Pokenator: a Semantic Web Guessing Game for Pokémon

Rávilon Aguiar dos Santos[1], Gustavo Pìnzon Pereira[2]

[1]Universidade Federal de Pelotas (UFPEL), Brazil
[2]Universidade Federal de Pelotas (UFPEL), Brazil

Abstract

The Semantic Web envisions a Web of data that is machine-interpretable, enabling applications to perform reasoning, integration and discovery on heterogeneous sources. While powerful, its concepts can be difficult to grasp for learners unfamiliar with RDF, OWL and SPARQL. In this article we present *Pokenator*, an interactive guessing game that applies Semantic Web technologies to a playful domain – the Pokémon universe – to serve as a didactic case study. Inspired by the popular Akinator game, Pokenator asks a player a series of yes/no questions about a hidden Pokémon and narrows down the candidates using SPARQL queries over an ontology-based knowledge graph. We describe the data sources used, including the Pokémon Ontology [1] and additional metadata from PokéAPI, outline the architecture of our system and its reasoning heuristics, and discuss how such an application can support teaching of Semantic Web concepts.

Keywords

Semantic Web, Ontology, Knowledge Graph, SPARQL, Pokémon, Akinator

1. Introduction

The Semantic Web provides a framework for publishing structured data on the Web so that it can be queried and reasoned over by machines. RDF graphs capture facts as triples of subject–predicate–object; ontologies define classes and properties to organise those facts; and the SPARQL query language enables expressive pattern matching over graphs. As the W3C specification explains, SPARQL can express queries across diverse RDF datasets and supports required and optional graph patterns as well as value constraints [2]. Despite its maturity, the Semantic Web remains abstract to many learners. We sought to create a hands-on demonstration that would anchor Semantic Web concepts in a familiar domain and provide intuitive feedback.

The classic online game *Akinator* invites players to think of a real or fictional character; the system then poses a sequence of questions and attempts to guess the character [3]. Our project *Pokenator* adapts this idea to the Pokémon universe, using Semantic Web technologies to drive the reasoning process. Pokémon are widely recognised by students and provide rich, structured attributes (types, abilities, habitats, generations) that lend themselves to ontological modelling. The goal of Pokenator is both to entertain and to expose learners to RDF, OWL and SPARQL in a didactic setting.

2. Semantic Web and ontologies

2.1. RDF and SPARQL

Resource Description Framework (RDF) represents information as directed, labelled graphs of triples; each triple comprises a subject, predicate and object. SPARQL is the standard query language for RDF graphs and allows users to specify graph patterns that must match the data[2]. The specification emphasises that SPARQL can query data stored natively as RDF or viewed as RDF via middleware, and supports optional patterns, conjunctions and disjunctions as well as result sets or graph constructions[2]. These features underpin Pokenator's ability to filter candidate Pokémon based on user responses and to construct explanations of its guesses.

CEUR-WS Workshop on Semantic Web Applications,
February 2026, Porto Alegre, Brazil

2.2. Ontologies

Ontologies provide the vocabulary and semantics for a domain. In Pokenator we use the Pokémon Ontology (PokémonKG Ontology) developed by Kevin Haller [1]. This ontology defines classes such as `pok:Pokemon`, `pok:Type`, `pok:Ability`, `pok:Habitat` and `pok:Generation` as well as properties including `pok:hasType`, `pok:hasAbility` and `pok:fromGeneration`. It was released on 27 July 2019 and the current version (v1.0.0) is publicly available [1]. Ontologies enable reasoning: for example, if a Pokémon has two types Fire and Flying, the ontology can infer that it is a dual-type Pokémon.

3. Data sources

3.1. PokémonKG and the Pokémon Ontology

The Pokémon Knowledge Graph (PokemonKG) aggregates information from community-driven sources such as Bulbagarden, PokeWiki, PokéAPI and PokemonDB. The knowledge graph contains approximately 129 834 triples and 18 568 entities, and is published under an open licence with a SPARQL endpoint. We use the Pokémon Ontology as the vocabulary for our knowledge graph and load the triples into a triplestore (Apache Jena Fuseki). A reasoner with RDFS/OWL capabilities enables class subsumption and property inheritance.

3.2. PokéAPI

While the Pokémon Ontology provides a semantic schema, not all attributes are available as RDF. We complement the ontology with detailed instance data from the PokéAPI, an open RESTful service that exposes comprehensive Pokémon data. The API advertises itself as "all the Pokémon data you'll ever need in one place," serving over 10 billion calls per month and offering a modern free and open-source interface [4]. It includes endpoints for Pokémon species, types, abilities and moves. We integrate selected JSON responses into our knowledge graph, mapping them to ontology classes and properties. Because PokéAPI is a constantly evolving community project [4], we snapshot the data relevant to our knowledge graph to ensure reproducibility.

3.3. Inspirational baseline: Akinator

Our design is inspired by the Akinator game developed by Elokence. On the official website the genie invites the player to "think about a real or fictional character" and promises to guess it [3]. We adopt a similar interactive flow: Pokenator prompts the player to imagine a Pokémon and then asks binary questions (e.g., "Is it of type Fire?", "Does it come from the first generation?"). Each answer translates into a SPARQL filter that reduces the candidate set. A heuristic chooses the next question to maximise information gain, akin to a decision tree. Once a single candidate remains, the system presents its guess together with an explanation of the triples that support the inference.

3.4. Project repository

The full source code for Pokenator, including the back-end, front-end and data preparation scripts, is publicly available on GitHub at https://github.com/ravilon/Pokenator. The repository hosts the Java implementation of the reasoning engine (built on Apache Jena) together with a Spring Boot back-end that exposes the game API. The front end is implemented with React and Vite and uses Tailwind CSS for styling. All code is open source under a permissive licence. We encourage readers to explore the project and contribute improvements.

4. System architecture

Figure 1 illustrates the high-level architecture of Pokenator. The front-end, built with React and the Vite bundler and styled with Tailwind CSS, provides the user interface for posing questions and showing guesses. The back-end is a Spring Boot application written in Java that converts user responses into SPARQL queries and interfaces with the triplestore. Within the back-end, a reasoning module implements heuristics to select the next question: it analyses the remaining candidates and chooses a property whose values are most balanced across the candidates. The knowledge graph resides in an Apache Jena Fuseki server and stores the ontology and instance data. Optionally, results can be enriched with data retrieved on demand from PokéAPI.

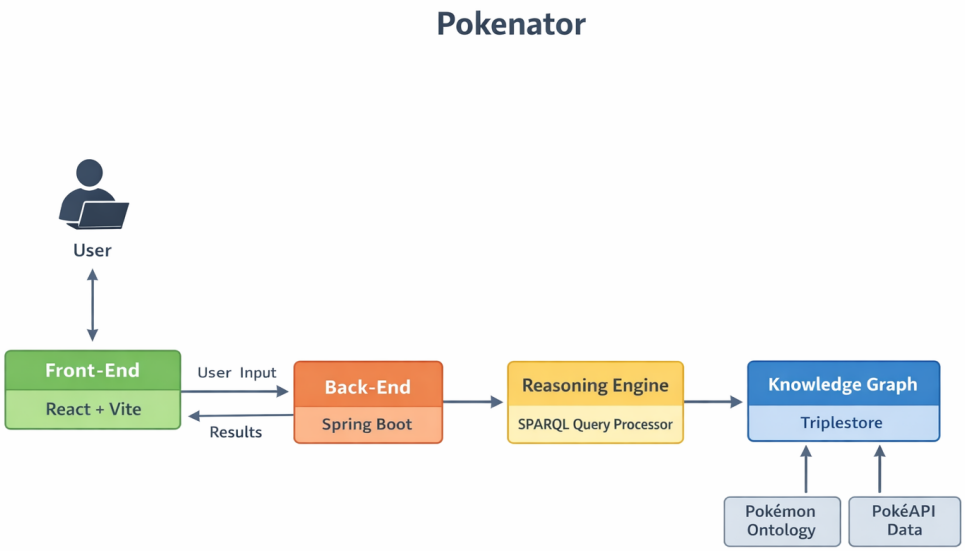

Figure 1: Overview of the Pokenator architecture. User responses from the web interface are translated into SPARQL queries by the reasoning engine. Queries are executed over a triplestore populated with the Pokémon Ontology and instance data; results are used to update the candidate set and determine the next question.

5. Implementation

5.1. Data preparation

We first loaded the Pokémon Ontology into Apache Jena Fuseki. We then used a Java utility built on top of Jena to fetch JSON data from PokéAPI and convert relevant fields (type, ability, habitat, generation and base stats) into RDF triples consistent with the ontology. Each Pokémon instance was assigned a persistent IRI (e.g., `poke:Pikachu`). Where attributes were missing in the ontology, we extended it by adding classes and properties in a separate module to avoid altering the original specification.

5.2. Heuristic question selection and search engine

The reasoning engine drives the guessing game by maintaining a **candidate set** of all Pokémon consistent with the player's answers. This set is stored in memory as a list of IRIs initially containing every Pokémon in the knowledge graph. Each time the player answers a yes/no question, the engine issues a corresponding SPARQL query to filter the candidates. For example, if the player responds "yes"

to “Is it of type Fire?”, the engine sends a query that requires the variable `?p` to have `pok:hasType` equal to `pok:Fire`. A “no” response results in a query that excludes Fire types. The answers are combined conjunctively to refine the candidate set after each step.

To decide which question to ask next, the engine enumerates candidate questions based on a small set of predicate families. These include direct properties such as `pok:hasType`, `pok:hasColour`, `pok:hasShape` and `pok:foundIn` as well as the generation, which is reached via the inverse predicate `pok:featuresSpecies` between generation resources and species. For each predicate family, the engine queries the triplestore to count how many of the remaining candidates have each possible value. Values already constrained by previous answers or already asked in the current session are skipped. Only values whose counts are strictly between 0 and the total number of remaining candidates are considered, as they can partition the set.

For each potential question, the engine records the number of candidates that would answer “yes” (i.e., those possessing the value) and “no”. It then sorts the candidate questions by how close this yes count is to half of the remaining candidates, because a balanced split yields maximal information. Among the top few questions with the best split, one is selected at random to avoid deterministic behaviour. Once selected, the question is marked as asked in the session so that it will not be repeated.

If the candidate set becomes small (two elements or fewer) or the question selector cannot find any suitable predicate–value pair, the engine switches to guess mode. It asks, “Is it [Pokémon]?” for one of the remaining species and, if necessary, offers the other candidate if the user responds negatively. These heuristics ensure that the game converges quickly while preserving a degree of variability across sessions. The back-end is implemented in Java using Apache Jena for SPARQL queries and exposed as a Spring Boot service.

5.3. Dynamic query construction and complexity

Every time the player answers a question, Pokenator updates its candidate set by constructing a fresh SPARQL query. The query begins with a basic pattern that selects all Pokémon instances (‘?p‘) of class `pok:Pokemon`. For each affirmative answer, we append an additional triple pattern of the form ‘?p <predicate> <value>‘ to require the value; for negative answers we add a `FILTER NOT EXISTS` clause to exclude that triple. Because SPARQL queries are expressed in text, the server assembles these clauses dynamically by concatenating template strings. The resulting query looks like this (simplified):

```
SELECT ?p WHERE {
  ?p a pok:Pokemon .
  # positive constraints
  ?p pok:hasType pok:Fire .
  ?p pok:fromGeneration pok:Generation_V .
  # negative constraints
  FILTER NOT EXISTS { ?p pok:hasAbility pok:Levitate . }
}
```

The engine executes this query against the triplestore to obtain the list of remaining candidates. Since each constraint reduces the search space, the cost of evaluating the query typically decreases over time. In the worst case, however, the query contains a conjunction of m positive and negative patterns and runs against a knowledge graph with n Pokémon. Query evaluation in SPARQL is NP-complete in general, but with our simple acyclic patterns the effective complexity is roughly $O(n \times m)$ as the store checks each candidate against each constraint.

When selecting the next question, the engine issues aggregation queries of the form ‘SELECT <value> (COUNT(DISTINCT ?p) AS ?count) WHERE ... GROUP BY <value>‘ to count how many of the current candidates exhibit each value for a given predicate family. Because the number of predicate families is small (about five) and each aggregation query examines only the reduced candidate set, the overhead remains manageable even on a graph with tens of thousands of triples. Randomisation among top

candidates further mitigates the risk of deterministic behaviour while still favouring questions that maximise information gain.

5.4. Explanation generation

To make the reasoning process transparent, Pokenator returns an explanation for its guess. The SPARQL query that led to the final candidate is retained, and the triples that match the query (types, abilities, generation) are presented to the player. For example, if the guessed Pokémon is *Charizard*, the system might list that it is a Fire/Flying type and originates from Generation I. Because the ontology includes RDFS/OWL semantics, explanations may include inferred facts (e.g., belonging to the class *DualType*).

6. User interface and gameplay

The Pokenator web application presents a simple and intuitive interface for players. The front end asks a series of yes/no questions about the hidden Pokémon and displays the current set of possible candidates. As the player responds, the list of candidates narrows in real time. When a unique candidate remains, the system reveals its guess and explains the reasoning by listing relevant attributes. This interactive flow helps users understand how the underlying knowledge graph and reasoning heuristics operate.

Figure 2 summarises the interaction flow. The player first thinks of a hidden Pokémon. Pokenator then repeatedly asks yes/no questions; after each answer the reasoning engine updates its candidate set via a SPARQL query and decides which question to ask next. When only one candidate remains, the system makes a guess and explains its reasoning.

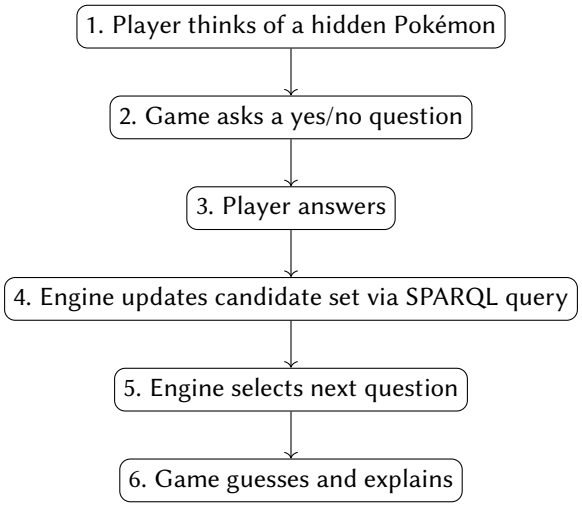

Figure 2: Gameplay flow of the Pokenator. Starting from a hidden Pokémon chosen by the player, the game asks binary questions, updates its candidate set using SPARQL queries and a reasoning engine, selects new questions based on heuristics, and finally guesses with an explanation.

Figure

7. Evaluation and discussion

Our primary objective was to create a demonstrator for teaching Semantic Web concepts rather than to compete with the efficiency of general guessing algorithms. Nevertheless, we evaluated Pokenator on selected Pokémon and found that the system typically converges to a unique candidate within 5–7 questions when the ontology contains sufficient discriminating attributes. In cases where attributes are missing or ambiguous (e.g., Pokémon that share types and abilities), the system may require more questions or cannot produce a unique guess.

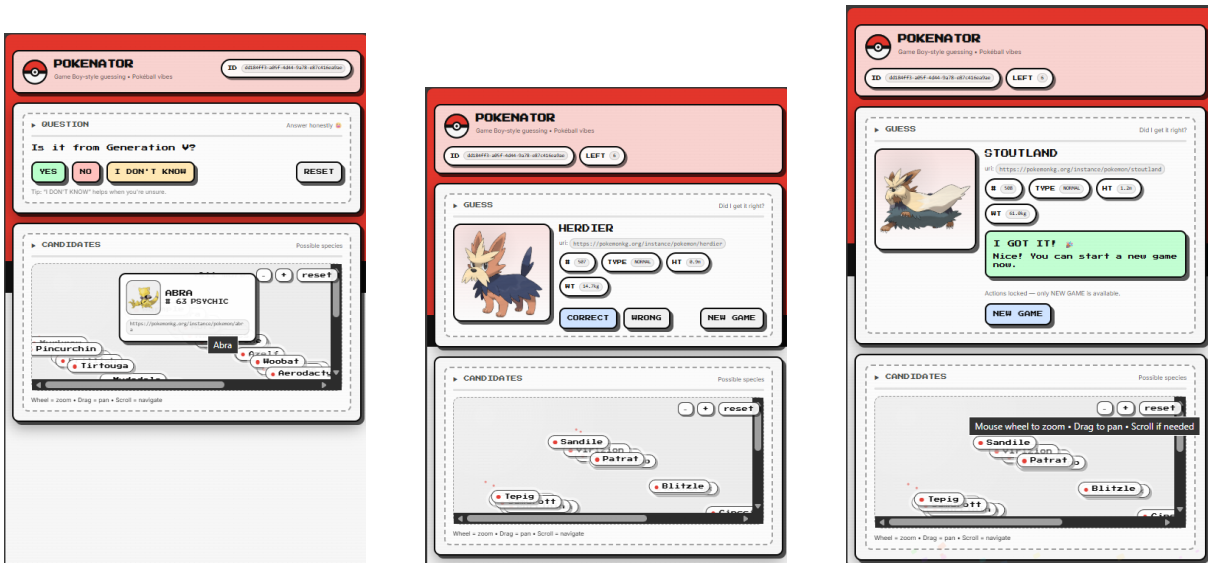

Figure 3: Screenshots of the Pokenator user interface. Left: the system asks if the Pokémon is from Generation V. Centre: the remaining candidate list after answering several questions. Right: the final guess with an explanation of inferred attributes.

The integration of PokéAPI data improved coverage, particularly for abilities and move lists. However, the heterogeneity of data sources requires careful mapping to the ontology. Future work could explore linking Pokenator to additional datasets (such as Pokémon locations or evolution chains) and implementing more sophisticated reasoning rules.

8. Conclusion

Pokenator demonstrates how Semantic Web technologies can underpin an engaging educational application. By combining an ontology-driven knowledge graph with SPARQL reasoning and a simple heuristic, the system invites users to explore structured Pokémon data while encountering key concepts such as RDF triples, ontologies, queries and inference. The project also showcases the richness of openly available resources like the Pokémon Ontology and PokéAPI[1, 4]. We hope that Pokenator will inspire learners to experiment with Semantic Web tools and to consider how knowledge representation and reasoning can support interactive web applications.

References

[1] K. Haller, Pokémon ontology, https://pokemonkg.org/ontology/version/1.0.0, 2019. Revision: v1.0.0.
[2] W. W. W. Consortium, Sparql 1.1 query language, https://www.w3.org/TR/sparql11-query/, 2013. W3C Recommendation.
[3] Elokence, Akinator: The web genie, https://en.akinator.com/, 2007. Online guessing game that inspired this work.
[4] P. Hallett, P. contributors, Pokéapi: The restful pokémon api, https://pokeapi.co/, 2026. Accessed 22 Feb 2026.