Renders ontology pages
«container»
SPARQL Database - Azure Cloud
[RDF]
Stores esoteric language information.
«container»
Embedding Model
[SentenceTransformer]
Computes language similarities.
«container»
Ontology Templates
[Flask]
Renders ontology details in HTML.

---

4. Development of the ontology

The process of developing the ontology involved multiple stages, utilizing both automated data retrieval and manual editing.

Ontology Structure Creation with Protégé

The initial step in the ontology development was to define the **structure** of the ontology using **Protégé**, a widely-used tool for creating and managing ontologies. Protégé enabled us to model various concepts and relationships based on the domain of esoteric programming languages.

In Protégé, we created several **classes** to represent different categories and concepts, including:

- **ComputationalClass**
- **Paradigm**
- **TechnicalCharacteristic**
- **Usability**
- **SpecificTypeOrFeature**

Each class was defined with relevant **properties** and **relationships** to other classes.

Data retrieval and ontology population with Python

To populate the ontology with data, we developed a **Python program** to automatically retrieve information from **esolang.org** and **dbpedia.org**.

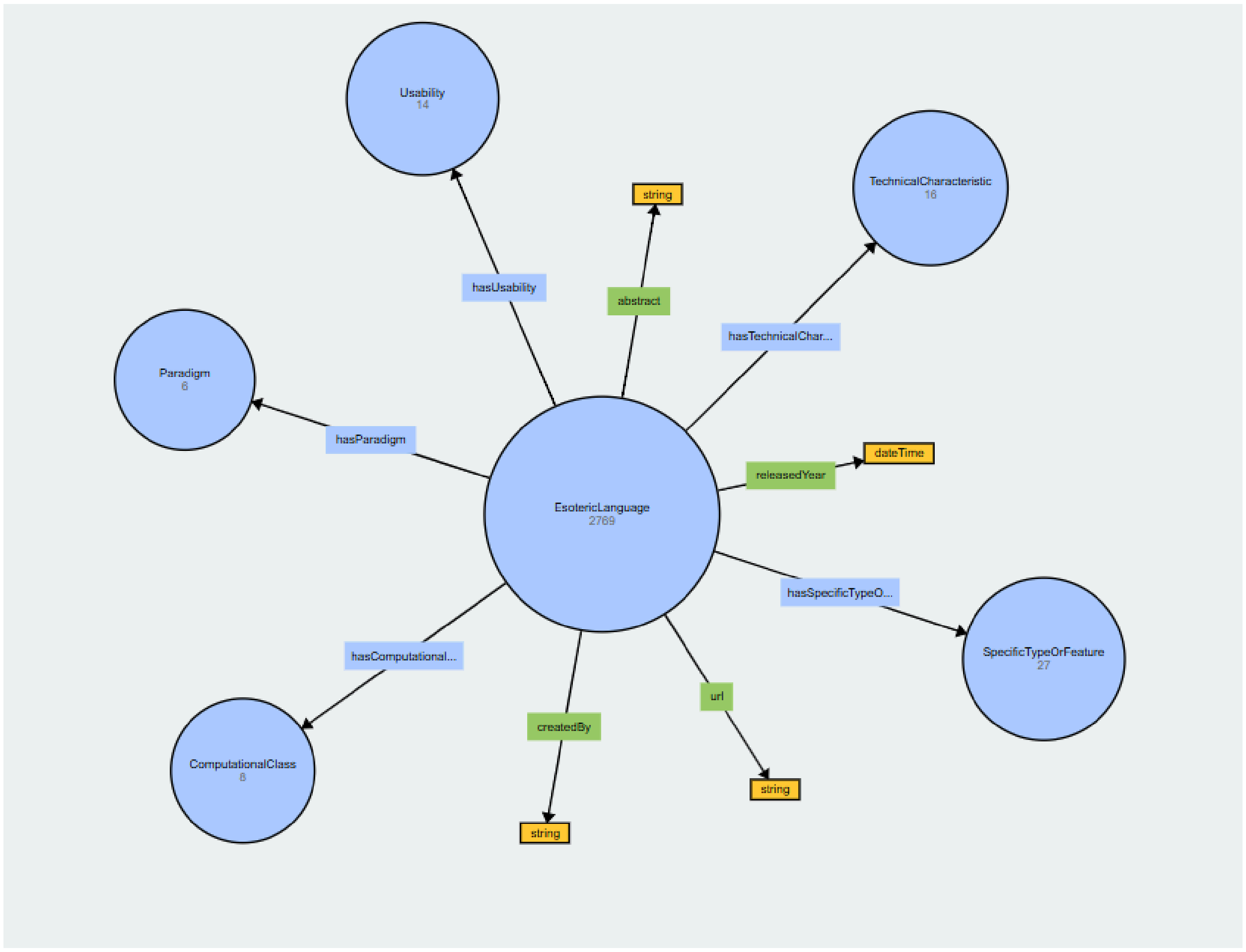
We initially started by gathering information from the esolang.org website. We began with an XML dump containing a list of esoteric programming languages, including their names, categories, and some basic details. We then parsed this XML file and converted the data into a CSV format. This allowed us to organize and work with the data more efficiently. The CSV file included columns for the language names, their categories, descriptions, and other relevant details.

Using Python and the RDFLib library, we then took the data from the CSV file and populated the ontology. We linked these entities to their respective categories, such as **Programming Paradigm**, **Computational Class**, and **Technical Characteristics**. Additionally, we included the language abstracts and URLs for further information.

However, some languages in the dataset lacked specific details, such as abstracts or the names of their designers. To address this, we turned to DBpedia, a structured database built from Wikipedia. We used DBpedia to retrieve missing information for those languages and added the relevant data to the ontology.

The result is a comprehensive RDF ontology with detailed information about various esoteric languages, including their **names**, **descriptions**, **paradigms**, **computational classes**, and **related technical features**.

After developing and populating the esoteric programming languages ontology, we visualized it using **WebVOWL**, an interactive tool designed for visualizing ontologies in a clear and intuitive manner. Below is the image of the final form of the populated ontology in WebVOWL:



---

5. Data Flow & Task Flow

Request Handling Flow

1. The user visits the webpage, and the frontend sends a `GET` request to the backend to fetch all languages.
2. The backend queries the ontology using **SPARQL** and returns a **JSON response**.
3. The frontend dynamically renders the list of languages.
4. If the user applies filters (by designer or year) or searches for a language:
    - The frontend sends a request with parameters.
    - The backend processes the request, queries the ontology, and returns filtered results.
    - The frontend updates the displayed list accordingly.

Data Flow Example

- User Action -> Frontend (JS Fetch API) -> Flask Backend -> Fuseki SPARQL Query

  -> JSON Response -> Frontend (JS) -> Update UI

---

6. Encountered difficulties

Handling the Large XML File (esolang.xml)

One of the biggest challenges was dealing with the *esolang.xml* file, which is a dump of data from the esolangs.org wiki. The file was 1.5 GB, which made it hard to work with. Some of the issues we faced were:

- **Parsing the File**: The file was so large that it was difficult to load and process in memory. We had to use methods to read the file in smaller parts to avoid crashing the system. This slowed down the process but was necessary to handle the file properly.
- **Understanding the Data**: The XML file wasn't easy to understand. We had to spend time figuring out how the data was structured to extract the information we needed, like language names, descriptions, and other details.
- **Cleaning the Data**: Some of the data was incomplete or incorrect, which required additional steps to clean and fix before using it.
- **Converting to CSV**: After cleaning the data, we needed to convert it into a CSV file. This was another challenge because we had to make sure the right information was in the right columns.

Designing the Ontology

Another major challenge was deciding what data to include in our ontology. We needed to figure out which details to keep in the model and display on the website. Some of the problems we faced were:

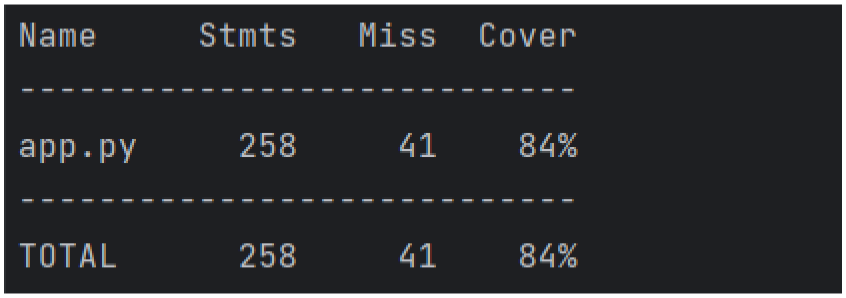
- **Choosing the Right Information**: We had to balance between including enough data to be useful and not overwhelming the users with too much detail. For example, we had to decide whether to show the technical aspects of a language or just the basics, like the designer and the language's features.
- **Making Sure It Made Sense**: The goal was to make the information easy to understand for users, but also accurate. We wanted the language data to be organized in a way that users could easily search and filter based on what mattered most to them.
- **Keeping It Flexible**: The structure of the ontology needed to be flexible enough to allow for new languages or changes later on. For example, initially, the ontology did not include a property for the URL link to each language's page on the esolangs.org wiki.

This was an important missing piece since users might want to visit the original page for more details about a language. We had to update the ontology to include this URL as a data property and ensure it was correctly linked to each language.

---

7. Testing

We thoroughly tested our Elan app to make sure it works smoothly. In functional testing, all unit tests successfully passed, affirming the accuracy and reliability of the application's core functionalities. Furthermore, the achieved test coverage of 84% reflects a thorough examination of the codebase, ensuring a solid foundation for future development.



```
Name      Stmts   Miss  Cover
-----------------------------
app.py      258     41    84%
-----------------------------
TOTAL       258     41    84%
```

---

8. Future Considerations

Scalability

- The current architecture works well for a lightweight web app.
- If future expansion is needed, **authentication** and **database integration** could be added.

Performance

- SPARQL queries should be optimized for faster retrieval.
- Implement **caching** if necessary to avoid redundant queries.

---

8. Resources

Below are the resources that were used in the research, development, and documentation process:

- WADe

- Protégé Tutorial
- WebVOWL
- SPARQL Introduction