C4 Diagram planning

# Detailing the Input module (Data Processing)

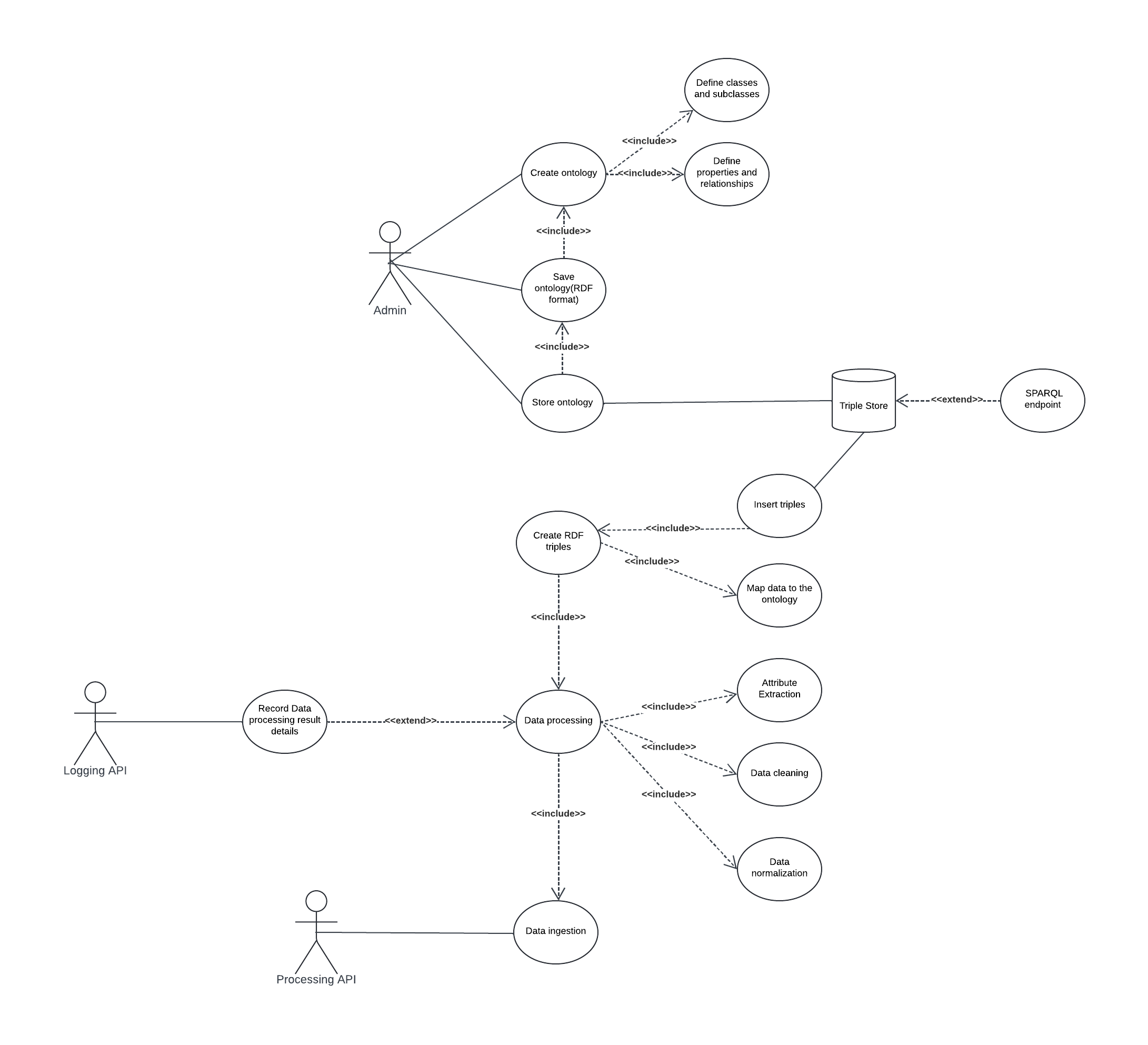
## The Use Case Diagram

Actors:

1. **Logging API**: An external API used for logging or event tracking.
2. **Processing API**: Another external API or component responsible for data processing.
3. **Admin**: A user or component with administrative privileges.

Use Cases:

1. **Record data**: The system can record migration-related data.
2. **Create RDF trips**: The system can create RDF (Resource Description Framework) triples to represent the migration data.
3. **Store migration**: The system can store the migration data.
4. **Data processing**: The system can process the migration data.
5. **Data ingestion**: The system can ingest or import data from various sources.
6. **SPARQL endpoint**: The system provides a SPARQL (SPARQL Protocol and RDF Query Language) endpoint for querying the migration data.
7. **Triple store**: The system manages and stores the RDF triples representing the migration data.



## The Data Analysis Module

**Actors**:

* **Admin**: The main user who interacts with the system.

**Use Cases**:

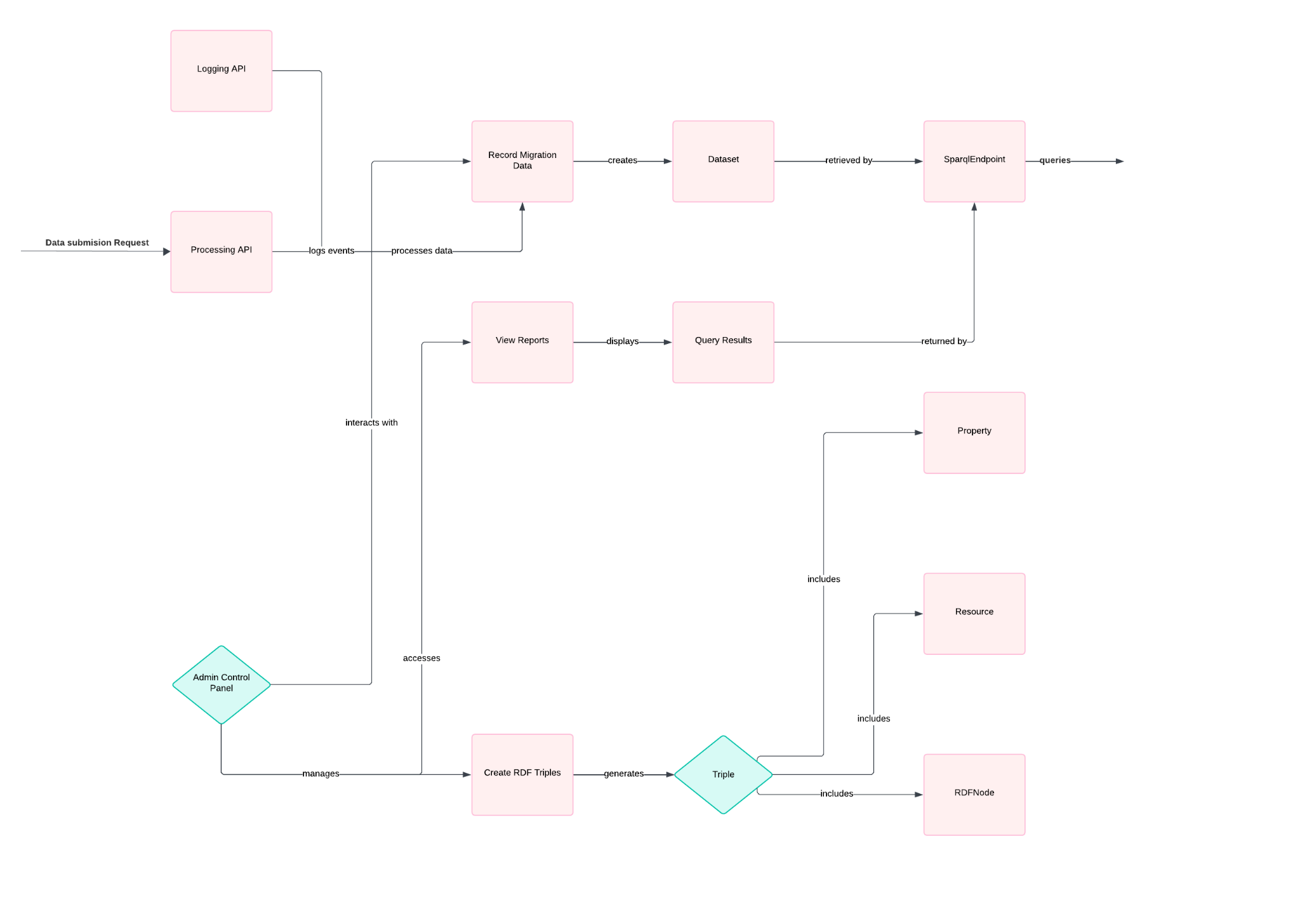
* **Record Migration Data**: The process of entering migration data into the system.
* **Create RDF Triples**: The action of generating RDF triples from the migration data.
* **View Reports**: The ability to access and display reports based on the recorded data.

**External APIs**:

* **Logging API**: Logs events related to data recording.
* **Processing API**: Handles data processing tasks.

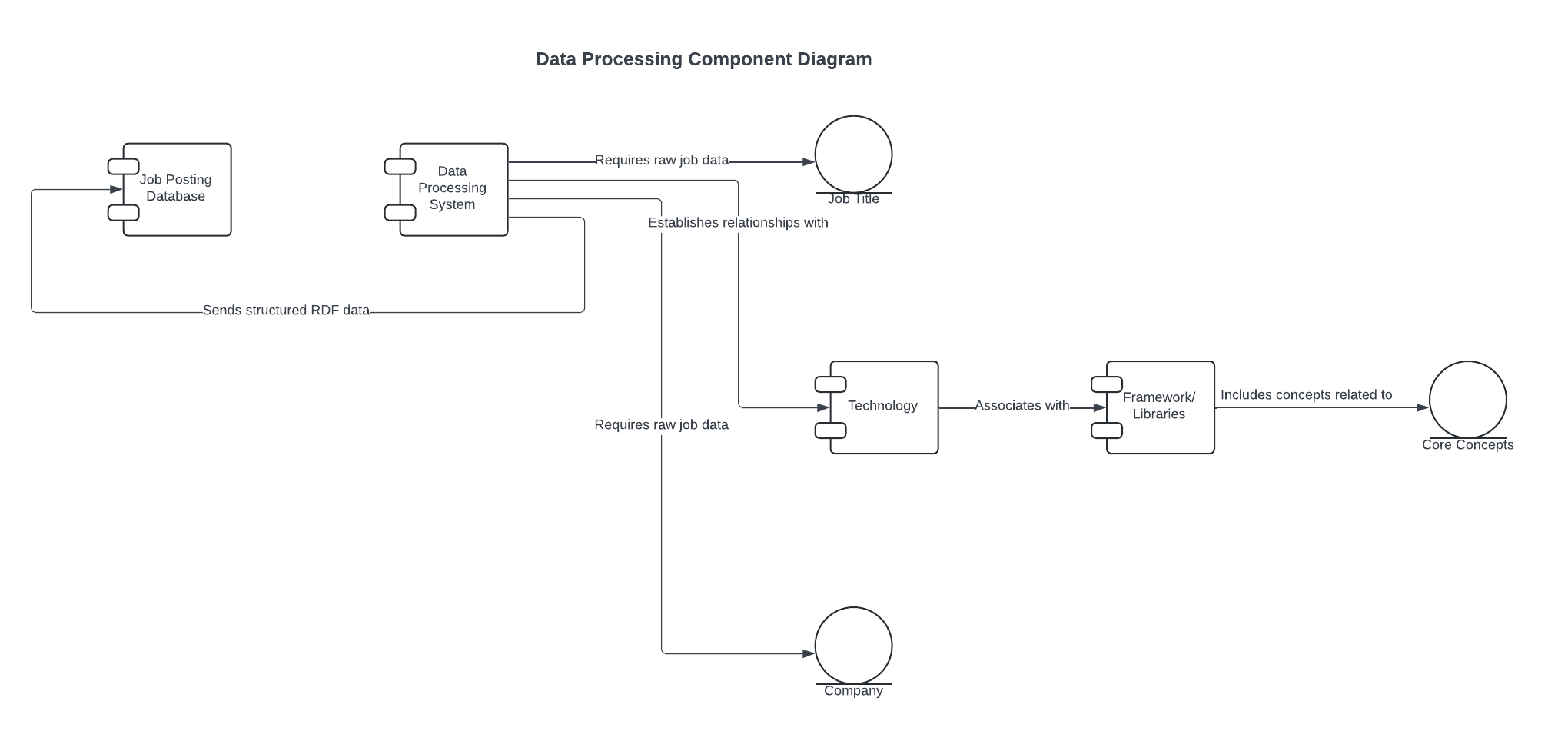
**Data Flow**:

* **Dataset**: Created when migration data is recorded.
* **Triple**: Generated from the dataset, which includes Resource, Property, and RDFNode.
* **Query Results**: Displayed as part of the reporting functionality, retrieved from the SPARQL endpoint.

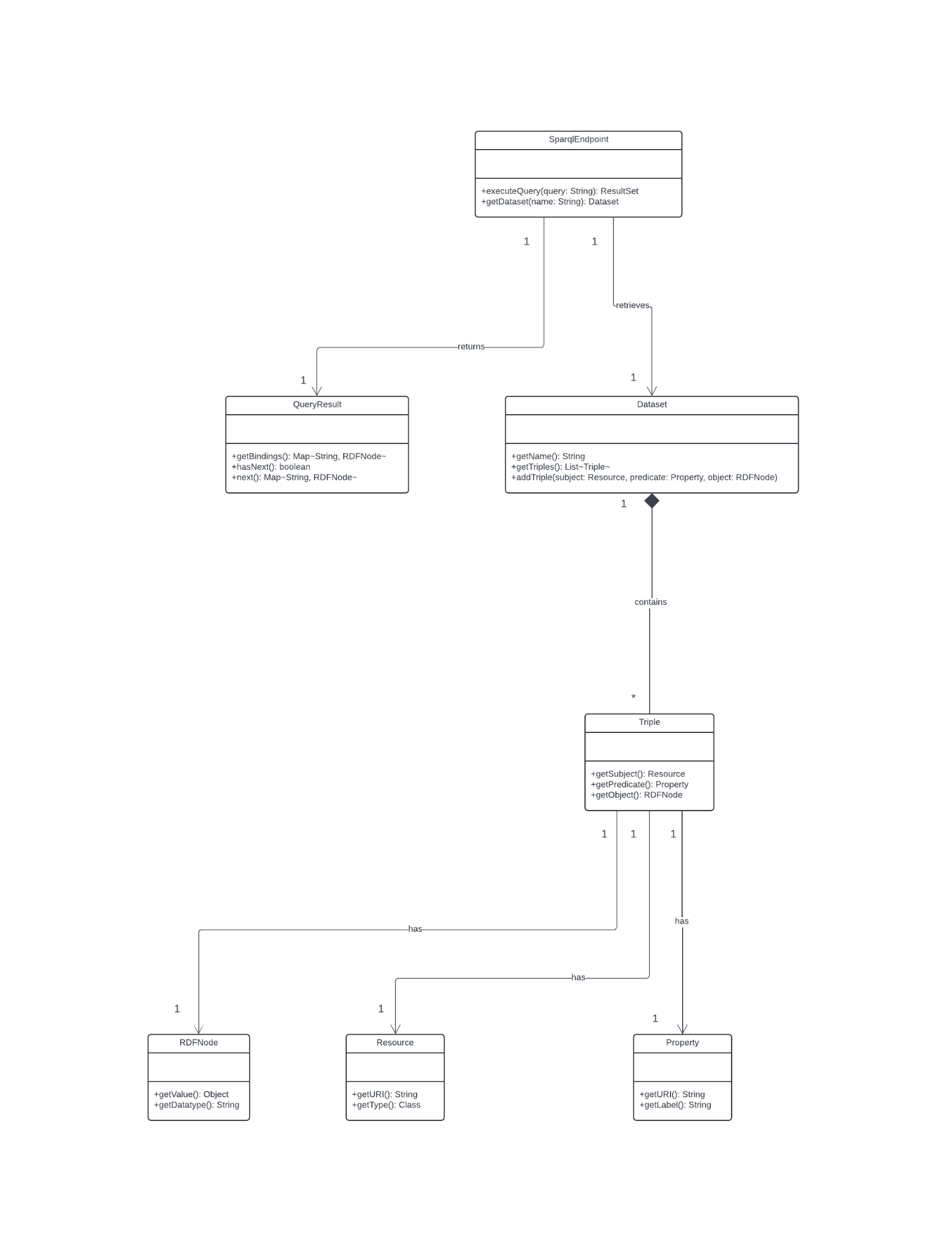


## The Data Processing System Component Diagram

The main goal of this diagram is to show how the system requires raw job data from the Company and Job Title entities, establishes relationships with Technology, associates with Framework/Libraries, includes concepts related to Core Concepts, and sends structured RDF data to the Job Posting Database.



## The Sparql EndPoint Class Diagram



### Facade design pattern

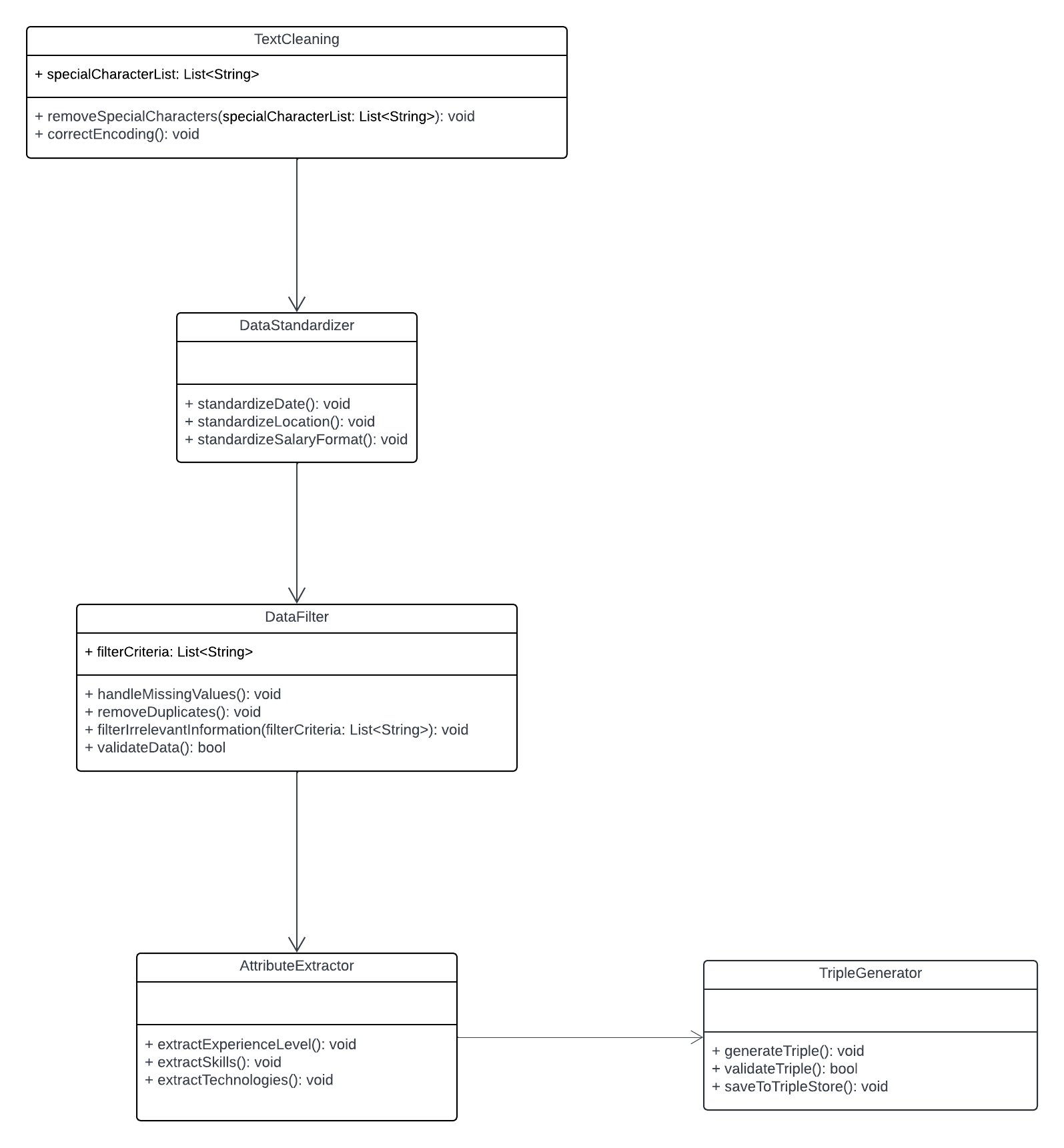
The **Facade design pattern** is a structural pattern that provides a simplified interface to a larger, more complex subsystem. In the context of the Data Processing Module, the SPARQL endpoint can be seen as a Facade that hides the complexity of the underlying RDF data management and querying.

Here's how the SPARQL-related components in the diagram fit the Facade design pattern:

1. **SPARQL Endpoint**:  
   * This component acts as the Facade, providing a simplified interface for users and other parts of the system to interact with the RDF data.
   * It encapsulates the complexity of the Triple Store and other RDF-related components.
2. **Triple Store**:  
   * This component represents the underlying subsystem that manages the RDF triples.
   * The SPARQL Endpoint Facade provides a simplified way to interact with the Triple Store, without exposing the details of its implementation.
3. **Create RDF trips**:  
   * This component is responsible for creating the RDF triples that represent the migration data.
   * The SPARQL Endpoint Facade may use this component to manage the RDF data, but the details are hidden from the users and other parts of the system.
4. **Data Ingestion**:  
   * This component is responsible for ingesting or importing data into the system.
   * The SPARQL Endpoint Facade can provide a unified interface for data ingestion, abstracting away the complexities of the underlying data processing and storage.

By using the Facade design pattern, the Data Analysis system can provide a user-friendly and efficient way to interact with the SPARQL-based RDF data, without requiring direct interaction with the complex underlying components. This promotes modularity, maintainability, and easier integration with other parts of the system.

## The Porcessing API Class Diagram



### **Pipeline Pattern**

* **Description**: The pipeline pattern is ideal for **data processing workflows** that involve a series of **stages** (or steps). Each stage performs a transformation on the data and passes the output to the next stage.
* **Use in Your Situation**:
  + Implement a **processing pipeline** where each step represents a distinct stage (e.g., RawDataLoader, DataCleaner, DataStanderdizer, AttributeExtractor, and TripleGenerator).
  + Each component processes the data in sequence, transforming it from raw data to cleaned data, then normalized data, extracted attributes, mapped ontology terms, and finally, RDF triples.
* **Advantages**:
  + Makes the processing flow modular and easy to maintain.
  + Each step can be implemented, tested, and modified independently.

### **Strategy Pattern**

* **Description**: This pattern allows you to **define a family of algorithms or processing methods**, encapsulate each one, and make them interchangeable. The algorithm used can vary depending on the context.
* **Use in Your Situation**:
  + For data cleaning, normalization, or attribute extraction, different strategies may be needed depending on the **data source** or **format**. For example, cleaning data from job boards might require different techniques than cleaning data from company websites.
  + You can define a common interface for data cleaning strategies and implement different strategies for different data sources.
* **Advantages**:
  + Enables the dynamic selection of processing methods.
  + Makes it easier to extend the system with new processing strategies.

Data Processing Diagram

