Developing an ontology-based system for retrieving and contextualizing petroleum production data

Fabrício H. Rodrigues[1,*], Marcos T. Leipnitz[1], Haroldo R. S. Silva[1], Rafael H. Petry[1], Nicolau O. Santos[1], Régis K. Romeu[1], Mara Abel[1] and João Netto[1]

[1]*Informatics Institute, Federal University of Rio Grande do Sul (INF-UFRGS). Av. Bento Gonçalves, 9090 - Agronomia, Porto Alegre - RS, 91540-000, Brazil.*

Abstract

Efficient access to relevant information is crucial for various data-driven operations within organizations. However, this process often involves significant time investment and the need for often manual and non-reusable mappings to utilize the available data properly. Ontologies offer a solution by enabling upfront integration and contextualization of data, facilitating the access to integrated and semantically enriched information. In this paper, we report on the process of developing an ontology-based system for the domain of petroleum production. We present its conceptual structure, architecture, and implementation, discussing some of the decisions and trade-offs imposed by the practical setting of the application.

Keywords

ontology, ontology-based systems, knowledge-based systems, data contextualization, petroleum production, oil and gas industry

1. Introduction

Much like any other large and complex application domain, the oil and gas industry relies on many systems to manage and serve data effectively. From cataloging parts to organizing sensor series and present measurements, data systems serve from the most time-critical operational activities to research and analytic tasks. In particular, production surveillance requires many service companies to collect data on distinct parts and functions of the plant installation. Many systems are built to store and process these data without considering the original context and meaning or how it relates to other elements within the physical system. The delivered data demands a large effort in real-time human processing to become a uniform, indexed data collection organized to support production decisions, besides requiring highly specialized users to select data from the right source and combine them with correct semantics. To address this issue, engineers often assign semantics to data to provide context and facilitate retrieval. Whether categorizing data by geographical area, production plant, or administrative division, these semantic approaches aid humans in finding the information they need. However, when it comes to automated systems searching for data, a clear and definite method of organizing this data is essential. This is where ontologies come into play.

Ontologies serve as crucial tools in information system development, offering standardized and semantically rich vocabularies for knowledge representation. Integrating ontology into system development enhances data consistency, interoperability, and efficient knowledge sharing and retrieval

Proceedings of the Joint Ontology Workshops (JOWO) - Episode X: The Tukker Zomer of Ontology, and satellite events co-located with the 14th International Conference on Formal Ontology in Information Systems (FOIS 2024), July 15-19, 2024, Enschede, The Netherlands.

*Corresponding author.

fabricio.rodrigues@inf.ufrgs.br (F. H. Rodrigues); mtleipnitz@inf.ufrgs.br (M. T. Leipnitz); hrssilva@inf.ufrgs.br (H. R. S. Silva); rhpetry&inf.ufrgs.br (R. H. Petry); nicolau.santos@inf.ufrgs.br (N. O. Santos); regis.romeu@ufrgs.br (R. K. Romeu); marabel@inf.ufrgs.br (M. Abel); netto@inf.ufrgs.br (J. Netto)
https://www.inf.ufrgs.br/~rhpetry/ (R. H. Petry); https://www.inf.ufrgs.br/site/docente/mara-abel/ (M. Abel); https://www.inf.ufrgs.br/site/docente/joao-cesar-netto/ (J. Netto)
0000-0002-0615-8306 (F. H. Rodrigues); 0000-0003-1959-1755 (M. T. Leipnitz); 0009-0003-0594-4852 (H. R. S. Silva); 0000-0001-6023-0826 (R. H. Petry); 0000-0003-0901-2465 (N. O. Santos); 0000-0001-7765-086X (R. K. Romeu); 0000-0002-9589-2616 (M. Abel); 0000-0002-5350-1728 (J. Netto)

within a defined domain. Utilizing an existing domain ontology – in our case about offshore petroleum production equipment – can facilitate data retrieval in information systems. Such systems enable stakeholders to access relevant data efficiently while ensuring consistency across different applications within cross functional domains of expertise.

In this paper, we report on the process of creating, from the ground up, an ontology-based system for the domain of petroleum production. We describe aspects of its conceptual structure, architecture, and implementation, discussing some of the decisions and trade-offs imposed by the real-world nature of the application.

The remainder of the paper is structured as follows: section 2 provides some context about how data is used in the domain; section 3 exposes how we conceived a proposed semantic infrastructure to meet the needs discussed in section 2; section 4 describes how the proposed semantic structure was effectively implemented; section 5 place our work in the context of related ones; finally, section 6 brings our concluding remarks.

2. Semantic Requirements

Recognizing the various aspects of the domain is the first step in creating a semantic definition capable of representing the data and assets involved in ontology-based systems. Thus, we need to understand how users interact with their daily tools.

During this discussion, we approach the decisions while considering the perspective of a petroleum engineer, whose time is mostly consumed by tasks of data search and conversion [1]. These engineers rely on tagged data collected from digital signals and measurement instruments, which record various asset properties like pressures, temperatures, and flow rates. This data is stored in historian systems, which are databases built for efficient storage of time-based information and based on the retrieval of data through a unique identifier, its tag. The engineers frequently need to access specific data sets related to the wells' properties or connected facilities to perform tasks such as forecasting a well's oil production or identifying anomalies. However, locating and interpreting these tags is susceptible to errors due to the lack of semantic context of the tags, making it challenging to understand and utilize the data correctly.

In this context, the purpose of a semantic layer is to ease this process by incrementing a system with expert knowledge. Figure 2 shows a prototypical example of an interaction the application is intended to support. There, we have a user who wants a full description of the profile of a given petroleum well, consisting of the values for all the properties of all its components. From this scenario, we can identify some requirements that an ontology-based application for the petroleum production domain should fulfill.

Users have their own domain of expertise. Petroleum production is a multidisciplinary area involving professionals from a wide range of domains, such as reservoir engineers, production engineers, and maintenance engineers. As is usual in highly technical fields, these professionals rely on considerable amounts of data to carry out their duties. Still, data is just a tool for their work, not their object of inquiry – they are experts in *petroleum*, not in *data*. Therefore, demanding from them an accurate knowledge of where and how the data is stored is not reasonable. Instead, they should be able to rely on their domain knowledge to get the data they need. Given that, the application should shield professionals from data management details by offering means to search data that reflect the semantics of the domain.

Pieces of data are useless in isolation. Typically, users do not access a system by chance simply to see what they can find. Instead, they resort to an information system to solve an issue or carry out some task. In petroleum production, this involves checking a single piece of data and analyzing it with respect to several other data items that characterize the broader context of the issue at hand. An example would be monitoring the behavior of a plant by checking a set of alarms designed to indicate the occurrence of

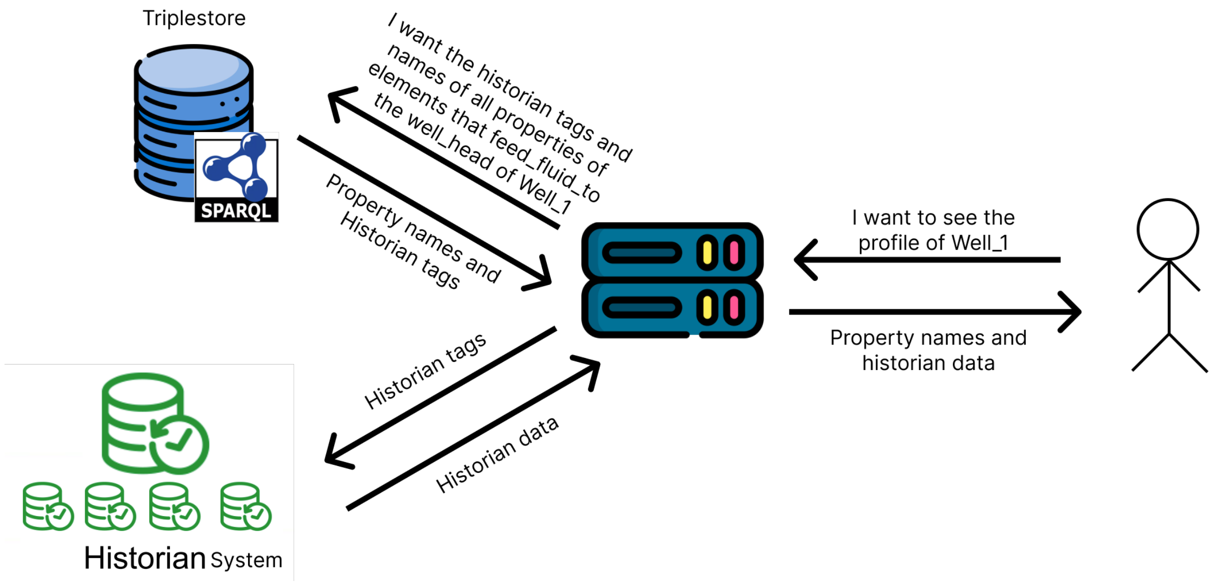
Triplestore
SPARQL
I want the historian tags and names of all properties of elements that feed_fluid_to the well_head of Well_1
Property names and Historian tags
I want to see the profile of Well_1
Property names and historian data
Historian tags
Historian data
Historian System

Figure 1: Example of system interactions.

anomalies. Usually, those alarms are rather simple, triggering when the value of some variable (*e.g.*, the pressure of some pipeline section) crosses a given threshold. In this case, the task is not simply verifying whether the variable meets the alarm threshold. Instead, it would require analyzing a diverse set of equipment properties to determine whether the alarm went off due to a genuine anomalous behavior or if it is a false-positive. Therefore, an application for this domain should focus its semantic capabilities on gathering meaningful packages of contextualized information, releasing the user from the burden of having to figure out what pieces of information must be gathered to support a given activity.

Sometimes, we need to figure out the question itself. Finally, as with any complex and dynamic field, the problems that arise in petroleum production are not always well-structured or predictable. Sometimes, part of the job is first finding out the precise nature of the problem and determining what is needed to solve it. In such cases, having a fixed set of predetermined queries is insufficient. With that, the application should offer means for the user to explore the data freely without being tied to any particular workflow.

Based on these requirements, the next section describes the semantic infrastructure we conceived to address this scenario. Also, we discuss some decisions we took during the process and some of the considered options to conform the solution to the constraints imposed by the practical setting.

3. Semantic Infrastructure

In the context of a database storing historical property measurements, the concept of time is paramount. However, integrating such temporal notions into the ontology can introduce a significant complexity. To address this challenge, our approach focused on taking all time series data provided by sensor measurements in the plant, each represented by a unique identifier, and instantiating this identifier in the knowledge base as an Information Content Entity (ICE). For those ICEs, we set an annotation property with the identifier value, so that from an atemporal vision of the production plant (present time), one can find the data that describes all readings of a property of an element. The relation between the ontology, the physical entities, and the necessary infrastructure that will be explained in this section is illustrated by Figure 2.

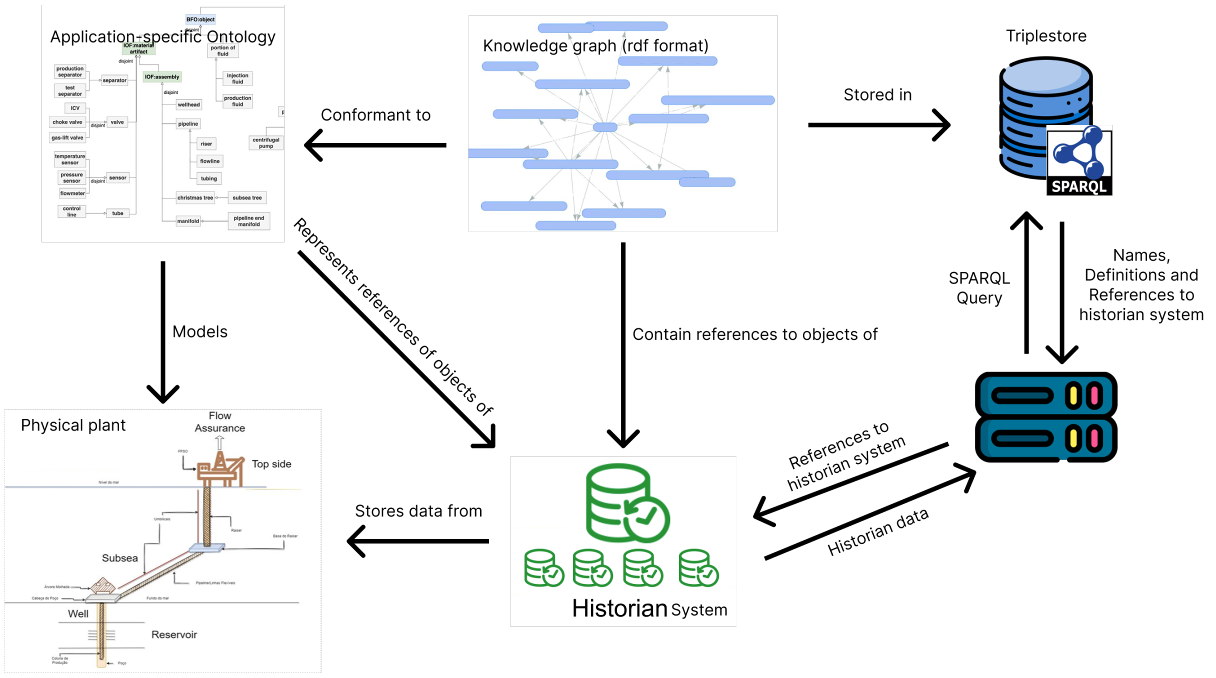
Application-specific Ontology
Knowledge graph (rdf format)
Triplestore
Conformant to
Stored in
SPARQL
Models
Represents references of objects of
Contain references to objects of
SPARQL Query
Names, Definitions and References to historian system
Physical plant
Flow Assurance
Top side
Subsea
Well
Reservoir
Stores data from
References to historian system
Historian data
Historian System

Figure 2: Schematic of the application main components

3.1. Employed ontologies

Our solution needed a domain ontology to tie sensor data from the oil plant to the actual pieces of equipment. We used the Offshore Petroleum Production Plant Ontology (O3PO) [2], which provides the formal vocabulary for this application. O3PO uses BFO as a top-level ontology and also uses IOF-Core [3] and GeoCore [4] as middle-level ontologies.

The ontology besides formally defining the main entities in an oil plant, also enables accessing the properties of equipment, the components of the facilities, and following the oil path from the reservoir to the platform through fluid connections defined as object properties.

3.2. Instantiation

To effectively use the employed ontologies, it is necessary for the modeled ICEs, containing the reference to the data, material entities, and properties for which such information is about, to be instantiated in the model. When working with preexisting data, doing so requires some description of the data and relevant assets to enable the creation of the instances. Furthermore there is a choice to be made regarding how to represent the data in the model, by either assigning the value directly to a property or by representing an identifier to the "location" of a property's respective data. The first option implies in inserting the values directly in the model, thus increasing the number of instances (consequentially increasing the complexity of the model) by an amount equal to the number of samples of data. This is not viable when working with time series, whose number of samples may surpass the thousands per data, as observed in our use case. As such, we determined that the best approach would be to utilize ICEs to represent the agglomeration of all samples of a piece of data (a time series) and utilize annotation properties to store the identifier of such data, enabling its retrieval from where it is stored.

The instantiation process can be done either manually or automatically, depending on how the necessary data is stored. Approaches for automatic instantiation start with the formalization of R2RML[1] by the W3C[2] as a language for describing how data in a relational database is related to entities in an ontological artifact. This and similar solutions require that the data is sufficiently structured in a way

[1]https://www.w3.org/TR/r2rml/
[2]https://www.w3.org/

compatible with the ontological definition of the domain, limiting its applicability to cases in which the schema of the data carries all the semantics related to the underlying taxonomy.

When working with the time-based information from historian systems, there is no guaranteed manner for representing such semantics. This system can represent as little as just a time series and its unique identifier and may provide some auxiliary description, as is the case with the *OSI PI* system. The taxonomical relations are extracted from the *PI Asset Framework*, which represents the hierarchical relations between the various involved assets but does so as a provider for a certain "target audience", not creating a general, consistent model [5]. As detailed before, this use case illustrates a shortcoming of the automatic approach to instantiation and is why we have decided on manual instantiation.

For the instantiation process, an ontology engineer who understands the domain will look at the available descriptions to determine to which asset the data is related, creating not only the instance of the ICE but also the instances of the related asset and its properties and all relevant relations between entities.

3.3. Approach to Dealing With a Complex Model

Reasoning is an important part of working with ontologies, being a tool during its formalization phase for logical consistency checking and in its implementation phase for inferencing. For its importance, we must choose carefully what reasoner to utilize for the domain ontology. In our implementation, we have utilized the Hermit reasoner, mainly for its overall good performance when working with OBO ontologies [6], like BFO and IOF-Core.

Another important choice in regards to reasoning is how to utilize the reasoner during the implementation phase (that is when a system is utilizing the ontology) as inferencing can change drastically how the information is presented after instantiation, after all "A reasoner is a program that infers logical consequences from facts, assertions, and axioms" [7] and instantiation is the act of incrementing the model with new assertions and axioms. Although it is desirable to infer at the most complete state of the model, the act of inference is computationally intensive and time-consuming. For this reason, it is not applicable to constantly run the reasoner during the system's uptime when dealing with complex ontologies. Such a limitation and the fact that the real data is not present in the model corroborated the need only to run the reasoner when the model itself needs to change, which means when there is a change to the disposition of the assets.

Ontology artifacts described in the RDF language (and derivated languages, like OWL) are represented by triples in the shape "subject-predicate-object", such that entities are represented in subjects and objects, while predicates are relations between entities. This structural characteristic resulted in using purpose-built databases, namely triple-stores (extended to quad-stores for working with named graphs), to store ontological definitions. These purpose-built databases are not only for storage but also for the retrieval of triples through semantic queries, normally utilizing SPARQL[3] as their query language.

3.4. Abstracting Information From The Knowledge Base

Once the instances have been created and the reasoner's inferences have been materialized, the ontology is ready to be used. Doing so requires retrieving the triples described in the ontological artifact, either using local files and APIs or through a semantic query language and an available endpoint.

One option is developing a specific API providing a list of methods for certain types of semantic queries (*e.g.,* retrieving the class that a given individual instantiates, all individuals linked via a given relation). Based on that, other services would consume such methods when needing to retrieve information. This approach has the advantage of establishing a clear separation of concerns between the different services of the system. It shields a front-end service from the technological details of how the model is implemented and queried, thus preventing changes to the model (like changes to the IRIs of entities) from reflecting in changes to the front-end.

[3]https://www.w3.org/TR/rdf-sparql-query/

For applications that require simple queries on their knowledge base, an API with a small and easily learned set of methods for semantic query would be enough. However, this approach would lead to difficulties if the application requires numerous and complex queries. An API with a large list of methods covering all required semantic queries would result in a hard-to-maintain implementation. An alternative would be sticking with a small API with a simple, basic set of methods that could be algorithmically combined to retrieve data that would only be retrievable via more complex queries. The downside of this approach is that the meaning of the query would be implicit in the algorithms that combine the API's methods. In addition, dynamic scenarios in which new functionalities are constantly being added would require constant expansion of the API, worsening the mentioned problems. With that, instead of a formal and standard method of expressing and processing queries, we would have *ad hoc* algorithmic solutions, which compromise their understandability, the evaluation of their correctness, and their overall maintainability.

In contrast, the direct use of SPARQL serves two purposes: being standardized enough to work on several possible storage implementations and being flexible enough to allow the service to choose how it wants the data to be presented. Furthermore, the abundance of reference material on SPARQL creates an easier environment for understanding and long-term service maintenance.

Even though triple-stores are very efficient databases that expose an endpoint for semantic querying, some environments may impose restrictions on their use, be it for enterprise norms or very strict architectural styles. In this situation it is still possible to create an architecture agnostic to the storage method of the ontological artifact by using SPARQL. This query language is supported by most triple-stores and by many APIs, enabling the creation of a self-implemented endpoint (one such implementation is discussed in section 4).

When working with SPARQL, the service that needs the data must first know what it needs. The service may always know this or may result from interaction with another service or the user. The application *Ontology Explorer* described by [8] is a good example of this aspect.

By making a query with the knowledge known by the service, it is possible to present the asset information to the user, who, by navigating such information, will present to the system what information he/she needs, generating a new query. In terms of implementation, this means having some queries ready to retrieve the base amount of information the service knows it will need and various "template queries" with certain gaps to retrieve information the service might need when the gaps are filled.

4. Software System Architecture

The infrastructure we used to deploy the application prototype consists of a cluster of 16 computers, 9 of which are servers connected to the internal network of the UFRGS Informatics Institute. The prototype is a web application with a microservices architecture. This software design pattern structures an application as a collection of small, low-coupling, and independently deployable services. Each microservice is responsible for specific and well-defined functionality, with communication being carried out via lightweight protocols, usually HTTP or message queues like MQTT. This approach contrasts with monolithic architectures, where an application is created as a single, cohesive unit. Microservices offer several advantages over monolithic architectures:

- **Scalability**: Each microservice can be scaled independently, allowing more efficient use of resources and better management of variable workloads.
- **Flexibility**: Teams can develop, deploy, and maintain each service independently, allowing faster development cycles and easier integration of new technologies.
- **Resilience**: The failure of a microservice is less likely to affect the entire system, as the services are isolated from each other.
- **Reuse**: Components can be shared and reused in different or similar applications, promoting modular design and reducing code duplication.

However, the microservices architecture presents some challenges, such as increased complexity, the need for efficient coordination of microservices, and the possibility of increased communication latency between them. Therefore, it is essential to evaluate the most appropriate way to implement this type of architecture according to the application's requirements. We used the Docker Engine [4] containerization technology to deploy the application components, which is responsible for packaging software artifacts and their dependencies in containers, i.e., portable units that can run consistently in different computing environments. Unlike virtual machines, containers share the kernel and libraries of the host operating system, resulting in lower resource usage and faster start-up times. For this reason, using containers has become common practice in software development and distribution, especially in microservice architectures, where each service can be packaged and deployed as a separate container. This practice reduces the complexity of deploying the application and allows for a more appropriate sizing of the necessary resources.

The Kubernetes [5] platform is responsible for managing the execution of the deployed containers on the provisioned computing infrastructure. Kubernetes is an open-source container orchestration platform - maintained by the Cloud Native Computing Foundation (CNCF) - that automates container deployment, sizing, and management in distributed computing environments. The platform aims to make it more flexible to manage workloads distributed across several nodes, which can be physical or virtual machines in a cloud or on-premise environment. In Kubernetes, the smallest and most basic deployment unit is called a Pod, representing a single instance of a process running in the cluster. A Pod encapsulates one or more containers, storage resources, a unique IP address, and other configuration options.

4.1. Components and Technologies

Figure 4.1 shows the application components configured for operation in the Kubernetes cluster. Of the 16 nodes provided by the infrastructure, one is exclusively used to intercept external requests for access to services and to configure and manage the cluster (master), while the other nodes (workers) are used to deploy the containers that encapsulate the execution of the microservices that make up the application.

4.1.1. Front End

The application's front end consists of a single container that exposes a web application developed in JavaScript with the React[6] library. This component, called Data Explorer, allows users to access a graphical interface for building and visualizing dashboards for analyzing production data (time series) collected from the historian system, using a domain ontology as a guide for navigating through the modeled physical plant. A more detailed front-end description is published at [8]. In addition, users can create annotations on specific points in the time series and save them in a document database for later retrieval. For this, the back end exposes HTTP endpoints to query the knowledge base built from the ontology and to access time series data and annotations. We used the Nginx[7] web server to expose the application to external traffic.

4.1.2. Back End

The back end comprises two REST API services developed in Python with the FastAPI[8] framework and deployed in separate containers: Data Hub and Knowledge Base. Both services use the Uvicorn[9] web server to expose HTTP endpoints to access resources. The Data Explorer consumes the Data

[4]https://www.docker.com/get-started/
[5]https://kubernetes.io/pt-br/
[6]https:/react.dev/
[7]https://www.nginx.com/
[8]https://fastapi.tiangolo.com/
[9]https://www.uvicorn.org/

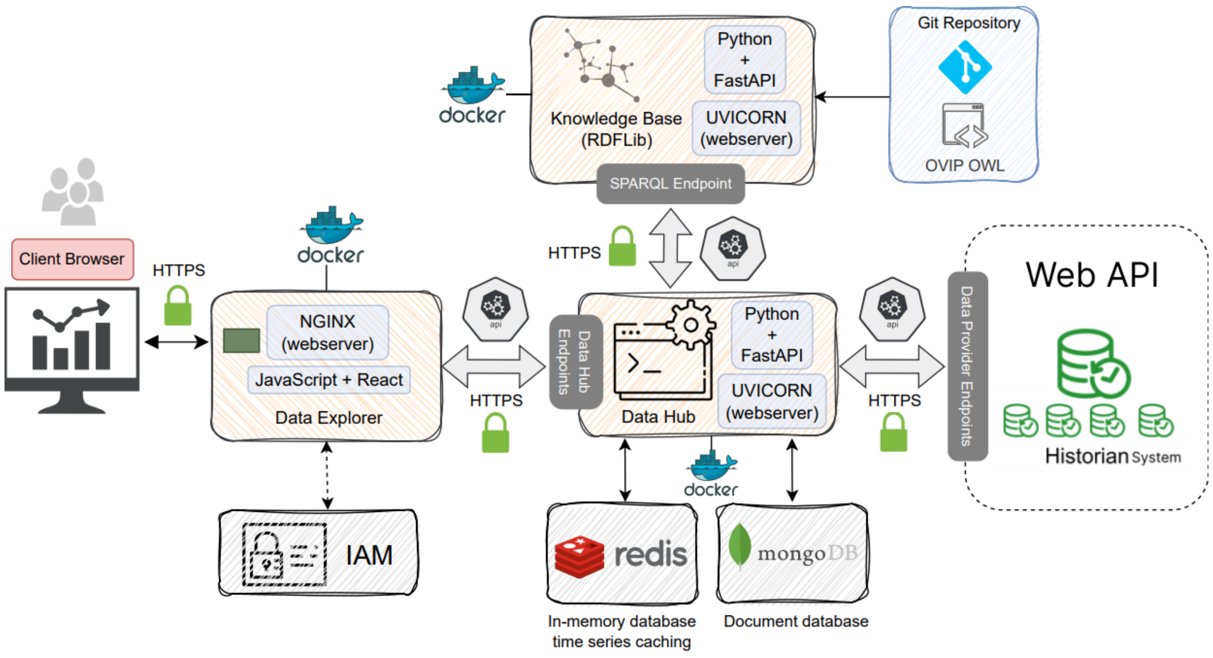

Figure 3: Application architecture and components

Hub service, which exposes endpoints that allow contextualized access to data (time series) retrieved by a web API based on the attributes of the entities represented in the ontology. Given the tags and timestamps, endpoints allow querying the knowledge base and searching for the time series in the historian system. The Data Hub service also exposes endpoints for CRUD operations on a document database to persist user profiles, configurations, and annotations associated with the dashboards built into the web application.

On the other hand, the Knowledge Base service is directly consumed only by the Data Hub. It creates an in-memory triplestore from the ontology's OWL file retrieved from an external repository like Git. The prototype's knowledge base is materialized as a knowledge graph, which includes the O3PO classes, other imported ontologies, and the instances corresponding to the domain entities. A reasoning tool generates this graph and performs logical inferences about the instances, then exported as a single RDF file. Finally, we use the RDFLib[10] package to import the RDF file, which allows access to the knowledge base via *SPARQL* queries on the graph, as can be seen in Figure 4.1. Consequently, the Knowledge Base service exposes a SPARQL endpoint to query the semantic structure of the knowledge base generated from the ontology and retrieve the material entities and the informational entities (tags) associated with the instances of the material entities included in the ontology.

Given the tags retrieved by SPARQL queries on the Knowledge Base, the time series data can be obtained through the data provider endpoints, as seen in Figure 4.1. In our deployment, these endpoints are from the WEB API, but they can be from other historian systems providing sensor data or even from more complex data platforms like the OSDU[11].

4.1.3. Storage

We used two services for data storage: MongoDB[12] and Redis[13]. MongoDB is a document-oriented database that stores user profiles, configurations, and annotations associated with the dashboards built into the web application. Redis is a service that makes it possible to create in-memory databases. We

[10]https://rdflib.dev/
[11]https://osduforum.org/
[12]https://www.mongodb.com/
[13]https://redis.io/

used Redis to cache the time series data retrieved by the data provider endpoints. The aim is to reduce latency in accessing previously retrieved data and reduce the load on the WEP API endpoint with redundant requests.

4.1.4. Data Security

We used the open-source platform Istio[14] to connect, protect, and manage the microservices in a unified way by including reverse proxies (Envoy) alongside the containers that implement the application's microservices, namely the Data Explorer and the Data Hub and Knowledge Base services. Therefore, each Pod created in the Kubernetes cluster comprises two containers: one for deploying the microservice and the other for intercepting all the requests that arrive or leave the microservice.

The control layer offered by Istio configures the reverse proxies to keep communication between microservices encrypted with the Transport Layer Security (TLS) security protocol, with Istio acting as the certificate authority for a data layer that uses mutual TLS. In this way, microservices are relieved of having to provide the security layer. They can focus solely on executing their functions. Moreover, Istio allows the implementation of a mesh of microservices with support for a range of features and functionalities to improve the observability, reliability, and security of container-based applications following the distributed computing model. Additionally, we used the Keycloak[15] as the Identity And Access Management (IAM) tool to provide OAuth 2.0-based authentication for the application prototype.

5. Related Work

In the context of oil and gas, where dealing with large ammounts of complex data is the norm, the idea of leveraging ontology-based solutions for dealing with correctness and timelyness of data retrieval is proeminent [9, 10, 11].

While these focus on the necessary semantic infrastructure, others concern with the different ways of articulating query interfaces and presenting the data properly to user [9, 8]. Hill et al. [12] used ontologies for dealing with real-time sensor data for event detection in oil and gas production. Kharlamov et al. [13] address the challenge of data gathering in large, data-intensive corporations like Equinor through a semantic data access system based on the Ontology Based Data Access (OBDA) approach. Our work encopasses the problem from beginning to end, from the semantic definitions to the implementation of the defined architecture, which provides a valuable contribution to future industrial ontology-based applications.

6. Concluding Remarks

This paper discusses a case study on an ontology-based information system developed to provide coherent and contextualized data across various applications within an offshore petroleum production plant. The system's architecture is outlined through a description of the components employed in the application.

Future work will expand the description of events and experiment on the temporalization of the model. Additionally, support to real-time data will be added to the application which is a necessity in terms of an effective digital twin.

Acknowledgments

The authors acknowledge CAPES-Brazil Finance Code 001, the Brazilian Agency CNPq, and the Petwin Project, supported by FINEP, Libra Consortium (Petrobras, Shell Brasil, Total Energies, CNOOC, CNPC).

[14]https://istio.io/
[15]https://www.keycloak.org/

References

[1] T. Brewer, D. Knight, G. Noiray, H. Naik, Digital Twin Technology in the Field Reclaims Offshore Resources, volume Day 1 Mon, May 06, 2019 of *OTC Offshore Technology Conference*, 2019.

[2] N. O. Santos, F. H. Rodrigues, D. Schmidt, R. K. Romeu, G. Nascimento, M. Abel, O3PO: A domain ontology for offshore petroleum production plants, Expert Systems with Applications 238 (2024) 122104.

[3] M. Drobnjakovic, B. Kulvatunyou, F. Ameri, C. Will, B. Smith, A. Jones, The industrial ontologies foundry (iof) core ontology, in: FOMI 2022: 12th International Workshop on Formal Ontologies Meet Industry, September 12-15, 2022, Tarbes, France, 2022, pp. 1–13.

[4] L. F. Garcia, M. Abel, M. Perrin, R. dos Santos Alvarenga, The geocore ontology: A core ontology for general use in geology, Computers & Geosciences 135 (2020) 104387.

[5] E. Kharlamov, F. Martin-Recuerda, B. Perry, D. Cameron, R. Fjellheim, A. Waaler, Towards semantically enhanced digital twins, in: 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 4189–4193.

[6] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang, Hermit: An owl 2 reasoner, Journal of Automated Reasoning 53 (2014) 245–269.

[7] A. Khamparia, B. Pandey, Comprehensive analysis of semantic web reasoners and tools: a survey, Education and Information Technologies 22 (2017) 3121–3145.

[8] N. O. Santos, J. C. Rivera, R. H. Petry, F. H. Rodrigues, G. S. Nascimento, J. L. Comba, M. Abel, Ontology Explorer: An Ontology-Based Visual Analytics System for Exploring Time Series Data in Oil and Gas, IOS Press, 2023.

[9] A. Soylu, M. Giese, R. Schlatte, E. Jimenez-Ruiz, E. Kharlamov, O. Özçep, C. Neuenstadt, S. Brandt, A. Bikakis, T. G. Stavropoulos, G. Meditskos, Querying industrial stream-temporal data: An ontology-based visual approach, J. Ambient Intell. Smart Environ. 9 (2017) 77–95.

[10] E. Kharlamov, D. Hovland, M. G. Skjæveland, D. Bilidas, E. Jiménez-Ruiz, G. Xiao, A. Soylu, D. Lanti, M. Rezk, D. Zheleznyakov, M. Giese, H. Lie, Y. Ioannidis, Y. Kotidis, M. Koubarakis, A. Waaler, Ontology based data access in statoil, Journal of Web Semantics 44 (2017) 3–36.

[11] E. Kharlamov, D. Hovland, E. Jiménez-Ruiz, D. Lanti, H. Lie, C. Pinkel, M. Rezk, M. G. Skjæveland, E. Thorstensen, G. Xiao, D. Zheleznyakov, I. Horrocks, Ontology based access to exploration data at statoil, in: ISWC 2015, 2015, pp. 93–112.

[12] M. Hill, M. Campbell, Y.-C. Chang, V. Iyengar, Event detection in sensor networks for modern oil fields, in: Int. Conf. on Distributed Event-Based Systems, DEBS, 2008, p. 95–102.

[13] E. Kharlamov, M. Skjæveland, D. Hovland, T. Mailis, E. Jimenez-Ruiz, G. Xiao, A. Soylu, I. Horrocks, A. Waaler, Finding data should be easier than finding oil, in: 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 1747–1756.