Chair of Computational Modeling and Simulation
TUM School of Engineering and Design
Technical University of Munich

Implementation of GraphQL Interfaces for Integrated Querying of Heterogeneous Domain Data

Scientific work to obtain the degree

Master of Science (M.Sc.)

at the TUM School of Engineering and Design
of the Technical University of Munich.

Supervised by Prof. Dr.-Ing. André Borrmann
Dr.-Ing. Sebastian Esser
Chair of Computational Modeling and Simulation

Submitted by Nepomuk Wolf [redacted]
[redacted]
[redacted]
e-Mail: nepomuk.wolf@tum.de

Submitted on 21 August 2024

Abstract

Collaboration in the Architecture, Engineering and Construction (AEC) industry is characterized by the involvement of numerous project partners, the utilization of diverse data formats and the frequent exchange of heterogeneous yet highly interconnected resources. Information is often exchanged monolithic, via individual files, documents and container formats, but many applications require more detailed, fine-grained data access.

The current literature exhibits a strong interest in solving these issues, for example, using Semantic Web technologies or the centralization of information through Common Data Environments and Model Servers. Despite these advancements, there remains a need for interfaces that provide direct, detailed access to specific resources and are easy to integrate.

GraphQL, a popular web API technology, enables precise data access and is known for its clarity and user-friendliness. This thesis explores the design and implementation of a GraphQL API to assess its suitability in the Building Information Modeling (BIM) context. Five specific use case scenarios, addressing typical data exchange needs in the industry, were defined and served as the foundation for developing and testing the API.

The results show that GraphQL offers significant potential for the construction industry, as it greatly facilitates access to diverse and interconnected resources. This capability is essential for enhancing collaboration and improving data exchange in future projects.

Zusammenfassung

Zusammenarbeit in der Architektur-, Ingenieur- und Bauindustrie (AEC) umfasst die Koordinierung zahlreicher Projektpartner und den häufigen Austausch heterogener, und dennoch stark vernetzter Ressourcen. Informationen werden oft monolithisch als Dateien oder über Containerformate ausgetauscht, obwohl ein feinerer Datenzugriff für viele Anwendungsgebiete notwendig wäre.
Die aktuelle Literatur zeigt, dass ein großes Interesse daran besteht diese Probleme zu lösen, beispielsweise durch den Einsatz von Semantic-Web Technologien oder durch die Zentralisierung von Informationen mit Hilfe von Common Data Environments und Model Servern. Trotz der Fortschritte in diesen Bereichen besteht weiterhin ein Bedarf an Schnittstellen, die direkten, fein-granularen Zugriff auf spezifische Ressourcen ermöglichen und gleichzeitig verhältnismäßig einfach zu integrieren sind.
GraphQL ist eine API Architektur und Abfragesprache, deren Popularität in den letzen Jahren stark zugenommen hat. GraphQL ermöglicht einen präzisen Datenzugriff und ist bekannt für Verständlichkeit und Benutzerfreundlichkeit. Im Rahmen dieser Arbeit wurde eine GraphQL API entwickelt und untersucht um die Eignung dieser Technologie im Kontext von Building Information Modelling zu bewerten. Dazu wurden fünf spezifische Anwendungsszenarien, die typische Anforderungen der Branche abdecken, definiert und als Grundlage für die Entwicklung und Erprobung der API verwendet.
Die Ergebnisse zeigen, dass GraphQL großes Potenzial für die Bauindustrie bietet, da es den Zugang zu heterogenen und stark vernetzten Ressourcen erheblich erleichtert. Dies ist entscheidend, um die Zusammenarbeit im Bauwesen zu verbessern und den Datenaustausch für zukünftigen Projekten zu optimieren.

Contents

Abbreviations

AEC	Architecture Engineering Construction
API	Application Programming Interface
AST	Abstract Syntax Tree
BCF	BIM Collaboration Format
BIM	Building Information Modeling
BOT	Building Topology Ontology
CDE	Common Data Environment
CET	Central European Time
CRUD	Create, Read, Update, Delete
foaf	friend of a friend
GIS	Geographic Information Systems
GUID	Global Unique ID
HTTP	Hypertext Transfer Protocol
HVAC	Heating, Ventilation, and Air Conditioning
IFC	Industry Foundation Classes
IoT	Internet of Things
JSON	JavaScript Object Notation
MEP	Mechanical, Electrical and Plumbing
OOP	Object Oriented Programming
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
regex	Regular Expression
REST	Representational State Transfer

SDAI Standard Data Access Interface
SOAP Simple Object Access Protocol
SPARQL SPARQL Protocol And RDF Query Language
SPF STEP Physical File
SQL Structured Query Language

URI Unique Resource Identifier
URL Unique Resource Locator
UTC Coordinated Universal Time

W3C World Wide Web Consortium

XML Extensible Markup Language

Chapter 1

Introduction

1.1 Problem Statement

Building Information Modeling (BIM) projects are characterized by the collaboration of many different project partners and stakeholders often with diverging motives and objectives. Several disciplines have to collaborate on large and complex projects, where the work performed in one discipline is dependant on the work carried out in other disciplines.
Structural engineering might be bound by the architect's design, ensuring that the building's structure aligns with the aesthetic and functional vision. Mechanical, Electrical and Plumbing (MEP) engineers need to integrate their systems without compromising the structural integrity or aesthetic vision. Civil and geotechnical engineers must coordinate with both, structural engineers and architects, to ensure that the site infrastructure supports the building's design and functionality. Construction managers rely on the detailed plans provided by these disciplines to plan and coordinate construction, while this entire process should adhere to the constraints of a given budget.
Constant communication, coordination and data exchange between all partners is required to navigate this web of inter-dependencies, where any change in one area usually affects multiple other disciplines and triggers numerous additional changes.

The project data resulting from this complex structure, consists of highly interconnected resources such as 3D models, material data, analysis reports, standards, specifications, project and construction schedules, among others. Even though the disciplines are dependent on one another, each discipline organizes and represents its design knowledge and related resources in a way that best suits its own specific purposes.
Collaboration and data management for large projects are therefore an important task and are tackled commonly through technologies like Common Data Environment (CDE), Model Server or other platform solutions that enable centralised data storage and facilitate information exchange.
Traditionally, the data management and exchange in this context happens on the

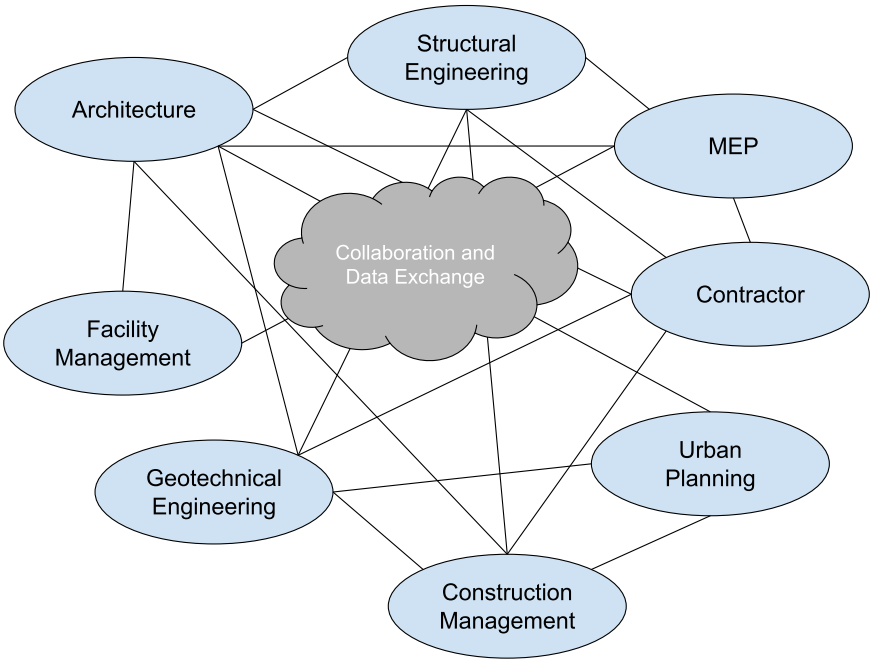

Figure 1.1: Web of collaboration and data exchange in the Architecture Engineering Construction (AEC) industry.

basis of files and documents or container formats. With an information granularity like this, smaller knowledge entities usually cannot be handled. To update, extend, or retrieve parts of a BIM model, entire files, for example, have to be up- or downloaded to or from the centralized storage solution. This approach implies that accessing information about a specific entity (e.g., a single building element in the model) the entire model containing the entity in question has to be retrieved. Increasing project complexity and interdisciplinary dependencies that can be observed in current projects further complicate accessing and providing the desired design information. This leads to problems, such as exchanging more information than necessary, making it difficult to implement and track changes consistently and often making access to specific, detailed information unnecessarily complex. This indicates a general demand in the AEC industry for powerful interfaces to integrate and exchange information of this kind in a fine-grained manner. One technology that enables the implementation of such interfaces in a web-based environment is GraphQL.

1.2 GraphQL as an API Language

GraphQL is a query language and an architectural Application Programming Interface (API) paradigm and as such an efficient alternative to query languages

like Structured Query Language (SQL) as well as API technologies like REST. It shares similarities with the Representational State Transfer (REST) architecture but in contrast to REST, GraphQL APIs expose the entire underlying data via a single endpoint and offers clients precise control over the returned data. Originally developed by Facebook, it is now maintained by the non-profit Linux Foundation.
In recent years, GraphQL gained notable popularity in the industry (Postman, 2023) and there is a rising interest in utilising graph-based technologies in the BIM context. Graph-based solutions promise to offer distinct advantages, especially when it comes to tasks like sub-model querying or accessing specific model elements without the need to traverse and read the entire model. Version control and tracking of changes are also promising applications of graph-based technologies and show that representing and processing building models as graphs brings a lot of potential.
GraphQL aligns with this development as it provides the means to interact with interconnected information of any kind, while offering a great flexibility in the specific representation that can be used to store information. Because of this, GraphQL seems like a suitable choice as a query language and API paradigm in the built environment.

1.3 Research Questions

This thesis consists of two separate parts. The first part provides a literature review and overview of existing methods related to data exchange, collaboration and GraphQL as a technology (chapters 2 and 3). The second main part presents the proposal and implementation of a GraphQL API (chapters 4 and 5). The first part describes the technical background and seeks to address the following research questions:

- What is the current state of research on fine-grained data access to heterogeneous, highly interconnected knowledge within the AEC industry?
- Have previous studies explored the application of GraphQL in the AEC domain?

The second part focuses on developing and evaluating a GraphQL API. With an emphasis on exploring GraphQLs capabilities to query interconnected data in the AEC context. This case study aims to:

- Explore the feasibility of using a GraphQL API to query heterogeneous resources as they are commonly found in the BIM context.

- Subsequently examine the usefulness, advantages and limitations of this approach.

To this end, the prototypical GraphQL API will be tested against queries developed from specific user scenarios. These user scenarios encompass typical requests, ranging in complexity from simple queries to the retrieval of complex interlinked resources.

1.4 Outline

Chapter 2 delves into the literature and current research on fine-grained access to heterogeneous knowledge representations, as well as GraphQL and associated technologies within the BIM context. A strong focus is placed on topics related to the Semantic Web and Linked Data. These technologies are currently considered the leading means by several researchers for integrating heterogeneous domain data and accessing interconnected resources, not only in the AEC domain but in other industries as well. Existing approaches to using GraphQL specifically in the BIM context are also examined.
Chapter 3 provides a comprehensive introduction to GraphQL as both, an API architecture design and as a query language. It introduces the terminology and concepts related to GraphQL, as well as offers a brief overview of the predominant alternative regarding API design, which is REST.
Chapter 4 illustrates the approach of utilizing GraphQL in the context of BIM and the AEC industry, providing the general design philosophy and direction of the implementation developed as part of this thesis.
Chapter 5 provides a detailed description of the case study conducted as part of this thesis. This includes a general introduction and overview of the GraphQL API that was developed and implemented, as well as a more detailed illustration of design decisions, difficulties and solutions.
Chapter 6 discusses the results, accomplishments and difficulties of the implemented API, examining the usefulness of GraphQL for accessing heterogeneous resources in the BIM environment. The limitations of the proposed approach and GraphQL as a technology in the context of BIM are also investigated in this section.
Chapter 7 provides a conclusion and outlook of the entire project, which includes the theoretical foundation and practical implementation.

Chapter 2

Related Works and Background

This chapter reviews current research on aspects that are related to this thesis. While the use of GraphQL in the AEC industry naturally stands out as a primary topic, as the title of this thesis indicates, there are only a few examples where GraphQL has been applied in the AEC context.
As a result, the scope is broadened to include other graph-based and web-related technologies that either share similarities with GraphQL or serve as alternatives for addressing specific challenges. This includes, primarily, principles related to the Semantic Web. Additionally, this chapter introduces key concepts that are essential for understanding the subsequent sections, providing the necessary background. For instance, it includes an overview of the data models used in the case study presented in chapter (chapter 5). Furthermore, the chapter also illustrates current collaboration and data exchange practices, highlighting the issues a GraphQL API may address, as well as the requirements for an API within the BIM environment.

2.1 BIM as Interconnected Knowledge Representations in the Built Environment

In the AEC industry, collaboration and data management have historically been characterized by the use of monolithic models within proprietary environments. Each discipline uses different – most of the time proprietary – software tools and stores its resources in various (proprietary) data formats. The use of different environments often hinders efficient and lossless data exchange, which is an integral part of almost every building project.
Historically, the exchange of information was done via (physical) documents like blueprints, plans, tables and similar, which imposes several issues regarding the consistency of shared models especially with increasing project size and complexity. Even though BIM technologies have been developed over decades to address these issues, bilateral document-based data exchange still remains a common form of collaboration to this day. Although documents are usually exchanged in a digital domain rather than on paper, some of the same problems still exist.
In practice collaboration typically involves the cumbersome process of exchanging

entire models in the form of files, despite the highly interconnected and hierarchical structured nature of building projects. This practice is further complicated by the diverse needs of various stakeholders, each working on separate but interconnected partial models. This section reviews the current state of these practices, examining the limitations of existing methods and exploring how GraphQL might offer a more efficient and flexible solution for managing heterogeneous resources in the AEC industry.

2.1.1 BIM Maturity Levels

The BIM Task Group, funded by the UK government, developed a BIM Maturity Model to describe and classify the stages of BIM technology development. This model has gained popularity due to its clear and practical depiction of how data can be exchanged at various levels of granularity given the level of development of the BIM technologies used. In order to understand the collaboration requirements, it is important to consider the evolution of collaboration across the different maturity levels, in particular the vision of 'ideal' collaboration and data exchange as described by the highest maturity level (level 3). This vision serves as a target definition that can guide the development of future technologies.

The following paragraph will give a short overview of the concept and the individual maturity levels following Borrmann, König, et al. (2021), given the topic of this thesis, the focus will be on data exchange.

- **Level 0**
  Level 0 describes the traditional data exchange of (printed) documents in the form of 2D-drawings.

- **Level 1**
  The data exchange of level 1 adds the possibility to exchange 3D-model files, while still relying on 2D-drawings for certain information. Both level 0 and 1 happen on a bilateral level. The data is somehow transferred from one project partner to the other.

- **Level 2**
  BIM Maturity Level 2 is characterised by the use of BIM software tools by all involved disciplines. Generally independent discipline models are created using proprietary software. The data exchange happens on the file level using centralized solutions like CDE.

- **Level 3**
  Level 3 is the final maturity level envisioned. The main difference to level

2 is the granularity of the data exchange. Information should be stored in a centralized location and should be accessible on the element level. This increases control over access rights, tracking of changes, exchange of partial resources and similar cases. For example, at Level 2, a building model is transferred between stakeholders using files. When a stakeholder makes changes to their version of the model, these changes need to be communicated to other project partners through the exchange of the entire model file. Each partner may then need to update their own designs accordingly. In contrast, Level 3 allows changes to be made directly to a centralized model, with all stakeholders working on the same version in real time.

According to Rasmussen et al. (2021) the vast majority of projects still happen on the maturity levels 0 to 2 depending on the country and the involved companies. Data is usually stored in a central, web-based location accessible to all project partners and is exchanged using formats like Industry Foundation Classes (IFC). Although, there is a lot of active research in this area, the access and exchange of information on the element level as envisioned by maturity level 3 is still to be established in practice. As a major prerequisite, this requires the development of methods and interfaces to access and connect information on the data level.

2.1.2 Current State of Collaboration and Data Exchange

Jaskula et al. (2024) provides a comprehensive analysis of current practices and challenges regarding collaboration in large scale construction projects. The paper includes a thorough literature review combined with user surveys to identify the current issues that are commonly encountered in AEC projects. This serves as a foundation to identify and list key characteristics and current issues of collaboration and data exchange in this field.
The AEC industry is inherently multidisciplinary, requiring collaboration among various professionals and stakeholders. Effective collaboration in this sector is shaped by the following several key requirements and prerequisites:

Diverse Disciplines and Software Tools
The AEC industry includes a multitude of disciplines, each utilizing specialized software tools tailored to the specific requirements of the discipline (Borrmann, Beetz, et al., 2021). Architects, engineers, project managers and construction specialists work on separate products using specialised software tools, which must somehow be combined into a single finished product: the building. This diversity leads to the existence of various different knowledge representations and data

formats, creating challenges for information exchange and communication. The heterogeneity of software tools poses significant challenges for interoperability.

Integration of Multiple, Highly Interconnected Resources

A fundamental aspect of collaboration in the AEC industry is the integration of multiple discipline models to create a single, consistent building model (Schapke et al., 2021). This integration necessitates the use of highly interconnected resources, where each discipline's contributions must align and synchronize with others. The successful merging of a unified building model relies on the precise and seamless coordination of various discipline-specific models. Typically, a large amount of data from several sources needs to be integrated and coordinated. Research indicates, that the use of a single centralized source of truth for all disciplines throughout the entire project life cycle is not practical (Jaskula et al., 2023). The findings indicate that in real-world examples, several CDE solutions are used over the life cycle of a building. This further emphasises the need for uniform interfaces in order to integrate information across these data silos.

Frequent and Extensive Data Exchange

Collaboration in the AEC industry involves frequent and extensive data exchange. As each discipline's progress often depends on the outputs of others, efficient and lossless exchange of information is vital. For instance, structural engineers may need architectural designs to proceed with their calculations, while Heating, Ventilation, and Air Conditioning (HVAC) specialists require structural layouts to plan their systems. This interdependence underscores the need for robust mechanisms to facilitate continuous and efficient data exchange. In practice, this frequently occurs through channels without proper documentation and traceability like emails, meeting, reports or generalized cloud storage solutions (Soman & Whyte, 2020). According to Jaskula et al. (2024) 47% of interviewees named the lack of interoperability as an issue in large scale projects.

Collaboration Over the Entire Life Cycle

The need for collaboration extends beyond the design and construction phases to incorporate the entire life cycle of a building. Collaboration must be maintained from initial design through to construction, operation, and eventual decommissioning, which in the case of building projects usually spans over several decades. This requirement, however, relates more to the longevity of the underlying technical infrastructure and services than the development of novel data exchange mechanisms and is therefore of lower importance for the topic of this thesis compared to the other issues.

Data Granularity

Another important topic not mentioned in detail by Jaskula et al. (2024) is data

granularity. It refers to the level of detail, at which information is accessible. It is very common to exchange information at the level of entire files. In practice, BIM models, for example, usually are exchanged as entire files using the IFC exchange format.
Several tasks, however, rely on data management and data access at a finer level, this includes, for example, version control (Esser et al., 2022) or model coordination (Schapke et al., 2021). These tasks require access to or traceability of single objects or entities, like e.g., singular building elements. Enabling data access on the object level is stated as a general goal for BIM development by several researchers, e.g. (Werbrouck, Senthilvel, Beetz, & Pauwels, 2019) and is one of the defined requirements for BIM maturity level 3 (see section 2.1.1).

All of these topics highlight the importance of standardized, high-performance interfaces that enable the integration, retrieval, authoring, and coordination of heterogeneous resources. Additionally, they provide an initial set of criteria for evaluating interfaces. Following these general requirements, interfaces should be **flexible** (to account for various data structures and user scenarios), **uniform** (provide a reliable, standardised and documented way of interacting with the resources) and allow **fine-grained** data access.

BIM Model Server

One approach to combat the described difficulties in collaboration is the use of model servers. The basic premise is to *„... collect all data of a building model in a shared representation“* (Jørgensen et al., 2008) . All relevant model data is stored, and hosted in a centralised location, combining and coordinating all discipline models. This idea is not new and the first model servers based on the IFC data format have been successfully implemented in the early 2000s (Kiviniemi et al., 2005). Depending on the actual implementation, these server implementations and available clients offer different functionalities, ranging from simple information retrieval to full-fetched model authoring. Beetz et al. (2010) present an implementation of an open-source model server aimed at enabling storage, maintenance and querying of building models based on the IFC data format. Instead of using a file-based representation as the persistence layer, the model data is imported into a object-relational database for easier access and more efficient querying. The server implementation offers a web API based on the Simple Object Access Protocol (SOAP) protocol that supports remote server integration in third-party applications. Even though it was developed around 2010 it is still under active development with the latest release being dated at February

6, 2023[1].

The use of model servers is suggested numerous times in the literature as a solution to various problems related to file-based data exchange (Beetz et al., 2010; Solihin & Eastman, 2016) and is one step towards BIM maturity level 3. The following advantages can be identified:

Integrating multiple discipline models

A model server provides a centralized location, where partial models from different disciplines such as architecture, structural, mechanical and electrical engineering, plumbing, and others, can be stored, accessed and integrated for efficient collaboration and data exchange, as well as for tasks such as clash detection.

Simplify data exchange

Model servers can simplify the exchange of information. Providing a centralized data storage for all stakeholders to access is already a major improvement over bilateral, file-based data exchange. In this way, a model server offers similar benefits compared to a CDE restricted to building model data.

Enable partial model exchange

Most model server implementations provide a mechanism to query and filter the model in a way that allows the retrieval (and potentially export) of partial models (Kiviniemi et al., 2005). This can theoretically range from entire discipline sub-models up to singular building elements. If and how these partial models can be exported or exchanged is, however, entirely dependent on the specific implementation and in most cases, tried and tested exchange mechanisms such as IFC are used.

In the recent years, this concept has evolved further and several companies in the industry have developed platforms for collaboration and data exchange that go beyond simple model servers and provide functionality for the entirety of the project management cycle in the building sector, including model server, document management, cost management and so on. Products like BimCloud[2], BIM360[3] or BIM+[4] can be seen as examples.

However, there still seem to be open questions about standardized data exchange between proprietary software solutions. This is particularly challenging if the information is exchanged on a level more fine-grained than file-based. Without the integration of standardized and accessible interfaces into these platforms

[1] https://github.com/opensourceBIM/BIMserver, accessed: 24.08.2024
[2] https://graphisoft.com/de/teamwork/bimcloud, accessed: 03.08.2024
[3] https://www.autodesk.com/bim-360/, accessed: 03.08.2024
[4] https://www.allplan.com/de/produkte/allplan-bimplus/, accessed: 03.08.2024

collaboration and data exchange exhibits the same well-known issues whenever information needs to leave the proprietary environment.

One general limitation that most model server approaches share is the lack of support for federated heterogeneous resources. Usually, there is a persistence layer in the form of a database that is responsible for the permanent storage of model data and all related information. In a way, this acts as a centralized data storage even if the database itself can be set up in a federated way. It still requires the resources to be (1) uploaded to the server and (2) compatible with the storage scheme of the database. Typically, it does not provide the means to integrate truly distributed resources as Linked Data technologies would make possible. This issue can be somewhat mitigated by integrating standardized and well documented interfaces to enable integration from outside the environment.

Common Data Environments

CDEs are another answer to the issues in collaboration and data exchange sketched above and are commonly used in bigger projects. Looking at the advantages, CDE share similarities with model servers that are discussed in the previous section. Both are based on the idea of centralizing data used by different project partners to increase consistency and simplify collaboration. A CDE provides a unified location to store, exchange and manage information related to a project and is specified in the ISO 19650-1 (2018) standardization. The primary focus are the data exchange and information consistency between disciplines (Preidel et al., 2018).

Even though there are conceptual similarities with model servers, a CDE is not required to provide fine-grained access to building model information. The data management and exchange usually happens on the level of files or even container formats which can contain multiple files accompanied by metadata (Preidel et al., 2018).

Jaskula et al. (2023) investigate and compare different CDE solutions that are currently used in the industry and conclude that there is the need to include multiple CDEs throughout the life cycle of a single project. Which in turn shows the need for standardized interfaces to exchange information not only inside a single CDE but between different CDEs. Jaskula et al. (2024) however, identify the lack of interoperability as one of the issues associated with the use of these proprietary CDE platform solutions. CDEs play a vital role in enabling collaboration and data exchange in the AEC industry, however data exchange often still happens on the level of documents. Moreover, there is little support for interoperability between

different CDEs, respectively interfaces to enable data integration from outside the CDE.

2.1.3 Data Models

Data modeling describes the process of formally describing an abstracted part of reality. A data model consists of the description of relevant classes, with corresponding attributes and the description of the relationships existing between classes (Artus et al., 2021).
The case-study implementation of this thesis uses two different types of resources: IFC and BIM Collaboration Format (BCF). Both are very common formats in the AEC industry and both aim to enable standardised exchange of information. While IFC was developed to exchange geometric, topological, and syntactic information about building models between different project partners, BCF is used for issue management and as a means to collaborate on existing problems, desired changes, and the coordination of tasks and solutions (Schapke et al., 2021). The GraphQL schema should reflect the information (or relevant parts of the information) provided by these two data types. An understanding of the internal structure and knowledge represented is necessary to inform the design of the GraphQL schema. Therefore, the following sections provide a brief introduction to both IFC and BCF

IFC

IFC is the dominant exchange format for building models. Almost all proprietary software products for creating, editing, and analyzing building models support the import and export of models in the IFC data format.
IFC, a data format developed by buildingSMART, is designed to be a neutral, open standard for the exchange of BIM models. The first version was released in 1997, and the current version is IFC 4.3, with ongoing development towards IFC 5.
IFC employs the data modeling language EXPRESS, which adheres to object-oriented principles. These include the use of classes (referred to as entities in IFC), attributes, relationships, and concepts, like inheritance to define the data model. Figure 2.1 depicts how the IFC entity *IfcWall* is described in EXPRESS. It highlights the extensive inheritance structure in IFC. For example, *IfcWall* inherits attributes from multiple super classes, including *GlobalId*, *OwnerHistory*, *Name*, and *Description*, which are inherited from the *IfcRoot* entity. A unique feature of the EXPRESS data modeling language is the definition of inverse relationships. Unlike direct relationships, which reference another entity through an attribute, inverse relationships indicate that another entity is referencing this entity. This simplifies the

```
ENTITY IfcWall;
   ENTITY IfcRoot;
      GlobalId                    : IfcGloballyUniqueId;
      OwnerHistory                : IfcOwnerHistory;
      Name                        : OPTIONAL IfcLabel;
      Description                 : OPTIONAL IfcText;
   ENTITY IfcObjectDefinition;
   INVERSE
      HasAssignments              : SET OF IfcRelAssigns FOR RelatedObjects;
      IsDecomposedBy              : SET OF IfcRelDecomposes FOR RelatingObject;
      Decomposes                  : SET [0:1] OF IfcRelDecomposes FOR RelatedObjects;
      HasAssociations             : SET OF IfcRelAssociates FOR RelatedObjects;
   ENTITY IfcObject;
      ObjectType                  : OPTIONAL IfcLabel;
   INVERSE
      IsDefinedBy                 : SET OF IfcRelDefines FOR RelatedObjects;
   ENTITY IfcProduct;
      ObjectPlacement             : OPTIONAL IfcObjectPlacement;
      Representation              : OPTIONAL IfcProductRepresentation;
   INVERSE
      ReferencedBy                : SET OF IfcRelAssignsToProduct FOR RelatingProduct;
   ENTITY IfcElement;
      Tag                         : OPTIONAL IfcIdentifier;
   INVERSE
      HasStructuralMember         : SET OF IfcRelConnectsStructuralElement FOR RelatingElement;
      FillsVoids                  : SET [0:1] OF IfcRelFillsElement FOR RelatedBuildingElement;
      ConnectedTo                 : SET OF IfcRelConnectsElements FOR RelatingElement;
      HasCoverings                : SET OF IfcRelCoversBldgElements FOR RelatingBuildingElement;
      HasProjections              : SET OF IfcRelProjectsElement FOR RelatingElement;
      ReferencedInStructures      : SET OF IfcRelReferencedInSpatialStructure FOR RelatedElements;
      HasPorts                    : SET OF IfcRelConnectsPortToElement FOR RelatedElement;
      HasOpenings                 : SET OF IfcRelVoidsElement FOR RelatingBuildingElement;
      IsConnectionRealization     : SET OF IfcRelConnectsWithRealizingElements FOR RealizingElements;
      ProvidesBoundaries          : SET OF IfcRelSpaceBoundary FOR RelatedBuildingElement;
      ConnectedFrom               : SET OF IfcRelConnectsElements FOR RelatedElement;
      ContainedInStructure        : SET [0:1] OF IfcRelContainedInSpatialStructure FOR RelatedElements;
   ENTITY IfcBuildingElement;
   ENTITY IfcWall;
END_ENTITY;
```

Figure 2.1: Definition of the *IfcWall* entity in the data modelling language **EXPRESS**.
Source: https://standards.buildingsmart.org/IFC/RELEASE/IFC2x3/TC1/HTML/ifcsharedbldgelements/lexical/ifcwall.htm

```
ENTITY IfcRelFillsElement;
 ENTITY IfcRoot;
  GlobalId          : IfcGloballyUniqueId;
  OwnerHistory      : IfcOwnerHistory;
  Name              : OPTIONAL IfcLabel;
  Description       : OPTIONAL IfcText;
 ENTITY IfcRelationship;
 ENTITY IfcRelConnects;
 ENTITY IfcRelFillsElement;
  RelatingOpeningElement : IfcOpeningElement;
  RelatedBuildingElement : IfcElement;
END_ENTITY;
```

Figure 2.2: Objectified relationship in IFC.
Source: https://standards.buildingsmart.org/IFC/RELEASE/IFC2x3/TC1/HTML/ifcproductextension/lexical/ifcrelfillselement.htm

navigation to related entities without the need to add new information (Borrmann, Beetz, et al., 2021). Related to this is the concept of *objectified relationships*, which enables the modeling of relationships between two entities as separate objects. The main advantage of this approach is that it allows additional information to be attached to the connection between the two entities. An example of this concept is shown in figure 2.2. *IfcRelFillsElement* connects a *IfcOpeningElement* with the entity *IfcElement*, signifying that this opening is being (partially) filled by a specific *IfcElement* instance. These concepts will be of further importance for the case-study described in the chapter 5 in general, and the relationships between the IFC schema and the design of the GraphQL schema in particular.

IFC was developed to facilitate the exchange of comprehensive building models, aiming to meet the modeling requirements of all sub-disciplines to describe an integrated overall model. As a consequence the schema, as well as instance models, are quite extensive and complex. IFC instance models can be serialized in several ways. The most widely used format is the STEP Physical File (SPF), based on ISO 10303-21 (2016), which provides clear text encoding of EXPRESS data models. Depending on the specific use case, other serializations may be more appropriate. Other popular encodings use Extensible Markup Language (XML), JavaScript Object Notation (JSON) or Resource Description Framework (RDF) to represent IFC model data.
JSON, for instance, is a highly popular data format that many web developers are familiar with. The use of well-known data formats facilitates access and thus enables improved integration of the IFC data model into other applications. In the context of Semantic Web and Linked Data a RDF based serialization like turtle (.ttl) would be more appropriate to represent IFC model data.

BCF

BCF regulates the exchange of model-based communication regarding issues, change requests, and similar matters (Schapke et al., 2021). It is a relatively simple data format that makes it possible to create and exchange so called *topics* related to specific viewpoints based on specific IFC models. A topic can relate to certain building elements via the IFC attribute *GlobalId* and include additional information, such as comments, which can be used as a communication channel about specific issues. There are several relationships that can be established between an IFC model and BCF, which are discussed in detail in section 5.3.3.
Two approaches are defined by buildingSMART to exchange BCF information between project partners:

- A file based exchange; based on XML and ZIP
- A RESTful web API; an extensive specification of all endpoints including required and allowed query parameters as well as the exact structure of the response body in the JSON format is provided on GitHub[5].

The case study uses the file-based approach to represent and store BCF. This was chosen to avoid the need to include either a third-party server or a custom server implementation.

2.1.4 Early and Late Binding

Overview

This thesis uses building models represented in the IFC format specifically in the STEP clear text encoding according to ISO 10303-21 (2016). There are two possible methods to structure the access to such files and organize access to the underlying information – *early binding* and *late binding*. This chapter provides an overview of both approaches and discusses them in the context of GraphQL. It is important to note that the concepts of early and late binding might not strictly apply to the context of a GraphQL API and schema design. There are, however, interesting similarities that can help categorize different approaches in schema design according to these concepts.
Binding in this context usually refers to the mapping of the EXPRESS definitions to a programming language that is used to process the model information. GraphQL, however, is not a programming language and can be seen in this context more like

[5] https://github.com/buildingSMART/BCF-API, accessed: 05.08.2024

a data modeling language. The GraphQL schema is usually not strictly aligned with the data structure of the resources which also means that in most cases there is no direct mapping between classes/entities in the resources to types in the GraphQL schema. The following description of the early and late binding approaches follows the introduction provided by Amann et al. (2021).

Early Binding

Early binding refers to the mapping of IFC entities to entities of the host programming language, this means all IFC entities have a corresponding entity in the host language. This can be done manually but most of the time it is preferable to automate this process using a code generator. The generator takes a specific IFC schema and produces definitions of the corresponding entities in the host language. "As a rule, EXPRESS entities are mapped to classes in object-oriented programming languages, inheritance is implemented using inheritance syntax and references are created using pointers" Amann et al. (2021). In our specific case the host language is GraphQL, which already has a few limitations to overcome. In order to be able to create an early binding (a representation of the IFC schema in GraphQL), the following concepts have to be represented adequately in GraphQL:

1. **EXPRESS-entities**: EXPRESS-Entities can be represented as GraphQL types, which are very similar to classes in Object Oriented Programming (OOP).
2. **Explicit references**: Types in GraphQL can use other GraphQL types as member variables. Using this, we can represent explicit references between entities.
3. **Inverse relationships**: Many connections between IFC entities are based on inverse relationships. This allows connecting entities without the use of a specific link in the entity definition. There is no concept similar to inverse relationships in GraphQL and all connections that exist between an entity A and an entity B have to be made explicit in the form of type attributes.
4. **Objectified relationships**: IFC uses objectified relationships, e.g., to connect a wall to its corresponding material. These references are a little bit less straight forward to map to graphQL, but can be done in two different ways: (1) Represent the implicit reference as its own type that references both the wall and the material. (2) Make the material an explicit reference of the wall.
5. **Inheritance**: GraphQL has no native support for inheritance. The closest we can emulate inheritance is by either using interfaces or directives. Inheritance

Algorithm 2.1: Representation of the IFC entity IfcWall in GraphQL

```
type IfcWall {
    globalId: ID!
    ownerHistory: IfcOwnerHistory
    name: String
    describtion: String
    objectType: String
    ObjectPlacement: IfcObjectPlacement
    representation: IfcProductRepresentation
    tag: String
    predefinedType: IfcWallTypeEnum
}
```

can be emulated by using directives[6]. Some of the features offered by inheritance can be achieved this way.

While it seems possible to create an explicit mapping between the IFC schema and GraphQL, implementations would need to overcome the mentioned issues. Listing 2.1 illustrates how an IFC entity could be mapped to GraphQL.

IfcWall, however, also has multiple (up to 24) inverse attributes that are used to further define information about material, the building structure this wall is located in, potential openings and more. This is vital information, which would need to be made explicit in the GraphQL schema, greatly increasing its size.

Late Binding

Late binding uses the Standard Data Access Interface (SDAI), which is specifically defined for the programming languages C, C++ and Java and in general in an abstract form. This approach follows the general premise to omit the step of direct mapping step between IFC schema and host language, providing a generic way of representing and accessing individual entities. This approach offers greater flexibility in exchange for expressiveness (Amann et al., 2021).
In a strict sense, this specification does not seem easily transferable to GraphQL, since GraphQL relies on the definition of a static schema that has to be known before starting the server and can not be changed on the fly. However, the general principles of late binding can still be applied to the GraphQL schema design.
Following these principles, entities are represented in the GraphQL schema in a generic way that enables a flexible definition of all corresponding attributes and relationships. Autodesk seems to follow this approach for their GraphQL

[6] https://nelsondominguez.hashnode.dev/inheritance-in-graphql-when-and-how-to-use-it, accessed: 10.07.2024

API implementation, with a very generalized and flexible schema design. The difference between both approaches will be discussed in more detail in the context of the case study implementation in chapter 5.

2.2 Graph-Based Representations in the Web Context

When talking about heterogeneous, domain-specific knowledge representation, Linked Data is often proposed to enable collaboration and interoperability on the data level. There is extensive research on incorporating Semantic Web and Linked Data technologies into the building and construction sector. The next section provides a closer look at some approaches and offers a general introduction into Linked Data and associated technologies like RDF, Web Ontology Language (OWL) and SPARQL Protocol And RDF Query Language (SPARQL).

2.2.1 Semantic Web Technologies: Linked Data, RDF, OWL, SPARQL

Linked Data within the context of the Semantic Web is a concept that was first introduced by Tim Berners-Lee (Berners-Lee, 2006). The main idea behind it is to define a method for extending the web from linking documents to a web of data, where resources are defined more fine-grained and are connected at the data level. It relies on using Unique Resource Identifiers (URIs) to identify entities and establish connections between them. The knowledge is represented in RDF, a schema to describe data in the form of triples consisting of subject, predicat and object, which represents a common concept of first-order logic (de Bruijn & Heymans, 2010). The links between the subject and object can thereby carry additional semantic information which distinguishes them from normal hyperlinks. The Semantic Web allows to attach meaning to data in a machine readable way that enables querying and automated reasoning. The efforts to establish data access and data exchange at element level in the BIM context have an equivalent (or precursor) in this evolution from the classic web to the so-called *Web of Data*. Therefore, it is not surprising, that there are substantial effort to apply Semantic Web technologies to the construction industry.
The Semantic Web builds on a suite of technologies known as the Semantic Web Stack. This includes RDF, Resource Description Framework Schema (RDFS), OWL and SPARQL The next paragraphs provide an overview of the concept of the

Semantic Web and the associated technologies, outlining the current development and use of Linked Data principles in the AEC industry. The following section is based on the Linked Data overview from Bizer et al. (2023) as well as the official documentation (Hartig, Champin, et al., 2024) and design issues of the World Wide Web Consortium (W3C) (Berners-Lee, 2006).

Semantic Web

The concept relies on the core principle of using Unique Resource Identifiers (URIs) to identify and link resources. Both, resources and links, are represented using the Resource Description Framework (RDF). In contrast to the classic document oriented web approach, links can be established between individual elements rather than between files. To achieve this, Linked Data relies on data being represented in the RDF format.

RDF

RDF is the predominant data model used to represent resources in the context of the Semantic Web. Information is represented in the form of triples, which create a logical connection between the Subject, Predicat and Object. This simple principle allows describing a link between two elements, making it possible to form extensive knowledge graphs that are directed, labeled and highly interconnected. RDF is based on formal logic (predicate logic, first-order logic) allowing powerful automated reasoning over RDF graphs (de Bruijn & Heymans, 2010). RDF is extended by RDFS to add vocabulary that allows the formulation of simple ontologies. There are several serializations available for the RDF data format, including N-Triples, Turtle, JSON-LD and XML.

OWL

OWL extends the syntax of RDF and RDFS with additional vocabulary to allow the definition of so-called ontologies. Ontologies can be used to describe domain knowledge, its concepts and the relationships between them (OWL Working Group, 2012). OWL attaches semantic meaning to data. The current version (OWL 2) was developed by the *W3C OWL Working Group*[7] in 2009. Pauwels and Terkaj (2016) point out the similarities between OWL and EXPRESS as a data modeling language. Both are used to define information and semantics about a domain and relationships between concepts and entities. However, OWL is part of the Semantic Web Stack and therefore offers a suite of general tools for semantic interpretation, reasoning, data exchange and querying. Pauwels and Terkaj (2016) conclude that given the similarities, it should be possible to represent IFC data as RDF graphs, opening up the possibility to use the already existing software tools available as part of the Semantic Web.

[7] https://www.w3.org/2007/OWL/wiki/OWL_Working_Group, accessed: 23.07.2024

SPARQL

SPARQL is a query language recommended by the W3C as the standard for querying resources represented in RDF (Harris & Seaborne, 2013). In this way, it can be seen as the equivalent of SQL for relational databases. SPARQL will be discussed in more detail than the other Linked Data technologies, since it can be directly compared to GraphQL as a query language. A query is formulated by defining a pattern using RDF syntax, where any part of the triple can be marked as a variable. This pattern is then matched against the graph store and the selected parameters are returned for all matching resources. Listing 2.2 shows a simple query expressed in SPARQL, executed on a small data set represented in RDF, including the result of the query. The example is taken from Hartig, Taelman, et al. (2024) and uses the popular friend of a friend (foaf) ontology[8]. A prefix definition is used to simplify the notation and avoid repeating the same resource Unique Resource Locators (URLs) in multiple places. The query consists of a SELECT statement, describing what information is requested and a WHERE statement, providing the condition in the form of a graph pattern. This pattern is matched against the currently active graph for this query (Hartig, Taelman, et al., 2024). The query returns all requested parameters for all sub-graphs that matched successfully against the provided pattern. While a query of this simplicity is relatively easy to comprehend, the complexity rises significantly with more complicated requests.

SPARQL also supports the use of the following concepts:

- **Optional patterns**
  It is possible to define patterns that **may** exist but **do not have to** exist be for a query to match successfully.

- **Filters**
  Filters can be used to match strings with Regular Expressions (regexs) or to restrict numerical values.

- **Negation**
  Two types of negation are natively available through the use of filters: **EXISTS**/ **NOT EXISTS**, which filters based on the existence of patterns and **MINUS**, which removes possible solutions from the result.

- **Aggregation**
  This feature allows directly returning aggregate values in the query results. This includes building sums, averages, counts and more. Obtaining aggre-

[8] http://xmlns.com/foaf/spec/, accessed: 10.08.2024

Algorithm 2.2: Simple SPARQL query.

```
1
2  DATA
3  PREFIX foaf:  <http://xmlns.com/foaf/0.1/> .
4  _:a  foaf:name   "Johnny Lee Outlaw" .
5  _:a  foaf:mbox   <mailto:jlow@example.com> .
6  _:b  foaf:name   "Peter Goodguy" .
7  _:b  foaf:mbox   <mailto:peter@example.org> .
8  _:c  foaf:mbox   <mailto:carol@example.org> .
9
10 QUERY
11 PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
12 SELECT ?name ?mbox
13 WHERE
14 { ?x foaf:name ?name .
15   ?x foaf:mbox ?mbox }
16
17
18 RESULT
19 "Johnny Lee Outlaw"      <mailto:jlow@example.com>
20 "Peter Goodguy" <mailto:peter@example.org>
```

gated results is a vital functionality for several userscenarios. Especially in the AEC industry, where tasks like *quantity take-off* play a major part.

- **Create, Update and Remove RDF graphs**
  Through the addition of SPARQL update, it is possible to perform all Create, Read, Update, Delete (CRUD) operations on a graph store using SPARQL. Besides reading it is therefore possible to create, update and delete resources through the query language directly.

Given these functionalities, SPARQL is a very powerful query language. Its expressiveness allows it to retrieve arbitrary sub-graphs through the aforementioned pattern matching.
There are, however, limitations to SPARQL as a query language that hinder its effectiveness and practical use in some scenarios. The first one is especially relevant for the AEC industry with its diverse data formats and exchange requirements.

1. According to Harris and Seaborne (2013) SPARQL "*...can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware*". The resources have to be accessible in one way or another in the form of RDF graphs. This is not an unusual limitation for query languages, as, for example, SQL is also dependent on the data being represented as a relational, table-based data store. It

shows, however, one of the strengths of GraphQL as a query language in comparison. GraphQL is not dependent on a specific data format but can be used to query diverse data stores.

2. SPARQL is associated with a steep learning curve (Werbrouck, Senthilvel, Beetz, Bourreau, et al., 2019) and is considered to be cumbersome to implement for developers not completely familiar with the Semantic Web and its technologies. If the target resource data is related to BIM it will most likely be in the IFC data format, specifically using ifcOWL or equivalent ontologies to store the information as an RDF graph. Familiarity with these concepts (which are complicated and complex on their own) is therefore additionally required to construct meaningful and consistent queries Guo et al. (2020). This highly limits the potential use of SPARQL even though it is a very powerful language. The threshold for developers to implement SPARQL queries and interfaces is therefore hindering its usability.

2.2.2 Semantic Web Technologies in the Context of BIM

As a response to the complexity of not only SPARQL, but other query languages as well, Guo et al. (2020) have proposed a method to automatically create SPARQL queries from the data requirements of the users. They use keywords provided by the user in conjunction with string matching to construct the actual SPARQL query. The query is executed on IFC data, which must be converted into a RDF representation beforehand (because of limitation number one).
A different approach is proposed by Zhang et al. (2018) by extending the SPARQL vocabulary in order to simplify querying of building data.
Through the use of Linked Data in the the context of building models, *„... resources can be linked to each other not only within the realm of a single model or file, but across file boundaries and networks“* (Pauwels et al., 2018) . This opens up the possibility of aggregating a model from several distributed partial models. Each discipline can work on their design separately, while sub-models are combined to form the meta-model through links (Pauwels et al., 2018).
The Semantic Web stack additionally offers a suite of well established software, tools and query engines already tested in other industries, that could be used in the construction environment.
Pauwels and Terkaj (2016) highlight the similarities between IFC as a schema specification and ontologies expressed in OWL built on RDF. Unsurprisingly, a lot of research is focusing on the integration of BIM into the Linked Data world. Since building models are usually exchanged using the IFC format, there are multiple approaches to map the IFC schema into an ontology represented in RDF.

The following two sections will examine two approaches of representing building information in RDF, using dedicated ontologies.

2.2.3 ifcOWL

Due to the similarities between OWL and EXPRESS as data modelling languages and the potential benefits, that Semantic Web technologies can yield for the AEC industry, there have been a lot of attempts to create a mapping from the IFC schema to ontologies expressed in OWL. Pauwels and Terkaj (2016) provide an extensive overview of the existing approaches and also present their own definition of an ontology for the IFC data model. This ontology, named ifcOWL, seems to see the most use and recognition compared to the other alternatives. Their approach is to follow the original IFC specification defined in a specific IFC schema as closely as possible to be able to transform model data from IFC to RDF and vice versa without any loss of information.
A detailed mapping between concepts, data types, entities, rules and so forth, between IFC schema and OWL is created to mirror the structure of a building model expressed through IFC in RDF. Based on this ontology an implementation of the conversion procedure between IFC and RDF has been developed and published by the author Pauwels (2023). This tool takes a IFC model serialized in the SPF format (ISO 10303-21, 2016) and outputs the RDF representation serialized as turtle file (Prud'hommeaux & Carothers, 2014). One of the main strengths of transforming IFC into ifcOWL is the possibility of seamlessly connecting the building model with information that would otherwise be out of scope, e.g., additional information about materials, sensor data or GIS data (Pauwels et al., 2017). After conversion, the resulting RDF graph can be queried using SPARQL. SPARQL, as the recomended query language to interact with Linked Data will be discussed in more detail in the following section (2.3).

Although this approach is very promising, it does not seem to be widely used in practice. On their official website buildingSMART advises against the use of ifcOWL in practice with the remark that it "*... was an acadamic experiment that concluded that IFC is difficult to represent in other formats than EXPRESS.*" As a reason they state that ifcOWL is "*full of exceptions and particularities*" (buildingSMART, 2024). Werbrouck, Senthilvel, Beetz, Bourreau, et al. (2019) state that the direct mapping between IFC and OWL produces a very large and complex ontology that, therefore, lacks the flexibility to deal with topics outside the defined schema, which is usually a building stone of the Semantic Web concept. Rasmussen et al. (2021) mention that adhering strongly to the IFC schema defined in express produces a very complex ontology that is inconsistent with best practices

established in the context of Linked Data and the Semantic Web, which leads to the inefficient reasoning. This also means that existing tools to interact with Linked Data can not be used to their full potential. Rasmussen et al. (2021) also note that by converting the entire IFC schema into a single ontology, modularity and extensibility (usually major strengths of Linked Data) are not incorporated in the design. This requires a user to handle the entire ontology even if only a few concepts are needed for a specific use case.

While ifcOWL may not be used by the industry for the reasons mentioned above, the development of numerous mappings between IFC and OWL still shows a need for technologies that allow to interconnect heterogeneous resources and enable efficient and powerful querying.

2.2.4 Building Topology Ontology: BOT

As a response to the shortcomings of ifcOWL, an alternative ontology was created in collaboration with the original author of ifcOWL, presented by Rasmussen et al. (2021). This ontology aims to provide a minimal set of definitions that are necessary to describe the topology of a building. It is intentionally kept as lean as possible with the idea of being easily expandable with specific domain ontologies, wherever required by the use case. To put the extent into perspective: ifcOWL consists of 1331 classes and 1599 properties, while the proposed Building Topology Ontology (BOT)consists of only 7 classes with 14 properties. As a result it will likely not be specific enough to be used on its own for the majority of specialized disciplinary use cases. However, it provides an easy to understand framework to describe the bare necessities of the building topology and include additional specialized ontologies from other domains, wherever necessary. This approach facilitates reliance on already existing (and potentially established) ontologies from other disciplines, thereby increasing interoperability and avoiding inconsistencies between domains.

2.3 Interfaces to Access Heterogeneous Building Data

There is a need for interfaces to access heterogeneous resources that enable seamless interoperability between diverse stakeholders. This need is prevalent across numerous industries but is especially crucial to the field of Architecture Engineering Construction (AEC). Construction projects are characterised by the

collaboration of a wide range of different project partners, using various software tools and data management solutions. Since the technologies discussed above are not exclusive to the AEC industry, however, it may be beneficial to also consider existing solutions in other areas of expertise. Given that this thesis focuses on web-based technologies, the review will be limited to web-based data exchange and access mechanisms. Special emphasis will be placed on Linked Data and related technologies, such as RDF, OWL and the query language SPARQL.

GraphQL is often viewed as an alternative or competitor to established Linked Data technologies. Therefore, it is therefore important to explore the capabilities and limitations of Linked Data in the AEC industry and identify any shortcomings that might be mitigated using GraphQL.

Costin and Eastman (2019) explore the requirements and mechanisms enabling interoperability in the context of "*smart and sustainable urban systems*" in combination with Internet of Things (IoT). They emphasize the need for interoperability with its enormous potential in reducing costs. A lack of interoperability leads to redundant processing and storage of resources, which increases project costs and time investment. The greater the role of collaboration, the more critical this issues becomes. As a potential solution to this problem, they identify the use of APIs. Lastly, they state the need for the possibility to exchange data in order to be able to integrate different systems.

The official query language for accessing Linked Data stored as RDF triples is SPARQL. SPARQL is a powerful query language but is considered cumbersome and complex to work with (Werbrouck, Senthilvel, Beetz, Bourreau, et al., 2019). There is a signigicant barrier for developers to implement SPARQL and include it into their projects (Verborgh, 2018). As a result, the Linked Data stack offers powerful capabilities, but its adoption on a larger scale is limited by the steep learning curve of the associated technologies.
When discussing API design, one important aspect is accessibility. An API has to be integrated into other products, to be considered useful. Lowering the entry barrier to key technologies is, for this reason, more important than it may initially seem. It is therefore, understandable that a lot of focus is placed on exploring and developing more user-friendly ways of interacting with Linked Data or interconnected data in general.

Werbrouck, Senthilvel, Beetz, Bourreau, et al. (2019) undertake an extensive comparison between SPARQL and two GraphQL-based technologies: HyperGraphQL and GraphQL-LD. They apply all three approaches to query a Linked Building Data model and compare them according to criteria like federated querying, reverse querying, updating functionality and so on. Both GraphQL-LD and HyperGraphQL

are technologies that extend the GraphQL specification to support querying (and mutation) of Linked Data. The approaches of both technologies are, however, quite different.

GraphQL-LD, introduced by Taelman et al. (2018), translates the incoming GraphQL query into valid SPARQL and performs a traditional SPARQL query on the resources. As a result, there is no need to set up a GraphQL schema, which is generally a key part of any GraphQL API (ref GraphQL). This schema-less setup also means that there is no introspection feature (ref introspection), which normally communicates to the user of the API (developer) which queries (types and fields) are available. This is because GraphQL-LD enables the user to query the entirety of the existing resources as long as they are available through SPARQL. GraphQL acts in this case only as the query language the client uses to communicate with the API, simplifying the communication compared to SPARQL. Consequently, there is no need to set up a GraphQL server; the translation from GraphQL to SPARQL can happen on the client application.
HyperGraphQL, on the other hand, adheres more closely to the original GraphQL specification. A schema is created (automatically) and is linked using a meta-model to the Linked Data resources. This means every type and field has to be mapped to a corresponding Linked Data URI. This approach offers GraphQL features like introspection and allows the API developer to control access to the underlying data in a fine-grained manner.
A similar approach to HyperGraphQL can be observed in existing proprietary software products like TopBraid[9] and Stardog[10] (Taelman et al., 2019). These publications show that GraphQL is a popular choice for querying RDF triple stores instead of using SPARQL, the officially recommended query language. Taelman et al. (2019) identify and illustrate four different approaches to access RDF using GraphQL.
When considering GraphQL as a alternative to SPARQL, it is important to note that GraphQL is not limited to resources described in RDF. The use of flexible resolver functions allows for the integration of resources from numerous sources and with very diverse structure. GraphQL lacks on the other side some of the flexibility and expressive power that come with SPARQL as a query language.

One of the limitations of Linked Data in the context of AEC projects is the lack of an innate access control mechanism. The web of data and associated technologies are designed to form an open, accessible web of resources where it is possible to directly follow the links from one resource to another. However, access control is a vital requirement for projects in the AEC industry to ensure the data sovereignty

[9] https://topbraidcomposer.org/html/, accessed: 10.07.2024
[10] https://www.stardog.com, accessed: 10.07.2024

and property of the stakeholders. Restricting access to resources is not inherent in the principles of Linked Data and has to be implemented separately. Villata et al. (2011), for example, present a method of implementing fine-grained access control by creating a dedicated ontology and enforcing it using SPARQL queries. Kamateri et al. (2014) propose a access control framework for Linked Data in the context of medical information, which is highly sensitive regarding privacy concerns. By comparison, REST APIs have access control mechanisms that are exceptionally mature and thoroughly validated in practical use.
Since GraphQL was developed on the basis of the established REST paradigm, its access control mechanism can be adapted from REST. This means that it is possible to draw back on authorisation and authentication mechanisms that have been developed, evolved and proven to be secure and reliable in practice.

When GraphQL is compared to other query languages, a brief mention of SQL is warranted since it is a very popular query language. There are approaches for utilizing SQL and relational Databases in the context of BIM. Several resources in the AEC context can be represented very well in form of a relational database. Building model data, however, is usually described in an object-oriented manner with entities, properties, associations and so on, that are not easily representable in an table-based relational manner (Li et al., 2016). In recent years, there has been active discussions in the technical community about storing IFC data in a relational database[11], as well as the development of an appropriate database schema[12]. One proposal is given with ifcSQL, a SQLite database schema for IFC by Bock and Eder (2024). However, buildingSMART classifies this encoding of IFC data as "experimental/unsupported" in contrast to e.g., the RDF variants *.ttl* and *.rdf*.

2.4 Summary and Research Gap

This literature review explores key topics related to collaboration practices and data exchange in the AEC industry. Specific focus is on the application of Linked Data technologies (indluding RDF, OWL and SPARQL) as well as the current and potential use of GraphQL as an interface for accessing heterogeneous building data. This addresses research questions 1 and 2 posed in chapter 1:

- What is the current state of research on fine-grained data access to heterogeneous, highly interconnected knowledge within the AEC industry?

[11] https://community.osarch.org/discussion/1535/ifc-stored-as-sqlite-and-mysql, accessed: 01.08.2024
[12] https://forums.buildingsmart.org/t/ifc-for-relational-databases-ifcsql/1524

- Have previous studies explored the application of GraphQL in the AEC domain?

The review highlights a significant interest in applying Linked Data technologies within the built environment. RDF and OWL are essential building blocks in representing and linking heterogeneous resources. Several methodologies have been proposed for converting IFC to OWL, enabling more seamless data integration and interoperability. Despite the advantages, using SPARQL for querying, Linked Data poses challenges, particularly due to its complexity. These limitations suggest the need for more intuitive and user-friendly querying solutions.

The centralization of knowledge as a single source of truth, using technologies like CDEs or model server, was also identified as a possible approach to enable and improve collaboration and data exchange in the AEC industry. Even though these are established solutions in the industry, there are still difficulties, especially if information exchange between different platform solutions is necessary. Additional issues arise from file based data exchange, which does not offer fine-grained access to information and is, however, still prevalent in many large scale projects.

GraphQL is emerging as a promising alternative for querying linked building data. Its flexibility and efficiency in managing complex queries position it as a viable solution to some of the shortcomings associated with SPARQL. The literature indicates some recognition of GraphQL's potential to enhance data accessibility and user experience in the built environment sector. Compared to the rising popularity of GraphQL there are, however, surprisingly little attempts to integrate this technology in the AEC industry. Most approaches are based on using GraphQL as an interface to access Linked Data representations (Taelman et al., 2018; Werbrouck, Senthilvel, Beetz, & Pauwels, 2019). One of the few examples, where GraphQL is used outside the Semantic Web context is given by Clemen et al. (2021), who developed a methodology to enable the integration of standardized data catalogues into software tools through GraphQL. However, the simultaneous integration of different, diverse data structures and knowledge representations, which is one of the strengths of GraphQL, is missing from the literature in this area. As GraphQL gained significant traction in recent years, there are plenty of examples regarding the application of GraphQL, but far less research in the context of BIM and CDE. Although, some methods and technologies might be transferable from other areas, the AEC industry with its specific characteristics might require unique solutions.

Most of the current research that combines BIM with graph-based technologies focuses on topics like model coordination (Zhao et al., 2020), version control

(Esser et al., 2022) or model-to-graph conversion (Tauscher & Crawford, 2018; Zhu et al., 2023).

This thesis aims to fill this gap by proposing a methodology utilizing GraphQL to query heterogeneous information in the context of BIM. This is done by implementing a prototypical GraphQL API and testing it against specific predefined user scenarios, exploring the advantages and disadvantages of GraphQL in the AEC industry.

Chapter 3

Basics of GraphQL

Since the following sections – the descriptions (chapter 4) of the approach and the implementation of a GraphQL API (chapter 5) – require an understanding of GraphQL's terminology and core concepts, this chapter provides an introduction and overview of GraphQL as a query language and API architecture. GraphQL can be seen as a response to several shortcomings of the REST paradigm for specific use cases, making it valuable to examine its similarities and differences towards REST. Before delving into GraphQL, a brief introduction of REST is warranted, which is still the most used API paradigm.

3.1 Application Programming Interfaces

An API is a component of a software that exposes data and functionality through an interface to other applications. It enables "interactions between computer programs and allow them to exchange information" (Masse, 2011). Since this thesis focuses on web-related context, API specifically refers to web API. A web API is an API that is accessible via the internet. Most current web APIs implement either the REST or GraphQL paradigms.

3.2 REST

Published by Fielding (2000), REST remains the predominant architectural paradigm for web APIs to this day. According to the *Postman State of the API Report*[1], REST is leading the list of most-used API architectures by a large margin, ahead of webhooks, GraphQl, SOAP and WebSockets. In order to be considered RESTful, an API needs to adhere to the following principles (Fielding, 2000):

1. **Client-Server relationship**:
   REST defines the relationship between a client and a server. The client initiates the conversation by sending a request to the server, which then sends back the appropriate response.

[1] https://www.postman.com/state-of-api/api-global-growth/, accessed: 28.06.2024

2. **Stateless**:
   All the information necessary to respond to a request must be included in the request itself. The server does not keep track of information about the state of a client session. This ensures scalability and enables a federated server architecture. There is no need for a specific client to communicate with a specific server for subsequent requests, since a request is independent of previous requests.

3. **Cache**:
   RESTful APIs must be cacheable: The response to a request is saved. If the same exact request occurs again, this saved response is send instead of generating a new response. Implementing this behavior has several benefits, such as reducing server load, minimizing network traffic and hiding network failures. Caching can be implemented on both the client and on the server side.

4. **Uniform Interface**:
   This constraint describes the use of URLs to identify and address resources. Every resource available through a REST API has a specific URL, through which it can be identified and accessed. It further demands that requests be self-describing, for example, through the specification of Hypertext Transfer Protocol (HTTP) methods.

5. **Layered System**:
   The API should support or implement a layered architecture. This increases scalability and simplifies the implementation of, e.g., security features.

Following the fourth constraint, each request must specify a *HTTP* method to inform the server of the intent of the request. The methods commonly recognized by APIs are *GET, POST, PUT* and *DELETE* which correspond to the already mentioned basic Create, Read, Update, Delete (CRUD) operations.

3.2.1 Limitations of the REST Paradigm

REST is extremely popular as an API architecture and is one of the integral building blocks of the web as we use it today. There are, however, limitations that have led to the development of alternative API architectures better suited for specialized scenarios. These limitations are discussed in more detail in the following section, since GraphQL directly addresses some of these issues.

Over- and Under-Fetching

When using a REST API, there is a high possibility that the information provided

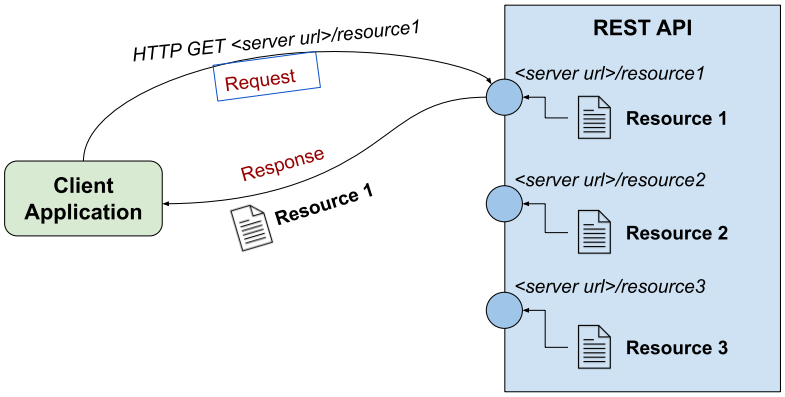

Figure 3.1: General structure of the client-server relationship using a RESTful API.

by a specific endpoint may not be needed in its entirety by the client. Frequently, only parts of a resource are required for a specific client application. Since every resource is accessible through a specific URL, there is no possibility for the client to request only parts of a resource; the resource is always returned in full. As a result, more information than necessary gets exchanged between client and server. This leads to unnecessary network traffic and the need to implement additional logic on the client side to filter the information. This is known as over-fetching.
On the other hand, if a client needs more information than what is available through a single endpoint, multiple independent requests to different endpoints must be sent, and the responses must be merged on the client side. This phenomenon is called under-fetching.

A small example to illustrate this issue: the World Time API[2] provides information about the current time based on a specific timezone with an endpoint for each timezone. A HTTP GET request to the endpoint http://worldtimeapi.org/api/timezone/Europe/Berlin will return information about the Central European Time (CET), including the current time and date (for this time zone) in different formats, current week number, a Coordinated Universal Time (UTC) offset and more. The actual returned result is depicted in listing 3.1. If the client application is only reliant on, e.g., the "datetime" parameter, all the additional information would be superfluous (over-fetching). If, on the other hand, the client application needs to include information about multiple time zones, a separate request must be made for each individual time zone (under-fetching).

This is a very small example and the amount of over-fetched data can be much higher in practice. However, the principle remains the same: it demonstrates that often there is no exact match of the client need and the provided endpoints.

[2] http://worldtimeapi.org, accessed: 30.07.2024

Algorithm 3.1: Response body from the WorldTimeAPI endpoint http://worldtimeapi.org/api/timezone/Europe/Berlin

```
{
    "abbreviation": "CEST",
    "client_ip": "xxx.x.x.xxx",
    "datetime": "2024-07-30T08:47:05.403089+02:00",
    "day_of_week": 2,
    "day_of_year": 212,
    "dst": true,
    "dst_from": "2024-03-31T01:00:00+00:00",
    "dst_offset": 3600,
    "dst_until": "2024-10-27T01:00:00+00:00",
    "raw_offset": 3600,
    "timezone": "Europe/Berlin",
    "unixtime": 1722322025,
    "utc_datetime": "2024-07-30T06:47:05.403089+00:00",
    "utc_offset": "+02:00",
    "week_number": 31
}
```

Multiple Endpoints

Efforts to avoid over-fetching and under-fetching typically result in the definition of numerous endpoints to ensure fine-grained access to the underlying data. However, this can lead to a certain level of complexity within the API, making the system harder to maintain and understand. Developers must carefully manage and document these endpoints to avoid potential issues and inconsistencies. Additionally, balancing the granularity of data access with performance considerations remains a significant challenge.

Multiple Roundtrips for Interconnected Resources

Since every resource has its own separate endpoint, interconnected resources can not be requested in a single request. If a resource references a different resource, this reference will be included in the response to the initial request. The referenced resource must then be requested in a subsequent request, resulting in one additional round trip between client and server for each level of nested references. Assuming a server provides two resources, A and B, each with its own endpoint. Resource A references resource B through the URL at which B is accessible. To illustrate: Resource A could be a JSON file with information about an author. This file contains a list of URLs referencing all the books associated with the author. Resource B could be a JSON file containing information about a specific book written by the aforementioned author. The communication flow to retrieve information about the author and their book would be as follows:

1. Client sends a request to endpoint A

2. Server responds with resource A
3. Client processes the response and extracts the reference to resource B from resource A
4. Client then requests resource B
5. Server responds with resource B

This communication includes two round trips between client and server at the level of the REST API communication (excluding HTTP-specific overhead in the communication such TLS handshake to establish encryption among others). The resulting communication flow between client and server, in the general case of related or referenced resources, is depicted in figure 3.2. For highly interconnected resources, this issue can lead to significant network traffic and complex logic that must be implemented on the client side in order to access all the required resources, making the client-server interaction complicated and time-consuming.

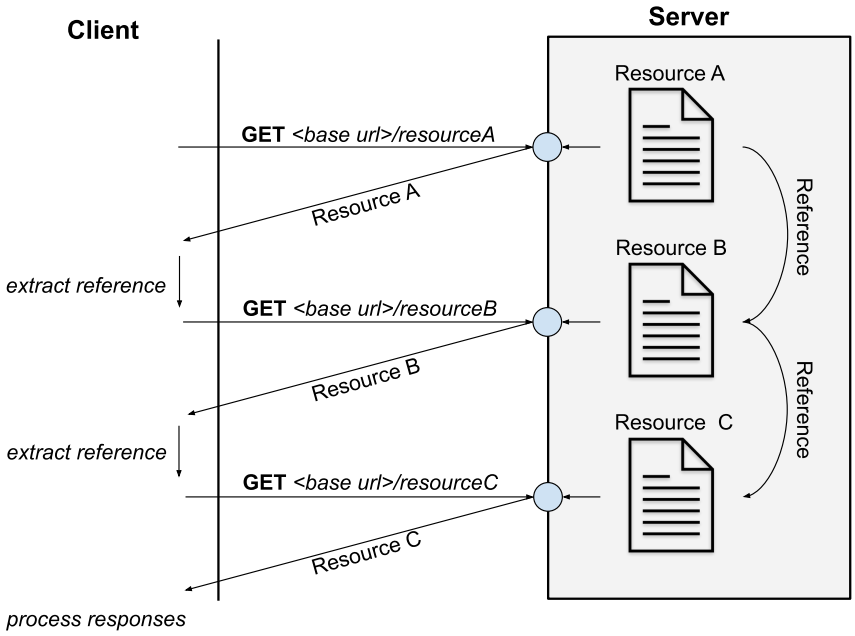

Figure 3.2: Requesting related resources using a REST API

3.3 GraphQL Overview

GraphQL is both an API architecture paradigm and a data query language (GraphQL, 2021). It was developed by *Facebook* and made available to the public[3] in 2015. It has gained significant attention since and quickly established

[3] https://github.com/graphql, accessed: 05.07.2024

itself as a respectable alternative to the most prevalent API architectures. It offers unique advantages over alternatives and is used by many mayor companies to manage and structure access to their data. The next sections will demonstrate how GraphQL works as an API architecture and query language.
The following sections are based on the official GraphQL specification (GraphQL, 2021), the GraphQL documentation[4] as well as documentations of established GraphQL frameworks like Apollo[5] and Ariadne[6].
GraphQL can be used to support all CRUD operations, using the default types query (to read data) and mutation (to create, update and delete data). The API developed in the context of this thesis will only support queries and omits the implementation of mutations, similar to other examples from the literature (HyperGraphQL, 2021; Taelman et al., 2018). Mutations will be covered in less detail in the following introduction to GraphQL since they are less relevant for this specific project. It is, however, possible to extend the presented API in the future to additionally support making changes to the data, instead of only reading the resources.

3.4 API Structure

GraphQL (similar to REST) provides a specification for the structure of a client-server relationship. The communication is initiated by the client as a request. The server listens for incoming requests, processes them and sends a response to the client.
While GraphQL is not bound to a specific protocol, it is predominantly served over *HTTP*. Implementations should support the *HTTP* methods *GET* and *POST* according to the official best-practice recommendations[7]. In both cases, a GraphQL query will be included as payload. When using *GET*, the query is included as a query parameter named *query*. The query is encoded as a string and passed to the server as part of the URL. Using the *POST* method, the GraphQL query is included in the request body. The content type should be set to *application/json* in this case.
The main difference between GraphQL and REST lies in how resources are identified and accessed. While REST provides URLs for resources, GraphQL describes the available resources as an entity graph (see section 3.6). This entity graph has

[4] https://graphql.org, accessed: 14.08.2024
[5] https://www.apollographql.com/docs/, accessed: 14.08.2024
[6] https://ariadnegraphql.org/docs/intro, accessed: 14.08.2024
[7] https://graphql.org/learn/best-practices/, accessed: 13.08.2024

a single root node for querying (called the *Query* type), which serves as the entry point to the graph. As a result, the GraphQL API is served over a single endpoint.

GraphQL, as an API paradigm, shares a lot of similarities with REST and is therefore easy to integrate into existing REST API ecosystems. GraphQL is usually served using the same communication protocol (HTTP) as REST. There are caching mechanisms for GraphQL and GraphQL follows the constraint of statelessness that is a requirement for REST APIs (for a description, see section 3.2). Specific methods of combining GraphQL with REST APIs are illustrated in section 3.5.

GraphQL offers the following distinct advantages over alternative technologies and specifically REST APIs:

- The required resources can be specified with fine-grained precision, which avoids over-fetching.
- Multiple resources can be requested and combined using a single query on the client side, avoiding under-fetching.
- Related resources can be requested in a single request by following links in the schema directly.
- The client application is able to control the extent and structure of the response.

3.5 Combining GraphQL with REST API

The GraphQL architecture is not necessarily a competing technology with the REST paradigm. Most companies that are implementing GraphQL seem to be using a combination of both technologies for different tasks[8]. Apollo, a company that provides one of the most extensive and widely used implementations for GraphQL, suggests a method for combining both paradigms in their official documentation. The recommended approach is to use REST APIs for backend data services and implement GraphQL as a middle layer between backend services and frontend applications. Following this approach, GraphQL is used as a common abstraction layer for all frontend applications that aggregates diverse backend services into a single endpoint.
Another approach that can be observed in the industry is to provide a GraphQL

[8] https://www.apollographql.com/docs/technotes/TN0044-graphql-and-rest-together/, accessed: 28.06.2024

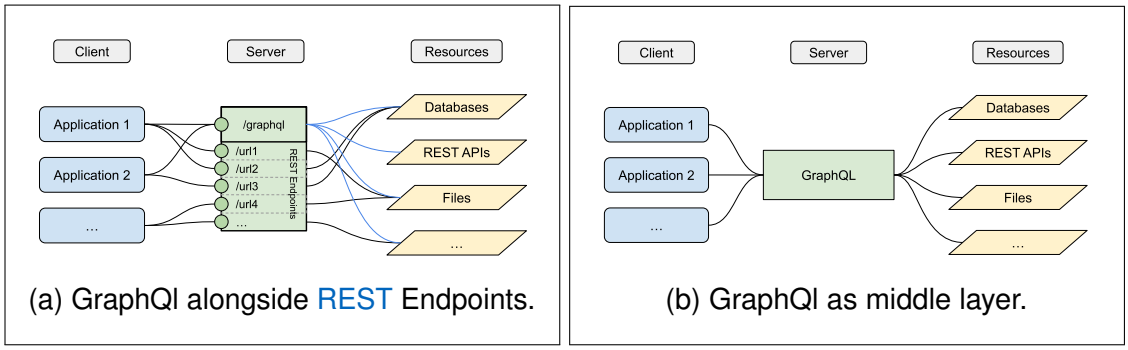

Figure 3.3: Combining GraphQl and RESTful Endpoints

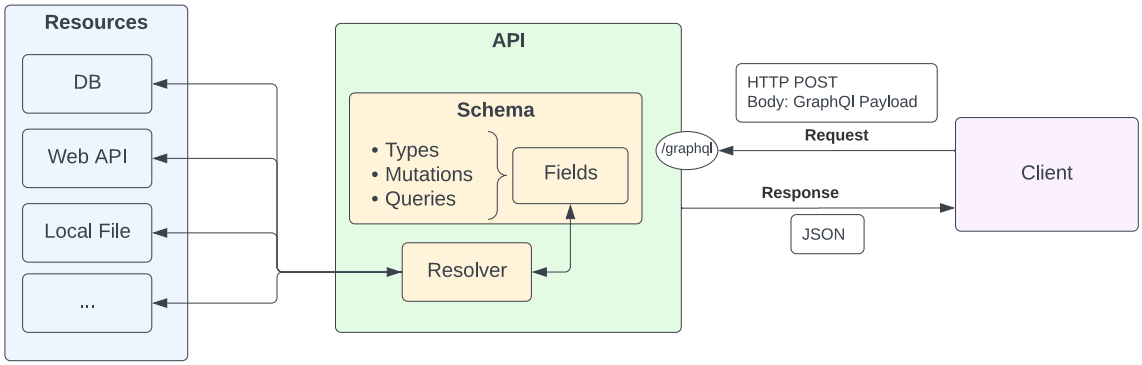

Figure 3.4: Structure of a GraphQL API.

endpoint alongside REST endpoints. In this way, GraphQL can also be integrated into existing REST APIs, either providing a different, powerful method to access the already provided resources or extending the APIs functionality. Both integration approaches are illustrated in figure 3.3.

3.6 Schema

At the heart of a GraphQL API implementation stands the GraphQL schema. The schema is usually written in GraphQL's own language and defines the structure of the resources that are available through the API. The resources are described in an object-oriented approach and follow a graph-like structure, where parameters of one object can link to another object. Although it is possible to model the GraphQL schema following the existing structure of the resources, the schema does not need to adhere to the structure of the underlying knowledge representations. A predominant design philosophy in the community is to develop the schema based on the user scenarios rather than the underlying resource structure.
This philosophy ensures that the API does not provide services or expose resources that are never required by any real use case. By focusing on actual requirements, the API remains lean and efficient, avoiding the overhead of un-

necessary functionality. As a consequence, the exact amount and structure of resources exposed are tailored to meet the specific needs of the client application, optimizing network performance and the utilization of resources. However, following this philosophy often requires manual designing the schema.

The resources are linked to the schema in an extremely flexible manner, allowing the restructuring, restriction, expansion and combination of different resources into a single GraphQL API. This flexibility enables the API to be tailored precisely to the needs of the expected use cases.

One important design decision is determining, how closely the resource structure and the GraphQL schema should be coupled. There are two general approaches:

1. Modeling the schema strictly following the structure of the resources: In this approach the schema is deduced from the schema underlying the existing resources. There needs to be a mapping from the data modeling language that describes the resources to GraphQL.
   Advantages:

   - The process can be automated.
   - No consideration is necessary in designing the schema, since it will be deduced from the resource schema.

   Disadvantages:

   - Can lead to a excessively bulky schema that supports types and fields that are not required by any use case.
   - For an automatic approach, there must be a consistent and uniform definition of the structure of the resources.

2. Align the schema with the needs of the user and use cases: This approach requires manually designing a schema depending on the proposed use case of the API. Consequently, it can produce a schema that is leaner, clearer, more concise and more user-friendly.

3.6.1 Types

In GraphQL, the class-like entities mentioned above are referred to as *"types"*. Every type has one or more fields, with each field having a defined name and data type. This type system corresponds directly to the concept of OOP, implemented in several programming languages. Types can be compared to classes in this context, while fields can be seen as their member variables. Following

Algorithm 3.2: Data Modelling in GraphQl

```
type Wall {
    globalId: ID!
    name: String
    volume: Float
    material: [Material]
}
```

this analogy, there are implementations of GraphQL frameworks that make use of this similarity by allowing the developer to design the schema using classes from the framework's programming language, instead of using GraphQL itself as a language to describe the schema. This approach is called *"code-first"* in contrast to the *"schema-first"* approach. In Python, a popular framework for developing GraphQL APIs *code-first* is Graphene[9]; there are, however, implementations of *code-first* approaches available in almost any major programming language that offers GraphQL implementations. This thesis, however, follows the *schema-first* approach, using GraphQL as a language to formulate the schema, including all type definitions. The data type of a field can be one of the following:

- One of the scalar data types supported by GraphQL: Int (32-bit integer), Float (signed double-precision), String (UTF-8), Boolean or ID (similar to a String but not supposed to be human-readable).
- A different custom GraphQL type defined somewhere in the schema.
- An enumeration (3.6.2).
- A list containing either a scalar data type or a custom GraphQL type. However, lists can only contain a single type.

All fields can be marked as required, including lists as well as the content of the list itself. It is convention to name the types in GraphQL, starting with an uppercase letter, while field names start with a lowercase letter. The following sections adhere to this naming convention, which means, for example, *"Author"* refers to a type, while *"author"* refers to a field.

3.2 shows a simplified example of an *IfcWall* type being modeled in GraphQL that showcases different available language concepts. The field *'globalId'* is required, signified by the *'!'*. A query is not allowed to return *'null'* for this field. The fields *'name'* and *'volume*' refer to available scalar types, while *'predefinedType'* refers to a custom type that has to be defined elsewhere in the schema definition. *'Material'* refers to a list of the custom type *'Material'*.

[9] https://graphene-python.org, accessed: 05.07.2024

Algorithm 3.3: Enumeration in GraphQl

```
enumeration IfcWallTypeEnum {
    SOLIDWALL,
    STANDARD,
    POLYGONAL,
    ELEMENTEDWALL,
    RETAININGWALL,
}
```

3.6.2 Enumeration

GraphQL supports the definition of enumerations. An enumeration restricts the available values for a field to a predefined selection. A query for this field must return one of the specified values (or 'null': if the field is not required). Listing 3.3 shows a simplified example of the *IfcWallTypeEnum* mentioned in listing 3.2.

3.6.3 Interfaces

Interfaces are a common concept in many object-oriented programming languages, serving as a blueprint for other classes by defining a set of common attributes and methods. In GraphQL, an interface plays a similar role. Interfaces are defined similarly to types, using the *interface* keyword. A type can then implement an interface, by including all the fields specified by that interface. For example, in Listing 3.4, both *Apple* and *Orange* implement the *Fruit* interface, meaning they must include all the fields defined in the *Fruit* interface, though they can also add additional fields.
While an interface can be used like any other type in the GraphQL schema, it can not be instantiated directly. In other words, there is no *Fruit* type that can be returned to the client – only *Apple* and *Orange* can be returned. When processing a request, the server must therefore determine whether an element that conforms to the *Fruit* interface is an *Apple* or an *Orange* before returning it to the client. This provides flexibility to the specific type that a field can have. By defining a field's type as an interface, that field can be resolved to any object that implements the interface. The concept of inheritance, which is missing from GraphQL, can also be mimicked to some extent using interfaces.

Algorithm 3.4: Interfaces in GraphQL

```
interface Fruit {
    name: String
    price: Float
}

type Apple implements Fruit {
    name: String
    price: Float
    variety: String
}

type Orange implements Fruit {
    name: String
    price: Float
}
```

3.6.4 Union Type

The *union* operator allows grouping multiple types within the schema, enabling a single field to return one of several types classified under a single *union*. When a query returns a *union* type, the result can be any one of the types included in that *union*.
Before the query result is sent to the client, the *union* must be resolved using a type resolver. This process involves determining the specific GraphQL type of the returned element. The developer is responsible for implementing this resolution logic, which can involve identifying the type by, for example, checking for unique fields associated with each type in the *union*. The example in listing 3.5 defines a query called *hardware*, which returns a list of *Hardware* elements. *Hardware* is defined as a *union* of two types: *CPU* and *RAM*. This allows the query to return a list that can include both types. Unlike interfaces, types within a *union* do not need to share any common fields.

3.6.5 Schema Federation

Generally speaking, one centralized schema is required for any GraphQL API. This schema has to explicitly include all types, queries and mutations that will be exposed to the client. All of this must be known at server start, which means that if any changes occur to the schema during operation, the server hosting the schema and GraphQL endpoint needs to be restarted. This is true for most GraphQL frameworks.
Some frameworks offer the possibility to create federated schemata to somehow

Algorithm 3.5: The union type in GraphQL

```
type Query {
    hardware: [Hardware]
}

union Hardware = CPU | RAM

type  CPU {
    name: String
    clockFrequency: Float
    cores: Int
}

type RAM {
    size: Int
    brand: String
}
```

mitigate the issues arising from the need to have one singular, complete schema at a centralized location.

Figure 3.5 depicts an example of schema federation as provided by the Apollo framework. Similar concepts are available in other frameworks as well. In this example, two schemata are merged into a so-called super-graph. This super-graph is an aggregated schema containing schemata 1 and 2, as well as metadata about routing requests to the correct sub-graph. This additional information is removed to obtain a different schema intended to be exposed to the client, forming the overall schema on which all client queries are based upon. Even though the combined schema consists of two separate sub-graphs in this example, it still functions as a single, central point for all queries. Which also means it must be constructed before a server start or reload.
This still does not provide a flexible way to dynamically integrate GraphQL schemata from different sources into a single endpoint. The main advantage this approach offers is the ability to create sub-schemata independently from each other. This is useful, for example, when two different disciplines have their own dedicated schemata and want to keep separate responsibility and accountability, while still combining both schemata into a single-combined graph. While this concept is not used or implemented in the following case study, it is included because it addresses a key aspect of collaboration in the AEC industry. Collaboration in this context is characterized by multiple different stakeholders, each working on separate parts of a bigger product. This means, several disciplines create their own sub-models, which are developed separately but must be combined into a single BIM model. Using schema federation, as mentioned above, provides

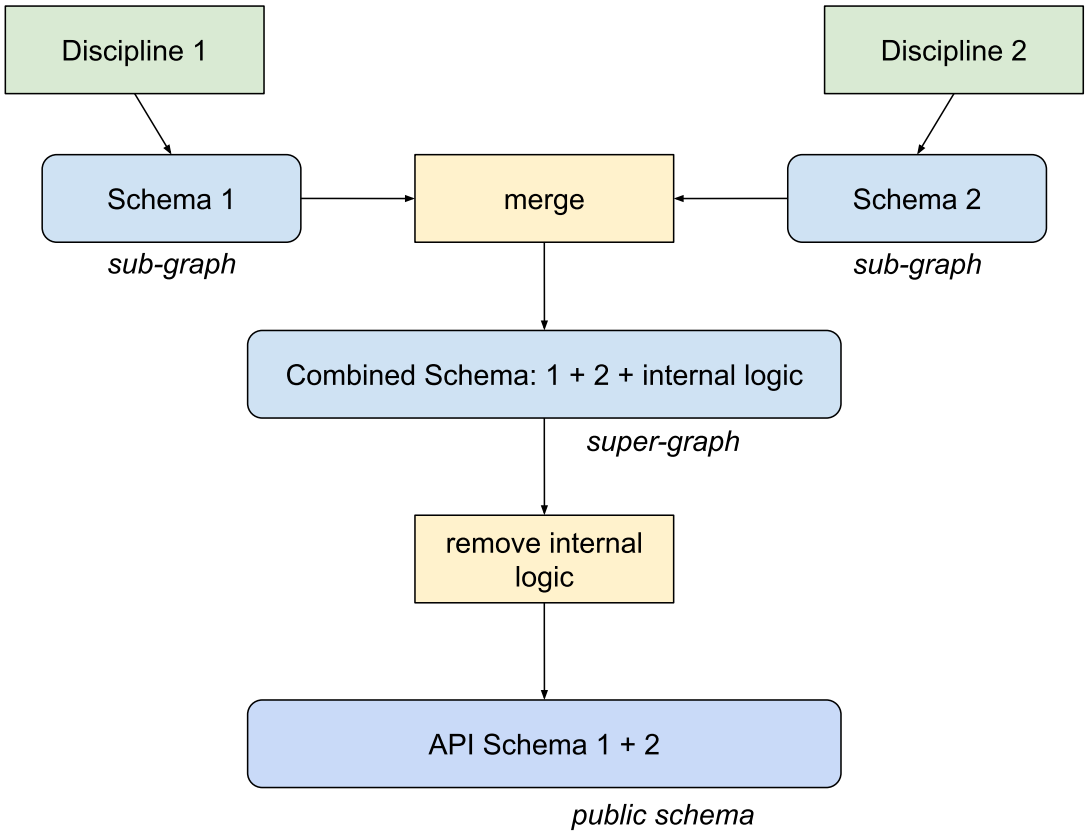

Figure 3.5: Schema federation approach as implemented by the **Apollo** framework.

the means to create sub-schemata independently and still form a combined API, following the requirement of collaboration in the AEC industry.

3.7 Queries and Mutations

Queries provide the possibility to retrieve information from the API, while mutations allow making changes to the underlying data, like creating, updating or deleting resources.
Queries are defined using the special *Query* type and every GraphQL API must have a Query type defined. It is special by providing an entry point to the GraphQL schema. Other than that, it behaves exactly like any other custom type defined in the schema. Each query requested by a client starts at this root node, which specifies further available queries as the fields of this root type, defining the name and return type of each available query.
Mutations are very similar to queries. A GraphQL API can define a mutation type, acting as the entry point to all available mutations. It is established best-practice to use a mutation for all operations that evoke a change to the server-side

resources[10]. This corresponds to the use of HTTP methods in REST APIs: even though it is possible to create a new resource as a response to a GET request, it is a common convention to use the POST method for a scenario like this.

3.8 Resolver

Resolvers are the link between the GraphQL schema and the resources that should be exposed through the API. Each field defined in the schema has a corresponding resolver function. This resolver function is called when and only when a field is requested as part of a query. Fields that are not requested are also not resolved. This modular architecture is one of the reasons why GraphQL is so successful and performant. The resolvers are organized in a hierarchical order, following exactly the structure of the entity graph defined by the GraphQL schema. This hierarchy starts at the query root node. The first resolver that is called, is the resolver defined for the specific field defined on the *Query* type that a client is requesting. This resolver then calls all resolvers next in line, providing its own result as an input. This is commonly known as the *"resolver chain"*. Since, however, there can be multiple fields requested on the same nesting level, "resolver tree" would be a better fitting description. This concept can be better demonstrated using an example. Figure 3.6 shows a typical GraphQL query requesting information about an author-based on the name of the author, including a list of all books associated with this author containing the title and ISBN for each one. The GraphQL schema for this example is very simple and depicted in listing 3.7. It consists of two types, *Author* and *Book*, and provides a single query called *author*, that returns a single element of type *Author* given a *name*.

The first resolver that is called, when this query is processed by the server, is the resolver bound to the *author* field of the *Query* type. This resolver then calls the resolver functions for the *name* and *books* fields of the *Author* type. Finally, the resolvers for *title* and *ISBN* on the *Book* type get called (in parallel) for every *Book* item in the list. Starting at the root node, the resolver functions traverse the entity graph until a leaf node is reached. A leaf node in this context is a field that can be resolved to one of the simple data type, like String, Boolean, Float and more. The corresponding hierarchical resolver structure can be seen in figure 3.6. It demonstrates the tree-like structure commonly called *"resolver chain"*.

Because of this modular structure, a specific resolver has very limited information about the query result in general. The results of all child resolver are not known during resolving of a specific field. A resolver furthermore has no knowledge about

[10] https://graphql.org/learn/queries/, accessed: 20.07.2024

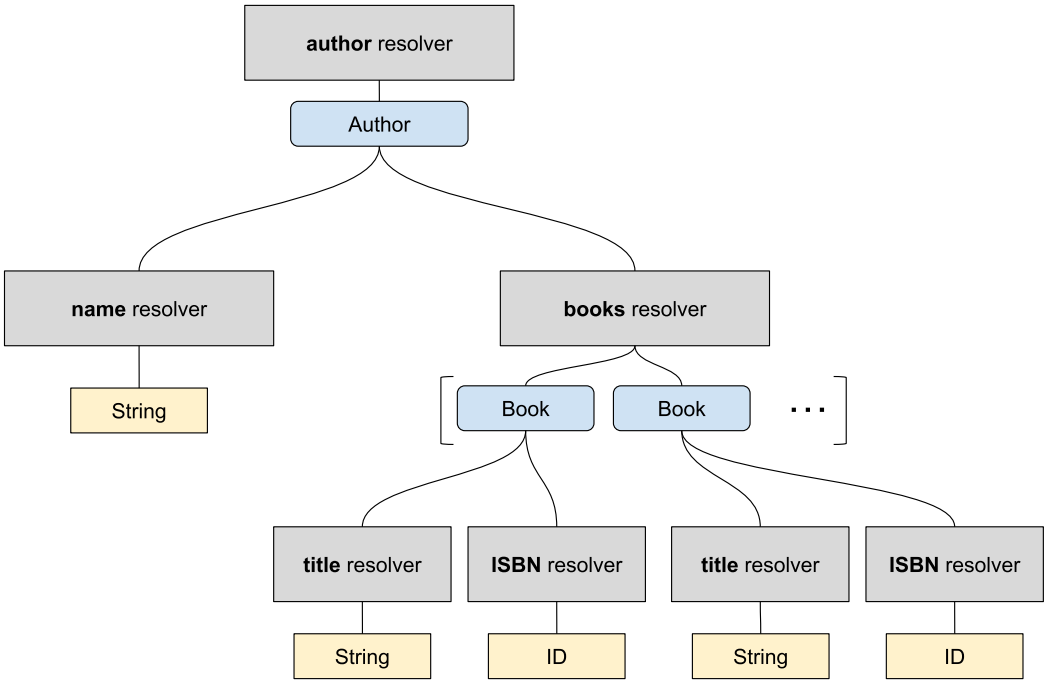

Figure 3.6: "Resolver Chain" or "Resolver Tree" originating from the query defined in 3.6.

any sibling resolvers. Both of these facts will become important in chapter 5 when it comes to the implementation of functionality like aggregation that is dependent on the result of multiple resolvers.

Algorithm 3.6: Query demonstrating the resolver chain.

```
query AuthorAndBooks($name: String){
    author(name: $name){
        name
        books {
            title
            ISBN
        }
    }
}
```

The exact structure and nomenclature of resolver functions vary between different frameworks. Resolver functions in the Ariadne[11] framework, for example, use two positional input arguments, while the same function in the Apollo[12] framework defines four input arguments. There are, however, common elements between

[11] https://ariadnegraphql.org, accessed: 13.08.2024
[12] https://www.apollographql.com, accessed: 13.08.2024

Algorithm 3.7: Simple schema.

```
type Query {
    author(name: String): Author
}

type Author {
    name: String
    books: [Book]
}

type Book {
    title: String
    ISBN: ID
}
```

all frameworks that are specified in the official documentation[13]. The following information is commonly included in the resolver arguments:

1. An object holding the result of the parent resolver.
2. A context object carrying information common to the entire query and all resolver functions. This Object can be accessed from within all field resolvers.
3. Information about the query itself, like the Abstract Syntax Tree (AST) representation of all requested fields, including all parameters the client defined.
4. Arguments provided to the field (if there are any).

Every function that adheres to this structure can be used as a resolver function by binding it to a specific field of the schema. This binding happens before server start. Most frameworks offer "*fallback*" or "*default*" resolvers to reduce the number of custom resolver functions that need to be implemented. These default resolvers are triggered when no specific resolver is defined for a field. Typically, the fallback resolver checks if the result from the parent resolver contains a key matching the field name. If a matching key is found, the field is resolved to the corresponding value. These kind of resolvers are extensively used in the case study (chapter 5), which, however, requires the parent resolvers to return objects that already contain information about the children.

[13] https://graphql.org/learn/execution/, accessed: 10.08.2024

3.9 Introspection

The introspection system is a distinctive feature of GraphQL, offering a way to explore the schema and all the queries that the API supports. Through this feature, a client can inspect the schema of a GraphQL API, reviewing all type definitions, fields, and associated data types. For example, it provides details on the interfaces a type implements and lists acceptable values for enumerations.
This capability is particularly useful during client application development, as developers can leverage introspection to discover the available queries before implementing them. This is one of the reasons GraphQL is often considered self-documenting; the schema, combined with introspection, reveals all available resources, serving a function that would typically require a formal API documentation.
However, introspection is sometimes disabled in production environments for security reasons to prevent the exposure of detailed information about the internal resource structures.

Chapter 4

Proposed Methodology

The fundamental idea is to explore the usability and usefulness of GraphQL in the context of the AEC industry. To this end, the potential of querying heterogeneous data, as typically found in construction projects, through a GraphQL interface is examined. This chapter outlines the general methodology for using GraphQL in the BIM context. An exemplary implementation of the principles discussed is presented in the next chapter. The following case study will implement both aspects, GraphQL as a query language and an API architecture. This approach ensures that the full potential of GraphQL is examined and its usefulness evaluated in all facets. This observation is crucial, as previous projects have shown the possibility of using GraphQL as solely a query language without implementing a complete GraphQL API, e.g., by translating a GraphQL query into a different query language (Taelman et al., 2018).

4.1 General Approach

The approach focuses on developing and implementing a GraphQL API at a proof-of-concept level, with an emphasis on creating a user-friendly experience centered around specific user scenarios. These scenarios are defined before the implementation process, helping to establish the scope and requirements of the API. Based on these scenarios, a GraphQL schema is designed to establish the necessary data structures for querying. Following the schema design, resolver functions are implemented to link the GraphQL schema with the underlying resources, determining how data is retrieved and returned in response to user queries. These queries are also derived from the initial user scenarios. Finally, the entire setup is integrated into a GraphQL framework, coupled with a minimal server implementation, to enable testing of the API's querying capabilities. The resulting API is then used to query heterogeneous resources within the context of the user scenarios and its effectiveness in answering the user queries is evaluated.

Two general implementation approaches were considered, which enable the use of GraphQL in the BIM context:

First Approach Candidate: Common Graph Representation

The first approach (illustrated in figure 4.1) relies on a conversion step for all resources that translates the entire data into a common graph representation. This could be a representation in RDF or any graph database. The connections that exist between different resources are then defined on the data level (e.g., integrating edges into the resulting graph). This homogenization of the data structures would make it possible to use existing solutions for graph querying. For instance, if all resources are represented in RDF, Linked Data technologies like SPARQL can be used to access the information. If the data is stored in a *Neo4j*[1] database, *Cypher* can be used for data access. In this approach, GraphQL would primarily function as a query language that allows the user to define queries, providing an easy-to-understand syntax, which would then be translated into a different query language to interact with the underlying data. A very similar approach was already presented and tested by Taelman et al. (2018) (see chapter 2). Even though they did not use different sources of information, the basic principle remains the same. This approach, however, is hindered in its usefulness by the need to represent all relevant resources in a graph representation. Although there are software tools to convert, for example, IFC into a RDF representation or import IFC models into a graph database, other resources might not be easily converted or would at least require significant additional effort during implementation. While GraphQL is used mainly because of its ease-of-use query language, its underlying mapping to Linked Data technologies means it inherits the limitations associated with these approaches. This includes the limitations of SPARQL as a query language and Linked Data in general.

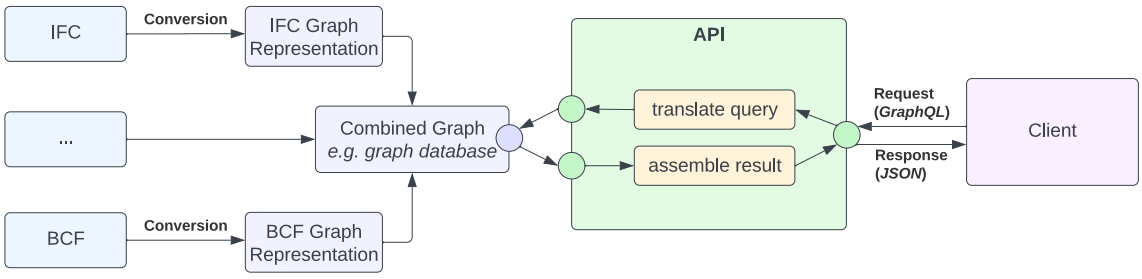

Figure 4.1: API architecture using conversion of data resources into graph representations.

Second Approach Candidate: Resource Connection on the Resolver Level

The second approach (illustrated in figure 4.2) involves implementing a more traditional GraphQL API without the need to convert resources into different data

[1] https://neo4j.com, accessed: 15.07.2024

formats. Instead, a GraphQL schema is defined to represent and reflect the connections between resources at the data level. In this way, resources are flexibly coupled using the GraphQL schema and accessed directly through GraphQL resolver functions.
These resolver functions can access resources directly or utilize intermediate interpretation layers and interfaces. This approach requires the manual design and development of a GraphQL schema. While this might be more labour-intensive for a large schema, it offers more flexibility and control over the data provided and the links between resources.

Even though both approaches seem promising, this thesis focuses on the second approach for two main reasons:

- it leverages the full strengths of a manually designed schema and can therefore be more easily tailored to specific user scenarios.
- it provides a higher flexibility in the resources that can be integrated without the need for any additional data conversion.

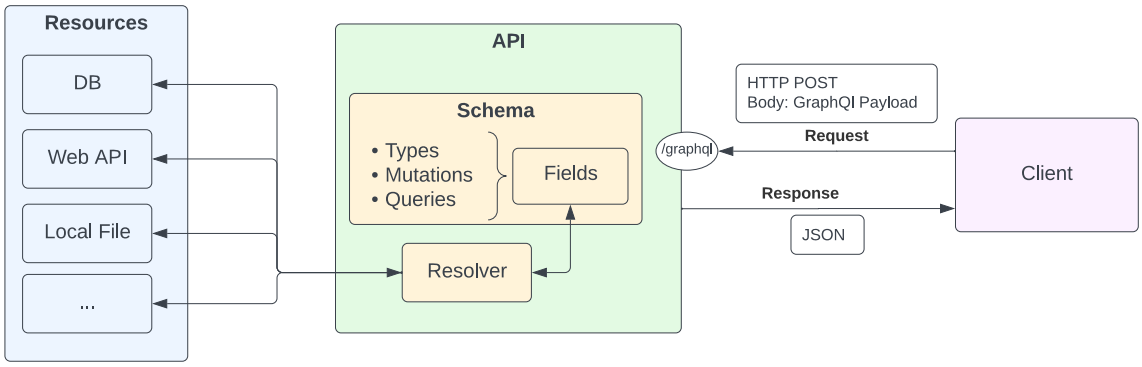

Figure 4.2: API architecture using a classical GraphQL implementation with data access through resolver functions.

4.2 Application of GraphQL for IFC and BCF

Two types of resources will be used in the case study: IFC and BCF. These data formats are widely used for information exchange in the BIM context, representing the diverse knowledge structures commonly found in the industry. Although this thesis focuses on these two data formats, the proposed methodology is transferable to other types of resources without changing the general structure and API design. IFC and BCF are chosen because they are both:

- established data formats, specifically designed to enable information exchange in the AEC industry.
- open-source data formats with thorough documentation, which greatly facilitates integrating them into custom projects.
- interconnected, which enables API design and testing against interconnected resources.

Chapter 5

Implementation

5.1 Overview

This chapter describes the actual implementation of the proposed GraphQL API. The resulting software has limited functionality and serves as a proof-of-concept to showcase the feasibility of the approach, only supporting the use cases defined in the following section. This chapter includes some code examples for better understanding of the underlying conceptual work.

The general implementation steps are depicted in figure 5.1. The initial step involves defining the user scenarios, which establishes the scope and extent of the API. This step significantly influences both, the schema design and resolver implementation. Although the flow chart lists *server and framework setup* as the next step, it is not necessary to include it at this exact stage. It can be developed in parallel to the schema design, which does not depend on either the specific server implementation nor the GraphQL framework used. While server setup is required for hosting the API without affecting other steps directly, the framework setup directly impacts the specific resolver implementation and should therefore precede this step. The next step is schema design, one of the most critical aspects of implementing a GraphQL API, as the schema defines all resources and possible queries that will be exposed through the API in great detail. Closely linked to the schema design is the implementation of resolver functions. These functions bridge

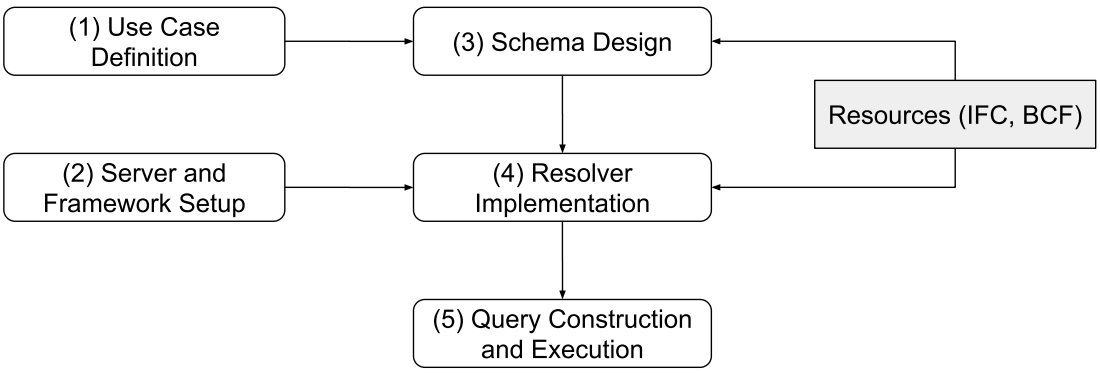

Figure 5.1: Flow chart of the implementation process.

the gap between the resources and the GraphQL schema, by defining how each schema field is populated with data. Finally, user queries should be formulated and tested against the API to evaluate the success and feasibility of the approach. The next sections cover these individual steps in more detail, discussing the challenges encountered and the design decisions made along the way.

5.2 User Scenarios

In order to explore the capabilities of the proposed approach, five distinct use cases are defined to showcase typical requests. These scenarios are designed to represent a wide range of potential interactions that might be expected from an API of this kind. The use cases are implemented and demonstrated in the following sections, providing concrete examples of the API in action without the need to support the entire IFC schema or develop a complete API. The use cases begin with a simple request and gradually increase in complexity to fully explore the capabilities of GraphQL within the BIM environment. Each use case will be presented as a task and then translated into corresponding GraphQL queries in the subsequent sections.

1. **Return global id and IFC type for all building elements that are mentioned in a specific BCF.**
   This request includes simple access to information from two different resources and merging it into one single response.

2. **Return the author, status and information about all comments for all BCF topics that relate to a specific building element.**
   This is the inverse of the first scenario, which comes with a different level of complexity. While building elements are directly referenced inside a BCF topic by their Global Unique ID (GUID), IFC building elements have no direct reference to BCF topics related to them.

3. **Return the total volume of all walls in a specific building storey.**
   In this scenario, it is assumed that the volume is available in a property set. The request for a total volume requires aggregating results, which is not natively supported by GraphQL and must be implemented at the schema and resolver level. This request also requires a filtering mechanism to access only elements belonging to a specific storey.

4. **Return material information and calculate volume of a specific wall.**
   In this scenario, it is assumed that there is no volume information available

in a property set and therefore needs to be calculated from the provided geometry.

5. **Return all walls with a specific material, which are mentioned by a specific author through BCF.**
   This scenario exhibits the highest complexity, as it requires a filtered list of walls based on BCF parameters. It involves connecting both resources (IFC and BCF) and applying filters to both. There are two filter conditions included: (1) filtering walls by material, and (2) filtering walls by the author of the BCF topic they are mentioned in.

5.3 Schema Design

Schema design is a central aspect of API implementation, as it dictates the available resources and defines the possible structures for queries and responses. The schema developed for this implementation is specifically tailored to meet the requirements of the five defined use cases. Consequently, many parts of the resource structure are omitted and not represented in the GraphQL schema, as only the types necessary for the specific use cases are supported. The schema can be easily extended to accommodate a broader range of use cases and provide access to additional parts of the resource. However, a comprehensive implementation for querying IFC models or BCFs in a generalized manner, falls outside the scope of this thesis.
Based on the use cases, two distinct data sources are identified: the IFC model and BCF (For a description of these models see section 2.1.3). While it is not necessary to maintain this distinction within the GraphQL schema, it is practical to implement them as separate sub-schemata and then model the relationships between them. Consequently, the schema implementation for the IFC model and for BCF can be addressed separately, followed by an examination of the connections that should be established between the two.

The schema design process can be broadly characterized by two key design decisions:

1. Should the schema reflect the structure of the underlying resources, or should it be designed with a user-centric approach in mind? The GraphQL schema is not required to adhere to the structure of the resources it exposes. It can (be designed to) mirror the underlying resource structures exactly or be completely based on the user scenarios the API is designed for. While these two options provide extreme approaches, any middle approach is also

feasible. It therefore has to be decided, how much both, the use cases and the resources, influence the schema design. In the GraphQL community it seems to be consensus to model the schema base on user needs while examples from research in the field of AEC exhibit a strong adherence to the underlying resource structure.

2. Should the schema be manually crafted, or should an automatic generation mechanism be employed? Manual schema design provides much more control over the exposed resources. If, however, an exact mapping between the GraphQL schema and the underlying resource structure is aimed for, automatic schema generation greatly reduces the implementation work. In this regard, both design decisions are dependent on each other.

For simplicity, in this case study, the mapping is created manually, implementing only the necessary parts of the IFC schema required for the proof-of-concept defined by the use cases. This manual approach offers the advantage of controlling which aspects of the IFC schema are exposed to the user, resulting in a lean API that provides only the information relevant to the use case. Instead of parsing and exposing the entire IFC schema, the focus can be on specific entities of interest while omitting those that are not needed. This streamlines the schema, reducing complexity and eliminating unnecessary details from the documentation and schema introspection.
By keeping this process manual, the GraphQL schema can be loosely coupled with the IFC schema. This flexibility allows entities and parameters to be omitted, added, or modified according to the specific use case, something that would not be possible with a fully automated approach.

5.3.1 GraphQL Schema for IFC

The IFC schema is very extensive, providing a large number of types, entities, enumerations etc., with a complex hierarchical structure of inheritance and relationships. In the scope of this thesis, it is not feasible to represent all of this in the GraphQL schema. The schema is focused on supporting the following concepts from the IFC model:

1. *IfcBuildingElement* with some consideration to its super-classes like *IfcRoot* (use cases 1 - 5)
2. *IfcBuildingStorey* (use case 3)
3. *IfcWall* is supported in higher detail, acting as an example how other model elements could be represented in GraphQL (use cases 3 - 5).

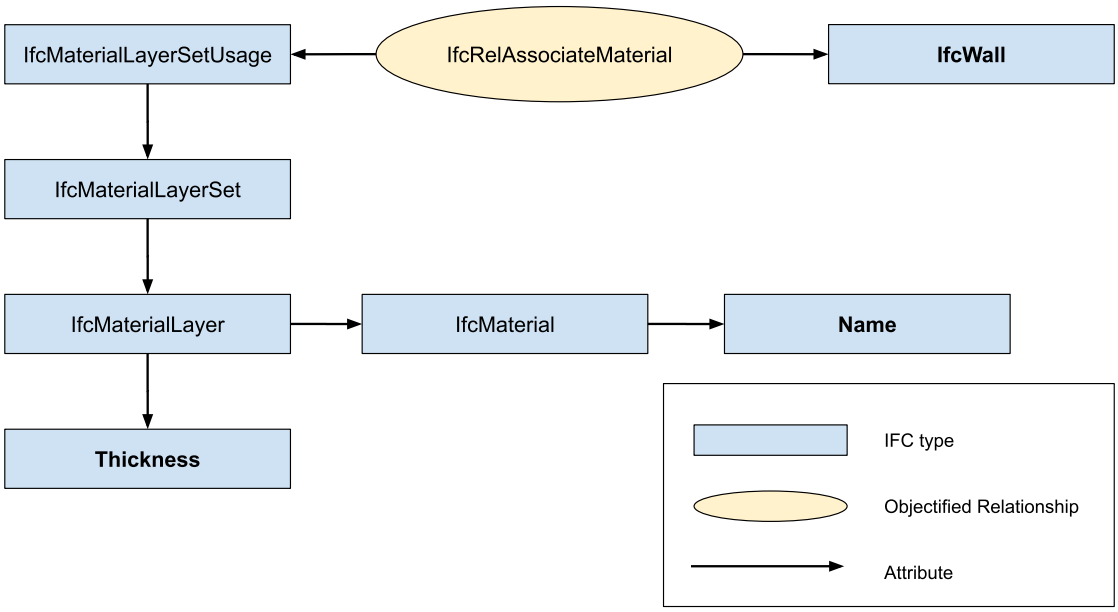

Figure 5.2: Relationship between IfcWall and Material

Following these main concepts, the schema supports some types that are connected to those entities as attributes. For example, *IfcBuildingElement* possesses the attribute *OwnerHistory* of type *IfcOwnerHistory*, which is also included in the schema even though, it is not strictly necessary regarding the user scenarios.

After examining which parts of the model need to be included, the next consideration is how closely the GraphQL schema should follow the IFC schema (see also section 3.6). IFC is designed to be used as an extensive and exhaustive exchange format for building models. It aims to provide many different disciplines with an integrated way of describing information. One result of this design goal is a relatively complicated structure of the information embedded in the data format.

The following example demonstrates this complexity: In IFC, the connection between an element and its material is not straightforward, as there are multiple different but complex ways to link a wall to its material. One of the possibilities is depicted in figure 5.2.

The *IfcWall* entity does not posses a material property that would link the wall element directly to its material. Instead, the link is realised using an objectified relationship instance of type *IfcRelAssocciatesMaterial*. This relationship object instance points to an instance of *IfcWall*, as well as to an instance of *IfcMaterialSetUsage*, which points to *IfcMaterialLayerSet*. *IfcMaterialLayerSet* has a list of *IfcMaterialLayers* as a property, which in return point to a specific *IfcMaterial* instance acompanied by a layer thickness. The *IfcMaterial* then possesses the material name as an attribute. This structure offers a lot of flexibility, describing different situations and dependencies between a wall and its material. According to

the official documentation[1], the data model is organized in this manner to "*... keep relationship specific properties directly at the relationship and (...) later handle relationship specific behavior*".
While this may be an appropriate structure to precisely describe the material and its relationship to the *IfcWall* element in an holistic approach that should cover all possible BIM related use cases, it is not very suitable for user-friendly querying. Depending on the user scenario, it might be favourable to have a direct and clear relationship between a building element and the material it consists of. Since the GraphQL schema can be as far decoupled from the IFC schema as necessary, it is possible to add the material information directly to the GraphQL *IfcWall* type. This way a single query can request a wall instance with the material information directly attached. This exchanges some of the flexibility the IFC schema offers with a concise definition, suitable for simple querying, where the information that has to be exchanged is limited by design.
The schema does not reflect the entire inheritance structure provided by IFC, but makes use of *interfaces* to mimic some of the inheritance relationships e.g., between IfcRoot, IfcBuildingElement and IfcWall.

IfcBuildingElement is defined to be "*... an abstract entity that cannot be instantiated*"[2]. It is therefore appropriate to implement the IfcBuildingIElement as an interface in GraphQL, since there will never be an instance of type IfcBuildingElement in the model. Instances that inherit from IfcBuildingElement will always have one of the sub-types of IfcBuildingElement as their types. Interfaces in GraphQL behave very similar to this. A query can define an interface as the return type. The query result must be resolved to a specific type, implementing the interface, at run time. In this way, an interface can never be returned directly to the client, without resolving its type first. Introducing this interface is very beneficial, as a query can then return a list of building elements with different types, as long as all of them implement the IfcBuildingElement interface. This is necessary because (unlike in Python or JavaScript, where the resolver functions are implemented) a GraphQL list can only contain elements of a single type. While this is, of course, not an uncommon concept in other programming languages, the most popular languages for the implementation of GraphQL APIs are JavaScript and Python, which makes it worthwhile noting this difference.
Designing the schema around user scenarios means that the GraphQL schema will not encompass the entire IFC data model, omitting most IFC entities. This is unproblematic, as long as the queries are limited to accessing only the elements

[1] https://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2_TC1/HTML/schema/ifckernel/lexical/ifcrelationship.htm, accessed: 10.07.2024
[2] https://standards.buildingsmart.org/IFC/DEV/IFC4_2/FINAL/HTML/schema/ifcproductextension/lexical/ifcbuildingelement.htm, accessed: 10.07.2024

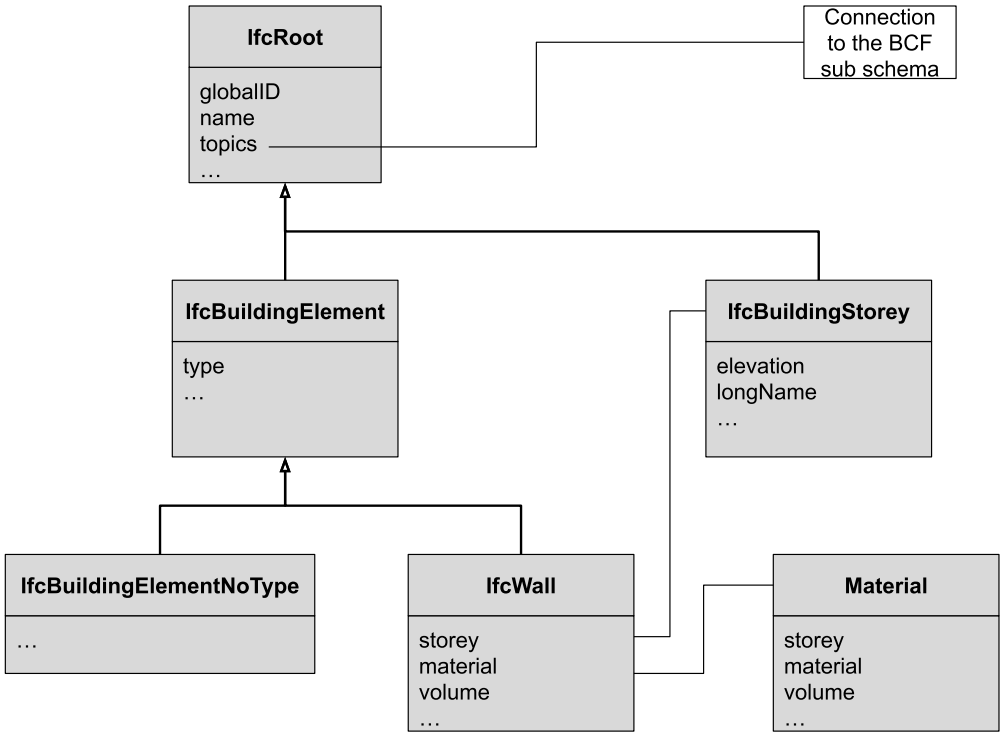

Figure 5.3: Simplified depiction of the GraphQL schema representing the necessary information from the IFC data model.

Figure 5.4: Inheritance schema for IfcWall.

defined in the GraphQL schema. However, BCF topics can reference any building element through its GUID, which raises the question how to handle referenced element types not included in the GraphQL schema. For example, a topic might reference an instance of IfcSlab, which is not represented in the simplified GraphQL schema.

One approach could be to restrict the elements a topic can reference in this case study to instances of IfcWall, since this entity is thoroughly represented in the GraphQL schema. However, this would involve controlling the resources, which the project aims to avoid. Instead, a fallback type was included in the schema to represent all building elements generically without specific GraphQL type representation. The IFC schema follows a similar concept with the entity IfcBuildingElementProxy, which is used to represent building elements that are not (yet) defined in the IFC schema.

5.3.2 GraphQL Schema for BCF

The specification for BCF is far less extensive than IFC. Therefore, most of the BCF structure was mapped to a GraphQL schema, following the buildingSMART specification very closely[3].

The BCF specification outlines a structure that seems suitable to be mimicked in GraphQL. There is no complex inheritance hierarchy and the overall structure is well understandable for human actors. Additionally, the existence of a REST API specification[4], describing the access to BCF over HTTP is another implication that the BCF structure can be represented as-is through a web API. Therefore, the BCF structure was mapped to GraphQL without changing, adding or removing any fields. Although, the schema is a nearly complete representation of the BCF specification, the manner in which the user can interact with the resources will entirely depend on the queries (and mutations), that are defined on top of the schema. At this stage, only the specified use cases are supported and not all interaction mechanisms defined for the REST API by buildingSMART are considered.
This demonstrates the possibility for a loose and a close coupling between resource structure and GraphQL schema and shows that both can be suitable depending on the resources and use cases. A visual representation of the resulting schema can be found in the appendix.

The BCF specification includes a few instances of mutually exclusive attributes. Following is an example. The *DocumentReference* element contains two attributes, which are mutually exclusive: *DocumentGuid* and *URL*. The *DocumentGuid* is used, if the referenced document is included in the BCF, the *URL* is used when an external document is referenced. GraphQL does not support the definition of mutually exclusive fields. A user can therefore request both fields if they are existent in the schema. This issue does not really affect querying, as the API only ever returns a single value for either one of the fields (there only ever exists a value for one of the fields in the resource). A deeper problem arises due to the fact that the existence of one of the attributes is required. Either *DocumentGuid* or *URL* must be present but never both. It is however not possible to mark the fields in the GraphQL schema as required since one of them will always be missing by design. A solution to accurately map this relationship to the GraphQL schema has (to the best of my knowledge,) not been proposed, and there appears to be little to no consideration of this problem within the GraphQL community. As a result, this behaviour has to be implemented on the resolver level or through the use of directives.

[3] https://github.com/buildingSMART/BCF-XML, accessed: 10.08.2024
[4] https://github.com/BuildingSMART/BCF-API, accessed: 20.07.2024

Algorithm 5.1: Solve by introducing a union

```
type PerspectiveCamera {
    cameraViewPoint: Vector3D!
    cameraDirection: Vector3D!
    cameraUpVector: Vector3D!
    viewToWorldScale: Float!
    aspectRatio: Float!
}

type OrthogonalCamera {
    cameraViewPoint: Vector3D!
    cameraDirection: Vector3D!
    cameraUpVector: Vector3D!
    fieldOfView: Float!
    aspectRatio: Float!
}

union Camera = PerspectiveCamera | OrthogonalCamera

type Visualization {
    components: Components
    camera: Camera!
    lines: [Line]
    clippingPlanes: [ClippingPlane]
    bitmap: [Bitmap]
}
```

A similar issue occurs with the mutually exclusive definition of *OrthogonalCamera* and *PerspectiveCamera*. A possible solution is addressed, using the union type in GraphQL. A union called *Camera* is introduced which combines *OrthogonalCamera* and *PerspectiveCamera* and can be used as a single value to the camera argument of the visualization information (see listing 5.1). This way the camera attribute can be marked as required and can still only take either one of the types.

Besides the issues mentioned above and the obvious differences that arise from mapping between two not completely congruent knowledge descriptions, the GraphQL schema clearly deviates in a few points from the specification:

1. **The *RelatedTopics* Element** creates a connection from the topic at hand to related topics. In the specification, this is realised by providing a list of topic GUIDs while in the GraphQL schema this is implemented as a list of *Topics*. This allows the client to directly get all the information about a related *Topic* in a single request to an arbitrary nesting level of related topics.

2. **Mutually Exclusive Attributes** can not easily be modelled in the schema, even more so if one of the fields is required. These kinds of relationships are not reflected in the schema.

3. **Mutually Exclusive Elements** are joined together using a *Union* definition.

Because the GraphQL schema representing BCF is kept as close as possible to the official specification, it can be adjusted and extended to create a GraphQL-based alternative to the BCF REST API.
This would encompass primarily defining all required queries (utilizing the REST API specification as a reference) and implementing the corresponding resolver functions.

5.3.3 Combining the IFC and BCF related Schemata

There are several connections that can be drawn between an IFC model and related BCF. This thesis focuses on the following key relationships:

1. BCF topics can link to one or more building components inside an IFC model by referencing the GlobalUniquId attribute.

2. Topics can be linked to a IfcSpatialStructureElement instance through the GlobalUniquId. This reference can be made for every file reference. It specifies the spacial structure element a topic is in relation to, e.g., which building storey the topic is referring to.

3. Building elements (like e.g., instances of IfcWall) can implicitly link to one or more topics. This information is not present in the model itself but can be inferred by the topic pointing to it.

These relationships are represented directly in the designed GraphQL schema. Just as described in the BCF specification, the Topic type in the GraphQL schema also directly references IFC elements by *GUID*. This is achieved through the *visualization* field and related references. The reference to the *IfcSpatialStructureElement* is managed in a comparable way. The implicit connection between building elements in the IFC model and BCF topics is made explicit by referencing the *Topic type* as a field within the *IfcBuildingElement* type.

5.4 Designing the Queries

When designing the queries a distinction between *query* as defined within the GraphQL schema and *query* as formulated and executed by the client application must be made. Both share the same name and are expressed using GraphQL as a language. It is important, however, not to confuse these two concepts.
The GraphQL schema specifies all available queries through the fields of a special *Query* type. These schema-designed queries serve as entry points for the client application to interact with the schema.
The query the client formulates uses one of these schema-defined entry points and then traverses the schema to request information related to the original query. Both of these concepts are obviously connected but are not the same.
Given the defined user scenarios, five queries need to be constructed. These (five) user queries must handle complex combinations of several interconnected resources, rather than just requesting single, clearly defined resources. Following this general distinction, there are two tasks:

1. Design the schema queries to ensure that they can provide the information required to fulfill the user scenarios.
2. Design the actual client queries to precisely address the specific question posed by each scenario.

Both of these tasks are interconnected and dependant on each other and have to be developed in combination, with the queries defined in the schema providing the basis and entry point for any query the client needs to execute. The next two sections discuss two general concepts, which are vital for most basic querying tasks.

5.4.1 Aggregation

Aggregation is a vital concept for many basic tasks in BIM. Aggregation is the process of creating statements or summaries about a collection of items, including statistics, sums and calculations of any kind. Aggregation is particularly significant in areas such as quantity take off. Questions like ’What is the number of windows in a specific building storey?’ or ’What is the total volume of all walls in a given building?’ require some form of result aggregation.
User scenario 3 is reliant on this concept, by requesting the *"total volume of all walls"*. Unlike SQL, which offers built-in aggregation functions *(COUNT, SUM, AVERAGE, MIN, MAX),* GraphQL lacks inherent aggregation functionality. Scenario 3

Algorithm 5.2: Simple query returning a list of IfcWall elements

```
type Query {
    ifcwalls: [IfcWall]
}
```

Algorithm 5.3: Use of a wrapper return type to attach additional information to a collection of elements.

```
type Query {
    ifcwalls: IfcWalls
}

type IfcWalls {
    elements: [IfcWall],
    count: Int,
    volume(calc: Boolean): Float
}
```

will serve as an example to demonstrate the issues and possible solutions to aggregation in GraphQL. A straightforward schema-defined query providing access to a list of *IfcWall* elements is shown in listing 5.2. Designing the query in this way does not offer the possibility to include aggregated information about the query result. The query returns a list of *IfcWall* elements to the client, any additional information, like the total number of returned elements would need to be calculated on the client application. One solution to include additional information about a collection of elements is to define a wrapper type, which includes the original collection alongside additional fields for aggregated values or other (meta)data. Instead of returning a list of *IfcWall* elements directly, the query defines this wrapper type as the return value (5.3). Note that the query does only return a single element of type *IfcWalls*, which then contains a list of the individual wall elements.
This return wrapper contains fields for aggregated values: (1) count (representing the total number of returned elements) and (2) volume (representing the total volume of all walls summed up). However, resolving these fields is still not trivial. In GraphQL a resolver function is not aware of sibling fields, let alone their resolver functions. This means, that the volume field does not have access to the result of the resolver of the elements field, although the volume is dependant on this information.

Algorithm 5.4: Making use of parameters to implement filtering functionality.

```
type Query {
    ifcwalls(storey: String): IfcWalls
}
```

5.4.2 Filtering

Another important concept, which is required to answer the use cases is filtering. Filtering is required to restrict the results depending on some condition. It is a standard functionality and part of almost any GraphQL API implementation. Filtering, just like aggregation is not build into GraphQL and has to be integrated through the schema and the resolver levels. Since filtering is such a standard technology, required by almost any application, there are established solutions and best practices regarding its implementation. Filtering is usually enabled on the schema level by attaching a parameter to the query that should return filtered results. This parameter (or multiple parameters) can be used by the resolver function to restrict the results returned by the query. Listing 5.4 shows the already known example of the *ifcwalls* query now containing an additional parameter called *storey* that enables filtering the walls according to the building storey in which they are located. It is important to note that this alone does not offer any functionality. It just enables the client application to include additional information in the query, in this case a parameter called *storey* of type *string*. Any actual filtering must happen in the resolver function that is responsible for enriching this query with information. The fact that filtering implementation is left to the developer means that any arbitrary filtering mechanism can be implemented, e.g., key-word based, full-text search and/or regex.

There are other functionalities that are part of almost any real world API implementation, like pagination, as well as mechanism for authentication and authorization. Since both of these are not required to support the defined user scenarios, there will be no implementation of these concepts as part of this case study.

5.4.3 Implementing User Scenarios

Incorporating the principles introduced above, the following queries as listed in figure 5.5 need to be defined in the schema in order to enable answering the user scenarios. It is important to note, that these queries do not provide all the information needed to satisfy the user scenarios directly but act as entry points to the entity graph defined by the schema. For example, the *topics* query returns a list

Algorithm 5.5: All available queries as defined in the GraphQL schema.

```
type Query {
    ifcwalls(storey: String): IfcWalls,
    ifcwall(globalId: ID!): IfcWall,
    topics: [Topic]
    topic(guid: ID!): Topic
}
}
```

of *Topic* elements, the user query can then define all the fields of the *Topic* element it wants included in the result. These fields point to other types in the schema, which in turns point to others and so on. With only these four queries defined on the schema level a variety of different queries requesting various different parts of the resources can be constructed and executed.

These queries, defined in the schema, form the basis for formulating user requests based on the user scenarios. Each user request starts with one of the defined queries and further includes connected types to address all aspects of the scenario. Each one is shortly presented below.

User Scenario 1: *Return global id and IFC type for all building elements that are mentioned in a specific BCF.*
This query (code listing 5.6) starts at the *topic* query defined in the schema and takes a parameter of type *ID* to identify a single topic by its GUID. The IFC elements, the topic refers to, are connected in a relatively counter-intuitive way: The topic contains visualization information, which contains a parameter called *components*, providing information about all referenced *components*. The *Components* type contains a parameter called *selection*, containing a list of elements of type *Component*. Each *component* refers to a specific IFC element and includes a parameter with the GUID pointing to this element. This convoluted way of referencing IFC GUIDs comes from the original BCF specification and was directly adopted. It would be possible to attach the referenced IFC GUIDs to the topic in a more obvious and direct way. It was, however, a design decision for implementing the schema to follow the BCF specification as closely as possible.

User Scenario 2: *Return the author, status and information about all comments for all BCF topics that relate to a specific building element.*
This query (code listing 5.7) starts at the *ifcwall* schema-query, providing an IFC GUID to access a specific building element. This query returns a single element of type *IfcWall*. The query further specifies, which field is needed from this element (*topics* field) and which fields are needed for all nested elements, including, for example, information about each comment of each topic.

Algorithm 5.6: GraphQL query supporting user scenario 1.

```
query Topic($guid: ID!) {
  topic(guid: $guid) {
    visualization {
      components {
        selection {
          element {
            globalId
            type
          }
        }
      }
    }
  }
}
```

Algorithm 5.7: GraphQL query supporting user scenario 2.

```
query Ifcwall($globalId: ID!) {
  ifcwall(globalId: $globalId) {
    topics {
      creationAuthor
      topicStatus
      comments {
        author
        comment
        date {
          ISO8601
        }
      }
    }
  }
}
```

User Scenario 3: *Return the total volume of all walls in a specific building storey.* The query for scenario 3 (code listing 5.8) is less verbose than the previous two, which however does not reflect the complexity of the implementation (this applies especially to the resolver level). The entry point into the entity graph in this scenario is the *ifcwalls* query, which returns a list of *IfcWall* elements inside a return wrapper and can be filtered using a string, specifying the building storey. A single (aggregation) field is requested on this return wrapper, which is the sum of the volume of all filtered elements. This volume field comes with a parameter, controlling if the volume for each wall should be calculated or read from a property set if it exists. The elements themselves are not included in the response, since the scenario does not required it.

Algorithm 5.8: GraphQL query supporting user scenario 3.

```
query Ifcwalls($storey: String, $calc: Boolean) {
  ifcwalls(storey: "Level 1") {
    volume(calc: false)
  }
}
```

User Scenario 4: *Return material information and calculate volume of a specific wall.*

The fourth query (code listing 5.9) shares similarities with scenario 2, as information is requested about a single (*IfcWall*) element. The requested fields are *volume* and *material*. In contrast to scenario 3, where the volume was available as part of a property set, here the volume must be calculated based on the geometry of the element.

Algorithm 5.9: GraphQL query supporting user scenario 4.

```
query CalculateVolume($globalId: ID!, $calc: Boolean) {
  ifcwall(globalId: $globalId) {
    volume(calc: true)
    material {
      name
      thickness
    }
  }
}
```

User Scenario 5: *Return all walls with a specific material that are mentioned by a specific author through BCF.*

The query for scenario 5 (code listing 5.10) makes use of two filtering parameters: *material* and *mentioned_by*. Since the resolver chain starts at the root query and works its way down to the leaf nodes, the filtering must be done at the top level of the resolver for the *ifcwalls* query. This means, that the resolver function bound to this field of the Query type needs to implement the necessary filters without any knowledge of the result of any child resolvers. While this is easily achievable in this use case, it highlights that filtering, based on the results of child resolvers, is not a trivial task in GraphQL.

While these five user queries have been worked out in detail to provide examples, the current schema and resolver implementation allow the construction of many more varied client queries without adding any new types and/or resolver functions.

Algorithm 5.10: GraphQL query supporting user scenario 5.

```
query WallsByMaterialAndBcfAuthor($material: String,
   $mentioned_by: String) {
  ifcwalls(material: $material, mentioned_by: $mentioned_by) {
    elements {
      globalId
      material {
        name
      }
    }
  }
}
```

5.5 Implementation of Resolver Functions

Implementing the resolver functions is the main programming task, besides setting up the infrastructure for the API in general (server setup, routing, setup of the specific GraphQL framework and similar tasks). Implementing the schema includes most of the general design decisions and already predefines the required resolver functions as well as their input parameters and return structure. In general every field of the schema has an associated resolver function. Most frameworks, however, provide default or fallback resolver functions for trivial fields (see section 3.8). A fallback resolver can be used if the parent resolver returns a dictionary containing a key with the exact field name. Most type resolvers make use of this feature and return a dictionary already containing information about most fields under the correct key (the name of the field). A lot of resolvers can be omitted because of this, which greatly reduces the necessary implementation work.
The resolver functions rely on the Python package *IfcOpenShell* to interact with both the IFC building model, as well as the BCF in XML format. The concrete implementation details of the resolver functions are not crucial for the general concept of the API. However, there are a few resolver functions that are worth examining in more detail.

5.5.1 IfcWalls Resolver

The resolver for the *ifcwalls* query is the most comprehensive, as the IfcWall type is implemented in the GraphQL schema with the greatest detail, and most user scenarios rely on this type. Derived from the user scenarios, this resolver function includes three (optional) parameters: *storey* (to support scenario 3), as well as *material* and *mentioned_by* (to support scenario 5).

The resolver function must (for each parameter): (1) check if the parameter is given and if so (2) filter the result accordingly. Since a resolver function has no way of accessing the result of any child resolvers down the chain, this filtering mechanism must be implemented at the top level of the *ifcwalls* query resolver.

5.5.2 Volume Resolver

The resolver function for the volume field of the *IfcWall* type occupies a unique role due to its increased complexity and the need to support both retrieving the volume from property sets and calculating it based on the geometric representation.
Reading the information from property sets presents the problem that there is no uniform standardization regarding how and under what name the volume is stored. This means that both, the name of the property set and the name of the required parameter must be known beforehand. Property sets have a GUID, a name, and contain a list of properties, each identified by a name and associated with a value. They are utilized to attach information that extends beyond the IFC schema to IFC instances. BIM authoring tools commonly use property sets to attach information exactly like volume information to building elements, following an internal nomenclature. A wall modeled in Revit, for example, might carry information about its volume in a property set named "PSet_Revit_Dimensions" under the key "Volume", while the same information coming from ArchiCAD might be found in a property set named "ArchiCADQuantities" with the key "Volume (Net)".
For this case study the model with all its property sets and name values is known. Correctly providing this information in a general case may not be trivial without losing the semantics of the property. This means, integrating property sets into GraphQL types in an abstract manner without attaching meaning to the properties is easily achieved. This could be done, for example, by using a list of key-value pairs as the value of a field called *propert_sets*. However, associating the correct property with the appropriate field in the GraphQL schema could be non-trivial.
Calculating the volume is done using the geometry processing library OpenCascade[5], more specifically the Python wrapper Pythonocc-core[6].

5.5.3 Resolver Optimization

The modular architecture of resolvers can sometimes result in sub-optimal behavior within the resolver chain. A common scenario in which this occurs, is when a query requests a list of elements stored in a relational database. In this

[5] https://dev.opencascade.org, accessed: 08.08.2024
[6] https://github.com/tpaviot/Pythonocc-core, accessed: 08.08.2024

case, each resolver acts independently, often managing a separate database connection to retrieve the necessary information for its specific field. As a result, multiple individual requests are sent to the database, rather than consolidating the information into a single query. This approach can lead to inefficiencies and unnecessary overhead. This issue is a common occurrence in many projects and is widely known as the *n+1* problem. It happens when a query requesting a list of *n* elements leads to *n+1* database requests. Facebook has introduced a solution called DataLoader, available on GitHub[7]. This approach involves creating an intermediary layer between resolver functions and the persistence layer, tasked with handling data access. The primary goal of this layer is to enhance data access performance through two key functionalities: caching and batching. Caching involves storing the results of frequently requested data, allowing subsequent requests for the same resource to be served from the cache, thereby avoiding repeated data access. Batching, on the other hand, combines multiple requests within a specific timeframe into one single database query, significantly improving performance compared to processing each request individually.
A more specific problem, which was encountered during implementation also stems from the modular resolver design. Consider a query, that returns a list of *IfcWall* elements, where a material field is requested for each element. The material resolver takes the GUID of an element and uses IfcOpenShell to (1) load the model, (2) find the material for the specific element, (3) close the file, and (4) return the material information. This way the resolver is completely atomic and does not rely on anything other than the element GUID. As a result, the IFC file is opened and read once for every element in the list, which is extremely slow. In order to solve this, the IFC file is loaded into memory by the route handler and is attached to the *context* object. The context provides information common to all resolvers for a specific query (see chapter 3). This way, every resolver function has access to the model without the need to read it first. Another advantage is the fact, that this way the server's route handler can manage which file is getting loaded for which query. The same approach was applied to the BCF, which gets decompressed and read by the route handler and attached to the query's *context* object.

5.6 GraphQL Frameworks

There are GraphQL frameworks available for almost all popular programming languages. The official GraphQL documentation lists libraries for languages including but not limited to: JavaScript, Go, Python, C#, C++, PHP, Rust and many more.

[7] https://github.com/graphql/dataloader, accessed: 01.08.2024

Two frameworks were tested as part of the case study based on two different environments: Ariadne which is available as a Python package and Apollo based on Node.js and JavaScript/TypeScript. Besides personal preference and familiarity with specific programming languages, the choice of a specific framework is not that influential. Both frameworks tested are fully capable of supporting the defined user scenarios. This can also be expected for other available alternatives. The final implementation of this case study relies on the Ariadne framework because it is available in Python. There are no competitive alternatives for the processing, authoring and querying of IFC models similar to IfcOpenShell available for Node.js. As a result, Python is used to interact with the IFC models. Implementing the server logic and GraphQL layer in the same language means, the resolver functions can access the resources directly (with the use of IfcOpenShell), instead of requiring an additional intermediate layer.

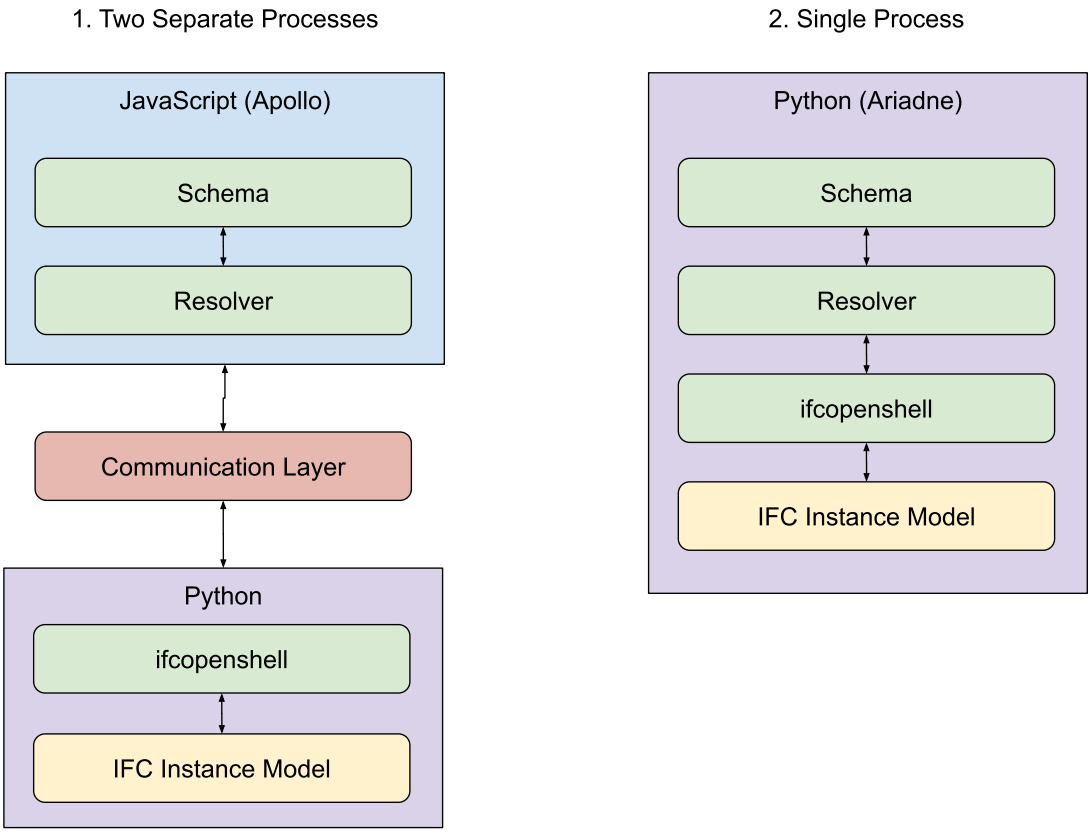

Figure 5.5: Two possible application designs depending on the choice of frameworks and software tools.

An alternative was explored using Node.js and Apollo to set up and host the GraphQL API, while still interacting with IFC using Python. This means, that there needs to be communication established between the GraphQL back-end and a Python process, providing access to the resources. An illustration of both

approaches is shown in figure 5.5. This communication layer can be established in a number of ways: The first option is to set up an intermediate API that can be used to manage the access to the IFC model and provide a means to access specific information. This can be implemented, e.g., as a REST API service. The advantage is the clear separation between data access and resolver function implementation with a clearly defined interface in between. This makes it possible to change implementation specifics for the data access without affecting the resolver layer, as long as the original REST endpoints remain available. The second option (which is the one that was tested in the context of this case study) relies on executing Python scripts directly from Node.JS, using child processes. To this end a main Python script was implemented and set up to function as a command line program, accepting input parameters that control which exact query should be executed on the IFC model instance. The query is then executed inside a Python environment, making it possible to use tools like IfcOpenShell. The results can be communicated back to the parent process in different ways:

- **Via files**: write the result to a JSON file and notify the parent process when the writing operation is done and the file is closed. The parent process (resolver function) can then open the file and resolve the field it is responsible for with the JSON object directly.
- **Via *stdio* streams**: the child process can directly send data to the parent process (and vice versa) using the *stdin* and *stdout* streams. The parent process then needs to parse the incoming stream. This avoids the need to write and read actual files to storage, increasing the speed of interaction.

Any of the above solutions to integrate two different programming languages consists of two separate processes with an additional layer in between, greatly complicating the whole process of data access. While this would be appropriate for a production setting, it seems to be too much effort for too little benefit in the context of a small case study. Which is why both, the GraphQL API and data access logic where ultimately implemented using Python as a programming language. This goes to show, that the available functionality in the host programming language regarding the resolver functions might be a key parameter, influencing the decision about the GraphQL framework.

5.6.1 Interpretation Layer

Implementing access to raw data sources can be cumbersome and labour-intensive. The IFC model instances used in this case study are available in

a clear text encoding with the information being relatively easy accessible in general. Extracting the information in a consistent and stable manner, however, can be quite complicated depending on the structure and data model of the resources. Usually, there are existing libraries that facilitate the data access for almost all common data formats. IFC is no exception, offering various libraries available that provide developers with simplified reading and authoring utilities. This means that most of the time, the intricacies of accessing and parsing data from different sources can be outsourced to established libraries. This allows developers to focus on new tasks by building on the work of others.
Probably the most popular library for IFC processing through code (at least when it comes to prototyping) is IfcOpenShell, providing easy access to information stored in IFC and handy utility functionality.
In general, there is usually an intermediate interpretation layer on top of the raw resources. This layer can consist of multiple levels of abstraction itself. IfcOpenShell for example is build upon other libraries (like Eigen, Boost, OpenCascade and more) that are responsible for different parts of its functionality. An example of this layered structure is demonstrated in figure 5.6.
Even though the use of third-party libraries like IfcOpenShell greatly simplifies the

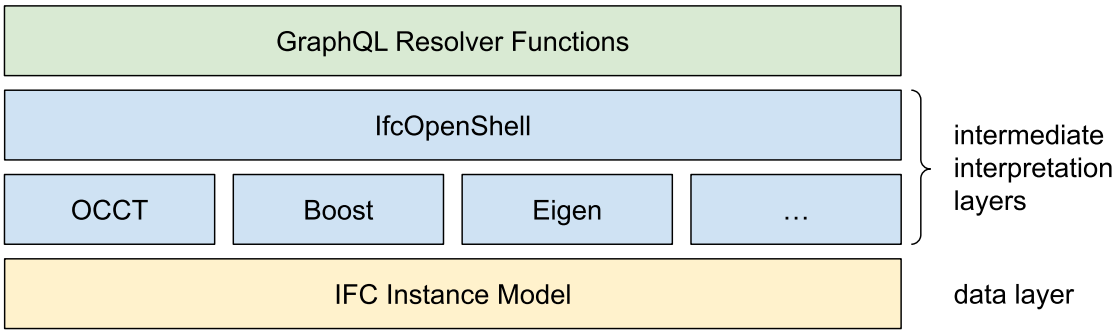

Figure 5.6: Data access orginized in layers.

implementation and development process, a GraphQL API is not reliant on these libraries. There is no technical reason against implementing custom parsing logic or even implementing access to raw data sources directly at the resolver functions level. This is not really relevant to the specific implementation offered by this case study, as there are excellent libraries that enable data access to the resources used. It highlights however, that GraphQL is agnostic to the persistence layer as well as the specific access mechanisms. Resolver functions can integrate almost any data format into the API with or without the use of intermediate interpretation layers.

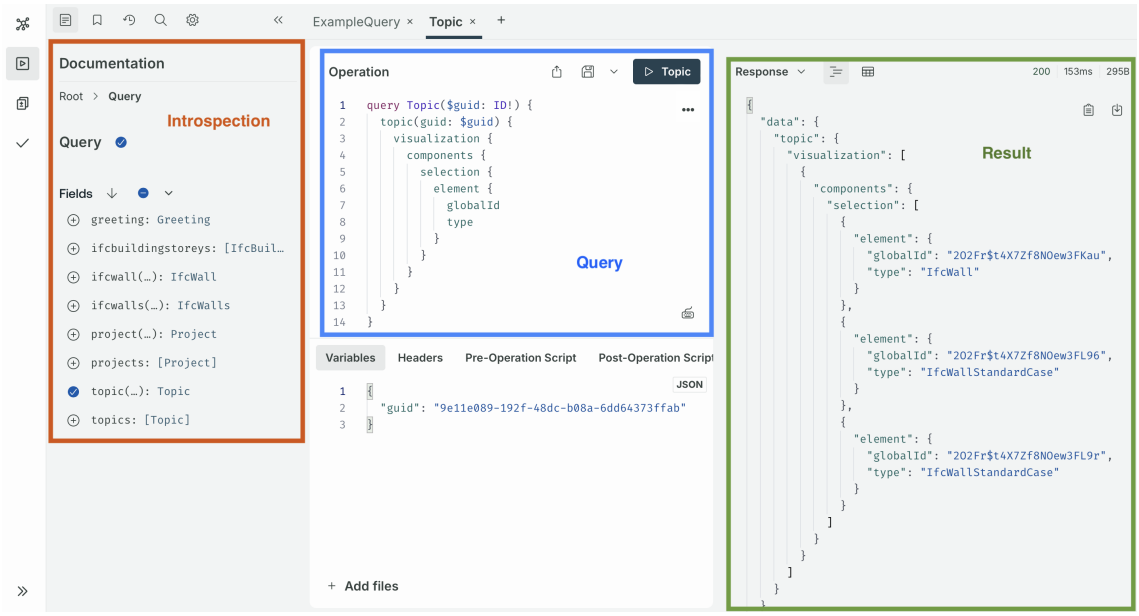

Figure 5.7: Dedicated client to test and explore the GraphQL endpoint.

5.7 GraphQL Client and Executing the Queries

The final steps are to host the GraphQL API, send the queries defined in the previous section as request payload and examine the result. The hosting is done locally using a simple Flask[8] server. The query payload is handed over to the GraphQL framework which executes the query against the entity graph, constructs the result, and returns it back to the Flask route handler. This query result is then returned as JSON back to the client. Queries can be send to the endpoint using HTTP POST requests. This can be done from client applications (using browser APIs like *fetch*) from API testing tools like Postman or using dedicated GraphQL clients (such as GraphiQL or Apollo Explorer). These clients offer the advantages of visualizing the introspection result, providing a documentation of available queries and fields. Figure 5.7 provides a depiction of the user interface that is provided by the Apollo Explorer. It includes introspection information and offers a simple interface to construct queries and examine the results.

This concludes the description of the provided implementation. A discussion of the results and limitations, as well as possible alternative design decisions, is provided in the next chapter.

[8] https://flask.palletsprojects.com/en/3.0.x/, accessed: 02.08.2024

Chapter 6

Discussion

BIM projects are characterized by the collaboration of many different project partners and stakeholders. The resources are heterogeneous and often highly interdependent, with varying representations of information between different disciplines. CDEs, model server, and Sematic Web technologies are some of the solutions to enable collaboration in this difficult environment. This thesis summarizes the main research findings about collaboration and data exchange technologies in the AEC industry and proposes a solution using GraphQL. This solution is tested against five use cases.

6.1 Achievements

The implemented GraphQL API is able to satisfy all requirements defined by the user scenarios in chapter 5. Some specific behaviors are not trivial to implement using GraphQL, e.g., filtering based on the result of child resolvers, which affected scenario 5. However, working queries could be constructed for all five scenarios and provide the requested information adequately. Commonly used API functionalities, such as filtering and aggregation, have successfully been implemented and deployed on heterogeneous resources. This demonstrates, that GraphQL is suitable for the use in a BIM context (research question 3). Several details are left open in the official GraphQL specification. This includes, e.g., popular features in data querying like filtering and pagination. What may seem like a disadvantage at first glance, actually offers a great flexibility in the integration of various heterogeneous resources. By leaving the implementation of these features to the respective developer, they can be adequately tailored for any data structure. This avoids scenarios, where a specific solution would not be suitable for a specific use case or a distinct data model because there is no predefined solution.
The exact same circumstance can, however, also be construed as a downside of GraphQL in general, depending on the use case of the API. Quality-of-life features are not provided automatically, which can be a disadvantage if all a user requires is a working product with minimal effort. Especially, if the use case will not benefit from the flexibility this design offers.

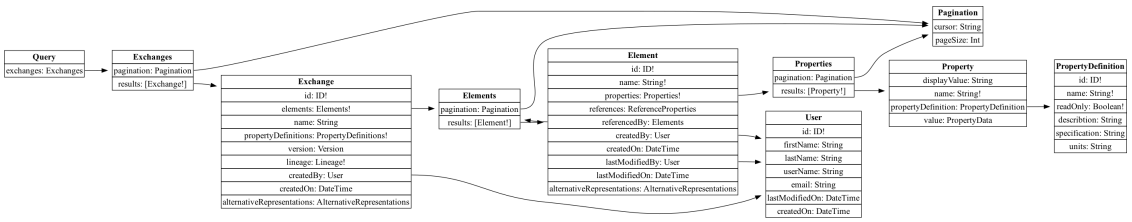

Figure 6.1: GraphQL schema with a generic representation of building elements.

6.2 Possible Different Approach in Schema Design

The relationship between resources and schema implemented as part of this master thesis can be labeled as an early-binding approach. The available model elements are represented explicitly in the GraphQL schema (which acts as the host language in this analogy). This analogy may not be entirely viable in the context of the GraphQL schema, it provides however, a useful lens to categorize different schema design approaches. In contrast to this, a late-binding approach is also conceivable and has been evidently implemented for a commercial GraphQL API by the company Autodesk[1].
Since this seems to be the first publicly available solution of a GraphQL API to access building model information a closer look was taken on the structure of this API and specifically the schema designed for this API. The comparison between the API implemented as part of this case study and the API developed by Autodesk reveals a fundamentally different approach to schema design, with distinct or even opposing advantages and disadvantages.

Autodesk takes a more generic approach when it comes to schema design. The rough structure and contents of the schema are demonstrated in figure 6.1. Model elements are generically treated as so-called *elements*.
The GraphQL schema (which is static) has to be able to represent very diverse element types. An element might be a wall, a window, a door, etc. These different element types come with different attributes attached.
In order for the schema to be able to represent diverse element types with divers attributes, the following approach was chosen: the element attributes are not modelled as a part of the GraphQL schema but rather are represented as a list of properties. Each property is essentially a key-value pair (with added supplementary information).
This approach offers the possibility to represent arbitrary element types while still keeping the GraphQL schema concise and lean. The downside is, that it looses some of the strength of GraphQL on the element level. Since the properties of

[1] https://aps.autodesk.com/en/docs/fdxgraph/v1/developers_guide/overview/, accessed: 14.08.2024

the elements are not represented as individual fields in GraphQL, they can not link to other GraphQL types. As a result, they can not be linked to other objects directly. A link could be established using a property, but a relationship of this kind can not be followed directly inside a single request, since this connection is not represented in the entity graph.

6.3 Limitations of the proposed approach

In order to fully answer research question 4, the limitations of the approach have to be examined as well. As a proof-of-concept implementation, the proposed GraphQL API can not be seen as a finished product. Several key features that any API should posses, are omitted in order to decrease complexity. These can be seen as limitations originating from the defined scope of the thesis and could easily be solved by extending the implementation.

Scope Limitations:

1. Key features are not implemented. This includes pagination, authentication, authorization and managing multiple files and projects.
2. There is only a very limited support of the resource schema. Only a handful of the classes available in IFC are represented in the GraphQL schema.

Besides obvious and deliberate gaps in the implementation, there are general technical challenges that a GraphQL API in this context would need to cope with.

Technical Challenges:

1. **Adding pagination and filtering complicates the schema:** Common pagination results like cursor-based pagination require additional information to be embedded into the schema. This is commonly done by adding fields like *item count* or *cursor information* to return types. Usually this is accompanied by creating wrapper types to hold this additional information. As a result the schema contains fields, that are not part of the exposed resources but must be included as "*helper*" fields to enable advanced querying functionalities. This complicates the schema by introducing additional fields and nesting levels, obfuscating the actual information provided by the API. This is a common issue and extending the schema to solve this is usually accepted. However, it is worth noting that this takes away some of the appeal of GraphQL, which is providing a very concise and self documenting description of the exposed resources.

2. **Schema design requires extensive of effort:** An intelligently designed schema is one of the main advantages of GraphQL, providing an easy to understand structure to access potentially complex data structures. However, designing a schema this manner requires a lot of manual work. The alternative is automatic schema generation, which can be observed in some of the examples from current research 2. This approach, however, often forgoes the advantages of a manually designed schema completely, exactly mimicking the (complicated) data model of the resources.

3. **Static schema:** The GraphQL schema is centralized and static and can therefore not be created dynamically. Any changes to the schema require a server reload of some kind.

4. **No global identifiability of resources:** Contrary to Linked Data, types in GraphQL can not be uniquely identified and addressed in a global context.

5. **Centralized schema:** A GraphQL API requires a central schema. Schema federation is possible but mainly refers to the workflow of creating one combined schema across different teams (see section 3.6.5).

6.4 Scalability and Application to Other Data Models

This section discusses the potential for extending the proposed method and applying it to more extensive use cases. With small prototypical implementations, there is always the question of whether or not the same principles can be applied to more complex scenarios. Future projects will need to determine how well the principles presented here can be applied to other areas. There are, however, general observations that can already be made about extending the case study or applying the same principles to different areas/domains.

6.4.1 Scaling

There are several ways to extend and scale the proposed implementation. Two categories will be discussed: The first one involves scaling the technical infrastructure that provides the basis for any API (vertical and horizontal scaling). The second one involves extending the API itself by, e.g., scaling the GraphQL schema, adding support for different data models, and so on.

Scaling Distributed Systems

Distributed systems in general can be scaled vertically and horizontally. Vertical

scaling refers to increasing the capabilities of existing infrastructure, e.g., by upgrading hardware, installing more server RAM and so on. Horizontal scaling refers to adding additional server instances to divide the load upon multiple points (server cluster).
Vertical scaling is naturally possible for almost any API and server architecture since increasing hardware capabilities do not change the structure of the API in terms of technical details like routing and load balancing. It just means that the existing system can answer quicker, store more information in memory, and so on. Vertical scaling is more complex and usually requires additional logic to divide and balance the load across multiple servers. According to the best-practices section of the official documentation, GraphQL supports horizontal scaling, however, the implementation of this feature is left to the API developers. The documentation refers to companies operating GraphQL APIs under extremely high loads.
Scaling server systems is not unique to GraphQL but follows the same principles for any kind of architecture, which goes into a technical area that is outside the scope of this thesis.

6.4.2 Extending the Schema

Extending the schema is an obvious way of broadening the presented implementation. The IFC schema offers approximately 800 entities and can represent building elements and concepts from numerous different disciplines. Transferring these existing concepts to GraphQL offers abundant opportunities to develop and support new user scenarios without the need to integrate new data models.

6.4.3 Transfer to other Data Models

IFC and BCF were used as the supported data models for this case study. However, there should be no difficulty in integrating other data sources into an API like this. As long as the knowledge can be accessed and parsed in the target environment, e.g., by resolver functions, any representation can be used. Examples from recent research have shown that GraphQL can successfully be used to interact with knowledge represented in RDF (Taelman et al., 2018; Werbrouck, Senthilvel, Beetz, Bourreau, et al., 2019). Several public GraphQL APIs showcase that various different resources and data storage schemes can be used as a persistence layer. Relational Databases, No-SQL, REST APIs, other GraphQL APIs, in-memory data, and file systems can be used to store the data. GraphQL is especially useful for structured data that can be represented in the JSON format. Even though the official specification (GraphQL, 2021) does not require a specific serialization for

the result, JSON is by far the most common one. Data that can't be adequately represented this way, such as images or PDF documents, is more problematic to include in an API like this.
Since exchanging information through files is still a major part of collaboration in the AEC industry, this might be a drawback of the architecture. It is somewhat mitigated by the fact that GraphQL can easily be combined with REST endpoints (see section 3.5). REST is, in contrast to GraphQL, very flexible when it comes to the data format of the response. The *media-type* header offers the possibility to distinguish between numerous data formats when sending a response. The client will check this header and treat and interpret the data accordingly.
One solution that is used in production is utilizing REST alongside GraphQL to handle tasks related to entire files, such as file uploads and downloads.

Chapter 7

Conclusion and Outlook

Collaboration and data exchange are important topics in the AEC industry. Construction projects are characterized by the collaboration of various different stakeholders with diverging motives. Even though centralized data management and exchange solutions are applied in almost all large scale projects, exchange of information still commonly happens on the document level. This granularity of data exchange has been identified as an issue by many in research. Enabling information access on the data level is solving several issues in collaboration.
All of this indicates a high demand for interfaces, which enable and facilitate an exchange of information in a fine-grained manner.

GraphQL gained a lot of popularity in recent years, both as a query language and as an API architecture. As such it provides one way of defining and implementing the mentioned interfaces in a web-context. Other approaches to improve collaboration practises in the AEC industry are not yet solving all existing issues. This includes centralizing the information as a single source of truth, e.g., through the use of CDEs and model servers as well as using Semantic Web technologies to enable fine-grained data access to highly interconnected resources.
GraphQL is praised for its simplicity as a query language, lowering the threshold to be integrated by developers. It furthermore aims to solve several issues associated with the REST architectural paradigm, which is still the predominant architecture for web APIs.
A GraphQL API was developed as part of this thesis and was tested against specific user scenarios. This proof-of-concept implementation was carried out in order to examine the usefulness of GraphQL in the context of BIM. While the resulting implementation has a limited scope by design, it demonstrates that GraphQL can be used to access heterogeneous, interconnected domain data, like building models in the IFC format and issue management information in form of BCF.

Even though GraphQL is compared to SPARQL as a query language, it is not in competition with the technologies associated with Linked Data in general. On the contrary, there are several approaches combining GraphQL and Linked Data replacing SPARQL as the query mechanism with a user-friendly, accessible alternative. Linked Data is one of the representation of resources that can be integrated using GraphQL instead of a competing technology.

Something similar can be stated for GraphQL compared to REST as an API architecture. Although GraphQL was designed to address shortcomings of REST, it should not be understood as an exclusive alternative, as both technologies can be combined well.
GraphQL seems to be a suitable building block to enable fine-grained information exchange and provides the means to implement user-friendly web APIs, which can help to improve the current practices of collaboration in the AEC industry.

Providing standardized, user-friendly interfaces to access heterogeneous data seems in high demand when looking at current research and the current state of collaboration in the AEC industry. GraphQL is known for its flexibility and valued for its simplicity as a query language. There are, however, very few approaches using GraphQL in the context of BIM. Most, if not all, of the currently available examples in research use solely Linked Data to represent the resources, while one of the compelling arguments for GraphQL is its flexibility when it comes to knowledge representations. While this thesis provides promising results at a proof-of-concept level, there are several topics that seem to offer potential for further research, either by extending the presented methodology or by exploring related questions that were outside the scope of this thesis. A few promising ideas are featured below.

- This thesis shows that GraphQL can be utilized to interact with both IFC and BCF data formats, and is able to adequately represent the connections between both resources. Future research could explore GraphQL in combination with **different data models**. In general, extending the use case by adding to the schema, implementing support for additional data models could be beneficial to investigate the usefulness of GraphQL for bigger and more complex scenarios.

- Exploring the potential of **federated schema design** appears promising, especially when considering the requirements for collaboration.

- Another interesting idea could be to further build on the schema design for BCF that was implemented as part of this case-study, potentially creating a **GraphQL BCF API** implementation that is a counterpart to the BCF REST API specification.

- **Automatic schema generation** is a topic that was merely touched by this thesis. While other research used this approach (e.g., to create a GraphQL schema from a given ontology), there is a need for further research to determine its usefulness in the BIM context for other data models as well.

Generally, GraphQL seems to have significant potential in the context of BIM, both as a query language and as a API architectural paradigm. However the specific use-cases and benefits still need to be further examined.

Appendix A

Bezeichnung des Anhangs

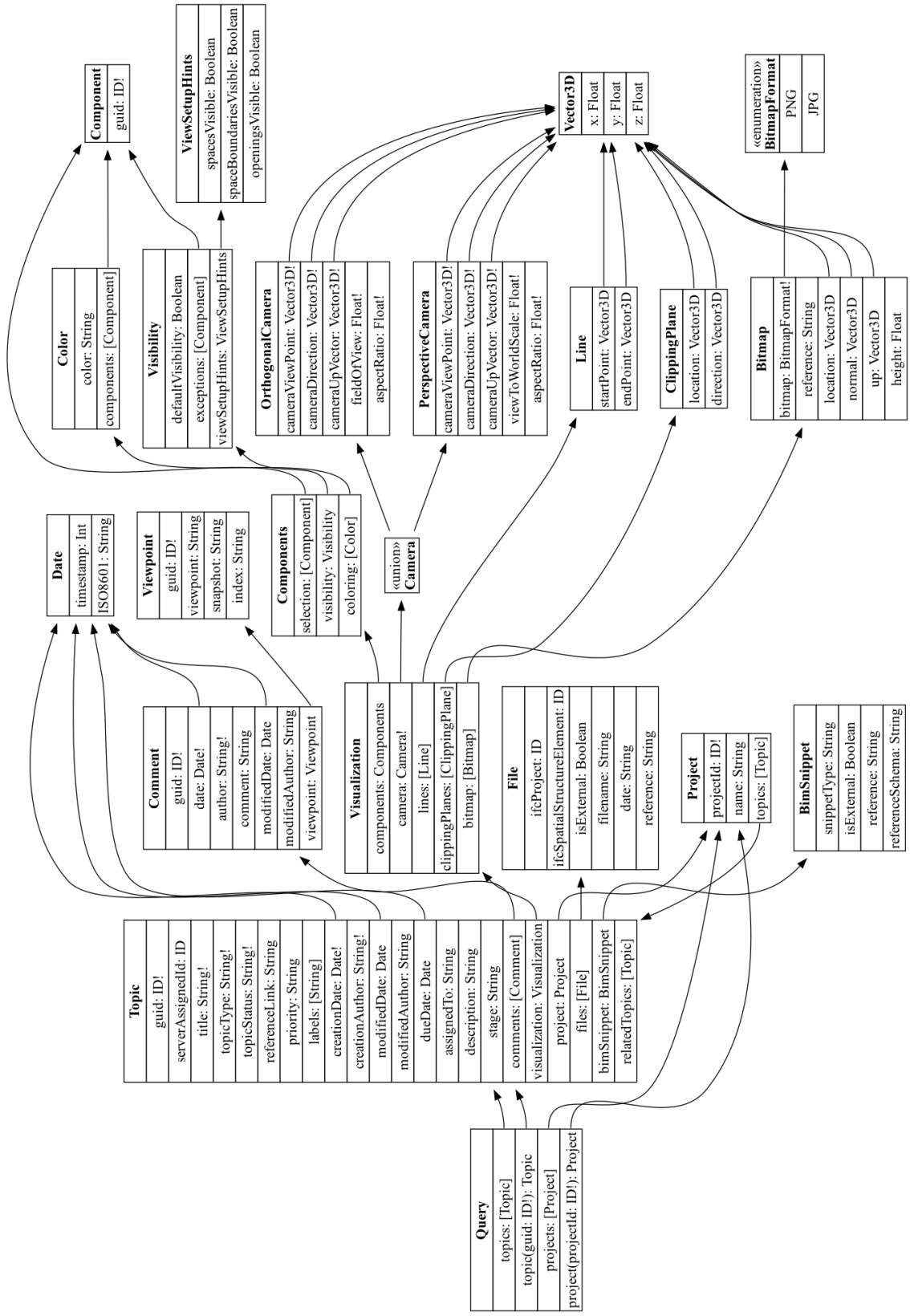

Figure A.1: Visual representation of the GraphQL schema for BCF.

Bibliography

Amann, J., Esser, S., Krijnen, T., Abualdenien, J., Preidel, C., & Borrmann, A. (2021). BIM-Programmierschnittstellen. In *Building Information Modeling: Technologische Grundlagen und industrielle Praxis* (pp. 263–290). Springer. https://doi.org/10.1007/978-3-658-33361-4_13

Artus, M., Koch, C., & König, M. (2021). Datenmodellierung. In *Building Information Modeling: Technologische Grundlagen und industrielle Praxis* (pp. 53–72). Springer. https://doi.org/10.1007/978-3-658-33361-4_3

Beetz, J., van Berlo, L., de Laat, R., & van den Helm, P. (2010). BIMserver.org – an open source IFC model server. *Proceedings of the CIP W78 Conference*, *8*. Retrieved July 2, 2024, from https://itc.scix.net/pdfs/w78-2010-51.pdf

Berners-Lee, T. (2006). *Linked Data - Design Issue*. Retrieved July 24, 2024, from https://www.w3.org/DesignIssues/LinkedData

Bizer, C., Heath, T., & Berners-Lee, T. (2023). Linked Data - The Story So Far. In *Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web* (pp. 115–143). https://doi.org/10.4018/jswis.2009081901

Bock, B. S., & Eder, F. (2024). *ifcSQL*. Retrieved August 10, 2024, from https://github.com/IfcSharp/IfcSQL

Borrmann, A., Beetz, J., Koch, C., Liebich, T., & Muhič, S. (2021). Industry Foundation Classes – Ein herstellerunabhängiges Datenmodell für den gesamten Lebenszyklus eines Bauwerks. In *Building Information Modeling: Technologische Grundlagen und industrielle Praxis* (pp. 95–146). Springer.

Borrmann, A., König, M., Koch, C., & Beetz, J. (2021). Die BIM-Methode im Überblick. *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*, 1–31. https://doi.org/10.1007/978-3-658-33361-4_1

buildingSMART. (2024). *ifcOWL*. Retrieved August 10, 2024, from https://technical.buildingsmart.org/standards/ifc/ifc-formats/ifcowl/

Clemen, C., Thurm, B., & Schilling, S. (2021). Managing and publishing standardized data catalogues to support BIM processes. *Proc. of the Conference CIB W78*, *2021*, 11–15. Retrieved July 27, 2024, from https://itc.scix.net/pdfs/w78-2021-paper-002.pdf

Costin, A., & Eastman, C. (2019). Need for Interoperability to Enable Seamless Information Exchanges in Smart and Sustainable Urban Systems. *Journal of Computing in Civil Engineering*, *33*(3), 04019008. https://doi.org/10.1061/(ASCE)CP.1943-5487.0000824

de Bruijn, J., & Heymans, S. (2010). Logical Foundations of RDF (S) with Datatypes. *Journal of Artificial Intelligence Research*, *38*, 535–568. https://doi.org/10.1613/jair.3088

Esser, S., Vilgertshofer, S., & Borrmann, A. (2022). Graph-based version control for asynchronous BIM collaboration. *Advanced Engineering Informatics*, *53*. https://doi.org/10.1016/j.aei.2022.101664

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-Based Software Architectures*. University of California, Irvine.

GraphQL. (2021). *GraphQL Specification*. Retrieved August 10, 2024, from https://spec.graphql.org/October2021/

Guo, D., Onstein, E., & Rosa, A. D. L. (2020). An Approach of Automatic SPARQL Generation for BIM Data Extraction. *Applied Sciences*, *10*(24), 87–94. https://doi.org/10.3390/app10248794

Harris, S., & Seaborne, A. (2013). *SPARQL 1.1 Query Language*. Retrieved July 24, 2024, from https://www.w3.org/TR/sparql11-query/

Hartig, O., Champin, P.-A., Kellogg, G., & Seaborne, A. (2024). *RDF 1.2 Concepts and Abstract Syntax*. Retrieved July 24, 2024, from https://www.w3.org/TR/rdf12-concepts/

Hartig, O., Taelman, R., Gregory, W., Pellissier, T., & Seaborne, A. (2024). *SPARQL 1.2 Query Language*. Retrieved August 5, 2024, from https://www.w3.org/TR/sparql12-query/

HyperGraphQL. (2021). *HyperGraphQL*. Retrieved August 10, 2024, from https://github.com/hypergraphql/hypergraphql

ISO 10303-21: 2016. *Industrial Automation Systems And Integration - Product Data Representation And Exchange - Part 21: Implementation Methods: Clear Text Encoding Of The Exchange Structure* (Standard). Geneva, CH, 2016, March.

ISO 19650-1: 2018. *Organization and digitization of information about buildings and civil engineering works, including building information modelling (BIM) — information management using building information modelling* (Standard). Geneva, CH, 2018, December.

Jaskula, K., Kifokeris, D., Papadonikolaki, E., & Rovas, D. (2024). Common data environments in construction: state-of-the-art and challenges for practical implementation. *Construction Innovation*. https://doi.org/10.1108/CI-04-2023-0088

Jaskula, K., Papadonikolaki, E., & Rovas, D. (2023). Comparison of current common data environment tools in the construction industry. *Computing in Construction*. https://doi.org/10.35490/EC3.2023.315

Jørgensen, K. A., Skauge, J., Christiansson, P., Svidt, K., Sørensen, K. B., & Mitchell, J. (2008). Use of IFC Model Servers. *Modelling Collaboration Possi-*

bilities in Practice. Retrieved July 4, 2024, from http://www.perchristiansson.se/reports/2008_Ifc_model_server.pdf

Kamateri, E., Kalampokis, E., Tambouris, E., & Tarabanis, K. (2014). The linked medical data access control framework. *Journal of Biomedical Informatics*, *50*, 213–225. https://doi.org/10.1016/j.jbi.2014.03.002

Kiviniemi, A., Fischer, M., & Bazjanac, V. (2005). Integration of multiple product models: IFC model servers as a potential solution. *Proc. of the 22nd CIB-W78 Conference on Information Technology in Construction*, 37–40.

Li, H., Liu, H., Liu, Y., & Wang, Y. (2016). An Object-Relational IFC Storage Model Based on Oracle Database. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *41*, 625–631. https://doi.org/10.5194/isprs-archives-XLI-B2-625-2016

Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc. Retrieved July 1, 2024, from https://pepa.holla.cz/wp-content/uploads/2016/01/REST-API-Design-Rulebook.pdf

OWL Working Group. (2012). *Web Ontology Language — w3.org*. Retrieved July 21, 2024, from https://www.w3.org/OWL/

Pauwels, P. (2023). *IFCtoRDF*. Retrieved July 24, 2024, from https://github.com/pipauwel/IFCtoRDF

Pauwels, P., McGlinn, K., Törmä, S., & Beetz, J. (2018). Linked Data. *Building Information Modeling: Technology Foundations and Industry Practice*, 181–197. https://doi.org/10.1007/978-3-319-92862-3_10

Pauwels, P., & Terkaj, W. (2016). EXPRESS to OWL for construction industry: Towards a recommendable and usable ifcOWL ontology. *Automation in Construction*, *63*, 100–133. https://doi.org/10.1016/j.autcon.2015.12.003

Pauwels, P., Zhang, S., & Lee, Y.-C. (2017). Semantic Web Technologies in AEC Industry: A Literature Overview. *Automation in Construction*, *73*, 145–165. https://doi.org/https://doi.org/10.1016/j.autcon.2016.10.003

Postman. (2023). *State of the API Report*. Retrieved August 10, 2024, from https://www.postman.com/state-of-api/api-global-growth/#api-global-growth

Preidel, C., Borrmann, A., Mattern, H., König, M., & Schapke, S.-E. (2018). Common Data Environment. *Building Information Modeling: Technology Foundations and Industry Practice*, 279–291. https://doi.org/10.1007/978-3-319-92862-3_15

Prud'hommeaux, E., & Carothers, G. (2014). *RDF 1.1 Turtle - Terse RDF Triple Language*. Retrieved August 5, 2024, from https://www.w3.org/TR/turtle/

Rasmussen, M. H., Lefrançois, M., Schneider, G. F., & Pauwels, P. (2021). BOT: The Building Topology Ontology of the W3C Linked Building Data Group. *Semantic Web*, *12*(1), 143–161. https://doi.org/10.3233/SW-200385

Schapke, S.-E., Beetz, J., König, M., Koch, C., & Borrmann, A. (2021). Prinzipien und Techniken der modellgestützten Zusammenarbeit. *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*, 309–333. https://doi.org/10.1007/978-3-658-33361-4_15

Solihin, W., & Eastman, C. (2016). A Simplified BIM Model Server on a Big Data Platform. *Proceedings of the 33rd CIB W78 Conference*. Retrieved July 26, 2024, from https://itc.scix.net/pdfs/w78-2016-paper-034.pdf

Soman, R. K., & Whyte, J. K. (2020). Codification Challenges for Data Science in Construction. *Journal of Construction Engineering and Management*, *146*(7). https://doi.org/10.1061/(ASCE)CO.1943-7862.0001846

Taelman, R., Vander Sande, M., & Verborgh, R. (2018). GraphQL-LD: Linked Data Querying with GraphQL. *ISWC2018, the 17th International Semantic Web Conference*, 1–4. Retrieved August 3, 2024, from https://ceur-ws.org/Vol-2180/paper-65.pdf

Taelman, R., Vander Sande, M., & Verborgh, R. (2019). Bridges between GraphQL and RDF. *W3C Workshop on Web Standardization for Graph Data*, 4–7. Retrieved July 28, 2024, from https://www.w3.org/Data/events/data-ws-2019/assets/position/Ruben%20Taelman.pdf

Tauscher, H., & Crawford, J. (2018). Graph representations and methods for querying, examination, and analysis of IFC data. *eWork and eBusiness in Architecture, Engineering and Construction: Proceedings of the 12th European Conference on Product and Process Modelling (ECPPM 2018), September 12-14, 2018, Copenhagen, Denmark*, 421. https://doi.org/10.1201/9780429506215-53

Verborgh, R. (2018). *Designing a Linked Data developer experience*. Retrieved August 10, 2024, from https://ruben.verborgh.org/blog/2018/12/28/designing-a-linked-data-developer-experience/

Villata, S., Delaforge, N., Gandon, F., & Gyrard, A. (2011). An Access Control Model for Linked Data. In R. Meersman, T. Dillon, & P. Herrero (Eds.), *On the Move to Meaningful Internet Systems: OTM 2011 Workshops* (pp. 454–463). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-25126-9_57

Werbrouck, J., Senthilvel, M., Beetz, J., Bourreau, P., & Van Berlo, L. (2019). Semantic query languages for knowledge-based web services in a construction context. *26th International Workshop on Intelligent Computing in Engineering, EG-ICE 2019*, *2394*. Retrieved August 1, 2024, from https://ceur-ws.org/Vol-2394/paper03.pdf

Werbrouck, J., Senthilvel, M., Beetz, J., & Pauwels, P. (2019). Querying Heterogeneous Linked Building Data with Context-Expanded GraphQL Queries. *7th International Workshop on Linked Data in Architecture and Construction*,

21–34. Retrieved July 6, 2024, from https://ceur-ws.org/Vol-2389/02paper.pdf

Zhang, C., Beetz, J., & de Vries, B. (2018). BimSPARQL: Domain-specific functional SPARQL extensions for querying RDF building data. *Semantic Web*, *9*(6), 829–855. https://doi.org/10.3233/SW-180297

Zhao, Q., Li, Y., Hei, X., & Yang, M. (2020). A Graph-Based Method for IFC Data Merging. *Advances in Civil Engineering*, *2020*. https://doi.org/10.1155/2020/8782740

Zhu, J., Wu, P., & Lei, X. (2023). IFC-graph for facilitating building information access and query. *Automation in Construction*, *148*, 104778. https://doi.org/10.1016/j.autcon.2023.104778

Normen

ISO 10303-21: 2016. *Industrial Automation Systems And Integration - Product Data Representation And Exchange - Part 21: Implementation Methods: Clear Text Encoding Of The Exchange Structure* (Standard). Geneva, CH, 2016, March.

ISO 19650-1: 2018. *Organization and digitization of information about buildings and civil engineering works, including building information modelling (BIM) — information management using building information modelling* (Standard). Geneva, CH, 2018, December.

Declaration

I hereby affirm that I have independently written the thesis submitted by me and have not used any sources or aids other than those indicated.

Location, Date, Signature