

Bau Geo Umwelt

Chair of Computational Modeling and Simulation

Prof. Dr.-Ing. André Borrmann

Development of a BIM-enabled Software Tool for Facility Management using Interactive Floor Plans, Graph-based Data Management and Granular Information Retrieval

Daniel Zibion

Masterthesis

for the Master of Science program Civil Engineering

Author:	Daniel Zibion
Student Number:	03627702
Supervisors:	Prof. Dr.-Ing. André Borrmann D.Phil. Vishal Singh Alexander Braun
Advisors:	D.Sc. Seppo Törmä D.Sc. Mehmet Yalcinkaya
Date of issue:	01. April 2018
Date of submission:	01. September 2018



Author Daniel Josef Zibion

Title of thesis Development of a BIM-enabled Software Tool for Facility Management using Interactive Floor Plans, Graph-based Data Management and Granular Information Retrieval

Master programme Building Technology

Code ENG27

Thesis supervisor Vishal Singh

Thesis advisor(s) Seppo Törmä

Date 01.09.2018

Number of pages 109

Language English

Abstract

Since its very conception Building Information Modeling incorporates the notion of using digital models—rich in geometric and semantic information—throughout the whole life cycle of a building. The creation of these models is a process tied to much effort, split by disciplines, executed by different parties and brought together under difficult collaboration. However, in reality the effective utilization of the BIM process ends with the conclusion of the construction project. The subsequent Operation & Management phase makes little to no use of the information contained in these files, although it would be valuable resource to boost productivity.

Especially the Facility Management phase, suffers from great inefficiency caused by challenges of data management and outside advances in digitization. Research suggests that BIM is able to provide benefits for processes in FM and O&M related tasks and increase their overall efficiency, but previous attempts to introduce BIM software have remained fruitless. We argue that current solutions have failed to meet expectations and requirements by FM community, which generally lack expertise in working with CAD-like software. Instead this thesis presents a concept which puts interactive, two-dimensional floor plans at the center of a possible BIM-enabled Facility Management (FM) software tool.

These floor plans are directly derived from BIM models and maintain linkage to all relevant semantic data, which is stored in a graph database. Users are able to navigate rooms, equipment and themselves on the floor plans. Further information about rooms can be accessed through 360° photo spheres—enabling remote exploration and conception—and room specific 3D model. The latter is generated beforehand and follows the underlying concept that FM seldomly requires a holistic view of the whole building but instead a cross section of many different domain models, tied by a specific location. Based on the mentioned features and concepts a prototypical web application is developed in order to investigate the feasibility and effectiveness of the proposed solution.

Keywords BIM, Facility Management, IFC, Graph Database, Floor Plan, Web Application

Abstract

Since its very conception Building Information Modeling incorporates the notion of using digital models—rich in geometric and semantic information—throughout the whole life cycle of a building. The creation of these models is a process tied to much effort, split by disciplines, executed by different parties and brought together under difficult collaboration. However, in reality the effective utilization of the **BIM** process ends with the conclusion of the construction project. The subsequent Operation & Management phase makes little to no use of the information contained in these files, although it would be a valuable resource to boost productivity. Especially the Facility Management phase, suffers from great inefficiency caused by challenges of data management and outside advances in digitization. Research suggests that **BIM** is able to provide benefits for processes in **FM** and **O&M** related tasks and increase their overall efficiency, but previous attempts to introduce **BIM** software have remained fruitless. We argue that current solutions have failed to meet expectations and requirements by **FM** community, which generally lack expertise in working with **CAD**-like software. Instead this thesis presents a concept which puts interactive, two-dimensional floor plans at the center of a possible **BIM**-enabled Facility Management (**FM**) software tool. These floor plans are directly derived from **BIM** models and maintain linkage to all relevant semantic data, which is stored in a graph database. Users are able to navigate rooms, equipment and themselves on the floor plans. Further information about rooms can be accessed through 360° photo spheres—enabling remote exploration and conception—and room specific **3D** model. The latter is generated beforehand and follows the underlying concept that **FM** seldomly requires a holistic view of the whole building but instead a cross section of many different domain models, tied by a specific location. Based on the mentioned features and concepts a prototypical web application is developed in order to investigate the feasibility and effectiveness of the proposed solution.

Contents

1	Introduction	1
1.1	Problem Statement	1
1.2	Motivation	2
1.3	Outline of the Thesis	3
2	Facility Management	4
2.1	Definition	4
2.2	Information Management	5
2.3	Opportunity for BIM-enabled FM	7
2.4	Barriers of BIM Integration	9
2.5	Summary	10
3	Current BIM FM Technology	11
3.1	Introduction	11
3.2	Construction Operations Building information exchange COBie	11
3.2.1	Overview	11
3.2.2	Problems with the COBie Standard	13
3.3	BIM integrated Software Tools	14
3.4	Challenges	15
4	Concept	16
4.1	Requirements	17
4.2	Solution Approach	18
4.2.1	Extracting floor plans from IFC Geometry	19
4.2.2	Export of semantic data from IFC	19
4.2.3	Implementation of a web-based interface	20
4.2.4	Integration of panoramic images and model viewer	20
4.3	Summary	20
5	Technical Foundations	22
5.1	Industry Foundation Classes	22
5.1.1	A Brief History	23

5.1.2	Architecture	24
5.1.3	Objects and their Relations	25
5.1.4	Application Scope of IFC	26
5.2	Graph Database	26
5.2.1	Definition	27
5.2.2	Characteristics	28
5.2.3	Application	29
5.3	Scalable Vector Graphics	30
5.3.1	Definition and Functionality	30
5.3.2	Application	31
5.4	Single Page Application	31
5.4.1	Traditional Multi-Page Approach	32
5.4.2	Single Page Approach	34
5.4.3	Application	34
6	Floor Plans	36
6.1	Purpose	37
6.2	Problems of 2D and how to solve them	38
6.2.1	Orientation and Navigation	38
6.2.2	Information Loss	40
6.3	Generation from IFC	41
6.4	Export to SVG	43
6.5	Summary	44
7	Graph-based Data Storage	45
7.1	Why a Graph Approach?	45
7.2	Graph Data Model	46
7.2.1	Best Practises	47
7.2.2	IFC Data Model	48
7.2.3	Facility Management	50
7.3	Graph Generation	51
7.3.1	IFC Objects	51
7.3.2	IFC Relations	53
7.3.3	Inter-domain Linking	54
7.3.4	MEP Systems	55
7.3.5	System Flows	56
7.4	Access	58
7.4.1	Get Spaces on Storey	59
7.4.2	Connectivity between Spaces	60
7.4.3	Supplying Air System for Selected Room	61

8	Software Tools and Libraries	63
8.1	Python-based Preprocessing	64
8.1.1	IfcOpenShell	64
8.1.2	PythonOCC	64
8.1.3	Py2neo	65
8.2	Neo4j	65
8.2.1	General	65
8.2.2	Cypher	66
8.3	Web Application	67
8.3.1	Node.js	68
8.3.2	React	68
8.3.3	Pannellum	69
8.3.4	Autodesk Forge	70
9	Prototypical Web Application	72
9.1	Use Cases	72
9.1.1	User Groups	72
9.1.2	Scenarios	73
9.2	Architecture	74
9.3	User Interface	75
9.3.1	Navigation Bar	77
9.3.2	Viewport	77
9.3.3	Sidebar	78
9.3.4	Modal	78
9.4	Interactive Floor Plan	79
9.4.1	Base Floor Plan	79
9.4.2	MEP Systems	80
9.5	Building Space Selection	81
9.5.1	Overview	81
9.5.2	Serving Systems	82
9.5.3	Photo Sphere	83
9.5.4	3D Space Model	84
9.6	Work Order	85
9.6.1	Creation	85
9.6.2	Linkage	86
10	Conclusion	87
10.1	Summary	87
10.2	Challenges	89
10.3	Future Improvements	90

10.3.1 Improved Graph Generation	91
10.3.2 Improved Web Service	91
10.4 Outlook	92

Abkürzungsverzeichnis

2D	Two-dimensional
3D	Three-dimensional
ACID	Atomic, Consistent, Isolated and Durable
AEC	Architecture, Engineering and Construction
API	Application Programming Interface
BAS	Building Automation Systems
BIM	Building Information Modeling
CAD	Computer-aided Design
CAFM	Computer-aided Facility Management
CDE	Common Data Environment
CMMS	Computerized Maintenance Management Systems
COBie	Construction Operations Building information exchange
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheets
DBMS	Database-Management System
DOM	Document Object Model
EDMS	Electronic Document Management Systems
EMS	Energy Management Systems
FM	Facility Management
GUID	Globally Unique Identifier
HTML	Hypertext Markup Language
IAI	International Alliance for Interoperability
IDM	Information Delivery Manuals
IFC	Industry Foundation Classes
IFMA	International Facility Management Association
IoT	Internet of Things
JSX	JavaScript XML
MEP	Mechanical, Electrical and Plumbing
MVD	Model View Definition
NBIMS	National Building Information Modeling Standard
NIBS	National Institute of Building Sciences

O&M	Operation & Management
SQL	Structured Query Language
STEP	Standard for the Exchange of Product Model Data
SVG	Scalable Vector Graphics
SPA	Single-Page Applications
UI	User Interface
UUID	Universal Unique Identifier Standard
XML	Extensible Markup Language
VR	Virtual Reality
W3C	World Wide Web Consortium

Chapter 1

Introduction

The adoption of Building Information Modeling (BIM) is steadily progressed in countries all over the world and is driven by the prospect of an increased productivity, quality of design and an improved collaboration between project partners. While some argue if BIM actually has been able to generate the promised benefits already (Miettinen and Paavola [2014]), there is no denying that numerous efforts have been made with valuable lessons learned on the way—especially during the design and construction phases. The BIM mentality of course doesn't stop with the completed construction of a building but includes the full life cycle; from design, construction over operations and maintenance of a building, till its eventual demolition.

At least this virtue is claimed since the early conception of BIM and has been repeated many times, despite the fact that to this day only few attempts have been made to push BIM beyond the construction phase and even fewer hard evidence has been collected [Kiviniemi and Codinhoto, 2014]. Yet, increased efficiency and improved information management would be direly needed in Facility Management and Operation & Management domain in general, which suffer greatly from data dispersal [GCR, 2004]. Data integration and interoperability could answer the problems, owners and facility managers are facing and indeed BIM has been conceived to deal with these very issues in mind. For this reason and despite the many barriers researcher have found for its successful integration into FM and Operation & Management (O&M) processes [Becerik-Gerber et al., 2011], the majority supports the benefits of a BIM integration and calls for actions.

1.1 Problem Statement

Building Automation Systems (BAS), Internet of Things (IoT) sensors and a general lifestyle that is increasingly determined by digital services are posing a challenge for future-proof FM

and O&M companies. BIM seems to have the capabilities to answer these issues of information management by providing FM and O&M representatives with means to tackle issues of interoperability, augment data with digital representations of facilities and by serving as a source of information—yet the gap between construction and operations persists. Researchers have found different reasons for that, but the general consensus seems to be that they are rooted in current work practises and organizational structures [Kiviniemi and Codinhoto, 2014]. The fundamental differences between a project based organization during design and construction versus the life cycle based view that dominates after the handover appears to be a dominant source of the observed disjunction between those two phases. There is, of course, more to it than that, but the exact causes are complex and intertwined and will be discussed in more detail in 2.4.

Taking a closer look ahead, reveals another layer of challenges, which concerns the few existing software tools that are already addressing the integration of BIM into the FM domain; in many cases focused around a powerful Three-dimensional (3D) model viewer. Those tools have arguably failed to generated the expected resonance of FM practitioners. Instead popular competing tools, so called Computerized Maintenance Management Systems, are solely focused on management tasks, pursue a mobile first approach and are omitting any of the powerful 3D visualization capabilities the BIM counterparts are providing. (see 2.2) Once more, there seems to be a disjunction between exceptions on the part of solution providers and the expected end user groups. In fact, architects and engineers—the main drivers in the design of current BIM FM solutions—distinguish themselves greatly from O&M domain, who are generally lacking any experience in the use of Computer-aided Design (CAD)-like solutions. 3D visualization alienates them, the interfaces tend to be overly complicated and core FM use cases are not clearly supported.

1.2 Motivation

This thesis aims to contribute to the academic discourse over BIM-enabled FM and help the process of further shrinking the gap between the two phases. A core principle of the introduced application is the abandonment of a 3D first approach. Instead we presents the development of a web application that puts interactive Two-dimensional (2D) floor plans at its center. These plans are extracted directly from Industry Foundation Classes (IFC) files and preserve their connection to other data items inside the file by storing Globally Unique Identifier (GUID)s alongside the exported Scalable Vector Graphics (SVG) format. By utilizing a third-party library we aim to further enrich semantics inside a single model file and establish links between different domain specific files through, among other things, the creation and intersection of bounding boxes.

Semantic data residing inside the IFC shall be made accessible by extracting and storing it in a hosted graph database. Graph databases are a type of NoSQL database that uses graph theory to store and retrieve data in form of nodes and relationships. The main benefits of graphs databases is the effective representation of highly connected data and the possibility to query complex relations relatively easy and with short execution times. We argue that the IFC structure is well-suited to take full advantage of the potential of graph databases and we are presenting fitting use cases from the FM domain alongside.

The loss of information in the third dimension is coped twofold by integrating 3D features in a focused and piecemeal fashion. The first feature includes a realistic photo sphere of rooms, that allows FM managers, repair personnel and other users to explore spaces remotely and in advance. Secondly, we integrate focused 3D models, that are extracted directly from IFC but only contain structures and systems directly relating to a given room. Our prognosis is that this approach grants more granular access of information and supports the work flows of FM practitioners in a more focused manner. The implementation of the presented concepts serves as a proof of their feasibility, suitability and effectiveness.

1.3 Outline of the Thesis

Chapter 2 starts by giving insights into the current state of Facility Management and the challenges the industry is facing. This is followed up by the next chapter 3, which highlights the few existing BIM-enabled FM tools.

Afterwards, the overall concept of our application is presented in greater detail in 3, discussing the concepts we envision to tackle the shortcomings of existing solutions. Next we introduce the fundamental technologies needed, in order to realize the presented software prototype.

The following two chapters 6 and 7 detail first the floor plan as the key graphical element of our application and its creation and secondly the graph database itself and aspects attached to it. Libraries and frameworks used during the development of the prototype are listed in chapter 8 before introducing its actual development, appearance and functionality in the chapter after. We end with a conclusion summarizing the gained insights and outlining future development.

Chapter 2

Facility Management

2.1 Definition

Facility Management is a relatively young profession that has been evolving since the 1980s from scattered individuals maintaining single buildings to managers overseeing large portfolios of various buildings.



Figure 2.1: 11 core competences of Facility Management according to [IFMA](#).

It is partly due to this evolution and the general diversity of the industry that [FM](#) is a term with unclear definition. The most recognized attempt of specification comes from the [IFMA](#) and states: “Facility management is a profession that encompasses multiple disciplines to ensure functionality of the built environment by integrating people, place, process and technology.” [Wagnon, 2009]. Furthermore, [IFMA](#) identified 11 core competences of the industry

(see figure 2.1), by conducting task analyses and interviews with industry professionals around the world; a further testimony to the diversity of the field.

While *IFMA* highlights diversity and collaboration, Atkin and Brooks [2014] are stressing the possible benefits for business operations by defining *FM* as an integrated approach to maintain, improve and adapt an organization's building to promote a fertile environment that supports the organization's primary objectives. No matter the viewpoint, the overall consensus is that *FM* is an interdisciplinary craft that brings together people from highly specialized fields, connected by the building facility and the potential to enhance its operations decisively. Consequently *FM* becomes a intersection for data from various fields and the correct organization, storage and serving of this data is a big challenge that still needs to be sufficiently answered.

2.2 Information Management

There is arguably no other stage during a life cycle of a building that has to deal with a larger flux of information than the operational phase. Information gathered over months

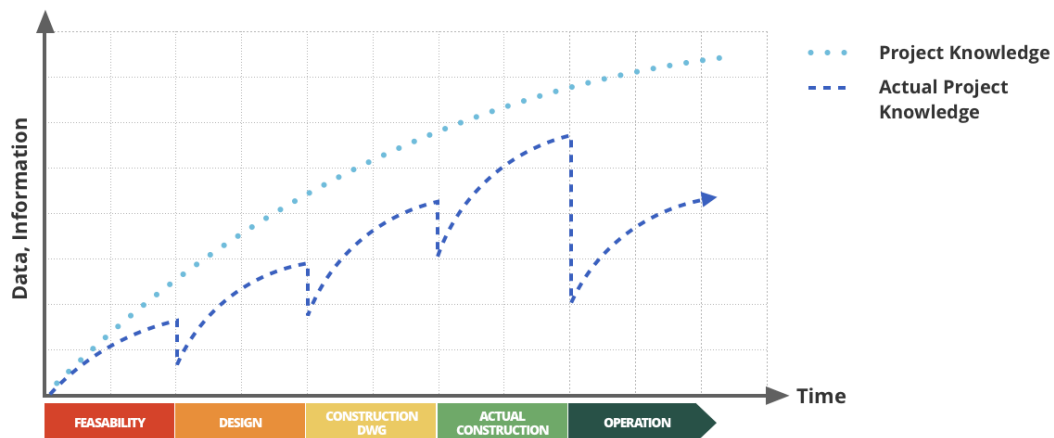


Figure 2.2: Project knowledge throughout the various construction project phases.

during design and construction and by various project partners and disciplines are delivered in a large bundle to the building owner and affiliates during the hand over. This data is potentially useful for various cases in later stages of the building life cycle and can be used to e.g. initialize databases of the *FM*.

On top of that, much more data is accumulated during the actual operation of the building, such as history of work orders and repairs, incoming mails, data about occupants, security footage, sensor data and much more. Many of the day-to-day tasks of facility managers require access to part of this information or would be accelerated if such access would exist. The value lost by spending hours on locating various information items is immense and was

put to numbers by a study by the National Institute of Standards and Technology in the U.S. This report from the year 2004 studied expenses due to issues of interoperability throughout the whole life cycle of a building and among various stakeholder groups. Two-thirds of the added cost is represented in the owners and operators, with most of the cost being generated during the the operational phase of the building. This mismanagement is summarized in the report by the following quote:

“An inordinate amount of time is spent locating and verifying specific facility and project information from previous activities. For example, as-built drawings (from both construction and maintenance operations) are not routinely provided and the corresponding record drawings are not updated. Similarly, information on facility condition, repair parts status, or a project’s contract or financial situation is difficult to locate and maintain.” [GCR, 2004].

The current shortcomings haven’t passed unnoticed and for the last years FM companies have attempted to solve the issue by introducing FM information systems such as Computerized Maintenance Management Systems (CMMS), Electronic Document Management Systems (EDMS), Energy Management Systems (EMS) and BAS [Becerik-Gerber et al., 2011]. Key component of this technology stack are the CMMS systems, which are responsible of managing the database of information about maintenance and human resource functions [Cato and Mobley, 2001].

The market of CMMS is special due to the fact that it is characterized by a very high number of individual solutions, with only few larger software companies serving a global market and many more local providers. This constellation is further disrupted with the uprising of new mobile-first start-ups like UpKeep, HippoCMMS or Fiix that are gaining in popularity. Naturally this competitive environment produces software products with varying focal points and solution approaches.

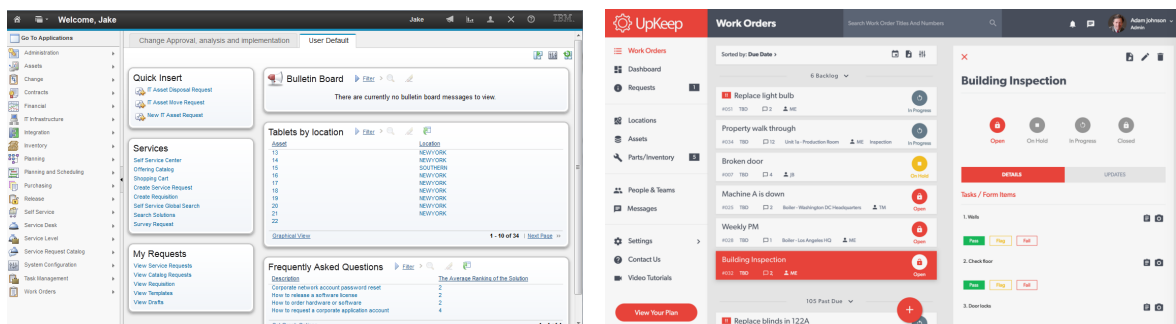


Figure 2.3: Old vs new: Interfaces of IBM Maximo as an established player on the left, next to Upkeep, a solution of rising popularity.

There is a certain set of core features that are addressed by any CMMS, no matter what, for example, work order management, asset administration, inventory management and task scheduling. CMMSs generally deal with the management of data that is produced during the operational phase. The facility itself—but also the various machinery inside it—contains

a magnitude of information, which has to be manually inputted into the systems; a time consuming and error prone task. Also, while almost all day-to-day data used in FM has spatial context to some extent, there exists no features that would provide such as part of their interfaces.

BIM is already well established in the construction industry and has the potential to not only provide spatial context but also to automate the laborious initial input of data by extracting information valuable for operations directly from the building models.

2.3 Opportunity for BIM-enabled FM

The benefits of BIM in FM has been conceived theoretically since the early beginnings of the BIM development, promising an increase in productivity and an improved transfer of information. These claims were commonly vague in their description of how exactly facility manager could benefit from it and were not backed by any hard evidence.

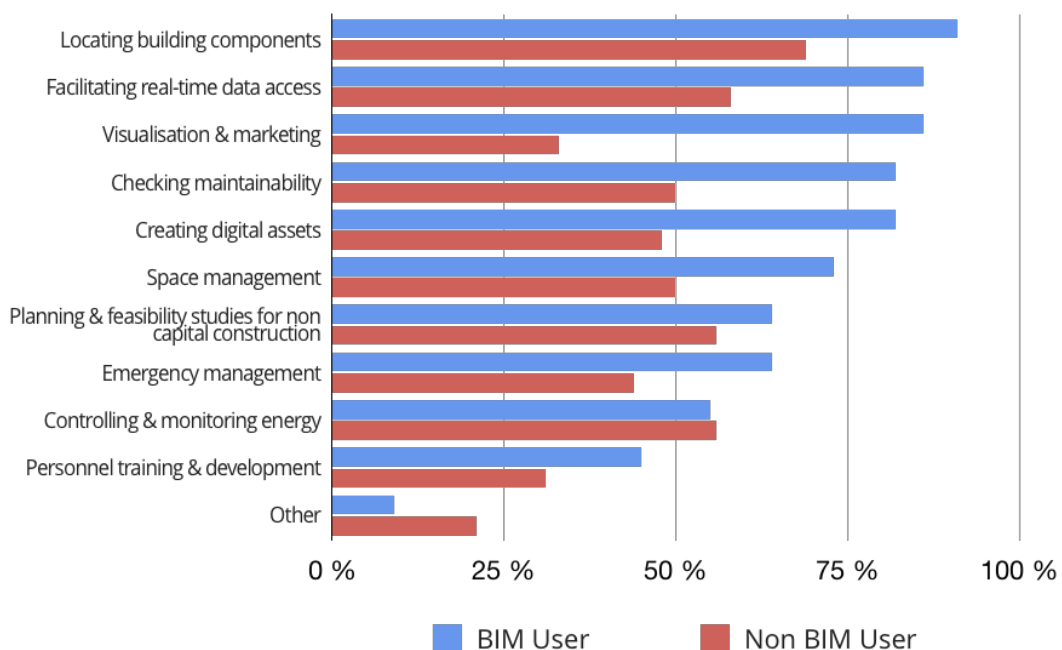


Figure 2.4: Potential BIM FM application areas as uncovered by the conducted studies (based on [Becerik-Gerber et al., 2011]).

Becerik-Gerber et al. [2011] initiated a first attempt to consolidate these claims by conducting 22 face-to-face interviews and 77 online surveys with FM professionals. The main objectives of the study were: 1. Identification of potential application areas for BIM in FM, 2. Definition of data exchange requirements, 3. The current status of the implementation of BIM in FM and 4. Challenges and barriers. During the course of her investigation, Becerik and her team

found a list of potential BIM FM uses, as supported by either BIM users or users outside the domain (see figure 2.4).

Among these, she and her team recommends to implement the following with priority: locating components, facilitating real-time data access, checking maintainability and automatically creating digital assets. These aspects touch on some of the more urgent challenges in the FM domain such as making implicit knowledge, hidden in the expertise of individuals, explicit and moving from reactive to a preventive style of maintaining facilities. BIM has the potential to bring these very benefits to the table.

Similar findings were made by [Codinhoto and Kiviniemi, 2014] in their assessment of a renovation project in Manchester. While their evaluation focused more on the challenges and barriers of an BIM in FM integration they also identified certain benefits and were able to back them with numbers. Figure 2.5 shows the different FM tasks that were monitored including a comparison between the conventional and BIM-enabled approach Overall they recorded a significant reduction in man hours needed per considered task (ranging between 57%...80%). The individual saving were relatively low and ranged between £108 and £838, but due to the high frequency of FM tasks this can add up to significant annual savings. However the largest benefit recorded, comes from the reduction in waiting time, which ranged between 96% and 99%. Responding as fast as possible to disruption not only helps to create a positive image of the operation business but also prevents potentially large expenses created through down times.

Scenario	Traditional		BIM Enabled		Savings		
	Time	Man/hours	Time	Man/hours	Time	Man/hours	£
Ventilation motor replacement	4 weeks	14	1 day	3	27 days	11	£286.00
Alcove light replacement	6 weeks	10	1 day	2	40 days	8	£108.00
Extract duct - unknown water build up	12 weeks	23	1 day	10	92 days	13	£838.00
Public lift repair	5 weeks	16	2 days	6	32 days	10	£260.00
Ceiling leak in heritage area	3 days	14	1 day	3	2 days	11	£286.00

Figure 2.5: Benefits and returns from BIM FM [Codinhoto and Kiviniemi, 2014]

More recent case studies are presented by Teicholz et al. [2013], who lists a total of six throughout the United States of America. In contrast to Codinhoto and Kiviniemi [2014], the presented case studies here focus almost solely on the actual construction, and do not reflect the operation phase well. Still, they give an interesting insight in the current challenges that exists at the moment, like the fact that most of the work is still done by the design team, which is lacking FM expertise and the general reluctance to allocate more investment for proper management and storage of the various data items.

The above cases are pioneering studies in a field that is still very much insufficiently studied. However their early results suggest that BIM has the potential to answer issues of integration and interoperability that exist in the current field of FM, increase the quality of information by providing spatial context and reducing the time searching for relevant information. These

benefits are not without cost and are accompanied by conflicts and barriers preventing a successful integration so far.

2.4 Barriers of BIM Integration

Again Becerik-Gerber et al. [2011] have been laying the groundwork in researching the problems of a successful BIM in FM integration. Their findings can be classified in challenges regarding *technologies and processes* and challenges regarding the *organizational structure*, as described in the following:

1) Technology and process related challenges

- *Unclear roles and responsibilities for loading data into the model or databases and maintaining the model;*
- *Diversity in BIM and FM software tools, and interoperability issues;*
- *Lack of effective collaboration between project stakeholders for modeling and model utilization;*
- *Necessity yet difficulty in software vendor's involvement, including fragmentation among different vendors, competition, and lack of common interests*

2) Organizational challenges:

- *Cultural barriers toward adopting new technology;*
- *Organization wide resistance: need for investment in infrastructure, training, and new software tools;*
- *Undefined fee structures for additional scope;*
- *Lack of sufficient legal framework for integrating owners' view in design and construction;*
- *Lack of real-world cases and proof of positive return of investment.*

These aspects are picked up and elaborated further by Kiviniemi and Codinhoto [2014]. They identify the difference between project-based working methods as applied during design and construction versus life cycle management as applied during the operational stage as a root obstacle of collaboration. Further, there is a general non-understanding between BIM experts that almost exclusively have backgrounds in either construction or architecture and facility managers which are experts in their respective fields but are generally unaware about BIM and practices in construction projects. However, in order to make BIM models valuable during the operational phase, information about e.g. building equipment and further relevant meta-data linked to it, has to be added to the modeling files.

Ensuring such an input of information requires the integration of **FM** experts as early as possible during the design and construction process. Since they are generally lacking expertise in the handling of **BIM** and modeling software, the input of such data is left to designers, who in turn have little knowledge about the subsequent **FM** requirements [Codinhoto and Kiviniemi, 2014]. On top of that, the novelty of the **BIM FM** integration has the consequence that there are no existing best practices or product libraries that would guide participants through the process of identifying relevant information, in short: Oftentimes not even owners or **FM** practitioners know at that stage what information they would require for their later business.

All of this creates an environment that discourages owners and **FM** experts to participate during the project phase and move responsibility to the design team, a dilemma that prevents the establishment of best practices and collaboration. Another layer of complexity is presented with the integration of already constructed buildings. Organizations that operate facilities in a large scale, generally own a considerable amount of older buildings without **BIM** models [Kiviniemi and Codinhoto, 2014]. Should those models be modeled in retrospective, how much would such a process cost or is there the possibility of facilitating a hybrid system that supports both existing and newly constructed, thus modeled buildings at the same time?

2.5 Summary

Facility Management is highly interdisciplinary field that currently deals with major challenges of information management and a client base that is further accustomed to digital services. **BIM** has the potential of provide the foundation of such digital solution by providing renderings of **3D** and **2D** models and by offering a rich set of semantic information. In order to make models accessible and valuable past the construction phase, **O&M** stakeholders need to be increasingly involved in the planning process. This is a challenge for both sides, which have divergent backgrounds, little understanding of the opposing side and no prior best practises to fall back on to. This can only be solved by changing existing processes, contractual frameworks and, last but not least, the mindset of people. Incentives by the public sector can be a crucial motor to ignite the needed transformation, which is already happening to a certain extent in the UK. Time will tell if the existing barriers will be overthrown. Despite all of this, some visionary companies and research teams already attempted first running products and prototypes of **BIM** integrated **CMMS** tools. The next chapter will examine those tools more closely and will take their lessons learned as an input for our envisioned prototype of a **BIM** integrated **CMMS** tool.

Chapter 3

Current BIM FM Technology

3.1 Introduction

Section 2.4 demonstrated that there are still many barriers preventing a successful implementation of BIM in FM practices. Despite all of that, an increasing number of companies and research groups are working on solutions to push forward and enable BIM techniques in the FM domain. This recent trend has been fired up by the decision of the UK government to demand the implementation of BIM level 2 for all public construction projects, which includes Construction Operations Building information exchange as a data exchange format for buildings operations [CDBB, 2009]. Construction Operations Building information exchange (COBie) is an international standard that defines and organizes information of assets and technical equipment of buildings into spreadsheets. These spreadsheets can be used to as an information source for facility information systems and are becoming to an extend a standard in the UK with mixed reception by practitioners [Pärn et al., 2017] In the following, first we discuss COBie in more detail before presenting some of the mentioned revolutionary software tools that are attempting said integration and the remaining challenges we see unanswered so far.

3.2 Construction Operations Building information exchange COBie

3.2.1 Overview

COBie is an exchange format that directly addresses the handover of building information from the contractor to the owner and stakeholders responsible for future operations and management of the building [Borrmann et al., 2015]. Thereby the focus is less on the geometry

of the building and instead on the actual rooms and connected technical equipment serving those spaces. Traditionally and as mentioned in earlier sections, the information regarding the building were provided to the owner at a single point in time during handover. This type of information transfer is a large challenge for the receiver, who finds himself flooded with information and the prospect of many hours of work to make information accessible.

COBie outlines a method that collects the needed information throughout the design and construction process and serves it to the owner during commissioning and handover in an easy-to-implement manner [Eastman et al., 2011]. However the specifics of what kind of information shall be included in the COBie exchange format, has to be agreed on between the owner and various other stakeholders. COBie was introduced by Bill East in 2007 in order to optimize the management of assets of the many facilities managed by the U.S. military [East, 2007]. Later, COBie was incorporated into the National Building Information Modeling Standard (NBIMS) and finally included into the British Standard and requirement for BIM level 2. This makes COBie a relatively new but successful standard. East [2007] summarized the goals of COBie as follows:

- Provide a simple format for real-time information exchange for existing design and construction contract deliverable
- Clearly identify requirements and responsibilities for business processes
- Provide a framework to store information for later exchange/retrieval
- Add no cost to operations and maintenance
- Permit direct import to owner's maintenance management system

Data defined through COBie can be delivered in four different formats: Standard for the Exchange of Product Model Data (STEP) physical file format, STEP Extensible Markup Language (XML) format, COBie lite transactional XML format and spreadsheets [BIM+, 2017]. No matter the format, the content of COBie stays the same and roughly translates into information about rooms, technical equipment, their connection to each other and how those are operated and maintained. Based on contractual agreements, the data is provided piecewise by designers, construction contractors and other possible actors from the operational and maintenance phase.

The overall structure of COBie is shown in figure 3.1. Rooms are organized first by building and then by floor and can be associated to zones; concepts that are commonly implemented during the design phase already. The implementation of components and their types is usually started during the design as well and further refined during the construction phase by respective contractors. The specific types are further augmented with information about maintenance routines, resources like manuals and spare parts later on. Common Items such as contacts, documents or general coordinates are attached to almost all of the other parts and can be attached in any phase.

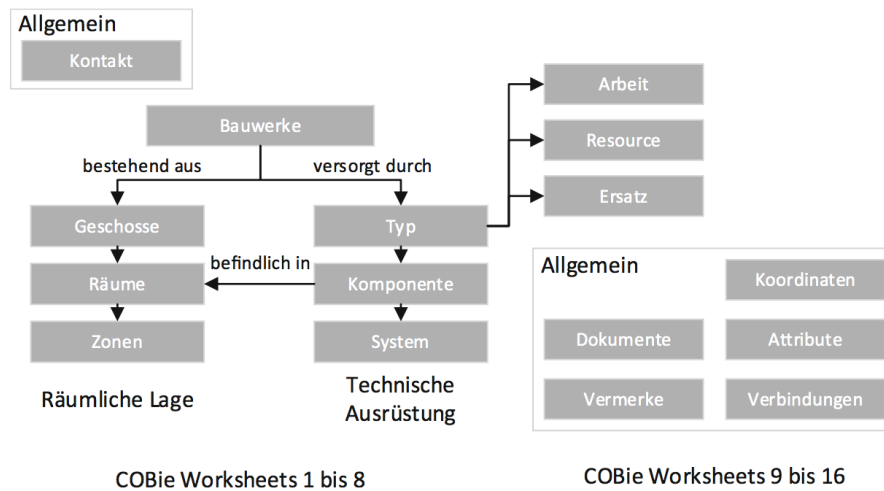


Figure 3.1: COBie organization [Borrmann et al., 2015]

Although COBie is supporting four different types of exports, the spreadsheet representation is by far the most popular and is often even seen as COBie itself instead of a simple way of packaging data [BIM+, 2017]. Users of course are generally very familiar with such spreadsheet formats and are able to open, review and indeed manipulate data through common tools like Microsoft Excel or LibreOffice Calc. This has been and still is the source of errors and misconceptions of the exchange standard.

3.2.2 Problems with the COBie Standard

One of the proclaimed goals of Bill East in 2007 was to “provide direct import/export support for COBie based on the parts of the building process on which those software systems are focused” [East, 2007]. He further elaborates: “Until a full suite of tools are developed and widely available, it is expected that users may need to directly input data into the spreadsheet”. The original goal of an automated process of extracting information from IFC files and importing it into FM systems has arguably lost focus. Many industry practitioner picked up the habit to manipulate COBie data directly in its spreadsheet form, which as a consequence introduces abundance of errors.

Similarly, experts working with COBie spreadsheets criticized their complexity, fragmentation of information and lack of easy interaction [Yalcinkaya et al., 2017]. The effective use of COBie requires software solutions that provide a reliable exporter and effective management and visualization of building data, the support of spreadsheets has arguable slowed down such developments. COBie has the potential of being an effective exchange format between the construction and the operation & maintenance phase. Furthermore, another challenge that hinders the effective use of COBie is the unclear roles of responsibility and the lack of

integration of representatives of the O&M phase, in regards of O&M relevant data input; Designers and other construction stakeholder regularly lack the expertise of identifying relevant O&M information.

Lastly there has been similar objections towards COBie as there have been towards the IFC standard, which has been criticized for its considerable complexity [Howard and Björk, 2008]. COBie fails to provide comprehensive semantic data for FM and does not guide the design team to source additional operational semantic data [Pärn et al., 2017].

3.3 BIM integrated Software Tools

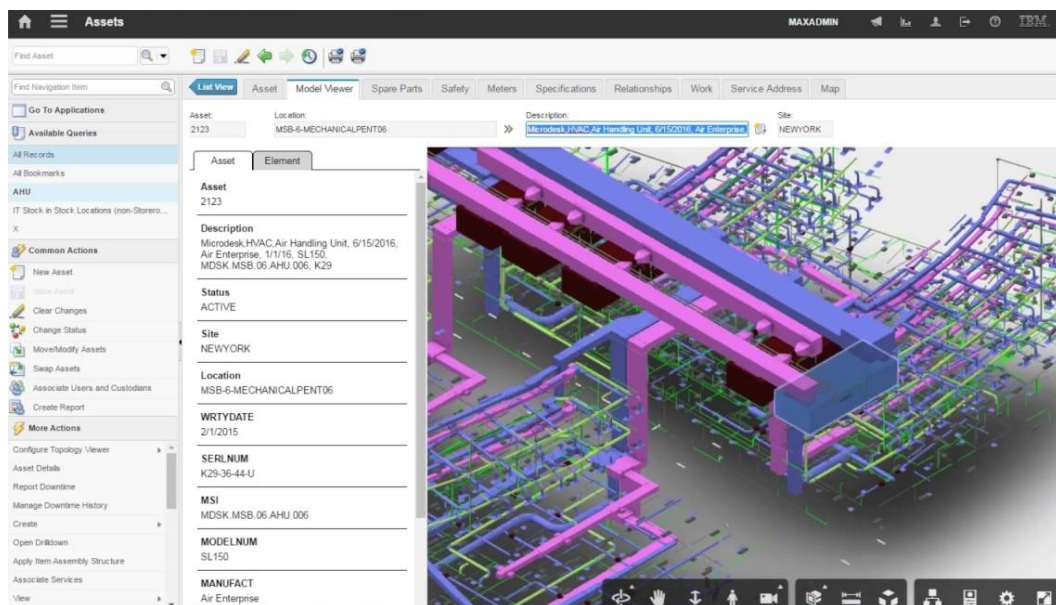


Figure 3.2: IBM Maximo showing functionality of its BIM extension.

A rising number of software vendors are taking the opportunity and are embracing BIM based solution in the sake of addressing issues of information management and its fragmentation inside the FM domain [Kiviniemi and Codinhoto, 2014]. Mentions of such systems (including CMMS and Computer-aided Facility Management (CAFM) tools) can be found e.g. in [Teicholz et al., 2013] and [Parsanezhad and Dimyadi, 2013]. [Teicholz et al., 2013] lists a total of 6 case studies including mentions of different software tools that have been used for FM organization. This list includes: FM:Interact, AIM CMMS, EcoDomus, AssetWorks and Maximo. [Parsanezhad and Dimyadi, 2013] complements this listing by WebTMA, Archibus and FAMIS. These commercial efforts are further augmented by various research project e.g. as discussed on [Pärn et al., 2017].

3.4 Challenges

Figure 3.2 shows the interface of IBM Maximo and its designated BIM extension, depicting FM functionality wrapped around a central 3D viewer. In many ways Maximo is exemplary for the approach most BIM FM solution provider have been taken so far: a strong emphasis on large and complex 3D models. As discussed before, we argue that such a concept is ill fitted and alienates practitioners from the O&M phase, who are unfamiliar with similar technology and do not see their core use cases supported. In truth many of the required task don't necessarily benefit from such a detailed and immediate visualization.

Looking at other, conventional and largely successful CMMSs, reveals that most of them are not integrating BIM and in fact, to a large extend, they do not even offer 3D or 2D integration at all. BIM-based solutions generally are hard to use for people from O&M phase which generally lack any knowledge of handling CAD like tools. Similarly BIM-enabled FM tools and traditionally CAD tools alike suffer from the same symptom of complex and hard-to-use interfaces, an issue that is amplified when trying to serve to a diverse set of people with little to no engineering background.

The diversity in activities, stakeholder and software tools makes it an absolute necessity to address issues of interoperability: How can external data be accessed and made usable and how can internal data made accessible for third parties. Another lesson learned from the recent success of small CMMS products like UpKeep, Fiix or HippoCMMS is the push of the FM industry to go mobile. Apps supporting 3D viewer are increasing, but compared to their 2D counterparts like Google Maps, navigation and interaction is still clunky and not as seamless. Finally, a pragmatic issue in the industry for years has been the access-rights to IFC or native models. Every BIM FM solution that puts such models at their center might face similar issues and ultimately restricts the reach of potential users.

Chapter 4

Concept

In the previous chapters we have seen that the **FM** domain is not only confronted with massive amount of information from the beginning—accumulated over months of design and construction—but also faces daily data streams from various scattered sources. Up until now, **FM** experts have answered this by using **CMMS**, which do a decent job in managing the different data items. Still, those systems are not able to respond to all issues, most namely: handling the initial data input (error prone and time intensive), providing spatial connection and establishing data interoperability. When applied correctly, **BIM** and the models created alongside, can provide such an initial data input, spatial context and aid interoperability, which it has been prominent for. Nonetheless, the few existing **BIM FM** solutions have arguably failed to establish a larger demand by the industry; testimony for the general problem of closing the gap between construction and the **O&M** phase.

The causes of these are manifold and dependent on ones perspective when studying the issue, yet a lack of understanding between participants of **O&M** phase and construction appears to be a root cause of the problem.[Kiviniemi and Codinhoto, 2014] This misunderstanding is based in the duality of project based thinking versus a life cycle based view, which is further consolidated by differences in professional backgrounds. On top of that, there is a lack of interest of **BIM** from most owners and operators due to the absence of hard evidence and clear business cases. We argue that the few revolutionary software vendors trying to close the gap between both phases, are repeating those mistakes by creating solutions better fitted for the construction domain than for **O&M** requirements.

The following chapter recaps said challenges and derives a set of requirements for a software system that successfully aims to address the issues of a **BIM-based FM** solution. Afterwards the envisioned concept is drawn out, which centers around the effective storage of data, its gradual retrieval and presentation via an easy to use interface.

4.1 Requirements

Front-runner in the **BIM FM** business are emphasizing the rich source of information inside an **IFC** file and the benefits that come with **3D** visualization, which in turn is usually a core component of their solutions. When checking facility management systems outside the **BIM FM** domain, one can observe that most popular solutions lack any **3D** information and only few are supporting **2D** in form of floor plans. Instead those systems are plain management tools that, above all, are supporting web-applications in combination with mobile use. Table 4.1 lists common tasks of **FM** teams and was collected by checking various **CMMS** systems and their supported work flows.

Writing and managing work orders
Communication with colleagues and tenants
Inspection and management of assets
Reservation management
Management of documents and contracts
Organize inspections
Manage suppliers
Purchase parts and request items
Create budget
Check location and stock store rooms
Locate assets

Table 4.1: Common **FM** tasks based on a survey of common **CMMS** solutions

This list of tasks shows nicely that a considerable amount of the common **FM** tasks do not directly benefit from spatial references, which is why practitioners are commonly alienated when confronted with **3D** heavy **BIM** solutions. **3D** solution can be very powerful still, e.g. when checking pipes and wiring behind walls, understand how to access certain parts or to comprehend connectivity and routs of larger networks; but in order to access such information one needs to load the complete content of multiple files, split by disciplines. More commonly practitioners could benefit from targeted request, accessing information from multiple files but connected to a very specific location, e.g. a certain room (see figure 4.1). Such information does not need the entirety of a commonly large **3D** model and could be answered simpler both in terms of the amount of data that is presented and its complexity.

Additionally, a shift away from a **3D**-first approach supports mobile-friendliness, since, despite all recent effort, the navigation of **3D** models on mobile devices remains an issue. As stated earlier, the trend of recent **CMMS** has shown a switch to mobile solution and should be considered for the designs of a **BIM-enabled FM** platform. Another common pitfall of **BIM/CAD** based systems in the construction field is the bad quality of user-interfaces, which are of high complexity and low intuitiveness. **FM** practitioners are generally not from an en-

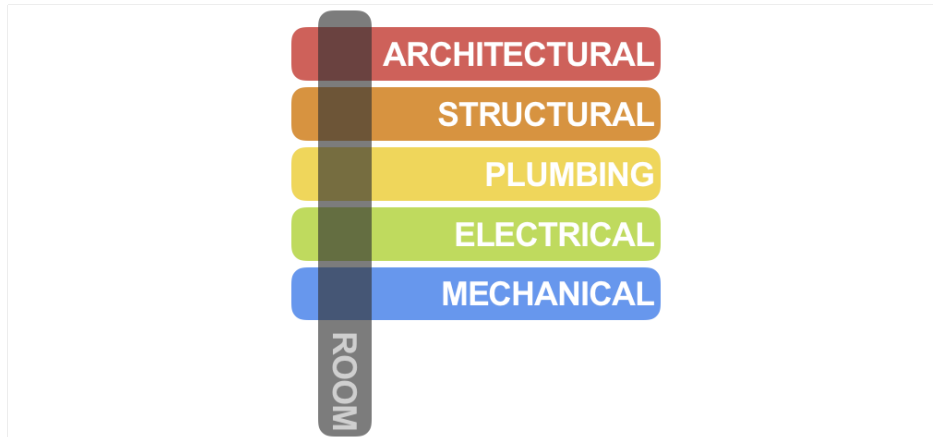


Figure 4.1: Room-based models as an intersection of various domain-specific IFC models.

engineering background and hence do not have prior knowledge in the use of CAD like systems, which makes the implementation of intuitive and easy user interfaces of at most importance.

Finally, we know of the fragmentation and diversity of the O&M phase, with a multitude of different actors and specialized tools. Data that is retrieved from IFC sources and accumulated throughout operations should be stored in a way that supports easy access and interoperability. The summarized requirements for a BIM-enabled FM system are:

1. Support accessibility and interoperability of data
2. Easy-to-use interface with a focus on FM tasks
3. Granular and controlled way of accessing information
4. Mobile support

4.2 Solution Approach

This thesis envisions a solution that addresses the previously mentioned requirements by extracting interactive 2D floor plans directly from the IFC model files and in a SVG format. Additionally, non geometric data residing inside the IFC is retrieved and stored on a graph database. Graph databases are optimized for storing data models of high connectivity, which is exactly the case when dealing with the IFC scheme [Robinson et al., 2013]. Through queries, data can be easily accessed and retrieved while respecting the IFC convention. IFC provides all object instances that follow the Universal Unique Identifier Standard (UUID) standard with a unique identifier or GUID [buildingSMART, 2018], which enables a distinct connection between graph data and interactive elements inside the SVG based floor plans. Core component of the proposed approach is a web-based dashboard that integrate said floor maps at its center, combined with User Interface (UI) component for displaying information retrieved from the connected graph database. Rooms are interactive elements that can be

opened and augmented with additional information. Tasks that requires more insight into the room layout beyond a 2D projection can be answered by the integration of 360° photo spheres, which are able to be expanded by information markers. Elements hidden behind facade or located inside walls can be understood through the use of partial 3D models that only include elements relevant for the current context. The steps to achieve such a system are described in the following.

4.2.1 Extracting floor plans from IFC Geometry

The extraction of floor plans is done using the Python programming language and the IfcOpenShell tool. IfcOpenShell is an open source software library that is specialized on working with IFC files.[IfcOpenShell, 2018] There are varying regulations about the cutting height in which floor plans are created from, typically around one meter. IfcOpenShell allows to define this cutting height dynamically and can be used to display elements that would otherwise be missed by the intersections, e.g. piping inside a ceiling. The target file format should be SVG. SVG was conceived by the World Wide Web Consortium (W3C) and is a language for describing 2D graphics in XML language. There are three base objects: vector graphics shapes, images and text [W3C, 2011]. Elements can be grouped and arranged in a tree-like structure and later accessed using the same structure, which enables to traverse individual sub elements inside a single file. Further, sub elements can be assigned with various attributes, allowing the storage of GUIDs originating from the IFC. For many years there has been issues with the support of browsers regarding the SVG format, however those have since been resolved [W3C, 2011]. All in all SVG is a superior format regarding our requirements.

4.2.2 Export of semantic data from IFC

IFC includes both geometry and semantic data, which are generally accessed through specialized software tools that will parse the respective file and display it including all the semantic information. This approach is not suitable when working in a web environment and doesn't support the access by third-parties, which are not specialized in parsing IFC. It is common practice to manage data through databases, however the IFC file format is of high complexity and makes the conception of effective relationships schemes difficult. Graph databases are a new kind of database, that do not need the definition of such schemes and are perfect to reflect and query highly connected content. As a matter of fact, the definition of relationships is a core principle of the IFC format [Borrmann et al., 2015]. Using IfcOpenShell once again in combination with Neo4j, which is one of the possible providers for graph databases, we are able to extract all relevant semantic data and make it accessible for us and third party software vendors.

4.2.3 Implementation of a web-based interface

In order to support future **FM** experts an interface is required, bringing together extracted geometry from the **IFC** file and its semantic data. Recent development in web technology has paved the way for powerful and full-fledged applications running in the browser [Taivalsaari et al., 2011]. Such applications with either standalone mobile applications or integrated mobile support in itself are the way forward, which has been confirmed by trends in the **CMMS** domain. Hence, the envisioned systems is designed as a web platform with mobile support. By putting **2D** floor plan at the center, several benefits regarding performance, mobile interaction and accessibility are achieved as mentioned in earlier sections.

4.2.4 Integration of panoramic images and model viewer

Naturally, the foregoing of the third dimension entails not only benefits but certain drawbacks alike. **3D** views of a room can help to far better understand the situation inside a particular room. Service engineers for example, could see how exactly certain equipment is located and prepare to reach it effectively or use the power of **3D** to trace the position of pipes behind a wall. Examining those two possible use cases, one can derive two different solutions, that answer the respective requirements in an optimal way. Firstly, getting insight into a remote location is best answered by photo realistic images. This effective approach of explorations is known from the Street View feature inside the Google Maps suite and has been applied by some innovative start ups like NavVis for indoor settings. Secondly information of elements hidden behind walls or suspended ceilings requires information provided by building models and to this sake the system shall incorporate a **3D** viewer. Technicians called to a specific room are oftentimes only interested in the situation around that particular space and possibly of the source and further course of piping adjoining it. To our knowledge there is no system at the moment allowing such a pointed request. By using IfcOpenShell yet again, an extraction of individual rooms is possible and was applied in this approach with future research and investigation still needed.

4.3 Summary

3D heavy solution mainly suffer from an overload of information both in terms of performance for the client machine and for the mental capacity of the client himself. **IFC** files are the standard of storing data collaboratively during a construction project, are a potential rich source of information, but are not suited for providing long-term, third-party access. Also **3D** solution are hard to realize satisfactorily in term of navigation on mobile devices, which appears to be the clear trend inside the **FM** community. All of this speaks against most of

the current approaches to bring **BIM** solutions into **FM**. We propose the use of interactive **2D** floor plans in a web environment, which keep the connection to semantic information down to the individual components. Those semantics do not remain inside **IFC** files but are instead unlocked and stored inside a hosted graph database, allowing easy access and execution of queries through third-parties. Core tasks by **FM** personnel is effectively supported by providing both 360° photo spheres and small partial **3D** models in addition to the basic **2D** projections. These partial models are generated beforehand and include information from various disciplines connected by a room-based context and thus preventing the loading of large, entire models.

Chapter 5

Technical Foundations

The envisioned prototype for a **BIM**-enabled **FM** software solution encompasses various technical and theoretical foundations. Those stretch from technologies inside the digital construction industry, database systems and finally various web-based solutions. In this chapter all of the relevant technologies are described in more detail and are put in relations to the construction and **O&M** interface and the envisioned software solution.

5.1 Industry Foundation Classes

Industry Foundation Classes is an open and standardized data model for representing building information and exchanging it between various Architecture, Engineering and Construction (**AEC**) software products and has since become deeply connected with the efforts of interoperability and collaboration through **BIM** applications worldwide. Its conception goes back to the 90s where it was first derived from the product manufacturing information standard called **STEP**. **IFC** is a powerful data model and is able to represent geometric and semantic data for various building parts and processes, connected to different disciplines and over the whole life cycle of buildings. [Borrmann et al., 2015]. This holistic approach comes with certain drawbacks, such as an increased complexity of the structure, ambiguous content and difficulty in being worked with. Nevertheless, it is still an important corner piece of the **BIM** concept and is increasingly adapted by software vendors and practitioners alike. In this chapter we will take a quick look on a brief history of the **IFC** standard, which gives valuable insight about the current situation. Afterwards the architecture of the data model is presented more closely, before detailing the various objects and their relations. Finally the interplay between **IFC** and the conducted work at hand is drawn.

5.1.1 A Brief History

The first version of **IFC** was released in 1997 by the International Alliance for Interoperability, a open consortium who came together with the vision ““To define, promote and publish a specification for sharing data throughout the project life cycle, globally, across disciplines and across technical applications”” [Wix and See, 1999]. Previously, similar ambitions were followed by representative of the **AEC** industry as part of the ISO **STEP** standard. **STEP** aspired a common and rich data model for product information from various kinds of industries, but the common effort was deemed too slow and unresponsive for the demands of the construction industry, which finally led to the forming of the International Alliance for Interoperability (**IAI**) and the creation of **IFC**. However, much of the technical foundation of the **IFC** model was directly inherit from **STEP**, such as definitions for geometric representation and the Express language. The first version of **IFC** was pretty limited in scope and mainly focused around the architectural domain of the building but was quickly followed by **IFC 2.0** in 1999. Much effort was put into the new version which included many new features like, schema for building services, cost estimation and construction planning. Still, adaption by the industry remained unexpectedly low with a general slow pace of progress and sparse resources. The holistic approach that was taken previously stemmed a data model of fairly large complexity, which left software vendors with difficulties of implementing sufficient support, while lacking the necessary business motivation to begin with [Laakso et al., 2012]. The year

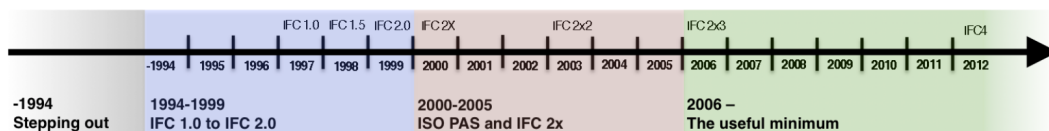


Figure 5.1: Timeline of the **IFC** format, showing the three major phases [Laakso et al., 2012].

2006 marked a new shift in the **IFC** process, which was accompanied with the renaming of the previously called **IAI** to buildingSMART and the release of **IFC 2.3**. In an effort to make **IFC** more accessible and usable it was decided to narrow data exchanges down into manageable, predictable and implementable specifications [Laakso et al., 2012], which finally gave rise to the concept of Information Delivery Manuals and later Model View Definition (**MVD**). Information Delivery Manuals (**IDM**) and **MVD** are supplements to the **IFC**, which help to narrow down the completeness of the model by defining clear data exchange requirements. All in all, the history of **IFC** was marked by a large heritage, a rocky road and major shifts in direction. An increasing support by the public sector in several countries around the world and the new release of **IFC4** will keep things interesting for the **IFC** standard.

5.1.2 Architecture

As mentioned in the previous section, **IFC** was conceived as an extensible model that incorporates geometry with various kinds of semantic information. In order to describe such complexity, an own modeling language was applied called EXPRESS. Relationships attributes, constraints and inheritance are core concept of EXPRESS and are consequently reflected in the **IFC** [Laakso et al., 2012]. As of version 2x4, **IFC** included 800 different entities, which verifies its complex nature. In order to improve maintenance and perception of the model, it was organized into four different layers as depicted in 5.2.

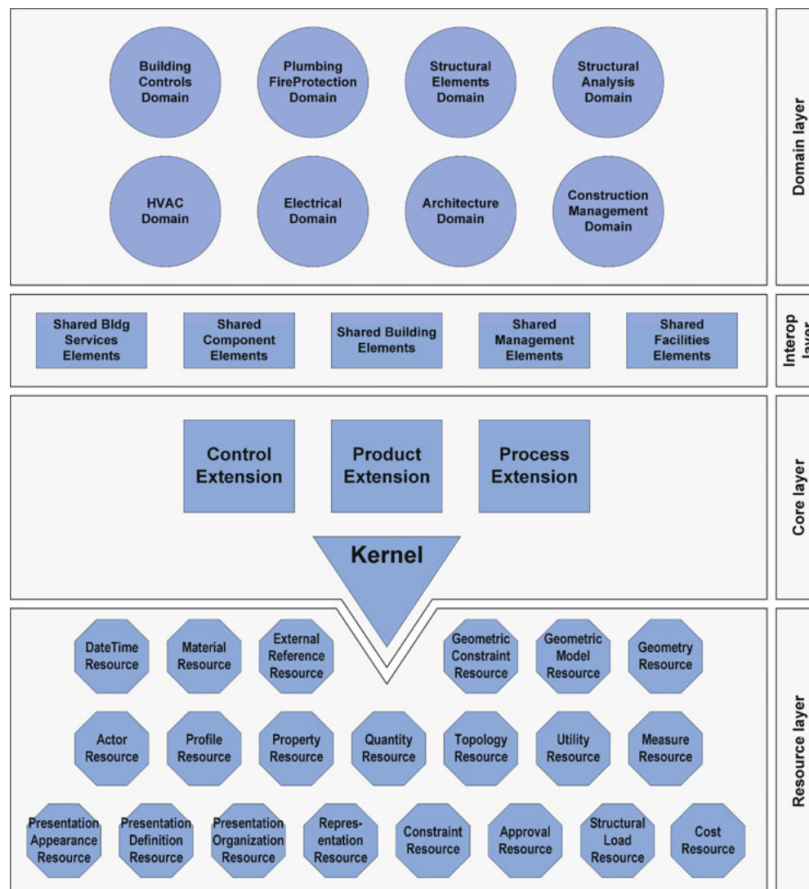


Figure 5.2: Layers of the **IFC** data model [Borrmann et al., 2015].

The *core layer* forms the overall foundation and incorporates important base classes such as *IfcProduct*, *IfcObject*, *IfcActor*, *IfcProcess*, *IfcRoot*, *ifcProject*, *IfcRelationship*. The *domain layer* lies above the core layer and includes specialized classes that are assigned to specific domains. Both core and domain layer are connected via the *shared layer*, which consists of defined derivations of the core's base objects such as *IfcWall*, *IfcColumn*, *etc* and are used in the schemes of the domain layer. Lastly, the *Resource Layer* bundles fundamental schemes and data structures that can be used by various other objects of the remaining layers. Due to **IFC** strict hierarchically object sub-typing, this connection only exists down stream and hence

resource entities do not share access to anywhere else. Resource schemes include geometry, topology and material among other things.

5.1.3 Objects and their Relations

IFC holds various different object types, which are organized by a strict hierarchically structure. A small extract of the most important entities inside said structure is shown in the figure below.

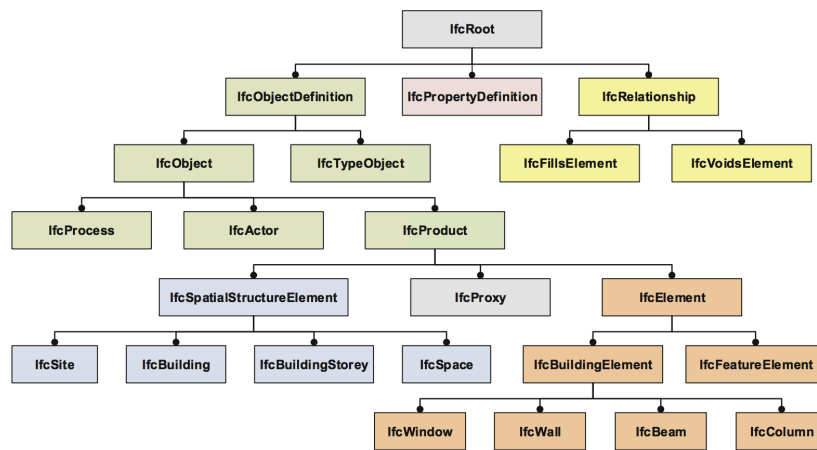


Figure 5.3: Extract of most important entities from the IFC hierarchy [Borrmann et al., 2015].

Each object is defined by aspects of inheritance and as such fits different categories. Notable classes include *IfcProduct*, which is the abstract super class for all classes that include physical or spatial representation. Spatial structures are defined as *IfcSite*, *IfcBuilding*, *IfcBuildingStorey* and *IfcSpace*, which are crucial for organizing building elements inside the IFC. The *IfcElement* resembles the root for many building elements that are organized by their spatial hierarchy and finally *IfcRelationship* ties the various elements together. Relationships are in fact an important concept of the data model and is one major contributor for its richness and complexity alike. A special characteristic of relationships in the IFC is the fact that connections are not presented through direct associations but through standalone relationship objects. The relationship object of course inherits from the *IfcRelationship* class and establishes the forward connection always from the *relating* element to the *related* objects, which is depicted for the example of a window inside a wall in 5.4.

A final, simple but important concept is the introduction of Globally Unique Identifier. Each element inheriting from the *IfcRoot* automatically implements such and is thereby uniquely identifiable. Especially when dealing with the IFC through third-party applications or while coordinating various domain specific models of the same building the concept of a reliable identifier is an indispensable aid.

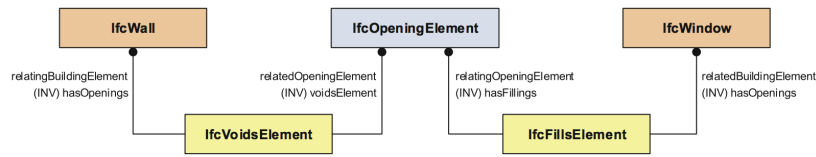


Figure 5.4: The principle of relations depicted as for the example of window-opening-wall [Borrmann et al., 2015].

5.1.4 Application Scope of IFC

In the scope of this thesis, IFC is utilized as the single source of geometric and semantic information. Especially important is the extraction of 2D projections of the various building elements for the floor plan integration later on, which is made possible by utilizing IfcOpenShell and PythonOCC frameworks. An alternative or rather supplementary format is the COBie exchange format, which as discussed previously, is of rising popularity and designed for the FM handover specifically. However, ultimately COBie is just a format for the exchange of data that is using IFC and other systems as sources, retrieving information based on its specifications. Generally those specifications are very broad, which has been criticized before [Pärn et al., 2017], and—more importantly in the application at hand—it does not include geometry. Hence, it was decided to limit the scope to the IFC as single source of information, being a unified package containing most of the relevant information needed.

5.2 Graph Database

In this section we will introduce the technology of graph databases. Graph databases belong to the family of NOSQL database systems and stands out in regard to its NOSQL siblings with a relationship first approach. In recent years there has been a rising popularity around graph databases and in the meanwhile several different vendors with varying types of systems have emerged. In the scope of this thesis, Neo4j was applied, which is, as up-to-date, the most popular solutions on the market [DB-Engines, 2018a]. Neo4j applies labeled property graphs, on which we will be limiting ourselves for the following sections. Other noteworthy data model types include *hypergraphs* and *triples*—latter being of special interest in regards of the semantic web movement in and outside the BIM context. For more information on graph databases we want to refer to Robinson et al. [2015], which is—while somewhat overemphasizing Neo4j—an informative read on graph databases in general and good starting point leading to other publication on this issue.

5.2.1 Definition

A graph database is a database that uses a graph models to query and store information in the form of nodes and edges. Graphs and the study of them, has been around since Leonard Euler and his publication on the seven bridges of Königsberg in 1736. Only recently though Database-Management System have started picking up on the concepts taught through graph theory. Naturally the devil is in the detail but the basic concept of graphs are easily conveyed. Graphs consists of *vertices* and *edges*—or ,in language of graph Database-Management System (DBMS), of *nodes* and *relationships*. Generally nodes represent entities in the domain of interest, which are interconnected through various relationships. A special type of graph, is the so called *labeled property graph* and has the following characteristics [Robinson et al., 2015]:

- It contains nodes and relationships
- Nodes contain properties
- Relationships are named, directed and always connected to a start and end node
- Relationships can also contain properties

A small example of such a graph can be seen in figure 5.5

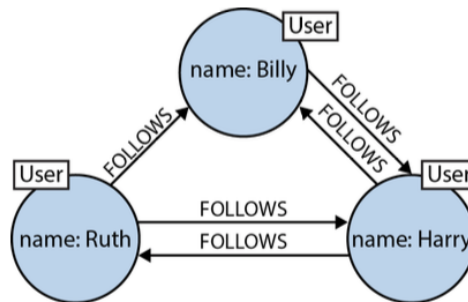


Figure 5.5: A small social graph [Robinson et al., 2015]

Graph databases also differentiate themselves on how they approach the underlying storage of the data; this can be almost identical to the external graph structure (as it is the case with Neo4j) or using other storage methods at its core and only wrapping graph functionality. Similar to most other DBMS, graph databases are designed for external access in form of transaction and support of Create, Read, Update and Delete methods. Interaction with databases requires query languages. The most common of these is SQL, which was designed specifically for relational databases and hence is not designed for traversing large graphs. Instead several multi-vendor query languages were conceived like Gremlin, SPARQL and Cypher.

5.2.2 Characteristics

A common first step during the creation of a data base is the conception of a data scheme, which—especially in the case of relational databases—can be rather difficult and once applied hard to change [Robinson et al., 2015]. Graphs on the other hand shine in their close resemblance of the real world—almost anything can be modeled as a graph. On top of that, they stay flexible for change and are easily additive for new data and concepts; they do not rely on the clear definition of table joins. Especially in today’s fast passed, incremental and iterative software delivery methodology, this flexibility is a valuable advantage. Data that is highly connected or data that is very heterogeneous by nature, is not easily queried by relational DBMS and require several table joins. Of course graphs deal very well with connectivity and due to their flexibility, which they share with their NOSQL siblings, they are able to deal with very heterogeneous data as well. Another advantages is their possibility to detach sub-graphs and apply queries locally, this allows them to scale immensely well in terms of sheer volume of data Their ability to easily traverse links between data items, sets graph from other databases apart—NOSQL and Structured Query Language (SQL) alike. This is further affirmed by [Vukotic et al., 2014] who performed a test between a relational database and a native graph database; the test setup: a database of one million users. The tasks for both systems consisted in retrieving friends-of-friends in an increasing level of depth(see figure 5.6) While both systems did fairly similar for the first degree friends-of-friends relations, it diverged quickly from there.

Depth	Execution Time - MySQL	Execution Time - Neo4j
2	0.016	0.010
3	30.267	0.168
4	1,543.505	1.359
5	Not Finished in 1 Hour	2.132

Figure 5.6: Benchmark results of a "friends-of-friends" query in the limits of a small social graph [Vukotic et al., 2014].

Of course, graphs are not the ideal solution for any application and they come with certain drawbacks. A common issue that runs through most of the family of NOSQL systems, is safeguarding transaction in terms of of Atomic, Consistent, Isolated and Durable, which is especially important for systems of the finance sector. Further, they generally fall short compared to relational databases in terms of queries that require batch access to the whole database system at once. In a nutshell, graph databases are not set in replacing old and

presumably decrepit relational [DBMS](#), instead both systems are valuable and fulfill their respective purposes. Graph databases as the "new player" and a technology that has yet to mature, will over time fill those specific spots in which it can excel thanks to its special benefits given by its design.

5.2.3 Application

Graph databases are a relatively new technology, that slowly find its spots in industry landscape. Especially in the era of big data, global players like Google and Facebook are faced with data-sets of enormous size and great entanglement; a perfect fit for the [aphttps://v2.overleaf.com/project/5b3f4ba522a329040e448bb5](https://v2.overleaf.com/project/5b3f4ba522a329040e448bb5)plication of graphs. The construction industry might very well be another one of such cases. In their evaluation from 2015, [Mordinyi et al. \[2015\]](#) are highlighting the benefits of NOSQL graph databases in the face of heterogeneous and complex engineering information—data that is not easily fit into a single scheme.

Another layer of application is specific to the construction industry, or to be even more precise, specific to the [IFC](#) standard. As described in [5.1](#), [IFC](#) is an standard of large complexity and completeness that encompasses many relationships between various entities. Such a structure can naturally be represented as a graph and—with the use of query languages such as Cypher—easily queried for all kinds of information, accessible for any stakeholder involved. Research in that area is limited with [Ismail et al. \[2017\]](#) working on first prototypes and ontology to translate [IFC](#) into a graph structure. A similar direction, yet way more ambitious in its nature, is the vision pursued by the [Linked Building Data Community \[LDB, 2018\]](#). Their proclaimed goal is to connect distributed building data that resides on different servers and in different limits of responsibility through a single, open network—a web of data. Those concepts touch more into future developments, however representing the [IFC](#) structure in a graph and making it accessible as an open database has tremendous benefits for parties of the [O&M](#) phase in specific, who more often than not lack the access to software solutions that can work with the [IFC](#) file format directly. Thus, this thesis aims to translate [IFC](#) files from different disciplines into a single graph data base. However, instead of representing the [IFC](#) one-to-one, we are only translating the part included in the [FM](#) handover [MVD](#) provided by [buidlingSMART](#). This is of course due to limitations of scope on the one hand but equally in interest of usability—not all the information in the [IFC](#) is important to [O&M](#) stakeholders. Special use cases such as path finding or comprehending and tracing the flow of Mechanical, Electrical and Plumbing ([MEP](#)) systems are applied as well, but will be discussed in more detail later on.

5.3 Scalable Vector Graphics

[IFC](#) in section 5.1 represents the main source of information, which is stored and made accessible through the graph database described in section 5.2. In this section we introduce [SVG](#) as a third set of technology that allows us to visualize geometric data in form of floor plans. As mentioned previously, the reasoning of emphasizing the [2D](#) projection of floors is manifold and includes, partly but not exclusively, the idea of gradually increasing complexity and richness of information, generally less load on the clients machine and better support and navigation capacity on mobile devices. Similar to the previous section, [SVG](#) will be briefly presented before putting it in context to our prototype.

5.3.1 Definition and Functionality

Scalable Vector Graphics is a vector image format for describing [2D](#) graphics with the support of interactivity, animations and a seamless operability with other web standards such as Cascading Style Sheets ([CSS](#)), Document Object Model ([DOM](#)) and Hypertext Markup Language ([HTML](#)) [MDN, 2018]. The creation has its roots in the late 90s when Chris Lilley defined a set of requirements for an open vector graphics standard [WC3, 2010]—who was at that time and is to this date, involved with the [W3C](#) consortium. These requirements became the criteria for a competition, in which different software companies—names such as Autodesk, IBM and Microsoft included—developed and presented their concept and/or implementation of a vector based graphics format. Finally, the [W3C](#) decided not to implement any of the final six submissions, but instead to create a new language, that would use various aspects and lessons learned by the contending proposals and thus [SVG](#) was born. [SVG](#) differentiate itself from other image formats such as PNG or JPEG in the sense that it defines the geometry and its behavior through its [XML](#) based file content. This vector-based approach allows [SVG](#) images to stay crisp no matter how large they are stretched. Furthermore, being defined in [XML](#) code allows outsiders to search, manipulate and index the various parts of the file; graphics can basically be written and edited with any text editor. While the [SVG](#) structure can get fairly complex, at its core is is rather simple and consists only of three basic types: shapes (based on a combination of straight and curved paths), text and raster images [Dengler et al., 2011]. A simple example of the [SVG](#) format can be seen in figure 5.7. There the `g`-tag can be used to group several graphical objects and apply styling, transformations or other, on all collected elements alike.

Although [SVG](#) has been around since the start of the new millennium, its adaption by modern browsers has been an issue for a long time. Especially Internet Explorer hindered a greater adoption on web pages by persistently not implementing support of the [SVG](#) standard. Almost two decades later, [SVG](#) finally has support by all major modern web browser

```
1 <svg viewBox="0 0 100 100" xmlns="http://www.w3.org/2000/svg">
2   <!-- Using g to inherit presentation attributes -->
3   <g fill="white" stroke="green" stroke-width="5">
4     <circle cx="40" cy="40" r="25" />
5     <circle cx="60" cy="60" r="25" />
6   </g>
7 </svg>
```

Figure 5.7: Small SVG example [MDN, 2018]

and developers are increasingly implementing SVG icons and other UI elements, driven primarily by the great opportunities in regards to animations. In recent years the format was constantly improved with much of the development effort put into decreasing the overall file size, increasing performance and expand on mobile support.

5.3.2 Application

The prototype introduced in the scope of this thesis uses floor plans as a point of reference and to support the presentation of data inside a facility. The interactivity that is provided by the SVG format is an obvious advantage; it allows the direct interaction between the user and the displayed floor plan. SVG's XML-based structure enables us to generate the format directly in code with relatively small effort using—depending on programming language and environment—available frameworks; in the case of this thesis: pythonOCC. There exists an interesting twists in the prototype presented here; instead of generating the SVG right away and loading them as files later on, the generations happens directly inside the web application. This approach enables a flexible and dynamic way of loading the 2D representation of different kinds of building elements and is enabled by the React framework, which is touched upon in more detail in section xx. To say this much already: the DOM like structure of the SVG is particular helpful in connection with frameworks like React. Examining current publications shows increasing interest in SVG for displaying maps and different purposes—one prominent application area being indoor navigation and positioning. One such case is presented by Ohrt and Turau [2013], who, in addition to issues towards indoor navigation, have evaluated the performance on desktop and mobile devices for various sizes of maps. They attest the good performance of SVG, which corresponds with the experience made in the course of the prototypical development.

5.4 Single Page Application

Single-Page Applications (SPA) is a web application or web site that reacts to interactions with the user by re-rendering either the whole or parts of its interface dynamically rather than

loading entire new pages from a server [Single-page application, 2018]; a key factor, which differentiates it from traditional multi-page websites. In the following we introduce first the traditional multi-page approach before explaining the specifics of SPA and concluding with the association to our solution stack

5.4.1 Traditional Multi-Page Approach

A website is a collection of web pages typically identified with a common domain name and published at least on one web server [Website, 2018]. Websites can be accessed via public Internet Protocol networks, such as the internet, through the use of browsers, which renders the content of the page, and by giving a valid uniform resource locator (URL). A website can consist of many web pages and they are typically composed of plain text which is structured through the use of HTML, styled through CSS and made interactive by using JavaScript: the three basic technologies of the World Wide Web.

HTML

HTML is the standard markup language for creating web pages. Typically, web browser receive HTML from a server when accessing a web page and render it accordingly through the given markup. For that, HTML provides the semantic structure of the web page and includes indications on how the page is to be styled by including classes, identifier and basic HTML elements themselves[HTML, 2018]; the styling is handled by CSS afterwards.

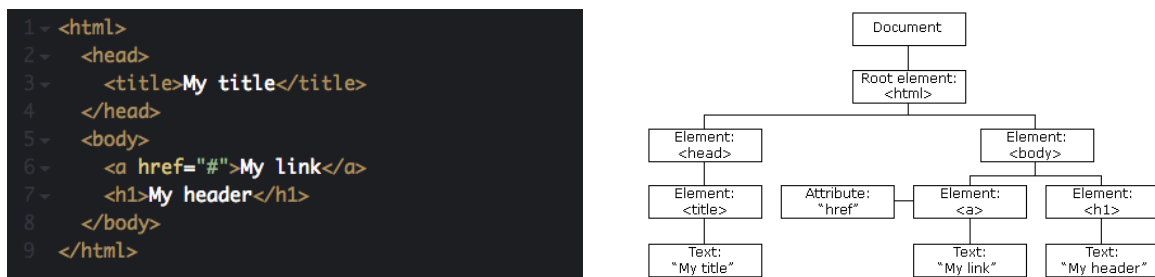


Figure 5.8: Simple HTML markup next to the corresponding DOM tree.

Figure 5.8 shows the fundamental structure of a web page and a number of exemplary HTML elements, such as <a> and <h1>. On the right side we see, what developers refer to as Document Object Model, which reflects the natural tree structure giving by the HTML markup and can be used to programmatically change the structure of a web page and trigger re-renderings—a feature that is of importance later on.

CSS

[CSS](#) is a language for defining styles and layouts and is commonly used in connection with [HTML](#) and in a web environment [[Cascading Style Sheets, 2018](#)]. The conception of [CSS](#) goes back to the idea of separating presentation and content and hence is consequently stored in separate `.css` files. This approach creates flexibility by either providing similar styling to multiple different pages or to give the same markup different style option, which can be helpful when providing different layouts for small and large screen devices. [CSS](#) uses a simple syntax, which is applied to form multiple rules. A rule consists of a set of *properties*, which describe how the [HTML](#) is supposed to be presented and a *selector* to identify which elements of the [DOM](#) are supposed to be affected by the rule (for a small example see [figure 5.9](#)).

```
1 | h1 {  
2 |   color: blue;  
3 |   background-color: yellow;  
4 |   border: 1px solid black;  
5 | }  
6 |  
7 | p {  
8 |   color: red;  
9 | }
```

Figure 5.9: Simple example of two different [CSS](#) rules.

There is of course much more to [CSS](#) and some advanced concepts includes inheritance, feature queries and specificity. For more information we would like to refer to the excellent coverage by the [W3C](#) and the [MDN web docs](#), provided by Mozilla.

JavaScript

Alongside [HTML](#) and [CSS](#), JavaScript is the last building block of the three fundamental technologies that form the World Wide Web. It is a object-oriented scripting language, which is mainly used to make web pages interactive by responding to events triggered by interactions of the user. Originally JavaScript was exclusively used for client-side code, but has since been successfully applied in various server technologies and non-web programs. In case of client-side code, JavaScript is able to change the structure and layout of a web page, which is made possible through the exposure of the [DOM](#). In the last years, JavaScript has gone under major changes, which gave rise to new technologies but also issues of browser support. Also, JavaScript is supported by a large community of developers from which many different frameworks and libraries evolved—some of which have been applied in the course of this thesis as well.

5.4.2 Single Page Approach

The traditional multi-page approach is accompanied with a number of issues, which is the reason that since then most modern web site have been turned into [SPA](#). Some of the encountered issues include: server loads when switching between web pages and often loading the same page over and over again, not able to continue working when the internet connection is lost, no real-time updates of data and finally, large and complex server code with much logic. [SPA](#) overcomes this by moving most of the logic into the the client and by using JavaScript and new frameworks built on top of it, to dynamically request data from the server and to change the layout of the page by manipulating the [DOM](#). In short, the split into three destined domains of [HTML](#), [CSS](#) and JavaScript is weakened and instead JavaScript moves into the center and is able to influence and manipulate both styling and the structure of the web page.

Not having to load after each page switch, gives [SPA](#) a distinct "application like" feeling and also provides the possibility to work offline over a longer period of time because most of the content is loaded with the initial load. By moving most of the logic to the client-side, developers were able to design slimmer, more focused back-end server—commonly referred to as Application Programming Interfaces—which made them easier to reuse for different applications. Previously, client-side code had only relatively simple tasks to control and developers didn't need much support through frameworks and mostly relied on native JavaScript. However, sustaining large, interactive [SPA](#) with ever-changing layouts reached soon the capacity of "just" JavaScript and as a response, one after the other, new frameworks on top of JavaScript were developed (see figure 5.10).



Figure 5.10: Popular front-end frameworks for developing [SPA](#).

Each of those try to solve similar issues while focusing on different aspects and resorting to different methods.

5.4.3 Application

As mentioned before, [FM](#) is a profound interdisciplinary domain where people from different backgrounds come together. In such settings, web solutions shine because they are accessible by everyone no matter the operating systems or device type and hence made them a natural choice for the front-end of our prototype. Alongside, a Application Programming Interface

(API) and a corresponding database (based on graph technology) were created, serving the dynamic data items and complementing the overall service. From the many possible client-side frameworks to choose from we opted for React, which provided unique opportunities in term of rendering SVG-based floor plans and will be covered in greater detail later on (see section 8.3.2).

Chapter 6

Floor Plans

In the domain of architecture and engineering, floor plans refer to drawings to scale, showing a view from above and the relationships between rooms, spaces, traffic patterns and other physical feature at one level of a structure [Floor plan, 2018]. Their use goes back many hundreds of years, in which they aided—architects and engineers alike—during conception, design and finally construction of in part monumental structures; the oldest, remaining architectural drawings dating back as far as to the 9th century [Plan of Saint Gall, 2017]. In days of 3D rendering and the availability of computers, the relevance of floor plans has decreased, although paper-based floor plans remain popular especially on site. Outside the immediate construction sector, floor plans are part of day-to-day exposure in form of fire escape plans or plain overviews for orientations in shopping malls or museums.

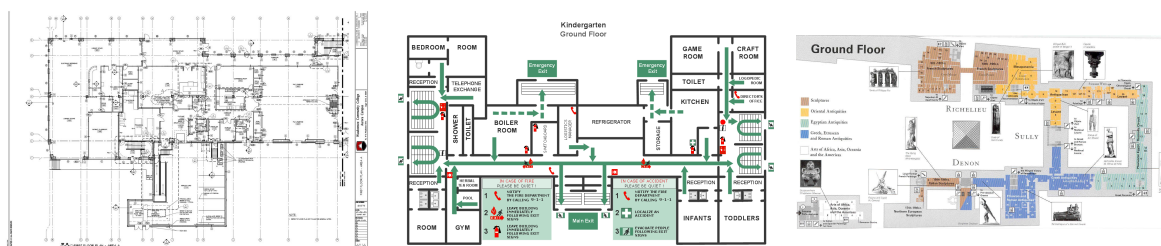


Figure 6.1: Floor plans in three different shapes (from left to right): architectural drawing, fire escape plan, Louvre museum map

Floor plans of course can be seen as a specialized form of regular maps and as such are used on mobile and other devices every day. We argue that floor plans—even in the digital age of 3D renderings—remain important and provide opportunities for improved data management, system integration and user experience. In this chapter we will expand on the mentioned points, elaborate on the reasons why we choose floor plans over a 3D first approach in the presented prototype and will discuss the problems caused by limiting in two dimensions. Next

the presented prototypical approach is highlighted in terms of generation of floor plans from IFC and the respective export into the SVG format before ending with a final summary.

6.1 Purpose

On their website, the UK-based National Building Specification explains the differences between the various BIM dimensions: 3D includes geometric and non-geometric information shared through a Common Data Environment, 4D represents construction scheduling, 5D for cost and 6D showing project life-cycle information [McPartland, 2017]. Comparing oneself to those n-dimensional utopias, the proclamation of using 2D floor plans seems almost ridiculous and astonishing outdated.

For years, designers, engineers and contractors have worked together in BIM projects, creating models of increasing complexity, tuning them against each other through the exchange of models. The fruits of those efforts have been regularly handed over to owner and other O&M representatives—copyright and contractual issues excluded—in form of e.g. IFC files. However, the same fruits of labour have been left hanging by the receiving side ever since, without utilizing the rich data inside. Of course we know in the meantime that the issues lies not with a single side but is rather caused, among other things, by mutual non-understanding and the lack of integration of the other side into the implementation process.

A new generation of FM-enabled BIM tools, such as EcoDomus, have entered the stage, setting out to redeem past mistakes by integrating more FM-relevant data. A common trait of those solutions still, is a 3D first integration, presenting IFC models of the architectural and MEP domain inside an integrated view-port. Those models typically represent the final most detail product after many iterations of improvements and refinements, representing the building as whole. This bird's-eye type of view is very common and necessary during a construction project; practitioners from different disciplines design and adjust their respective models towards a single holistic product, which is the building. However, FM practitioners have a very different requirement. They are not interested, at least first thing, in a holistic view of a facility and they don't care about the split in disciplines either. The information FM practitioners do care about is mostly connected to a specific location inside the facility, say a room, and all relevant information connected to it.

The obvious mismatch here are holistic models of complete buildings and split by discipline on the one hand and the FM practitioners need in a specific location or room and connected data coming from different construction disciplines. Meeting his or hers needs requires the loading of all those models at the same time. Depending on the particular buildings, those model files can be huge and include many thousands of different elements; a true flood in information. Modern BIM tools such as Solibri, are providing the user with means to deal and

ultimately make sense of the presented mess. Learning how to use such tools though, requires time and training; especially so for FM practitioners that oftentimes lack a background in engineering.

On top of that, rendering 3D models casts high demands on any client machine; loading large models from different disciplines can easily take minutes and hence severely diminish any user experience. This factor even gains importance considering the recent trend—both in construction and FM—towards smaller, specialized mobile tools [Sattineni and Schmidt, 2015]. The move towards mobile also rises the question of user interaction. Navigating 3D spatial information on mobile is not a trivial task and is subject to much research effort. Future application centering around 3D will face a difficult task of providing tools of satisfactory user experience. A final issue, less discussed but often experienced, is the issue of sharing 3D models in form of e.g. IFC; the main concern being privacy and security related. Using 3D as a core feature limits the reach of such application and potentially blocks out the integration of user groups like residents or other occupants of buildings.

Right off the bat, switching to a 2D first approach addresses most of the issues just mentioned. *Access right* of floor plans are generally less of a concern, after all they are accessible in form of fire escape plans in any larger building. 2D floor plans also have excellent *support on mobile*, both in terms of user interaction and performance, as seen in applications such as Google Maps. The same decrease in *performance* is of course true in desktop application and by projecting information onto a single plane per floor, information is presented in a piece meal fashion rather than an *information flood*. Of course there are downside to the concept of decreasing dimensions, which can be compensated for when applying the correct approaches, which will be explained in the following.

6.2 Problems of 2D and how to solve them

There are two major problems that can be identified when swapping the classical 3D first with a 2D centered approach: Firstly, the additional layer of abstraction reduces the capabilities of orientation and secondly, the loss of valuable information such as height and overall positioning of especially mechanical equipment. Both concerns are valid, but can be reduced to a minimum by applying appropriate techniques.

6.2.1 Orientation and Navigation

In theory, 3D-based approaches benefit from rapid identification of objects and an improved ego-centric alignment of the user, aspects that are missing in respective 2D projections and although applying 3D alone doesn't guarantee satisfactory results in space navigation

[Oulasvirta et al., 2009], the same problems remain a challenge for the 2D counterparts. Nossun [2013] highlights the unique challenge imposed by indoor settings compared to outdoor ones: Information density is often higher, landmarks can change frequently, north-south orientations are hard to maintain and multi-storey building add complexity by adding an additional dimension.

Classic cartography is many century old and offers a large repertoire of guidelines and best practises, however most of them apply for outdoor settings [Nossun, 2013]; yet, fundamental principles retain their importance no matter the setting. The following set of design principles should be aspired: clarity, order, balance, contrast, unity and harmony.

Tyner [2010] connects those principles to the domain of cartography as follows:

Clarity: Closely examine the objective of your map and remove anything that does not enhance the given message.

Order: Evaluate the logic of your map, avoid cluttering of elements and make certain that placement is done in a logical order.

Balance: Every component in a map carries a visual weight, place those component based on their weights and the overall optical center, which is slightly vertically off center.

Contrast: Contrast is an significant concept to convey meaning and give importance to certain elements of a map.

Unity: Refers to the relationship between the various map elements and their general composition; each one should be placed with the overall look and impression in mind.

Harmony Addresses the interplay of e.g. colors, patterns, fonts and other visual elements. While not essential to the core functionality, it influences user satisfaction greatly.

These principles are a valuable guideline for a general design focus and should be chooses in such a way that the intent of the specific map is supported. As mentioned before, we envision floor plans as highly interactive elements, that help to organize, understand and communicate data related to facilities. Users should be able to grasp the spatial layout of a facility quickly, reading cues even on first glance. Architectural floor plans dominate the field of indoor maps but are criticized for their low usability due to an high level of detail and their low visual attractiveness, which breaks the principles of clarity, contrast and harmony.

In order to establish a good user experience and the means to orient and navigate the facility effectively following set of features should be respected. On a first layer important visual landmarks should be highlighted on the map, such as common kitchens, rest rooms, cafeteria areas and entrance halls. Landmarks are a concept that exists in traditional cartography

but has been tested in indoor settings with satisfactory results as well [Lorenz et al., 2013]. Another technique to improve general readability by establishing clarity and contrast is the introduction of different color coding of rooms. This can be used to differentiate between public hallways and private office rooms or to highlight the affiliation to certain complex parts. Augmentation of the floor plan with diverse kind of information can help to improve readability, but in order to prevent data cluttering and ensure clarity and order, the user should be able to filter and show/hide information freely at any point in time.

A successful transformation of a classical floor plan into an improved and more readable version can be seen below. 6.2 depicts the floor plan of the main building of the Aalto University on the Otaniemi campus, which is notorious for its complex layout and indeed the floor plan on the left is overloaded with details with little to no contrast. However, the updated floor map on the right uses colors according to the building parts and highlight points of interest such as info points, lecture halls and bistros, which arguably creates a better user experience. By consequently applying the mentioned design principles, the users capabilities

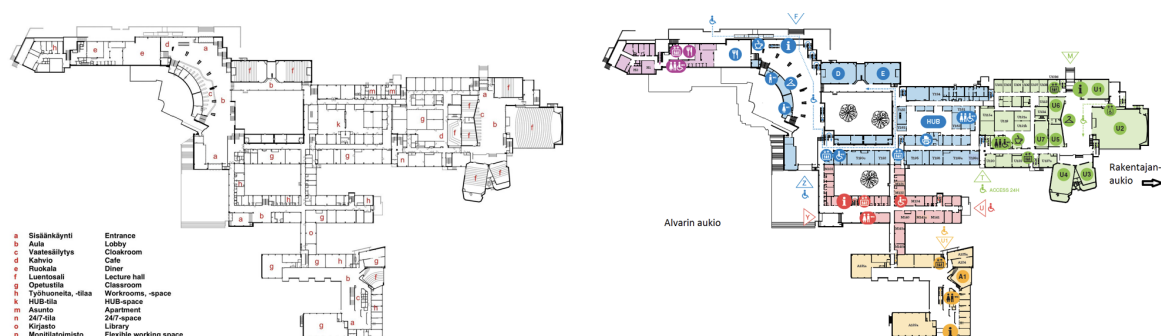


Figure 6.2: Floor plan of the undergraduate center of the Aalto University on the Otaniemi campus in two different versions [Aalto University, 2017].

to orient and navigate himself on a given map are increased, which leads to an higher overall user satisfaction.

6.2.2 Information Loss

When an architect creates a floor plan of a building, he will typically choose a certain height (around one meter in most cases) and create a cross section. Structures cut by this section are represented via a single footprint on that particular height and other elements that either lie beneath or above it are omitted, which means that information about e.g. piping under a suspended ceilings or the position of light element are not represented. Especially for the FM team, these information can be of value, which is why we applied a combination of a traditional cross section and orthographic projections. For each relevant element in the IFC we created a cross section at its center point, which are collected and finally congregated into

a single planar projection (see section 6.3). This way we are able to represent all elements with a distinct foot print, no matter its vertical alignment.

A side effect of this approach is the cluttering of the resulting floor plan: walls, windows, doors and spaces are suddenly overshadowed and convoluted by a network to pipes and ducts. However, by applying interactive, SVG based maps in a React environment, elements of the floor plan can be removed or added from/to the scene by command and at any point in time (see 9). On loading the web service for example, the floor plan is presented in its default form, only highlighting spaces, walls, doors and windows. Only by further interacting with the service, more information, like the layout of air ducts, will be added to the scene. Still, the very nature of such a projection implies that any vertical position is lost with the collapse of that dimension—an observer is able to locate a light installation in regard of x- and y-coordinates but doesn't know about its height. Overall, it is harder to comprehend a space, in spatial dimensions and looks, just from its 2D projection.

We addressed this issue in two ways. First, we applied photo spheres, which are 360° panoramic images. This simple methods allows users to effectively explore spaces remotely and understand local circumstances. Additionally we applied means to link context to these scenes, which is expanded on in chapter 9. Photo spheres are able to convey information visible to the naked eye, but the location and course of piping stays hidden behind walls. This is where the second method becomes relevant: the integration of 3D viewer. However, instead of showing a holistic view of the building, we are only displaying elements and parts of the systems that are supplying the particular space, hence minimizing the cognitive and computational load. This filtering is applied beforehand using pythonOCC and relations inside the IFC and afterwards the resulting models are hosted via a separate service; chapter 9 elaborates on the specifics in greater detail. With these means in place, FM practitioners and other users are able to access spatial information gradually and depending on their specific need.

6.3 Generation from IFC

The generation of the floor plans is done prior to the actual web service and uses Python scripts to extract the relevant information from the IFC. IFC actually supports a concept of strictly separating between semantic and geometric information. Product shapes are described through the *IfcShapeRepresentation* entity and denoted using the *Representation-Identifier* attribute, which supports variety of different representation.

One of which is the *FootPrint* type, which is of course potentially interesting for the desired representation of floor plans. However, during early assessment of different IFC files, both public and private, we failed to find an example implementing said presentation. As men-

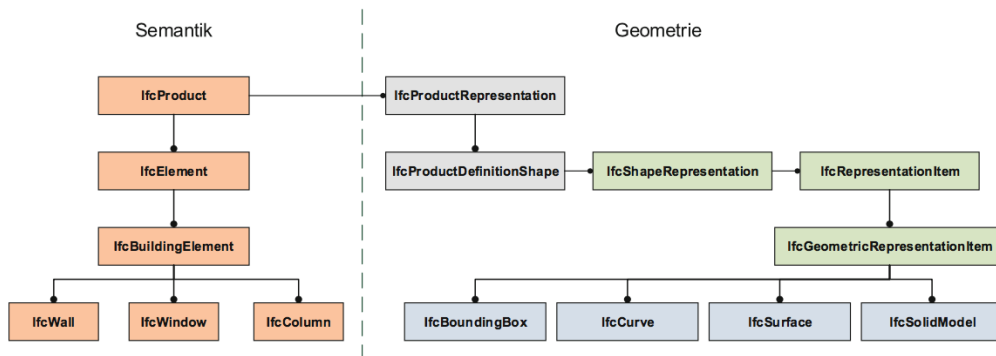


Figure 6.3: The IFC data models separates between semantic and geometric information, enabling a dynamic linking [Borrmann et al., 2015].

tioned in 5.1 the holistic approach of the IFC has the effect that only a small set of the supported features are implemented. In order to reliably expect a given, less common feature to be implemented, it has to be arranged so among the project partners and also ultimately supported by the exporters of the applied BIM software tools. Consequently the search for an alternative was necessary and such was found within the IfcOpenShell package. In addition to opening and reading information inside the IFC, this package allows to generate shape representation of the different entities. Using yet another library, in fact the same one used inside the IfcOpenShell package, we are able to not only display those shapes but also to manipulate and intersect them. As hinted before there are different strategies when calculating the cross section of an three-dimensional object.

After first trials, the selection was narrowed down to two methods: *Central Cut* and *Maximum Cut*. The *Maximum Cut* method takes a certain step size of vertical height units, which is defined before the execution of the program. During execution each individual entity inside the IFC, provided it has a geometric representation, is intersected with a 2D plane, generated at the current height step. After each loop, the area of the active cross section is compared with the newly calculated one and if exceeded, replaced by the new cross section.

Central Cut is a simpler approach, by applying only a single section per entity, that is—as the name suggest—at its central height. Both methods were applied during the scope of this thesis. *Maximum Cut* produced better results for objects with a very irregular shape in vertical direction, but was considerable more computational intensive. Another issue was the granularity of the chosen step size: In many cases, like Walls, the recalculation of cross section was a pointless operation and would have benefited in decreasing the refinement. However, such a reduction resulted in skipping maximal cross section in object like pipes.

Finally we settled with the *Central Cut* approach, which had a superior execution time and decent results for most of the object of interest for us. For instance, pipes and ducts tend to have their maximum around the center point. Of course this approach runs into problems

very easily and was chosen mainly due to time concerns and the above mentioned conditions; chapter 10 elaborates on enhanced methods.

6.4 Export to SVG

Section 5.3 introduces the SVG format, which is used to represent floor plans in the final prototype. One important feature of SVG is its XML-based text structure, each component, styles and transformations are defined through it. This structure is very repetitive and thus favors the generation via computer code, for example through an exporter. Such an exporter was implemented during an early iteration in the scope of the python based preprocessing. During the exporting process, the previously extracted IFC elements are taken as input and transformed into SVG based `<path>` objects, based on the coordinates of their cross section. There are certain aspects to be respected: Firstly, the coordinate system inside the SVG has a y-axis pointing in the opposite direction as in CAD based software tools. Hence, as part of the extraction, the sign of each y-coordinate was flipped. Also in order to avoid the generation of lines connecting disjointed faces belonging to the same intersection, a special distinction had to be made. The `<path>` element supports different predefined commands, e.g. *moveto* and *lineto*. Those commands either connect the current point with the subsequent one via a line or by simply moving to it without drawing anything, which of course answered said issue.

Another aspect to be regarded is the opening footprint of doors. This issue cannot be solved by simple geometry but requires information about the type of door, the location of hinges and the direction it's opening. IFC as a container of semantic data holds such information and is commonly included in most exporters as well. The opening type of a door is specified via the *IfcDoorType* entity, which holds a relationship to the *IfcDoorTypeOperationEnum*. This enumeration defines a set of common operation types, including the position of hinges, the exact style of opening and, if applicable, the number of panels. The opening direction however, is defined through the positive y-axis of the local coordinate system of each individual door. By checking those various points, we were able to export the opening footprints into SVG. Of course most of such footprints include arches, which can be drawn as part of the `<path>` object through the *elliptical Arc* command. In the case of our prototype three common operation types were implemented with the possibility to extend it by more cases. Finally to be able to connect to semantic information outside the SVG file, each generated `<path>` object is wrapped through a group tag and extended by an id property, holding the respective GUID and another property for storing the *IfcType* of the component.

The end result of applying the previous steps is a SVG file containing every element marked for export and including GUIDs as links to the information stored outside. On its own the generated file was little useful, after all it includes not only walls and spaces but all MEP

related elements as well, rendering it hard for direct use due to cluttering. However, it becomes useful when incorporated into a web environment, which is naturally supportive of [SVG](#) as a result of the close relation between [XML](#) and the [HTML](#) structure. In an early version of the prototype, the whole [SVG](#) was first loaded into the web application and then filtered based on [GUIDs](#) retrieved from the database. This approach came with several shortcomings, most notably performance issues due to the sheer size of the [SVG](#) and the many traversals needed, when filtering for the respective elements. Also by generating the file prior to the use in the web application, styles are defined beforehand, which limits variability. The solution to this dilemma was found by migrating the web app to the React framework.

React allows to build [HTML](#) pages in a component-based approach, which also includes [SVG](#) being indirectly [HTML](#)-based. This concept is very powerful because it does not require a single hefty file that has to be loaded at once, but instead remains dynamic with individual [SVG](#) component loaded on demand. Hence, instead of creating `<path>` objects as strings and composing them into a single file, the naked path coordinate strings are stored alongside the semantic information of each element in the database. During the execution of our app and depending on the nature of the selected task—e.g. showing the base floor plan or displaying a certain systems—elements and their [SVG](#) coordinates can simply be retrieved. The styling is no longer handled statically through a precompiled [SVG](#) file but can be assigned dynamically on the front-end. This approach is further elevated through the powerful rendering checks inside the React framework: preventing the re-rendering of components needlessly.

6.5 Summary

In this chapter we highlighted floor plans, their origin and their special relation to traditional cartography. Indoor positioning and the [SVG](#) format provides interesting opportunities for the application of a new generation of interactive floor plans. Further we introduced their use in a [FM](#)-based setting for the augmentation of data with spatial context and argued their superiority over [3D](#) approaches. Problems of readability and loss in three dimensional information through the abstraction into the [2D](#) plane are answered via the application of good design principles and by integrating photo-realistic 360° panoramas alongside selective [3D](#) room models. The geometric data for the floor plans is aggregated through [IFC](#) files and the use of the `IfcOpenShell` framework, generating cross section for various elements separately. The compilation into a single holistic [SVG](#) file was explored but ultimately abandoned, due to poor performance. Using the popular React framework enabled the rendering in a dynamic and distributed manner and was ultimately applied for the prototype.

Chapter 7

Graph-based Data Storage

Database-Management System are part of almost any considerable software application and crucial for its performance, usability and supported use cases. An example of such a system is Neo4j, which was applied during the implementation of the presented web-based prototype. Neo4j stands out from other popular [DBMS](#) by being graph-based, which is less established when compared to traditional relational [DBMS](#) or the popular MongoDB system. In this chapter we will provide a justification for choosing the applied graph approach over more common ones. Next the data model is introduced and the various factors of consideration are highlighted. Similar to the generation of floor plans, the graph is derived from [IFC](#) but further condensed using [FM-Handover MVD](#) (see section [7.3](#)). Finally the access of the stored data using the Cypher query language is explained, before ending with the introduction of important use cases.

7.1 Why a Graph Approach?

Section [5.2](#) already provided some of the reasoning on why graph databases have been favored during the assessment of different solutions; in this short section we will recapitulate those reasons, add new ones and expand on others.

All of the current top four most applied [DBMS](#) are based on a relational data scheme [[DB-Engines, 2018b](#)]. This popularity is not without reason and is based on their consistent, secure and —given a proper design and a sensible query— fast retrieval. Of course relational databases have been around the longest, are a very mature technology consequently and are well understood by most. However, a shortcoming of significance is the inflexibility of their data schemes, which have to be defined in advanced and under much effort. Any change requests at a later point in time require, more often than not, to restructure the entire data base. The [O&M](#) domain is very heterogeneous and while [FM](#) applications are highlighted in

the particular case of this thesis, the envisioned **DBMS** might be used by multitude of different systems and stakeholders. Considering the needs and requirements of every single potential user in advance and formulating a omnipotent scheme is near impossible. Consequently relational databases fell out of consideration and instead focus moved to NoSQL databases, which provide a more flexible scheme, one that can be changed and adapted with more ease later on.

As already mentioned in 5.2, the NoSQL systems encompasses a whole family of different types, including graph-based approaches. However those actually present the minority and are left far behind more popular systems like MongoDB; what made graph systems more attractive to the presented use case? A crucial aspect that ultimately led to said decision was given by the structure of the **IFC** itself. **IFC** is the standard of exchanging information between partners in **BIM**-based construction projects and is able to store a magnitude of information—geometric and semantic alike. Representing such a structure inside a database is a difficult tasks, due to the sheer complexity of the **IFC** data model. Graphs happen to be the perfect medium for representing the **IFC**'s complex and tangled structure, which builds heavily on relationships. This tendency was further consolidated by consideration towards typical **FM** use cases that often deal with system flows and way finding, both of which are dealt effectively by graphs. At least on paper, graph databases appear to be the optimal solution to the proposed problem statement of the **FM** domain, which of course was further examined during the implementation of the prototype.

7.2 Graph Data Model

A common misconception among novice users of NoSQL databases is that using a more flexible systems that can adapt over time, makes the formulation of a data model a pointless task. If anything, dealing with the this flexibility makes it even more important to formulate a model that gives room for extensions and changes. A nice aspect of graph databases in particular are their "whiteboard friendliness" [Robinson et al., 2015]. During early drafts it is common practice to sketch out entities and relations on paper or whiteboards, which is typically done in the form of graphs. For the conception of the particular data model concerning the developed prototype, various factors had to be considered.

First and foremost, by applying Neo4j as the graph database solution of choice, we are bound to the same type of graphs that are used under their hood. As mentioned earlier (see 5.2.1), Neo4j applies *labeled property graphs* and hence any further examinations apply specifically to those. Neo4j also supports a set of best practises that ensure that applied queries can be executed in a simple and efficient way, which were also uphold whenever possible. Another essential influence is of course the **IFC** for reasons stated earlier and became the backbone of the designed data model. Finally, the placement in a **FM** setting provides a distinct set of

requirements, which were input of highest priority. The mentioned aspects are expanded on in the following.

7.2.1 Best Practises

One of the big virtues of graph databases is the possibility to run queries on sub-graphs, virtually omitting the complete rest of the data base. The effective filtering for sub-graphs depends on a sensible data model of the underlying graph. The following provides some best practises.

Neo4j consists of four basic elements: nodes, properties, relationships and labels. Nodes generally represent entities in a system: objects, either physical or virtual, that carry identity (see figure 5.5). A challenge that is often encountered during modeling is connected to distinguishing between entities that should be modelled as nodes in a graph or as mere properties attached to another nodes. Generally an outsourcing makes sense if the property is of higher complexity—not a primitive type—and carries importance in the larger graph scheme. In the case presented in this thesis, we mainly translated *IFC* entities to nodes—if included in the *FM*-handover *MVD*—and the included attributes inside the *IFC* as their properties.

Another good practise when modeling a domain as a graph, is to avoid relationship with very generic names. Using queries in connection with these generic relationships, that are applied for various kind of semantically different relations, will result in considerable larger sub-graphs and hence increase the execution time. The other extreme of defining specific names for each relationship combination will produce the fastest queries but increases the complexity of the graph itself and complicates the formulation of queries. It is recommended to find the balance between to the two ends and in case of doubt it can make sense to define both; after all increasing the relationship count does not necessarily increase the evaluation time of queries.

During the modeling process, it is often hard to decide from which reference point a certain relation shall be named: e.g. *CONTAINS* or *IS_CONTAINED* and a possible answer could be to introduce both, hence presenting a bidirectional relation. While such bidirectional connections make sense from a conceptional perspective, they carry no new meaning and only introduce wasteful load. This is further enforced by the fact that the speed of Cypher queries is not dependent on the definition of a direction, in fact the language allows to omit the direction all along. Therefore the clear recommendation is to avoid bidirectional direction and instead decide for a single reference point.

7.2.2 IFC Data Model

Having a clear image of the best practises of graph database modeling, one can start looking at the IFC structure. Let's recall the structure of the IFC as discussed in section 5.1 (specifically figure 5.3). The inheritance from the *IfcRoot* entity gives a first indication for the modeling of nodes—after all, entities inheriting from it are uniquely identifiable through the GUID property. Consequently labels of our graph are given through the various types inside the IFC, such as *IfcSpace*, *IfcWall* and so forth. Properties describing a certain objects are either defined through direct attributes on the object or through indirect relation via relationships like *IfcDefinesByProperty* or *IfcDefinesByType*. A possible first step in modeling the IFC into a graph is the representation of the EXPRESS schema as proposed by [Ismail et al., 2017]. Ismail et al. [2017] are using said representation to check and validate IFC files, to track changes between IFC versions and providing a tool for further analysis. Due to time constraint this step was not applied in the presented case and instead we only focused on the direct translation between IFC entities to graph nodes.

When comparing the IFC data model with the general concept of Neo4j's graph database, one can identify a conflicting treatment of relationships. IFC applies objectified relationships, meaning that relationships between objects are represented via its own separate relating object. Such a concept could be implemented in graph, such as Neo4j, but lacks the necessity to do so, with relationships already being "first-class citizen", including their own properties. Also such an approach conflicts with the best practises of avoiding non-value-adding relationships. Hence relationships inside the graph model were represented as direct connection between nodes. Furthermore, after applying a first translation between IFC and the corresponding graph schema, certain barriers were encountered:

1. The applied IFC, supplied by National Institute of Building Sciences (NIBS), missed most of the possible relationship types while others were implemented faulty
2. Despite introducing direct relationships, the graph was still many thousands of relationships large

A set of simplification was needed. One such action was to abandon the modeling of separate attribute nodes, as implemented inside the IFC, and instead simply storing attributes directly with the respective object. This approach can lead to duplication of attributes, but is—considering the size of the test model—tolerable. More importantly though was the application of a method, already well known in other conventional BIM cases: Model View Definition.

FM Handover Model View Definition

As stated before the sheer size of the resulting **IFC** graph made it difficult to comprehend the structure in its entirety. A similar problematic has been encountered by the industry when working with **IFC** files in certain specialized settings in which only a subset of the data model is relevant and as a result the so called **MVD** was envisioned. **FM Handover MVD** is one possible example of those and describes the requirements needed in order render the **IFC** file useful for the **FM** domain. On their website buildingSMART [2009] provides, alongside other resources, a detailed document including the necessary **IFC** entity types, their attributes and the relationships to each other that has to be included. In order to fulfill the requirements set by the **MVD** all of the given aspects have to be met. Figure 7.1 depicts the various **IFC**

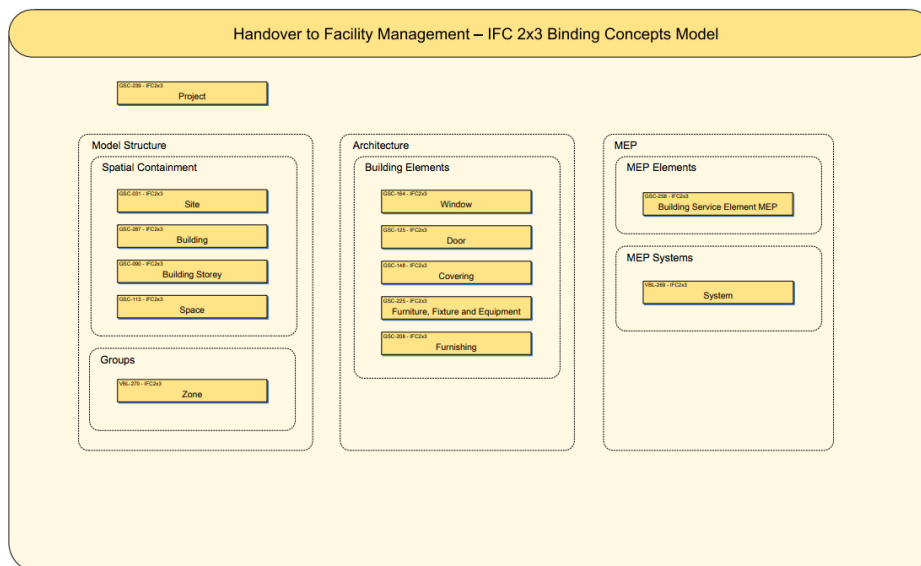


Figure 7.1: Binding concept model of the **FM Handover MVD** [buildingSMART, 2009].

entities to be included, split into model structure, architecture and **MEP**. buildingSMART also provides a more formulated set of recommendations towards software vendors that aim to either export or import **IFC** complying the **MVD**. Some interesting points include:

- Assignment of doors, windows, furniture, fixture and equipment to spaces
- Assignment **MEP** components to spaces and technical systems
- Export of base quantities

Additionally several recommendation towards **COBie** are listed, which is a valuable source of information, especially for manufacturer related information. However, as stated before, **COBie** is not considered here, due to limitations of scope and the fact that it does not hold spatial information, which is necessary to support the proposed concept.

Summary

Applying the FM Handover MVD greatly reduced the scope of to be considered entities and relationships inside the IFC. There were however small notifications applied. Firstly, relations towards the *IfcProject* and *IfcSite* were excluded, both of which are remnants of a project-based mindset and seldom useful inside the FM domain. *IfcZones* were omitted due to not being implemented inside the trial IFC; so were *IfcSystems* however their functionality is crucial and hence a workaround was applied (see section 7.3) Furniture and furnishing elements were omitted due to their practical absence in the trial model file. Furthermore, *IfcCoverings* were swapped for *IfcWalls*, which carry significant information regarding doors and windows and are also crucial part in the floor plan representation. Finally, in order to comply with the best practises of graph modeling, the relationship names defined in the IFC were mapped onto more descriptive ones, easing queries by providing better specification.

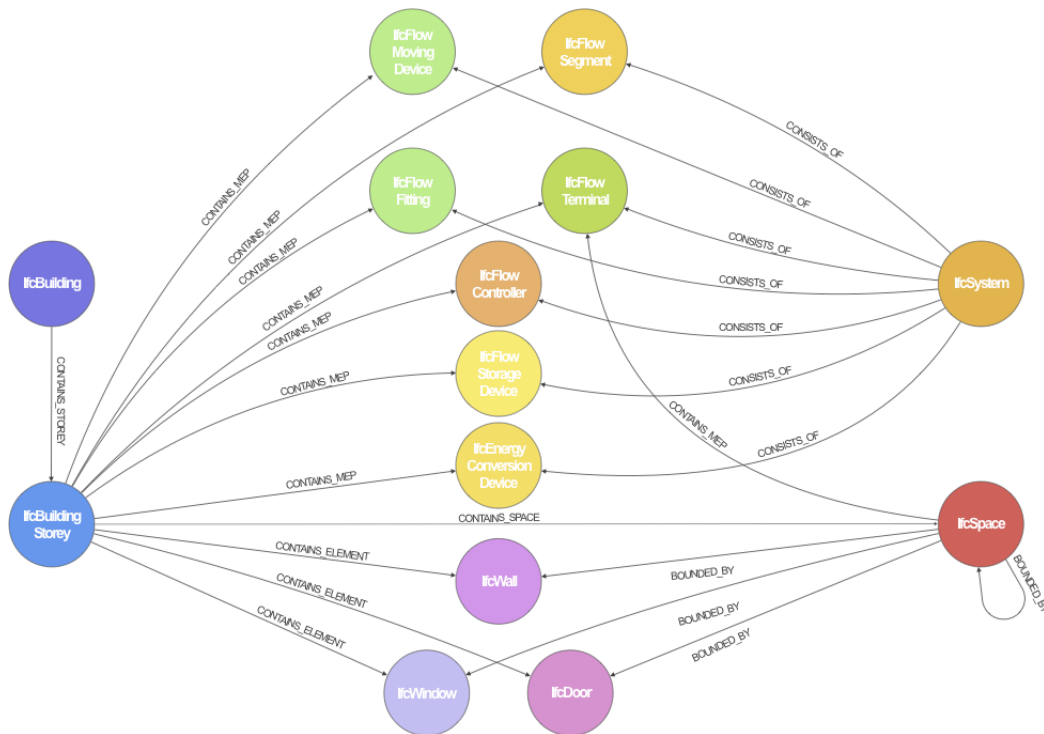


Figure 7.2: Intermediate graph model including IFC entities and relations.

7.2.3 Facility Management

Facility Management has a diverse set of requirements and data items to capture and maintain. Some examples—commonly found in CMMS landscape—are work orders, status checks, assets and equipment, repair-history, work force integration, scheduling and more. Of course those various activities have to be supported by a database system, such as the one envision here, by extending the existing IFC-based graph models with the respective concepts from

the **FM** and **O&M** domain. During the development of the prototype we exemplarily implemented the concept of work order, which are connected to spaces and to so called *HotSpots*, which are part of the interactive photo sphere concept presented in chapter 9. This leaves us with the following resulting graph:

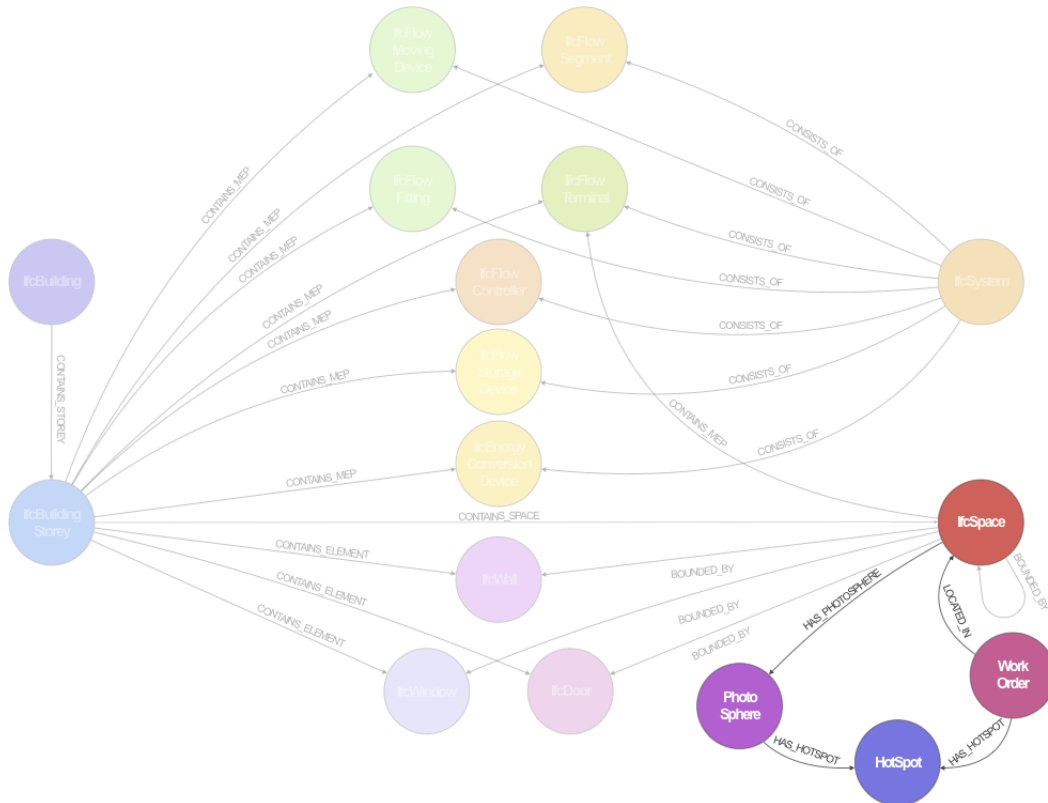


Figure 7.3: Graph model including **IFC** entities, relations and a preliminary **FM** concept of work orders.

7.3 Graph Generation

The generation of the graph is executed prior to the web service of the presented prototype and uses the scripts written in the Python language alongside modules such as IfcOpenShell, PythonOCC and Py2neo. In the following we present the exact process of bringing **IFC** semantics, geometry and more information beyond that into the graph database.

7.3.1 IFC Objects

In a first step, various objects of specific types are extracted from an arbitrary number of **IFC** files, which are specified in the beginning of the script together with the types to be extracted

from each file. The extracted types for the case of our prototype can be seen below, in figure 7.4.

ARCH	lfcBuilding	lfcBuildingStorey	lfcSpace	lfcWall	lfcWindow	lfcDoor
MEP	lfcEnergyConversionDevice	lfcFlowMovingDevice	lfcFlowTerminal	lfcFlowSegment	lfcFlowFitting	lfcFlowTreatmentDevice
	lfcFlowTreatmentDevice	lfcFlowController	lfcFlowStorageDevice			

Figure 7.4: IFC types extracted from the "office" IFC files, provided through open access by NIBS.

IFC supports a complex technique of storing attributes, which are either stored directly on the entity itself, through the affiliation to a type (e.g. certain type of wall) or on a designated entity, *IfcProperty* (see figure 7.5).

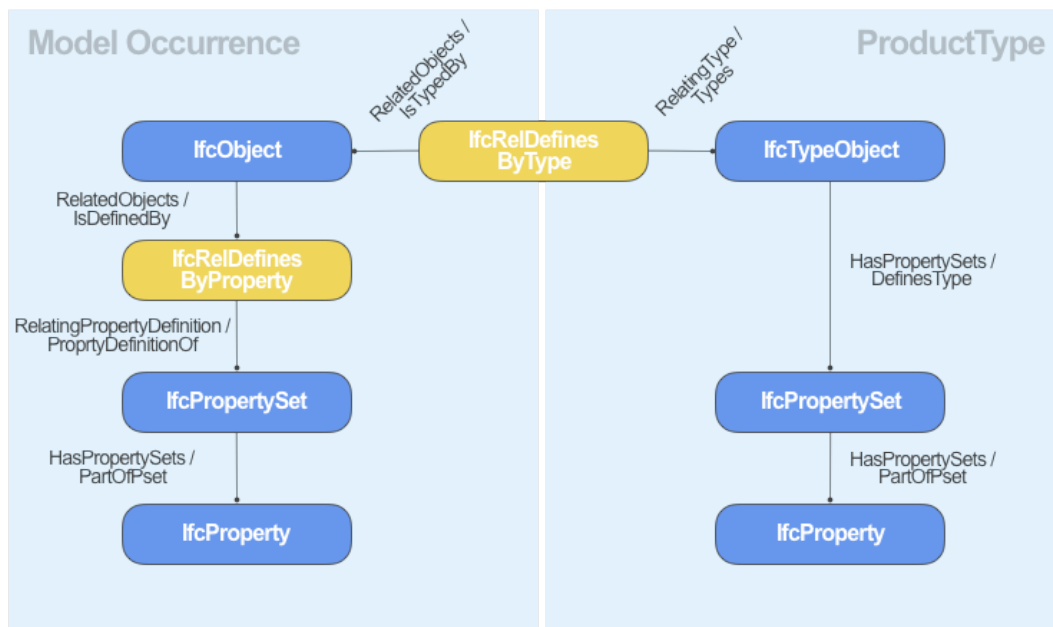


Figure 7.5: IFC property storage scheme for both object and type related sets (based on [buildingSMART, 2006])

As stated before instead of supporting a similar complexity inside the graph model, the various attributes are stored directly with the object itself. In order to provide a relatively easy and flexible way of defining and retrieving said properties, the user is able to formulate configuration files.

Three distinct extraction are supported: Firstly and most simple, the extraction of direct attributes by giving the exact name of the respective attribute name. Secondly, the access of attributes on secondary entities, given through an array of attributes names. This can be used to retrieve information, stored on indirect IFC entities. For example, such an extraction was done to retrieve the address of the building, which is stored indirectly through the *IfcPostalAddress* entity. Finally, the object specific property sets can be accessed by specifying

```

1  {
2      "IfcBuilding":
3      {
4          "ifc_global_id": "GlobalId",
5          "ifc_name": "Name",
6          "ifc_address": ["BuildingAddress", "AddressLines"],
7      },
8      "IfcBuildingStorey":
9      {
10         "ifc_global_id": "GlobalId",
11         "ifc_name": "Name",
12     },
13     "IfcSpace":
14     {
15         "ifc_global_id": "GlobalId",
16         "ifc_name": "Name",
17         "height": ["@PythonOCC", "occ_height"],
18         "volume": ["@PythonOCC", "occ_volume"],
19         "_svg_path": ["@PythonOCC", "occ_path"]
20     },

```

Figure 7.6: Excerpt of the attribute configuration file.

`PropertySet` keyword followed by the name of the target property (as seen in the figure above). Another key word is the `@PythonOCC`, which indicated a geometric computation such as height, area, volume and path, using functionality of the `pythonOCC` library. Each of those secondary keywords trigger the relevant geometric calculation with the latter referring to the calculation of the `SVG <path>` coordinates.

7.3.2 IFC Relations

Similar to their object counter parts, relationships are extracted via the use of configuration files (see figure 7.7). They consists of the `IFC` type to be extracted, the name of the attribute referencing the root entity (commonly referred by the "Relating" prefix), the leaf entity/entities (connected to the "Related" prefix) and finally the definition of the considered types for both root and leaves. For improved querying and better contextual meaning inside the graph database, relationships are mapped to new names, which are specified in the beginning of each definition.

```

1  {
2      "CONTAINS_STOREY":
3      {
4          "ifc_type": "IfcRelAggregates",
5          "ifc_root_rel_attr": "RelatingObject",
6          "ifc_leaf_rel_attr": "RelatedObjects",
7          "ifc_root_type": "IfcBuilding",
8          "ifc_leaf_type": "IfcBuildingStorey"
9      },
10     "CONTAINS_SPACE":
11     {
12         "ifc_type": "IfcRelAggregates",
13         "ifc_root_rel_attr": "RelatingObject",
14         "ifc_leaf_rel_attr": "RelatedObjects",
15         "ifc_root_type": "IfcBuildingStorey",
16         "ifc_leaf_type": "IfcSpace"
17     },

```

Figure 7.7: Excerpt of the relationship configuration file as applied during the pre-processing.

Inside the algorithm the respective IFC relationships are traversed and the respective root and leaf elements are accessed according to the specification inside the configuration file. If successful, the matching IFC objects are retrieved from the previous step and an edge is written to the graph database using the Py2neo library, which is a wrapper for Neo4j functionality (see section 8).

7.3.3 Inter-domain Linking

BIM-based construction projects usually employ various IFC models coming from different disciplines like, architecture and mechanical engineering, and are usually created by different software products. In the construction industry it is common practise to check the different models for the alignment of coordinates systems and if any relevant clashes of building elements need to be solved; after all the same problems would resurface during the physical construction. However, besides these geometric checks, there exists no further logic that would link the models further: a structural wall is not aware that it is connected to the respective wall entity in the architectural model and the same building floors appearing in multiple models might have the same attributes but divergent GUID. This is problematic, when bringing semantics of several files into the same graph database and in order to truly create singular representations and obtaining a cohesive representation of the complete semantics, links between the respective models are needed.

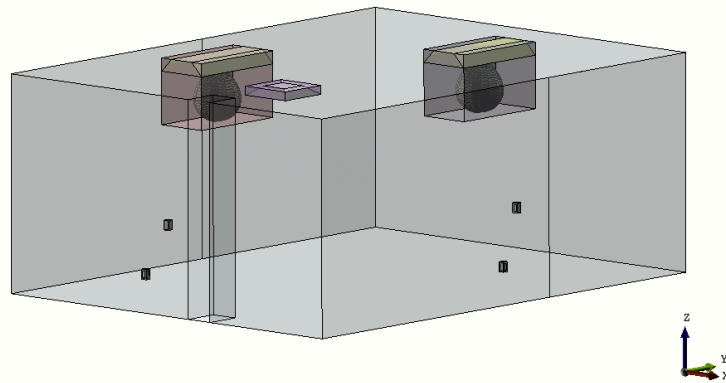


Figure 7.8: Intersecting bounding boxes of several *IfcFlowTerminal* elements with surrounding *IfcSpace* (visualized in pythonOCC 3D viewer).

Such inter-domain linking can be achieved by utilizing the fact that models are geometrically aligned and was done so in the presented prototype. After extracting the specified objects from the IFC and generating the shape of each—as long as having a geometric representation—their corresponding bounding boxes are calculated using the PythonOCC package. Bounding boxes are a simple tool but powerful in determining intersection or containment between each other. This approach was mainly utilized to assign the various ele-

ments coming from the mechanical file of the trial model with the storeys and spaces coming from the architectural. In most cases we achieved good results, which is due to the relative simplicity of the analyzed model, however the actual validity depends heavily on how well bounding boxes are matching their respective source shape.

7.3.4 MEP Systems

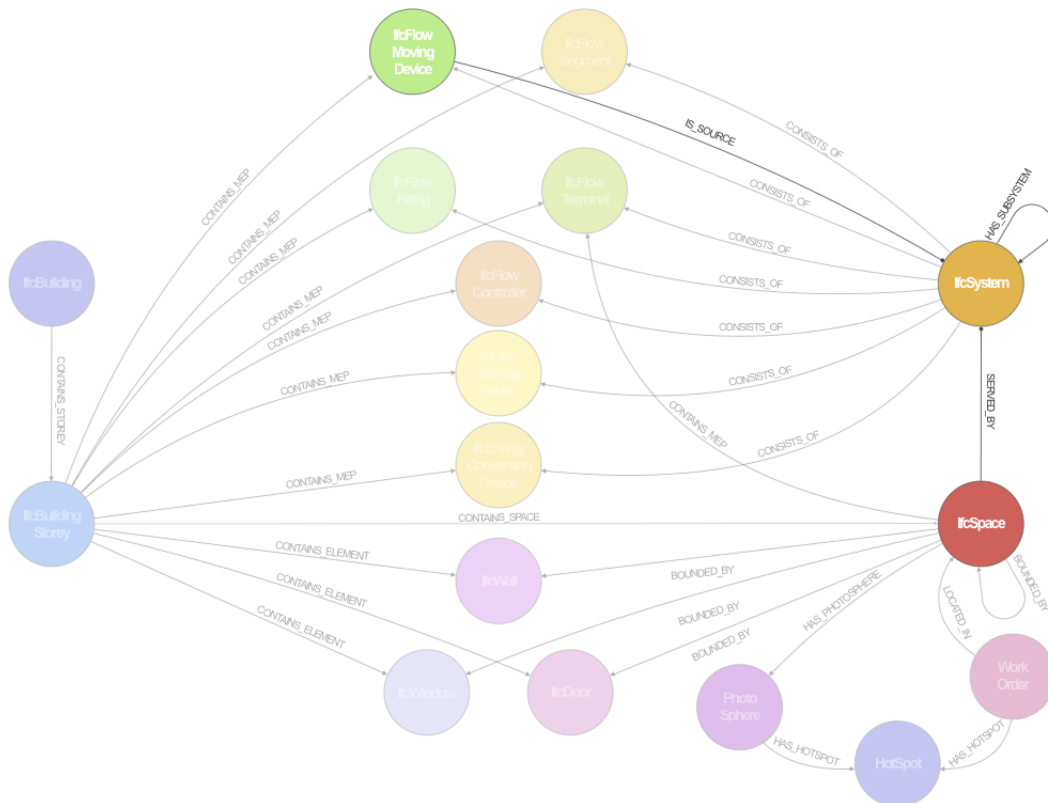


Figure 7.9: Updated graph model including the concept of systems having sources, serving spaces and consisting of other sub-systems.

buildingSMART defines *IfcSystems* as ”a organized combination of related parts within an AEC product, composed for a common purpose or function or to provide a service. (...) The use of *IfcSystem* often applies to the representation of building services related systems, such as the piping system, cold water system, etc.” [buildingSMART, 2006]. Naturally systems are of high interest for the FM team and an important component for any IFC model during handover. Yet, when assessing publicly available IFC models for the implementation of the *IfcSystem* entity and respective relations, almost none did so—another testimony to the previously mentioned current problems with IFC and its handling by the industry. Additionally, this might reflect also the poor maturity of FM and O&M related BIM, there has been too few real-life applications to provide any significant example files. While the example projects provided by NIBS do provide COBie files—clearly implying FM use cases—their models are

not FM Handover conform and as such do not implement any *IfcSystem* relations. Another work around had to be found. During closer evaluation of the trial IFC file, we found that the property sets of various MEP elements contained system-relevant information. In total it contained 20 distinguishable strings, stored under the property name "System Type". Using the different values as a filter and by visualizing the respective elements using PythonOCC's custom viewer a total of 10 systems were identified after close inspection.

On top of the found systems, we defined 3 common super systems: ventilation, plumbing and sprinkler and grouped the found them accordingly; establishing a system to subsystem relationship. After generating custom GUIDs, those systems were written to the graph database and the remaining MEP elements assigned to system and subsystem based on their property values. Furthermore, special elements, such as chiller, heater and pumps were marked additionally as source of the corresponding system. The graph model was further refined by establishing a "space is served by system" relationship by using the prior established relationship between *IfcFlowTerminals* and spaces and the new assignment of the very same terminals and their respective systems—making implicit knowledge explicit (see figure 7.9).

7.3.5 System Flows

Possible questions a FM engineer might ask include: "Where is the air of vent X coming from?" or "Which rooms are effected when fan Y stops working". In order to answer such questions not only do we need information about system-element assignment and what spaces are served by which system, but also about the particular flows between the various MEP elements. Such a concept can be implemented inside the IFC either via *IfcRelConnectsElements* relation (more generic) or by defining *IfcPorts* (more explicit), connecting different system elements and establishing a continues flow. By now it should be little surprising that those relations are seldom implemented by exporters and were missed in the applied trial file as well. Once again a solution outside the IFC had to be found and finally a algorithm was applied for the case of the ventilation system.

The pseudo code of said algorithm is depicted in figure 1. At its basis, this algorithm consists of recursive call, taking a current element of the system and a list of remaining elements as arguments. As any other recursive function a termination condition is needed, which in this case is either reaching an element of type *IfcFlowTerminal* indicated a space has been reached, or when the given list is empty. Inside the function, the list of elements is traversed and for each a intersection with the given "super" element is checked. If such a connection is found the relationship is written to the graph database and the current element is popped out of the list, preventing duplicate intersection. Using the same element as primary argument and the now reduced list as a second, the function is called again; recursively. The initial condition are given by using the source component of a system.

Algorithm 1 System Flow

```

1: procedure CREATE_SYSTEM_FLOWS(sub_system_type, level, source_guid)
2:   sys_components  $\leftarrow$  fetch_from_graph_db()
3:   calculate_bounding_boxes(sys_components, margin)
4:   for sys_comp in sys_components do
5:     if sys_comp['guid'] is source_guid then
6:       source  $\leftarrow$  sys_comp
7:       sys_components.pop(source)
8:       break
9:   create_flow(source, sys_components)
10: procedure CREATE_FLOW(current_comp, components)
11:   if len(comps) is 0 then
12:     return True
13:   else
14:     if comp['type'] is 'IfcFlowTerminal' then
15:       return True
16:   for tmp_comp in components do
17:     if is_bounding_box_contained(current_comp, tmp_comp) is True then
18:       create_flow_relation(current_comp, tmp_comp)
19:       components.pop(tmp_comp)
20:       create_flow(tmp_comp, components)

```

The concept of $(\text{ }) - [\text{FEED}] \rightarrow (\text{ })$ relations was lend from the Brick Schema, which is an open source development effort to create a uniform schema for representing metadata in buildings [Brick, 2017]. Brick is maintained by an alliance of various US and European universities and IBM and follows the principles of a minimal set of simple relations to connect the various entities inside a building (see figure 7.10).

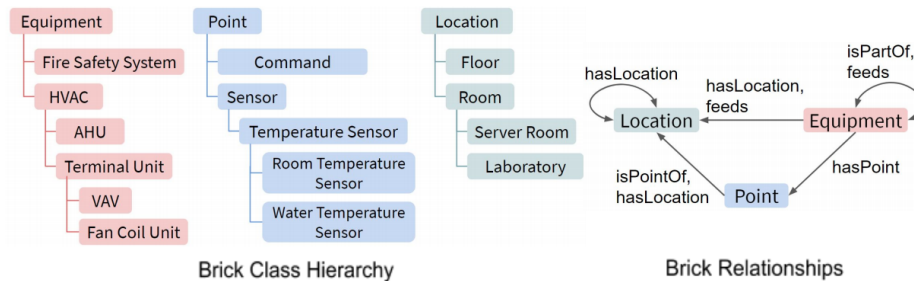


Figure 7.10: Brick relationship model [Brick, 2017].

Together with the existing graph model so far and the new concept lend from Brick, we arrive at the final version of our graph which is depicted in figure

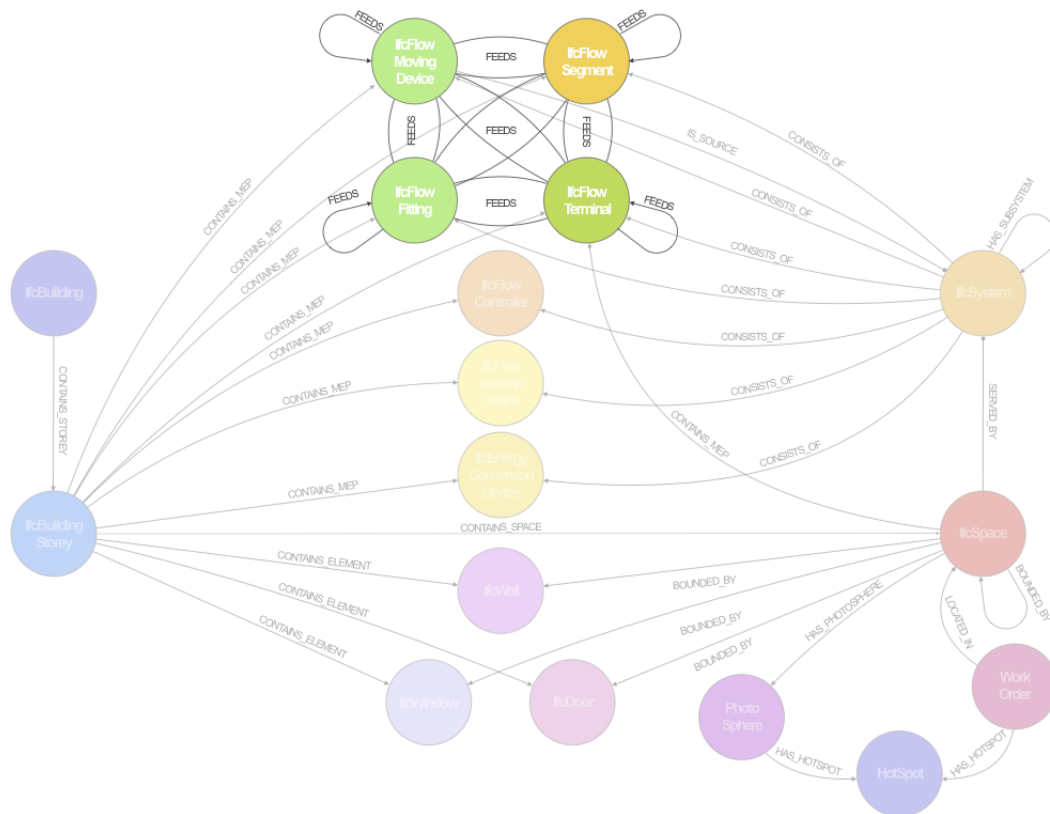


Figure 7.11: Final graph model including flow relations between the various MEP components.

7.4 Access

After filling the database and establishing all relevant relations we are able to run queries and access the information we stored beforehand. Neo4j—the DBMS used for the prototype presented in this thesis—provides a default shipped browser, wrapped around the actual database. This browser provides users with the possibility to get an overview of the underlying graph, see the various labels and relationships that currently reside in the graph, a set of default queries ready to be executed and an input field to write and execute manually written commands. The result of such queries are then, depending on the return type of the query, either visualized as data array or as a full-fledged graph representation of the requested nodes and relationships (see figure 7.12).

In the following some example queries are demonstrated using the using Neo4j’s de facto default query language Cypher. Cypher will be shortly introduced in the following with more details in section 8.2.2. Also the presented browser is just a tool to validate and test certain queries or for demonstrations like this one; later on, the main data access is handled through the endpoints of a custom API, which is connected to our web-based prototype.

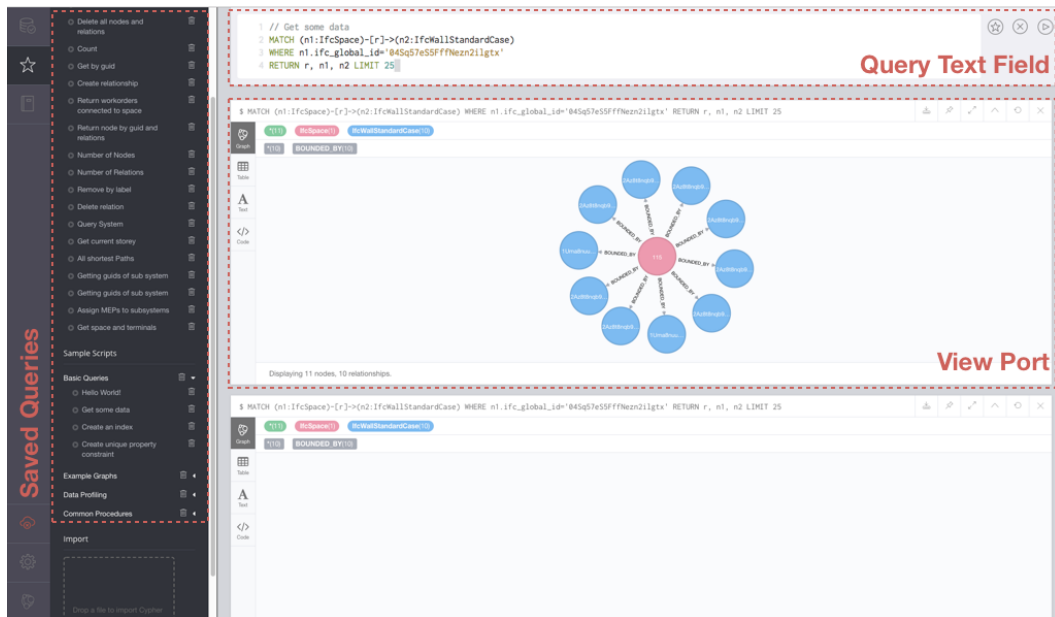


Figure 7.12: Interface of the Neo4j Browser with example query and result visualization.

7.4.1 Get Spaces on Storey

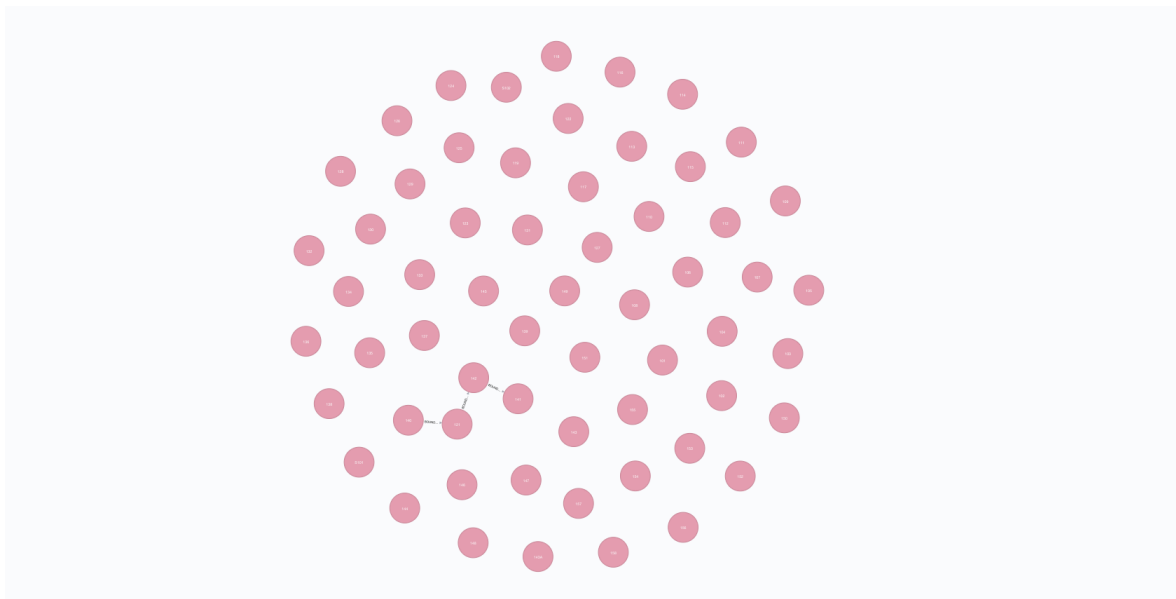


Figure 7.13: Retrieved space nodes for the ground floor of the trial IFC file.

In the first of our set of queries, we are simply retrieving all *IfcSpace* nodes on the ground level. This can be done with the following simple query:

```

MATCH (storey:IfcBuildingStorey) -[:CONTAINS_SPACE]->(space:IfcSpace)
WHERE storey._level = 0
RETURN collect(space)

```

Generally, the `MATCH` clause is used to specify graph patterns, which correspond to sub-graphs of the complete graph inside the database [Neo4j, 2018]. After the data was matched, it is filtered using the constraints given by `WHERE` clause. Finally the resulting *IfcSpace* nodes are retrieved and submitted to the local visualizer using `RETURN`. The resulting visualization can be seen in figure 7.13.

7.4.2 Connectivity between Spaces

In the following query we are retrieving all spaces, doors and the connection between them given by the `BOUNDED_BY` relationship. This will give us all spaces and how they are connected to each other. However, there is a special case that has to be considered, in some cases two spaces are connected via an opening element instead of a door, therefore the relationship exists directly between them. The query for retrieving such a connectivity graph is given below with the visualized result in figure 7.14.

```
MATCH (storey:IfcBuildingStorey)-[:CONTAINS_SPACE]->(space:IfcSpace)
WHERE storey._level = 0
MATCH (space)-[bounds:BOUNDED_BY]->(door:IfcDoor)
RETURN DISTINCT space, bounds, door
```

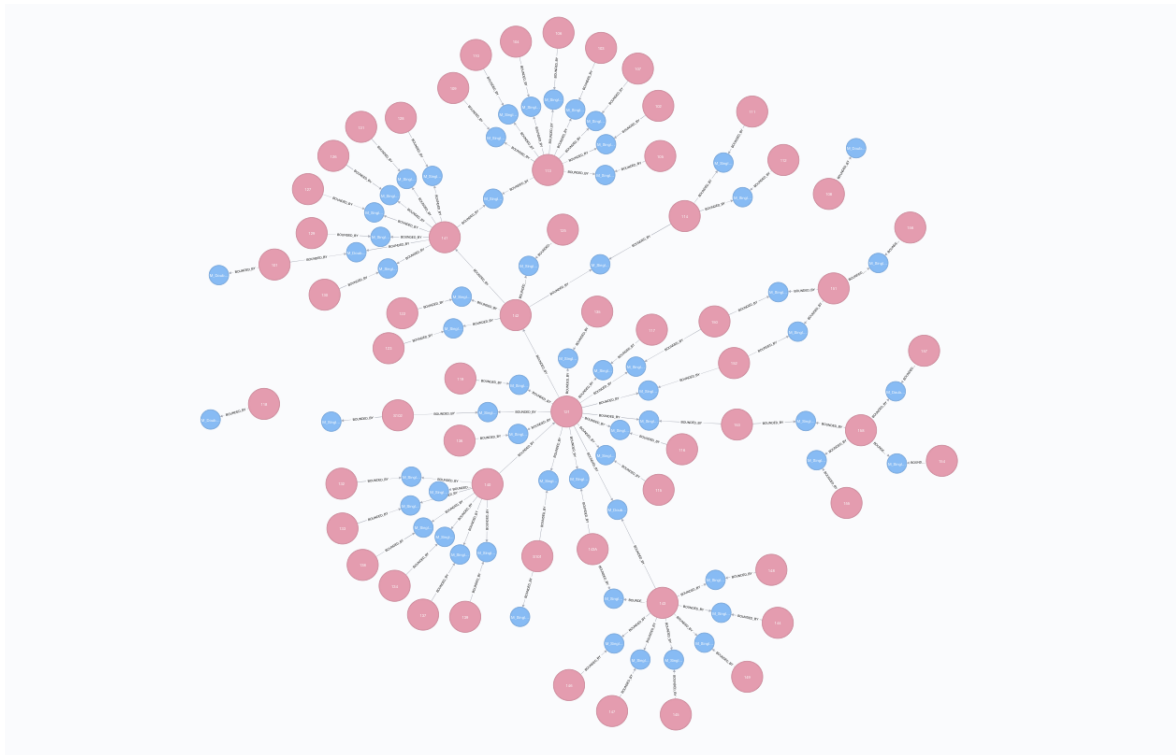


Figure 7.14: Spaces of the ground level and connected via door or opening elements.

7.4.3 Supplying Air System for Selected Room

The last and most complex query uses the various system relationships and the "Feed" concept between the different components to find the path to the supply unit coming from a selected room. Such a query reflects a common use case of FM experts, when dealing with a broken supply of air and the interest where this originates from. In a first step the terminals of the selected space are determined. Starting from them, a path can be created leading all the way back to the indicated source of the respective systems. Finally the space itself and the system node are excluded from the result and the remaining nodes, which represent all the various system elements, are displayed.

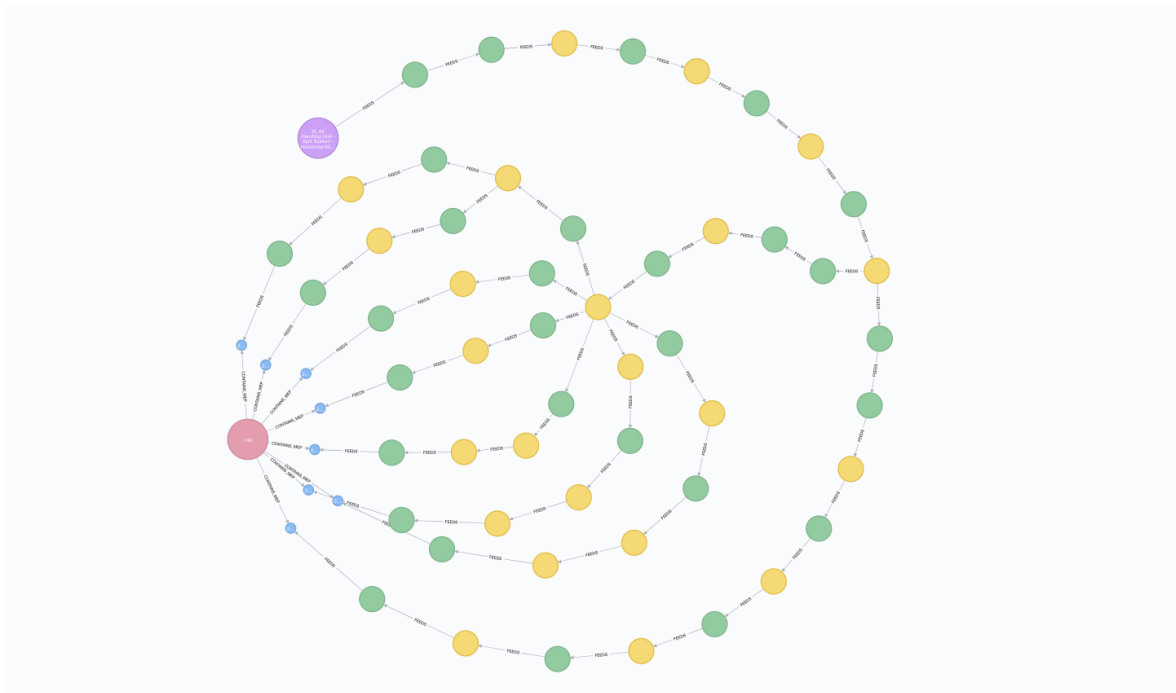


Figure 7.15: Part of the ventilation system that is supplying air to room 104 (red) from the source in purple.

```

MATCH (sto:IfcBuildingStorey) -[:CONTAINS_SPACE]- (space:IfcSpace)
WHERE sto._level = 0
WHERE space.ifc_global_id = '04Sq57eS5FffNezn2ilgCu'
MATCH (sys:IfcSystem)
WHERE sys.ifc_system_type = 'Return Air'
MATCH (sys) <-[:IS_SOURCE]- (source:IfcFlowMovingDevice)
MATCH (space) -[:CONTAINS_MEP]- (terminals:IfcFlowTerminal)
MATCH p = (terminals) <-[:FEEDS]- (source)
WHERE system.ifc_system_type = 'Return Air'
WITH nodes(p) AS sys_nodes
UNWIND sys_nodes AS sys_n

```

```
WITH sys_n  
WHERE NOT 'IfcSpace' AND NOT 'IfcSystem'  
RETURN collect(sys_n.ifc_global_id)
```

Chapter 8

Software Tools and Libraries

The development of the presented prototype was composed of two major phases: the Python-based pre-processing of the [IFC](#) and the subsequent generation of the graph database on the one hand and the web application for presentation and user interaction on the other. The reason for choosing Python over other high-level languages such as C++ or Java was mainly due to its general simplicity, support for writing script-like programs and the presence of a big lively community with many third party libraries. The decision to make the prototype web-based was a very natural choice as well. Web solutions have the benefit of being highly accessible—no matter the operating system or device type, users will be able to access the service. This is especially important for heterogeneous environments as encountered during [O&M](#), where many different stakeholders come together. To be a bit more precise, we decided to use Node.js in the back-end, which is a solution to build asynchronous JavaScript-driven runtime environments, building network applications [[Node.js, 2018](#)].

For the front-end we opted for React, which is a new JavaScript library for building interfaces. React follows the philosophy of splitting individual elements into components which can be controlled and rendered separately from each other. This concept turned out to be paramount when joined with the approach of giving individual building elements [SVG](#)-based representations. Both technology will be described in the following course and put into context with the completed prototype in [chapter 9](#). Additionally we will shortly introduce the different other libraries applied during the prototype implementation, providing a deeper look into the processes behind our development and the chance for the reader to explore further capabilities own their own.

8.1 Python-based Preprocessing

As already mentioned Python enjoys the support of a wide range of different 3rd party libraries. Their presence is mainly due to Python's module support, which allows developers to organize code into logical units and load them as self contained units into any other program. This gave rise to a number of libraries even in the domain of construction and BIM. In the following the most important libraries are discussed.

8.1.1 IfcOpenShell

[IfcOpenShell, 2018] is a software library that helps users and software developers to work with the IFC file format and is distributed under a LGPL licence. The main suite incorporates a diverse suite of software products with IfcConvert being one of their most prominent solutions. Implicit geometry inside the IFC is internally handled using Open Cascade, which is another open-source library, written in C++, for 3D modeling and visualization. IfcOpenShell also includes a Python branch, which makes major functionality of the main products available for a Python environment. Natively IfcOpenShell is written in C++, however its Python branch was still preferred considering our rather basic requirements and Python being a simpler programming language as well as better equipped for scripts.

In our case, IfcOpenShell was solely used to extract information from the IFC, by filters like `by_guid` or `by_type`. Matched entities and their attributes are wrapped inside custom classes, which allows a convenient access of relevant information and subsequent entities. Being open-source and without any major industry support, development is slow and infrequent. However, this is opposed by a small but active and highly supportive community. The simple but powerful concept of direct access to the IFC paired with the supportive community, made IfcOpenShell a ideal match for our prototype.

8.1.2 PythonOCC

Similarly to IfcOpenShell, PythonOCC represents the Python derivative of the previously mentioned Open Cascade library. It is developed by a small developer team but updated frequently and is distributed under the terms of the GNU Lesser General Public License (LGPL). All major functionality of Open Cascade are supported and well recorded through documentation and example code. At start PythonOCC was only used indirectly during the shape extraction from IFC using IfcOpenShell. Next it was used to calculate and generate the cross-sections of the various entities, which—put together— makes up the floor plan. Further, it was used for the generation of bounding boxes, various intersections and the export of model files.

Ultimately these features provided the means to impose semantic connections between domain models, which were either faulty implemented or missing in the first place. PythonOCC, also supports a built-in viewer and while not utilized in the final web application, it was helpful as a checking tool during development. Overall the capabilities derived from Open Cascade are very powerful and with no comparison, considering it being open source. There are a couple of other features that can be used for future improvements of the prototype, which are touched upon in chapter 10.

8.1.3 Py2neo

In order to be able to connect and transfer information to the applied graph database, a connector library is needed. Neo4j offers such for all major programming languages. In the case of Python, Py2neo is a popular alternative, which wraps itself around the default driver and adds various features such as, support for HTTP, admin tools and a higher level [API](#). Popular functions offered by Py2neo are shortcut functions for simple queries such as creating nodes or relationships, matching based on single properties and the translation of JavaScript objects into nodes. At the same time, queries written in Cypher and stored as plain string can still be executed as well. All in all Py2neo is a well maintained and documented library that offers additional features while supporting all function of the official Neo4j driver and thus was a natural choice for the development process.

8.2 Neo4j

Neo4j is used as the primary data storage in our application and is the link between the preprocessing and the actual web application. In this section we will repeat some general aspects about Neo4j before diving deeper into the Cypher querying language.

8.2.1 General

Neo4j is a [DBMS](#) that uses graphs to store and process data, is licensed under GPLv3 and is developed by Neo4j, Inc. Graphs are natively supported inside Neo4j, which means that they are not only used for representation but also in the underlying architecture, which is designed for optimizing fast management and traversal of nodes and relationships [Neo4j, 2018]. Graphs support different models, the most popular one—and supported by Neo4j—is the *property graph model*, which was already covered in section 5.2 and is the basis of the considerations in this thesis. Property graphs consists of nodes, relationships and either of them can be attached with key-value properties. Relationships are named, directed and

always have a start and end node. Property graphs are intuitive and easy to understand while providing the means to model the majority of use cases [Neo4j, 2018].

According to their documentation, Neo4j also provides Atomic, Consistent, Isolated and Durable (ACID) transactions, which guarantee validity of transaction even in the face of errors, such as power failures, connectivity problems and others. Data can be stored directly through nodes and edges and the properties attached to them. Those are key-value pairs with the key being a string and the value that can either be a number, string, Boolean, spatial type or temporal type. After installing Neo4j on a server or other device, it is possible to access and manipulate the underlying database through HTTP calls. Such requests are made easy through the use of driver libraries, which are offered by Neo4j for major programming languages such as .Net, Java, JavaScript and Python. Those provide a collection of functions for easy authentication and execution of various queries. On top of that, Neo4j provides a built-in browser application that can be reached on a separate port. This browser includes links to documentation, tutorials and sample graphs, an overview of the underlying database and means to execute queries using the Cypher language (see 7.12).

8.2.2 Cypher

Cypher is a declarative inspired query language developed by Neo4J but has since been opened up and is used by other software vendors alike [Neo4j, 2018]. During its conception, human readability was a pivotal aspect and as such was not only designed to be understood by developers but domain experts as well. As a results patterns are generally expressed through ascii-art syntax—intuitively recognizable by common users(see).



(n1:lfcBuildingStorey)—[r:CONTAINS_SPACE]—>(n2:lfcSpace)

Figure 8.1: Ascii-art representation of a simple relationship.

Also Cypher was influenced heavily by SQL, SPARQL and to a certain extent by Haskell and Python. Queries in Cypher consist of various clauses. Clauses can be chained and intermediate results can be fed among each other [Neo4j, 2018]. The most common clauses are MATCH, WHERE and RETURN. MATCH is used to retrieve a sub-set of a graph based on e.g. certain node labels and relationship patterns, followed by WHERE, which adds constraints on previously retrieved nodes and behaves just as a filter and finally RETURN simply specifies what should be sent back.

```
MATCH (node1:Label1)-[rel:TYPE]->(node2:Label2)
WHERE node1.propertyA = {value}
RETURN node2.propertyA, node2.propertyB
```

The above code shows a simple example using the previously mentioned clauses: First a certain pattern of two nodes of specific labels, connected via a single relationship is defined. The retrieved sub set is filtered before finally two distinct properties are returned. The matching of nodes can be further tweaked by manipulating the relationship variable: for example additional properties can be matched beforehand `-[{year:2018}]->` or a certain length can be defined `-[:TYPE*..3]->`, which will retrieve all nodes that are reachable within 3 links. Generally nodes and relationships represent patterns contained in the complete graph. Some common patterns are:

```
Shortest path: path = shortestPath( (user)-[:KNOWS*..5]-(other) )
Friend-of-a-friend: (user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)
Tree navigation: (root)<-[:PARENT*]-(leaf:Category)-[:ITEM]->(data:Product)
```

Furthermore, Cypher supports the alternation of the graph using queries and clauses such as `CREATE`, `MERGE` and `SET`. There is much more to Cypher such as a variety of native functions, definition of schemes, advanced use of indexes, etc; such topics are outside of the scope of this thesis but readers are referred to Neo4j's extensive documentation [Neo4j, 2018].

8.3 Web Application

So far we discussed tools and libraries used in preprocessing and Neo4j as our major data storage. In this section we will highlight the solutions used during the development of the web application. All content on the internet is powered by a combination of [HTML](#), [CSS](#) and JavaScript, which creates a situation of many different individuals—designers, developers and others—working in the same environment with similar boundary conditions. The web community is known for its openness and strong exchange and although the problems are similar, people and companies usually find many different solutions to similar problems. Some of those solutions target what is known as the *front-end*, which refers to the part of the application interacting with the user and in charge of the graphical rendering. This is opposed by the back-end, which is generally connected with the server, the web page is running on. The domain of back-end development is special in a sense that it is less strictly bound by the restriction of the world wide web, which gave rise to many languages outside JavaScript to be used like PHP, Python, Perl and many others. In the following we will introduce the technologies used for the front- and back-end of our prototype followed by the major smaller libraries in charge of core features.

8.3.1 Node.js

According to their website Node.js is an asynchronous event driven JavaScript runtime for building scalable network applications [Node.js, 2018]. It was developed 2009 by Robert Dahl in response to the poor performance of current technologies in regard to handling many concurrent connections. Node.js solves this issue not through threading—in fact Node only applies a single thread—but by using non-blocking I/O calls in connection of a event-driven architecture, that informs the systems about the completion of processes, which accounts for its high scalability. Beforehand JavaScript has been a language almost exclusively used client-side and connected to websites rendered in browsers. With the introduction of Node.js and the possibility to use it for server-side scripting—its predominant use up until now—the foundation was laid to use a single programming language between front- and back-end; ”JavaScript everywhere”. This was one of the reasons for the large success of Node.js and a leading factor for us, choosing it over other available technologies.

Another major factor was Node.js’s large support of different libraries through the package management system called npm. It simplifies the publication and integration of third party source code, which naturally sparked the creation of a large and open community of sharing code with the public. In a sum, Node.js was a natural choice, because it offers a powerful server-scripting technologies that uses JavaScript and thus eliminates the need to add another programming language to the stack. On top of this, it is backed by a large and active community with the access to many open libraries, tutorials and example projects.

8.3.2 React

Traditionally when building [UIs](#), developers would often use templates written in [HTML](#) and [CSS](#) and connect the various [UI](#) elements to JavaScript afterwards—the full set of abstraction is dedicated by the given templates [Hunt, 2013]. Interactive components powered by JavaScript in the background had to be manual created each time. In 2013 Facebook released React, a library that followed a different concept: Instead of splitting markup and logic, React divides a User Interface into individual components that contain both [React, 2018]. Facebook also introduced a new language extension to JavaScript called JSX (JavaScript XML), which allows developers to write [HTML](#)-like markup inside a JavaScript file and although not necessary it naturally supports the previously introduced concept. Each component represents an element inside the [UI](#) and implements a certain amount of so called life-cycle methods, which allow the execution of code at certain points of time during the component’s lifetime.

```
4 const panelButton = (props) => {
5
6   const Icon = require(`react-icons/lib/${props.iconName}`)
7
8   let classes = [styles.Button]
9
10  return(
11    <button className={classes.join(' ')} onClick={props.click}>
12      <Icon />
13      <p style={{}>{props.text}</p>
14    </button>
15  );
16 }
```

Figure 8.2: Example code showing JSX, bringing [HTML](#)-like markup into JavaScript.

The most important life-cycle method is `render`, which is required in any component and contains the information needed for being displayed on the interface. Other life-cycle methods can be used for data retrieval from [APIs](#) or for verifying renderings.

Another major aspect of React is its state management. Previously whenever elements in a [UI](#) change (e.g. the color of a button or the value inside a text field), developers had to take care how these changes are executed. Using React, developers only have to specify *what* is supposed to be shown and don't have to care of the *how*, all changes to the [DOM](#) are handled internally. On top of that React maintains a virtual mirror image of the actual [DOM](#) tree inside memory. During renderings, React compares the virtual tree with its counter image and checks for differences; only the parts that are different will be rendered anew. This is called *Reconciliation* and is one of the reasons of React's good performance. In [section 5.3](#) we introduced the [XML](#) structure of [SVG](#), its similarity to [HTML](#) and how it can be rendered natively inside the browser—becoming part of the same [DOM](#) tree. This allows developers to reach into the [SVG](#) and to execute manipulations such as changing the fill color of an icon. Normally [SVG](#) are very small compared to the remaining [DOM](#) tree. However, in the case of our prototype we are displaying large, complex floor plans, including the [2D](#) projection of large networks of pipes and ducts, which can easily span a couple of hundred elements. Constantly rendering such large amounts of elements are extremely resource intensive, which is why React's ability to selectively render certain parts of the [DOM](#) is especially powerful in our application.

React was chosen for this very fact and is a key component of the overall concept of the presented prototype (see [chapter 9](#)). Also, React becomes a more and more popular front-end library and as such offers a rich community support, which was another factor for choosing it over frameworks like Angular or Vue.

8.3.3 Pannellum

As part of the concept of incremental increase of complexity and a tool for exploration and augmentation of [FM](#) data, 360° photo spheres are an essential component. A library

supporting such is Pannellum. Pannellum is a lightweight, free and open source panorama viewer for the web and uses WebGL and other web technologies for its rendering [Pannellum, 2018]. Pannellum is developed by Matthew Petroff, is distributed under a MIT License and is well documented with an active community support. There is an increasing amount of libraries for the visualization of 360° images, which might be due to the rise of the virtual reality technology, which gave us a selection to choose from. The deciding aspect that elevated Pannellum from other libraries is its support for so called hot spots, which are markers connected to a panoramic image and which can hold either textual information or hyperlinks to new panoramic images or other websites. This feature is used in the enhancement of information inside room images through for example work orders—more on this in chapter 9.



Figure 8.3: 360° image displayed using the Pannellum library, including two "Info" and one "Tour" hot spot [Pannellum, 2018].

8.3.4 Autodesk Forge

Autodesk Forge was introduced during AU 2015 and is a common data environment for uploading, storing and manipulating CAD models and other data of various file types [Autodesk Forge, 2018]. Forge encapsulates currently 8 different APIs covering functionality such as authentication, data management, design automation, model translation, reality capturing, viewer and more. The services of highest interest for our case are the *Model Derivative API*, which is responsible for holding model files and translating them into new formats. In most of the cases a translation is needed in order to make use of the second relevant API which is the *Viewer*. Autodesk's Viewer is WebGL-based and able to render a variety of model files but require them to be translated previously into the SVF format, which is rarely used in the industry. Viewer and Model Derivative API are working hand-in-hand and allow developers to make use of functionality such as extraction of meta data and powerful 3D viewer features such as section, measure and explode.

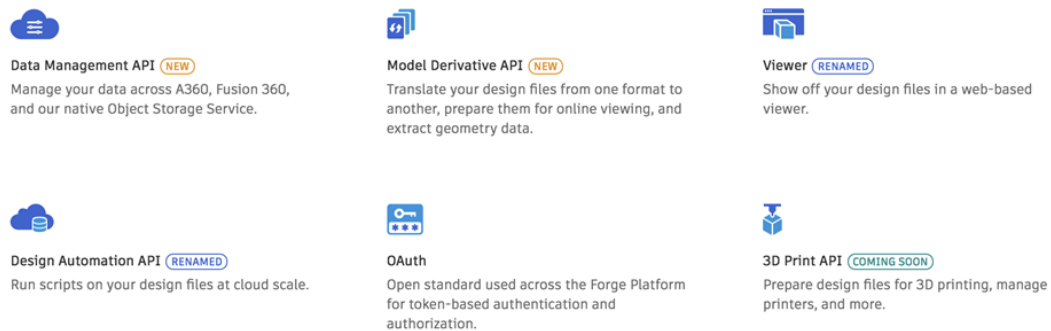


Figure 8.4: List of the available APIs offered by Autodesk Forge.

Forge is not a free service and pricing is handled via *Cloud Credits*. On sign-up, 100 of such credits are granted to each user for free and can be used for services in the cloud with different cost depending on the type of job; additional credits have to be purchased. As hinted before, we mainly use Forge for its ability to effectively render 3D models, which we use for displaying room specific sub-models of spaces and related systems opposed to complete building models—again more of this in chapter 9. Utilizing a paid and company specific solution is not ideal from a research perspective and additionally models can not be hosted customarily, which can be an issue depending on the contractual framework. Nevertheless, we chose Forge because of its compelling visualization capabilities, the absence of good open-source alternatives and the presence of good documentation and developer support. In the conclusion 10 we will reflect on alternative solutions.

Chapter 9

Prototypical Web Application

This chapter introduces the web application, which represents the second major part of the development process next to the Python-based pre-processing. We start by repeating the use case, while going into more detail about the involved user groups and supported scenarios. Further, hardware architecture and the rough sketch of the user interface are given before detailing the most important components and concepts further. Enhancements and possible extensions are discussed in the following conclusion.

9.1 Use Cases

Facility Management is only one of many domains shaping a facility after the completion of the construction, but in comparison, [FM](#) is highly involved with the building itself and the installed systems inside it (see [chapter 2](#)). As such it was ideal to fix the scope of the prototype and give it a tangible framing. In the following we will introduce the expected actors in such a setting and the scenarios we see them experiencing when using the web application.

9.1.1 User Groups

The user group targeted by the presented solution mainly covers the [FM](#) team, which usually includes the full range from general management all the way to repair engineers or cleaning personal. In contrast to complex [3D](#) models, [2D](#) floor plans are light-weight enough to be easily supported on mobile devices and simple enough to be used by any kind of end-user. This creates ideal conditions to open our service to the general occupants of a facility, integrating them into the feedback cycle and thus improving the communication with the [FM](#) team; a factor that has been a major point of concern so far.

9.1.2 Scenarios

Due to time concerns we decided to focus on a small set of use cases wrapped around work order management, which is a core part of any CMMS. In the following we introduce two possible use cases.

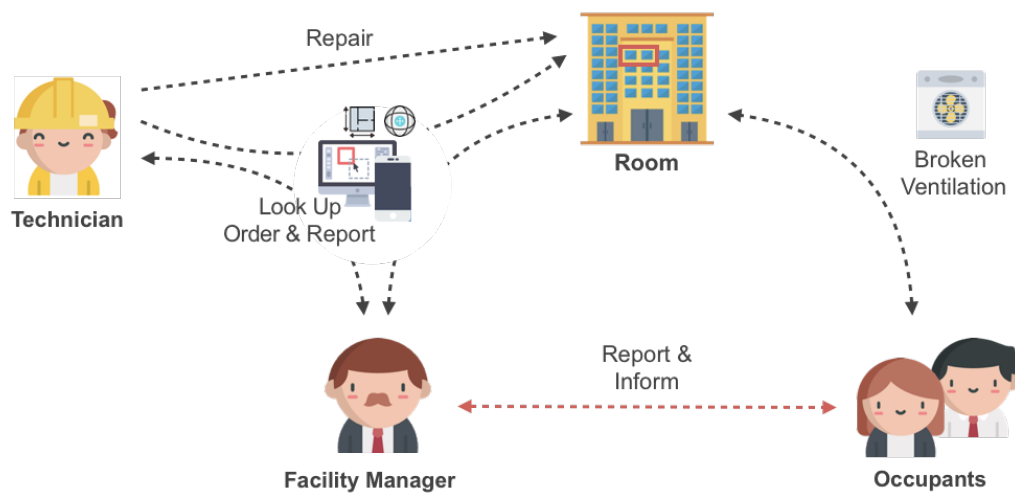


Figure 9.1: Scenario in a FM setting using the presented prototype.

Create Work Order

Seppo, manager of a FM team that maintains a larger portfolio of facilities, gets informed about a broken air supply terminal in room with number "103" in one of his supervised facilities. He opens the web application and navigates to the respective facility and level the room is located on. After selecting the room, he first takes a look at the 360° photo sphere to get an overview of the situation and to match the attached image with the terminals he can observe in the photo sphere. The terminal in question is found and Seppo navigates to the dialog for creating a work order. He is specifying a title, description, assign a repair engineer and further marks the terminal on the photo sphere.

Processing Work Order

On the receiving end, Ville the repair engineer spots the work order in the list of pending tasks, he opens it and reads the description. The work order describes the issue well and in order to get a better idea of the situation on-the-spot, Ville opens the attached link to the photo sphere. He identifies the terminal on first glance, grabs the fitting equipment and ladder he needs to reach the terminal. Ville has been working on the terminal for some time but notices that something is off—the problem is actually not with the terminal itself but somewhere in the upstream system. He gets out his mobile phone, opens the floor plan and

fades in the ventilation system: the duct is running straight to the opposing wall, does a sharp turn and disappears in a tangle of overlapping ducts. Luckily the web app provides a feature to highlight the supplying air system of the particular room.

Ville now has a clear image of the route of the ducts up to the supplying air unit. The air supply is working for the rest of the floor and was just inspected two days ago, but there is a fan unit among the coil of pipes and ducts; this one could be responsible for the failure. However hidden behind a suspended ceiling and in between a jungle of pipes it is difficult to identify the exact location. Ville opens the 3D model of the room, which includes the room itself and only the systems servicing that particular room. With that Ville is able to understand the exact position of the fan unit and sets out to check it out (see figure 9.1).

9.2 Architecture

The architecture of our service rests on three main components: graph data base, custom API and the dashboard itself; all depicted in figure 9.2.

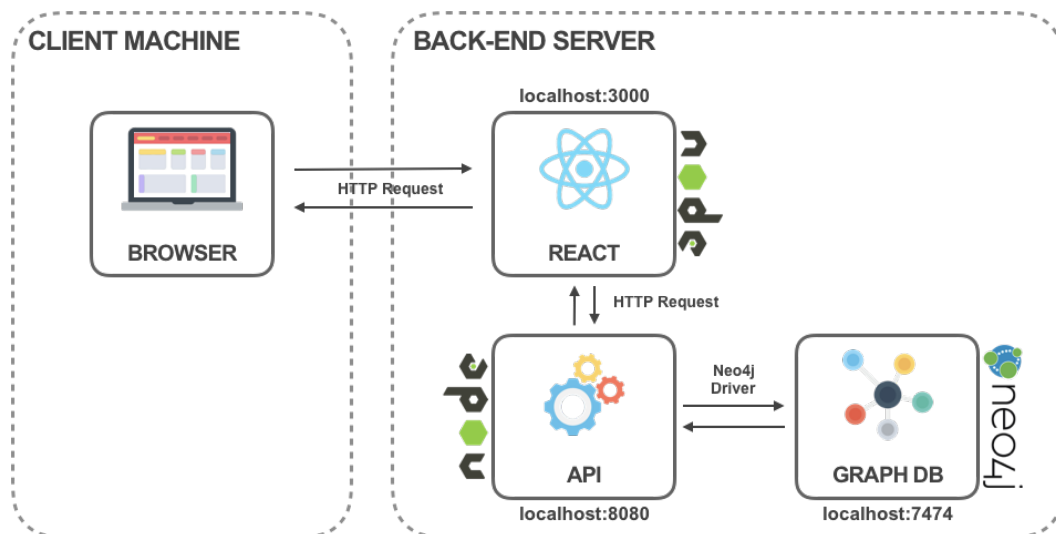


Figure 9.2: Architecture of the presented software prototype.

As stated before Neo4j is the graph DBMS applied during the development of the prototype and is hosted on our server. It is possible to interact with the underlying graph directly through the browser that Neo4j ships with. However, in our solution stack we access, manipulate and extend it through the offered HTTP endpoints. In theory it is possible to execute queries directly from the client to Neo4j but instead of doing that an intermediate API is put in between. This API offers several simple endpoints that trigger respective queries based on the given input. The advantage of such a method is the reduced code duplication and the middleware that can be wrapped around such endpoints—e.g. for authentication. The API is realized using Express, which is a framework built on top of Node.js and is specifically

designed for the design of server-side of web applications and [APIs](#). Additionally, requests made through our web application directly are send first to the local server, React is running on, and are proxied to the [API](#) afterwards.

/	GET	/api/building/:buildingId/storey/:level/space/	GET
/api/forge/	GET	/api/building/:buildingId/storey/:level/space/spaceld/	GET
/api/forge/:token/	GET	/api/building/:buildingId/storey/:level/space/:spaceld/photospheres/	GET
/api/building/	GET	/api/building/:buildingId/storey/:level/space/:spaceld/photospheres/:photold/hotspot/	GET
/api/building/:buildingId/	GET	/api/building/:buildingId/storey/:level/space/:spaceld/photospheres/:photold/hotspot/	POST
/api/building/:buildingId/storey/	GET	/api/building/:buildingId/storey/:level/space/:spaceld/workorder/	GET
/api/building/:buildingId/storey/:level/	GET	/api/building/:buildingId/storey/:level/space/:spaceld/workorder/workorderId/hotspot/	GET
/api/building/:buildingId/storey/:level/svg/	GET	/api/building/:buildingId/storey/:level/space/:spaceld/workorder/	POST
/api/building/:buildingId/storey/:level/systems/	GET	/api/building/:buildingId/storey/:level/space/:spaceld/systems/type/subsystems/:subtype/	GET
/api/building/:buildingId/storey/:level/systems/:type/svg	GET		

Figure 9.3: Endpoints provided by the [API](#) of the developed web service.

Figure 9.3 shows the available endpoints. Initially endpoints were supposed to comply the RESTful standard, which is a widespread architectural style for designing web services and supports interoperability between systems on the internet. Defining a RESTful interface requires the development team to define a well defined scheme in order to provide proper endpoints. This conflicts with the graph mentality, which is characterized by flexibility, complex and always differing queries. As an answer to this problematic, Facebook developed GraphQL, which provides a more powerful and flexible way of designing web services, by structuring it based on data flows. GraphQL was only discovered at a rather late stage of the overall development process and thus not implemented inside the prototype. Further exploration into GraphQL is highly recommended for future iterations (see conclusion 10; as for our web service, we tried to comply to REST as best as possible.

9.3 User Interface

Earlier we introduced React as the underlying framework of our web application—front-end side of things—and while the overall looks are generally independent of the technology in the use, it had strong implications on the implementation process. In traditional web services, it is common practise to load third party [CSS](#) files, when using libraries and to append those by own, custom made ones. A common problem encountered in this sort of scenario, are conflicts due to identical style selector—in some cases intended in order to overrides default behavior but more often than not unintended by the developer. React lives the concept of distinct, independent components and has little use for global [CSS](#) files. Instead styles are

commonly defined directly in JavaScript, which is an evident solution considering that React already brings [HTML](#) into the the JavaScript environment through JavaScript XML ([JSX](#)). In our prototype we choose another practise, called CSS Modules.

CSS Modules retains the concept of separation of styling and underlying JavaScript but circumvents the problem of name conflicts by utilizing the build process when using transpilers. Transpilers are used commonly throughout the web for various reasons, for example to provide wide spread browser support while simultaneously using the newest JavaScript syntax: converting the code written in the new syntax into the equivalent code written in the old one. As a matter of fact React uses transpilers as well in order to generate proper web page structure from the mix between [HTML](#) and JavaScript defined through [JSX](#). CSS Modules hooks onto this transpiling process to generate scopes for each [CSS](#) file per component, thus allowing to style components individually without considering possible naming conflicts.

We use the local [CSS](#) files to define our own styles but also to override styles coming from outside frameworks. One of such is Semantic UI, a component library that simplifies the process of creating beautiful and responsive layout by providing developers with ready made components. There are many of such libraries available, but Semantic UI was mainly chooses because of its clear and simple naming convention, which allows to get started quickly. [Figure 9.4](#) shows the complete interface of the final prototype and in the following we will detail the individual elements.

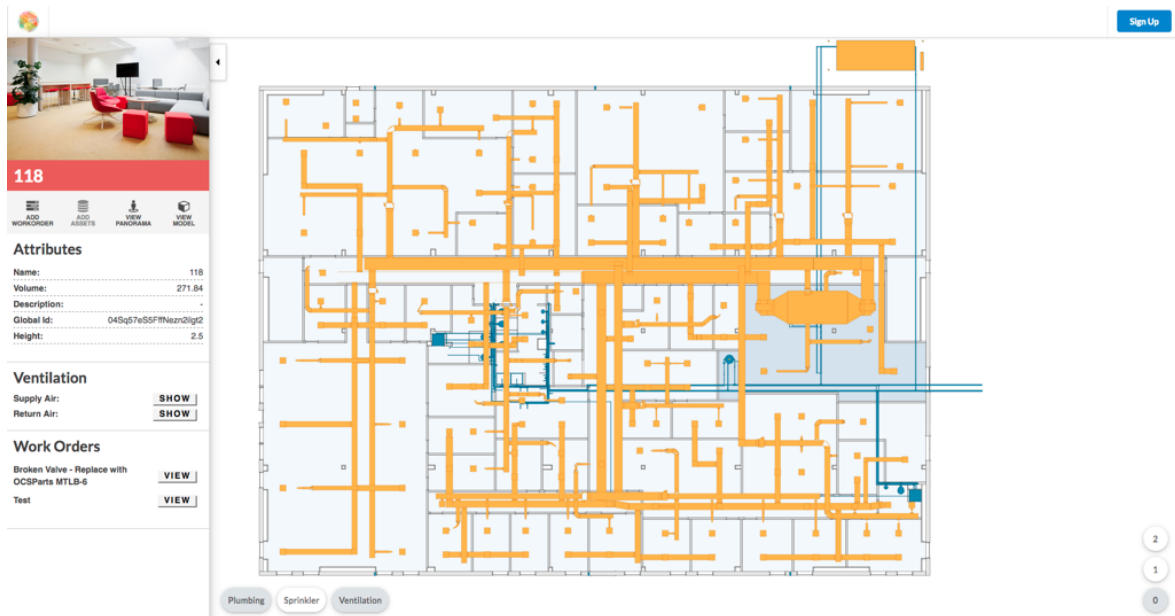


Figure 9.4: Complete user interface of the presented prototype.

9.3.1 Navigation Bar

Navigation bars are a crucial part of almost any web design and are usually displayed on any page of a service. As such they hold essential information and links to most important parts of the web service, such as links to profile details or to the landing page. During development, the navigation bar was only given slight focus. In its current form it only holds the logo of the web service and a mock-up for sign up or logout button, depending on the context (see figure 9.5). In future iteration the navigation bar would be a suitable space to hold information about the user, means to select between different facilities and details on the currently selected one. Using React it is also imaginable to put varying content depending on the context of the current route.

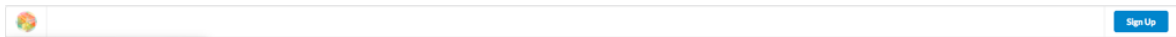


Figure 9.5: The navigation bar.

9.3.2 Viewport

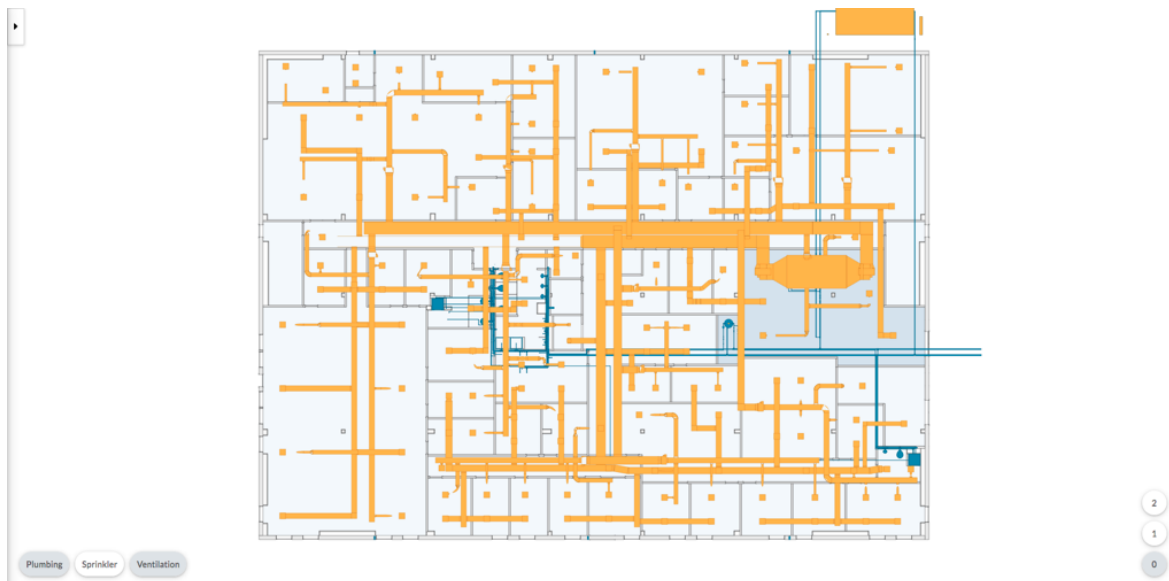


Figure 9.6: The viewport.

The viewport is the heart of our web application and mainly holds the [SVG](#)-based floor plan. Depending on the selected facility and the existing number of floors, respective *floor button* are generated in the lower right corner. The styling is mainly derived from Semnatic UI and the information on the floor is retrieved through HTTP request to the underlying [API](#). In a similar fashion the buttons on the different system types are generated when selecting a floor and will only be drawn if those systems exist on the particular floor. The [SVG](#) itself is dynamically loaded and put centric into the viewport with hover and click events attached

to the components representing spaces. Viewport and the mentioned button groups can be seen below.

9.3.3 Sidebar

Mobile friendly layouts commonly implement sidebars to present elements and links of the navigation bar on smaller screens. However, in our application the use is a different one. The sidebar element is used to display important information about selected spaces and links to advanced features of the service (see figure 9.7). As for now, this is applied for the selection of spaces in our prototype, but future iterations could use the sidebar for more detailed information about work orders and more.

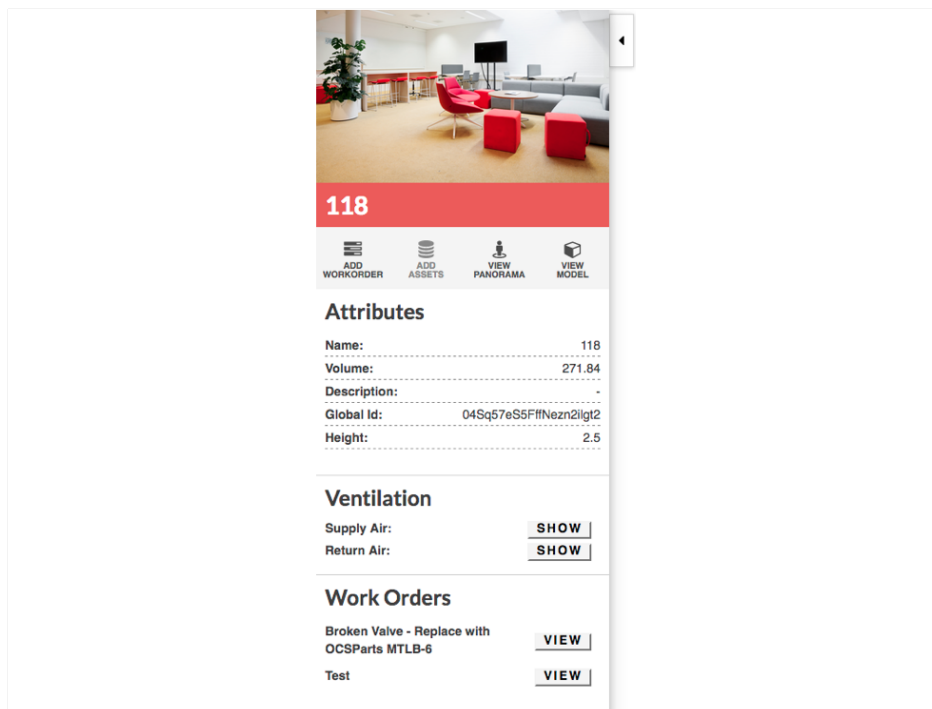


Figure 9.7: The sidebar.

9.3.4 Modal

The most versatile UI element in our service is the modal, which is a popular element of web design. Basically, modals are windows that are put in front of the current main view of the application, which stays visible in the background but is disabled for the time being. Modals go hand-in-hand with so called backdrops, which span the whole screen size, are subordinate to the active modal and attached with a click event that will close said modal and re-activate the main view. Chapter 5 introduced the concept of single-page application—to which our

prototype is counted to—which make common use of modals. Figure 9.8 shows the use of a modal for the display of a 360° photo sphere.

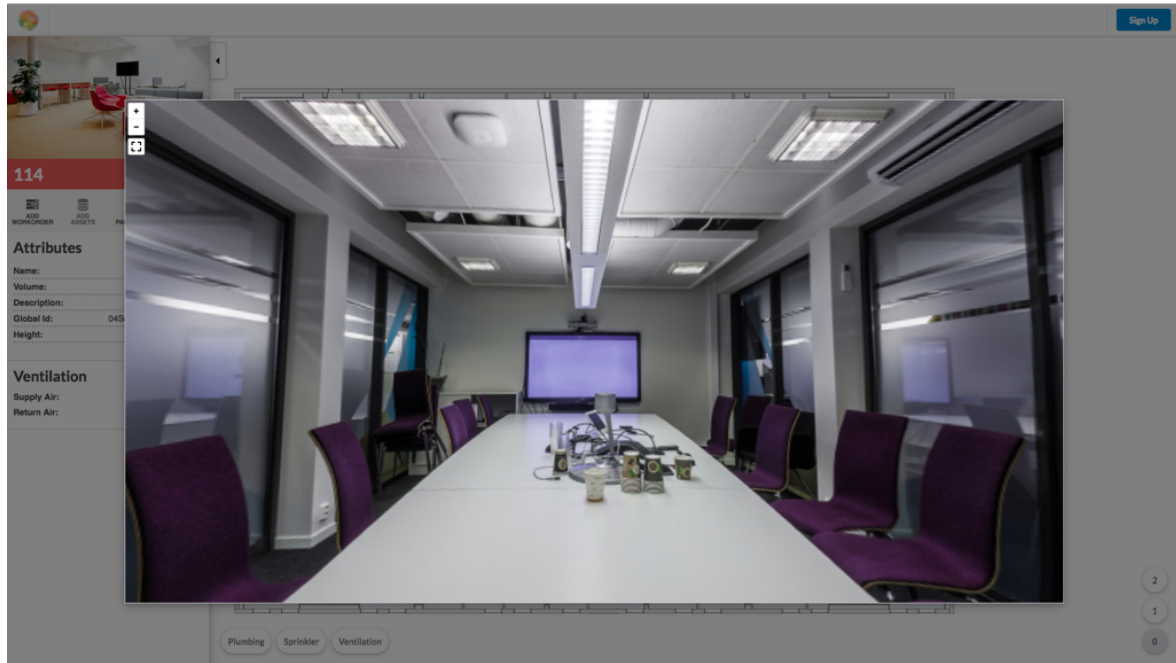


Figure 9.8: Modal showing the photo sphere of an exemplary room.

9.4 Interactive Floor Plan

Interactive floor plans are the center piece of our explorational prototype and are included in the viewport element. As outlined before, a major shortcoming of current CMMS is the absence of any spatial relation, although data items predominantly are connected to a location inside a facility. Similar to services like Airbnb, we are able to connect such data items to spaces on the floor plan while offering a visualization that is light-weight, mobile friendly and easy to understand and navigate. 3D solutions are integrated in a gradual fashion through photo spheres and partial models of rooms, which will be detailed later on. In the final version of the prototype, floor plans consists of the base layer including walls, spaces, doors and windows and the system layer, that allows to fade in the layout of the various pipes and ducts; both will be detailed in the following.

9.4.1 Base Floor Plan

The base floor plan depicts spaces, walls, doors and windows and is loaded through an HTTP call to API of the prototype making the related query. The received information contains the coordinate of each individual IFC element describing its outline, *IfcType* and *GUID*. Using

the React philosophy of component-based splitting, the different elements and coordinates are assembled into an overall **SVG** component. Styles are given through an intermediate **SvgLayer** component, which is based on *IfcTypes* (see figure 9.9).

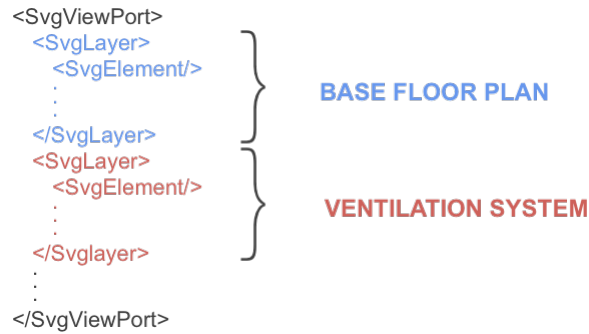


Figure 9.9: React structure showing the split into **SvgLayer**.

Additionally, we are checking for elements of the *IfcSpace* type and attach hover and click events to them. Rooms are important pieces in the daily work of the **FM** team, usually tasks, assets or other things are connect to them in one way or the other. In chapter 6 we introduced some design principles that improve the readability of maps. Due to time concerns only very few of those have been applied in case of the developed prototype, future extensions could improve the general representation in that matter. Also there are some interesting development in the field of indoor navigation, who would open up the potential for many more interesting extensions, on which we will have a closer look in the conclusion 10.

9.4.2 MEP Systems

The second important component of the floor plan is the system layer. Systems are retrieved through the **IFC**, stored inside the graph database and queried using the intermediate **API**. As stated before there was a particular problem with the *IfcSystem* definition in the sample **IFC** used for the prototype: although being offered directly by **NIBS**, the model file did not include any system definition and a work-around had to be found (see chapter 7). In any case, if present on the particular floor, ventilation, plumbing and sprinkler can be faded-in or out using the respective button group in the bottom-left corner. This helps users to understand the course of the various pipes and ducts throughout the floor. Similar to the base floor plan, systems are given different styles based on their assignment to a particular **SvgLayer** group. An important aspect of React that helps to keep performance stable is the underlying checks done on the virtual **DOM**, which prevent unnecessary re-renderings of the base or other unaffected systems layers.

9.5 Building Space Selection

A second aspect, crucial to our web application is the selection of rooms/spaces, making the floor plans interactive to begin with. Naturally a user can select rooms through the floor plan, which opens the sidebar menu in order to display information about the room. Also much of the gradual information retrieval—successively increasing in complexity—is directly tied to rooms. In the following we will first provide a rough overview of the the room focused view. Afterwards we highlight advanced features of displaying parts of a system involved in serving the selected room, its photo spheres and partial 3D model.

9.5.1 Overview

Figure 9.10 shows an example of a sidebar menu displaying information about a room in the final version of the prototype. The very top holds an image of the space, which is retrieved from the graph-database. In a future running product, such images could either be uploaded by regular users or set by users with admin rights. A bar for the most important controls is just situated below the image; most of the features presented in it, are covered shortly after.

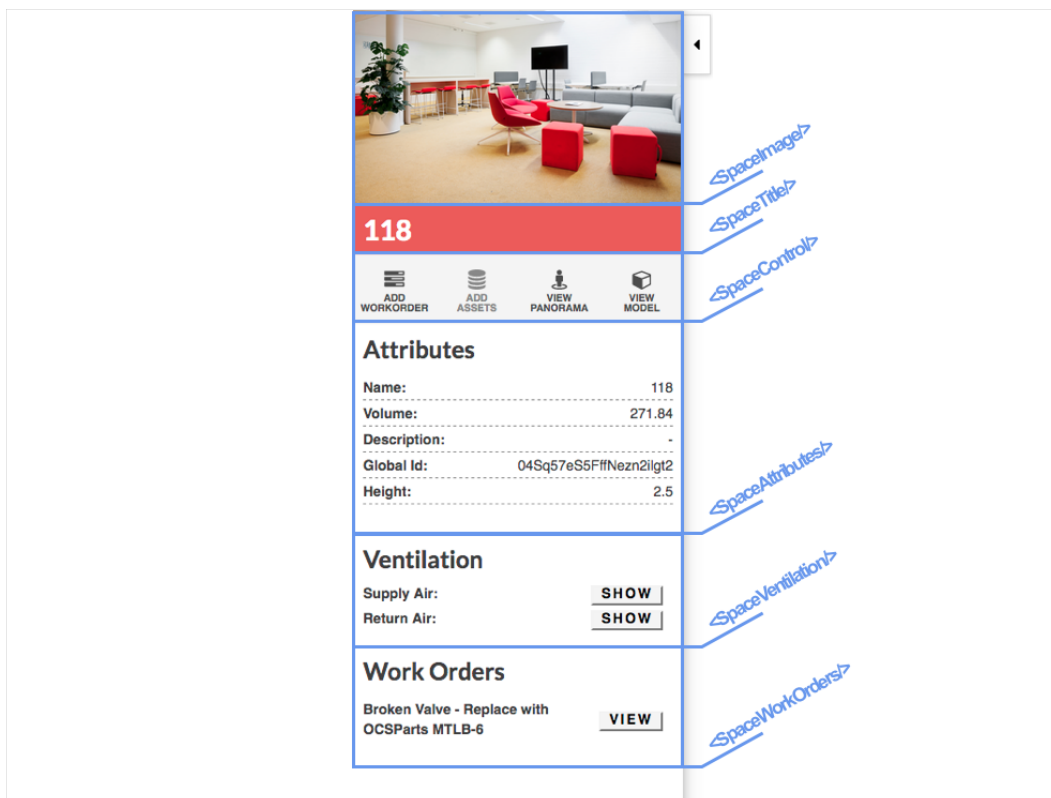


Figure 9.10: Sidebar displaying information about a selected room.

Next, several attributes, either retrieved from IFC or calculated during preprocessing, are listed in what is called *Attributes* section, followed by sections for ventilation and work orders.

9.5.2 Serving Systems

When maintaining air terminals, repair engineers are commonly interested where the air is either coming from or going to. As described in section 7.3.5 the relationships describing the manner of how flow distributes through the network was not given by the original sample IFC file, but was possible to be established using a work-around of property sets and bounding boxes and was written into the graph database. Retrieving the parts of a system supplying or taking returning air is relatively simply retrieved by using Cypher to query the graph database (see section 7.4.3). In our tool, this feature is represented inside the space focused view in the sidebar under the *Ventilation* headline, for both supply and return air. By clicking the respective buttons the queries are send first to the underlying API and then to the graph to retrieve the GUIDs of all elements involved in the function. Those elements are specially marked and during the re-rendering of the viewport left out while other elements of the system are made transparent, which can be seen in figure 9.11. Thus, the user can better

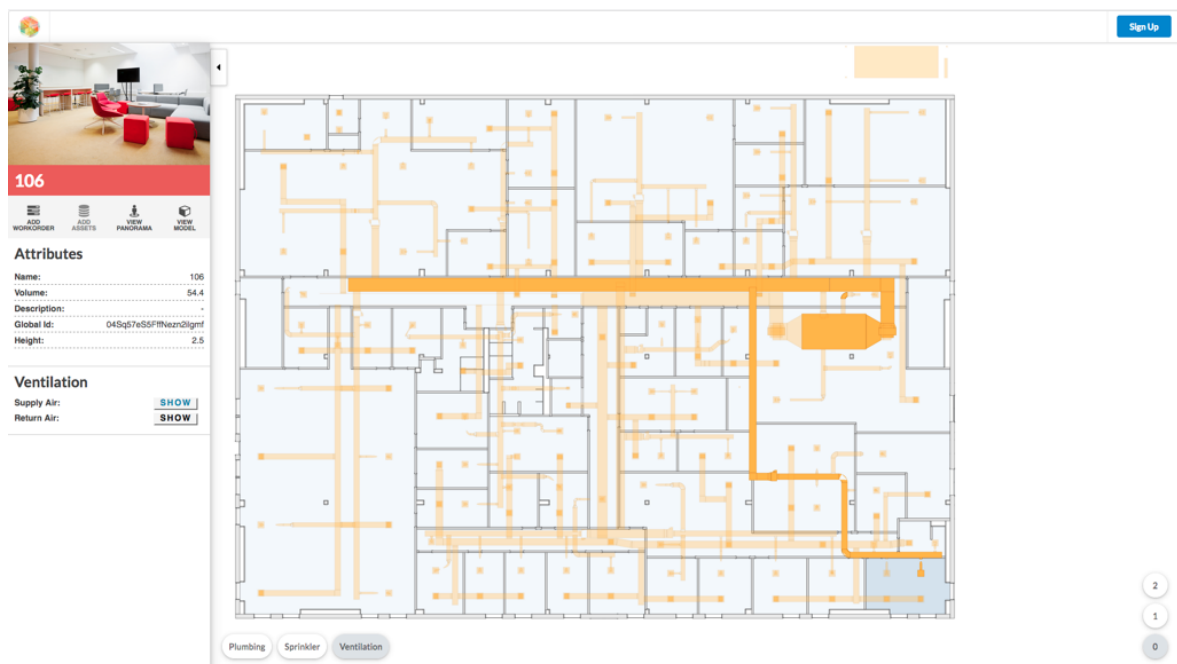


Figure 9.11: Supplying air system for room 106.

understand, which parts of the systems are involved in the particular function, where the source of the air flow lies and which parts of the way could be causes of the faulty behaviour.

9.5.3 Photo Sphere

One of the drawbacks of using 2D projections in the form of floor plans is the increased abstraction level and the loss of 3D related tasks. A very effective and light-weight way of establishing such an insight is given by photo spheres. Photo spheres are created by taking multiple images in all directions—including up and down—and without changing the initial location. Using a piece of software, the images are combined through key points into a single spherical image. Depending on the resolution, those images are only a couple of megabytes large, are easy to store and quickly loaded. In order to make them properly accessible for users they have to be loaded into special viewer, supporting basic interactions such as rotation, panning and zooming. As presented in section 8.3.3 we applied the Pannellum library for that, which supports all of those feature. On top of that, it also allows the placement of so called "Hot Spots", which can be used as part of a virtual tour or to hold further information. We are using them to provide a marker and links for active work orders, which we will detail in the following sections.

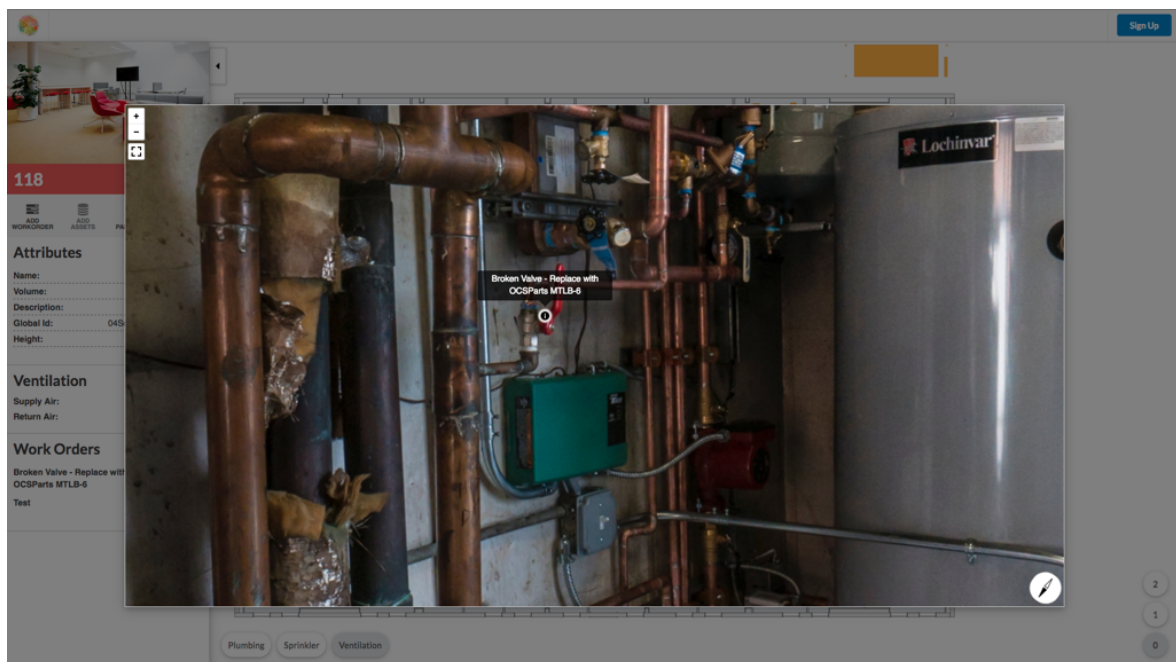


Figure 9.12: Photo sphere including marker of a active work order.

Members of the FM team or other users can use the photo spheres to explore remote rooms, understand the local situation and visually identify objects and systems. In the case of our prototype, photo spheres are stored directly on the file system with the graph database holding the respective file name inside a separate "Photo Sphere" node connected to the room it is connected to. Accessing those photo spheres through the browser is one way; another way that includes some interesting interactions is through mobile devices, which can make use of the built in gyroscope to change the viewing direction in a more engaging way—

even applications using virtual reality are possible. Recent advances in virtual reality has also increased the offer of products to create such photo spheres, which are getting more and more common. In a possible production setting, photo spheres could be either created centralized by the **FM** team and regularly updated or additionally crowd-sourced through low level end users, such as occupants. All in all, photo spheres offer much potential—especially so in the verge of advancing Virtual Reality (**VR**) technology—, are light-weight in terms of load and easy to use.

9.5.4 3D Space Model

We introduced photo sphere as one tool to compensate for the loss of **3D** information and while effective in use, it comes with certain drawbacks: firstly, apart from the attached hot spots the photo spheres are not interactive and elements inside it are not related to any underlying data. Secondly, elements such as piping or ducts, which are located behind walls or a suspended ceiling, are not visual inside the photo spheres. Using those to get information about piping is little helpful. In cases such as these, **3D** models hold information of much larger interest, but as mentioned before, they can be problematic just through their sheer size of the overall files and the fact that one often need access to models form multiple disciplines, e.g. architectural and mechanical. At best we would like to have the capabilities of **3D** models but without loading much unnecessary information and instead focus on the things pivotal to the current task.

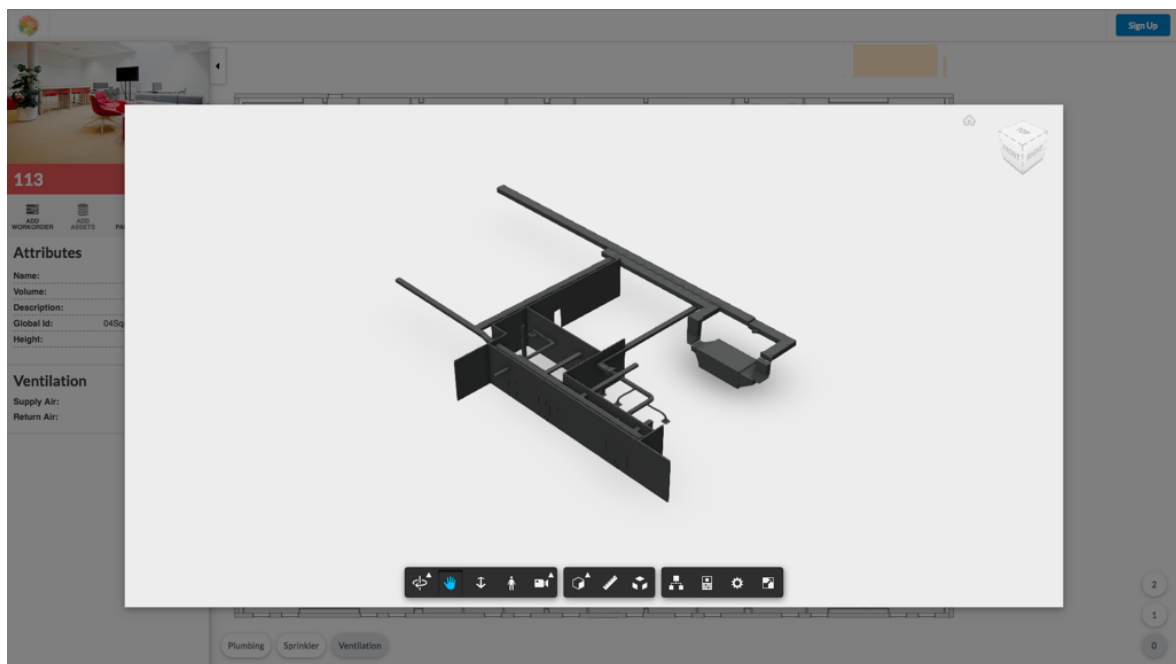


Figure 9.13: Room-based sub-model of room 113, including walls and the servicing ventilation system.

We solved this issue by splitting the complete building model into smaller sub-models based on rooms. Each of these small room-based models includes the space itself, the adjacent walls, windows and doors and finally all ducts or piping that is connected through a "serving" relationship. These sub-models are created during the python-based preprocessing and uploaded to Autodesk Forge cloud, where it is transpiled into a viewable format and retrievable through a key that is stored inside the graph. The resulting sub-models are considerable smaller in size and instead of including only information from a single discipline, they hold information from many different models but all wrapped around the same context, which is the selected room.

Afterwards, [FM](#) members are able to directly follow and perceive the layout of the various pipes that are hidden behind the wall. This feature has been implemented to the prototype as a prove of concept but there are still much possible improvements to make. One issue observed is the fact, that *IfcWall* elements inside the [IFC](#) are not aligned with the spaces they are neighbouring; one wall element can stretch over many rooms, which results in odd looking results. Another concern lies with the nature of the sub-models. PythonOCC, the geometric framework used during the pre-processing, supports the export of STP files by default. While the STP format is capable of storing all kinds of semantic information—it is the ancestor of [IFC](#) after all—we would be interested to only store the respective [GUIDs](#) alongside. Similar to the [2D](#) floor plans, such an approach would allow us to later on, in the Forge powered viewer, to retrieve semantic data of particular elements in the background or manipulate them according to their identifiers. Currently, the prototype does not support such features but would surly benefit through that. We will deepen this discussion and address further points in the following [conclusion 10](#). However just from a conceptional point, breaking the holistic models down in smaller, more dynamic pieces, is very powerful.

9.6 Work Order

Work orders are a corner stone of the [FM](#) business, along side assets, purchases and others. In our prototype, work orders and their surrounding use cases were chosen for its prominence and as one possible example. In the following we will touch on the features for creating them and the linkage to other components inside the service.

9.6.1 Creation

Work orders can be created through the control bar inside the room focused sidebar menu. When clicking on the respective button, the user is redirected to a new dialog as seen in the figure below [9.14](#).

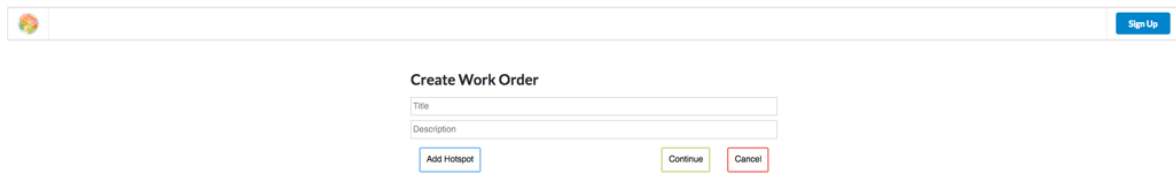


Figure 9.14: Dialog for creating a work order.

The dialog requests currently only a title and a description but many more concepts such as priority, schedule, assigned persons, etc could be added. Additionally, the user is able to add a hot spot to the corresponding photo sphere of the selected room. Upon clicking, a modal displaying the said photo sphere is opened and the user can click the respective piece of equipment or the location of interest in order to store it to the graph database. Later on, when the photo sphere of that room is loaded again, all hot spots connected to it are automatically attached to the scene.

9.6.2 Linkage

As mentioned in the section before, one way of linking work orders to spatial context is done through using hot spots in photo sphere scenes. Additionally, work orders active on a room are displayed under an additional section inside the sidebar menu and clicking on them will automatically open the respective photo sphere with the camera pointing directly to the location in question. Due to time constraints these are the only linkages installed at the moment. Future iterations could install marker on the floor plan, which could show more accurate where work orders are located in a specific room. Also some work orders might be viewed preferably through the [3D](#) model viewer, in which case the introduction in such an environment makes sense as well. Generally, there is much more logic that could be implemented around the concept of work orders; much of this logic however is already well established in existing [CMMS](#). The innovative aspect of our approach—and thus the focus of the implementation here—is the connection to a spatial context, which is missing in nearly all other systems on the market.

Chapter 10

Conclusion

In the chapters leading to this point, we introduced the field of **FM** and its challenges, how **BIM** can bring tools to the table to solve these and how current attempts have failed, due to failing to understand **FM** requirements. We introduced our own concept of putting interactive floor plans at the center of a web application, retrieving data from a graph database and granular accessing information of increasing complexity. In order to proof that the proposed concept is feasible, a software prototype was created consisting of a pre-processor step, translating the **IFC** file and a web application, serving the data and interacting with the end user.

In our conclusion we will shortly summarize the goals of this thesis and the things that were achieved. Afterwards we will highlight some of the challenges that were encountered during the wide-ranging development process. Finally we discuss future improvements that can be made to the presented web application and the python-based preprocessing before ending with an outlook on future trends we expect to persist.

10.1 Summary

The domain of Facility Management is extremely diverse and incorporates many different aspects. As such, its core business has always been a junction point of various data streams; a trend that increasingly turns to a core challenge of the whole field. On the other hand in construction, **BIM** has been leading in answering such issues, like interoperability and scattered data. A number of researchers have studied a potential application of **BIM** for **FM** and most of them support its positive effects and call for action; however they also warn against the many issues of such an integration. Answering those issues remains a serious challenge for both construction and **FM** representatives. Still, some innovative researchers

and companies have started developing tools for bringing **BIM** into the **FM** domain, but the desired demand not picked up so far.

We argue that the reason for that lies with the fact that those solution fell short to address a core problem—also pointed out by Kiviniemi and Codinhoto [2014]—, which is the general misunderstanding of **FM** requirements by the construction industry. Current **BIM**-enabled **FM** solutions almost exclusively employ **3D** models as core elements of their interfaces, which is typical for software tools in the construction industry. This stands as a heavy contrast to classical **CMMS**, which are predominately data table based and often lack any spatial visualizations—it is hardly surprising that **FM** professionals are alienated by such an approach. On top of that we see real problems in terms of the usability of **3D** dominant solutions: models in the construction industry are often huge, need a long time to load, are hard to navigate and have poor support on mobile devices.

We propose a new concept, which puts interactive **2D** floor plans at the center of a easy to use and access web application. Floor plans are directly extracted from the **IFC** and both geometry and semantics are stored inside a graph-based **DBMS**, liberating data from the **IFC** file and making it easier to access. Additionally, the graph layout of the chosen storage not only fits the natural structure of the **IFC** but also supports many of the common use cases in **FM**, which deal with finding routes in buildings or the flow of air or water. Using the Python language, third-party libraries and the official **FM**-handover **MVD**, we were able to show that it is possible to execute such a data extraction. However, during the implementation we were faced with many issues regarding the quality and extend of **FM** relevant data inside the **IFC** (see section 10.2).

On top of the resulting graph database we built a web application using current technologies like Node.js and React. Using React, floor plans can be displayed and updated without much impact, systems can be dynamically faded in or out and rooms are selectable. Another important aspect of the introduced concept is the granular access of data of—in parts—increasingly complexity. Users of the service are able to either open panoramic images of rooms in order to get a simple three-dimensional understanding of the space or a more complex **3D** model, which is extracted from the cumulative **IFC** model beforehand. All in all we were able to proof that the anticipated concept is realizable. The prototype only represents a minimal proof of concept and can be improved in many aspects. More critical challenges are connected mainly with the **IFC** and other fundamental concepts and will be discussed in greater detail in the following.

10.2 Challenges

There have been a number of challenges during the development of the prototype, some conceptually and some from an implementation point of view. One of the most influential and most impactful for the the whole process, has been connected with our work with the **IFC** file format—an issue in two layers.

First and foremost we have found it to be difficult to find openly accessible models; a situation that seems symptomatic for the whole industry. The construction industry is strongly characterized by its project-based nature, which promotes the creation of knowledge silos. This is alright when operating in the the context of a project, but this context promptly ends with the completion and the construction and the handover to the new owner. **O&M** is determined by a new set of rules and holds a diverse set of people: who should be allowed access to the models? and to what extends? To this day this question stays mainly unanswered and getting access to models and the data is difficult—not only from a legal but also from a technological point of view. It is high time for academia to do the first step and create an extensive set of open source models, in order to promote more research and new advances.

The second aspect—connected to some extend to the first one—is the unsatisfactorily quality of **IFC** models. In their publication in 2012, Laakso et al. [2012] give valuable cues for the low quality of **IFC** files. One part of their argumentation targets the exporters of large to small software vendors, which have been notoriously badly implemented; enabled by the low standards set by buildingSMART. Additionally, the **IFC** data model has been very structuralistic from the start, trying to include as many concepts as possible. This gave rise to a data model in which only a small part of the possible supported functions are actually used and also to the situation that the same function can be implemented in multiple ways.

In order to answer this short coming **MVD** were introduced, in order to make **IFC** more predictable. Theoretically this should bring an end to the existing problematic but the truth is that many **IFC** files either don't implement certain relationships at all or have them implemented in strange ways. This is especially true for the **FM**-handover **MVD**, which is already quite outdated on its official website on buildingSMART (most recent version from 2009). Also, **NIBS**, one of the very few open-source repositories for **IFC** files, offer models files not implementing any **FM** handover **MVD**, although use for the **O&M** is clearly implied by offering **COBie** files. On top of that, the **IFC** models are again outdated and in partly faulty implemented. Due to the absence of better alternative, we choose the "Office Building" project as a basis for our prototype but had to apply numerous workarounds, as described in the other chapters.

Another challenge occurred when bringing the **IFC** structure into a graph-based structure. Just due to the limited scope, we decided to make the **FM** handover **MVD** the basis of the file-

graph transition process. Also, in order to improve performance in terms of working with the graph, we skipped relationship nodes—as supported in the **IFC** model—and we introduced more implicit relationship names (see chapter 7). A more sensible approach could be to bring the **IFC** data model as natively as possible into a graph structure, which, on the other hand, would reduce the performance in which the graph can be queried. More problematic even is that the **IFC** model is very complex and hard to understand by a typical user base; this conflicts with the concept of graph **DBMSs**, which encourage discovery by ordinary users. Further even, there are questions about how such a database could be extended with information and concepts not found in the **IFC**. There is a lot of complexity hidden in finding a realistic and resistant way of performing such a mapping.

A last issue, which we encountered during the development of our prototype, is connected in the split of models into useful sub-parts. The division in disciplines is a remnant from the way construction has been conducted so far. Future development also indicated a move to a more modular way of building, which breaks the concept of disciplines and replace it with a modular approach. Similarly, **FM** and other domains during **O&M** are seldomly interested in disciplines but instead care about context and/or location bound information. The current structure of model files don't support such use cases. Hence, we applied to a new split into sub-models by rooms and including bounding walls, doors, windows and the systems serving it. This approach arguably fairs better than loading several large models of different disciplines at the same time, but at the same time it still obliges a certain logic onto the user: what if we want to see a certain room, its serving ventilation system and finally the rest of the ventilation system to understand which other rooms are affected. Such an approach requires a structure that is modular to its core, transcending the concept model information just by spaces. Future research should aspire a concept in which each element is attached with a singular **3D** visualization, that could then be loaded individually.

10.3 Future Improvements

The main objectives of the presented prototype has been to make the information inside the **IFC** accessible, to give spatial context to common data items of the **FM** domain and to support a gradual access of increasing complexity. We were able to show and thereby prove the possibility of the implementation of all of these aspects, through a minimal viable prototype. As such there are of course various things that can be improved. In the following we will focus on important concepts that would enhance the overall software product while, for now, blocking out trivial improvements related to general software engineering.

10.3.1 Improved Graph Generation

As just mentioned in the section before, the conception of the bringing [IFC](#) into a graph layout requires greater detail and more work in general. This also touches on the fact how to either add external data coming from other sources or—even preferred—how to connect to external sources, serving such data, which stays unanswered so far. A problematic covered on the other hand, is the inter-domain linking, which was mainly achieved by utilizing bounding boxes and checking for intersection. This approach will give wrong results in many cases and a first and rather simple improvement would be the switch to oriented bounding boxes, which represent the actual placement of the respective elements more realistic.

Another core method applied during the python-based pre-processing has been the creation of floor plans—not traditionally around the one meter mark but by rather projection all elements into a two dimensional plane. As described before, we applied a central cut through the individual elements, which produced good results for our sample file, but which is admittedly flawed (see section [6.3](#)). In the future a more advanced algorithm should be explored in order to provide a true projection of the various [3D](#) elements onto the [2D](#) floor plan. Lastly, so far we treated the [IFC](#) as only source of information; other formats such as [COBie](#) and even native model files should be explored in addition to that.

10.3.2 Improved Web Service

Moving back into the domain of graph [DBMS](#), Neo4j offers a feature for *spatial indexing*, which has not yet been utilized in our prototype. By storing coordinates in the custom point class of Neo4j instead of simple strings or numbers, queries on the graph can be made using spatial awareness, meaning nodes "know", which other nodes are close around them. Such a feature is helpful for routing and finding e.g. the closest restroom or fire exit. This also works for points in a geographic coordinate system, which would open our system to be embedded in map frameworks such as Leaflet or Google maps.

Other than that the service can be improved by providing more interactivity, not only by rooms but systems and other features; overall more common tasks by the [FM](#) team should be implemented. In chapter [6](#), we made a strong point about the usability of maps and best practises to increase their quality. Due to time constraints only a small portion of the best practises presented could be realized; future development should definitely catch up on this and implement them diligently.

On a last note, at the moment the floor plan takes an extremely dominant position in the overall service. Bringing a similar tool into actual production, a more subordinate role is likely to be applied, giving more focus on the data heavy common tasks of [FM](#) individuals and with the option of fading in the floor plan element as neededs.

10.4 Outlook

In the scope of this thesis we were able to proof the validity of the concept of moving away from 3D and instead gradually access information that is served via graph database, inside a FM setting. Future development will be needed to support more valuable tasks by the FM team and to create a stable release that is truly usable in an actual use cases. Fundamental concepts such as the split of model files into more atomic pieces and the ontology of presenting IFC and other data in a graph have to be finalized in a more detailed way. As soon as a larger maturity has been reached, it will be of at most importance to start involving FM professionals and to conduct usability tests. The maxim should be "the sooner the better", because such an involvement will truly show well working features, ones that are failing and features that might be outright missing to begin with.

On a positive note, the graph-based storage has revealed itself as a strong data management system, supporting many relevant use cases in the FM domain and beyond. Especially in the face of ever advancing development in the field of indoor position, a 2D floor plan approach, integrated in a map framework and using spatial indexing, opens new areas and application outside of just Facility Management. Still, FM offers interesting potential—especially when applied on the flow inside pipe networks—but the problems described in section 2.4 remain to be answers in a satisfactory way. This is accompanied by the problems encountered using the IFC file format, which is either not available in the first place or in such a bad state of implementation that it is hardly usable.

It is a common scheme inside the whole construction industry, which started to collect more and more data without taking serious efforts to store, maintain and ultimately use the data produced. The use of data in Facility Management and O&M in general, depends on the actors of the construction phase to store data in a way that is accessible for outside parties, tractable and most importantly up-to-date. This is a complex issue that is not only controlled by the means of doing so but also by contractual-entitlements and incentives of utilizing the data at a later stage at time. Current development in the UK will show if the value of data will be recognized and managed in such a way that it can be relevant for actors well after the end of the construction project.

List of Figures

2.1	11 core competences of Facility Management according to IFMA.	4
2.2	Project knowledge throughout the various construction project phases.	5
2.3	Old vs new: Interfaces of IBM Maximo as an established player on the left, next to Upkeep, a solution of rising popularity.	6
2.4	Potential BIM FM application areas as uncovered by the conducted studies (based on [Becerik-Gerber et al., 2011]).	7
2.5	Benefits and returns from BIM FM [Codinhoto and Kiviniemi, 2014]	8
3.1	COBie organization [Borrmann et al., 2015]	13
3.2	IBM Maximo showing functionality of its BIM extension.	14
4.1	Room-based models as an intersection of various domain-specific IFC mdoels.	18
5.1	Timeline of the IFC format, showing the three major phases [Laakso et al., 2012].	23
5.2	Layers of the IFC data model [Borrmann et al., 2015].	24
5.3	Extract of most important entities from the IFC hierarchy [Borrmann et al., 2015].	25
5.4	The principle of relations depicted as for the example of window-opening-wall [Borrmann et al., 2015].	26
5.5	A small social graph [Robinson et al., 2015]	27
5.6	Benchmark results of a "friends-of-friends" query in the limits of a small social graph [Vukotic et al., 2014].	28
5.7	Small SVG example [MDN, 2018]	31

5.8	Simple HTML markup next to the corresponding DOM tree.	32
5.9	Simple example of two different CSS rules.	33
5.10	Popular front-end frameworks for developing SPA.	34
6.1	Floor plans in three different shapes (from left to right): architectural drawing, fire escape plan, Louvre museum map	36
6.2	Floor plan of the undergraduate center of the Aalto University on the Otaniemi campus in two different versions [Aalto University, 2017].	40
6.3	The IFC data models separates between semantic and geometric information, enabling a dynamic linking [Borrmann et al., 2015].	42
7.1	Binding concept model of the FM Handover MVD [buildingSMART, 2009].	49
7.2	Intermediate graph model including IFC entities and relations.	50
7.3	Graph model including IFC entities, relations and a preliminary FM concept of work orders.	51
7.4	IFC types extracted from the "office" IFC files, provided through open access by NIBS.	52
7.5	IFC property storage scheme for both object and type related sets (based on [buildingSMART, 2006])	52
7.6	Excerpt of the attribute configuration file.	53
7.7	Excerpt of the relationship configuration file as applied during the pre-processing.	53
7.8	Intersecting bounding boxes of several <i>IfcFlowTerminal</i> elements with surrounding <i>IfcSpace</i> (visualized in pythonOCC 3D viewer).	54
7.9	Updated graph model including the concept of systems having sources, serving spaces and consisting of other sub-systems.	55
7.10	Brick relationship model [Brick, 2017].	57
7.11	Final graph model including flow relations between the various MEP components.	58
7.12	Interface of the Neo4j Browser with example query and result visualization.	59
7.13	Retrieved space nodes for the ground floor of the trial IFC file.	59
7.14	Spaces of the ground level and connected via door or opening elements.	60

7.15	Part of the ventilation system that is supplying air to room 104 (red) from the source in purple.	61
8.1	Ascii-art representation of a simple relationship.	66
8.2	Example code showing JSX, bringing HTML-like markup into JavaScript. . .	69
8.3	360° image displayed using the Pannellum library, including two "Info" and one "Tour" hot spot [Pannellum, 2018].	70
8.4	List of the available APIs offered by Autodesk Forge.	71
9.1	Scenario in a FM setting using the presented prototype.	73
9.2	Architecture of the presented software prototype.	74
9.3	Endpoints provided by the API of the developed web service.	75
9.4	Complete user interface of the presented prototype.	76
9.5	The navigation bar.	77
9.6	The viewport.	77
9.7	The sidebar.	78
9.8	Modal showing the photo sphere of an exemplary room.	79
9.9	React structure showing the split into <code>SvgLayer</code>	80
9.10	Sidebar displaying information about a selected room.	81
9.11	Supplying air system for room 106.	82
9.12	Photo sphere including marker of a active work order.	83
9.13	Room-based sub-model of room 113, including walls and the servicing ventilation system.	84
9.14	Dialog for creating a work order.	86

Bibliography

- Reijo Miettinen and Sami Paavola. Beyond the bim utopia: Approaches to the development and implementation of building information modeling. *Automation in construction*, 43: 84–91, 2014.
- Arto Kiviniemi and Ricardo Codinhoto. Challenges in the implementation of bim for fm—case manchester town hall complex. In *Computing in Civil and Building Engineering (2014)*, pages 665–672, 2014.
- NIST GCR. Cost analysis of inadequate interoperability in the us capital facilities industry. *National Institute of Standards and Technology (NIST)*, 2004.
- Burcin Becerik-Gerber, Farrokh Jazizadeh, Nan Li, and Gulben Calis. Application areas and data requirements for bim-enabled facilities management. *Journal of construction engineering and management*, 138(3):431–442, 2011.
- John Wagnon. What is facility management, 2009. <https://www.ifma.org/>[Accessed: 05.06.2018].
- Brian Atkin and Adrian Brooks. *Total facility management*. John Wiley & Sons, 2014.
- William W Cato and R Keith Mobley. *Computer-managed maintenance systems: a step-by-step guide to effective management of maintenance, labor, and inventory*. Elsevier, 2001.
- Ricardo Codinhoto and Arto Kiviniemi. Bim for fm: a case support for business life cycle. In *IFIP International Conference on Product Lifecycle Management*, pages 63–74. Springer, 2014.
- Paul Teicholz et al. *BIM for facility managers*. John Wiley & Sons, 2013.
- CDBB. Uk bim programme, 2009. www.cdbb.cam.ac.uk/BIMLevels/ukbim[Accessed: 15.06.2018].
- EA Pärn, DJ Edwards, and MCP Sing. The building information modelling trajectory in facilities management: A review. *Automation in Construction*, 75:45–55, 2017.

- André Borrmann, Markus König, Christian Koch, and Jakob Beetz. *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. Springer-Verlag, 2015.
- Charles M Eastman, Chuck Eastman, Paul Teicholz, and Rafael Sacks. *BIM handbook: A guide to building information modeling for owners, managers, designers, engineers and contractors*. John Wiley & Sons, 2011.
- E William East. Construction operations building information exchange (cobie): Requirements definition and pilot implementation standard. Technical report, ENGINEER RESEARCH AND DEVELOPMENT CENTER CHAMPAIGN IL CONSTRUCTION ENGINEERING RESEARCH LAB, 2007.
- BIM+. Cobie – getting down to basics, 2017.
www.bimplus.co.uk/people/cobie-getting-down-basics/[Accessed: 15.06.2018].
- Mehmet Yalcinkaya et al. Understanding the technical and cognitive challenges, and closing the gaps in architectural, engineering, construction-facility management standards, 2017.
- Rob Howard and Bo-Christer Björk. Building information modelling–experts’ views on standardisation and industry deployment. *Advanced Engineering Informatics*, 22(2):271–280, 2008.
- Pouriya Parsanezhad and Johannes Dimyadi. Effective facility management and operations via a bim-based integrated information system, 2013.
- Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. ” O’Reilly Media, Inc.”, 2013.
- buildingSMART. **IFC GUID** summary, 2018.
www.buildingsmart-tech.org/implementation/get-started/ifc-guid[Accessed: 23.06.2018].
- IfcOpenShell. About ifcopenshell, 2018.
[www.http://ifcopenshell.org/](http://www.ifcopenshell.org/)[Accessed: 23.07.2018].
- W3C**. Scalable vector graphics (svg) 1.1 (second edition), 2011.
<https://www.w3.org/TR/SVG11/>[Accessed: 23.06.2018].
- Antero Taivalsaari, Tommi Mikkonen, Matti Anttonen, and Arto Salminen. The death of binary software: End user software moves to the web. In *Creating, Connecting and Collaborating through Computing (C5), 2011 Ninth International Conference on*, pages 17–23. IEEE, 2011.
- J Wix and R See. An introduction to the international alliance for interoperability and the industrial foundation classes. *International Alliance for Interoperability*, Oakton, Va, 1999.

- Mikael Laakso, Arto Kiviniemi, et al. The ifc standard—a review of history, development, and standardization. *Journal of Information Technology in Construction*, 2012.
- DB-Engines. Db-engines ranking of graph dbms, 2018a.
<https://db-engines.com/en/ranking/graph+dbms>[Accessed: 05.07.2018].
- Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data.* ” O’Reilly Media, Inc.”, 2015.
- Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in action.* Manning Publications Co., 2014.
- Richard Mordinyi, Philipp Schindler, and Stefan Biffl. Evaluation of nosql graph databases for querying and versioning of engineering data in multi-disciplinary engineering environments. In *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pages 1–8. IEEE, 2015.
- Ali Ismail, Ahmed Nahar, and Raimar Scherer. Application of graph databases and graph theory concepts for advanced analysing of bim models based on ifc standard. *Proceedings of EGICE*, 2017.
- LDB. Linked building data community, 2018.
www.w3.org/community/lbd/[Accessed: 08.07.2018].
- MDN. Svg: Scalable vector graphics, 2018.
developer.mozilla.org/en-US/docs/Web/SVG[Accessed: 08.07.2018].
- WC3. The secret origin of svg, 2010.
www.w3.org/Graphics/SVG/WG/wiki/Secret_Origin_of_SVG[Accessed: 08.07.2018].
- Patrick Dengler, A Grasso, C Lilley, C McCormack, D Schepers, and J Watt. Scalable vector graphics (svg) 1.1, 2011.
- Julian Ohrt and Volker Turau. Simple indoor routing on svg maps. In *Indoor Positioning and Indoor Navigation (IPIN), 2013 International Conference on*, pages 1–6. IEEE, 2013.
- Single-page application. Single-page application — wikipedia, the free encyclopedia, 2018.
en.wikipedia.org/wiki/Single-page_application[Accessed: 11.08.2018].
- Website. Web site — wikipedia, the free encyclopedia, 2018.
en.wikipedia.org/wiki/Website[Accessed: 11.08.2018].
- HTML. Html — wikipedia, the free encyclopedia, 2018.
en.wikipedia.org/wiki/HTML#Elements[Accessed: 11.08.2018].
- Cascading Style Sheets. Cascading style sheets — wikipedia, the free encyclopedia, 2018.
en.wikipedia.org/wiki/Cascading_Style_Sheets[Accessed: 11.08.2018].

- Floor plan. Floor plan — wikipedia, the free encyclopedia, 2018.
www.w3.org/Graphics/SVG/WG/wiki/Secret_Origin_of_SVG[Accessed: 08.07.2018].
- Plan of Saint Gall. Plan of saint gall — wikipedia, the free encyclopedia, 2017.
en.wikipedia.org/wiki/Plan_of_Saint_Gall[Accessed: 11.07.2018].
- Richard McPartland. Bim dimensions - 3d, 4d, 5d, 6d bim explained, 2017.
www.thenbs.com/knowledge/bim-dimensions-3d-4d-5d-6d-bim-explained[Accessed: 11.07.2018].
- Anoop Sattineni and Taylor Schmidt. Implementation of mobile devices on jobsites in the construction industry. *Procedia Engineering*, 123:488–495, 2015.
- Antti Oulasvirta, Sara Estlander, and Antti Nurminen. Embodied interaction with a 3d versus 2d mobile map. *Personal and Ubiquitous Computing*, 13(4):303–320, 2009.
- Alexander Salveson Nossum. Developing a framework for describing and comparing indoor maps. *The Cartographic Journal*, 50(3):218–224, 2013.
- Judith A Tyner. Principles of map design. Technical report, Guilford Press., 2010.
- Alexandra Lorenz, Cornelia Thierbach, Nina Baur, and Thomas H Kolbe. Map design aspects, route complexity, or social background? factors influencing user satisfaction with indoor navigation maps. *Cartography and Geographic Information Science*, 40(3):201–209, 2013.
- Aalto University. Floorplan of otakaari 1, undergraduate centre, 2017.
<http://biz.aalto.fi/en/>[Accessed: 24.07.2018].
- DB-Engines. Db-engines ranking dbms, 2018b.
<https://db-engines.com/en/ranking/>[Accessed: 05.07.2018].
- buildingSMART. Fm basic handover, 2009.
<http://www.buildingsmart-tech.org/specifications/ifc-view-definition/fm-handover-aquarium/fm-basic-handover/>[Accessed: 30.07.2018].
- buildingSMART. Release. ifc specification, 2006.
- Brick. A uniform metadata schema for buildings, 2017.
<https://brickschema.org/>[Accessed: 01.08.2018].
- Neo4j. The neo4j developer manual v3.4, 2018.
<https://neo4j.com/docs/developer-manual/3.4/>[Accessed: 01.08.2018].
- Node.js. About node.js, 2018.
<https://nodejs.org/en/>[Accessed: 01.08.2018].

Pete Hunt. Why did we build react?, 2013.

<https://reactjs.org/blog/2013/06/05/why-react.html>[Accessed: 11.08.2018].

React. React documentation, 2018.

<https://reactjs.org/docs/getting-started.html>[Accessed: 11.08.2018].

Pannellum. Pannellum documentation, 2018.

<https://pannellum.org/>[Accessed: 11.08.2018].

Autodesk Forge. Autodesk forge documentation, 2018.

<https://developer.autodesk.com/>[Accessed: 11.08.2018].

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Master-Thesis selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ich versichere außerdem, dass die vorliegende Arbeit noch nicht einem anderen Prüfungsverfahren zugrunde gelegen hat.

Munich, TT. Monat JJJJ

Daniel Zibion

Daniel Zibion
Adalperostraße 19
D-85737 Munich
e-Mail: daniel.zibion@tum.de