

Univerza v Ljubljani
Fakulteta *za gradbeništvo*
in geodezijo



ARSENI KIRILLOV

APPLICATION DEVELOPMENT USING THE BRICK
FRAMEWORK

RAZVOJ PROGRAMSKE OPREME Z UPORABO BRICK
OGRODJA



European Master in
Building Information Modelling

Master thesis No.:

Supervisor: Assist. Prof. Matevž Dolenc, Ph.D.

Ljubljana, 2023



Co-funded by the
Erasmus+ Programme
of the European Union

ERRATA

Page	Line	Error	Correction
-------------	-------------	--------------	-------------------

»This page is intentionally blank«

BIBLIOGRAFSKO – DOKUMENTACIJSKA STRAN IN IZVLEČEK

UDK:	004.655.2:69(043.3)
Avtor:	Arsenii Kirillov
Mentor:	doc.dr. Matevž Dolenc
Somentor:	
Naslov:	Razvoj programske opreme z uporabo Brick ogrodja
Tip dokumenta:	Magistersko delo
Obseg in oprema:	54 str., 45 sl., 1 pregl.
Ključne besede:	Brick ogrodje, razvoj aplikacij, Python, analiza podatkov

Izveček: Nestrukturirani, časovno odvisni, podatki pomenijo izziv za razvoj prenosljivih in razširljivih programskih rešitev. V fazi uporabe grajenega sredstva so, kot rezultat prejšnjih faz gradbenega procesa, dostopni strukturirani in nestrukturirani podatki ter podatki, ki se ustvarjajo v fazi uporabe in se nanašajo na delovanje in vzdrževanje grajenega sredstva.

Za opis in upravljanje z raznolikimi podatkovnimi modeli pogosto uporabljamo RDF (*angl. Resource Description Framework*), za formalno opisovanje ontologij. Brick je odprto-kodna ontologija, ki opisuje podatkovni model upravljanja zgradb, vključno s sistemi za ogrevanje, prezračevanje in klimatske naprave.

V raziskovalni nalogi je prikazan razvoj programske rešitve z uporabo ogrodja Brick za analizo heterogenih podatkov v fazi uporabe grajenega sredstva. Povdarek pri razvoju programske rešitve je na uporabi odprto-kodne programske opreme in standardiziranih ali odprtih podatkovnih zapisih. Arhitekturna zasnova programske rešitve sledi smernicam sodobnih, razširljivih in porazdeljenim programskim sistemom, ki temeljijo na vsebnikih Docker.

»This page is intentionally blank«

BIBLIOGRAPHIC– DOKUMENTALISTIC INFORMATION AND ABSTRACT

UDC:	004.655.2:69(043.3)
Author:	Arsenii Kirillov
Supervisor:	doc.dr. Matevž Dolenc
Cosupervisor:	
Title:	Application development using the Brick framework
Document type:	Master Thesis
Scope and tools:	54p., 45 fig., 1 tab.
Keywords:	Brick Ontology, Application Development, Python, Data Analysis

Abstract:

Unstructured operational phase-related data is an obstacle on a way to portable and scalable building maintenance applications. For the development of any application from a mobile game to a complex distributed web system developer tools and standards are required. When it comes to the operational phase of construction, there is a lot of static data produced in previous stages and unstructured data for building maintenance which is usually handled by facility managers. One of the most efficient abstractions for handling heterogeneous maintenance data is a Resource Description Framework (RDF) in general and Brick ontology in particular. Brick is a community-driven and open-source ontology whose domain is in Building Management Systems and HVAC equipment. Structured by Brick maintenance data can be used in applications for energy analysis.

This master thesis is an attempt to build a data analysis application for the operational stage of construction that will use both static and structured maintenance data for visualizing the indoor environment. The domain of thesis research and tools that were used for development avoid any proprietary formats and software. The application itself was built in a self-sustainable way to keep the opportunity of including it in a complex distributed system open based on Docker containers.

»This page is intentionally blank«

ACKNOWLEDGEMENTS

I would like to express my gratitude to Professor Miguel Azenha for the opportunity to participate in the BIMA+ Master's journey and for his pedagogical talent, and to Maria Laura Leonardi for the guidance in developing a case study for this thesis.

I must also express my sincere appreciation to the Open Source BIM and IT community. I can point out the pure interest that comes up while reading articles from the Computer Science Department of the University of California Berkeley, the IfcOpenShell developers and the OSArch community, who are making really big and crucial efforts to push the BIM industry towards freedom from the proprietary world of software, and Guido Van Rossum and all the people who have contributed to the creation of such open and extensible programming language as Python, which can be used by people without a computer science education like me to express and test ideas.

I am also grateful to my supervisor Matevž Dolenc.

»This page is intentionally blank«

TABLE OF CONTENTS

ERRATA	II
BIBLIOGRAFSKO – DOKUMENTACIJSKA STRAN IN IZVLEČEK	IV
BIBLIOGRAPHIC– DOKUMENTALISTIC INFORMATION AND ABSTRACT	VI
ACKNOWLEDGEMENTS	VIII
TABLE OF CONTENTS	X
INDEX OF FIGURES	XII
INDEX OF TABLES	XIV
LIST OF ACRONYMS AND ABBREVIATIONS	XV
1 INTRODUCTION	1
1.1 Defining a Stage of Construction	4
1.2 Defining a Research Workflow	4
1.3 Brief Description by chapters	5
2 KEEPING STATIC BUILDING DATA STRUCTURED, ACCESSIBLE, AND INTEROPERABLE	7
2.1 IFC for storing HVAC relevant data	8
2.1.1 HVAC equipment	9
2.1.2 HVAC Data in IFC	10
2.2 Other Formats	11
3 KEEPING DYNAMIC BUILDING DATA STRUCTURED, ACCESSIBLE, AND INTEROPERABLE	13
3.1 Building Management System or BMS	13
3.3 BMS Protocols	15
3.3.1 BACnet	16
3.3.2 Modbus vs BACnet	19
3.4 Levels of Abstraction for BMS Representation	21
3.4.1 Project Haystack	21
3.4.2 Brick Schema as a Step to Ontologies	23

3.5	Linked Data for Buildings Applications	29
4	KEEPING APPLICATIONS INTEROPERABLE	32
4.1	Existing Brick Based Softwares	32
4.2	Docker	33
4.3	Mortar	35
4.4.	Brick Applications in Use	37
4.4.1	Detecting Passing Valves	37
4.4.2	Occupant Satisfaction	38
5	PROOF OF CONCEPT DEVELOPMENT	40
5.1	Sources of Data	41
5.1.1	ASHRAE Thermal Comfort Database	41
5.1.2	Brick Model	42
5.1.3	Data From Sensors	44
5.1.4	Static Building Data	45
5.2	Application's logic	45
5.2.1	Workflow	45
5.2.3	Key logical steps in Python	46
5.3	Application deployment	48
5.3.1	Application in a docker container	48
6	CONCLUSION	51
6.1	Main Conclusions	51
6.2	Future Work and Limitations	51
	REFERENCES	55

INDEX OF FIGURES

Figure 1: Sensors naming practice - Source (Detecting passing valves).....	1
Figure 2: Python Levels of Abstraction – Source (Own work).....	2
Figure 3: Serialization and Deserialization - Source (Developedia).....	3
Figure 4: RIBA Stages - Source: (Royal Institute of British Architects).....	4
Figure 5: Mind map of the thesis research - Source (Own)	5
Figure 6: BIM Maturity - Source (BibLus).....	7
Figure 7: Data Schema Architecture of IFC - Source (A Method to generate a Modular ifcOWL Ontology)	9
Figure 8: Typical VAV-based HVAC distribution system – Source (I Optimize Reality).....	10
Figure 9: BMS logic – Source (Own)	13
Figure 10: BACnet and OSI Architecture – Source (Own).....	17
Figure 11: BACnet Objects – Source (Own).....	18
Figure 12: BACnet Object Properties – Source (Own).....	18
Figure 13: BACnet Services – Source (Own)	19
Figure 14: ModBus typical message frame – Source (Own)	20
Figure 15: Building encoded in Haystack – Source: (Haystack website)	22
Figure 16: LOD Ranking – Source: Tim Berners-Lee	24
Figure 17: RDF Concept – Source (Own).....	24
Figure 18: SPARQL querying – Source (WordLift).....	25
Figure 19: SPARQL Select – Source (Own).....	25
Figure 20: Brick Classes Hierarchy and Relationships – Source (Na Luo et al., 2022).....	27
Figure 21: Comparison of different schemata for buildings – Source (Balaj et al., 2018).....	28
Figure 22: Updated Semantic Web Layer Cake – Source (Kingsley Uyi Idehen).....	29
Figure 23: Use case developed and use of metadata models – Source (Pritoni et al., 2021)	30
Figure 24: Docker - Source (Docker Website).....	33
Figure 25: Docker and Virtual Machine Architecture – Source (Own)	34
Figure 26: Docker Application Deploying – Source (Own)	35
Figure 27: Mortar architecture – Source (Fierro et al., 2018)	36
Figure 28: Left) schematic of a variable air volume (VAV) with reheat terminal unit and right) a Brick data model of the VAV terminal unit – Source (Duarte Roa et al., 2022)	37
Figure 29: Experimental setup of the commercial office space used in the experiment – Source (Cory Mosiman et al., 2021).....	39
Figure 30: Schema of Proof Of Concept – Source (Own).....	40
Figure 31: Insertion of geographical data into ASHRAE Database – Source (Own, Jupyter screenshot)	42

Figure 32: Brick model development. Defining a graph – Source (Own, Nvim screenshot).....	43
Figure 33: Brick model development. Creating a triple – Source (Own, Nvim screenshot).....	43
Figure 34: Brick Model Development. Adding relationship triple – Source (Own, Nvim screenshot)	44
Figure 35: Brick model visualization – Source (Own).....	44
Figure 36: Extracting geographical data from IFC – Source (Own, Nvim screenshot)	45
Figure 37: Git Workflow – Source (GitHub)	46
Figure 38: Applied SPARQL query – Source (Own, Nvim screenshot).....	47
Figure 39: Application frontend 1 – Source (Own).....	47
Figure 40: Heatmap – Source (Own).....	48
Figure 41: Docker File – Source (Own, Nvim screenshot)	49
Figure 42: Shell Script – Source (Own, Nvim screenshot)	49
Figure 43: Application Structure – Source (Own).....	50
Figure 44: BACnet Integration – Source (Own)	52
Figure 45: BACnet data pulling Python script (IP addresses changed) – Source (Own, Nvim screenshot)	53

INDEX OF TABLES

Table 1: BACnet and Modbus comparison	20
---	----

LIST OF ACRONYMS AND ABBREVIATIONS

AFDD	Automation Fault Detection and Diagnostics
AHU	Air Handling Unit
ANSI	American National Standards Institute
ASHRAE	The American Society of Heating, Refrigerating and Air-Conditioning Engineers
AWS	Amazon Web Services
BIM	Building Information Modelling
BMS	Building Management System
BOT	Building Ontology Topology
CDE	Common Data Environment
CI/CD	Continuous Integration Continuous Delivery
CLI	Command Line Interface
CSV	Comma-separated Values
GUI	Graphical User Interface
HTO	Haystack Tagging Ontology
HTTP	Hyper Text Transfer Protocol
HVAC	Heating Ventilation and Air Conditioning
IFC	Industry Foundation Class
IP	Internet Protocol
IRI	Internationalized Resource Identifier
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LOD	Linked Open Data
LXC	Linux Container

MORTAR	Modular Open Reproducible Testbed for Analysis and Research
MSTP	Master Slave Transfer Protocol
OGC	Open Geospatial Consortium
OS	Operating System
OWL	Web Ontology Language
RDF	Resource Description Framework
RIBA	Royal Institute of British Architects
RTU	Remote Terminal Unit
SAREF	Smart Applications Reference Ontology
SET	Standard Effective Temperature
SHACL	Shapes Constraint Language
SPARQL triplestores	Standard Query Language and Protocol for Linked Open Data on the web or for RDF
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
VAV	Variable Air Volume Box
VRF	Variable Refrigerant Flow System
VM	Virtual Machine
W3C	World Wide Web Consortium
XML	Extensible Markup Language

1 INTRODUCTION

Buildings constitute 32% of the energy and 51% of electricity demand worldwide (Balaji et al., 2018). Therefore, optimizing energy consumption is essential for reducing carbon emissions and operational expenses. Modern buildings and systems are interconnected with devices and sensors, enabling centralized operation and management. These controlling sensors contribute a network that requires a digital representation for each. The digital twin is comprised of structured data on both the sensors and the building systems, along with life-streaming data, and enables safe and efficient facility maintenance. However, extensive amounts of static data are also generated during earlier stages of construction. As a consequence, during the construction's operational phase, there exists an abundance of dissimilar data that does not contribute to the creation of a digital twin (Mavrokapnidis et al., 2023).

Modern building management systems (BMS) must operate with unstructured and complex data. An example of this type of data is illustrated in Figure 1. It is important to note that even in the case of the sensor, which has a lengthy title, certain key components of the equipment are not explicitly mentioned, but rather implied. Lack of standardisation of data during the operational phase of construction restricts developers from creating energy analysis applications that could significantly enhance building efficiency, primarily because of the efforts required to map all of this data into a common format. Mapping can turn out to be a costly and complicated task (Balaji et al., 2018).

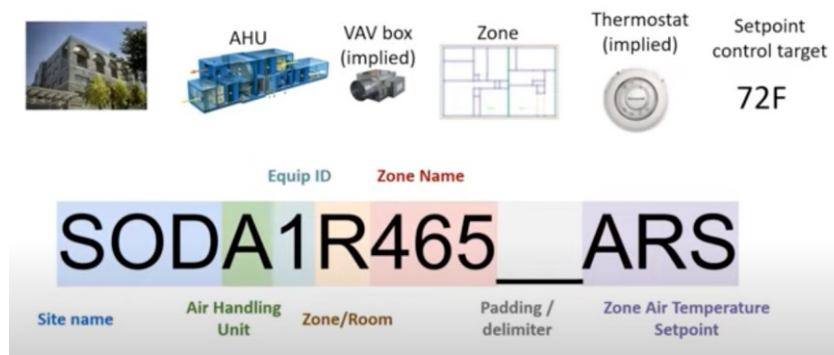


Figure 1: Sensors naming practice - Source (Detecting passing valves)

Unlike platforms that allow developers or enthusiasts to build on a defined structure, such as those available for mobile development that provide a range of programming languages and tools to facilitate the development process.

Naming conventions or development tools can be considered as a level of abstraction, which exerts a significant influence on various industries. Although problems can be resolved at a lower level of abstraction, managing a large number of objects can consume substantial resources, while a higher degree of abstraction enables the saving of time and resources. An abstract layer can keep many sub-processes implicit for the user. This particular technique is widely used in the IT industry. For example,

the Python programming language, which will be mentioned frequently throughout this thesis, is written in the C programming language. Python code is interpreted into bytecode during program execution, although a Python programmer may not be aware of this. This indicates that Python is an abstraction built over bytecode implemented using the C programming language, while Python itself is also a lower level abstraction for a Python-based library for working with Pandas datasets. Selecting the appropriate level of abstraction for a given task is essential.



Figure 2: Python Levels of Abstraction – Source (Own work)

The problem is that the abstraction of naming convention illustrated in figure 1 is human-readable, with some level of components implied. The experienced facility manager is aware of the existence of these implicit components, but the computer that is supposed to work with them is not. This brings the idea of a machine-readable abstraction for the representation of heterogeneous building data.

Any data produced during a program's lifecycle, intended for future use or distribution, must be transformed into a specific, appropriate format. This involves the use of specific algorithms to put a complex data structure into a file, a process known as serialisation (CIS, 2020). The example of serialisation is illustrated in Figure 3.

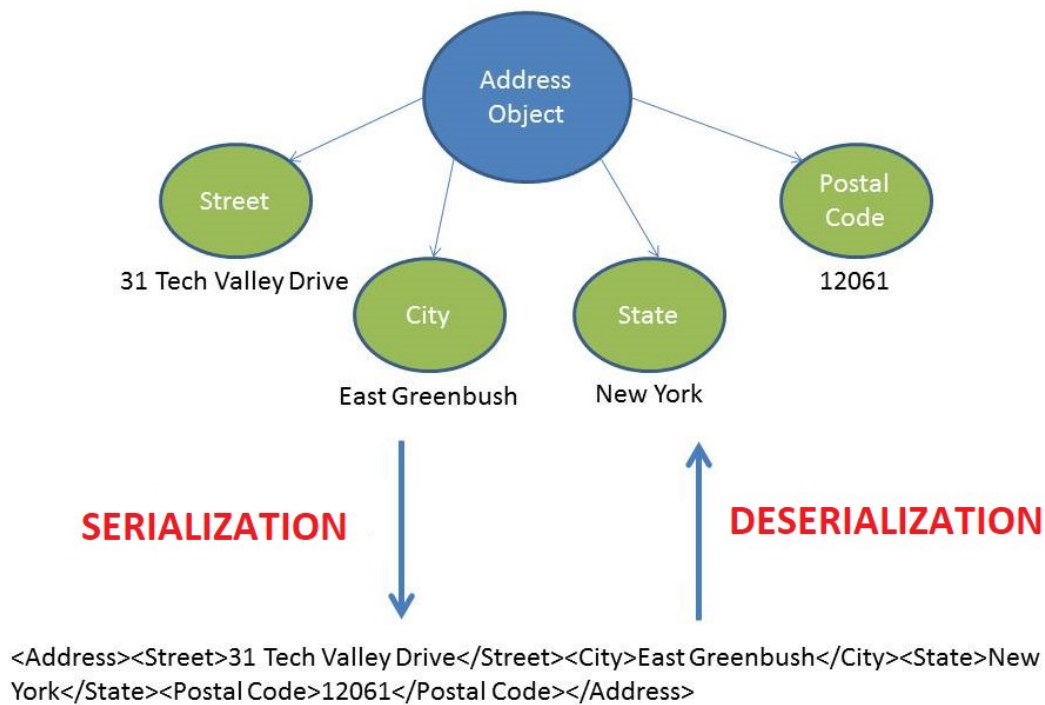


Figure 3: Serialization and Deserialization - Source (Developedia)

If the sensor naming convention illustrated in Figure 2 is considered, it is also a serialization performed by humans into a human-readable format. This type of serialization introduces the problem of chaotic data into the operational phase of the construction. However, if such data was serialised by a machine into a machine-readable format, it would help to overcome this problem.

One of the most effective tools for storing heterogeneous data is the Resource Description Framework (RDF). Brick Ontology is an abstraction built on top of RDF with a focus on BMS and HVAC systems. Brick is capable of modelling equipment, physical entities utilised by such equipment, equipment locations, and the relationships among them. This abstraction enables the structuring of heterogeneous data, which facilitates the development of portable and scalable applications for energy analysis and optimisation.

The scope of thesis research and case study application will cover the following:

- Methods for storing fixed building data in an unrestricted and accessible manner
- Structuring of heterogeneous and dynamic livestreaming data using brick ontology
- Development of a mini application utilizing static and dynamic data

1.1 Defining a Stage of Construction

It is important to define a stage of construction in the area of the research for the dissertation. In this regard, it is relevant to use the Royal Institute of British Architects (RIBA) work plan (Figure 4). The research will primarily concentrate on the 7th stage - Use.

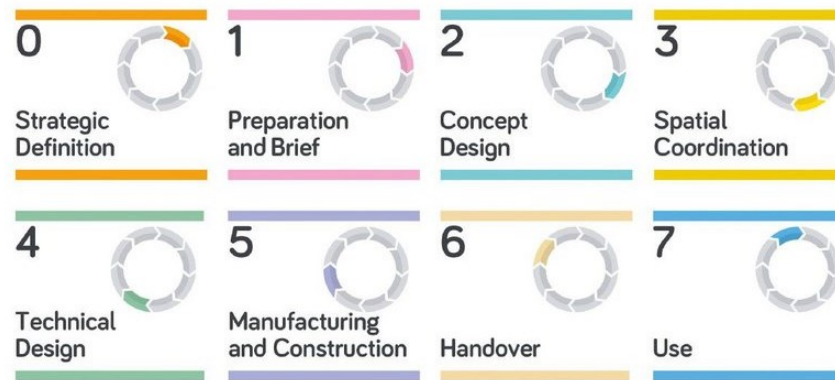


Figure 4: RIBA Stages - Source: (Royal Institute of British Architects)

According to the RIBA plan, the objective of this stage is to enable the smooth operation and management of a building. The primary responsibilities of this phase include:

- Implement Facilities Management and Asset Management
- Undertake post-occupancy Evaluation of building performance in use
- Verify Project Outcomes including Sustainability Outcomes

The main requirement for Stage 7 of the statutory Process is to comply with planning conditions. At the end of this stage, information exchange should include feedback from post-occupancy evaluation, as well as an updated building manual containing the health and safety file and fire safety information (CMS Group, 2023).

1.2 Defining a Research Workflow

The primary goal of this thesis is to develop an application that utilises static and dynamic data and is scalable. To achieve this objective, it was imperative to have convenient access to all case studies implementing the same technology and relevant theoretical articles and notes. The Obsidian Notebook was chosen for this purpose as it enables the storage of all files in a single folder and linking them to each other. The research's outcomes are presented in a mind map (see Figure 5). This mind map was used to identify several technologies that should be used for application development by filtering various links to a specific article.

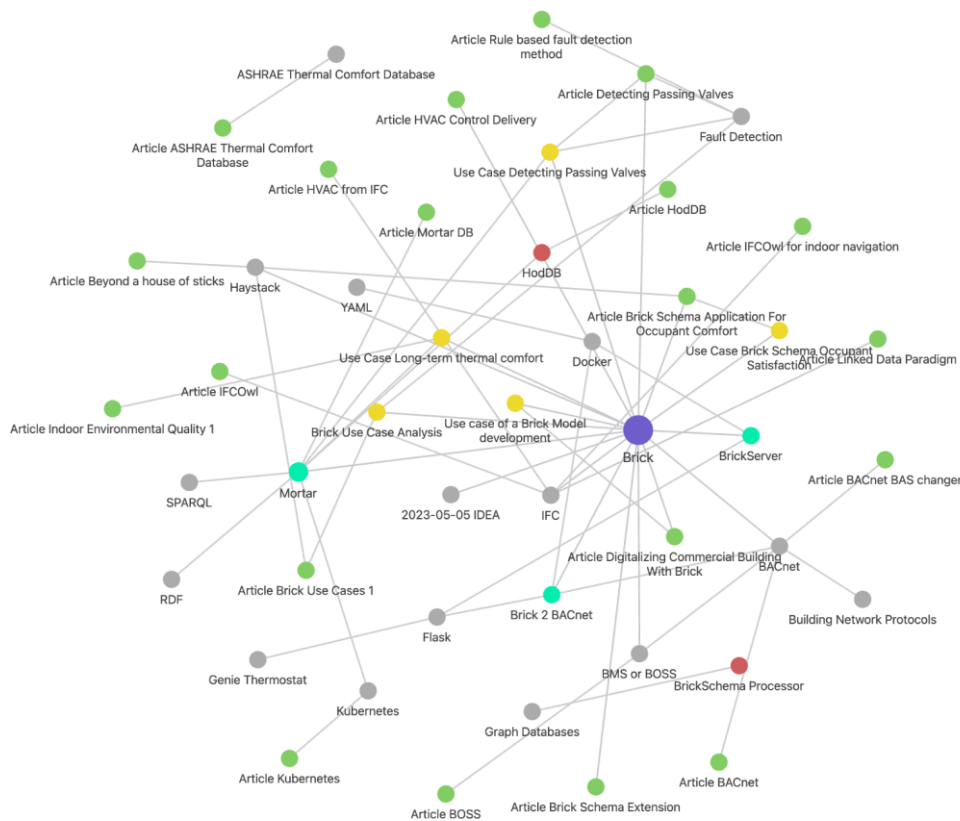


Figure 5: Mind map of the thesis research - Source (Own)

As for the application's development, it followed the open-source paradigm. BIM is a relatively young industry, highly reliant on standards proposed by major corporations. Every software and physical equipment vendor seeks to keep its customers within its own environment. Therefore, the critical mass of companies using BIM is stuck in closed formats and chaotic data formats. Therefore, it was crucial to avoid any proprietary software, otherwise, achieving scalability would not be possible.

1.3 Brief Description by chapters

Every chapter in this thesis has the purpose of introducing key concepts that were used for application development or were used by people to develop similar software:

- The second chapter describes open standards for storing static building data with a focus on IFC.
- The third chapter describes abstraction layers that are used for the representation of building management systems and sensors with a focus on open-source ontologies and Brick in particular.
- The fourth chapter introduces an approach for application development to make it executable on every platform and even as a part of bigger software architecture with a focus on Docker containerization.

- The fifth chapter describes the steps, tools, and strategy of development of the described application.
- The sixth chapter or conclusion describes not implemented parts of the software, contains the roadmap for further development, and key conclusions that were made during the development process.

2 KEEPING STATIC BUILDING DATA STRUCTURED, ACCESSIBLE, AND INTEROPERABLE

Progress in the BIM industry brings with itself a variety of softwares and encoding standards. It makes the process of data transferring from one stage to another quite complicated (BibLus, 2023). To overcome such problems, Building Smart offers a set of tools called Open BIM (Building Smart, 2020). This set of tools is supposed to improve data accessibility and interoperability. All these tools are supposed to be independent from software vendors and open.

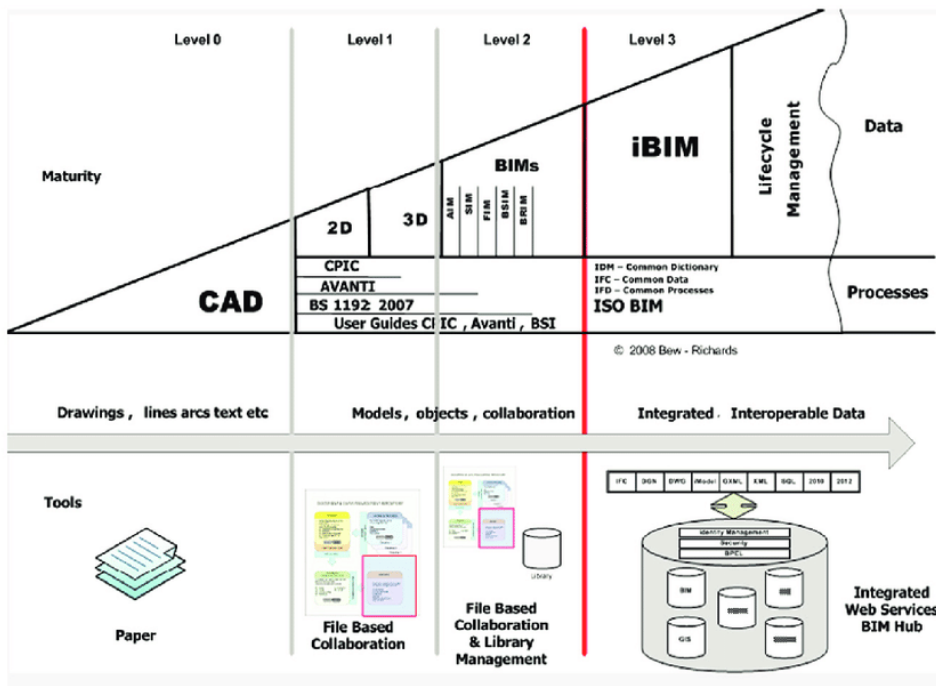


Figure 6: BIM Maturity - Source (BibLus)

In the list of the most famous Open BIM tools can be included such formats as IFC, BCF, COBie, CityGML, gbXML, etc. The most relevant format according to the area of research is an IFC or Industry Foundation Class. This format allows to store geometrical data relatively accurately and metadata about the instances of a building. IFC is an open format and is supported by the majority of popular BIM softwares and is an ISO certified standard.

Opposite site of Open BIM that is offered by Building Smart is total control of the committee. The Building Smart committee is responsible for choosing the direction where IFC classes will be extended. This means that IFC is not community driven and cannot match fully the needs of industry. However, IFC solves a very complex task of storing both geometrical data and metadata and has to be accurate enough due to the risks that exist in the construction in general. If such a complex thing was fully “forkable”, those forked versions would cause misunderstanding, version conflicts and other serious

consequences. Anyway, it has to be admitted that IFC is probably the best software neutral open format for BIM nowadays.

2.1 IFC for storing HVAC relevant data

Any data has to be modelled according to some set of rules. Any data modelling language is supposed to define some entities of data and relationships between them. In the construction industry EXPRESS data modelling language is quite popular. This language is also certified by ISO 10303. Any data that is modelled according to EXPRESS language can be represented in both text and graphical way. Second way of data interpretation is called EXPRESS-G.

Industry Foundation Class (IFC) is certified by ISO open standard that was developed to store BIM data in an interoperable way. Usually, IFC is represented in EXPRESS schema, but it can also be represented in XML. Unfortunately, this format wasn't widely applied for storing data from the operational stage or any non-static data in general. But this format can be used to store static data about building and equipment installed in it.

Data Schema Architecture of IFC consists of 4 layers (Figure 7): *Resource layer*, *Core layer*, *Interoperability layer*, and *Domain layer*. Core layer contains various generic entities or resources that are not necessarily applied to the buildings. However, all other layers will reference this layer. Core layer contains abstract concepts such as space or location, that are used to define entities. This layer is self-sustainable, so it can exist even if not referenced by any other upper layer. Core layer also contains the base class "IFCRoot" that other classes use to inherit from. The "IFCRoot" can be characterized as a superclass that allows all classes that inherit from him to be self-sustainable independent classes, whereas classes that inherit from Resource Layer directly cannot be used independently. This superclass contains 4 attribute definitions:

- GlobalId or "*IfcGloballyUniqueId*" that stands for assignment of unique ID
- Owner History of "*IfcOwnerHistory*" that is optional and contains information about ownership
- Name or "*IfcLabel*" that stands for optional name that can be used by users or specific softwares
- Description or "*IfcText*" that contains optional comments for exchange

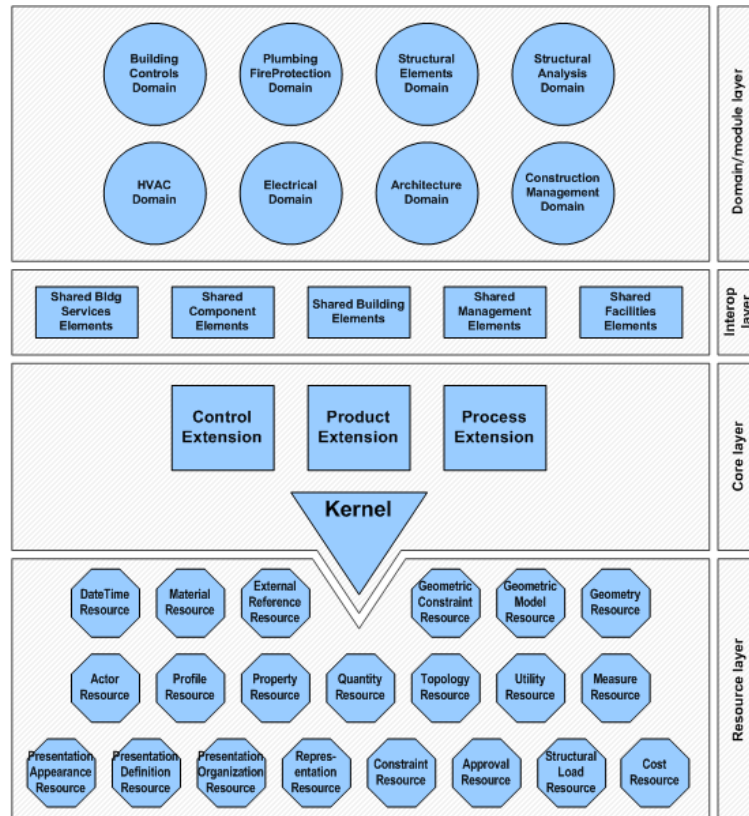


Figure 7: Data Schema Architecture of IFC - Source (A Method to generate a Modular ifcOWL Ontology)

The Interoperability layer provides more specialized entities that can be both objects and relationships. It contains such elements as “*IfcDoor*”, “*IfcSlab*” etc. Highest layer of IFC Architecture is a Domain layer that contains specific information related to one of the domains like architecture or HVAC systems.

2.1.1 HVAC equipment

After main concepts of storing building-related data in IFC open standard are defined, it must be clarified which type of data is the most relevant for an application. Scope of the thesis is the operational stage of construction, HVAC systems, and Building Management Systems (BMS). Since BMS-related data cannot be stored in IFC properly, this paragraph will keep focus on HVAC equipment.

Heating Ventilation and Air Conditioning (HVAC) system is a complex system that keeps the indoor environment of a building comfortable for occupants. HVAC systems can measure and affect various parameters such as temperature, humidity, air quality etc. It consists of various types of equipment that work under controlling logic. Simplified HVAC system is illustrated in a Figure 8.

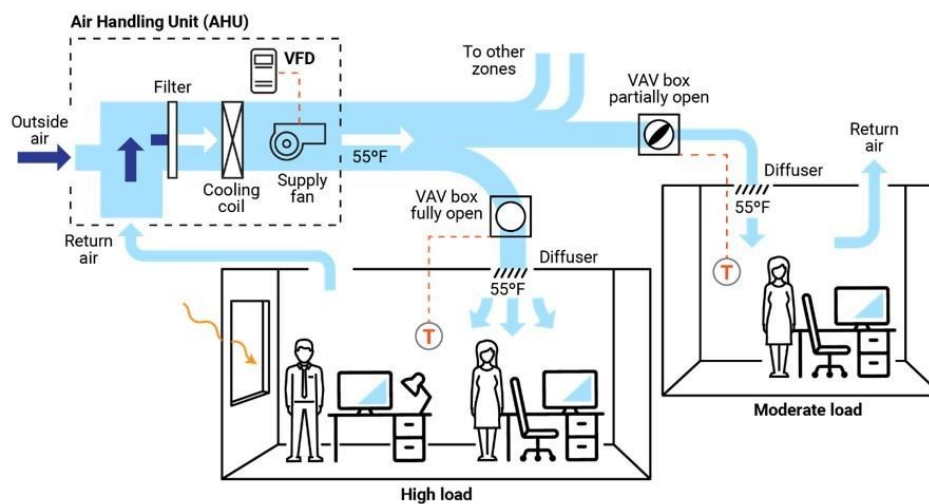


Figure 8: Typical VAV-based HVAC distribution system – Source (I Optimize Reality)

Air Handling Unit (AHU) is used to supply buildings with fresh air and return used air back to the atmosphere. AHU uses different technologies of air filters, humidifiers, heaters, coolers etc. FAHU (Fresh Air Handling Unit) is used to supply only fresh air to the building while AHU can recirculate used air to save energy. Incoming air is supposed to be filtered, measured, and heated or cooled.

Variable Air Volume System (VAV) is an upgraded version of CAV (Constant Air Volume System) that regulates the flow of air that is supplied to a zone. There are two ways to control temperature in a room with VAV. It can change flow intensity and change temperature in a heater. So, if the temperature is too high, the flow of cold supplied air must be more intense. Also, the opposite, if it's cold inside the temperature of the heater can be increased and flow reduced. VAV has a sensor installed that analyses temperature in a zone. Control damper is used to increase or reduce supplied air flow. In case of many zones that are controlled with VAV's pressure of supplied air can decrease if the VAV is far from AHU that pushes air. Therefore, there is a need to install a pressure sensor that can notify if pressure is low, and AHU will increase it.

Usually, HVAC systems are much more complex and contain much more equipment. But according to the scope of this thesis, just these 2 crucial HVAC parts will be used for application development, therefore focus is on them.

2.1.2 HVAC Data in IFC

Approach of storing data in IFC is complex and limitations of format are crucial when it comes to storing dynamic data. Transforming IFC to IFCOWL is not efficient because of the huge amount of data that has to be stored in RDF triples. This requires much more memory to be stored and resources to work with that data. In a case study of converting IFC to existing Building Ontology Topology (BOT) and merging it with Brick (Mavrokapnidis et al., 2023). Brick data model covered BMS and time-series data

because IFC format can't cover this aspect, whereas BOT contained all relevant data from IFC. Research team concluded that there is a huge limitation from the side of IFC because authoring tools in general and Revit in particular can export incorrect spatial information to IFC, which is a very crucial aspect when it comes to location of sensors and controlling systems.

HVAC as an important part of the MEP system can be stored in IFC. In the research (et al., H. Lia and J. Zhang 2022) buildings that were encoded in IFC and on top of each it was performed an analysis, it was defined that such components as boilers or chillers had a definition of *IfcEnergyConversionDevice*, all connecting elements like pipes or ducts were defined as *IfcFlowSegments* and *IfcFlowFitting*. Such equipment as pumps or fans that are supposed to circulate flows had a definition of *IfcFlowMovingDevice*, diffusers had a definition of *IfcFlowTerminal*. Most common elements were connecting elements such as pipes, ducts, and diffusers. Also it has to be noticed that for every HVAC terminal it is important to understand a thermal zone that it serves. In IFC this type of data is supposed to be stored in *IfcRelContainedSpatialStructure* that consists of *RelatedElements* and *RelatingStructure*. It means that all data can be extracted by iterating through every entity instance. Methodology of extracting HVAC data from IFC was complex due to identifying HVAC system type, loops and all related components, and relations between components in every loop.

As a result of research, it was developed an algorithm that was able to extract HVAC systems from IFC but it had some limitations

- lack of comprehensive industry standards for classifying IFC entity instances. IFC format is not rigorous enough to store components related to water and air supply separately.
- IFC is not able to store all data that is required for BEM such as schedules.

Regardless of all limitations and complexity IFC seems to be the best open format on the market nowadays that can store static building-related data. Dynamic data limitations are supposed to be overcome by other tool, whereas IFC will be used as a source of static information.

2.2 Other Formats

For the development of case studies, all proprietary closed formats are avoided. Thus, a format such as Autodesk Revit's ".rvt" is out of scope, regardless of its popularity. This format and any changes made to it can only be visualised using a proprietary Autodesk software, which limits the scalability of an application. Building Smart formats like gbXML, on the other hand, have a specific purpose. For example, gbXML transfers data from a BIM model to energy analysis software (Green Building XML, 2023). The same applies to open formats from other communities like CityJSON, which is a standard of the Open Geospatial Consortium (OGC, 2023) with a specific purpose. CityJSON is a JSON-based format used to develop digital twins (CityJSON, 2023). Its main objective is to accurately locate objects

and geometry. However, it is not capable of storing as much information as IFC and lacks specific classes for representing HVAC systems. As the aim of this chapter is to identify a suitable data source format for the application, it is preferable to use a more general format like IFC.

3 KEEPING DYNAMIC BUILDING DATA STRUCTURED, ACCESSIBLE, AND INTEROPERABLE

IFC is a preferable format for storing static data, but it may not be the best option for dynamic data due to its complex structure. Nevertheless, there is a substantial amount of sensor data that can be subject to frequent changes or be sourced from various vendors. It has been suggested that RDF format would be an appropriate encoding for storing this type of data, but first there needs to be defined abstraction layers that represent BMS data in a digital world.

3.1 Building Management System or BMS

Every facility is supposed to be maintained and maintained efficiently. For this purpose, Building Management Systems (BMS) are used. BMS must control various systems of the building like mechanical, electrical, etc. BMS can also control security, fire safety systems, and elevators. Simplified and decomposed logic of BMS can be represented as relationships between input device, controller, and controlled device.

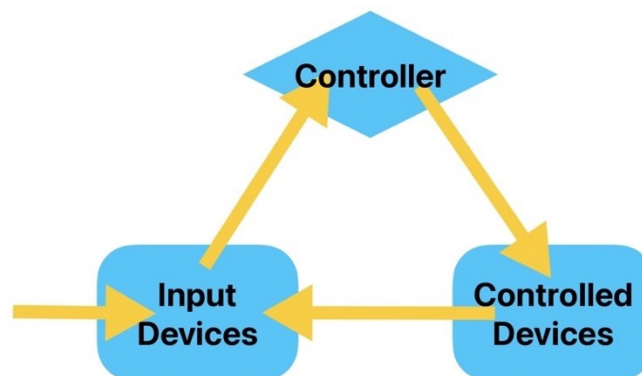


Figure 9: BMS logic – Source (Own)

- Data must be measured by sensors and provided as input to the system. This data could be temperature, humidity, pressure, airflow.
- Measured data has to be compared to a set of desired outcomes or instructions
- Output is produced based on measured data to change or maintain the environment

The building's HVAC systems comprise various components, including multiple AHUs, VAVs that supply the building with fresh air, water pumps, pipes, security, and fire systems. To ensure proper functioning, all these systems and subsystems require closed control loops consisting of sensor/actuator pairs. Programmable Logic Controllers (PLC) are usually employed for logic control, and they are hard-wired to the sensor/actuator pair. This method of control is frequently used as it minimises the quantity

of equipment and connections needed, although it is unable to effectively coordinate controllers and is limited by difficult-to-overcome hard boundaries.

3.2 BMS Architecture

BMS systems currently used operate at a low level of abstraction, with direct access only available for specific vendor sensors. As a result, scalability of BMS systems is limited. BMS systems currently used operate at a low level of abstraction, with direct access only available for specific vendor sensors. To address this issue, additional layers of abstraction are required to make BMS more adaptable and programmable. Researchers at the University of California, Berkeley have proposed several layers to achieve this (Dawson-Haggerty et al.).

Hardware Presentation Layer (HPL). Every BMS consists of numerous sensors, controllers, actuators, and linking components. There is a challenge to map all those physical entities into virtual digital entities. This process must be performed at a low level. This level is called HPL (Hardware Presentation Layer) and it uses the self-describing protocol to overcome the mentioned challenges. HPL abstracts physical entities into points that are supposed to produce time-series data or time-stamped sequences of values. This layer also includes:

- Naming - each point has to have global unique identifier
- Metadata - that helps to in a process of data interpretation
- Buffering and Leasing - that is used to overcome possible mistakes, missing data and can make system more fault tolerant
- Discovery and Aggregation - helps to aggregate sensors into a single source

Hardware Level Abstraction (HAL) enables the examination of different aspects such as layouts, mechanical and electrical systems, weather, and control logic at a higher abstraction level. HAL conducts inspections using a query language that is based on entity relationships. Abbreviations of technical terms will be explained upon first use. The query language enables searches based on criteria such as HVAC, spatial, and electrical, among others. In addition, HAL facilitates the implementation of control logic by incorporating drivers that provide the interface. Those drives enable the creation of commands and the implementation of control loops over HPL. Typically, sensors within a system do not store the data they generate. This data could be useful for machine learning, aiding fault detection, data visualisation, etc. Thus, the most challenging task is to efficiently store time-series data and allow applications to access it in near-real time.

Time Series Service (TSS) provides an interface that will allow to access huge amounts of data with a low latency and at different granularities. This process can be divided into two parts: stream selection

language and data transformation language where the first part is responsible for fast access to the data and the second is responsible for data cleaning.

Any change that is performed in a system must be under control. Especially in case of incorrect work of the system that can leave some data in an uncertain state. This problem can be solved via a transaction mechanism. **Transactions** ensure consistency of changes that are performed while modifying pieces of any state. Transactions must be integrated with HAL which can translate high-level requests into point-level operations such as read, write, and lock. To implement this process control transactions, require the following mechanisms:

- lease time - that sets time during which performed changes are valid
- revert sequence - mechanism of cancelling performed actions
- error policy - rules in case of partial failure

Resource of BMS has limitations therefore there must be defined rules of access. Authorization service is based on the approximate query language of HAL. The first part of the workflow is to check permission in the HAL, second action is performed in HPL which is a prove/verify process. It can be defined steps to be done forward to a smarter BMS:

- Higher level of abstraction can perform more precise control. For instance, if there is a need to set exact setpoint is inefficient because it will make nearly every zone to be always heated all cooled.
- Occupant satisfaction can be higher in case occupant is able to affect environment with personalized control application
- Having a context of data that was created by points can make system more predictable and efficient. This means that real time information can be coupled with semantic data in an analysable way.

3.3 BMS Protocols

Once all the systems are installed, problems can arise when they work independently, utilise diverse control interfaces, and are managed by multiple facility managers. This complicates the development of applications that depend on data-driven energy optimisation due to inaccessibility of some data. To overcome this challenge, various protocols have been developed.

3.3.1 BACnet

Any protocol is a set of rules for communication between systems. BACnet (Building Automation Control Network) is an ISO and ANSI standard protocol for achieving interoperability between the various systems within a building. It was created by ASHRAE (American Society of Heating, Refrigerating and Air Conditioning Engineers) in 1987. BACnet has been evolving through an open-consensus process, allowing for free participation without any fees.

According to Open System Interconnection (OSI, 2023) 7 layers of abstraction can be used by computers to connect (Figure 10). This concept was introduced in 1984 by representatives from the biggest companies of that time and was certified by ISO. However, nowadays modern networks use a simplified TCP/IP (Transmission Control Protocol) model. OSI layers are represented by:

- 1) *Physical layer* that defines types of connection that can be physical cable or wireless. This layer also executes bitrate control.
- 2) *Data link layer* that defines connection between two entities on the network. This layer could be decomposed into 2 parts: Logical Link Control (LLC), which keeps under control network protocols, error control, and synchronization and Media Access Control (MAC) that is responsible for permissions.
- 3) *Network layer* which functionality also could be decomposed into 2 main tasks: breaking up information into transferable entities and reassembling them after and finding most effective ways for data transferring across network
- 4) *Transport Layer* which is responsible for supplying session layer with valid data. It keeps rates, error checks, ensures that data was received, etc.
- 5) *Session Layer* that establishes connection channels between devices. This layer is also responsible for keeping connection open, when it is requested and ensure that it was closed after.
- 6) *Presentation Layer* that prepares data for the application layer. It defines encoding and encryption of data and compares received data to ensure that there was no mistakes.
- 7) *Application Layer* that is used by end-user software. It provides a list of protocols that allows user to exchange meaningful data with each other. This layer includes such protocols as Hyper Text Transfer Protocol (HTTP), Domain Name System (DNS), File Transfer Protocol (FTP), etc.

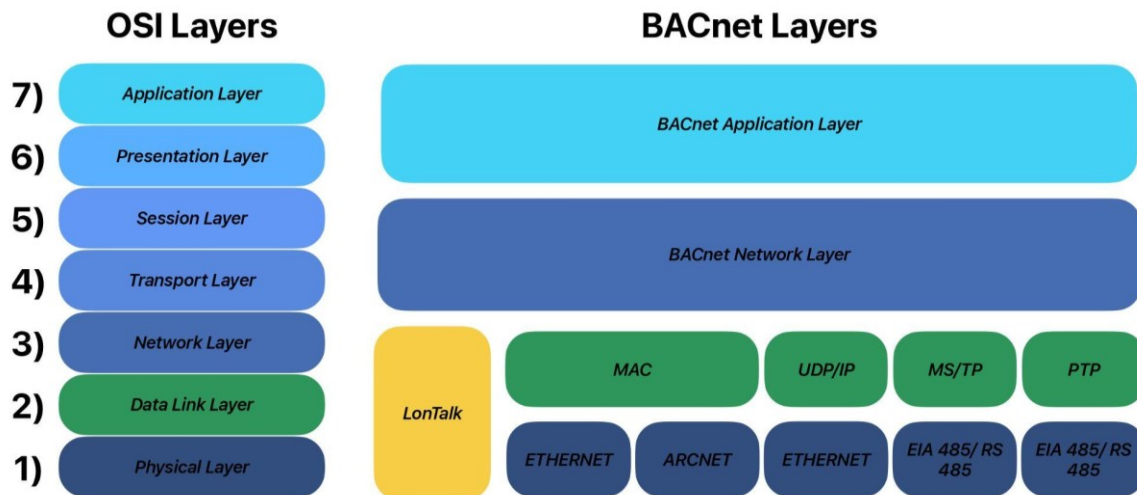


Figure 10: BACnet and OSI Architecture – Source (Own)

BACnet is based on 4 OSI layers: Physical, Data Link, Network, Application (Chipkin, 2023).

- 1) Physical Layer: means of connecting devices, transmitting electronics signals to convey data, defines hardware specifications, data transmission and reception, topology and physical network design.
- 2) Data Link Layer: data to frames or packets, rules for addressing, error-checking/ network access, flow control, presentation, message format.
- 3) Network layer: routing BACnet messages from one network to another.
- 4) Application layer: message processing and device addressing, interfaces with application software, be responsible for transport and session layer, BACnet objects and properties are defined.

BACnet cannot replace control logic, or the programming of devices, but it provides networking options. BACnet utilises an object-oriented approach and contains a library of 54 standard objects. This library can also be extended by the user. Moreover, BACnet presents comprehensive application services that may be applied to support building devices. These services are categorised into areas like scheduling, networking, accessing, etc. BACnet can be viewed as having three key components. Figure 11 illustrates Objects that define methods of information representation, Services that describe actions such as requests or methods for interoperation, and Transport Systems that define ways of physical connection between entities.

BACnet devices are utilised to support the BACnet protocol, incorporating a microprocessor-based controller alongside software. Each device comprises an object which represents metadata of the device,

including the input and output points it is intended to control. This device object is required to possess a unique identifier across the entire system, referred to as a device instance.

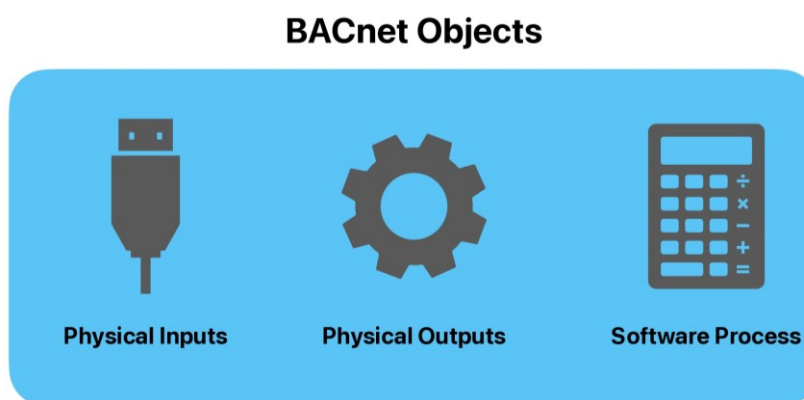


Figure 11: BACnet Objects – Source (Own)

BACnet Object (Figure 12) is responsible for:

- Object contains all information that can be used by BACnet.
- Object can represent information of a single entity and of collection of entities.
- Object has unique identifier that is a 32-bit binary number that contains information about object type and number of instances of object.
- Object has a list of properties that define an object. The minimal set for the property contains name and value

An object's functionality is defined by the collection of properties that it contains. The role of properties in the BACnet system is to allow other parts of the system to read information about objects or even write it if permissions allow. The total list of mandatory and optional properties is defined for each case.

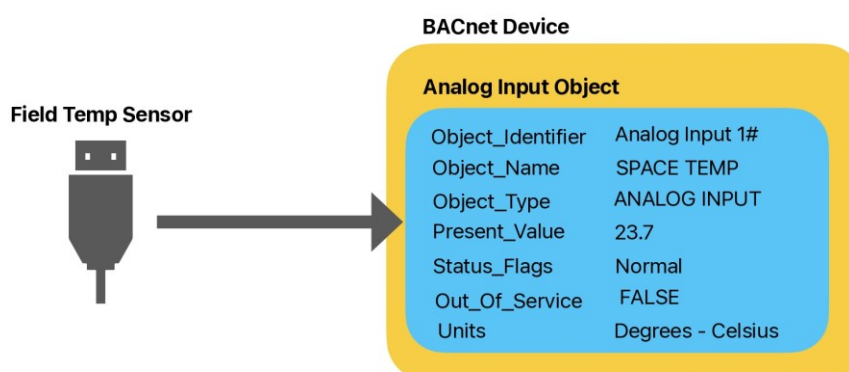


Figure 12: BACnet Object Properties – Source (Own)

Services are specific types of requests inside BACnet. There are 5 types of groups of service functionality:

- 1) object access that defines rights to read, write, delete, create
- 2) device management that sets timings of synchronization, rules for initialisation, backups and restoring database
- 3) alarm and event (changes of states)
- 4) file transfer
- 5) virtual terminal (frontend part of BACnet)

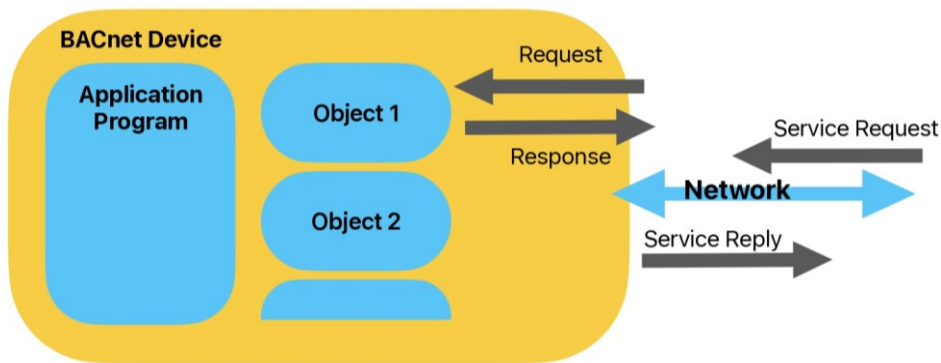


Figure 13: BACnet Services – Source (Own)

BACnet IP is a way of implementation of BACnet based on IP (Internet Protocol). BACnet IP uses UDP (User Datagram Protocol) instead of TCP. It defines 16 UDP ports: from 0xBAC0 (47808) to 0xBACF (47823) so each BACnet device is assigned an IP address. BACnet Mac address is a combination of an IP address and a UDP port. The same UDP port shall be used for all BACnet devices in a network. An example could be 192.168.1.10: 0xBAC0. For instance, 47808 is a port for BMS Devices and 47809 is a port for Alarm devices. In this case, two BACnet networks will exist on a single IP and will not be able to communicate and interrupt each other. BACnet network can consist of a single IP or be spread or can span across multiple physical or logical IP subnets.

3.3.2 Modbus vs BACnet

Modbus is a messaging protocol that is also prevalent in Programmable Controller Networks (PCN). It was initially developed by Modicon in the late 1970s. The protocol's use is free, which makes it comparable to other tools discussed in this thesis. Because Modbus is a messaging protocol, its functionality is reliant on the physical OSI layer of the network. This aspect makes it less flexible than BACnet.

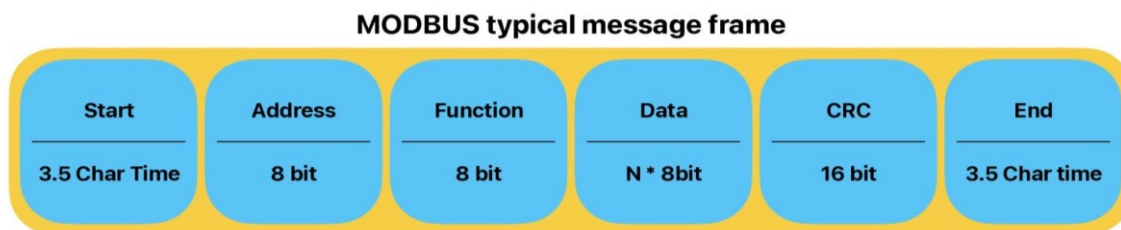


Figure 14: ModBus typical message frame – Source (Own)

Each data exchange process in the Modbus network can be decomposed to a request from the master and a response from the slave like in master-slave transfer protocol (MSTP). Every request or response message starts with a slave or device address (Modbus tools, 2023).

Modbus also can be used at the abstraction of TCP. For this, Modbus encapsulates remote terminal unit (RTU) requests or responses (Figure 14) to a TCP package that can be transferred through ethernet as well. However, the address at the abstraction of TCP doesn't remain the same as in MSTP. In this case, Modbus uses IP addresses like 192.168.1.100 which is a standard Modbus address. The standard port for Modbus TCP is 502, but it can be changed.

It has been mentioned that LonWorks is one more popular networking protocol that is used for operating management. But this protocol is proprietary, therefore it will be left out of the scope of this thesis.

Table 1: BACnet and Modbus comparison

	BACnet	Modbus
Developed By:	ASHRAE	Modicon Inc.
Use	Communication across devices	Connection Between Devices
Markets	Industrial, Transportation, Energy Management, Building Automation, Regulatory and health safety	HVAC, Lighting, Life Safety, Access Controls, transportation and maintenance
Examples	Boiler Control, Tank Level Measurements	Request temperature reading, send status alarm, fan schedule
Transmission modes	Ethernet, IP, MS/TP, Zigbee	ASCII, RTU, TCP/IP
Standards	ANSI/ASHRAE Standard 185; ISO-16484-5; ISO-16484-6	IEC 61158

Costs	Free of charge	Free of charge
Network Interfaces	Existing LANs and LANs infrastructure	Traditional serial and Ethernet protocols
Testing	BACnet Testing Labs	Modbus TCP Conformance Testing Program
Advantages	Scalability between cost, performance and system size, High endorsement and adoption, Robust networking, Unrestricted growth potential	Easy connected to Modicon, sustainable on small amounts of data (≤ 255 bytes), high adoption in industrial applications, easy to deploy and maintain, transfers raw bits or words
Disadvantages	Security standard is not implemented in all devices, MS/TP wire length limitations,	Large binary objects are not supported, no security protocols, limited number of data types, requires contiguous transmissions, requires great amount of configuration and programming

So, which is better: BACnet or Modbus? Based on the comparison in Table 1, Modbus is a simpler protocol. It transmits raw bytes in a message frame, as demonstrated. However, this simplicity can be a disadvantage on a larger scale. Modbus protocol does not support more complex data types and even large binary objects because of its simplicity. Moreover, Modbus lacks any security protocol, whereas BACnet provides one. BACnet is capable of supporting a variety of network protocols, can handle more intricate messages, and is more scalable compared to other options (Setra, 2016). For this reason, BACnet will be the primary BMS networking protocol focused on in this thesis.

3.4 Levels of Abstraction for BMS Representation

All sensors and their communication can be managed using specific networking protocols, such as BACnet or Modbus. To integrate them into BMS specific proprietary plug-ins are usually required. However, these plugins could be swapped for an abstraction capable of encapsulating entities related to communication protocols and physical entities of a building in the homogeneous environment.

3.4.1 Project Haystack

Project Haystack represents an open, structured metadata standard that describes building entities using semi-structured sets of tags, with the aim of substituting unstructured labels (Fierro, 2019). It is a widely used and well-documented open-source toolkit for modelling IoT data. It has a structure similar to JSON (JavaScript Object Notation) in that all values are stored in key-value pairs. This labelling system

accommodates string values, Booleans, numbers, lists, and dictionaries, as well as bespoke Haystack entities such as units of measurement and URIs. Haystack's collection of tools and technologies includes:

- 1) **Set of Data Types** that are used to model IoT data. This Data Types include all general data types that are required to handle data exchange.
- 2) **File Types** that define a set of text formats that could be encoded and decoded without any data loss. These formats are Zinc or strongly typed CSV (Comma-separated Values) tabular data format, JSON, Trio, or more complex version of YAML format for handling hand-written data, CSV, and RDF that could be encoded in turtle or JSON-LD.
- 3) **HTTP API** that allows the exchange of data between different servers and devices. The protocol is based on set of short operations like *read* to query entity data about buildings, rooms, or sensors, *hisRead* to read historized time-series point data, *hisWrite* to push historized time-series to a remote system, *watchSub* to subscribe to real-time sensor data.
- 4) **Ontology** that is used to standardize rules of data modelling. It includes such tags as *site* that is used location of building or sensors, *space* that is used to describe rooms, floors, HVAC zones, *equip* that is used for description of any equipment like boiler or AHU, *device* that could be any microprocessor or sensor, *weather* that is used to describe related to environment conditions, etc.
- 5) **Defs (Definitions)** that is a set of mechanism for transforming each tag into ontology-like format. Defs include *value type* that could be string, number, etc, *supertype* that is a more specific version of common types, *human description of the tag*, *ontological relations* between tags.

An example of a building from the official Haystack website looks like a common file with a structure of a dictionary, which contains key – value pairs (Figure 15).

```
id: @whitehouse
dis: "White House"
site
area: 55000ft2
geoAddr: "1600 Pennsylvania Avenue NW, Washington, DC"
tz: "New_York"
weatherStationRef: @weather.washington
```

Figure 15: Building encoded in Haystack – Source: (Haystack website)

Haystack is a flexible system with advantages such as its ease of use on a small scale and the fact that it does not require facility managers to be familiar with complex technologies. However, it poses challenges when converting to a more organized structure. Additionally, the system offers the Haystack

Tagging Ontology (HTO), which is limited to tags and cannot represent building entities encoded in the collection of tags (such as zone temperature sensors). The methodology for composing complex building systems is not present; instead, the process relies on tags to map raw metadata to the ontology. (Gabe Fierro, 2019).

3.4.2 Brick Schema as a Step to Ontologies

It was previously noted that Haystack incorporates a process similar to modelling data in an ontological way. But what are Ontologies? Ontology can be described as a dictionary in a particular domain that contains a description of knowledge entities and the relationships between them. Ontology should be structured in accordance with linked data principles. Linked data is a set of regulations and optimal approaches for retaining and distributing machine-readable data on the internet. These practices include:

- 1) Using URI as a name for entities
- 2) Using HTTP URI to allow to get more information from the web
- 3) When someone looks up a URI, provide useful information using RDF and SPARQL
- 4) Include cross linking between URI to make data more knowledgeable

Linked data is considered one of the key foundations of the Semantic Web (Ontotext, 2023). Its first principle utilises Uniform Resource Identifiers as an entity identifier which helps in modelling real-world entities digitally. The second highlights the data retrieval mechanism through Hyper Text Transfer Protocol (HTTP) for accessing data on the web. The third principle is about the efficient use of linked data. It incorporates RDF as a graph-based means of representing data, which facilitates the storage of interconnected data, thereby enabling the derivation of new information from pre-existing facts. The Standard Query Language and Protocol for Linked Open Data on the Web or for RDF triple stores (SPARQL) is also part of the third principle, providing a query language for data retrieval. The fourth principle aims to establish a connection between all the data and provide a context for authorship.

Linked Open Data (LOD) is a concept of storing linked data in open source. One of the most popular LOD sources is DBpedia (DBpedia) which stores data from Wikipedia according to the linked data principles. According to Tim Berners-Lee (W3.org, 2009), any data can be ranked in the domain of openness (Figure 16).

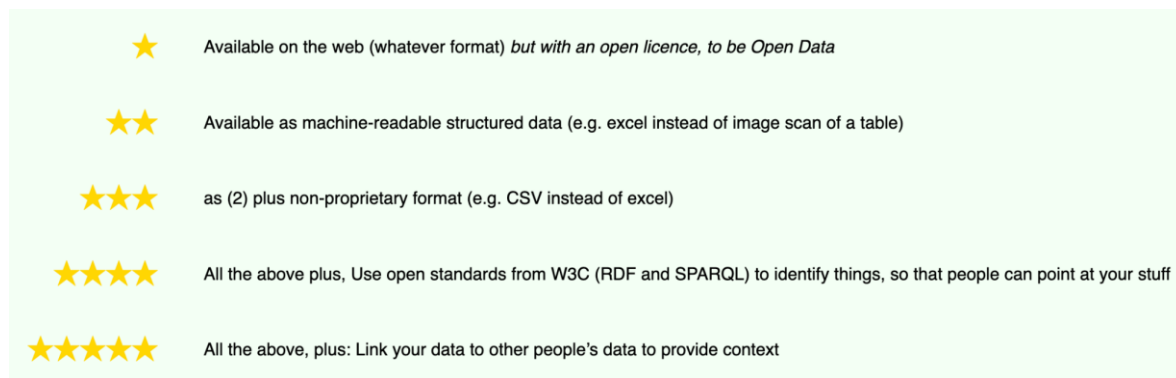


Figure 16: LOD Ranking – Source: Tim Berners-Lee

According to Tim Berners-Lee's ranking system one star can be considered a PDF file that is accessible through the web, two-star data is any structured and machine-readable data available on the web, three stars data has to be encoded in non-proprietary format, four-star data has to be stored in W3C standard, finally, to achieve five stars data has to be linked to data from third-party sources to be enriched with context (W3.org, 2009).

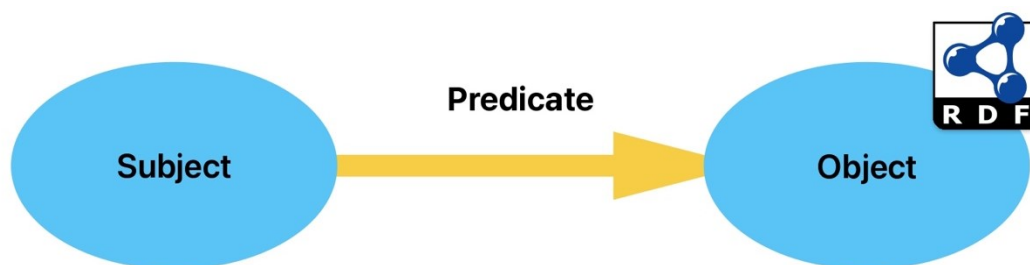


Figure 17: RDF Concept – Source (Own)

Any literal or Internationalized Resource Identifier (IRI) is a mapping of a physical entity to a digital world. IRI's are a generalization of URI's and URL's, which means that IRI provides a wider range of Unicode characters. Such structure allows to execution of queries for data extraction.

SPARQL is comparable to Structured Query Language (SQL) as it enables the retrieval of data through a specific syntax, whereby a majority of the keywords come from SQL (see Figure 18).

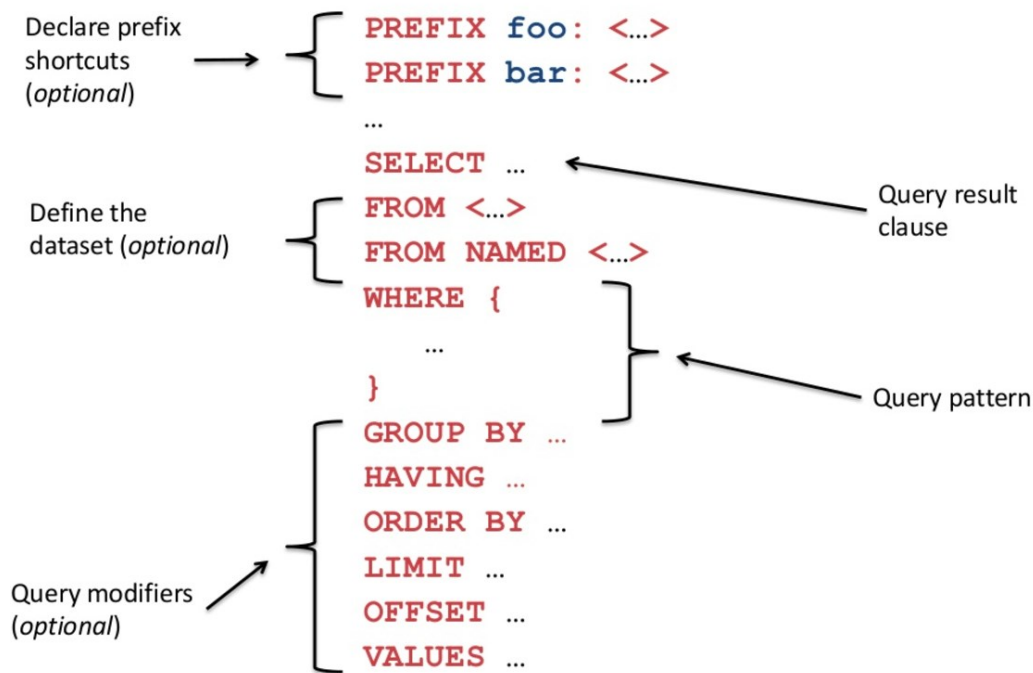


Figure 18: SPARQL querying – Source (WordLift)

The primary dissimilarity between SQL and SPARQL is that in order to query an RDF graph, prefixes previously used in the graph must be defined. Additionally, the select line must specify entities such as subject, predicate, and object. To select all instead of utilizing (`SELECT *`) as in SQL, the SPARQL query for selecting everything needs to look like this (Figure 19).

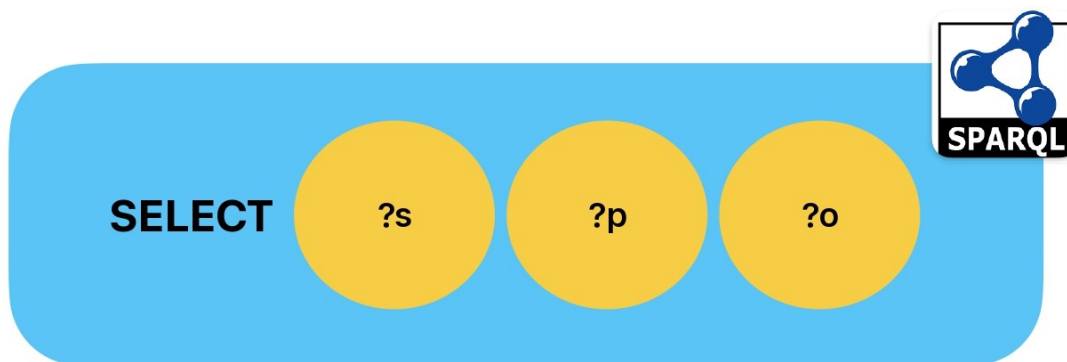


Figure 19: SPARQL Select – Source (Own)

One additional aspect of linked data tools that requires clarification is the serialization format for the created graph. RDF has the ability to represent data about entities that can be identified on the internet, even if they are not immediately accessible via the Web. Examples may comprise details regarding products accessible via online shopping platforms, such as specifications, prices, and stock availability, or the outlining of a web user's predilection for information presentation (W3C, 2007). In order to be locally accrued, an RDF graph must be serialised. The established models for saving RDF graphs include Extensible Markup Language (XML) and Turtle. XML is a widely used format for machines, but it is

not easily readable for humans. In contrast, Turtle is a format that is both machine-readable and human-readable. Turtle data encoding is based on two main datatypes, URIs and Literals, which contribute to Triples. In Turtle, URIs refer to conceptual entities, while Literals can be represented as integers, strings, and other literal datatypes (Haystack Blog, 2017). Triples in turtle consist of subject property objects where:

- Subjects must be URI
- Properties must be URI
- Objects can be either URI or Literal

Turtle is a preferred datatype for storing RDF graphs, therefore it will be used for further application development. As for linked data applications for the specific purpose of encoding BMS-related data, it is important to consider the sensitivity of this type of data. This sort of information ought to be kept confidential. Therefore, level 4 is best suited for structuring heterogeneous data so that it can be easily filtered and accessed when needed.

Monitoring systems, present in most large commercial buildings, collect data from sensors which can be accessed through BMS or Supervisory Control and Data Acquisition (SCADA) systems. However, it should be noted that all data collected is stored in a heterogeneous manner based on the vendor's specifications. This results in human-readable storage only, rendering the app non-scalable or requiring significant resources for porting the application to another building. The Brick ontology can overcome this problem because it stores all the information according to a well-defined dictionary (Balaji et al., 2018). Brick has been formulated to address the limitations posed by current industry standards, catering to the demands of both application and vocabulary requirements. It features an extended tagging model from Haystack, and adopts location concepts from IFC.

Brick is an open-source ontology based on graphs that describes the assets of a building and the relationships between them. The primary concept of Brick is the hierarchical representation of physical, logical and virtual entities. It is designed by an open consortium of researchers and industry professionals. Brick follows RDF structure (Figure 17), where RDF graphs are sets of subject-predicate-object triples. These elements can be IRIs, blank nodes, or datatype literals, and they are employed to express descriptions of resources (W3C, 2023).

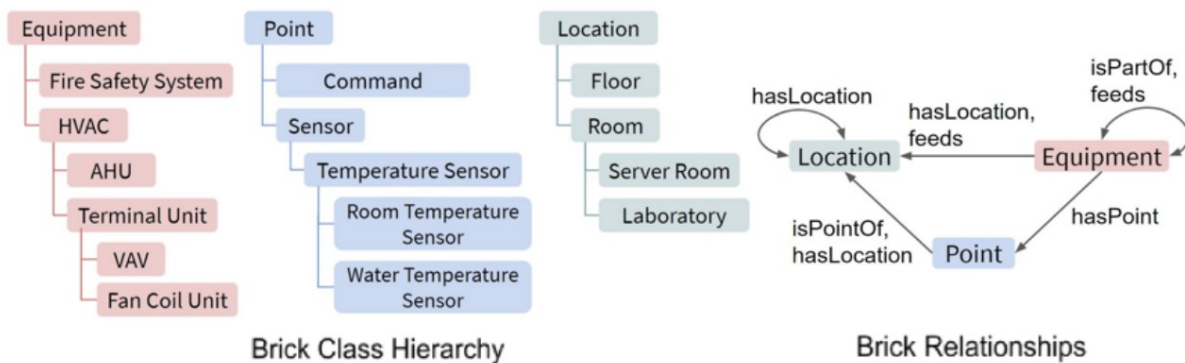


Figure 20: Brick Classes Hierarchy and Relationships – Source (Na Luo et al., 2022)

Class is a fundamental element of Brick. It is a category that defines groups of entities and organises the hierarchy. The Brick ontology comprises of three fundamental classes (Figure 17): Equipment, Point, and Location. However, in the latest version of Brick, this count has been increased to five, as presented in Brick version 1.3. All versions and change logs can be found on the official Brick Ontology website. (Brick Ontology).

- Equipment class represents devices and tool that are supposed to serve the building (pumps, chillers, heaters, lighting etc)
- Point class represents sensors that produce data
- Location class represents location of physical and logical entities
- Collection class that represents systems like air and water loops, gas, electrical and other systems
- Measurable class that consists of Quantity and Substance, where Substance can be fluid or solid

Brick has been developed to enable the deployment of large-scale building applications in a consistent and usable manner. Its main advantage lies in its machine-readable format. In contrast, Haystack must first be comprehensible and easy to understand by facility managers. This can prove to be challenging when multiple teams of facility managers become involved at a large scale. That's why it is crucial for extensive metadata to have consistency, and for the schema utilised to describe it to specify the guidelines for organising, defining, constructing, and expanding data.

Haystack uses sets of value tags and marker tags to define entities. Value tags have the structure of a dictionary (key-value structure), and marker tags are singular annotations. A set of marker tags constitute an entity that is called a “tag set”. Haystack lacks an explicit class hierarchy therefore automation of the generation of the Haystack model can be complicated.

One of Brick's greatest advantages is its extensibility due to its open-source nature, allowing it to be tailored to specific use cases. Brick defines nodes as building assets, equipment, and subsystems, while edges represent their relationships, such as location, control, and type of connection. One case study developed a fourth root class, "Occupant," (Na Luo et al., 2022). This lesson covered all occupant interactions, accesses, and even physical information about the occupant such as age, gender, etc. Following testing, the data coverage of traditional Brick, extended Brick, and IFC for occupant data was analyzed. In an initial case study of an educational building in Rende, Italy, IFC captured 54% of the data points, traditional Brick captured 69%, along with 40% of occupant behavior, while extended Brick managed to cover 100%. In a second case study exploring educational buildings in Shanghai, China, that was more behaviour-oriented, extended Brick achieved complete coverage of the buildings, while IFC only covered 45%. Moreover, it is possible to extend Brick using existing classes without creating new ones. In one of the case studies, Brick was expanded to model the Variable Refrigerant Flow System (VRF) in a complex manner (Jingming Li et al., 2021).

Even without any extensions, the Brick ontology provides considerable depth to represent almost everything in the domain of HVAC and BMS systems. According to research by Balaji et al. (2018), IFC, Haystack, Brick, and Smart Applications Reference Ontology (SAREF) were utilized to describe BMS metadata of existing buildings, where Brick achieved the highest percentage of both vocabularies and application requirements (see Figure 18).

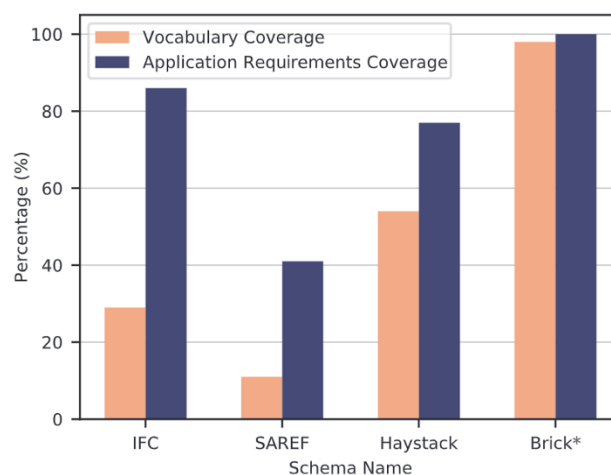


Figure 21: Comparison of different schemata for buildings – Source (Balaji et al., 2018)

The most recent version of Brick 1.3 has been enhanced to enable the inclusion of external references via the ref-schema (Ref-Schema, 2023). The software has already integrated references to IFC or BACnet, and additional references can also be added manually.

Brick also provides validation functionality. Version 1.3 of the Brick ontology follows W3C recommendations and moves from an OWL-based ontology to a SHACL-based ontology (Brick Schema, 2023). This enables the ontology to include a validation feature. SHACL is a high-level vocabulary containing properties that distinguish the validation of data from the deduction of new information (SPIN, 2017). It is also featured on the updated semantic web layer cake (Figure 19).

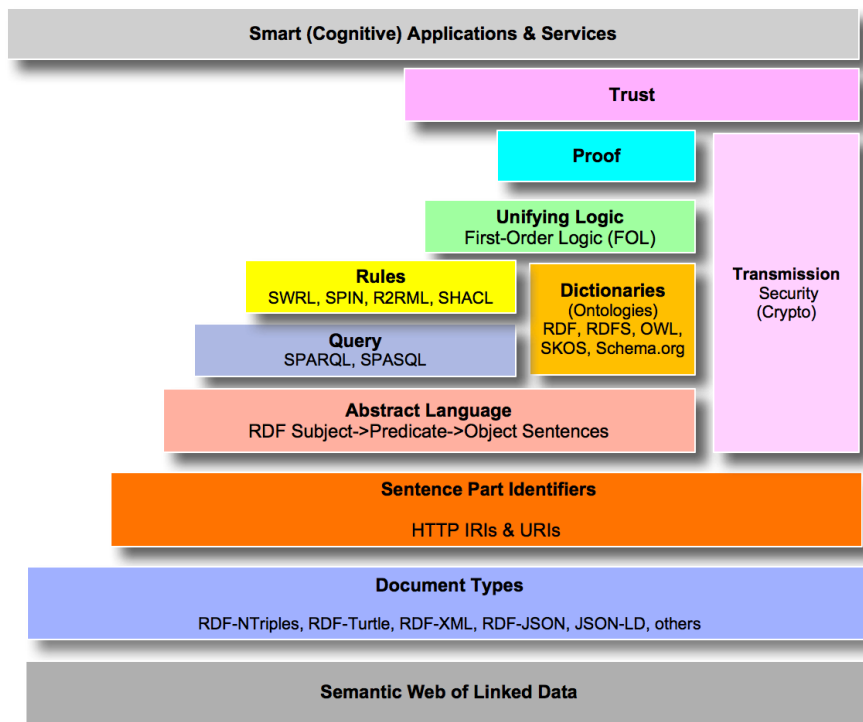


Figure 22: Updated Semantic Web Layer Cake – Source (Kingsley Uyi Idehen)

3.5 Linked Data for Buildings Applications

The absence of standardized, machine-readable data hinders the development of portable building applications that can include fault detection, energy audits, and optimal controls. By storing data in a machine-readable format, it would be possible to create applications that can be scaled and reused for multiple purposes. Existing metadata from a building automation system (BAS) can be mapped to an ontology, but this will require considerable time and cost for each specific case (Pritoni et al., 2021). In contrast, if all data is stored in a machine-readable format from the outset, it can be a useful data source that can also store important data such as hierarchical relationships between components of the HVAC system.

To facilitate the analysis of the selected metadata schemas, it was defined use cases (Figure 20):

- Energy audits
- Automated fault detection and diagnostics
- Optimal control

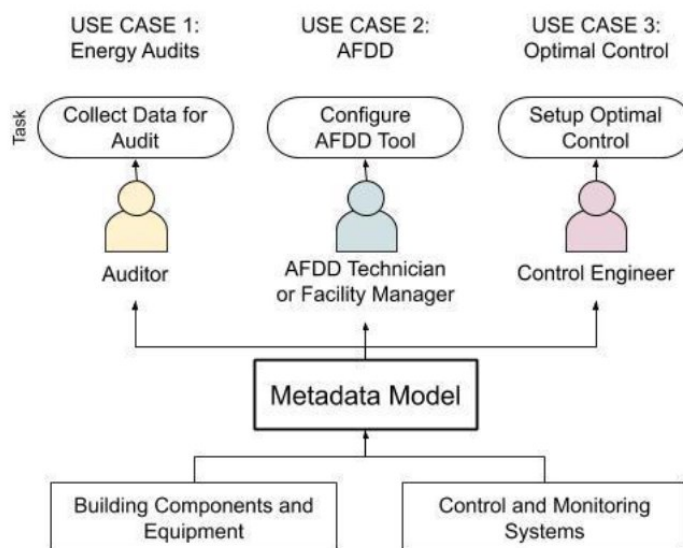


Figure 23: Use case developed and use of metadata models – Source (Pritoni et al., 2021)

Use case 1. Energy Audits: Clear and efficient data collection is essential for conducting an energy audit. The auditor should gather data on building monitoring and control systems. Once the data has been collected and evaluated, the auditor should suggest energy efficiency measures (EEMs) and estimate the energy and cost savings that could be achieved if these EEMs were implemented. A metadata model helps to ensure an accurate and efficient collection of data during energy audits (Pritoni et al., 2021).

Use case 2. Automated Fault Detection and Diagnostics: In this scenario, a facility manager or third-party technician is responsible for configuring and monitoring the output of an automated fault detection and diagnostics (AFDD) tool, which is designed to optimize the performance and lifetime of one or more building systems, and for performing preventive or reactive maintenance on the monitored systems. This includes preventive and reactive maintenance as well as access to building component and system data and control and monitoring system information to configure the AFDD tool. A metadata model enables the effective and precise configuration of the tool, the analysis of faults, and the reporting of diagnostics.

AFDD tools analyse historical time series data in combination with knowledge of system capabilities, schedules, and sequences of operation to identify operational faults and opportunities for control improvement. Commercially available AFDD tools are primarily developed for monitoring HVAC BAS data, but similar approaches may also be applied to lighting and other systems. The tool outputs tables and/or graphics.

The interconnection between each system is also required. Thus, the metadata must outline that the AHU delivers air to the two VAV boxes and the two thermal zones, with the bathroom fan ventilating air from HVAC Zone 1. Additionally, the tool must associate each fan with its respective submeter. Time series

data must be provided for all sensors, actuators, and virtual points, indicating their unit of measure and expected reporting interval or interval configuration options.

Notably, the architecture of the system's input and output can differ. For instance, a cluster of lighting devices could be created and fixed to cater to a specific area, but the electrical subsystems powering the devices could be created and fixed to operate in multiple areas or parts of them. All metered parameters require time series data, including their unit of measure, their anticipated reporting interval, or configuration options.

Use case 3. Optimal Control of HVAC: In this scenario, an engineer installs and sets up a supervisory control system for the optimal operation of one or more building systems. To configure the supervisory control system, the engineer requires access to both the building system and component data, as well as control and monitoring system data. A metadata model aids in the precise and effective setup of the control system or software.

Model-predictive control (MPC) strategies calculate optimal inputs by minimising an objective function over a given prediction horizon, given a set of constraints. MPC has yet to be implemented at scale, primarily due to the significant effort required to configure its models. The configuration of such models requires the collection of detailed information about the HVAC system, its components, and the relationships between these components and system performance. Time series data from sensors and actuators is frequently required to train the model hyperparameters and implement the optimisation algorithm.

To effectively utilize the MPC model, it is essential to possess all the HVAC information highlighted in the AFDD use case. In addition, one must have comprehensive knowledge about the building, such as the location and orientation of the building, the size of the windows, the estimated properties of the various building elements, as well as the internal mass of the building. To accurately define the internal heat transfer between the building's different zones, information about the adjacency of these zones is necessary.

4 KEEPING APPLICATIONS INTEROPERABLE

Most BIM models nowadays are developed using proprietary software that is only available for the Windows operating system. Industry standard practices involve exchanging data via models which must be properly exported into a proprietary format such as Revit's ".rvt" format or an open format such as IFC. Following this, the model-specific files may be stored in a Common Data Environment (CDE) which manages access, data security and storage. That is the current reality of the BIM industry, where individuals must work with heavy, proprietary applications that do not interoperate.

But this is a reality of BIM as a developing branch of the IT industry in general. Objective evaluations should be excluded as a matter of principle. When it comes to software products that are used by millions of people across the world, these software products cannot afford to be outdated, otherwise they will be outcompeted for customers. Big software products are typically either web-based or offer a desktop client, which is essentially just another version of the front-end. All the logic is executed on servers. Those servers also do not simply run applications and stream data to the end user. Nowadays, most modern applications possess a complex, decentralised architecture. It is unimaginable that a service like Spotify runs on one large computer. Instead, it has multiple data centres - one for the storage of billions of songs, another for the processing of payments from listeners and payments to artists, and a third for storing album covers. Therefore, this software is no longer monolithic; rather, it has become distributed.

This methodology may prove beneficial in the development of open-source applications, as distributed software components are more easily modified or removed than their monolithic counterparts. Additionally, the market offers a variety of tools to enable the platform-independent execution of these decentralized components, using lightweight virtual machine versions. A distributed approach will be employed for the development of a case study application.

4.1 Existing Brick Based Software

The Brick community has already created various software tools for working with Brick on different levels of abstraction. These tools are available on the Brick Schema GitHub (BrickSchema). Some facilitate BACnet interaction, while others enable users to operate with Brick models at the HTTP abstraction level. However, all of them share a distributed architecture based on Docker containers. Developing applications based on Docker containers for further sharing and testing among users is a generally recommended strategy.

4.2 Docker

Docker is a platform that enables independent operation of software infrastructure and application architecture management, reducing time to deployment. Docker applications are created using containers, which are simplified Linux virtual machines with a reduced level of isolation. These containers can be run on a single host and interconnected with each other. Containers can also be shared within a team or with anyone who is interested. Docker provides a set of tools that can be used to manage the lifecycle of Docker containers, including:

- Develop application and supporting tools based on containers
- Container becomes a unit for exchange and testing
- After container is tested it is ready to be deployed as single container or orchestrated service in any environment

Docker-based development occurs within a standardised environment and is a practical solution for applications intended for expansion. This process is known as continuous integration and continuous delivery (CI/CD). Go programming language is employed in developing Docker, which is a reliable multiprocessing language with a robust type system. One of Docker's features is the ability to expand. It utilizes Linux kernel functionality and was originally developed on top of Linux containers (LXC); however, it has since been updated and now offers a fully integrated environment for application development and deployment.

To proceed, it is important to distinguish between docker and Virtual Machine (VM). A VM simulates an entire computer, including its hardware components, such as CPUs, USBs, sound cards, network features, etc. VM offers a higher level of isolation, but requires more resources to operate. Conversely, docker containers operate in isolated instances, or containers (Figure 25).

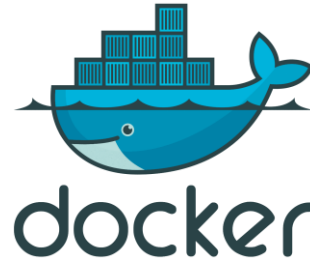


Figure 24: Docker - Source (Docker Website)

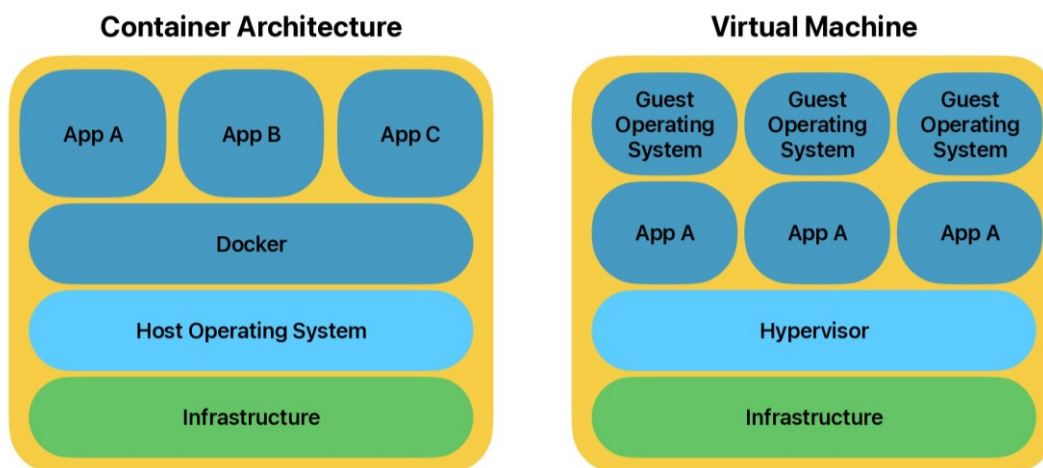


Figure 25: Docker and Virtual Machine Architecture – Source (Own)

Docker containers possess their own file system, dependency structure, processes, and network capabilities, equipping the application with all the necessary resources to run seamlessly no matter where it is executed. As Docker container technology utilises resources of the underlying host operating system kernel directly, it provides a flexible and portable solution.

As for containerizing an application the workflow could be defined as following (Figure 26):

- 1) In accordance with the application's functionality, generate a Dockerfile within the application's folder. The Dockerfile should indicate the ports that will connect the enclosed container environment to the local machine. The Dockerfile shall oversee the management of Docker volumes, although this technology is not applicable to the case study and henceforth excluded from our scope.
- 2) Once the Dockerfile has been created, it's important to confirm that the Docker platform is installed on the computer. Docker offers different applications for each platform, including Mac and Windows. Since Docker runs on a virtual Linux machine, If the operating system is Linux, there is no need to download a specific application, and everything can be managed through the Docker Command Line Interface (CLI). Therefore, the Docker application should be installed before building the image. Image is a snapshot of code that will be used as a blueprint for a container. Images could be exchanged between users using Docker Hub (Docker Hub, 2023).
- 3) As long as an image has been built successfully, Docker can execute a container that functions as a fully operational application. The "run" command in Docker may include various flags to influence the container's behaviour such as deleting the container after it shuts down, running a container in interactive or detached mode, establishing volumes for cross-container communication, and so on.

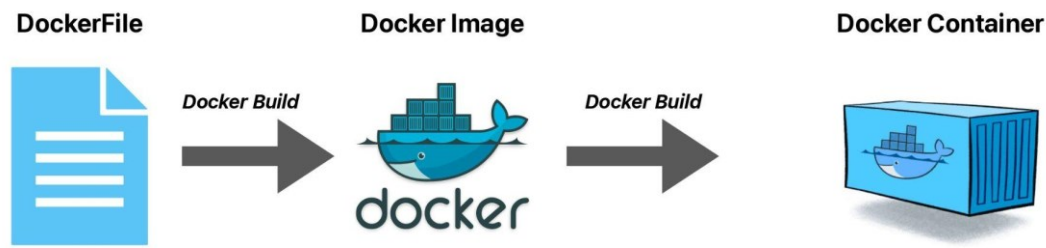


Figure 26: Docker Application Deploying – Source (Own)

After an application is developed it can be run on a localhost or on any webserver like Amazon Web Services (AWS). Application can be self-sustainable or be a part of a big, distributed architecture.

4.3 Mortar

Usually, information about buildings is collected from various sources of data. Information can be collected from drawings, by scanning BMS, documentation, from staff that works in the building. It makes implementation of control logic more complicated, and a lot of things stay untested and unevaluated. Even if a single building can be evaluated with big efforts by structuring all collected data, a developed tool can't be scalable without structuring all other building's data.

Mortar (Modular Open Reproducible Testbed for Analysis and Research) is a platform enabling developers to build and assess portable applications for constructing analytics. It comprises mainly of time-series data, collected from sensors of various buildings. Mortar can manage each brick model of these buildings and incorporates analytics applications, providing access to the time-series data extracted from models through executing queries.

The architecture of mortar has to meet such requirements as:

- linking historical data with it's context
- be able to execute queries to extract data
- be scalable

The primary objective for time-series data storage is to enable users to upload data produced from existing points and newly annotated points. Concerning the storage of brick models, Mortar also needs to track all modifications made to the models. Mortar's current architecture is reliant on Kubernetes-managed clusters.

- As a time-series database it uses BTrDB which has a fast storage and query system for scalar-evaluated time-series data.

- As a brick storage it is used HodDB which is RDF/SPARQL high performance database
- Query processor is developed in “Go” programming language and interacts with BTrDB and HodDB that are deployed on the same cluster.

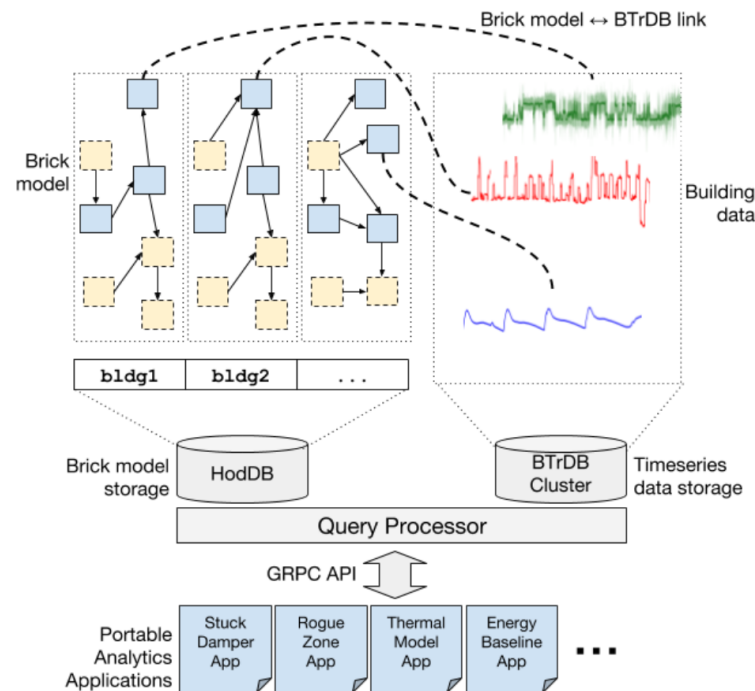


Figure 27: Mortar architecture – Source (Fierro et al., 2018)

Portable applications should be able to be moved easily between different computing environments. Traditional applications designed for a single building are often non-portable due to hard-coded point names, assumptions about building structure, and other factors.

- **Qualify.** Qualify component of portable application defines data requirements for an application. This executes constraints: building topology and other properties, data context and available relations, data availability and data resolution. These constraints are executed via Brick queries.
- **Data Retrieval.** Fetch component is actual extraction of data from time-series database corresponding to data streams that were defined in a previous step. The result of fetch component is an access to Brick queries represented as object.
- **Data Cleaning.** Clean component has to be executed on the result of fetch component. This component is performed to successfully execute analyse component

- **Application Execution.** Analyse component contains all logic of application. It can perform all visualizations that can be a basis for decisions. There also can be optional aggregate component that can execute analyze component to data from all sites.

4.4. Brick Applications in Use

Most of the mentioned applications were developed for a particular case study. The most relevant field for those applications is analytics of building-related data. Some of them provide just a backend or minimalistic CLI interface, others are based on microservices and have complex structure.

4.4.1 Detecting Passing Valves

During the operation of building VAV is supposed to keep the indoor environment under control. But during operation of VAV some mistakes can occur. One of them is distribution losses caused by penetration of hot water through the valve when it was supposed to prevent it. Those losses are usually not accounted. It can be caused by several reasons such as:

- age of valves and conditions of exploitation
- fouling or blockage
- lost communication, fault of closing mechanism or any other BAS fault
- human factor

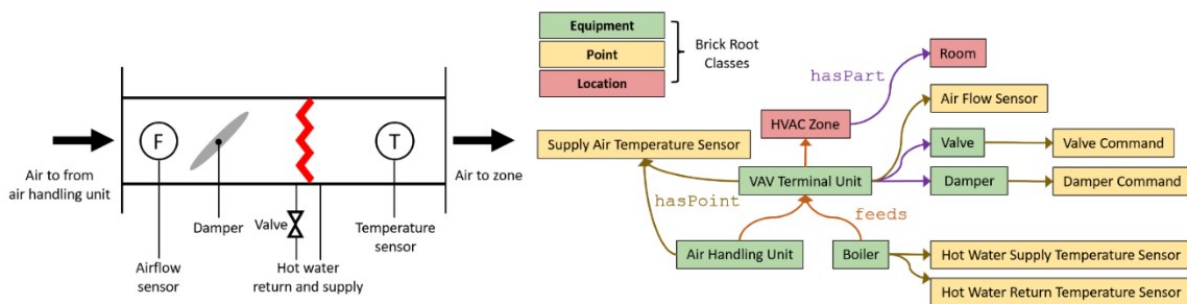


Figure 28: Left) schematic of a variable air volume (VAV) with reheat terminal unit and right) a Brick data model of the VAV terminal unit – Source (Duarte Roa et al., 2022)

Representation of VAV in both ways schematic and brick. Brick model allows retrieval of data by its purpose instead of using non-standardized naming conventions. Minimal data for this application to run are:

- supply air temperature for AHU
- supply air temperature for VAV

- VAV hot water valve position

The Role of Brick: The defined ontology of Brick enables predictable building representation. This means that the relationships between AHU, VAV, and sensors are expected to be consistent. Therefore, developing an application that retrieves data from a specific set of points will allow for scalability simply by altering the input "path". The algorithm developed can be applied to each set of data from the case study building specifically, as well as to any other building encoded in BRICK.

As for the process of developing a detection application for passing valves, it can be described as data-driven. The data needs to be extracted from the brick model in accordance with the defined pattern and verified. Subsequently, a methodology for fault detection needs to be established. Finally, the developed algorithm is applied to the data accessed from the available mortar brick data to ensure its portability.

Results: "Algorithm categorized 5% of VAV units as having a sensor fault, 14% with a potential passing valve fault, and 81% with no faults detected". "These faults resulted in an average heat rate loss of 1,375Bth/hr with a cumulative 14,400 kBtu of energy loss or 8% of the total international reheat energy used by all VAV units analyzed for the time period" (Duarte Roa et al., 2022). This case study shows that brick can be a key to data driven approach for detection of faults. Results can be a basis for decision making for facility management teams.

4.4.2 Occupant Satisfaction

Data structured according to a standard can also be utilised to assess occupant satisfaction. As a case study (Mosiman et al., 2021), it has been used in a living laboratory in Boulder, Colorado, USA. The laboratory was fitted with different sensors and systems to regulate the indoor environment. A digital model was created using both Brick and Haystack approaches. The office building was supplied with a shared air handling unit positioned on the roof of the building. AHU had a natural gas furnace and lacked metering infrastructure. The schema and digital representation are illustrated in Figure 29.

After developing the models, various methodologies were employed to establish correlations between space utilization and energy consumption. Consequently, a multiple linear regression was conducted using energy as the dependent variable and space utilization and outdoor air temperature as independent variables. The evidence strongly suggests a relationship between the percentage of space utilization and energy consumption. This finding suggests that while the precise number of occupants in a space remains unknown, the methodology for calculating percentage space utilization is still valuable when employed as an explanatory variable in a regression model for predicting energy usage. (Mosiman et al., 2021).

Main idea that can be extracted from this case study is that development of a model according to well defined ontology allows to perform different types of data analysis.

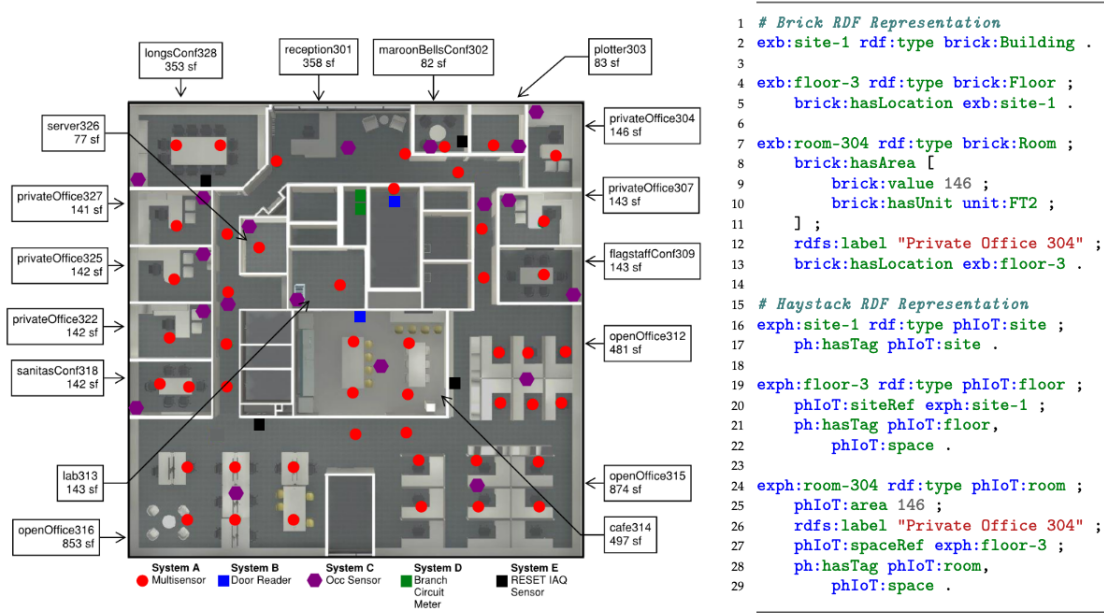


Figure 29: Experimental setup of the commercial office space used in the experiment – Source (Cory Mosiman et al., 2021)

As it was shown in case studies, brick-based workflows and softwares allow to test application of a larger scale, due to defined encoding of a facility and defined requirements to run a software.

5 PROOF OF CONCEPT DEVELOPMENT

According to Pritoni et al. (2021), ontologies should be applied in one of three fields: Energy Audits, Automated Fault Detection and Diagnostics, or Optimal Control of HVAC. Furthermore, the application should be developed to be integrated into a larger system. The application will utilize IFC in the EXPRESS STEP encoding format for static data, CSV files for temperature data, and an RDF graph encoded in Turtle for dynamic BMS-related data. Figure 30 provides a visual representation of the application schema.

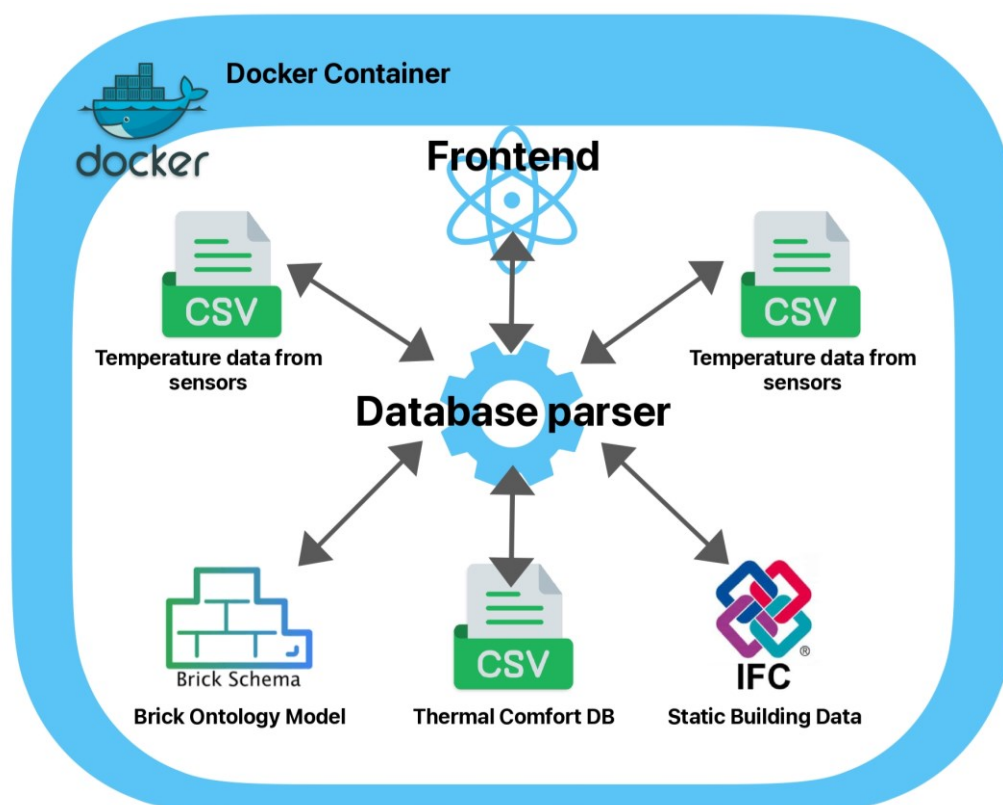


Figure 30: Schema of Proof Of Concept – Source (Own)

The application processes various sources of data for evaluation, which can be displayed in a simple front-end. Given the complexity of the task, the application avoids interfering with the BMS. However, if there is a potential impact on HVAC equipment, the application will require safety mechanisms, security layers, etc. Also, the application must be interoperable and available for secondary use as part of a larger application. Therefore, it will rely on container technology, specifically Docker.

5.1 Sources of Data

5.1.1 ASHRAE Thermal Comfort Database

The ASHRAE Thermal Comfort Database is an accessible and empirical source of data for assessing occupant comfort. Data is stored in a massive CSV file containing different types of data collected in different ways and locations. Objective temperature measurements, indoors and out, were recorded alongside survey-based data. Surveys were filled out by individuals who engaged in various activities in locations, such as a home or office. All data was standardised and published.

Every CSV file has named rows. In the case of the Thermal Comfort Database, rows contain descriptive metadata. The database's latest version includes column headers such as Name of Contributor, Publications, Year, Country and City, Season, Climate Zone, Building Type, Cooling Strategy, Sample Size, Directory, and List of Objectives, among others (Földvary Licina et al., 2018). The information can be grouped into the following categories:

- Identifiers (Building Code, heating and/or cooling strategies, geolocation)
- Personal Information about subject of questionnaire (weight and height, sex, age)
- Subjective (preferences, sensation)
- Instrumental (Indoor and outdoor temperature, humidity, air velocity)
- Comfort Indicators (Predicted Mean Vote – PMV, Predicted Percentage Dissatisfied – PPD, Standard Effective Temperature – SET)
- Available Indoor environmental controls (heaters and coolers, doors and windows)
- Outdoor meteorological information (average temperature)

After data was collected, it should pass different quality assurance procedures. Data was visualized to exclude anomalies. Absent data fields were filled with null values. ASHRAE Thermal Comfort Database stores data that was collected from 23 countries all over the world. Database also stores information about different types of buildings. Most of them are offices (55238). There are also classrooms (12755) and multifamily houses (10120), senior centres (312) and other (3421). Information about cooling strategy is also crucial. Database stores information about buildings with natural ventilation (38584), air-conditioned buildings (28544), mixed (11745), mechanically ventilated facilities (1804) and other (1169).

For the proof of concept, the most relevant field is Standard Effective Temperature (SET). This temperature will be used as a standard for evaluation. SET temperature is distributed according to location, which is defined with columns “Country” and “City”, but there is no mathematical entity for comparison or mapping with any other source of data. Therefore, the thermal comfort database should be extended with columns with geographical data. For this purpose Jupyter Notebook (Jupyter Notebook), Pandas (Pandas), and NumPy (NumPy) Python libraries and tools were used.

Jupyter Notebook is an open source web environment that allows to perform calculations and visualizations of data. Jupyter Notebook uses the Ipython kernel to perform calculations in Python programming language. NumPy is open source, widely used, and a fundamental python package for scientific array calculations. It is implemented in C programming language, therefore it works much faster than calculations performed in a pure python. Pandas is a python library based on NumPy that allows calculation based on numerical tables and time series dumps. Pandas is also open source and widely used in data analysis. Using described tools coordinates were inserted manually and mapped to all cities from the database (Figure 31).

```
[3]: print(comfort_df["City"].unique())

['Tokyo' 'Texas' 'Berkeley' 'Chennai' 'Hyderabad' 'Ilam' 'San Francisco'
 'Alameda' 'Philadelphia' 'Guangzhou' 'Changsha' 'Yueyang' 'Harbin'
 'Beijing' 'Chaozhou' 'Nanyang' 'Makati' 'Sydney' 'Jaipur' 'Kota Kinabalu'
 'Kuala Lumpur' nan 'Beverly Hills' 'Putra Jaya' 'Kinarut' 'Kuching'
 'Bedong' 'Bratislava' 'Elsinore' 'Gabes' 'Gafsa' 'El Kef' 'Sfax' 'Tunis'
 'Midland' 'London' 'Lyon' 'Gothenburg' 'Malmo' 'Porto' 'Halmstad'
 'Athens' 'Lisbon' 'Florianopolis' 'Brasília' 'Recife' 'Maceio' 'Seoul'
 'Tsukuba' 'Lodi' 'Varese' 'Imola' 'Shanghai' 'Liege' 'Mexicali'
 'Hermosillo' 'Colima' 'Culiacan' 'Mīrida' 'Tezpur' 'Imphal' 'Shilong'
 'Ahmedabad' 'Bangalore' 'Delhi' 'Shimla' 'Bandar Abbas' 'Karlsruhe'
 'Bauchi' 'Stuttgart' 'Hampshire' 'Wollongong' 'Goulburn' 'Singapore'
 'Cardiff' 'Bangkok' 'Jakarta' 'Montreal' 'Brisbane' 'Darwin' 'Melbourne'
 'Ottawa' 'Karachi' 'Multan' 'Peshawar' 'Quetta' 'Saidu Sharif' 'Oxford'
 'San Ramon' 'Palo Alto' 'Walnut Creek' 'Townsville' 'Liverpool'
 'St Helens' 'Chester' 'Grand Rapids' 'Auburn' 'Kalgoorlie' 'Honolulu']

[4]: comfort_df.loc[comfort_df["City"] == "Tokyo", "Coordinates_lat"] = 35.6762
      comfort_df.loc[comfort_df["City"] == "Tokyo", "Coordinates_long"] = 139.6503
```

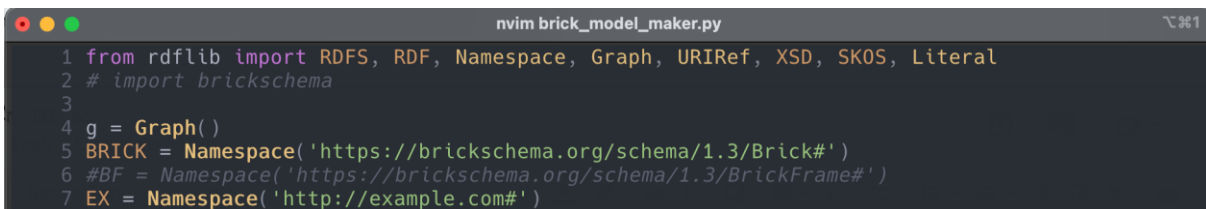
Figure 31: Insertion of geographical data into ASHRAE Database – Source (Own, Jupyter screenshot)

5.1.2 Brick Model

To evaluate the fundamental idea of obtaining information from various sensors, we created a Brick model depicting a system consisting of two rooms, two temperature sensors, two VAVs, and an AHU that supplies the VAVs with air. The model required a minimum of two rooms to ensure the possibility of using an ontology model to switch between sensors. The Brick ontology served as a reference point, supplying an already established set of classes to define entities and the connections between them. The classes shall be organised in an RDF graph and converted to Turtle format. The model was created using

the RDFlib Python package (RDFlib, 2023), which permits the substitution of frequently used namespaces with Python code and serialization of the graph.

In the RDFlib development process, a coherent addition of triple by triple is necessary. Initially, a designated space for all such triples, known as "Graph" (Figure 32), is defined. As the ontology model must conform to certain rules, a namespace called "BRICK" has also been created. This namespace serves as a link to the domain class of Brick. All entities to be used in the future will inherit from this domain.



```
nvim brick_model_maker.py
1 from rdflib import RDFS, RDF, Namespace, Graph, URIRef, XSD, SKOS, Literal
2 # import brickschema
3
4 g = Graph()
5 BRICK = Namespace('https://brickschema.org/schema/1.3/Brick#')
6 #BF = Namespace('https://brickschema.org/schema/1.3/BrickFrame#')
7 EX = Namespace('http://example.com#')
```

Figure 32: Brick model development. Defining a graph – Source (Own, Nvim screenshot)

Also, an entity was created in the "EX" namespace to serve as a subject in triples requiring a physical entity. With all preparations completed, a Brick Schema entity can now be introduced. An example of a triple is demonstrated in Figure 33.



```
nvim brick_model_maker.py
16
17 #Creating Building and 2 Rooms
18 building = (EX['Building-1'], RDF['type'], BRICK['Building'])
19 g.add(building)
20
```

Figure 33: Brick model development. Creating a triple – Source (Own, Nvim screenshot)

Translated into understandable human language, the triple describes an entity as a brick building using Brick ontology. The relation between the entity and the building class is "is" and is represented in RDF[type] predicate as "a" in Turtle encoding. Once the triple has been defined, it can be added to the Graph.

After placing the first triple in Graph, it becomes possible to add a second and connect them through a specific relationship. The entity for the room was created and is shown in Figure 34. The "BIMA_room" triple establishes the entity for the room, while "BIMA_room_building" defines the relationship predicate between the building (Figure 33) and "BIMA_room". According to the ontology, they can be connected using the "isLocatedIn" relationship. So the triple "BIMA_room_building" denotes that "RoomBIMA" is situated in "Building". Once these triples have been defined, they can be included in the Graph.

```

20
21 BIMA_room = (EX['RoomBIMA'], RDF['type'], BRICK['Exercise_Room'])
22 BIMA_room_building = (EX['RoomBIMA'], BRICK['isLocatedIn'], EX['Building-1'])
23 g.add(BIMA_room)
24 g.add(BIMA_room_building)

```

Figure 34: Brick Model Development. Adding relationship triple – Source (Own, Nvim screenshot)

A brick model is unable to hold sensor data as a lengthy string, necessitating encoding each sensor in an RDF graph with a specific triple. The subject of this triple is "hasTimeseriesReference," and its object is a literal containing the path and name of the corresponding timeseries database. For simplicity, the timeseries database will be stored locally as a CSV file, thereby providing a local copy of the timeseries database. In subsequent iterations, the subject can be updated to a database authorization token or any other access management tool.

After all triples are added to the graph, it can be serialized into Turtle format. There are a lot of visualizers that can display Turtle as a schema. Visualization is illustrated in Figure 35.

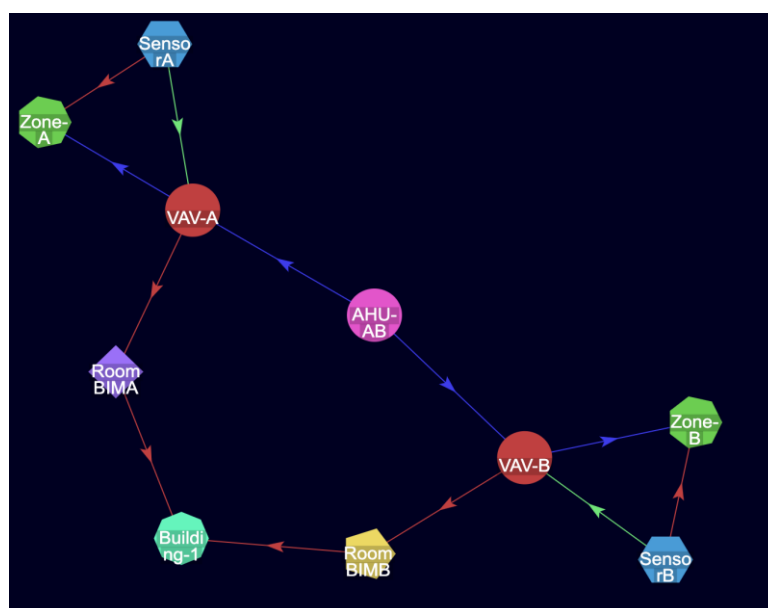


Figure 35: Brick model visualization – Source (Own)

5.1.3 Data From Sensors

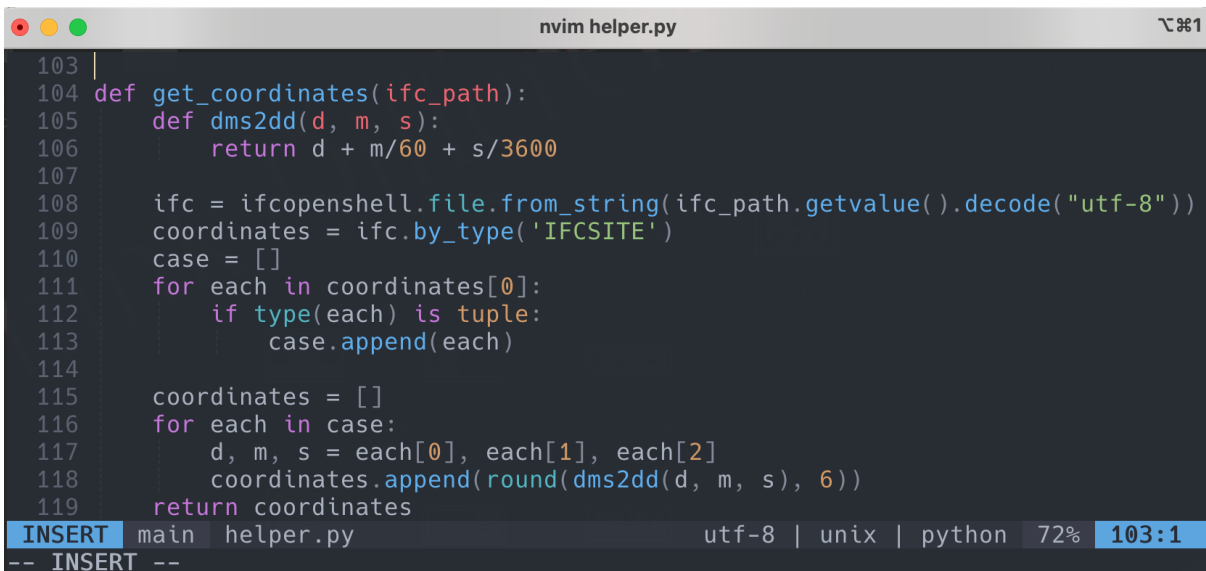
Data collected from sensors should be stored in an easily accessible and clear format. Initially, all data was stored in an SQL database using the SQLAlchemy Python package for interactions. However, this was later replaced with CSV files due to their accessibility and neutrality. Additionally, the exclusion of SQLAlchemy from the application had a positive impact on its size. Instead, previously employed libraries such as NumPy and pandas were utilised to manipulate CSV files.

All temperature data was randomised in Python and serialized into CSV files. It is the simplest solution which was used to prove a concept of the application.

5.1.4 Static Building Data

The final aspect to define was the static data for building construction, which we achieved using the IFC format. However, during the proof-of-concept stage of application development, full integration of IFC into the parser logic was not possible. The primary objective of IFC in this version of the application is to provide geographical data, enabling us to establish the building's location. The `IfcOpenShell` (`IfcOpenShell`, 2023) Python package was used to extract this data. `IfcOpenShell` is an open-source tool kit that enables the mapping of all data from IFC to Python's language entities and the manipulation of that data inside Python's environment.

After IFC file was read and all data was mapped into python entities, "IFCSITE" type was found which contains geographical coordinates of the building. After coordinates were converted to an appropriate format (Figure 36).



```
103 |
104 def get_coordinates(ifc_path):
105     def dms2dd(d, m, s):
106         return d + m/60 + s/3600
107
108     ifc = ifcopenshell.file.from_string(ifc_path.getvalue().decode("utf-8"))
109     coordinates = ifc.by_type('IFCSITE')
110     case = []
111     for each in coordinates[0]:
112         if type(each) is tuple:
113             case.append(each)
114
115     coordinates = []
116     for each in case:
117         d, m, s = each[0], each[1], each[2]
118         coordinates.append(round(dms2dd(d, m, s), 6))
119     return coordinates
INSERT main helper.py utf-8 | unix | python 72% 103:1
-- INSERT --
```

Figure 36: Extracting geographical data from IFC – Source (Own, Nvim screenshot)

5.2 Application's logic

After defining and preparing all data sources, the next step was to develop the logic of the application. Python programming language and a variety of external packages were used for all calculations and data operations. Each package was installed using the Python package manager "pip". The requirements for the packages were exported to a text file named "requirements.txt", which is accessible on GitHub in the application folder.

5.2.1 Workflow

The development of an application is not a linear process since some changes can have a negative impact. Developers may be required to take a step back. For this reason, an application development

process must incorporate a change-tracking system. Currently, one of the most widely-used tools is Git, a decentralised version control system created by Linus Torvalds in 2005 specifically for the development of the Linux Kernel (Git website, 2023). The case study was created by an individual on a solitary machine, thus not all git features were utilised.

The git development process can be broken down into four stages, within the context of a single branch. When git is initialised in a folder, all files are initially marked as untracked. Executing the command "git add <filename>" tracks a file. Upon making modifications, the file is moved to the modified stage. The file can then be returned to the stage by executing the command "git add <filename>". When all modifications are complete, the files can be committed by utilising the "git commit -m <comment>" command (Figure 37). If desired, they can also be pushed to a remote repository using the "git push <branch>" command.

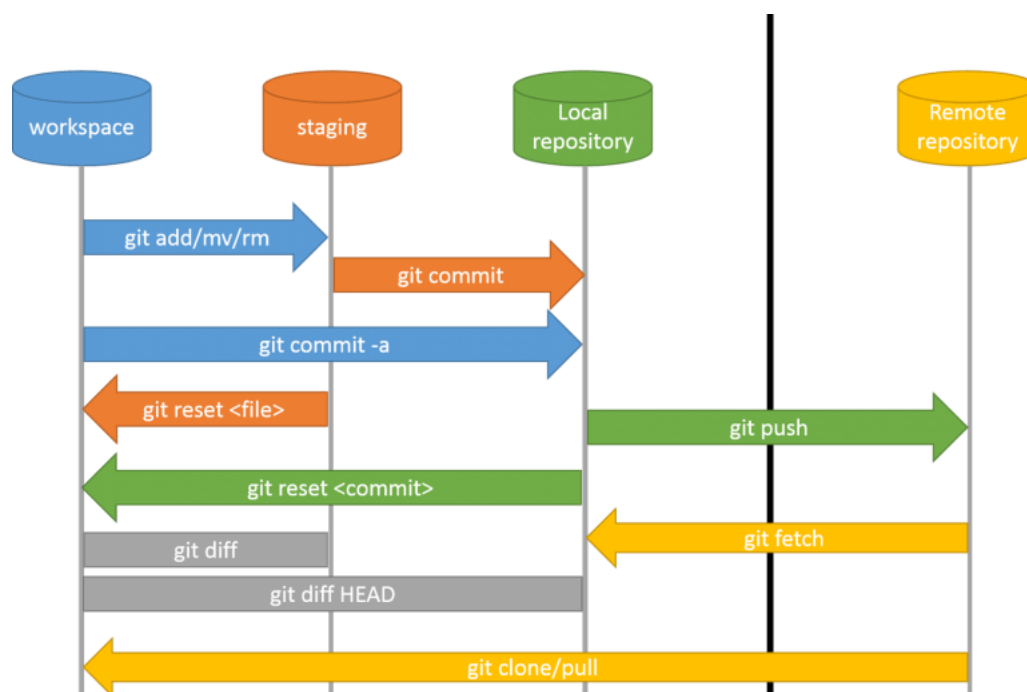


Figure 37: Git Workflow – Source (GitHub)

5.2.3 Key logical steps in Python

When defining all data storages, the following step was implementing the correct extraction of data and presenting decision-making data. As the Brick model is used instead of a complex tagging system, it needs to be queried correctly for accurate data extraction. The Brick model is RDF-based and can be queried using SPARQL. During the modelling process, the predicate that represents the relationship between the sensor and the database is "hasTimeseriesReference," according to the Brick Ontology Dictionary. So, the objective meaning of the SPARQL query (Figure 38) that was employed to recognize

the correlation between the sensor and database means: " Select all subjects and objects in a model that have the relationship "hasTimeseriesReference". The query was implemented with RdfLib.

```

14     q = """
15     PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
16     PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
17     PREFIX brick1: <https://brickschema.org/schema/1.3/Brick#>
18     PREFIX ex: <http://example.com#>
19
20     select ?s ?o
21     where {
22         ?s brick1:hasTimeseriesReference ?o .
23     }
24     """

```

Figure 38: Applied SPARQL query – Source (Own, Nvim screenshot)

Output from the query execution is a list of path's to databases – CSV files. After all paths to databases are known, last thing to be done was simple calculations like approximation, mapping and comparison. All calculations were performed using Pandas and NumPy.

After implementing the backend, the next step was to develop a simple frontend. Initial data visualisations were carried out using Jupyter notebook (Jupyter Lab, 2023). However, it quickly became clear that for non-linear application logic and a more interactive user interface (UI) it had to be replaced by something else. The Streamlit framework was chosen to stay within a Python environment. Streamlit is an open-source Python framework that facilitates the creation and sharing of applications for data analysis, data science, and machine learning (Streamlit, 2023). It functions as interactive puzzles that embody specific backend logic. The current version of the application has limited functionality (see Figure 39), which may be expanded in the future.

Temperature Heatmap

Put Brick model and optionally IFC

Load Brick Model

Drag and drop file here
Limit 200MB per file

Browse files

Load IFC

Drag and drop file here
Limit 200MB per file

Browse files

Select Values in Comfort database

Selecting Location

Ahmedabad

▼

Select Type of Facility

Office

▼

To build a temperature heatmap Brick model is required

Figure 39: Application frontend 1 – Source (Own)

This application allows to build a temperature heatmap based on data from sensors and be compared to optimal SET value from the ASHRAE Thermal Comfort Database. Possible heatmap is illustrated in

Figure 40. Heatmap may vary depending on randomized temperature data and location that can be set manually or extracted automatically from IFC.

Select Values in Comfort database

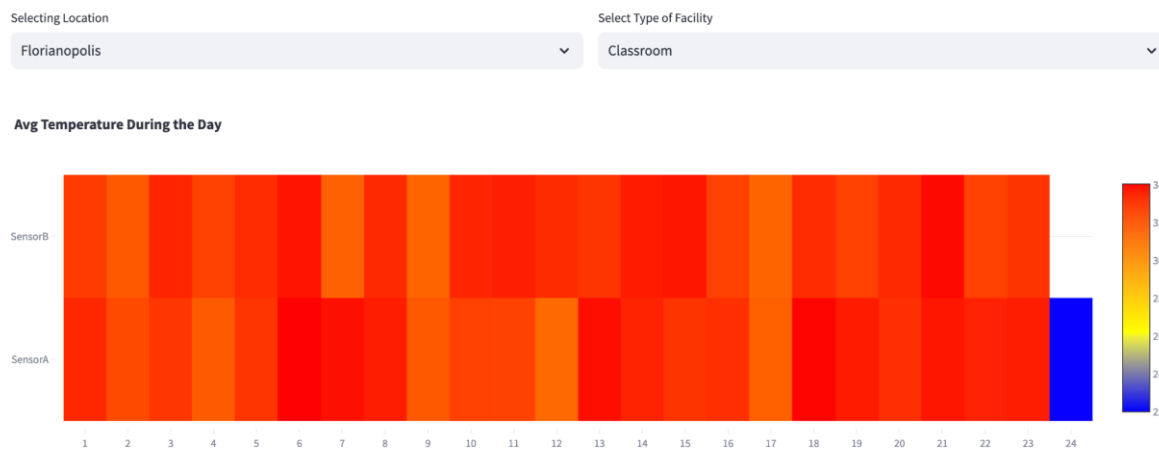


Figure 40: Heatmap – Source (Own)

Application testing is not currently covered, hence there may be errors or blank spaces. Nevertheless, facility managers can use this application for simple comfort assessment. Further instructions and documentation are available on GitHub.

5.3 Application deployment

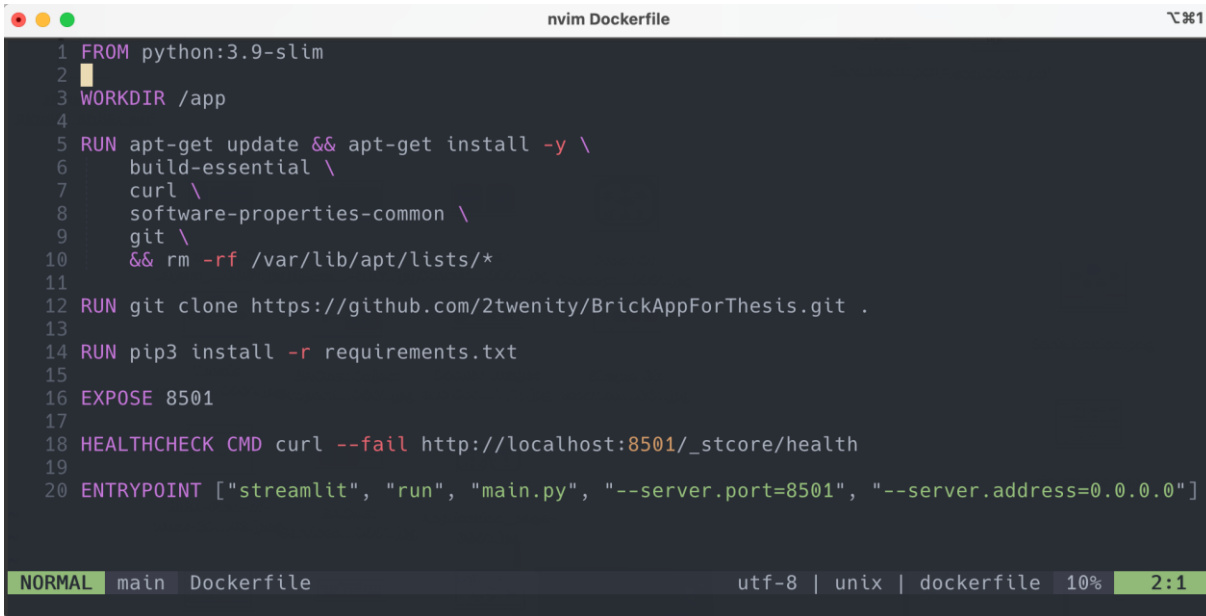
All the tools employed in the development process were open source and available for free use, just like this application. As it lacks commercial intent, there is no rationale for hosting it elsewhere and incurring costs. Consequently, the application must be run locally. To enable easier sharing and deployment, a Docker container was used.

5.3.1 Application in a docker container

Container technology was already introduced in the thesis. Purpose of the container in the scope of this application in particular is easier deployment and possibility of integration in into bigger application in the future. In other words the possibility of integration into continuous integration and continuous delivery (CI/CD) deployment.

Almost any application can be converted into a "Docker" container by adding a Docker file to the application folder. These containers function as a black box, and therefore the Docker file must include instructions for the container's logic execution and its path out of the black box. In the case of the Brick application, all the logic is written in Python, so a Docker image of Python 3.9 will be downloaded from Docker Hub. The next crucial step in the process involves downloading an application from a remote GitHub repository via a provided link. Once downloaded, Docker will execute the Python Package

Manager pip to install all the packages stipulated in the requirements file. Subsequently, Docker will specify the Local Host port number to establish a connection between the container and the PC. Finally, Docker provides instructions on accessing the appropriate localhost through the browser (Figure 41).

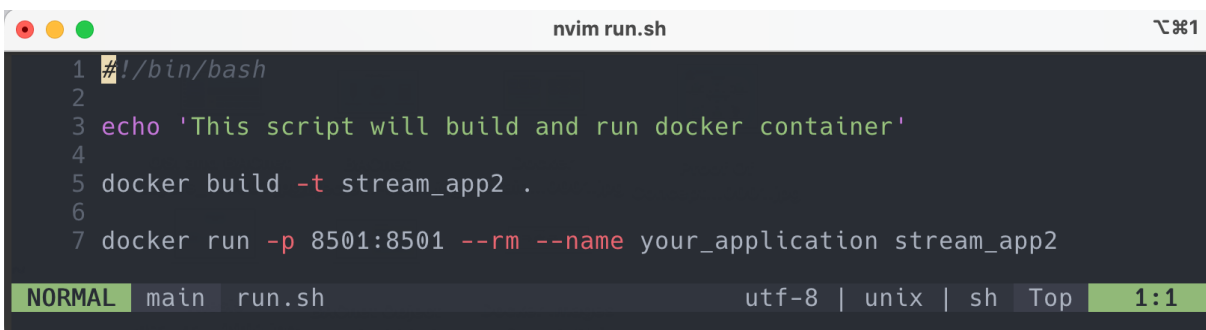
A screenshot of a Nvim editor window titled 'nvim Dockerfile'. The editor shows a Dockerfile with the following content:

```
1 FROM python:3.9-slim
2
3 WORKDIR /app
4
5 RUN apt-get update && apt-get install -y \
6     build-essential \
7     curl \
8     software-properties-common \
9     git \
10    && rm -rf /var/lib/apt/lists/*
11
12 RUN git clone https://github.com/2twenity/BrickAppForThesis.git .
13
14 RUN pip3 install -r requirements.txt
15
16 EXPOSE 8501
17
18 HEALTHCHECK CMD curl --fail http://localhost:8501/_stcore/health
19
20 ENTRYPOINT ["streamlit", "run", "main.py", "--server.port=8501", "--server.address=0.0.0.0"]
```

The status bar at the bottom shows 'NORMAL main Dockerfile utf-8 | unix | dockerfile 10% 2:1'.

Figure 41: Docker File – Source (Own, Nvim screenshot)

Executing Docker file produces a Docker image, which serves as a template for a Docker container. Docker automatically assigns names to images and containers, but for ease of access it is preferable to assign names manually using the "-t" flag. Once the Docker image is created, a container based on it can be started with specific flags. Firstly, to define the port as a path out of a Docker container, use the "-p" flag. The "--rm" flag ensures that the container is deleted after stopping. Users can also assign a custom name to the container instead of using the one assigned by Docker. These commands can be saved as a shell script and executed with a single command (see Figure 42).

A screenshot of a Nvim editor window titled 'nvim run.sh'. The editor shows a shell script with the following content:

```
1 #!/bin/bash
2
3 echo 'This script will build and run docker container'
4
5 docker build -t stream_app2 .
6
7 docker run -p 8501:8501 --rm --name your_application stream_app2
```

The status bar at the bottom shows 'NORMAL main run.sh utf-8 | unix | sh Top 1:1'.

Figure 42: Shell Script – Source (Own, Nvim screenshot)

Final version of application's structure is illustrated in Figure 43.

```
~/Documents/Python/FromGitHub tree BrickAppForThesis
BrickAppForThesis
├── 2RoomsFacility.ttl
├── Dockerfile
├── LICENSE
├── README.md
├── ashrae_db2.01_customized.csv
├── brick_model_maker.py
├── csv_data
│   ├── SensorA.csv
│   └── SensorB.csv
├── helper.py
├── main.py
├── pics
│   ├── BuildingHeatmap.png
│   ├── ChoosingLocation.png
│   └── ChoosingWindow.png
├── requirements.txt
├── run.sh
└── temperature_data_maker.py
```

Figure 43: Application Structure – Source (Own)

Docker image of application can be seen in a terminal by executing “docker images” command. This image also can be exchanged between users on Docker Hub as an executable application.

6 CONCLUSION

Modern Building Management Systems (BMS) must control numerous sensors and systems in a facility. All these entities require some level of abstraction to represent all the chaotic data in a structured way. Thus, the Brick Ontology was introduced, which contains a dictionary of rules and tags that can be used to encode the building into a structured RDF graph and serialise the building into a suitable format. Subsequently, computers can analyse facility data. To demonstrate its use, a simple model of a 2-room facility was created, structured in RDF and serialised in Turtle format using the Python RDFLib package. Furthermore, a basic application was developed to prove that this type of data can be used and analysed by computers. The application logic functions as follows:

- Queried the linked data model to retrieve all sensors and database paths where their stored data can be traced and normalizes and parses all data into a Pandas dataset.
- The ASHRAE thermal comfort database was parsed to obtain the SET temperature according to the user's chosen location or automatic selection from IFC.
- A comparison was made between the temperature from the sensors and database.
- The compared data was visualized in a simple front-end based on Streamlit.

Once the application was developed, it was encoded into a Docker image and made available for execution by running a shell script. Once executed, the application can be located as a Docker image on a hosting system or shared with users via Docker Hub.

6.1 Main Conclusions

Main conclusions can be pointed out as following:

- 1) Open source and non-proprietary abstractions can be utilised for energy analysis application development. Structured diagrams are suitable for storing heterogeneous data such as BMS points and HVAC systems.
- 2) Open source tools such as Python and free Python libraries provide sufficient capabilities to develop a proof of concept for an energy analysis application and ensure further scalability.
- 3) The Docker environment allows for the deployment of applications on any platform within a consistent environment and facilitates the sharing of applications between users. Docker is a suitable tool for developing portable and scalable building applications.

6.2 Future Work and Limitations

The application has been developed to proof of concept. The roadmap for further development could be as follows:

Cover current application with tests: Testing is one of the most important parts of application development, especially in CI/CD deployment. Therefore, for further development and integration, this application needs to be covered with tests to reduce the chance of bugs appearing.

BACnet integration: The current version of the application doesn't work with BACnet or any other BMS networking protocol. Further application development could follow the scheme shown in Figure 44.

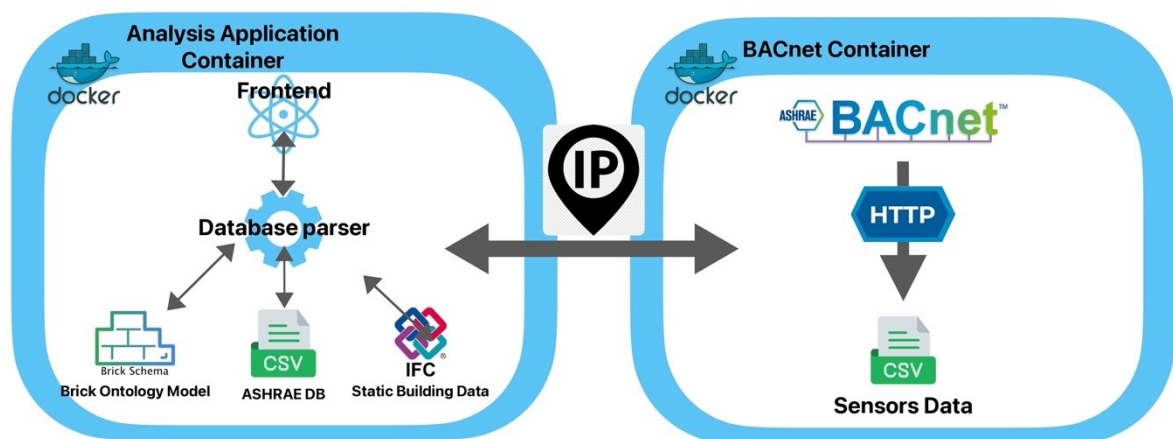
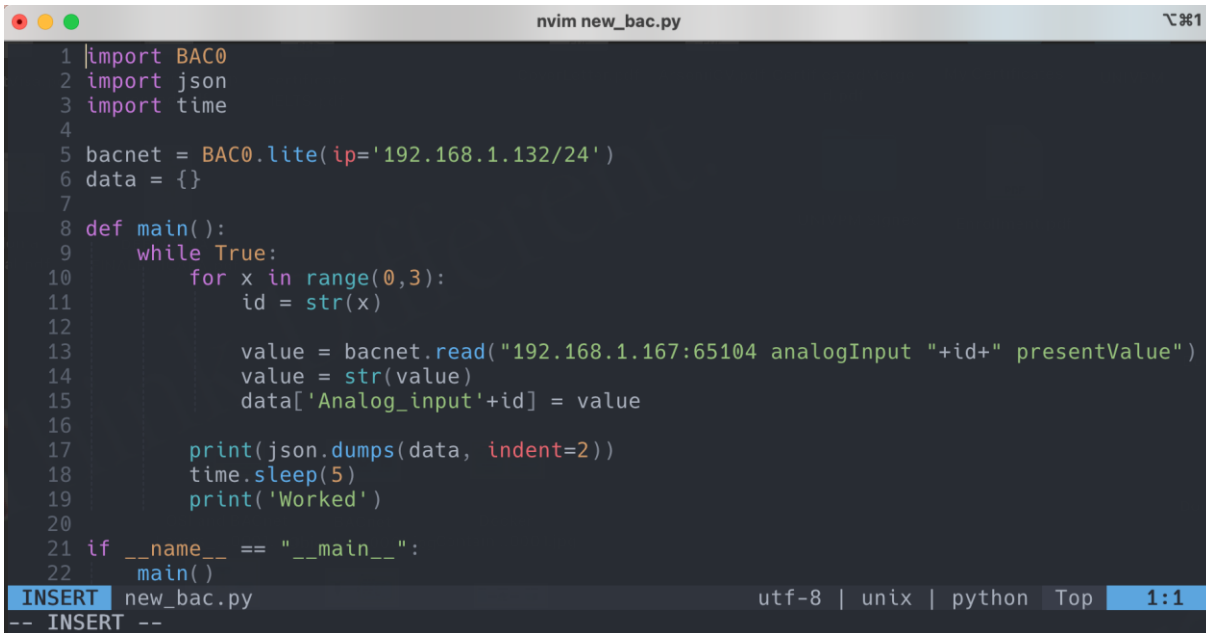


Figure 44: BACnet Integration – Source (Own)

A Docker container should be added to manage the collection and standardisation of BACnet data. IP should be used to exchange data between containers. This demonstrates that changing input into the main application is a straightforward process with Docker containers. The same applies to BACnet simulation. A Python script, as shown in Figure 45, was used to successfully retrieve data from a BACnet room simulation on another computer (YouTube).



```
1 |import BAC0
2 |import json
3 |import time
4
5 |bacnet = BAC0.lite(ip='192.168.1.132/24')
6 |data = {}
7
8 |def main():
9 |    while True:
10 |        for x in range(0,3):
11 |            id = str(x)
12
13 |                value = bacnet.read("192.168.1.167:65104 analogInput "+id+" presentValue")
14 |                value = str(value)
15 |                data['Analog_input'+id] = value
16
17 |            print(json.dumps(data, indent=2))
18 |            time.sleep(5)
19 |            print('Worked')
20
21 |if __name__ == "__main__":
22 |    main()
INSERT new_bac.py utf-8 | unix | python Top 1:1
-- INSERT --
```

Figure 45: BACnet data pulling Python script (IP addresses changed) – Source (Own, Nvim screenshot)

This script, or any other more complex BACnet interaction logic, can also be stored in a Docker container, which will have exposure ports for inter-container communication.

Ontology model validation: The application developed also didn't cover ontology validation. As mentioned previously, the latest version of Brick, 1.3, is switching from OWL to SHACL ontology, which will bring a model validation option. Brick model validation should be implemented to reduce a human factor in a model development process, as tools for machine conversion to Brick are not yet developed.

Higher IFC integration: The current version of the application has a low level of IFC integration as a source of static data. The next step could be to automate the creation of space and equipment entities in Brick based on IFC data. The IFC to Brick converter is under development by the Brick research team and hasn't been released yet.

Testing application on a larger scale: The Brick community has released Mortar testing software that includes already built Brick models of real-world facilities with a high percentage of coverage. These models could be used to test applications that use Brick models as a data source to ensure the scalability of an application. Mortar is outside the scope of application development due to technical issues.

Data visualizations improvement: The current version of the application provides visualisation as a heat map, which is not accurate enough for decision making. The analysis would be improved if the application showed a visualisation of historical data for each room. As the data of the operational phase is stored in a machine-readable format, some machine learning techniques could be applied to predict

the energy consumption of this facility, based on the consumption of similar facilities located in the same climate zone.

Brick ontology extensions integration and higher usage of ASHRAE database: In some case studies, the Brick ontology has been extended to store occupant data. Such an extension of the ontology could be used to integrate data collected from occupants and stored in the ASHRAE open source thermal comfort database.

REFERENCES

Balaji B. et al. 2018, Brick: Metadata schema for portable smart building applications, *Applied Energy* 226. Available at: <https://www.sciencedirect.com/science/article/pii/S0306261918302162>

BibLus, BIM maturity levels: from stage 0 to Stage 3. Available at: <https://biblus.accasoftware.com/en/bim-maturity-levels-from-stage-0-to-stage-3/> [Accessed 21.08.2023]

Brick. A uniform metadata schema for buildings. Available at: <https://brickschema.org/> [Accessed 30.08.2023]

Brick Ontology. Available at: <https://brickschema.org/ontology> [Accessed on 31.08.2023]

Building Smart. What is Open BIM? Available at: <https://www.buildingsmart.org/about/openbim/openbim-definition/> [Accessed 23.08.2023]

CIS - Center for Internet Security. Data Serialization. Available at: <https://www.cisecurity.org/insights/blog/data-deserialization> [Accessed 21.08.2023]

CMS Group. 6.02.2023. The 8 Riba Stages explained. Available at: <https://cms-group.co/the-8-riba-stages-explained/> [Accessed 25.08.2023]

Chipkin. BACnet – How is BACnet architecture designed? Available at <https://store.chipkin.com/articles/bacnet-how-is-the-bacnet-architecture-designed> [Accessed 28.08.2023]

CityJSON. A JSON-based encoding for 3D city models. Available at <https://www.cityjson.org/> [Accessed 15.08.2023].

DBpedia – Global and Unified Access to knowledge graphs. Available at <https://www.dbpedia.org/>

Dawson-Haggerty S. et al., BOSS: Building Operating System Services, *Computer Science Division, University of California, Berkeley*. Available at: https://www.researchgate.net/publication/262206891_BOSS_building_operating_system_service_s

Docker – Develop faster. Run anywhere. Available at: <https://www.docker.com/> [Accessed 1.08.2023]

Docker Hub – Build and Ship any application anywhere. Available at: <https://hub.docker.com/> [Accessed 1.08.2023]

Duarte Roa C. et al. 2022, Detecting Passing Valves at Scale Across Different Buildings and Systems: A Brick Enabled and Mortar Tested Application, *Lawrence Berkeley National Laboratory*. Available at: <https://escholarship.org/content/qt4xq5b54t/qt4xq5b54t.pdf>

Fierro G. et. Al 2019, Beyond a House of Sticks: Formalizing Metadata Tags with Brick. Available at: <https://dl.acm.org/doi/pdf/10.1145/3360322.3360862>

Fierro G et al. 2019, Mortar: An Open Testbed for Portable Building Analytics. Available at: <https://dl.acm.org/doi/pdf/10.1145/3366375>

Git – free and open source distributed version control system. Available at: <https://git-scm.com/>

Green Building XML (gbXML) Schema. Available at: https://www.gbxml.org/About_GreenBuildingXML_gbXML [Accessed 15.08.2023].

Haystack Blog. Quick Tutorial on the Turtle RDF Serialization. Available at: https://ai.ia.agh.edu.pl/_media/pl:dydaktyka:semweb:quick-tutorial-rdf-turtle.pdf

IfcOpenShell. The open source IFC toolkit and geometry engine. Available at: <https://ifcopenshell.org/>

Imperva. OSI Model. Available at: <https://www.imperva.com/learn/application-security/osi-model/>

Jupyter. Available at <https://jupyter.org/>

Li Jingming et al., Research on Brick Schema Representation for Building Operation with Variable Refrigerant Flow Systems, *Journal of Building Engineering*. Available at: <https://www.sciencedirect.com/science/article/pii/S2352710222008051>

Lia H. and J. Zhanga 2022, IFC-based Information Extraction and Analysis of HVAC Objects to Support Building Energy Modeling, *39th International Symposium on Automation and Robotics in Construction (ISARC 2022)*, *Automation and Intelligent Construction (AutoIC) Lab, School of Construction Management Technology, Purdue University, West Lafayette, IN 47907, USA*. Available at: https://www.iaarc.org/publications/fulltext/022_ISARC%202022_Paper_7.pdf

Ličina VF., Cheung T., Zhang H. 2018, Development of the ASHRAE Global Thermal Comfort Database II, *Building and Environment*. Available at: <https://www.sciencedirect.com/science/article/pii/S0360132318303652>

Luo Na. et al. 2022, Extending the Brick Schema to Represent Metadata of Occupants. *Automation in Construction*. Available at: <https://www.sciencedirect.com/science/article/pii/S0926580522001807>

Mavrokapnidis D. et al. 2023. A linked-data paradigm for the integration of static and dynamic building data in Digital Twins. *Conference: BuildSys '21: The 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. Available at: https://discovery.ucl.ac.uk/id/eprint/10146571/1/Balances_Workshop_pre_print.pdf

Mosiman C., Henze G., Els H. 2021, Development and Application of Schema Based Occupant-Centric Building Performance Metrics, *Energies*. Available at: <https://www.proquest.com/openview/05b0db5df7d15e2736550e4b2b4865ea/1?pq-origsite=gscholar&cbl=2032402>

Modbus tools. Available at: <https://www.modbustools.com/modbus.html> [Accessed 28.08.2023]

NumPy. The fundamental package for scientific computing with Python. Available at: <https://numpy.org/>

Ontotext. What are Linked Data and Linked Open Data (LOD)? Available at: <https://www.ontotext.com/knowledgehub/fundamentals/linked-data-linked-open-data/> [Accessed 29.08.2023]

Open Geospatial Consortium. Available at <https://www.ogc.org/> [Accessed 15.08.2023]

Pandas Python Library. Available at: <https://pandas.pydata.org/> [Accessed 10.07.2023]

Pritoni M et al. 2021, Metadata Schemas and Ontologies for Building Energy Applications: A Critical Review and Use Case Analysis, *Energies*, Available at: <https://www.mdpi.com/1996-1073/14/7/2024>

Rdflib – pure Python package for working with RDF. Available at: <https://rdflib.readthedocs.io/en/stable/> [Accessed 25.07.2023]

Ref-Schema. Available at: <https://github.com/gtfierro/ref-schema> [Accessed 30.08.2023]

SPIN. SHACL and OWL Compared. <https://spinrdf.org/shacl-and-owl.html> [Accessed 31.08.2023]

Setra. What is the difference between BACnet, Modbus and LonWorks? Available at: <https://www.setra.com/blog/what-is-the-difference-between-bacnet-modbus-and-lonworks>

Streamlit – A faster way to build and share data apps. Available at: <https://streamlit.io/> [Accessed 3.06.2023]

W3C. Tim Berners-Lee. Available at: <https://www.w3.org/DesignIssues/LinkedData.html> [Accessed 29.08.2023]

W3C. RDF Concepts and Abstract Syntax. Available at: <https://www.w3.org/TR/rdf12-concepts/>

W3C. RDF Primer – Turtle version. Available at: <https://www.w3.org/2007/02/turtle/primer/> [Accessed 29.08.2023]

YouTube. Read data from BACnet devices over BACnet/IP using Python. Available at: <https://www.youtube.com/watch?v=TyEXDnjBsD8> [Accessed 23.07.2023]