

Project Report: Vehicle Ontology and Data Integration

Authored By: Faisal Saimeh

Overview

The "Vehicle Ontology and Data Integration" project aims to harness the power of semantic web technologies to unify and enhance the accessibility of diverse vehicle-related data sources. By adopting the Resource Description Framework (RDF) and creating a dedicated vehicle ontology, this project seeks to standardize data representation across different datasets, enabling more effective data management and querying capabilities.

The project focuses on integrating various forms of vehicle data, including features, models, and pricing from multiple sources, into a coherent ontology-based framework. This approach not only facilitates robust data interconnections but also supports advanced data retrieval through SPARQL queries, demonstrating significant advantages over traditional relational database systems.

The project's scope included handling data from structured CSV files and converting this data into RDF format, although it was limited to non-commercial usage and focused on passenger vehicles.

By employing technologies such as **“rdflib”** for RDF graph management and Flask for API development, the project lays the groundwork for future expansions and potential applications in automotive sales platforms, vehicle market analytics.

Methodology

Data Collection

Data Source Description

The datasets used in this project were obtained from Kaggle, a well-known platform that provides access to a wide variety of data for analysis and machine learning projects. These datasets contain comprehensive information about vehicles, including details such as brands, models, features, prices, and technical specifications. The decision to use Kaggle datasets was driven by their richness in detail and the diversity of the data, which provided a robust foundation for developing vehicle ontology.

Datasets Details

1. Car Data Set 1:

- This dataset includes basic vehicle information such as make, model, year of manufacture, features, and price. It is structured to give a broad overview of the vehicle market and includes data spanning multiple years and manufacturers.

2. Car Data Set 2:

- The second dataset complements the first by providing additional details such as engine specifications, transmission type, fuel type, and more detailed pricing information. This dataset allows for a deeper analysis and richer integration into the vehicle ontology to provide a more detailed representation of each vehicle.

Data Preparation and Challenges

The data obtained from Kaggle required significant preprocessing to ensure compatibility with the ontology schema used in this project. The following steps were undertaken to prepare the data for integration:

- **Data Cleaning:** Inconsistencies and missing values were addressed, and data formats were standardized across different fields to ensure uniformity.
- **Data Transformation:** The data from CSV format was converted into RDF triples, which involved mapping CSV columns to the corresponding properties defined in the vehicle ontology.
- **Data Integration:** The RDF data from both datasets was merged into a single RDF graph to enable comprehensive querying and analysis. This step was crucial for maintaining relational integrity and ensuring seamless navigation across different data points within the ontology.

Challenges Encountered

The primary challenges during data collection included dealing with incomplete or inconsistent data entries and the complexity of accurately mapping the CSV data to the ontology without losing contextual details. Additionally, ensuring that all relevant data from both datasets was integrated into a cohesive RDF structure posed certain technical challenges, which were addressed through iterative testing and refinement of the data transformation scripts.

Ontology Design: Vehicle Ontology Schema Overview

Vehicle Ontology provides a comprehensive structure for representing complex vehicle data in a semantically rich manner. The schema, visualized in the provided diagram, encapsulates the intricate relationships and properties associated with vehicles. This schema is crucial for enabling detailed queries and analyses, leveraging the interconnected nature of the vehicle data.

Classes and Relationships

- **Vehicle:** The central node in the ontology, representing a vehicle. It connects directly to various attributes and categories, making it the focal point for data retrieval and analysis.
- **Brand and Model:** These classes are crucial for categorizing vehicles. The 'Brand' represents the make of the vehicle, while 'Model' provides specific versions or designs under a brand. Each vehicle is linked to a single brand and model, denoted by the properties `hasBrand` and `hasModel`.
- **Engine and Transmission:** These classes detail the mechanical aspects of a vehicle. 'Engine' covers specifics such as engine type and horsepower, while 'Transmission' details the type of transmission system (e.g., manual, automatic). Vehicles are connected to these attributes through `hasEngine` and `hasTransmission` properties.
- **FuelType and DriveType:** 'FuelType' indicates the kind of fuel a vehicle uses (e.g., gasoline, diesel, electric), and 'DriveType' describes the drive configuration (e.g., FWD, RWD). These are linked to the Vehicle class by `usesFuel` and `hasDriveType` properties.
- **VehicleType and Feature:** 'VehicleType' categorizes vehicles into types like SUV, sedan, or hatchback (`isTypeOf` property), while 'Feature' encompasses additional features a vehicle might have, such as sunroofs or navigation systems, connected via the `hasFeature` property.
- **Owner and Dealership:** These classes provide ownership and sales information. A vehicle is linked to its owner and the dealership from which it was purchased using `ownedBy` and `soldBy` properties.
- **Warranty and ServiceRecord:** 'Warranty' provides warranty details, and 'ServiceRecord' contains the vehicle's service history. These are crucial for after-sales service and maintenance tracking, linked by `coveredBy` and `hasServiceRecord`.
- **MarketCategory and PerformanceStats:** 'MarketCategory' classifies vehicles into market segments like luxury or economy, and 'PerformanceStats' details performance characteristics such as acceleration or top speed. These are associated with vehicles through `inMarketCategory` and `hasPerformanceStats` properties.

Diagram Layout

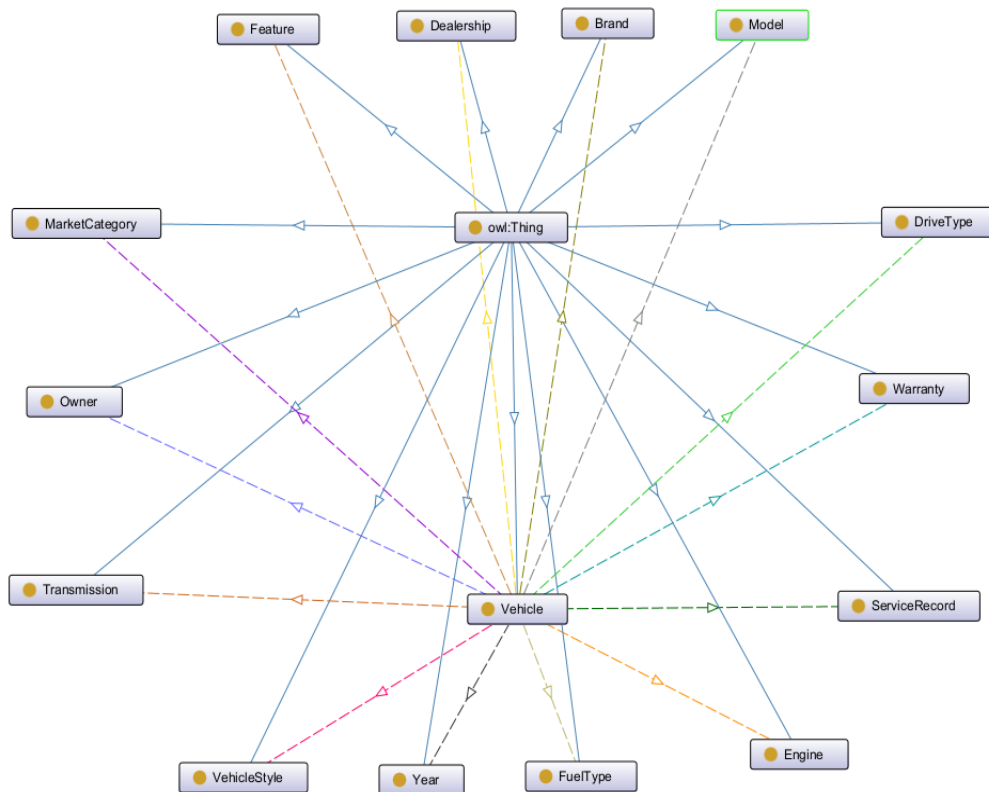
- The diagram visually presents these classes and their connections radiating from the central 'Vehicle' class. Each class is linked by directed arrows, indicating the nature of their relationships (e.g., a vehicle 'hasModel' a specific model).

- Different line styles may represent various types of relationships, enhancing the diagram's readability and providing a quick reference to understand possible queries that can be made against the ontology.

Implementation and Use

- The ontology is implemented in a format suitable for semantic web technologies, likely using RDF/OWL standards. This allows for robust data integration and sophisticated querying capabilities using SPARQL.
- Practical applications of this ontology include querying for specific vehicle features, comparing models across different brands, or analyzing market trends based on vehicle data.

Ontology Diagram



RDF Integration

Overview

The integration of vehicle data into a unified RDF graph is a critical step in leveraging the semantic web capabilities of the project. This process involves converting structured data from CSV files into RDF format using Python scripts. The RDF (Resource Description Framework) allows for the representation of information in a way that makes semantic relationships explicit, which is crucial for the subsequent querying and data analysis phases.

Tools and Libraries Used

- **Python:** The scripting language used for handling data transformation and manipulation.
- **RDFlib:** A Python library designed for working with RDF. RDFlib provides tools to create, store, manipulate, and retrieve RDF data.
- **Namespace and Literal from RDFlib:** These functionalities are used to define custom namespaces and create typed literals, ensuring data consistency and adherence to the ontology schema.

Data Conversion Process

The data conversion involves several key steps, outlined below for each dataset:

1. **Initialization of the RDF Graph:**
 - An instance of the RDF graph is created using `Graph()` from `RDFlib`.
 - A custom namespace is defined (`http://example.org/vehicle/`) to ensure that all RDF triples are properly namespaced.
2. **Definition of Custom Properties:**
 - Properties such as `brand`, `model`, `price`, `fuelType`, `transmission`, `drive`, `engineHP`, etc., are defined using the namespace to ensure they are consistent with the ontology.
3. **Adding Data to the RDF Graph:**
 - For each record in the dataset, a unique URI for the vehicle is created. This URI acts as the subject of the RDF triples.
 - RDF triples are then constructed for each attribute of a vehicle (e.g., `brand`, `model`, `price`) by linking the vehicle URI with the defined properties and their corresponding values from the CSV file.
4. **Serialization of the RDF Graph:**
 - The completed RDF graph is serialized into a Turtle (TTL) format file. This format is chosen for its readability and ease of integration with other semantic web tools.

Challenges Encountered

- **Data Quality Issues:** Inconsistencies and missing values in the original CSV data required preprocessing to ensure that the RDF conversion did not propagate errors.
- **Type Handling:** Ensuring that data types (e.g., strings, integers) were correctly handled and converted for RDF literals required careful scripting and validation.

Screenshots of Code for Data Conversion

Convert data_1 to RDF

```

Code to Convert data_1 to RDF

# Create an RDF graph
g1 = Graph()
ns = Namespace("http://example.org/vehicle/")
Vehicle = ns.Vehicle
✓ 0.0s

# Define custom properties
brand = ns.brand
model = ns.model
price = ns.price
fuelType = ns.fuelType
transmission = ns.transmission
drive = ns.drive
engineHP = ns.engineHP
age = ns.age
✓ 0.0s

# Add data to the RDF graph
for index, row in data_1.iterrows():
    vehicle_uri = URIRef(ns[f"Vehicle_1_{index}"])
    g1.add((vehicle_uri, RDF.type, Vehicle))
    g1.add((vehicle_uri, brand, Literal(row['Brand'], datatype=XSD.string)))
    g1.add((vehicle_uri, model, Literal(row['Model'].strip(), datatype=XSD.string)))
    g1.add((vehicle_uri, price, Literal(row['Price'], datatype=XSD.integer)))
    g1.add((vehicle_uri, fuelType, Literal(row['Fuel Type'], datatype=XSD.string)))
    g1.add((vehicle_uri, transmission, Literal(row['Transmission'], datatype=XSD.string)))
    g1.add((vehicle_uri, drive, Literal(row['Drive'], datatype=XSD.string)))
    g1.add((vehicle_uri, engineHP, Literal(row['Engine HP'], datatype=XSD.integer)))
    g1.add((vehicle_uri, age, Literal(row['Age'], datatype=XSD.integer)))
✓ 16.6s

# Save the graph to a file
rdf_file_path_1 = "data_1.ttl"
g1.serialize(destination=rdf_file_path_1, format="turtle")
✓ 16.0s

<Graph identifier=Nc0e4688cb4cd4e488d74f44e123bffb5 (<class 'rdflib.graph.Graph'>>

```

Convert data_2 to RDF

Code to Convert data_2 to RDF

```
# Create an RDF graph
```

```
g2 = Graph()
ns = Namespace("http://example.org/vehicle/")
Vehicle = ns.Vehicle
```

[16] ✓ 0.0s

```
# Define custom properties
```

```
brand = ns.brand
model = ns.model
price = ns.price
fuelType = ns.fuelType
transmission = ns.transmission
drive = ns.drive
engineHP = ns.engineHP
year = ns.year
engineCylinders = ns.engineCylinders
numberOfDoors = ns.numberOfDoors
marketCategory = ns.marketCategory
vehicleSize = ns.vehicleSize
vehicleStyle = ns.vehicleStyle
highwayMPG = ns.highwayMPG
cityMPG = ns.cityMPG
popularity = ns.popularity
```

[17] ✓ 0.0s

```
# Add data to the RDF graph
```

```
for index, row in data_2.iterrows():
    vehicle_uri = URIRef(ns[f"Vehicle_2_{index}"])
    g2.add((vehicle_uri, RDF.type, Vehicle))
    g2.add((vehicle_uri, brand, Literal(row['Brand'], datatype=XSD.string)))
    g2.add((vehicle_uri, model, Literal(row['Model'].strip(), datatype=XSD.string)))
    g2.add((vehicle_uri, year, Literal(row['Year'], datatype=XSD.integer)))
    g2.add((vehicle_uri, fuelType, Literal(row['Fuel Type'], datatype=XSD.string)))
    g2.add((vehicle_uri, engineHP, Literal(row['Engine HP'], datatype=XSD.float)))
    g2.add((vehicle_uri, engineCylinders, Literal(row['Engine Cylinders'], datatype=XSD.integer)))
    g2.add((vehicle_uri, transmission, Literal(row['Transmission'], datatype=XSD.string)))
    g2.add((vehicle_uri, drive, Literal(row['Driven Wheels'], datatype=XSD.string)))
    g2.add((vehicle_uri, numberOfDoors, Literal(row['Number of Doors'], datatype=XSD.integer)))
    g2.add((vehicle_uri, marketCategory, Literal(row['Market Category'], datatype=XSD.string) if pd.notna(row['Market Category']) else Literal("N/A")))
    g2.add((vehicle_uri, vehicleSize, Literal(row['Vehicle Size'], datatype=XSD.string)))
    g2.add((vehicle_uri, vehicleStyle, Literal(row['Vehicle Style'], datatype=XSD.string)))
    g2.add((vehicle_uri, highwayMPG, Literal(row['Highway MPG'], datatype=XSD.integer)))
    g2.add((vehicle_uri, cityMPG, Literal(row['City MPG'], datatype=XSD.integer)))
    g2.add((vehicle_uri, popularity, Literal(row['Popularity'], datatype=XSD.integer)))
    g2.add((vehicle_uri, price, Literal(row['Price'], datatype=XSD.integer)))
```

[18] ✓ 8.2s

```
# Save the graph to a file
```

```
rdf_file_path_2 = "data_2.ttl"
g2.serialize(destination=rdf_file_path_2, format="turtle")
```

[19] ✓ 8.5s

... <Graph identifier=N9a831a5292c548c18c1333a44851746f (<class 'rdflib.graph.Graph'>)>

Implementation

RDF Graph Setup

Overview

Setting up the RDF graph for the project involved loading individual datasets and the vehicle ontology into separate RDF graphs, then merging them into a single unified graph. This process is critical for ensuring that all data is interconnected, and that the ontology accurately represents the relationships within the data.

Tools and Libraries Used

- **RDFlib:** A Python library that facilitates the manipulation and querying of RDF data. RDFlib supports parsing and serializing RDF graphs in various formats, making it ideal for handling the project's RDF needs.
- **Python:** The scripting language used to execute RDFlib functions and manage RDF data.

Graph Initialization and Loading

The RDF graph setup involved several key steps:

1. **Initialization of RDF Graphs:**
 - Separate instances of RDFlib's `Graph()` are created for the ontology and each dataset to ensure that data sources are individually manageable before merging.
2. **Loading the Ontology:**
 - The vehicle ontology, which provides the schema for the data, is loaded into its dedicated graph. The ontology is typically in RDF/XML format and is parsed using RDFlib's `parse()` function with the appropriate format specified.
3. **Loading Data Graphs:**
 - Each dataset, previously converted into Turtle format, is loaded into its respective graph. Turtle (TTL) is chosen for its readability and ease of use within the semantic web community.

Merging the Graphs

- **Graph Merging Strategy:**
 - Once individual graphs for the ontology and datasets are set up, they are combined into a single graph. This merging ensures that the ontology definitions are directly applicable to the dataset entries, facilitating accurate querying and data analysis.
 - The `+` operator in RDFlib is used to merge these graphs seamlessly.

- **Serialization of the Merged Graph:**
 - The merged graph is then serialized back into Turtle format for storage or further processing. This step consolidates all the data and schema information into one file, making it easier to handle and distribute.

Screenshot of Code for Graph Setup and Merging

Graph Setup and Merging the Graphs with Fatcs & Schema

```
# Initialize RDFLib Graphs
ontology_graph = Graph()
data_graph_1 = Graph()
data_graph_2 = Graph()

# Load ontology - update path and format as needed
ontology_graph.parse("Vehicle Ontology Schema Overview.rdf", format='xml')

# Load your data from Turtle files
data_graph_1.parse("data_1.ttl", format='turtle')
data_graph_2.parse("data_2.ttl", format='turtle')

# Merge the graphs
merged_graph = ontology_graph + data_graph_1 + data_graph_2

# Serialize the merged graph to a Turtle file
merged_graph.serialize("merged_ontology_with_data.ttl", format="turtle")

print("Merged RDF graph created and saved.")
```

✓ 1m 33.8s

Python

Merged RDF graph created and saved.

API Development

Overview

The Flask application serves as the interface for querying the RDF graph. The application is designed to expose a variety of endpoints that allow users to retrieve information about vehicles, such as brands, models, prices, and more detailed attributes like horsepower and warranty services.

Flask Application Setup

- **Initialization:** The Flask application is initialized along with basic logging to help with debugging.
- **RDF Graph Loading:** The merged RDF graph containing the ontology and data is loaded into the application to enable SPARQL querying.

Screenshot of Initializing Flask Application

```
from flask import Flask, request, jsonify
import rdflib
import logging

# Initialize Flask application
app = Flask(__name__)

# Set up logging
logging.basicConfig(level=logging.DEBUG)

# Initialize RDF graph and load the TTL file
g = rdflib.Graph()
g.parse("d:/Master/Semantic Technologies/Project/2/merged_ontology_with_data.ttl", format="turtle")
```

API Endpoints

Each endpoint corresponds to a specific query operation. Below are descriptions of several key endpoints, including the some of the SPARQL queries they were executed (**showing 5 out of 10 queries**):

1. /vehicle_brands - Retrieve Vehicle Brands

This endpoint returns a list of vehicles along with their associated brands.

SPARQL Query:

```
# Query the the ontology (SPARQL)

# 1:
@app.route('/vehicle_brands', methods=['GET'])
def vehicle_brands():
    query = """
PREFIX ex: <http://example.org/vehicle/>
SELECT ?vehicle ?brand
WHERE {
    ?vehicle a ex:Vehicle ;
            ex:brand ?brand .
}
LIMIT 10
"""

    results = g.query(query)
    output = [{"vehicle": str(row.vehicle), "brand": str(row.brand)} for row in results]
    return jsonify(output)
```

2. /models_prices - Fetch Models and Their Prices

Retrieves models along with their pricing information.

SPARQL Query:

```
# 2:
@app.route('/models_prices', methods=['GET'])
def models_prices():
    query = """
PREFIX ex: <http://example.org/vehicle/>
SELECT ?model ?price
WHERE {
    ?vehicle ex:model ?model ;
            ex:price ?price .
}
LIMIT 10
"""

    results = g.query(query)
    output = [{"model": str(row.model), "price": str(row.price)} for row in results]
    return jsonify(output)
```

3. /vehicle_hp - Vehicle Horsepower

Returns models and their engine horsepower.

SPARQL Query:

```
# 3:
@app.route('/vehicle_hp', methods=['GET'])
def vehicle_hp():
    query = """
    PREFIX ex: <http://example.org/vehicle/>
    SELECT ?model ?hp
    WHERE {
        ?vehicle ex:model ?model ;
                ex:engineHP ?hp .
    }
    LIMIT 10
    """
    results = g.query(query)
    output = [{"model": str(row.model), "horsepower": str(row.hp)} for row in results]
    return jsonify(output)
```

4. /vehicles_sorted_by_price - Vehicles Sorted by Price

Lists vehicles ordered by price.

SPARQL Query:

```
# 4:
@app.route('/vehicles_sorted_by_price', methods=['GET'])
def vehicles_sorted_by_price():
    query = """
    PREFIX ex: <http://example.org/vehicle/>
    SELECT ?vehicle ?price
    WHERE {
        ?vehicle ex:price ?price .
    }
    ORDER BY ?price
    LIMIT 10
    """
    results = g.query(query)
    output = [{"vehicle": str(row.vehicle), "price": str(row.price)} for row in results]
    return jsonify(output)
```

5. /average_price_by_brand - Average Price by Brand

Calculates the average price of vehicles by brand.

SPARQL Query:

```
# 5:
@app.route('/vehicles_by_years', methods=['GET'])
def vehicles_by_years():
    query = """
    PREFIX ex: <http://example.org/vehicle/>
    SELECT ?vehicle
    WHERE {
        ?vehicle ex:year ?year .
        FILTER(?year >= 2005 && ?year <= 2010)
    }
    LIMIT 5
    """

    results = g.query(query)
    output = [{"vehicle": str(row.vehicle)} for row in results]
    return jsonify(output)
```

Challenges Encountered

- **Performance Optimization:** Handling SPARQL queries efficiently, especially when dealing with large RDF graphs.
- **Data Accuracy:** Ensuring the queries return accurate and expected results, necessitating thorough testing and adjustments in SPARQL queries.

Results

The outputs from the API, showing the effectiveness of querying RDF data using SPARQL through a web interface.

API Operation Demonstration

Before presenting the specific outputs from the API, it's crucial to demonstrate the API in action. The screenshot below shows the command line output when the Flask application is running.

This output confirms that the API endpoints are actively responding to HTTP GET requests, returning successful status codes (HTTP 200), which indicate that the requests have been processed correctly without any errors.

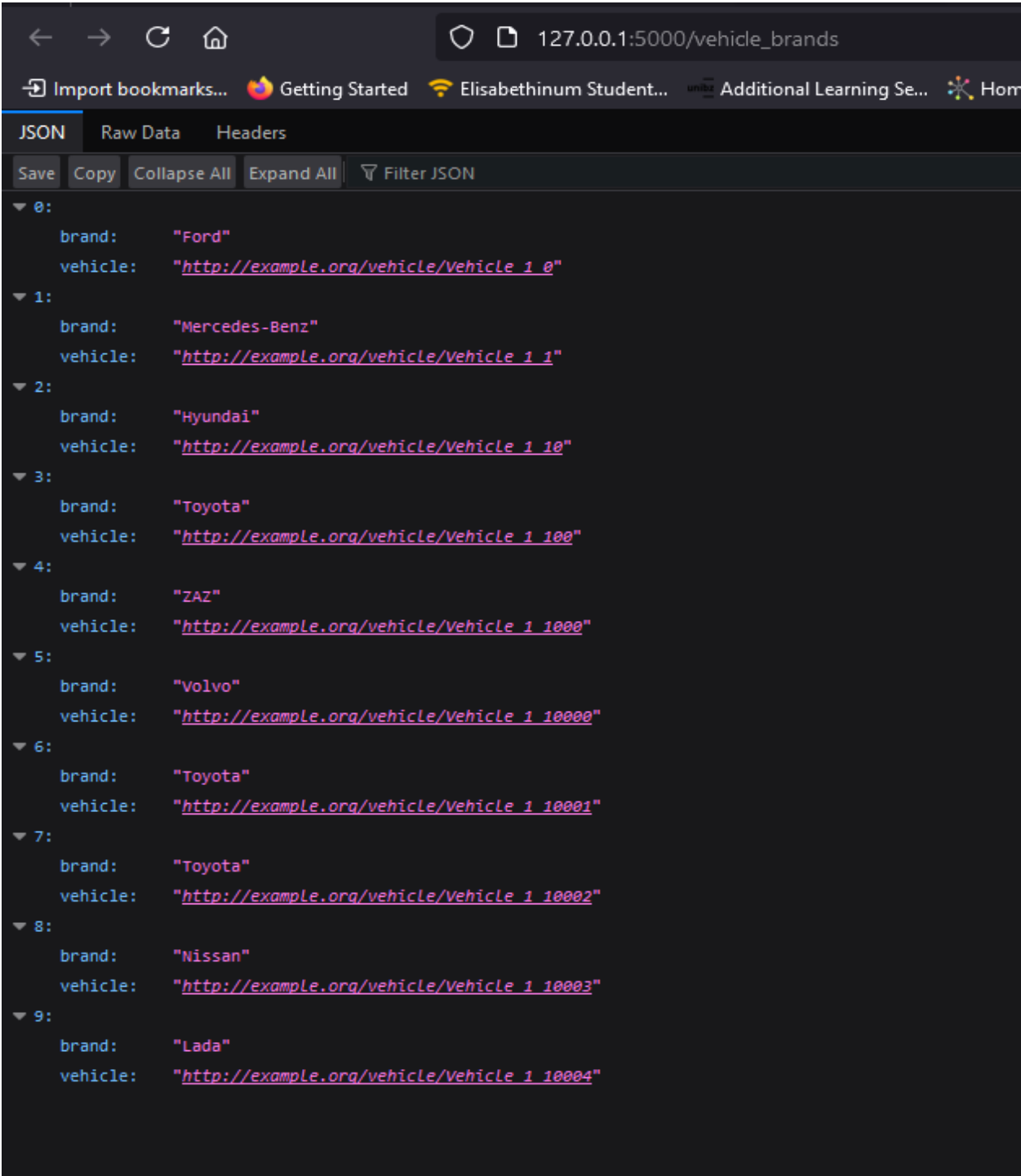
```
PS C:\Users\faisa> & C:/Users/faisa/AppData/Local/Programs/Python/Python312/python.exe "d:/Master/Semantic Technologies/Project/2/application.py"
* Serving Flask app 'application'
* Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
INFO:werkzeug:Press CTRL+C to quit
INFO:werkzeug: * Restarting with stat
WARNING:werkzeug: * Debugger is active!
INFO:werkzeug: * Debugger PIN: 672-920-172
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 20:56:31] "GET /models_prices HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 20:56:51] "GET /vehicle_brands HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 20:57:34] "GET /vehicle_hp HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 20:58:15] "GET /vehicles_sorted_by_price HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 20:59:34] "GET /vehicles_by_years HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 21:00:03] "GET /average_price_by_brand HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 21:00:34] "GET /vehicle_optional_equipment HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 21:01:07] "GET /vehicle_warranty_service HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 21:01:27] "GET /vehicle_basic_info HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 21:01:48] "GET /vehicles_by_year_and_price HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [30/Jun/2024 21:06:07] "GET /vehicle_hp HTTP/1.1" 200 -
█
```

Description of the Screenshot:

- **Flask Application Execution:**
 - The output begins with the initialization messages from Flask, noting that the application named 'application' is serving with debug mode on. It warns that this is a development server, which is not suitable for a production environment.
- **API Requests and Responses:**
 - The screenshot captures a series of API requests made to different endpoints, such as /models_prices, /vehicle_brands, /vehicle_hp, and others. Each request is logged with a timestamp, the request path, the HTTP version (1.1), and the response status (200 OK), indicating successful execution.
 - These logs demonstrate the Flask application's capability to handle multiple requests efficiently and provide quick responses to various data queries about vehicles.
- **Server Information:**
 - The server is running locally, accessible via http://127.0.0.1:5000. This local address shows that the server is intended for development and testing purposes.
 - Instructions like "Press CTRL+C to quit" suggest how to stop the server, typical in development environments.

The outputs from the API (Flask, Web Interface):

1. Vehicle Brand



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/vehicle_brands`. The browser's developer tools are open, showing the JSON response of the API call. The JSON is a list of 10 objects, each representing a vehicle brand and its corresponding vehicle URL. The browser's address bar shows the URL `127.0.0.1:5000/vehicle_brands`. The browser's tabs include "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", and "Home". The browser's developer tools are open, showing the JSON response of the API call. The JSON is a list of 10 objects, each representing a vehicle brand and its corresponding vehicle URL. The browser's address bar shows the URL `127.0.0.1:5000/vehicle_brands`. The browser's tabs include "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", and "Home".

```
JSON
Raw Data
Headers

Save Copy Collapse All Expand All Filter JSON

0:
  brand: "Ford"
  vehicle: "http://example.org/vehicle/Vehicle 1 0"
1:
  brand: "Mercedes-Benz"
  vehicle: "http://example.org/vehicle/Vehicle 1 1"
2:
  brand: "Hyundai"
  vehicle: "http://example.org/vehicle/Vehicle 1 10"
3:
  brand: "Toyota"
  vehicle: "http://example.org/vehicle/Vehicle 1 100"
4:
  brand: "ZAZ"
  vehicle: "http://example.org/vehicle/Vehicle 1 1000"
5:
  brand: "Volvo"
  vehicle: "http://example.org/vehicle/Vehicle 1 10000"
6:
  brand: "Toyota"
  vehicle: "http://example.org/vehicle/Vehicle 1 10001"
7:
  brand: "Toyota"
  vehicle: "http://example.org/vehicle/Vehicle 1 10002"
8:
  brand: "Nissan"
  vehicle: "http://example.org/vehicle/Vehicle 1 10003"
9:
  brand: "Lada"
  vehicle: "http://example.org/vehicle/Vehicle 1 10004"
```

2. Models Prices

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/models_prices`. The browser's bookmark bar includes links for "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", and "Hom". The page content is displayed in a JSON viewer interface with tabs for "JSON", "Raw Data", and "Headers". The "JSON" tab is active, showing a JSON array of 10 objects. Each object contains "model" and "price" fields. The "model" field for all objects is "Focus", and the "price" field contains various numerical values in string format. The interface includes buttons for "Save", "Copy", "Collapse All", "Expand All", and a "Filter JSON" search bar.

```
0:
  model: "Focus"
  price: "550000"
1:
  model: "Focus"
  price: "1350000"
2:
  model: "Focus"
  price: "675000"
3:
  model: "Focus"
  price: "1150000"
4:
  model: "Focus"
  price: "830000"
5:
  model: "Focus"
  price: "1481900"
6:
  model: "Focus"
  price: "580000"
7:
  model: "Focus"
  price: "1299000"
8:
  model: "Focus"
  price: "933900"
9:
  model: "Focus"
  price: "670000"
```


3. Vehicle Horsepower

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/vehicle_hp`. The browser's bookmark bar includes links for "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", "Home", and "Open". Below the browser window, a JSON viewer interface is visible, showing the response data in a collapsible tree view. The JSON is an array of 10 objects, each containing "horsepower" and "model" fields. The "horsepower" field is consistently "125" for all entries, while the "model" field varies. The viewer includes tabs for "JSON", "Raw Data", and "Headers", and buttons for "Save", "Copy", "Collapse All", "Expand All", and "Filter JSON".

```
{
  "0": {
    "horsepower": "125",
    "model": "Focus"
  },
  "1": {
    "horsepower": "125",
    "model": "Camry"
  },
  "2": {
    "horsepower": "125",
    "model": "Expert"
  },
  "3": {
    "horsepower": "125",
    "model": "Camry"
  },
  "4": {
    "horsepower": "125",
    "model": "Allion"
  },
  "5": {
    "horsepower": "125",
    "model": "Camry"
  },
  "6": {
    "horsepower": "125",
    "model": "Allion"
  },
  "7": {
    "horsepower": "125",
    "model": "Corona Exiv"
  },
  "8": {
    "horsepower": "125",
    "model": "RAV4"
  },
  "9": {
    "horsepower": "125",
    "model": "Allion"
  }
}
```

4. Vehicles Sorted By Price

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/vehicles_sorted_by_price`. Below the browser window, a REST client interface is visible, showing the JSON response of the GET request. The response is an array of 10 objects, each representing a vehicle. The 'price' field for all vehicles is '2000', and the 'vehicle' field contains a URL. The URLs are sequential, starting from `http://example.org/vehicle/Vehicle 2 10000` and ending with `http://example.org/vehicle/Vehicle 2 10061`.

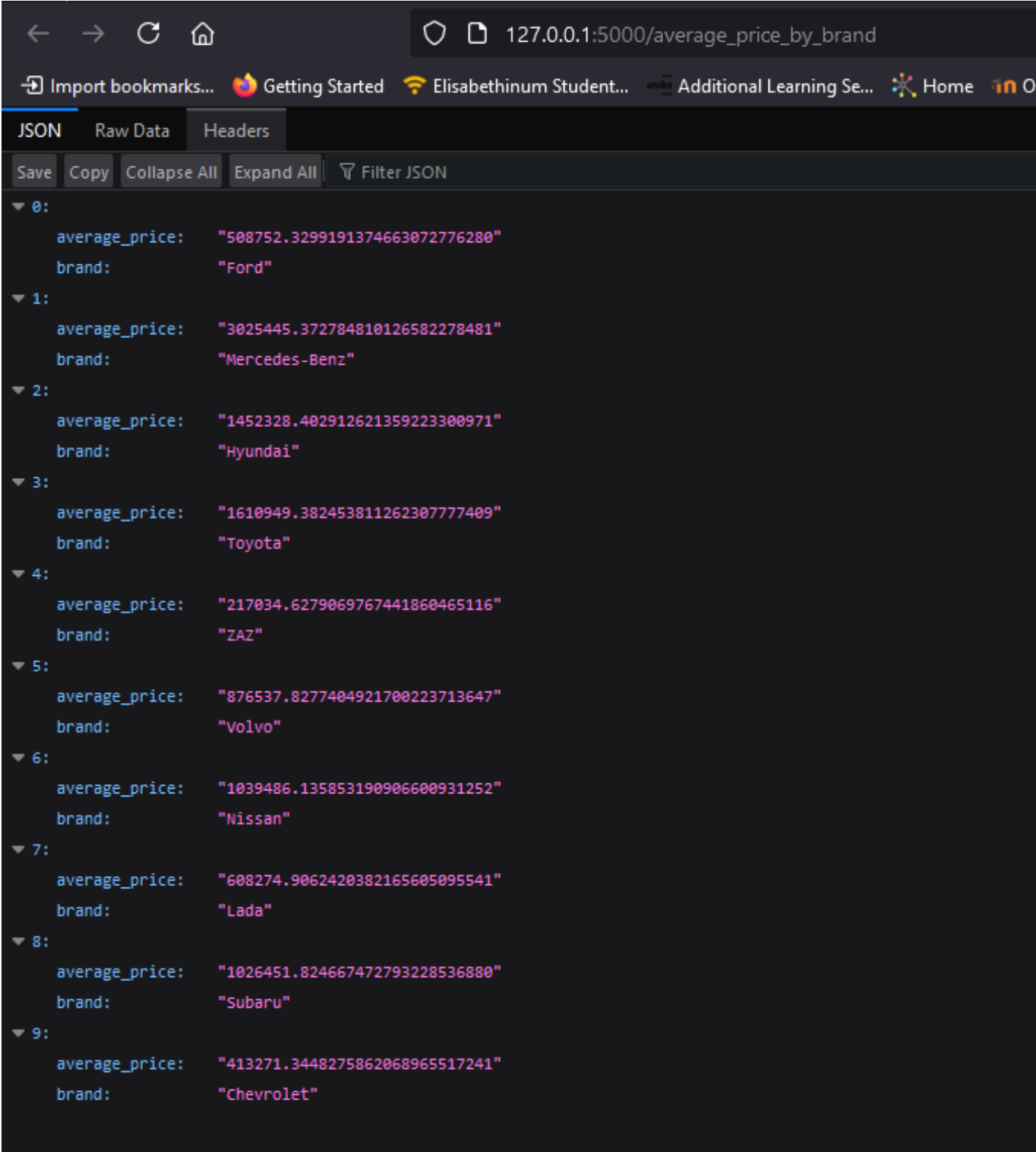
```
{
  "0": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10000"
  },
  "1": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10002"
  },
  "2": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10003"
  },
  "3": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10004"
  },
  "4": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10005"
  },
  "5": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10058"
  },
  "6": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10059"
  },
  "7": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 1006"
  },
  "8": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10060"
  },
  "9": {
    "price": "2000",
    "vehicle": "http://example.org/vehicle/Vehicle 2 10061"
  }
}
```

5. Vehicles By Years

The screenshot shows a web browser window with a REST client interface. The address bar displays the URL `127.0.0.1:5000/vehicles_by_years`. The browser's bookmark bar includes links for "Import bookmarks...", "Getting Started", "Elisabethinum Student...", and "Additional Learning Se...". The REST client interface has tabs for "JSON", "Raw Data", and "Headers", with "JSON" being the active tab. Below the tabs are buttons for "Save", "Copy", "Collapse All", "Expand All", and a "Filter JSON" input field. The JSON response is displayed in a collapsible tree structure, showing an array of five objects. Each object has a "vehicle" property with a URL value. The array indices 0 through 4 are visible on the left, each with a downward arrow to expand the object.

```
{
  0: {
    vehicle: "http://example.org/vehicle/Vehicle 2 10014"
  },
  1: {
    vehicle: "http://example.org/vehicle/Vehicle 2 10015"
  },
  2: {
    vehicle: "http://example.org/vehicle/Vehicle 2 10016"
  },
  3: {
    vehicle: "http://example.org/vehicle/Vehicle 2 10017"
  },
  4: {
    vehicle: "http://example.org/vehicle/Vehicle 2 10018"
  }
}
```

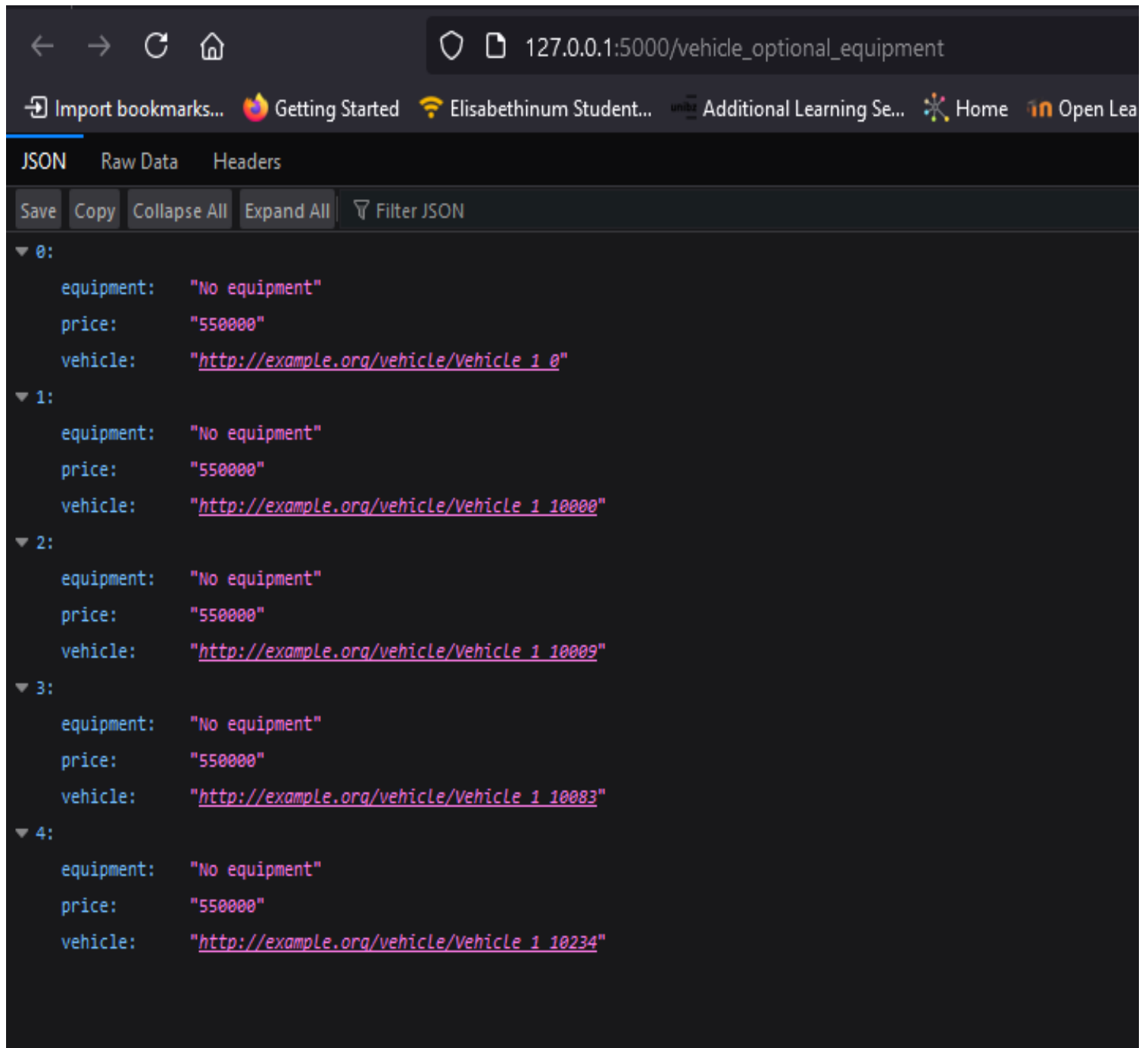
6. Average Price By Brand



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/average_price_by_brand`. The browser's tab bar includes links for "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", "Home", and a user profile icon. The main content area is a JSON viewer with tabs for "JSON", "Raw Data", and "Headers". The "JSON" tab is active, showing a list of 10 objects. Each object contains two fields: "average_price" and "brand". The data is as follows:

Index	average_price	brand
0	508752.3299191374663072776280	Ford
1	3025445.372784810126582278481	Mercedes-Benz
2	1452328.402912621359223300971	Hyundai
3	1610949.382453811262307777409	Toyota
4	217034.6279069767441860465116	GAZ
5	876537.8277404921700223713647	Volvo
6	1039486.135853190906600931252	Nissan
7	608274.9062420382165605095541	Lada
8	1026451.824667472793228536880	Subaru
9	413271.3448275862068965517241	Chevrolet

7. Vehicle Optional Equipment



The screenshot shows a web browser window with a REST client interface. The address bar displays the URL `127.0.0.1:5000/vehicle_optional_equipment`. The browser's bookmark bar includes links for "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", "Home", and "Open Lea". The REST client interface has tabs for "JSON", "Raw Data", and "Headers", with "JSON" selected. Below the tabs are buttons for "Save", "Copy", "Collapse All", "Expand All", and a "Filter JSON" input field. The JSON data is displayed as an array of five objects, each representing a vehicle with optional equipment. Each object contains three fields: "equipment", "price", and "vehicle".

```
{
  "equipment": "No equipment",
  "price": "550000",
  "vehicle": "http://example.org/vehicle/Vehicle 1 0"
},
{
  "equipment": "No equipment",
  "price": "550000",
  "vehicle": "http://example.org/vehicle/Vehicle 1 10000"
},
{
  "equipment": "No equipment",
  "price": "550000",
  "vehicle": "http://example.org/vehicle/Vehicle 1 10009"
},
{
  "equipment": "No equipment",
  "price": "550000",
  "vehicle": "http://example.org/vehicle/Vehicle 1 10083"
},
{
  "equipment": "No equipment",
  "price": "550000",
  "vehicle": "http://example.org/vehicle/Vehicle 1 10234"
}
```

8. Vehicle Warranty Service

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/vehicle_warranty_service`. The browser's bookmark bar includes links for "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", and "Home". Below the browser window, a JSON viewer displays the response data in a collapsed state. The JSON is an array of five objects, each representing a vehicle warranty record. Each record contains three fields: `last_service_date`, `vehicle`, and `warranty_period`. The `last_service_date` field is consistently "No service info". The `vehicle` field contains a URL for each vehicle, and the `warranty_period` field is consistently "No warranty info".

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
{
  "0": {
    "last_service_date": "No service info",
    "vehicle": "http://example.org/vehicle/Vehicle 1 0",
    "warranty_period": "No warranty info"
  },
  "1": {
    "last_service_date": "No service info",
    "vehicle": "http://example.org/vehicle/Vehicle 1 10370",
    "warranty_period": "No warranty info"
  },
  "2": {
    "last_service_date": "No service info",
    "vehicle": "http://example.org/vehicle/Vehicle 1 10445",
    "warranty_period": "No warranty info"
  },
  "3": {
    "last_service_date": "No service info",
    "vehicle": "http://example.org/vehicle/Vehicle 1 10470",
    "warranty_period": "No warranty info"
  },
  "4": {
    "last_service_date": "No service info",
    "vehicle": "http://example.org/vehicle/Vehicle 1 10521",
    "warranty_period": "No warranty info"
  }
}
```

9. Vehicle Basic Info

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5000/vehicle_basic_info`. The browser's bookmark bar includes links for "Import bookmarks...", "Getting Started", "Elisabethinum Student...", "Additional Learning Se...", and "Home". The main content area displays a JSON array of 10 vehicle objects, each with the following fields: `brand`, `model`, `price`, `vehicle`, and `year`. All vehicles are Ford Taurus X models from the year 2009. The prices range from 26800 to 34175. Each vehicle entry includes a unique URL in the `vehicle` field.

```
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "31620",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10272",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "28890",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10273",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "29770",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10274",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "27030",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10275",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "32600",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10276",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "30750",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10277",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "32325",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10278",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "33100",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10279",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "34175",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10280",
  "year": "2009"
},
{
  "brand": "Ford",
  "model": "Taurus X",
  "price": "31330",
  "vehicle": "http://example.org/vehicle/Vehicle_2_10281",
  "year": "2009"
}
```

10. Vehicle By Year and Price

The screenshot shows a web browser with the address bar displaying `127.0.0.1:5000/vehicles_by_year_and_price`. The browser's developer tools are open, showing the JSON response of an API call. The JSON is an array of 10 objects, each representing a vehicle. Each object contains the following fields: `brand`, `model`, `price`, `vehicle`, and `year`. All vehicles are of the brand "Ford" and model "Expedition", and all are from the year "2016". The prices vary, and each vehicle has a unique URL in the `vehicle` field.

```
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "58185",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4318",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "62025",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4314",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "62985",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4312",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "59375",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4313",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "66025",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4314",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "57795",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4315",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "60835",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4316",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "64945",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4317",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "63375",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4318",
  "year": "2016"
},
{
  "brand": "Ford",
  "model": "Expedition",
  "price": "60835",
  "vehicle": "http://example.org/vehicle/Vehicle_2_4314",
  "year": "2016"
}
```


Conclusion

This project successfully demonstrated the integration and utilization of semantic web technologies for managing and querying vehicle data. Through the development of a comprehensive vehicle ontology and the transformation of traditional data sets into RDF format, we have established a robust framework that enhances data accessibility and interoperability.

Key accomplishments include:

- **Ontology Development:** The creation of a detailed vehicle ontology that effectively represents complex relationships within vehicle data.
- **Data Integration:** Seamless conversion of CSV data into RDF format, ensuring that data is semantically enriched and ready for advanced querying.
- **API Implementation:** Development of a Flask-based API that provides flexible and efficient access to the RDF data, supporting a wide range of queries that demonstrate the power of SPARQL in real-world applications.

Repository (GitLab)

<https://gitlab.inf.unibz.it/FaisalAbdelazizFareed.Saimeh/vehicle-ontology-and-data-integration/-/tree/main>

References

<https://www.kaggle.com/datasets/CooperUnion/cardataset>

<https://www.kaggle.com/datasets/volkanastasia/dataset-of-used-cars>

<https://rdflib.readthedocs.io/en/stable/>

<https://www.w3.org/TR/sparql11-overview/>

<https://flask.palletsprojects.com/>

<https://protege.stanford.edu/>