JThermodynamicsCloud: Case Study in an Ontology-Driven NoSQL Database Cloud Based Application in Chemical Domain

Edward Blurock[a]
Blurock Consulting AB, Lund, Sweden

Keywords: Model Driven Development, MDD, MDA, Ontology-Driven, NoSQL, Cloud-Based Application, Chemistry, Thermodynamics, Database Management.

Abstract: JThermodynamicsCloud is software service for the combustion research domain to perform thermdynamic calculations and manage the data needed to make those calculations. The JThermodynamicsCloud service can be said to be a model driven application, where the ontology is a platform independent model of the data and operational structures. All the ontology concepts outlined here, from the ontology definition to the utilization of this definition in the application, have been implemented. The ontology, as used by the service, has three distinct purposes: documentation, data structure definition and operational definitions. One goal of the ontology is to place as much of the design and domain specific structures in the ontology rather than in the application code. The calculation itself is highly dependent on the varied types of molecular data found in the database The complete service is a system with three interacting components, a user interface using Angular, a (RESTful) backend written in JAVA (with the JENA API interpreting the ontology) and the Google Firestore noSQL document database and Firebase storage.

1 INTRODUCTION

JThermodynamicsCloud is software service for the chemical, or more specifically, the combustion research domain. The primary purpose is to perform the calculation and manage the data needed to make the calculation. The complete service is a system with three interacting components, a GUI interface, a RESTful backend and a noSQL document-based database. The service uses these three components to make calculations for thermodynamic quantities based on molecular species structure. The calculation itself is highly dependent on the varied types of molecular data found in the database.

JThermodynamicsCloud service can be said to be a model driven application where the model is defined in the ontology. JThermodynamicsCloud is comprised of three different platforms with three different data formats. The user interface uses Angular (*Angular*, n.d.), the (RESTful) backend is written in JAVA and the Google Firestore (*Cloud Firestore | Store and Sync App Data at Global Scale*, n.d.) noSQL document database has a map-based data format. These different data formats are united through the platform independent ontology definitions. All definitions and methodologies within all three systems of the service have a corresponding ontology classes. This means that communication between systems has an ontology reference. In particular the data structures. This is particularly important because the data structures in each of the systems are in a free format, namely, property name and untyped value pairs. The ontology provides the underlying structure by defining the property name and the information that field contains. The ontology definitions are translated to the exact data structures within each of the respective systems. In all systems, the data structure is a JSON like data structure.

1.1 Background and Context

The use of ontologies in software engineering is, of course, not new (Bhatia & Beniwal, 2016; Espinoza-Arias et al., 2021). This application can be seen as an application of ontologies in a Model Driven Development (Bučko et al., 2019; Silva Parreiras et al., 2010) or Model Driven Engineering (Gaševic et al., 2009) context, where the ontology guides the

[a] https://orcid.org/0000-0001-9487-3141

Blurock, E.
JThermodynamicsCloud: Case Study in an Ontology-Driven NoSQL Database Cloud Based Application in Chemical Domain.
DOI: 10.5220/0012254700003598
In *Proceedings of the 15th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2023) - Volume 2: KEOD*, pages 288-295
ISBN: 978-989-758-671-2; ISSN: 2184-3228

development of the software, particularly the data structures and the operational definitions using the data structures. In a sense, all the modeling languages such as the Unified Modelling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM), are taken over by the use of ontology of the application. The modelling descriptions needed for the JThermodynamicsCloud application are handled solely by the ontology.

The ontology data model is currently not used to actually generate software code, though this has been experimented with in the CHEMCONNECT(E. Blurock, 2021; E. S. Blurock, 2019) application (which is at the base of JThermodynamicsCloud). But the ontology definitions do specify, within a Model Driven Architecture (Bučko et al., 2019), the top two levels of abstraction, namely the Computer Independent Model (CIM) and Platform Independent Model (PIM). A common ontology class defines data and operational classes that are common to the three platforms within the system, namely the interface, the background and the database. The ontology is interpreted under runtime by the background service, written in JAVA, and uses the JENA API(Apache, n.d.). SPARQL queries are made, and the answers are converted to appropriate data structures.

Another advantage that was exploited with using ontologies in the data modelling approach is working toward developing a 'normalized system'(De Bruyn et al., 2018; Suchánek et al., 2021). The five necessary conditions can be seen directly in the ontology definition. 'Separation of Concerns' can be seen in the data structure class and operational definition classes. 'Separation of Actions' can be seen not only in the operational classes' definition, but also the concept of transactions (see section Ontology Transaction Definition), which break down a task into individual and traceable subtasks. 'Action Version' and 'Data Version' transparency can be seen in modularity and how data is passed between operational units in the software.

2 MOTIVATIONS AND USE OF ONTOLOGIES

The following outlines how the ontology was used and implemented. All the concepts, definitions and code have been implemented in the current version of JThermodynamicsCloud. The Results: EXPERIENCE section outlines the experiences and advantages of the use of ontologies as outlined in this paper.

Within this section, ontology class names and identifiers have the form:

```
namespace:nameofclass
```

The `namespace` denotes which ontology the class is defined. For example, all with the namespace `dataset` are from the JThermodynamicCloud ontology. The JThermodynamicCloud ontology with all the definitions shown in this section can be found in github:

https://github.com/blurock/Angular/releases

2.1 General Goals of Ontology

The ontology as used by JThermodynamicsCloud has three distinct purposes, documentation, data structure definition and operational definitions. These are briefly outlined in the following sections.

2.1.1 Documentation

Through annotations and the hierarchy of ontology classes a level of documentation is provided that is not inherently present within the software implementation definitions. Through the class hierarchy a context is given to each object and through the annotations there is human and machine readable documentation.

2.1.2 Data Structure Definition

Every data object in JThermodynamicsCloud has a respective ontology definition. This gives a common reference for how each data structure is translated in the different software components of the system. Since the definition is machine readable, this allows a certain amount of automation, for example in the user interface, to take place. For the software engineer, this common reference provides the template for accessing data from the data structures which is not otherwise present because the data structures themselves are inherently non-typed and free format.

2.1.3 Operational Definitions

In addition to data structures within the system, there are also sets of algorithms using these data structures. Certain classes of algorithms have corresponding ontology definitions. For example, all RESTful services have a corresponding ontology definition. In addition, within the background system, certain classes of working algorithms (programmed using the enumeration class in JAVA) have ontology definitions. These are often associated with pull-down list choices within the interface or specific related manipulations dependent on classes of data types. This is an example of the backend interpreting the ontology and delivering the appropriate data structure to the interface.

2.2 Uses of the Ontology

The following subsections provide a summary of how ontologies are implemented and utilized in multiple capacities in JThermodynamicCloud. In later sections, some of these applications are elaborated with more details.

2.2.1 JSON Data Object Definitions

The JSON (like) representation, meaning property-value pairs is used throughout JThermodynamicsCloud. However, JSON is a an untyped and free-format language, meaning that the only information about the fields in the JSON object are the property labels. Use of ontologies provides context and documentation of the fields within the JSON objects. There is a one-to-one correspondence between the data object definition and the ontology class. The ontology is what defines the type of the data object for each property label within the free-format context. This is further outlined in JSON data object definitions.

2.2.2 RESTful Service Definition

The functionality of JThermodynamicsCloud are accessed through the RESTful services. The RESTful service ontology is used to provide context, through a hierarchy of definitions, and, within each class definition, the input and output objects expected from the restful service. In addition, annotations to each class give further documentation. These are defined in a subclass of `dataset:DatabaseServicesBase` which is a subclass of the `prov:SoftwareAgent` (of the PROV ontology). This is further outlined in RESTful Service Definition.

2.2.3 Transaction Description

Transactions represent class of RESTful services which modify the database (as opposed to services which perform only queries on the ontology and the database). The corresponding ontology class specifies the input, output and prerquisites of the transaction. See the Section Ontology Transaction Definition.

2.2.4 Database Structure Hierarchy

The noSQL document database of JThermodynamicsCloud is a hierarchy of collections and documents. All catalog objects are 'documents' (in the sense of the noSQL database) located at the end root nodes of the hierarchy. The hierarchy position of the collection of catalog objects types gives a context to the objects. The database definition ontology is used to define this hierarchial structure. Within this definition is which catalog object type is being defined and how the nodes leading to the catalog object are to be named. These are defined in a subclass of `dataset:CollectionDocumentHierarchy` which is a subclass of the `skos:Concept` (of the SKOS ontology). This is further outlined in Database Hierarchy Specification in the Ontology.

2.2.5 Classification Components

Some single string valued components (in the sense of the DCAT ontology) represent classifications (enumerations) with specific values. The ontology definition of the component points to the top of a hierarchy (for a tree classification) or the top of a set of subclasses (for a list). In the annotations of these subclasses are labels and comments that can be used for the user interface (and documentation). The subclasses can be components, if the selection should be, for example, types or records or catalog object (in the sense of the DCAT ontology), if the selection should have more information.

2.2.6 Implementation Operations by Type

Often within the implementation there are set of similar operations, with the same input, but different functionalities based on type. In JAVA this is an enumeration with abstract methods. In the ontology, these are defined in a subclass of `dataset:DataObjectManipulation` which is a subclass of the `prov:SoftwareAgent` (of the PROV ontology (Timothy Lebo, Satya Sahoo, Deborah McGuinness, 2013)).

2.2.7 Units

An important aspect of scientific numerical data is units. JThermodynamicsCloud uses the QUDT ontology (QUDT, 2018) to enable the transformation of one unit to another. In general, the original units, i.e. that of the source data, are stored in the database. In the calculation and presentation, the user can choose the preferred units. The QUDT ontology includes all the transformation parameters needed. This is further outlined in Parameters and Units: QUDT ontology.

2.3 JSON Data Object Definitions

The representation of data objects in JThermodynamicCloud has a JSON like structure, i.e. property label with a corresponding value. This design decision allows the same data representation to be used even though the syntactical form within each of the different system components can differ. The JAVA objects in the background services are based on `com.google.gson.Gson`, the data objects in the Angular Material are based on typescript JSON objects, the RESTful service representation of data in the body are JSON objects and in the database the JSON objects translate directly to the mapping structure used in the Google Firebase Firestore representation. This common format design makes the transitions, between user interface, backend computation/manipulation and database access seamless. The ontology definition also gives the software engineer managing and programming the system a common reference.

The JSON format is non-typed and free-format and thus the only documentation of the objects themselves are the keywords of the fields. In JThermodynamicsCloud, the ontology definitions provide machine and human interpretable documentation and context to the JSON data objects. There is a one-to-one correspondence between each JSON object type and the ontology class. The JSON objects are modelled in the ontology after the DCAT ontology. The keywords of the JSON objects are defined in the `dcterm:identifier` field in the annotations. The annotations also give human interpretable labels (`rdfs:label`) and comments (`rdfs:comment`) about the objects. The position of the ontology hierarchy of the object gives added context to the objects. From a programming perspective, the ontology object definition is an invaluable tool for keeping track of the objects structure. Since the JSON object is inherently free format it can be difficult to keep track of which fields are available or which fields must be filled in. This task, which is often accomplished with the programming environment, cannot be applied to JSON objects. Using the ontology as a reference facilitates this task. The additional documentation within the ontology is also helps the programming process.

Due to the machine readability of the ontology, a certain degree of automation can be utilized. This is particularly useful when a generalized algorithm can read the ontology and base its manipulation on the ontology definition. This means that updates to include more data can be made only updating the ontology and not modifying the program. One common example of this are pull-down lists in the user interface. More choices can be added within the ontology without touching the program.

2.4 RESTful Service Definition

The input and output of RESTful services are also, to a certain extend, free-format, similar to the free format of JSON objects. The purpose of the ontology, with regard to RESTful services, is to provide machine readable and standard documentation of each of the services provided by JThermodynamicsCloud. A POST to a RESTful service, as used by JThermodynamicsCloud, sends JSON data to the server and with this information performs a task and sends a response, also a JSON object, back to the client. JSON objects are, in general, free-format, so they can be of any property-value pair, where the value can be a simple data object like a string or a nested object like another JSON object. There is nothing in the RESTful service definition that specifies the form of the JSON object. So the user must 'know' the form of the valid data that the server expects and also the form of the response. In the JThermodynamicsCloud ontology definition, the exact form of the JSON data needed to perform the task and the expected JSON response is specified. Thus the ontology provides documentation for the user of the service. However, since the ontology is machine-readable, its role of just documentation can be expanded. For example, the ontology could provide a level of input checking to see that the JSON object in the POST has all the necessary information. Also the machine readable ontology could be used to set up the user interface.

2.5 Ontology Transaction Definition

Transactions are one of the more important features promoting the traceability of data as it is created and transformed within the database. A single transaction event performs a single task. Transactions also promote modularization of tasks (one important aspect of ‘normalized system’). A transaction task could be dependent on other prerequisite transaction tasks. In other words, the prerequisite transactions set up the necessary data for the current task. Thus, inherent in the definition of a transaction is the list of prerequisite transactions that are needed. The transaction event can be thought of as a node in the tree of manipulations that data undergoes. The transaction also is a tool to isolate single tasks making the organization of data manipulation more transparent.

The ontology's role in this design is to give the specification to the input to the transaction, this includes the list of transactions that are needed to perform the current transaction. Due the structure of the database and the organization of transactions, often just knowing the type of transaction needed is enough to isolate the particular prerequisite transaction needed (an aid to automation). If the exact prerequisite transaction cannot be isolated, the choices for the user are fewer.

2.5.1 Transactions and Traceability

The purpose of transactions is to keep track of every change in the database. Whenever the database is to be modified, a RESTful transaction is performed. The purpose of a transaction is to have a trace of the evolution of a database catalog object. To perform a transaction, the prerequisites to the transaction have had to be performed. These prerequisites are themselves transactions. In this way the user can trace the evolution of the database catalog object by tracing through the transactions that were used to create the current database object. To perform a transaction, first the prerequisites of the transaction must be collected. The transaction ontology (see Transaction Specification in Ontology) definition contains the list of prerequisite transaction classes. The prerequisite for the process is a database transaction object of this class. When a transaction process is initiated, the set of database transaction catalog objects are retrieved. Each of these transaction objects (created previously by another transaction) has the FirebaseID's (the address within the database) of the needed prerequisites. The FirestoreID's are used to retrieve from the database the actual prerequisite catalog objects. These catalog objects are then passed to the transaction process.

In order to perform the transaction, in addition to the list of prerequisite transaction catalog objects, some additional information may be needed to guide the process. These are specified through a `dataset:ActivityInformationRecord` (a subclass of `dcat:CatalogRecord` of the DCAT ontology) record object. The properties of this record object supplements the prerequisite data. In the ontology definition of the transaction, the specific class of `dataset:ActivityInformationRecord` gives a specification of the specific data needed. In the data sent to the RESTful service this information is under the `dataset:activityinfo` property.

The final output of a transaction process is an array of catalog objects of the same class. These are passed back through the RESTful process response. Each catalog object has the FirestoreID of the transaction that created it. The FirestoreID of each of the output catalog objects are listed in the database transaction object of the process.

2.5.2 Transaction Specification in Ontology

The ontology provides the complete specification of each transaction, i.e. the form of the expected input and output information. Each transaction is described with an ontology class which is in the subclass hierarchy under `dataset:TransactionEvent`, a subclass of Dublin Core class `dcmitype:Event`. The specification in the form of an ontology class has several properties:

`dcat:catalog`: This is the class of the catalog object that the transaction creates.

`dcterms:source`: This is the class of the `dataset:ActivityInformationRecord` JSON input information object that is needed to derive the output catalog objects. The ontology class specifies the supplementary data needed to perform transaction.

`dcterms:type`: This is the class of transaction that is produced. This class is a subclass of `dataset:ChemConnectTransactionEvent` (which ultimately is a subclass of `dataset:SimpleCatalogObject` which is a subclass of `dcat:Catalog`). The database transaction catalog object is not the same class as the transaction class specification. Different transaction class specifications can produce a database transaction object of the same class.

`dcterms:requires`: These are the classes of the transaction specifications, i.e. subclass of `dataset:TransactionEvent`, that are required before this transaction can be performed. Through these transaction specifications the database transaction object class that was produced by the transaction is specified. This class information is used to retrieve the specific database transaction.

2.6 Database Hierarchy Specification in the Ontology

The Google Firestore (*Cloud Firestore | Store and Sync App Data at Global Scale*, n.d.) noSQL database has a document oriented data-model (*Cloud Firestore Data Model*, n.d.). The structure is an alternating hierarchy of collections and documents. Within a collection, which is specified by a string label, is a set of documents. A document is a map of property-value pairs. The value specified by the document can either be an object, such a string or numerical value, or it can be a label to a further subcollection. Thus the final document (one with no further subcollections under it) is a set of property-object value pairs at the end of a hierarchy. Access to this document is through a alternating set of collection labels and subcollection labels within a document.

Within the JThermodynamicsCloud implementation, the catalog objects, a mapping similar to a JSON object, are found at the end nodes of hierarchy of collection and documents nodes. The nodes of the branch leading to the catalog object are designated by string labels. The string labels are either a collection label or a document property label pointing to a subcollection. Thus each of the documents in the hierarchy leading to this final catalog object only have a set of unique single labels, each pointing to a different subcollection. In the document form of noSQL databases, the design model is to group all 'documents' into a single collection. In JThermodynamicCloud, all collections of documents, the catalog objects, are at the end of a branch designated with string labels for each node. The address, meaning the set of labels of the nodes in the branch leading to the catalog object, are specified in the FirestoreID class.

In JThermodynamicsCloud the position of each catalog with a collection-document hierarchy object is specified under the `dataset:CollectionDocumentHierarchy`. The ontology classes and subclasses under this class specify the classes of collection-document nodes leading to the final catalog objects at the end of the branch. Each class in this hierarchy specifies how the node should be labeled. A given catalog object class has a unique position in the hierarchy defined by the ontology. The label specification of the nodes leading to the catalog object can be a constant or can be derived from the current catalog being stored. The end class in this specification ontology hierarchy specifies which class of catalog object is to be stored in the database. In each of the ontology class specifications, the `rdfs:isDefinedBy` annotation specifies how the label is to be created. The set of methods to create these labels are defined in the ontology under the `dataset:GenerateStringLabel` class (this is an example of an implementation of operations by type). Each of the methods in the ontology has a corresponding code within a JAVA Enumeration class (using the ontology class name as reference). If the (final) node is to store a catalog object, then the class of the catalog object is specified by `skos:member`. Thus when a class object is to be stored, the `skos:member` fields are searched for the class name. The catalog object address (to be specified in the FirestoreID catalog address) is generated by the branch of classes leading to this node.

For example, suppose we are to store a specific instance of a collection set of the class `dataset:ChemConnectDatasetCollectionIDsSet`. Following the ontology class hierarchy under `dataset:CollectionDocumentHierarchy`, we find there are four classes in the hierarchy leading to the class specifying the instance class as the `skos:member`. This means that this class is the document specified by four labels. All four classes together specify the labels of the branches leading to the specific instance (and specified in the FirestoreID address).

2.7 Parameters and Units: QUDT Ontology

One of the most important aspects of managing scientific data is the handling of units. Although there is a standardized system of units, the SI, International System of Units, units(*SI Brochure*, n.d.), it is not absolutely said that researchers are using them. For example, in the combustion thermodynamics domain (the domain of this tool), though the trend is toward SI units, it is not said that the available data needed by the application is in SI units. In addition, even if SI units are used, there is still some conversion that is necessary. For example, depending on the range of values of the parameter, the simple unit could be used or if it has a larger range, with the prefix, for example, kilo-, or if with a smaller range, for example, milli-.

There are two philosophies of handling units in a database. The first is to convert them to a 'standard', relative to the database. Each parameter would have it's expected unit. With this philosophy, the units are implicit and do not need to be stored with the value. The disadvantage is that it is left to the user on input to convert the units. This can lead to errors and this 'hidden' step makes it more difficult to trace the value to and check the value with the original source value. The other philosophy, is to store not only the value, but the unit used for the value. In JThermodynamicsCloud the second philosophy is used for basically two reasons. The most important reason is that keeping the unit with the parameter means that the value stored can be exactly that of the original source. This promotes traceability and error-checking.

To convert between different units requires a knowledge base of units. For a application within a small domain, this knowledge base could be 'hard-coded' into the application. However, a more general approach is provided by the QUDT ontology(QUDT, 2018). In the QUDT ontology each 'kind' of unit is an instance of `qudt:QuantityKind`, for example `qudt:MolarEnergy`. In the annotations of the instance of `qudt:QuantityKind`, the available units of this kind are given by `qudt:applicableUnit`. For example, on applicable unit of `qudt:MolarEnergy` is `qudt:CAL-PER-MOL`, i.e. *calories/mol* of substance. In the applicable unit, the conversions to the SI units are given. To convert calories/mol to the SI unit *joule*, one multiplies the calories by *4.184*.

The ontology definition for a parameter data object in JThermodynamicCloud stores both the value and the specification, meaning the units for the value. In addition, uncertainty values are taken into account. A parameter is represented by the `dataset:ParameterValue` class with three fields:

`dataset: ValueAsString`: This is a string representation of the value. If the value is numeric, there is no requirement for its representation, it would just be able to be converted by a string to numeric algorithm.

`dataset:ValueUncertainty`: This, if used, is the uncertainty of the value, of course in the same units as the value. If there is no uncertainty, then this value is zero.

`dataset:ParameterSpecification`: This is a record, meaning several components, giving the specification of the value.

The specification of the value is defined by `dataset:ParameterSpecification`. This specification is used not only within the parameter definition, but also to specify what type of data is expected in other data structures. The specification is made of the following:

`dataset:ParameterLabel`: This is a string keyword or label given to this parameter value. In a matrix, this would be the column name.

`dataset:ParameterTypeSpecification`: This is the `qudt:QuantityKind`, of the QUDT ontology, specifying the type of unit, for example, `qudt:MolarEnergy` in the above example.

`dataset:ValueUnits`: This is the QUDT label for the specific unir of the `qudt:QuantityKind`. For example, for `qudt:MolarEnergy`, a specific instance could be `qudt:CAL-PER-MOL`.

`dataset:DataPointUncertainty`: This is a classification parameter and specifies the type of uncertainty is given.

3 RESULTS: EXPERIENCE

All the above concepts using have been implemented in JThermodynamicsCloud. During the simultaneous development of the ontology and the application, distinct advantages of the approach were realized.

First and foremost, one of the greatest advantages of using the ontology is having the documentation of the data classes centralized and, furthermore, categorized (meaning the class structure of the ontology definitions). This is not only convenient, but almost essential considering that the data structures are JSON-like and free format. Furthermore, JThermodynamicsCloud is made of three distinct subsystems, the GUI interface in Angular, the backend in JAVA and the database in Google Firestore and without the centralized data structure definition offered by the ontology, developmental programming and the (programmed) communication between these subsystems would have been more difficult and error-prone.

The use of the ontology in the definition of the RESTful services, both for the services and for the transactions, provides a clear documentation of what inputs and what outputs are to be expected for the RESTful services in a machine and human readable format. Without this, the only specification is within the backend code itself.

In terms of automation, the specification of the database hierarchy is extremely efficient and convenient. Where catalog objects are stored in the hierarchy of the noSQL document based database is handled completely by the ontology specification. If a new catalog data object is to be created, this only need be specified in the ontology. This is done by placing the class name of the new catalog object in the ontology hierarchy and, if necessary, define the labeling of the branches leading to the class in the hierarchy. This also means that the structure of the database can be changed purely by changing the ontology specification. No (JAVA) programming is involved in the backend.

The use of the QUDT ontology for units increases flexibility and efficiency in unit manipulation. Once again all the domain knowledge of units in the ontology and any changes in units is done through the ontology without modification of the code. In terms of database management and the results in the calculations of the system, the added flexibility of unit conversion when needed eliminates the restriction that the units of the parameters stored in the database have to be ‘standardized’. Elimination of this restriction reduces, or at least facilitates the detection of, errors in the database. This is particularly important for the combustion community where there two standards for units of energy. Which is preferred is more a personal preference of the researcher involved.

4 CONCLUSIONS

This paper has described the use of ontologies in the model driven development of JThermodynamicsCloud. The ontology serves is platform independent description of the data structures and the operational structures that are used in the three distinct platforms of the system, namely the interface (written in Angular), the backend (written in JAVA and accessed through RESTful services) and the database (the noSQL database of Google Firestore). Though the ontology was not used to generate code for the application, the ontology is queried extensively (using the SPARQL queries of the JENA API of the JAVA backend) to perform its tasks.

On github further information about JThermodynamicsCloud can be found:

https://github.com/blurock/Angular/releases

This link includes more detailed documentation (both from the ontology point of view, but also from a combustion application point of view), a link to the current ontology and a link to the working version of JThermodynamicsCloud on the Google Cloud platform.

REFERENCES

Angular. (n.d.). Retrieved September 5, 2023, from https://angular.io/

Apache. (n.d.). *Apache Jena—SPARQL Tutorial*. Retrieved April 30, 2019, from https://jena.apache.org/tutorials/sparql.html

Bhatia, M. P. S., & Beniwal, A. K. and R. (2016). Ontologies for Software Engineering: Past, Present and Future. *Indian Journal of Science and Technology*, *9*(9), 1–16. https://doi.org/10.17485/ijst/2016/v9i9/71384

Blurock, E. (2021). *Use of Ontologies in Chemical Kinetic Database CHEMCONNECT*. 240–247. https://www.scitepress.org/PublicationsDetail.aspx?ID=ai7xFiBEN8E=&t=1

Blurock, E. S. (2019, June 29). *CHEMCONNECT: An Onotology-Based Repository of Experimental Devices and Observations*. 8th International Conference on Advanced Information Technologies and Applications (ICAITA 2019), Copenhagen, Denmark. https://icaita2019.org/index.html#home

Bučko, B., Zábovská, K., & Zábovský, M. (2019). Ontology as a Modeling Tool within Model Driven Architecture Abstraction. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 1525–1530. https://doi.org/10.23919/MIPRO.2019.8756968

Cloud Firestore | Store and sync app data at global scale. (n.d.). Firebase. Retrieved September 5, 2023, from https://firebase.google.com/products/firestore

Cloud Firestore Data model. (n.d.). Firebase. Retrieved September 5, 2023, from https://firebase.google.com/docs/firestore/data-model

De Bruyn, P., Mannaert, H., Verelst, J., & Huysmans, P. (2018). Enabling Normalized Systems in Practice – Exploring a Modeling Approach. *Business & Information Systems Engineering*, *60*(1), 55–67. https://doi.org/10.1007/s12599-017-0510-4

Dublin Core. (2012, June 12). *Dublin Core Metadata Initiative*. Dublin Core Metadata Initiative. http://dublincore.org/

Espinoza-Arias, P., Garijo, D., & Corcho, O. (2021). Crossing the chasm between ontology engineering and application development: A survey. *Journal of Web Semantics*, *70*, 100655. https://doi.org/10.1016/j.websem.2021.100655

Gaševic, D., Djuric, D., & Devedžic, V. (2009). Model Driven Engineering. In V. Deved¿ic, D. Djuric, & D. Ga¿evic (Eds.), *Model Driven Engineering and Ontology Development* (pp. 125–155). Springer. https://doi.org/10.1007/978-3-642-00282-3_4

Maali, F., & Erickson, J. (2014, January 16). *Data Catalog Vocabulary (DCAT)*. https://www.w3.org/TR/vocab-dcat/

QUDT. (2018, December 16). *Quantities, Units, Dimensions and Data Types Ontologies*. Quantities,Units, Dimensions and Data Types Ontologies. http://qudt.org/

SI Brochure. (n.d.). BIPM. Retrieved September 5, 2023, from https://www.bipm.org/en/publications/si-brochure

Silva Parreiras, F., Walter, T., Wende, C., & Thomas, E. (2010). Bridging software languages and ontology technologies: Tutorial summary. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, 311–315. https://doi.org/10.1145/1869542.1869626

Suchánek, M., Mannaert, H., Uhnák, P., & Pergl, R. (2021). Towards Evolvable Ontology-Driven Development with Normalized Systems. In R. Ali, H. Kaindl, & L. A. Maciaszek (Eds.), *Evaluation of Novel Approaches to Software Engineering* (pp. 208–231). Springer International Publishing. https://doi.org/10.1007/978-3-030-70006-5_9

Timothy Lebo, Satya Sahoo, Deborah McGuinness. (2013). *PROV-O: The PROV Ontology*. https://www.w3.org/TR/prov-o/