AUTODESK. UNIVERSITY

SD318864

Unlock the Full Potential of Your MEP Data: The Case for a Unified Data Model

Will Reynolds
Hoare Lea

Learning Objectives

- Discover the uses of GraphQL in MEP, and how it can define a common data model for services to communicate
- Learn about graph databases and how they work
- Learn how to host your own cloud model data with Neo4j
- Learn how to create your own or adapt existing apps or services to use a common data model

Description

A big inefficiency in MEP design lies in communicating data between disparate design software. What if all of these applications could communicate seamlessly with a single data model? This class will showcase a solution in which disparate applications communicate via a common schema (GraphQL) and a cloud database (Neo4j).

Introducing the design-as-a-service concept, we'll consider the huge potential unlocked by moving the primary data model from Revit software to a connected cloud service. We'll see how a common data model can store data beyond that which can be stored in the Revit model, and explore how you can easily integrate your own apps and services. Examples will include integrating an electrical calc service, and how the cloud data functions as a digital twin, where live sensor data is visualized within the Forge Viewer.

Finally, we'll consider how we can work together to further define and improve the data schema as something we can all own as innovators in MEP design.

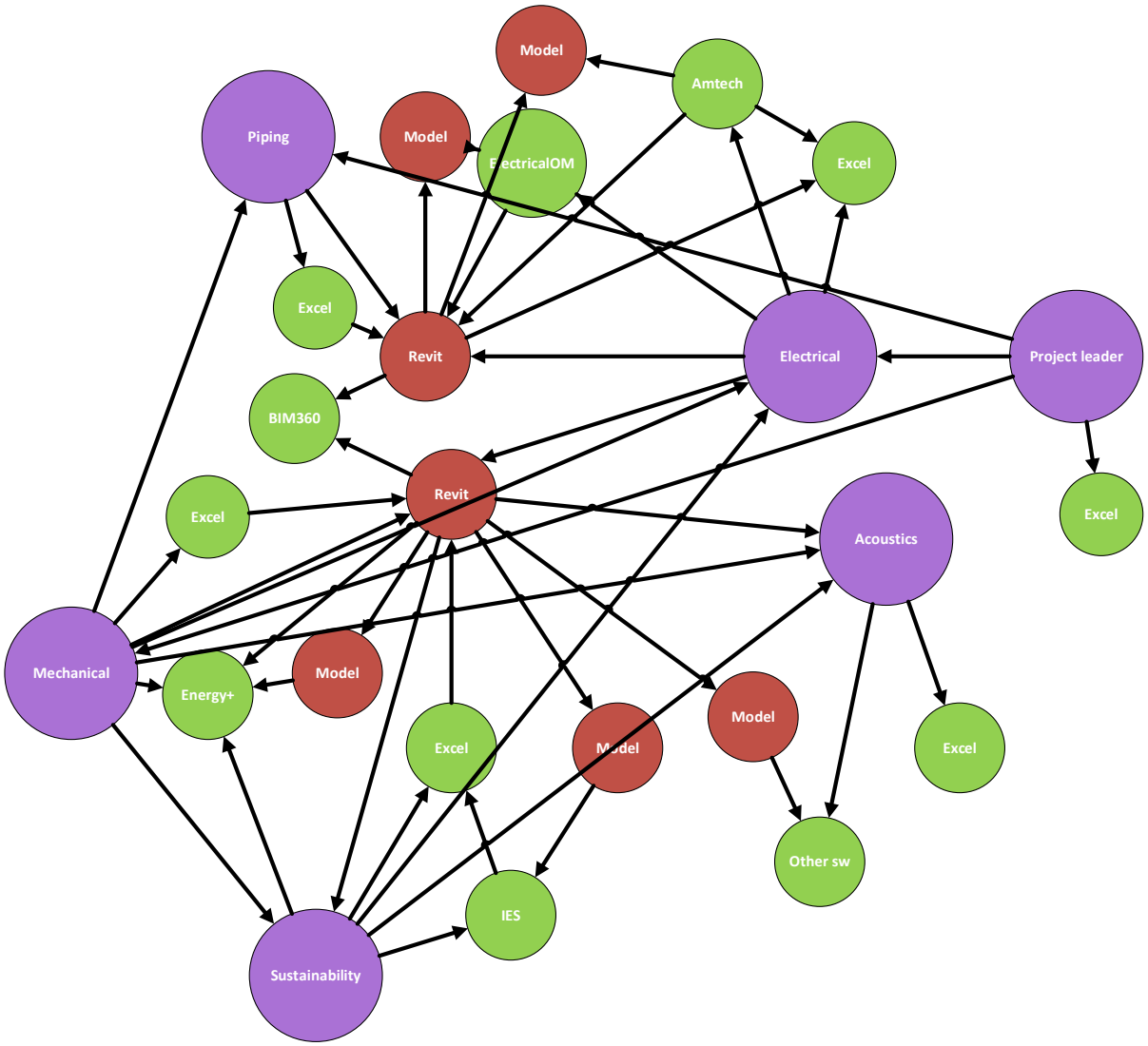
Speaker(s)

Will Reynolds

The problem with MEP data

It is often seen that each user, application and service have their own view on MEP data, and frequently create their own versions of the same model. In turn, each application might have its own way of analyzing and performing calculations on the data, often using different parameter names for the same data.

This causes significant headaches in getting one application (such a Revit) to use data created by another application (such as IES). This is represented by the image below, where each line represents flow of data or interaction with the model data.



In particular there can be heavy reliance on Excel, where important data and essential engineering calcs are siloed into disparate files, causing:

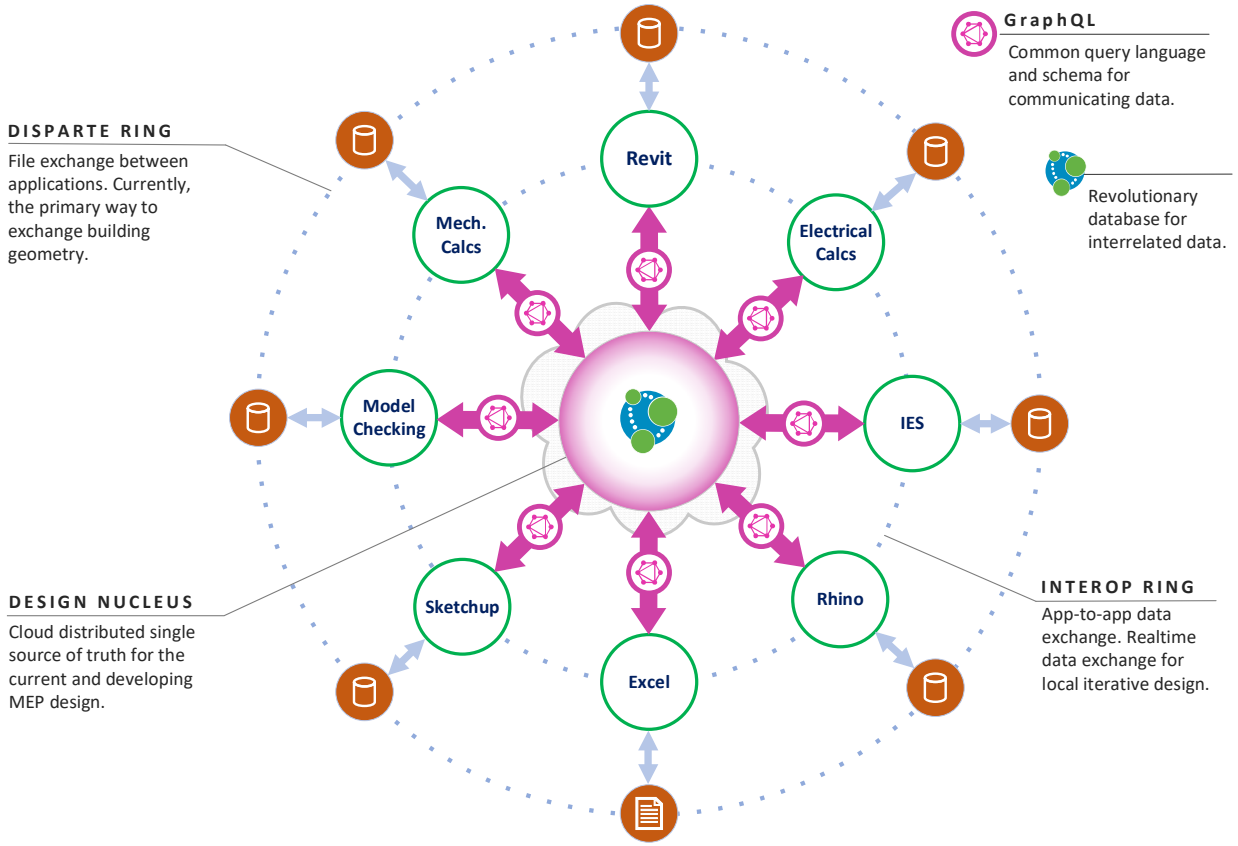
- Replication of data across models, excel files, and other application data
- Knowledge, calcs, and techniques known by individuals, lost for other teams, and other projects
- Uncontrolled divergence from tested and mandated standard calculations

While services such as BIM360 offer a common data environment with which users can collaborate with the model files, to get at the actual model data the whole model might need to be downloaded and opened in another application, when a user or service only want to interact with a specific part of the model data.

The case for a common data model

What if each application could communicate the data it needs with other applications through a common model? The barriers of presented through file interoperability are significantly reduced, allowing for a much more fluid flow of data.

In the below diagram the layers of application data flow are generalized to a series of rings.



In the outer (disparate) ring each application communicates with file-based data, there is a degree of data sharing with open file standards such as something simple like CSV to more complex data such as IFC or gbXML. In some cases, the file data might be only readable or writable by the application which created it.

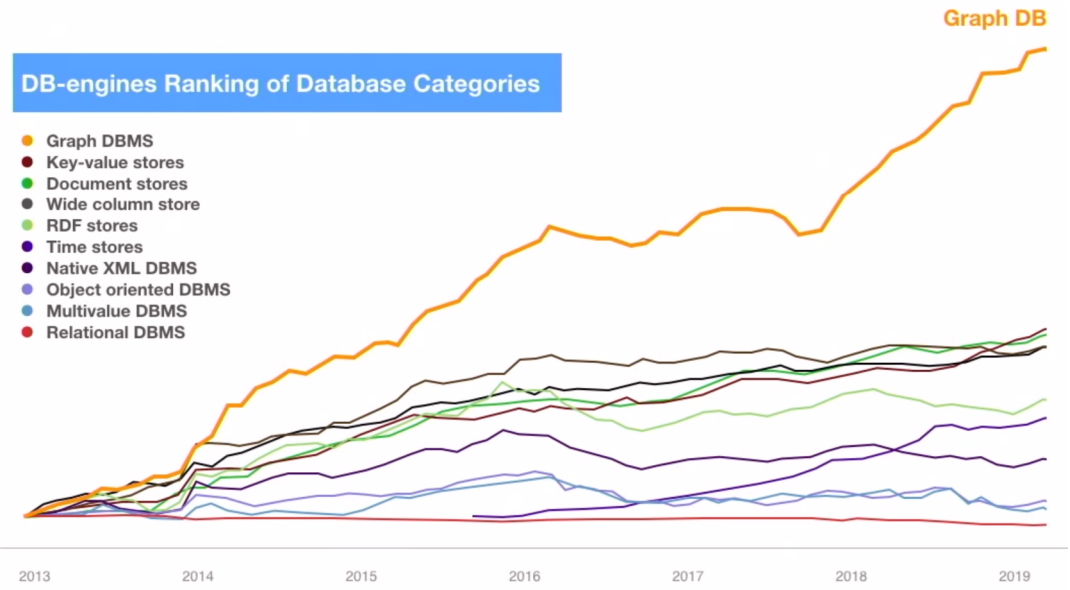
The first inner ring (Interop ring) defines the communication which takes places between the applications themselves, either natively or through add-ins utilizing the internal APIs offered by the host application. For example, using the Revit API and/or Dynamo from import and export of data with Excel, or, direct Revit interoperability with energy simulation software such as Energy+.

In the center nucleus, are two developing technologies which, essentially, are the driving force for making this whole concept possible.

Neo4j Graph Database

Graph databases have gown significant over recent years. It's now possible to design a complex and adaptable database architecture which would have been too onerous or complicated with traditional relational databases.

Graph database are growing significantly in popularity:

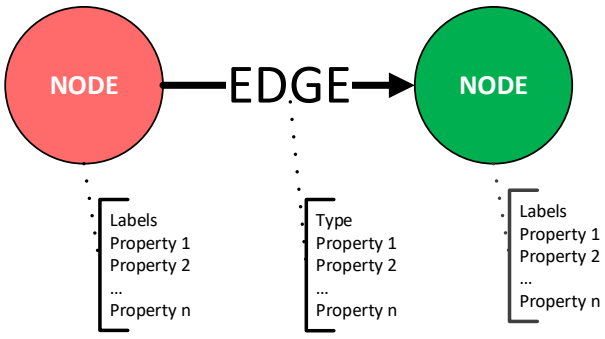


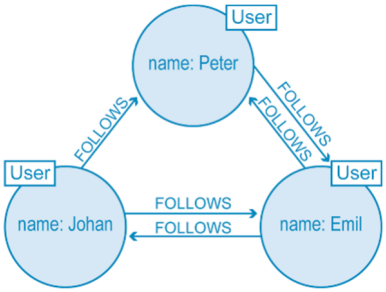
Neo4j NODES 2019 – Keynote
https://youtu.be/AfhJcyys108?t=529

For more information and learning about Neo4j: https://neo4j.com

Quick recap and comparison with RDBS

A graph database essentially consists of nodes and edges, each node can have one or more labels, and any number of properties. Each node is connected by an edge, where each edge has a type and, again any number of properties.



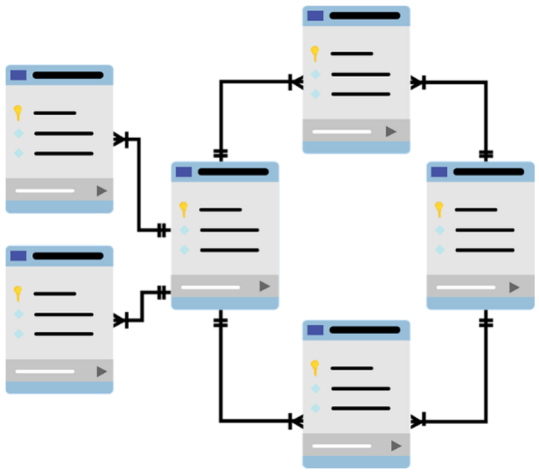


The query language of Neo4j is Cypher, to return the above graph the query could be something like this:

```
MATCH p=()-[r:FOLLOWS]->() RETURN p
```

With a graph database, the relationships are data too, sometimes this data is even more valuable than the data in the nodes themselves.

This is in contrast to a relational database where the data is stored in tables with predefined columns, and related to each other by one or more primary and foreign keys.



Querying complex relationships can often be prohibitively complicated in SQL.

GraphQL

Originally created by Facebook, GraphQL is another rapidly developing technology which seeks to simplify and enhance web API development. Essentially, it replaces many API REST endpoints traditionally used for web APIs with one single endpoint. A schema then defines the types and operations that can be used, and the data which will be returned.

A key principal with GraphQL is that only data that is requested is returned, and in the same structure as the original query. For example, a query for all buildings, their spaces, and the name of everything in each space might look like this:

Request	Response

{
  Building {
  Name
  Spaces {
  Name
  Number
  AllElements {
  Name
  }
  }
  }
}
	{
	  "data": {
	  "Building": [
	  {
	  "Name": "WTS",
	  "Spaces": [
	  {
	  "Name": "02-W4 : EC 02-W4",
	  "Number": "02-W4",
	  "AllElements": []
	  },
	  {
	  "Name": "02-V4 : Level 02 Floor Void 02-V4",
	  "Number": "02-V4",
	  "AllElements": [
	  {
	  "Name": "HL_CableTray_MediumDuty - LV"
	  },
	  {
	  "Name": "HL_CableTray_HorizontalBend : Cable Tray - LV"
	  },
	  {
	  ...

The same query in Cypher would look like this:

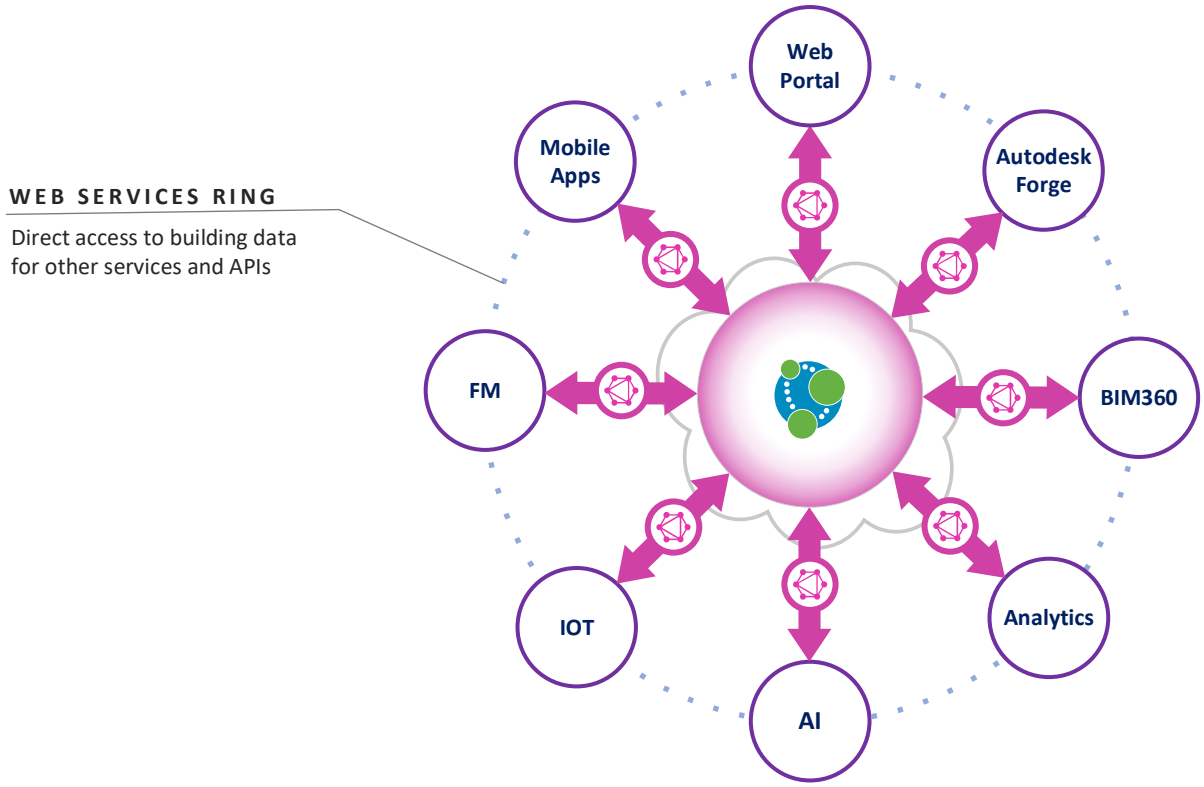
```
MATCH (b:Building)<-[:IS_IN]-(sp:Space)<-[:IS_IN_SPACE]-(elm) RETURN b.Name, sp.Name,
sp.Number, elm.Name
```

Filter, sorting, pagination is all also possible directly in GraphQL queries as part of the GraphQL spec.

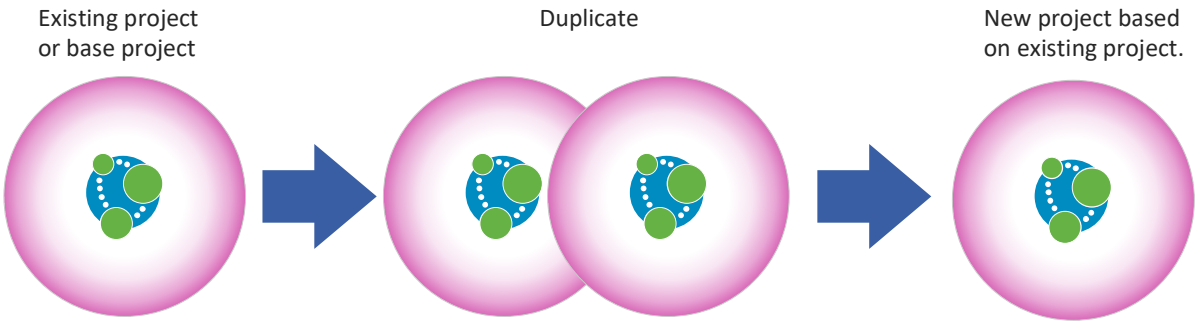
For more information and learning about GraphQL: https://graphql.org/learn/

Design as a service

With a central graph based design model, the data is now accessible to a wide range of services and applications. A bit like “Design as a service”, where the “design” part can encompass to calculations, simulations, analytics, testing and validation, and many other applications of the building data.



By virtue of a single common data schema, other potential applications emerge, such as the ability to directly compare models, or even derive new models from existing data.



The GraphQL Schema

The principal part of the case for common data model is the GraphQL schema. This consists of one or more graphql files which define the types, fields, and other key aspects in a JSON-like format.

- Object types – the entities or classes, such as Ducts, Spaces, or Models
- Fields – the property names and data types available on each type
- Arguments – Additional data which can be passed to a field.
- Queries – Predefined queries which
- Mutations - Functions which change the data, such as create a Duct, Space or model entity.

GraphQL allows each field type to be a scalar, enum or object, and singular or array thereof.

Scalar Types:

- Int
- String
- Float
- Boolean
- ID
- Custom scalar

Another key part of the GraphQL schema is introspection, this is included as part of the specification which allows applications to discover all the support types, fields and other information of the available schema.

The following sections present example parts of the building graph schema, together with visual examples and GraphQL representation. In the example graphql schemas you will see @cypher and @relation directives which are explained later in section on GRANDstack.

Each of the individual schema files represent only the relevant types of direct concern. This enables the schema to grow or shrink in flexible way and to be easily maintained or augmented. See https://www.apollographql.com/docs/apollo-server/federation/concerns/

For the full schema .graphql files, see https://github.com/willhl/BuildingGraph-Server.

Units

A specific provision has been developed for the building graph using custom scalars to support specific data units. This enables any query can request values in any supported unit by supplying the required units in the query argument, like this:

```
{
  Space{Area(unit:ft2)}
}
```

In this case the Area is returned in square feet instead of the default units of square meters.

```
{
  "data": {
    "Space": [
      {
        "Area": 11211.52469548794
      },
      ...
```

To accomplish this, the area units are defined as a custom scalar in a .graphql file:

```
scalar SquareMeters
enum AreaUnits{
   m2
   ft2
   ...
}
```

Then, in any field definition which is of square meters, in a .graphql file:

```
type Space implements AbstractElement {
 Area (unit: AreaUnits = m2): SquareMeters
 ...
}
```

This allows the default units to be understood by the client application through introspection. A custom resolver is used by the Apollo server so that it can perform unit conversion, in the index.js of the Apollo server initialization:

```
const resolvers = {
  UnitFloat : new UnitFloatScalarType("UnitFloat"),
  Meters: new UnitFloatScalarType("Meters", "m"),
  SquareMeters : new UnitFloatScalarType("SquareMeters", "m2"),
  ...
}
const schema = makeAugmentedSchema({typeDefs, resolvers});
```

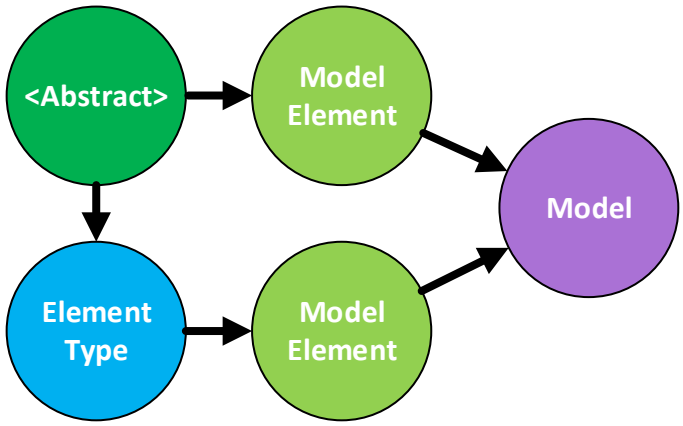
The UnitFloatScalarType class defined in the UnitFloatScalarType.js which makes use of the convert-units NPM library and extends the `GraphQLScalarType`. Refer to https://github.com/willhl/BuildingGraph-Server for more info.

Abstract elements, Types, and model elements

The data schema includes a distinction between abstract elements, and model elements. The purpose of this is to allow an abstract element to be present in zero, one, or more models, such that the design intention can form a single unified model, with loose coupling to other files and data structures.

Abstract elements in the graph are related to their respective representative model elements in other model files or model data. Model elements usually have geometry and parameter data embedded in other file formats or cloud services. For example, spaces may be exported from a Revit model to gbXML, in which case the gbXML spaces can be related to the same abstract spaces, and the same spaces present in the Revit model.

Each abstract element can be associated with one or more elements in another model, for example a Revit model where each element has an ElementId or UniqueID.

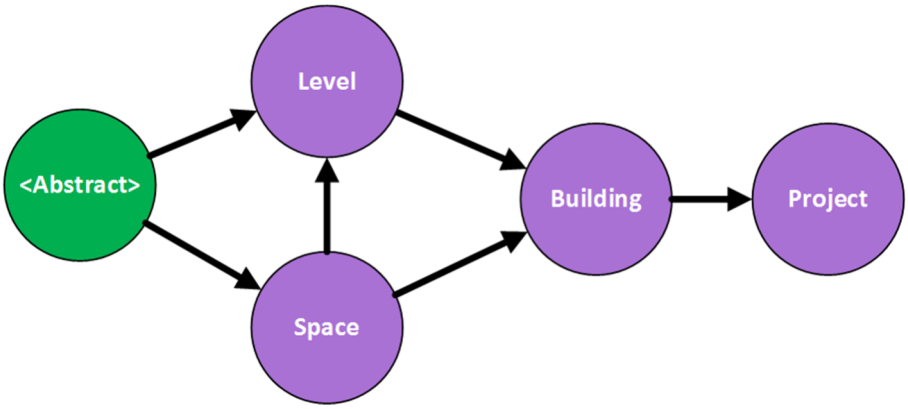


Example GraphQL Schema:

```
interface AbstractElement{
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
}

type Model {
  Project_Name: String
  URI: String
  Identity: String
  ModelElements: [ModelElement] @relation(name:"IS_IN",direction:IN)
}
```

Buildings, levels and spaces

Each abstract element is related to the space, level and building it's in

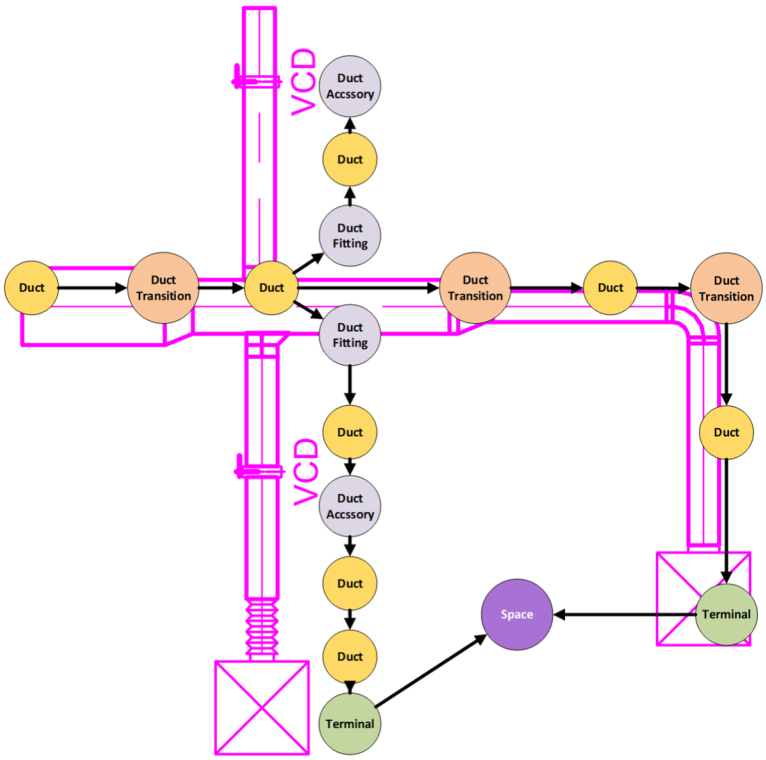


Example GraphQL Schema:

```
type Project {
  Name: String
}

 type Building {
  Name: String
  Spaces: [Space] @relation(name:"IS_IN",direction:IN)
  Models: [Model] @relation(name:"IS_OF",direction:IN)
  Levels: [Level] @relation(name:"IS_IN",direction:IN)
  Projects: [Project] @relation(name:"INCLUDES",direction:IN)
}

type Level implements AbstractElement {
  Name: String
  Id: ID!
  Elevation (unit: LengthUnits = m): Meters
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  AbstractElements: [AbstractElement] @relation(name:"IS_ON",direction:IN)
  Spaces: [Space] @relation(name:"IS_ON",direction:IN)
  Building: Building @relation(name:"IS_IN",direction:OUT)
}
```

Distribution: Ducts, Pipes and Cable Trays

Duct, pipe and cable tray networks are related by the physical flow, usually hydronic or air, which flows between them.

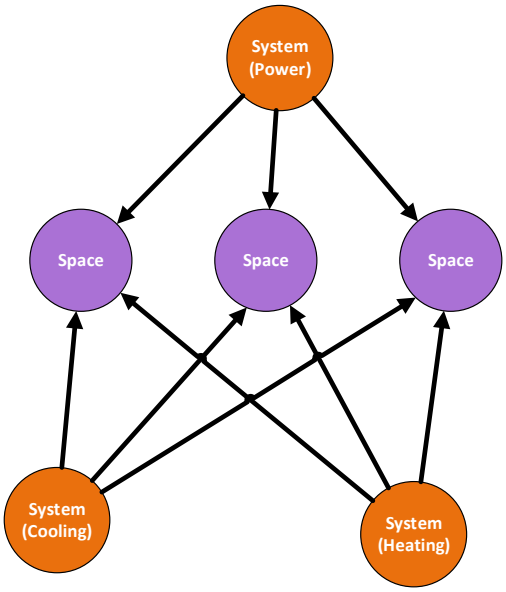


Example GraphQL Schema:

```
type Duct implements AbstractElement {
  Name: String
  Id: ID!
  LocationX (unit: LengthUnits = m): Meters
  LocationY (unit: LengthUnits = m): Meters
  LocationZ (unit: LengthUnits = m): Meters
  Area (unit: AreaUnits = m2): SquareMeters
  Width (unit: LengthUnits = m): Meters
  Height (unit: LengthUnits = m): Meters
  Flow (unit: VolumeFlowRateUnits = l_per_s): LitersPerSecond
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  Space: Space @relation(name:"IS_IN_SPACE",direction:OUT)
  FlowIn: AbstractElement @relation(name:"AIR_FLOW_TO",direction:IN)
  FlowOut: AbstractElement @relation(name:"AIR_FLOW_TO",direction:OUT)
}
```

Systems

From the schematic level to the detailed design, the graph can incorporate these relationships to enable other services to perform calculations before any 3D model information is developed.

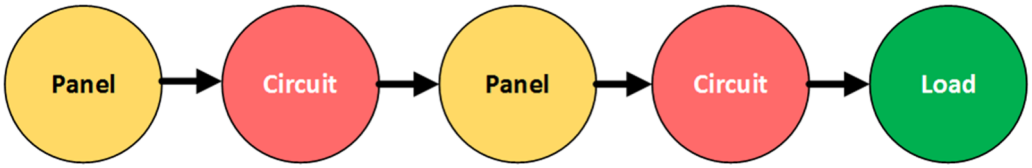


Example GraphQL Schema:

```
type HeatingCoolingSystem {
    Name: String
    IsPrimary: boolean
    HeatedElements: [AbstractElement] @relation(name:"ENERGY",direction:OUT)
    CooledElements: [AbstractElement] @relation(name:"ENERGY",direction:IN)
    EnergySources: [AbstractElement] @relation(name:"ENERGY",direction:IN)
}

type Boiler implements AbstractElement {
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  HeatingSystem: [HeatingCoolingSystem] @relation(name:"ENERGY",direction:OUT)
}

type Space {
    HeatsSpace: [HeatingCoolingSystem] @relation(name:"ENERGY",direction:IN)
    CoolsSpace: [HeatingCoolingSystem] @relation(name:"ENERGY",direction:OUT)
}
```

Electrical

Electrical elements are arranged so that each element connects to a circuit which in turn connects to another element, and ultimately to a load.

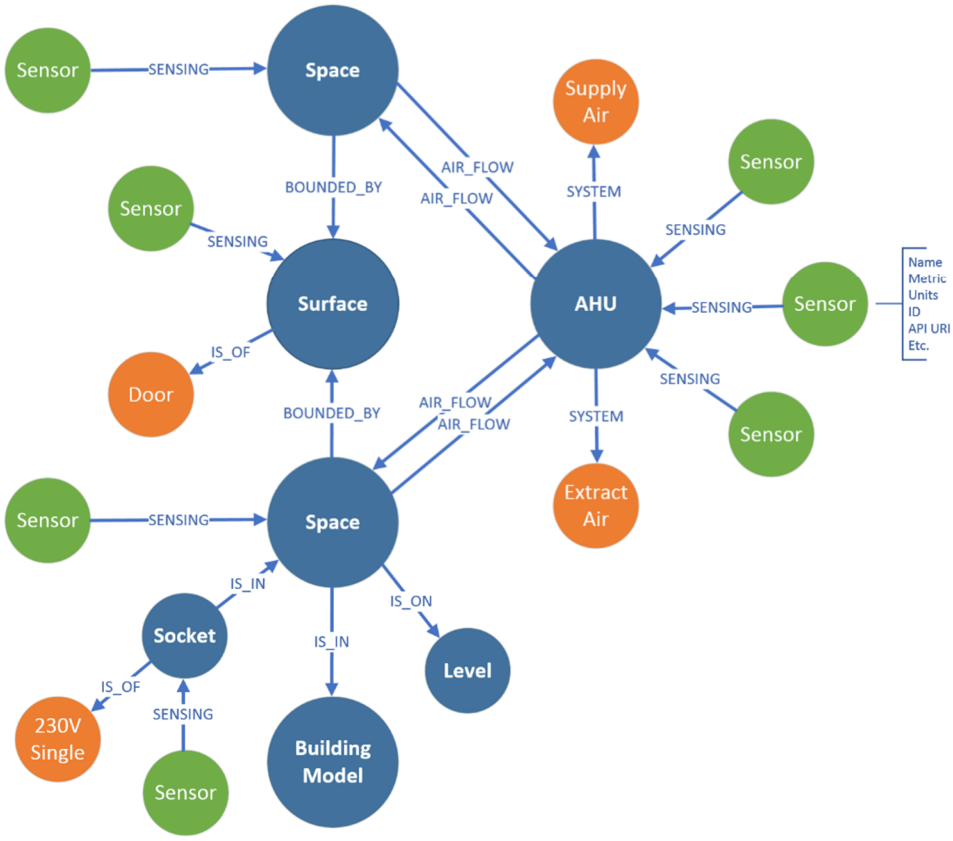


Example GraphQL Schema:

```
type DBPanel implements AbstractElement {
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  Space: Space @relation(name:"IS_IN_SPACE",direction:OUT)
  Level: Level @relation(name:"IS_ON",direction:OUT)
  OutgoingCircuits: [Circuit] @relation(name:"ELECTRICAL_FLOW_TO",direction:OUT)
  IncommingCircuit: Circuit @relation(name:"ELECTRICAL_FLOW_TO",direction:IN)
}

type Circuit implements AbstractElement {
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  Panel: DBPanel @relation(name:"ELECTRICAL_FLOW_TO",direction:IN)
  Loads: [ElectricalLoad] @relation(name:"ELECTRICAL_FLOW_TO",direction:OUT)
}

type ElectricalLoad implements AbstractElement {
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  DBID: String @cypher(statement:"MATCH (this) RETURN id(this)")
  Space: Space @relation(name:"IS_IN_SPACE",direction:OUT)
  Level: Level @relation(name:"IS_ON",direction:OUT)
  Circuit: Circuit @relation(name:"ELECTRICAL_FLOW_TO",direction:IN)
}
```

IoT

IoT Sensors and controllers are related to the spaces or devices which they sense or control.

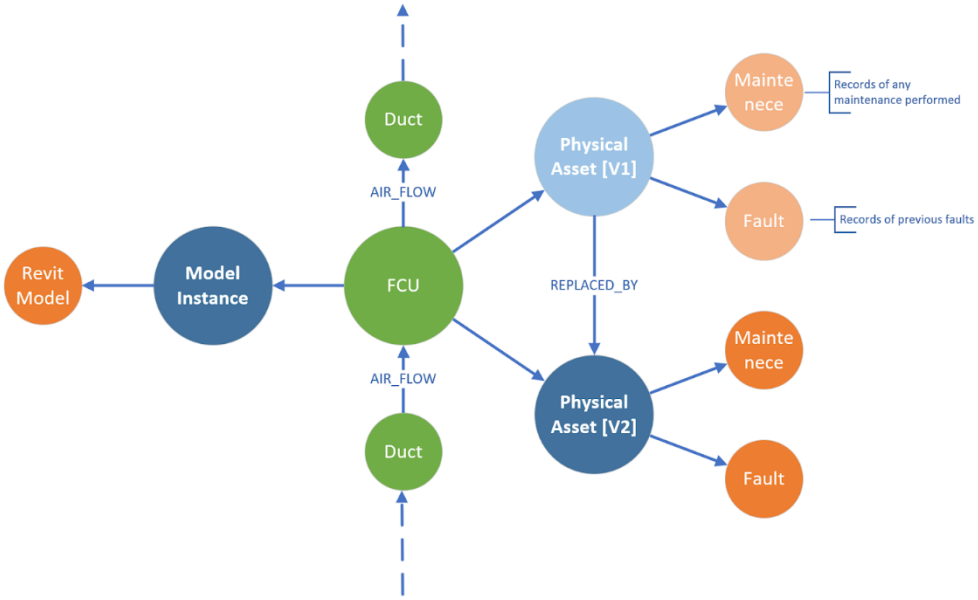


Example GraphQL Schema:

```
type Sensor implements AbstractElement {
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  Space: Space @relation(name:"IS_IN_SPACE",direction:OUT)
}

type Control implements AbstractElement {
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
  Space: Space @relation(name:"IS_IN_SPACE",direction:OUT)
}

type Space {
  SensorElements: [Sensor] @relation(name:"IS_IN_SPACE",direction:IN)
  ControlElements: [Control] @relation(name:"IS_IN_SPACE",direction:IN)
  Controls: Control @relation(name:"CONTROLS",direction:OUT)
}
```

Facilities Management

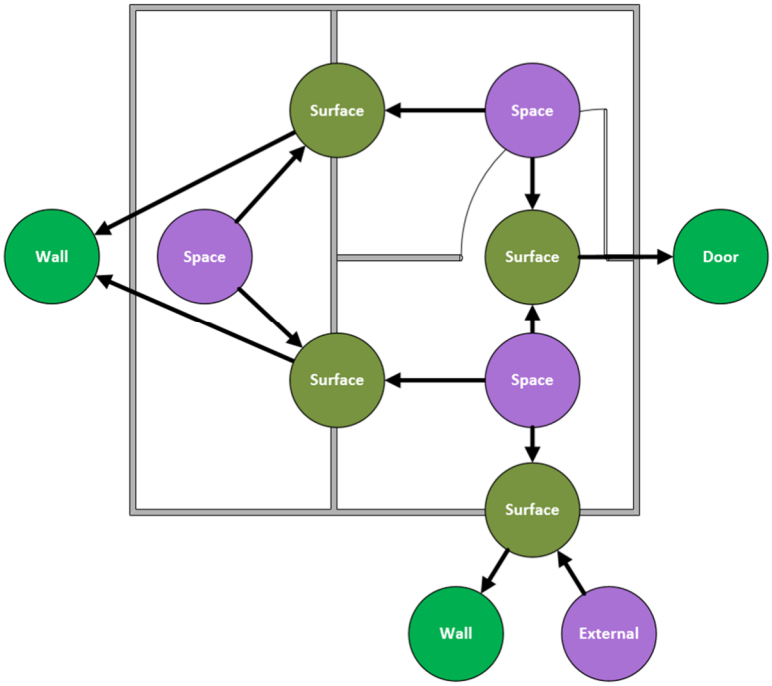
Information on faults and maintenance events can potentially be stored in the graph.




```
type Asset {
  Name: String
  Id: ID!
  Replaces: [Asset] @relation(name:"REPLACES",direction:OUT)
  FaultEvents: [FaultEvent] @relation(name:"EVENT",direction:OUT)
  MaintenanceEvents: [MaintenanceEvent] @relation(name:"EVENT",direction:OUT)
  Space: Space @relation(name:"IS_IN_SPACE",direction:OUT)
}

type MaintenanceEvent {
  Name: String
  Id: ID!
  Description: String
  Asset: [Asset] @relation(name:"EVENT",direction:IN)
}

type FaultEvent {
  Name: String
  Id: ID!
  FaultLevel: FaultLevel
  IsResolved: Boolean
  Asset: [Asset] @relation(name:"EVENT",direction:IN)
}

enum FaultLevel {
    Information
    Warning
    Critical
}
```

Geometric Features

Geometric features expose data about geometry which has been pre-computed. For example the facing directions and areas of all surfaces bounding a space. The purpose is to allow algorithms to run analysis or calculations on these features without having to continually analyze the geometry. It also presents a normalised why with which to convey geometric features, be it from the graph data or live from parametrically controlled geometry in native applications.

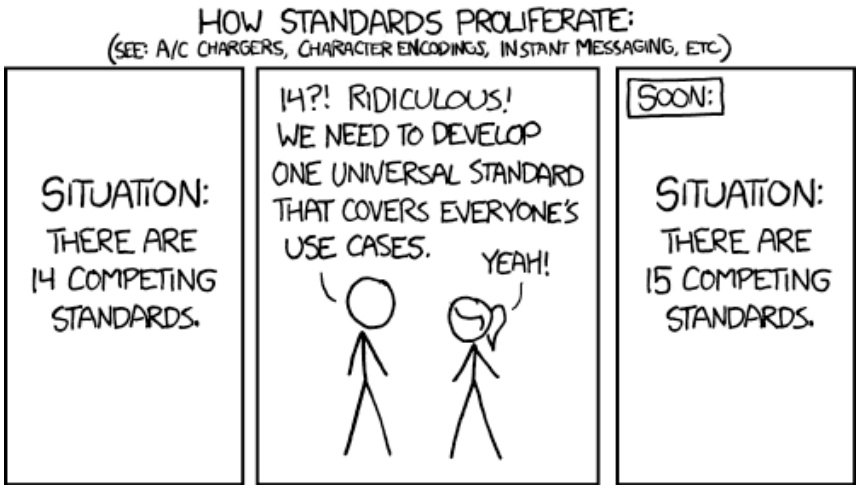


The intention at this stage is not to describe the precise geometry but rather the existence and relationship of these elements and extracted feature data. The abstract geometry is referenced to real elements and can be rendered by their associated application.

Example GraphQL Schema:

```
type Section implements AbstractElement {
  Name: String
  Id: ID!
  ElementType: ElementType @relation(name:"IS_OF",direction:OUT)
  ModelElements: [ModelElement] @relation(name:"REALIZED_BY",direction:OUT)
}

type SurfaceBoundary @relation(name: "BOUNDED_BY") {
  from: Space
  to: Surface
  SectionFacingVectorX: Float
  SectionFacingVectorY: Float
  SectionFacingVectorZ: Float
  SectionArea: Float
}

type Surface {
  Id: ID!
  Name: String
  Spaces: [Space] @relation(name:"BOUNDED_BY",direction:IN)
  Host: AbstractElement @relation(name:"IS_ON",direction:IN)
}
```

But wait… we already have IFC, gbXML, and more, isn't this just another standard?



https://xkcd.com/license.html

Well… yes, it is in a way, but there's a key difference; it's defining the communication schema, and not the necessarily the object classifications themselves. In fact, there's no reason why specific standards can't be incorporated as part of the building graph schema.

The graph can contain abstract and non-building elements before a 3D model is developed. It's also a lot more flexible and puts the power in the hands of the developer augment the schema and to build distributed applications based on the data and their application requirements.

Inevitable though, the types, fields and mutations do need to be standardized for this to work as a common data model across separate organizations. You may be approaching this from one or more of these angels

- As a third-party vendor looking to use this concept internally, and/or potentially sell integrations with customer's own data.
- As MEP engineer looking to use this concept internally, but not necessary linking to third party services or exposing internally developed schema or any services built around it.
- As a developer looking to integrate this concept into already established projects, either open source or closed source.
- As a developer wanting to build on this concept and contribute (awesome! 😊)

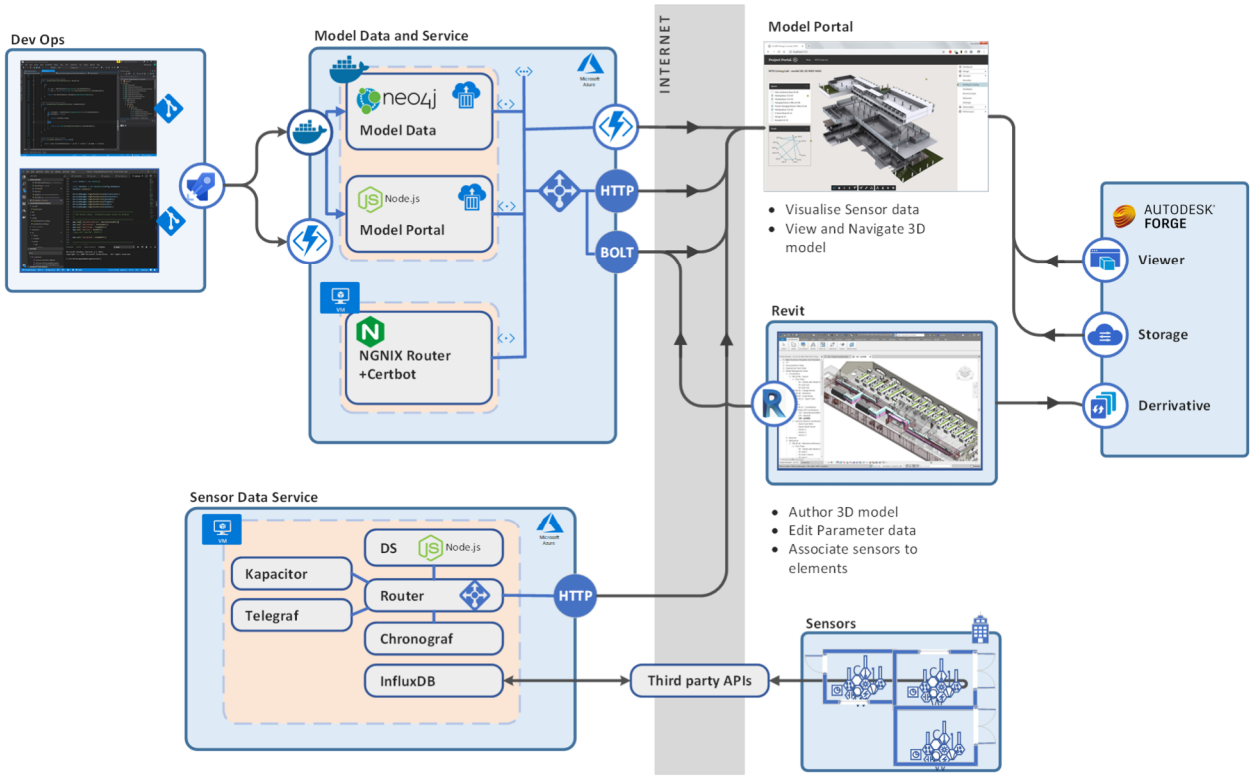
The potential of this concept is applicable to open as well as closed projects.

Some examples of the building graph in use

The following videos demo a proof of concept application targeted at the building occupancy stage of the model data lifecycle. A GraphQL query is used to find information about all the sensors available for each space as the user walks through a model in the Autodesk Forge viewer. The second video is an example integration with NodeRED which uses the same GraphQL endpoint.

https://youtu.be/hsbBHZTMWtM
https://youtu.be/VXadWvZcSZ0

Application architecture:



This particular app uses a hybrid approach, using both GraphQL and bolt. This was developed pre-GRANDstack by using the GraphQL plugins for Neo4j instead, but essentially is the same. Azure functions are used as a proxy for the GraphQL queries.

Bringing to all together: GRANDstack

GRANDstack: GraphQL, React, Apollo and Neo4j Database
https://grandstack.io/

GRANDstack is a Node.js based platform on which to develop applications utilizing GraphQL and Neo4j. It vastly simplifies the process of building applications by removing much of the requirement to create all CRUD operations and other boilerplate code manually.

The @cypher and @relation directives seen in the graphql schemas are specific to Apollo and Neo4j. These provide the glue cypher statements which form an essential part of directing how the data is translated to Neo4j.

A snippet of the index.js for the building graph server:

```
import { typeDefs } from "./graphql-schema"; ← combines all .graphql files to a single schema

const schema = makeAugmentedSchema({typeDefs, resolvers}); ← creates all the CRUD and other
boiler plate functions

/*
 * Create a Neo4j driver instance to connect to the database
 */
const driver = neo4j.driver(
  process.env.NEO4J_URI || "bolt://localhost:7687",
  neo4j.auth.basic(
    process.env.NEO4J_USER || "neo4j",
    process.env.NEO4J_PASSWORD || "neo4j"
  )
);

/*
 * Create a new ApolloServer instance, serving the GraphQL schema
 */
const server = new ApolloServer({
  context: { driver },
  schema: schema,
  introspection: true});

// Specify port and path for GraphQL endpoint
const port = process.env.GRAPHQL_LISTEN_PORT || 4001;
const path = "/graphql";

/*
* Optionally, apply Express middleware for authentication, etc
* This also also allows us to specify a path for the GraphQL endpoint
*/
server.applyMiddleware({app, path});

app.listen({port, path}, () => {
  console.log(`GraphQL server ready at http://localhost:${port}${path}`);
});
```

Building your own service and integrating with your own apps

https://github.com/willhl/BuildingGraph-Server

Check the above repo first as this could be subject to change, but generally, this will be the procedure:

Prerequisites:

- GIT
- Docker, including Docker Compose

To get started, use these commands in your favorite terminal:

```
$ git clone https://github.com/willhl/BuildingGraph-Server.git
$ cd BuildingDataModelGraphQL
$ docker-compose up
```

This will:

- Download all the code and files required
- Compile the building graph server GRANDstack Node.js API service
- Bring up a building graph server container instance
- Bring up a Neo4j Database container instance. Essential directories will be mounted outside of the container

Now what?

- Push a Revit model to the graph,
  - https://github.com/willhl/BuildingGraph-Client-Revit

- Try out some GraphQL queries in GraphiQL:
  - https://github.com/graphql/graphiql

- Try out some example integrations:
  - https://github.com/willhl/BuildingGraph-Client-Examples

- Customize and develop you own schema to suit your needs

Caveats and things it doesn't do (yet)

Multiple projects

As of Neo4j 3.5, the recommended way to deal with multiple projects is to create a Neo4j database, and optionally building graph API server per a project. This raises some infrastructure issues around routing and administration. Each project would need to be set up manually and port routing set with a service such as NGNIX.

Neo4j 4 will potentially solve this issue, as well as present many new opportunities, by allowing sperate databases to be defined within the same server instance or cluster.

Security, authentication and authorisation

As it stands, there is no authentication with the API and as such is currently only suited where each end can establish a trusted connection, this could be via an internal network, by VPN, or other external authentication service such as those offered by cloud providers.

This is on the todo list, at the simplest level, an authentication middleware service such as Passport.js can be integrated within the GRANDstack implementation.

Authorization is another big item on the todo list. It is possible to define this directly within Neo4j, particularly with Neo4j 4 it will allow fine grained access control to read/write data at the node level.

Backup

With Neo4j community the supported backup method requires the database to be stopped. This can raise some issues with Neo4j docker containers if the data directory is not mounted outside the container. This is not an issue with Neo4j Enterprise which supports live backup of data.

Conclusion

Hopefully this document has presented enough of a case that you're as enthused by the idea as the person who wrote it. At least, hopefully you'll take away the virtues of GraphQL and building applications with the GRANDstack.

Further learning resources about the technology used:

https://graphql.org/learn/

https://grandstack.io/

https://www.apollographql.com/

https://neo4j.com/

Github repositories for the BuildingGraph:

https://github.com/willhl/BuildingGraph-Server

https://github.com/willhl/BuildingGraph-Client-Revit

https://github.com/willhl/BuildingGraph-Client-Examples