2025 European Conference on Computing in Construction

CIB W78 Conference on IT in Construction

Porto, Portugal
July 14-17, 2025

IFC2GRAPHQL: SCHEMA MAPPING AND AUTOMATED API CONSTRUCTION FROM IFC TO GRAPHQL

Nepomuk Wolf[1,3], Sebastian Esser[1,2,3], and André Borrmann[1,3]
[1] Chair of Computing in Civil and Building Engineering,
[2] Chair of Computational Modeling and Simulation,
[3] TUM Georg Nemetschek Institute,
School of Engineering and Design, Technical University of Munich, Germany

Abstract

Large-scale building projects involve extensive information exchange, typically conducted through file-based formats. However, many use cases require only a small subset of this information. To prevent over- or underfetching, enabling element-level access to building model data is essential. Standardized, web-based interfaces offer a promising solution. GraphQL is a well-established HTTP-based protocol for querying complex object-oriented data models in a flexible manner. This paper proposes a mapping from the IFC data model specified in the EXPRESS modeling language to a GraphQL schema, allowing access to building models via a GraphQL API over the web. A prototypical API implementation demonstrates the feasibility of this approach, supporting automatic API generation and enabling efficient object-level access to existing models.

Introduction

Large-scale building projects require extensive exchange of model information among stakeholders. Due to reliance on proprietary software across disciplines, interoperability issues and incompatibility between data models are common.

The Industry Foundation Classes (IFC) standard addresses these issues by offering an open, standardized format for cross-disciplinary information exchange. Its data model is comprehensive, covering both geometric and semantic building information. However, models are typically exchanged as large, monolithic files, resulting in long transfer times and poor traceability of changes (Esser, 2024).

Many use cases, however, require only specific subsets of information. While Model View Definitions (MVDs) can help reduce schema complexity, they do not support partial model exchange. Currently, there is no widely adopted method for accessing fine-grained data such as individual building elements.

BIM maturity level 3, as outlined in DIN EN ISO 19650-1 (2018) and DIN EN ISO 19650-2 (2018), envisions object-level interaction with building model data to overcome this limitation (Esser et al., 2023). Achieving this vision requires standardized interfaces that enable fine-grained access to specific information within a building model rather than exchanging entire datasets. At the same time, well-defined vendor-neutral exchange data models like IFC must be acknowledged to increase acceptance of component-based approaches. Ultimately, object-level information exchange can bring massive benefits to various application scenarios, each requiring different subsets of data tailored to their needs.

To achieve the aforementioned vision, this paper aims to introduce web-based, fine-grained access to BIM models encoded in IFC without modifying or redesigning the underlying data model.

GraphQL, a flexible query language and API paradigm, has emerged as an alternative to REST (Brito and Valente, 2020). This paper examines the suitability of GraphQL as an interface to IFC models and addresses the following research questions: **RQ1:** Is mapping the IFC schema, defined in EXPRESS, feasible to a GraphQL schema?

RQ2: Can GraphQL be effectively used to query IFC models?

To answer RQ1, we developed a conversion method and tool to map the IFC schema to GraphQL. For RQ2, we implemented a prototypical API and evaluated its ability to handle typical query scenarios and the complexity of resulting GraphQL queries.

We begin by reviewing related work on GraphQL and alternative methods of accessing granular building model information, like Linked Data. Then, we present our mapping methodology, followed by implementation details of a prototypical API, evaluation, and a discussion of limitations.

Background

GraphQL is a query language and server-side runtime (Hartig and Pérez, 2018) that offers an alternative to REST by addressing common limitations such as overfetching and underfetching, the need for multiple roundtrips to retrieve related data, and the need to manage numerous endpoints required for fine-grained resource access (Quiña-Mera et al., 2023).

Originally developed by Facebook in 2012 and released publicly in 2015[1], GraphQL has since become a widely adopted API paradigm. Unlike REST, which exposes resources via URLs, GraphQL defines resources through a

[1] `https://github.com/graphql`, accessed: 15.12.2024

schema, forming a structured entity graph. This schema outlines all accessible objects, the available queries, and their interrelationships (GraphQL Foundation, 2021).
A typical GraphQL API consists of three core components: the schema, which describes the available resources and their relationships as an entity graph; the queries, which define entry points for clients to access that graph; and the resolver functions, which link the fields in the schema to actual data sources. This structure is illustrated in Figure 1.

Related Works

Accessing model information at the element level has long been a goal in the AEC industry, emphasized in BIM maturity level 3 (Bew and Richards, 2008). As part of IFC 5 development, van Berlo et al. (2021) highlight the need for fine-grained access to support partial model updates, filtering, low-latency exchange, and machine learning applications, advocating a move away from file-based information silos. Both indicating that fine-grained data access and exchange will be an integral part of future development in this area.

One way the requirement for granular access has been tackled in the AEC industry is by using Linked Data and other Semantic Web technologies. Significant efforts have been made to map the IFC data model (written in EXPRESS) to OWL and convert STEP instance files to RDF for Linked Data integration.
Linked Data and Semantic Web (accompanied by technologies like RDF, OWL, and SPARQL), pioneered by Berners-Lee et al. (2001), provide robust querying and inference capabilities. These have been widely applied in the building sector (Pauwels et al., 2018), as seen in ifcOWL (Beetz et al., 2009) and the Building Topology Ontology (BOT) (Rasmussen et al., 2021). The basic idea is a transformation of the data model (e.g., the IFC schema) into an OWL ontology and representing the instance model data in the form of RDF graphs. This opens up the entire Semantic Web tool stack with powerful inference and querying capabilities. Once a model is converted to an RDF representation, SPARQL, the dedicated and official query language for RDF graphs, can interact with the model (Zhang et al., 2018).
However, Linked Data technologies present challenges, including the steep learning curve of SPARQL and RDF, the need to convert existing models to RDF, the expertise required to manage ontologies, and difficulties in data serialization (Pauwels et al., 2015). Pauwels et al. (2015) point out and discuss the difficulties that arise when working with lists in RDF. These accessibility issues have been addressed, for example, by using GraphQL instead of SPARQL to interact with building models in the RDF (ifcOWL) format (Angele et al., 2022; Taelman et al., 2018). Taelman et al. (2019) compare different approaches to integrate GraphQL with RDF data, reflecting ongoing efforts to make Semantic Web technologies more user-friendly. Moreover, web developers are generally more familiar with JSON and REST than with RDF, Linked Data and SPARQL. This makes GraphQL a viable alternative for scenarios where the power of Semantic Web technologies is not required, produces overly complex solutions or might not be viable due to incompatible data format of the resources.
Afsari et al. (2017) proposed a JSON serialization schema for IFC to enable web-based data exchange and improve accessibility for BIM applications. BIMserver.org (Beetz et al., 2010) also uses JSON to represent IFC models. Since GraphQL closely aligns with the structure of JSON and usually uses JSON as the exchange format for query results, our approach shares similarities in mapping the IFC schema to a data format frequently used in web applications.
The application of GraphQL is slowly emerging in the AEC industry (e.g., as an alternative to SPARQL or in commercial tools like Autodesk's GraphQL interface as part of its API ecosystem[2]). However, there are still few examples of the application of GraphQL to topics in the AEC industry found in the literature, e.g., Werbrouck et al. (2019b,a).

Besides being related to SPARQL and Linked Data, GraphQL is often compared to REST as a web API paradigm. Lawi et al. (2021) evaluate both approaches across four metrics (response time, throughput, CPU load, memory consumption). Their findings suggest that REST outperforms GraphQL in response time and throughput, whereas GraphQL is more efficient in memory and CPU usage, making it ideal for dynamic data requirements. Vesić and Kojić (2020) demonstrate GraphQL's ability to reduce HTTP requests, significantly improving request execution speed in real-world scenarios.
Conversely, Vadlamani et al. (2021) report comparable response times between GraphQL and REST for simple use cases, while noting GraphQL's advantages in developer experience, such as strong typing, type validation, and inline documentation. The heterogeneous results of studies comparing both approaches show that comparisons depend heavily on the specific context and requirements. When comparing both approaches (especially regarding efficiency), it is easily possible to create test cases that favor either. Despite technical differences, both are usually served over HTTP, which means that the behavior regarding the general communication protocol is quite similar. Generally, GraphQL seems to be suitable for highly interconnected information, which would lead to multiple round-trips using conventional REST architecture.

Methodology

The primary objective of this paper is to develop an API that enables fine-grained access to Building Information Modeling (BIM) data over the web. This approach aims to address a key limitation in current practices, where file-based exchanges dominate, even though specific use cases

[2] https://aps.autodesk.com/en/docs/fdxgraph/v1/developers_guide/overview/

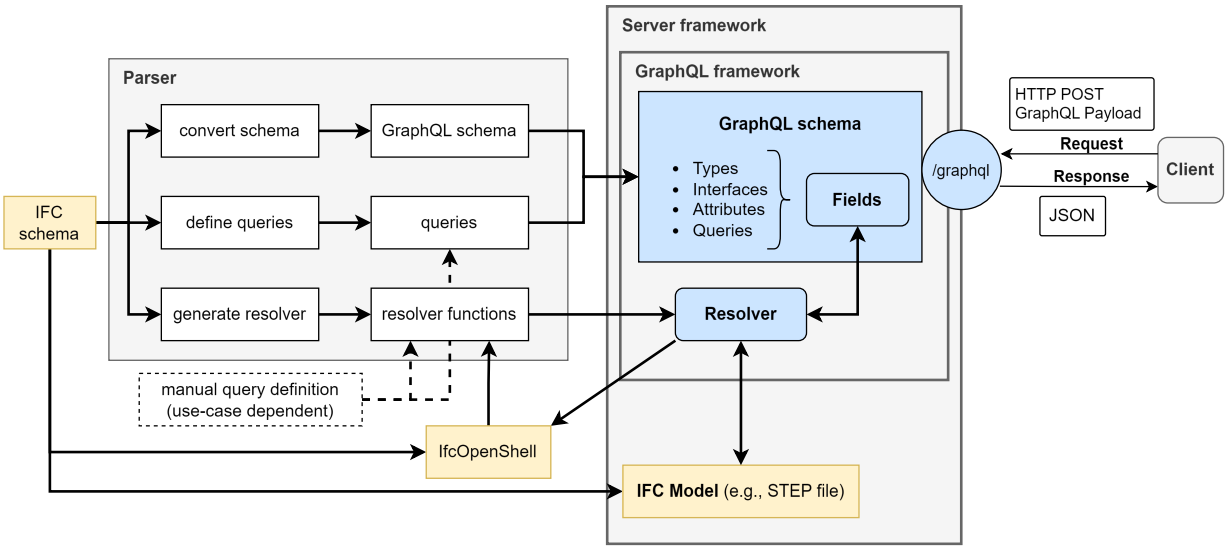

Figure 1: General methodology of the approach and architecture of a GraphQL API.

often require only a small subset of information.

In most scenarios, manually designing a GraphQL schema is the preferred approach, as it allows for tailored solutions that eliminate unnecessary overhead and focus on specific use cases. However, this paper explores the feasibility of generating a GraphQL API automatically from an existing data model.

The Industry Foundation Classes (IFC) schema[3], defined in the EXPRESS modeling language, serves as the foundation for this approach. The proposed framework (illustrated in Figure 1) consists of a parser that takes a specific IFC schema[4] as input and automatically generates a GraphQL API (including the GraphQL schema, queries, and resolver functions).

Since IFC is modeled using the EXPRESS data modeling language, a reasonable mapping between EXPRESS and GraphQL is required. This mapping is unidirectional, as there is currently no apparent advantage in converting a GraphQL schema back into an EXPRESS representation. Consequently, not all aspects of the IFC schema need to be represented in the GraphQL schema. For instance, rules and functions defined in the IFC schema are excluded since (1) they cannot be represented in a GraphQL schema and would need to be included on the resolver level and (2) they are more relevant for applications exporting IFC models than for querying. We also introduce certain simplifications, where appropriate, which are explained in more detail later.

GraphQL, as a data modeling language, offers less expressiveness than EXPRESS. Consequently, certain concepts in EXPRESS may not be fully translatable into GraphQL. The following sections detail the mapping process for specific data types and structures from EXPRESS to GraphQL, including simplifications and other limitations. The implementation is available on GitLab [5].

Simple data types serve as the basic building blocks of the schema, defining scalar data entries. The mapping between EXPRESS and GraphQL scalar types is summarized in Table 1 and involves certain simplifications due to the smaller set of scalar types supported in GraphQL. For example, the Binary type (used in IFC entities such as *IfcBlobTexture* and *IfcPixelTexture*) has no direct equivalent and must be mapped to the String type. Similarly, the Logical type in EXPRESS extends Boolean logic by allowing an additional "undefined" state. In GraphQL, this can be partially replicated by defining an *optional* Boolean field, however, this approach cannot distinguish between a deliberately undefined value and a field that is simply omitted.

Notably, the IfcOpenShell library follows a similar mapping strategy when converting EXPRESS types to Python[6]. This alignment is beneficial, as IfcOpenShell was also used in this work to parse the EXPRESS schema and handle instance model files.

The IFC schema employs **type declarations**, which often serve as aliases for simple types, occasionally with additional constraints. For example:

```
TYPE IfcLabel = STRING (255);
END_TYPE;
```

In some cases, a type declaration references another named type rather than a simple type. This referenced type ultimately resolves to a simple type. For instance:

```
TYPE IfcPositiveInteger = IfcInteger;
  WHERE
```

[3] https://technical.buildingsmart.org/standards/ifc/ifc-schema-specifications/, accessed: 15.01.2025

[4] Supported IFC versions: IFC2X3, IFC4, IFC4X1, IFC4X2, IFC4X3, IFC4X3_ADD1, IFC4X3_ADD2, IFC4X3_TC1

[5] https://gitlab.lrz.de/wolf-nepomuk/ifc2gql

[6] https://github.com/IfcOpenShell/IfcOpenShell/blob/84fc198bbec5c98b12bdd6a5b68a34c0b2153c4b/src/ifcopenshell-python/ifcopenshell/validate.py#L43

Table 1: Mapping of data types from EXPRESS to GraphQL

EXPRESS	GraphQL
Number	Float
Real	Float
Integer	Int
String	String
Boolean	Boolean
Logical	Boolean
Binary	String

```
    WR1 : SELF > 0
END_TYPE;
```

with:

```
TYPE IfcInteger = INTEGER;
END_TYPE;
```

In GraphQL, such types could be represented as custom scalars, which would involve implementing additional logic for serialization, de-serialization, and validation, resulting in a concise definition in the schema:

```
scalar IfcLabel
```

However, this method presents a significant limitation: GraphQL unions cannot include scalars as members; they only include types. This restriction becomes problematic when representing IFC SELECT statements, which are commonly used in IFC and are mapped to unions in GraphQL.
To address this limitation, type declarations in EXPRESS are mapped to types in GraphQL. Although this approach increases schema complexity, it offers flexibility to include these types in unions (which will be introduced in more detail later in this section). Additionally, it better aligns with the structure of the original IFC EXPRESS schema, where types often act as wrappers for scalar values with added constraints.
Following this design, IfcLabel, for example, is represented in GraphQL as:

```
type IfcLabel {
    value: String
}
```

This design eliminates the need for custom scalar logic while ensuring compatibility with unions and maintaining fidelity to the original EXPRESS schema.

Aggregations are an essential feature of the IFC schema and present challenges when integrating with Linked Data technologies (Pauwels et al., 2015). EXPRESS defines four aggregation types: LIST (ordered, variable size), ARRAY (ordered, fixed size), SET (unordered, no duplicates), and BAG (unordered, allows duplicates). Of these, LIST and SET are the most commonly used, while BAG is unused and ARRAY appears only in specific IFC 4 elements, such as *OffsetValues* in *IfcMaterialLayerWithOffsets* and *IfcMaterialProfileWithOffsets*.
A LIST supports ordering and optional cardinality constraints, with a fixed lower bound and an optional upper bound. A SET, by contrast, is unordered and disallows duplicates. In GraphQL, however, all EXPRESS aggregation types must be mapped to a single construct: the LIST. This simplification leads to a loss of semantic detail such as ordering, fixed sizes, and uniqueness constraints, making the mapping one-way and non-reversible.
Moreover, GraphQL lacks native support for expressing list length or cardinality, features often used in IFC. While this is not critical for queries (as IFC models can be pre-validated), it poses challenges for mutations, where input must be validated against expected bounds. Since these constraints cannot be encoded at the schema level, they must be enforced in the resolver logic.

Entities are arguably the most essential elements provided by the EXPRESS language. They serve as the primary building blocks of schemas, supporting inheritance and encapsulating properties. The properties of entities can have various types, including scalars, named types, and other entities. Entities can furthermore be declared ABSTRACT, which means the entity cannot be instantiated (but can be used as a super type).
Non-abstract entities map naturally to the concept of types in GraphQL, making a translation straightforward. Abstract entities are represented as interfaces, a concept similar to abstract super types. They serve the purpose of defining shared properties between different types and cannot be instantiated themselves. An example of this mapping is illustrated in figure 2.

The IFC schema employs a rich **inheritance** hierarchy, allowing entities to inherit properties and behavior from parent entities. GraphQL, however, does not support inheritance in the same way. Although interfaces can emulate some inheritance-like behavior, they come with limitations. For instance, interfaces cannot be instantiated, which makes them suitable for abstract entities such as *IfcRoot* or *IfcElement*, but unsuitable for concrete types like *IfcSlab* or *IfcWindow*. Additionally, GraphQL types can only implement interfaces (not other types) making it impossible to directly represent inheritance between non-abstract super-types and their subtypes.
These constraints complicate the direct mapping from EXPRESS to GraphQL. For example, *IfcWall* is a concrete entity and also serves as a super-type of *IfcWallStandardCase*. If *IfcWall* is modeled as a type, it cannot be extended by its subtypes; if it is modeled as an interface, it cannot be instantiated.
To resolve this, we introduced supplementary interfaces for all non-abstract entities with subtypes. While this approach increases schema size due to duplication, it ac-

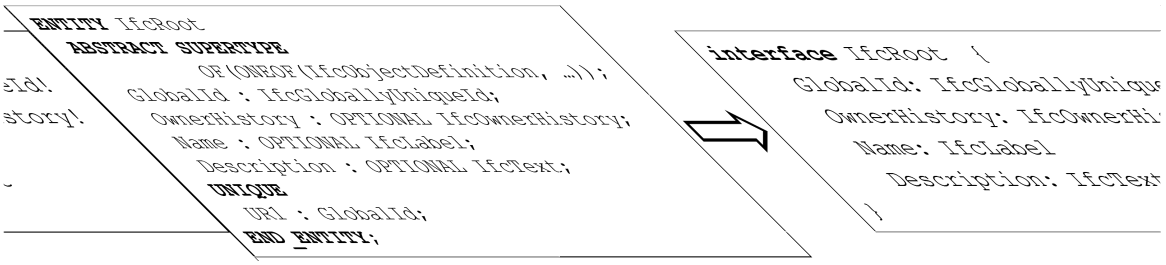

Figure 2: Mapping of an abstract EXPRESS entity to a GraphQL interface.

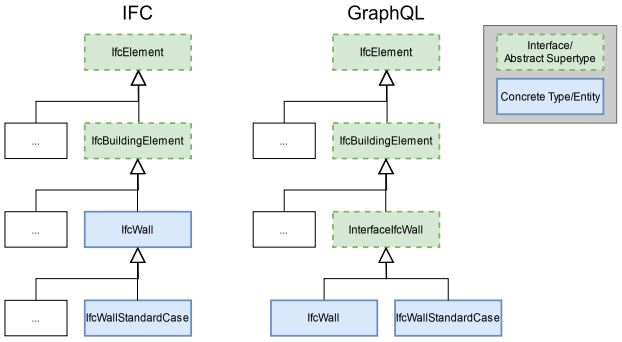

Figure 3: Use of interfaces for inheritance.

curately preserves the inheritance relationships present in the original schema. This strategy is illustrated in Figure 3.

Attributes define the properties of an entity in IFC, specifying both the name and data type. They are inherited from all super types of the entity, forming a hierarchical structure of properties. In GraphQL, this concept is closely mirrored by fields, which define the properties of types. GraphQL fields offer considerable flexibility in specifying data types. They can represent scalars, types, unions, enumerations, and interfaces. This flexibility ensures that mapping attributes from EXPRESS to GraphQL fields is straightforward, with no significant challenges.

IFC makes extensive use of inverse attributes and objectified relationships, i.e., attributes that are not directly attached to an entity but refer to it through one of their own attributes. However, every connection that should be accessible through GraphQL has to be made explicit. This means the concept of inverse attributes is not available, and all inverse attributes have to be made explicit.

The EXPRESS language frequently uses **SELECT statements** to define a type that can take values from a predefined set of types, which may include scalars, named types, or entities. For example, the *IfcSimpleValue* type can represent any of the following: *IfcInteger*, *IfcReal*, *IfcBoolean*, etc.

In GraphQL, the closest equivalent is a union, which allows a field to return one of several object types. However, directly mapping SELECT statements to unions presents challenges due to GraphQL's specification: unions cannot include other unions, interfaces, or scalars. As a result, nested SELECTs must be flattened, abstract types (modeled as interfaces) must be replaced by listing their concrete implementations, and primitive types like IfcInteger and IfcReal must be represented as object types rather than custom scalars.

To accommodate these constraints, the mapping avoids modeling primitive types as scalars and flattens both SELECT statements and inheritance hierarchies within unions.

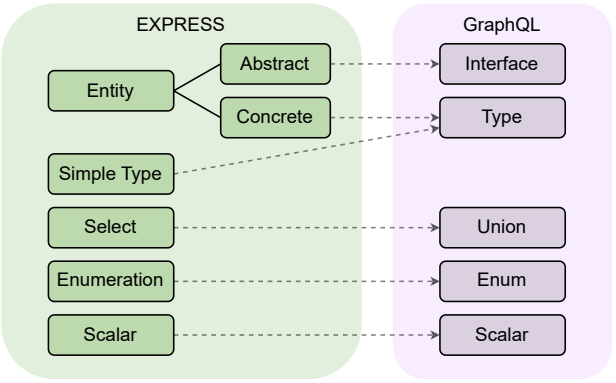

Figure 4: General mapping between EXPRESS and GraphQL data structures.

The mapping principles introduced in this chapter were used to implement a parser that takes an IFC schema and generates (1) a GraphQL schema, (2) simple queries as entry points, and (3) the necessary resolver functions. Constructs from EXPRESS were thereby mapped to appropriate structures in GraphQL as described above and illustrated in Figure 4. The next section illustrates how this output can be used to generate a GraphQL API to enable fine-grained access to IFC building models.

Case Study

The parser output (see previous section) was used to create a GraphQL API by loading the schema and resolvers into a GraphQL framework, which was then exposed via a simple web server (see Figure 1).

Since the resolvers use IfcOpenShell[7], available as a Python package, the framework is implemented entirely in Python. Flask serves as the web framework for hosting

[7] `https://docs.ifcopenshell.org/`, accessed: 03.01.2025

Listing 1: Basic query to retrieve all slab elements.

```
query Slabs {
  getAllIfcSlabs {
    GlobalId {
      value
    }
  }
}
```

Listing 2: Query to retrieve all windows filling openings in filtered wall elements.

```
query WindowsFiltered
        ($filter: String) {
  getAllIfcWalls(filter: $filter) {
    HasOpenings {
      RelatedOpeningElement {
        ... on IfcOpeningElement {
          HasFillings {
            RelatedBuildingElement {
              ... on IfcWindow {
                GlobalId {
                  value
                }
        [ ... ]
}
{"filter": "location=\"Level␣1\",
␣␣␣␣PSet_Revit_Dimensions.Volume␣>␣5"}
```

the GraphQL endpoint and managing model and schema loading, while Ariadne is used for GraphQL integration. The resulting API enables web-based, fine-grained access to building models in the IFC format.

The evaluation of the API revealed that the effectiveness of the approach is strongly dependent on the specific information being queried. The IFC schema is not inherently optimized for querying, and query complexity ranges from simple to highly verbose depending on the data required. To illustrate this, we present three scenarios demonstrating varying query complexity (Listings 1, 2, and 3).

Query 1: *Retrieve a list of GlobalIds for all slabs in the building model.*

This is an example of a simple query that can easily be represent as a GraphQL query (listing 1) Requesting a *list of GlobalIds for all IfcSlabs* for example can be expressed in a very simple query shown in listing 1.

Query 2: *Retrieve the GUID of all windows contained in all walls at a specific building level, whose volume is bigger than* $5m^2$

The second query we present is more complex, yet it remains straightforward to express in GraphQL (listing 2).

Listing 3: Querying a parameter from a property set of a specific element.

```
query GetById($Id: String) {
 getById(id: $Id) {
  ... on IfcWall {
   IsDefinedBy {
    ... on IfcRelDefinesByProperties {
     RelatingPropertyDefinition {
      ... on IfcPropertySet {
       Name {
         value
       }
       HasProperties {
        ... on IfcPropertySingleValue {
          NominalValue {
            ... on IfcVolumeMeasure {
               value
            }
        [ ... ]
}
```

In comparison to the simpler query presented before, this query is more complex, both in terms of nesting and the use of inline fragments (indicated by the three dots "..."). Since GraphQL enforces strict typing and does not support inheritance, inline fragments are required to define the behavior for fields that return interfaces or unions (abstract entities or SELECTs in EXPRESS).

Query 3: *Retrieve the property "Volume" from the property set "PSet_Revit_Dimensions" of a specific wall identified by GlobalId.*

Although this is conceptually a straightforward extraction of a specific attribute, the resulting GraphQL query is quite verbose and requires several inline fragments. This complexity arises from the way external properties are linked to elements in the IFC through property sets.

The last example shows, that it might be necessary to extend the automatically generated schema to more explicitly include information that should be exposed through the API, which will be discussed in the subsequent section.

Discussion

This section outlines the limitations of the schema mapping methodology and the current API implementation, along with potential strategies for improvement. The EXPRESS modeling language is significantly more expressive than GraphQL, which was designed with a different goal in mind. As a result, the conversion from EXPRESS to GraphQL is inherently lossy. Aggregation types such as LIST, ARRAY, SET, and BAG are all mapped to GraphQL's single LIST type, resulting in the loss of semantics like order, uniqueness, and size constraints. Cardinality specifications and inheritance from non-abstract

types cannot be fully preserved. Additionally, due to GraphQL's limitations on union and scalar handling, EXPRESS SELECT statements must be flattened, leading to verbose union definitions. The current implementation also lacks support for mutations and includes only a limited set of basic queries.

While some of these issues (particularly those rooted in GraphQL's language constraints) can be addressed through schema extensions, custom directives, or validation logic in resolver functions, others reflect the limited scope of the current prototype. The addition of mutation support and broader query coverage is planned in future work.

GraphQL's strength lies not in raw efficiency, but in its ability to traverse interconnected data in a single request. Unlike REST, which often requires multiple roundtrips and endpoints to access related resources, GraphQL enables direct, structured queries across relationships. However, in use cases where such capabilities are unnecessary, its added complexity may not be justified.

Linked Data, mainly accessed through SPARQL, is powerful but often perceived as complex and challenging for developers unfamiliar with Semantic Web technologies. GraphQL, by contrast, is designed to be intuitive and accessible, borrowing concepts from REST (still the dominant paradigm for APIs) and usually using JSON as the serialization format for results, a well-established and widely understood standard among web developers.

While GraphQL is arguably less potent than SPARQL in terms of querying flexibility, it offers advantages that may make it a practical choice depending on the use case. Its schema is built from a small number of basic constructs, making it straightforward to understand and implement. GraphQL also supports introspection, enabling developers to explore the schema and construct queries interactively. This is further supported by a wide range of graphical tools for schema navigation and query building, enhancing usability.

The resulting GraphQL schema closely follows the structure of the IFC schema. While this alignment provides several advantages, it also introduces notable challenges.

One significant drawback is the verbosity of queries, particularly when accessing specific properties linked to objects through property sets. Since GraphQL does not support returning multiple types for a single field, queries must explicitly define the types to be retrieved. As a result, the API is highly effective for retrieving information that is directly connected to elements. However, accessing more intricate data, such as properties stored in property sets, often requires verbose queries with inline fragments to accommodate the various types specified in SELECT statements.

Building models often contain sensitive data requiring protection from unauthorized access. The current implementation lacks access control and does not address security concerns related to exposing models via a web API.

GraphQL's widespread use in industry has led to various strategies for implementing authentication and authorization, notably through fine-grained, resolver-level control. Future work will explore the applicability of these methods to the proposed API.

Conclusion

The increasing complexity of building information models necessitates interfaces that enable fine-grained, efficient access to data. This paper introduces a methodology for mapping the IFC schema, defined in the EXPRESS language, to a GraphQL schema, offering a web-based interface for querying IFC models. By automating the generation of GraphQL schemas and resolver functions, the proposed approach simplifies the creation of APIs for accessing and interacting with building models.

The resulting GraphQL API closely follows the IFC schema, which aligns with established industry standards. This adherence, while advantageous for familiarity and interoperability, introduces challenges when accessing complex or nested information, such as properties embedded in property sets. To address this, extending the automatically generated schema to expose specific attributes in a more straightforward manner could improve usability at the cost of deviating slightly from the original schema structure.

The authors plan on enhancing the methodology in the future to address current limitations, including the lack of mutation support and the need for a broader range of optimized queries.

This work contributes to advancing object-level interaction with building models and is an attempt of integrating GraphQL as a practical, standardized tool for the AECO industry. By bridging the gap between data modeling standards and modern web technologies, it is one step towards efficient and accessible data exchange in large-scale building projects.

Acknowledgement

The research presented has been financially supported by the Federal Ministry of Education and Research (Bundesministeriums für Bildung und Forschung) in the frame of the project BauPuls360 under grant number 02K23A096. The responsibility for the content of this publication lies with the authors.

During the preparation of this work, the authors utilized ChatGPT-4 to assist with writing and language refinement.

References

Afsari, K., Eastman, C. M., and Castro-Lacouture, D. (2017). JavaScript Object Notation (JSON) data serialization for IFC schema in web-based BIM data exchange. Automation in Construction, 77:24–51.

Angele, K., Meitinger, M., Bußjäger, M., Föhl, S., and Fensel, A. (2022). Graphsparql: a graphql interface for linked data. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pages 778–785.

Beetz, J., van Berlo, L., de Laat, R., and van den Helm,

P. (2010). BIMserver. org–An open source IFC model server. In Proceedings of the CIP W78 conference, volume 8.

Beetz, J., Van Leeuwen, J., and De Vries, B. (2009). IfcOWL: A case of transforming EXPRESS schemas into ontologies. Ai Edam, 23(1):89–101.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. Scientific American, 284(5):34–43.

Bew, M. and Richards, M. (2008). A Report for the Government Construction Client Group: Building Information Modelling (BIM) Working Party Strategy Paper. Technical report, Government Construction Client Group. Accessed via Academia.edu.

Brito, G. and Valente, M. T. (2020). REST vs GraphQL: A controlled experiment. In 2020 IEEE international conference on software architecture (ICSA), pages 81–91. IEEE.

DIN EN 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 – Part 1: Concepts and principles.

DIN EN ISO 19650-2 (2018). Organization and digitization of information about buildings and civil engineering works, including building information modelling (BIM) – Information management using building information modelling – Part 2: Delivery phase of the assets.

Esser, S., Vilgertshofer, S., and Borrmann, A. (2023). Reference framework enabling temporal scalability of object-based synchronization in BIM level 3 systems. In EC3 Conference 2023, volume 4, pages 0–0. European Council on Computing in Construction.

Esser, S. J. (2024). Inkrementelle Versionskontrolle verteilt vorliegender Objektmodelle im Bauwesen. PhD thesis, Technische Universität München.

GraphQL Foundation (2021). Graphql specification. `https://spec.graphql.org/October2021/`. Accessed: 2025-01-05.

Hartig, O. and Pérez, J. (2018). Semantics and complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference, pages 1155–1164.

Lawi, A., Panggabean, B. L. E., and Yoshida, T. (2021). Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. Computers, 10(11):138.

Pauwels, P., McGlinn, K., Törmä, S., and Beetz, J. (2018). Linked data. Building information modeling: Technology foundations and industry practice, pages 181–197.

Pauwels, P., Terkaj, W., Krijnen, T., and Beetz, J. (2015). Coping with lists in the ifcOWL ontology. In Proceedings of the 22nd EG-ICE Workshop 2015, 13-15 June 2015, Eindhoven, The Netherlands, pages 111–120.

Quiña-Mera, A., Fernandez, P., García, J. M., and Ruiz-Cortés, A. (2023). GraphQL: A systematic mapping study. ACM Computing Surveys, 55(10):1–35.

Rasmussen, M. H., Lefrançois, M., Schneider, G. F., and Pauwels, P. (2021). BOT: The building topology ontology of the W3C linked building data group. Semantic Web, 12(1):143–161.

Taelman, R., Vander Sande, M., and Verborgh, R. (2018). GraphQL-LD: linked data querying with GraphQL. In ISWC2018, the 17th International Semantic Web Conference, pages 1–4.

Taelman, R., Vander Sande, M., and Verborgh, R. (2019). Bridges between GraphQL and RDF. In W3C Workshop on Web Standardization for Graph Data. W3C, pages 4–7.

Vadlamani, S. L., Emdon, B., Arts, J., and Baysal, O. (2021). Can GraphQL Replace REST? A Study of Their Efficiency and Viability. In 2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), pages 10–17. IEEE.

van Berlo, L., Krijnen, T., Tauscher, H., Liebich, T., Van Kranenburg, A., and Paasiala, P. (2021). Future of the industry foundation classes: towards IFC 5. In Proc. of the 38th International Conference of CIB W, volume 78, pages 11–15.

Vesić, M. and Kojić, N. (2020). N. comparative analysis of web application performance in case of using REST versus GraphQL. In Proceedings of Fourth International Scientific Conference on Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture (ITEMA), Online-Virtual, pages 17–24.

Werbrouck, J., Senthilvel, M., Beetz, J., Bourreau, P., and Van Berlo, L. (2019a). Semantic query languages for knowledge-based web services in a construction context. In 26th International Workshop on Intelligent Computing in Engineering, EG-ICE 2019, volume 2394.

Werbrouck, J., Senthilvel, M., Beetz, J., and Pauwels, P. (2019b). Querying heterogeneous linked building data with context-expanded GraphQL queries. In 7th International Workshop on Linked Data in Architecture and Construction, pages 21–34. CEUR-WS. org.

Zhang, C., Beetz, J., and de Vries, B. (2018). BimSPARQL: Domain-specific functional SPARQL extensions for querying RDF building data. Semantic Web, 9(6):829–855.