OGC 3D-IoT Platform for Smart Cities Engineering Report

Publication Date: 2020-07-29

Approval Date: 2020-03-10

Submission Date: 2020-03-02

Reference number of this document: OGC 19-073r1

Reference URL for this document: http://www.opengis.net/doc/PER/3D-IoT-Platform

Category: OGC Public Engineering Report

Editor: Volker Coors

Title: OGC 3D-IoT Platform for Smart Cities Engineering Report

---

OGC Public Engineering Report

Table of Contents

1. Subject . . . 6
  1.1. Executive Summary . . . 6
  1.2. Document contributor contact points . . . 7
  1.3. Foreword . . . 7
2. References . . . 8
3. Terms and definitions . . . 9
  3.1. Abbreviated terms . . . 9
4. Overview . . . 10
  4.1. Background . . . 10
  4.2. Overview the ER . . . 11
5. Concept of 3D IoT Smart City Platform . . . 12
  5.1. Goal . . . 12
  5.2. Conceptual Model . . . 12
    5.2.1. Model with a time space . . . 13
    5.2.2. Sensor view . . . 13
    5.2.3. Feature view . . . 13
    5.2.4. Relation to the SensorThings . . . 14
    5.2.5. Summary . . . 16
6. Architecture of the 3D IoT Smart City Platform . . . 18
  6.1. Use Cases and Activity diagrams . . . 18
    6.1.1. Use Case 1: air quality . . . 18
    6.1.2. Use Case 2: IndoorGML and Occupancy . . . 25
  6.2. Overall Architecture . . . 29
    6.2.1. Overview GeoPortal Cyient . . . 29
    6.2.2. Overview GeoPortal Gaia3D . . . 30
    6.2.3. Overview WPS Helyx . . . 32
    6.2.4. Overview SensorThings API SensorUp . . . 36
    6.2.5. Air Quality Simulator . . . 37
    6.2.6. Overview Augmented Reality GeoPortal Skymantics . . . 42
    6.2.7. Overview GeoPortal STT . . . 43
7. Implementation of the 3D IoT Smart City Platform . . . 47
  7.1. Components / Services used in both air quality and indoor occupancy use case . . . 47
    7.1.1. Preparing 3D Models in CityGML and IndoorGML . . . 47
    7.1.2. WFS model server by GISFCU . . . 51
  7.2. Outdoor 3D City Model and outdoor air quality sensors . . . 57
    7.2.1. STT GeoPortal . . . 57
    7.2.2. Web Feature Service to provide 3D building models in glTF . . . 60
    7.2.3. SensorThings API for air quality measurement . . . 62

7.2.4. Helyx Implementation regarding out door air quality sensors . . . 66
7.3. Indoor Building Model and indoor sensors . . . 68
7.3.1. Gaia3D Implementation of the GeoPortal(only occupancy reading) . . . 68
7.4. Overview and Architecture by Cyient . . . 73
7.4.1. Cyient 3D GeoPortal . . . 74
7.4.2. Air Quality Dashboard . . . 75
7.5. Skymantics 3D GeoPortal . . . 77
7.5.1. IndoorGML . . . 78
7.5.2. CityGML (3D Tiles) . . . 79
7.5.3. SensorThings API . . . 80
7.5.4. Recommendations for Future Work . . . 81
8. Summary . . . 83
8.1. Recommendations and change requests . . . 83
8.1.1. Features of Interest and observed/inferred properties in the built environment . . . 83
8.1.2. Derived observations and virtual sensors O&M and the SensorThings API . . . 84
8.1.3. Smart building / district design implications of 3D-IoT work . . . 85
Appendix A: Open API Specification of Property Estimator WPS for synthetic air quality Sensor Measurements . . . 86
Appendix B: Revision History . . . 96
Appendix C: Bibliography . . . 97

Chapter 1. Subject

Recent years have seen a significant increase in the use of three-dimensional (3D) data in the Internet of Things (IoT). The goal of the 3D IoT Platform for Smart Cities Pilot was to advance the use of open standards for integrating environmental, building, and IoT data in Smart Cities. Under this initiative a proof of concept (PoC) has been conducted to better understand the capabilities to be supported by a 3D IoT Smart City Platform under the following standards: CityGML, IndoorGML, SensorThings API, 3D Portrayal Service, and 3D Tiles.

1.1. Executive Summary

An OGC Pilot was carried out to bring together city models (indoor or outdoor) and sensor observations so that observable properties could be defined and visualized for both individual and aggregate components of buildings and larger city units. Dynamic sensor observations were provided for measuring Particulate Matter (PM) - the mixture of solid particles and liquid droplets in the air. The observations measured both $PM_{2.5}$ air quality (fine particulate matter of a mass per cubic meter of air of particles with a size generally less than 2.5 micrometres) and building room occupancy. The $PM_{2.5}$ air quality was measured and synthesized for outdoor point locations and building room occupancy was synthesized for single room locations. Observations were made available through services that implemented the OGC SensorThings API standard. Features represented using the OGC IndoorGML building model and 3D-Tiles / glTF city model features were provisioned through implementations of the OGC API – Features – Part 1:Core standard. IndoorGML is a profile of the OGC Geography Markup Language (GML) standard. Various clients deployed in the pilot fetched both features and observations keyed to them by a GML identifier (GML ID), written as ‘gmlid’ in markup, so as to allow users to interact with city feature rendered according to their dynamically observed properties. Clients included stand-alone dashboard applications, applications based on 3DPS (3D Portrayal Service), and AR (Augmented Reality) visualization tools.

Web Processing Service (WPS) and OGC API - Processes implementations were deployed to aggregate and/or interpolate observations from multiple locations in order to estimate the observable properties of larger city features such as building floor interiors and the air masses above city streets or blocks. These derived observations could either be visualized directly in client applications or posted to implementations of the SensorThings API as new data.

The pilot demonstrated a viable standards-based distributed architecture for connecting dynamic sensor observations with modeled city features. It also demonstrated the importance of aligning feature / sensor identifiers and other infrastructure data in order to sustain robust Smart City 3D-IoT capabilities.

Smart cities are communities where information technology and data are used to address social, economic, and environmental challenges. Smart Cities solutions are both popular for improving city livability, and necessary for responding to trends such as climate change and increasing urbanization.

The pilot recommends the following future work:

- There is a need to look into optimization of real-time access to geospatial data for indoor maps and navigation. Development of an indoor viewer plugin that can stream data from a server.

- Vuforia SDK is only specific to Microsoft Windows and Apple MacOS. Exploring ARCore and ARkit to handle markerless Augmented Reality (similar to the Pokemon Go game) would be worth investigating as they run on Linux.
- Future development should include a mobile application that retrieves IndoorGML from an OGC API - Features server to display indoor maps in Augmented Reality.
- There is a need to consider improvements for serving representations of buildings, based on zoom level, via an implementation of the draft OGC API – Tiles specification.

1.2. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Contacts

Name	Organization	Role
Charles Chen	Skymantics	Contributor
ChenYu Hao	GISFCU	Contributor
Hyemi Jeong	Gaia3D	Contributor
Josh Lieberman	OGC	Contributor
Ki-Joune Li	Pusan National University	Contributor
Logan Stark	Skymantics	Contributor
Ravi Nishesh	Cyient	Contributor
Steve Liang	SensorUp	Contributor
Theo Braun	Helyx	Contributor
Thunyathep Santhanavanich	HFT Stuttgart, Steinbeis	Contributor
Volker Coors	HFT Stuttgart, Steinbeis	Editor

1.3. Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 2. References

The following normative documents are referenced in this document.

- OGC: OGC 18-053r2, OGC® 3D Tiles Specification, 2019 [https://www.opengeospatial.org/standards/3DTiles]
- OGC: OGC 15-001r4, OGC® 3D Portrayal Service 1.0, 2017 [https://www.opengeospatial.org/standards/3dp]
- OGC: OGC 12-019, OGC® City Geography Markup Language (CityGML) Encoding Standard, 2012 [https://www.opengeospatial.org/standards/citygml]
- Khronos Group, glTF 2.0 Specification [https://www.khronos.org/gltf/]
- OGC: OGC 14-005r5, OGC® IndoorGML, 2018 [https://www.opengeospatial.org/standards/indoorgml]
- OGC: OGC 15-078r6, OGC® SensorThings API Part 1: Sensing, 2016 [https://www.opengeospatial.org/standards/sensorthings]
- OGC: OGC 17-079r1, OGC® SensorThings API Part 2: Tasking Core, 2019 [https://www.opengeospatial.org/standards/sensorthings]
- OGC: OGC 09-025r2, OGC® Web Feature Service 2.0 Interface Standard, 2014 [https://www.opengeospatial.org/standards/wfs]
- OGC: OGC 17-069r3, OGC API - Features - Part 1: Core, 2019 [https://www.ogc.org/standards/ogcapi-features]
- OGC: OGC 14-065, OGC® WPS 2.0 Interface Standard, 2018 [https://www.opengeospatial.org/standards/wps]
  - OGC: OGC 08-126, The OpenGIS® Abstract Specification Topic 5: Features, 2009
  - OASIS: MQTT Version 3.1.1 Plus Errata 01, 2015

Chapter 3. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2] shall apply. In addition, the following terms and definitions apply.

- **feature**

  A feature is an abstraction of real world phenomena (source: OGC 08-126)

- **WFS model server**

  A server component that delivers a representation of a feature such as a building. The interface of the model server is OGC API - Features in this Pilot.

- **particulate matter**

  The mixture of solid particles and liquid droplets in the air (source: UK DEFRA)

- **Sensor**

  An entity capable of observing a phenomenon and returning an observed value. Type of observation procedure that provides the estimated value of an observed property at its output. [OGC 12-000]

Further terms and definitions used in the Pilot are specified in detail the Conceptual Model in section 5 .

3.1. Abbreviated terms

- FoI Feature of Interest
- GL Graphics Library
- glTF GL Transmission Format
- GML Geography Markup Language
- I3S Indexed 3D scenes
- IDW Inverse distance weighting
- IoT Internet of Things
- JSON JavaScript Object Notation
- LOD Level of Detail
- STAPI SensorThings API
- UML Unified Modeling Language
- WFS Web Feature Service
- WPS Web Processing Service

Chapter 4. Overview

4.1. Background

Smart cities are communities where information technology and data are used to address social, economic, and environmental challenges. Smart City solutions are both popular for improving city livability, and necessary for responding to trends such as climate change and increasing urbanization.

Sejong City, founded in 2007, is the new administrative city of South Korea. The Sejong 5-1 District is the site of a wide-ranging Smart City Initiative, led by the Korea Land and Housing (LH) Corporation. Projects under this initiative include the following:

- AR/VR Service (Smart City Experience Zone)
- Smart Street
- Smart Park
- Smart Facilities

A video of the Sejong City Smart City Initiative is available on YouTube: https://www.youtube.com/watch?v=beSEhFawY_I&feature=youtu.be.

Figure 1. visionary smart city

The Korea Land and Housing (LH) Corporation sponsored this OGC Pilot that complements the work being advanced in the Sejong Smart City Initiative.

4.2. Overview the ER

This ER is organized as follows:

Section 5 introduces the concept of a 3D IoT Smart City Platform and its main components, 1) outdoor 3D City Model (CityGML), Indoor 3D model (IndoorGML), Geo-IoT (SensorThings API). It describes the core concept of the integration of implementations of the SensorThings API and 3D Building model, representing both Indoor and Outdoor environments.

Section 6 analyses the use cases in more detail and develops a software architecture to implement a 3D IoT Smart City Platform. It provides recommendations on preferred strategies.

The concept of the 3D IoT Smart City Platform described in section 6 has been implemented by the participants of this Pilot. The result is not one unified system but the implementation of the two main use cases on outdoor air quality and indoor occupancy. Some components have been used in both use cases, others are specific for one use case only. The Technology Integration Experiment (TIE) tables give a good overview of the components and how they are used in the different prototypes.

The WFS model server, with an OGC API - Features interface, is used in both use cases as well as in data preparation as documented in section 7.1.

Section 7.2 documents the implementation of the first use case on the Outdoor 3D City Model and outdoor air quality sensors, esp. real-time monitoring on micro-dust.

Section 7.3 focuses on the Indoor Building Model and indoor sensors such as real-time monitoring on indoor occupancy.

Section 8 provides a summary of the main findings and discusses links to other tasks such as deployments of implementations of OGC API - Features, WPS, and 3DPS.

Appendix A provides the API definition of the WPS to get access to the spatial interpolation of the air quality measurements as developed in this pilot

Chapter 5. Concept of 3D IoT Smart City Platform

5.1. Goal

The goal of the pilot was to advance the use of open standards for integrating environmental, building, and internet of things (IoT) data in Smart Cities. Under this initiative a proof of concept (PoC) was conducted to better understand the capabilities to be supported by a 3D IoT Smart City Platform under the following standards:

- Fully textured building models represented in CityGML LOD (Level of Detail) 2
- IndoorGML Model of Building interior
- An implementation of OGC API – Features deployed and configured to deliver 3D Building Models including Building interior (referred to as WFS model server in this engineering report)
- 3D Portrayal Service to stream 3D content for Web-based visualization using glTF and the OGC community standard 3D-Tiles
- SensorThings API to serve the sensor data
- WPS to provide a spatial distribution model of measured data

The scenarios selected to demonstrate this concept include:

- Real-time monitoring on indoor occupancy of a building
- Real-time monitoring on micro-dust concentration in an urban district

The test area of this Pilot was Sejong City. The district Sejong 5-1 is the site of a wide-ranging Smart City Initiative, led by the Korea Land and Housing (LH) Corporation. A CityGML model of the district has been made available. However, the CityGML model was to be processed on a server hosted in South Korea. For the 3D GeoPortal, the CityGML model was provided as a glTF model via an implementation of OGC API - Features. In addition, a 3D-Tiles data set was made available as a secondary data store derived from the CityGML model. One building in the district was made available as an IndoorGML model. In this pilot, outdoor air quality measurements had been connected with the 3D building model to visualize the air quality of the district in real time. From a technical point of view, it required a way to integrate the measurements and the 3D building model either on the client or on the server. The measured data was provided via a SensorThings API. In this pilot, both measured and synthetic observations have been used for test purposes. In the second use case, the occupancies of rooms and floors of a building interior have been connected with an IndoorGML model. The occupancy data was provided again via a SensorThings API. In this pilot, only synthetic data of occupancy was used.

5.2. Conceptual Model

One key question of this pilot was how to integrate observations such as air quality measurements with 3D models of the built environment. This is a fundamental concept in Smart Cities, as almost all use cases require the integration of environmental and internet of things (IoT) data with City

Objects such as buildings. The concept used here is explained in detail to ensure that there is a common understanding of the different elements as well as the used terms.

The objects in the real-world can be defined as a set of phenomenon space A — $\{a_1, a_2, \ldots, a_n\}$ and a set B — $\{b_1, b_2, \ldots, b_m\}$ which can be derived through observation from the phenomenon space A.

- f: $A \rightarrow B$
- $b = f(a)$, a □ A, b □ B

An example is the case of the sensor systems are installed in one building observing the number of people and Carbon Dioxide (CO_2) concentration in room 1. In this case, the phenomenon space is a room, and A is a set of rooms. A function f_0: $A \rightarrow B_0$ maps the number of people to a room. B_0 is the set of natural numbers □. A second function f_1: A → □ maps the CO_2 concentration of a room in ppm.

- A — Set of rooms $(a_1, a_2, \ldots, a_n)$
- f_0 — The number of people [people]
- f_0: A → □
- f_1 — The CO_2 concentration [ppm]
- f_1: A → □
- $b^0{}_1 = f_0(a_1)$ — The number of the people in room 1
- $b^1{}_1 = f_1(a_1)$ — The CO_2 concentration in room 1

5.2.1. Model with a time space

To include time depended measurements into the model, the functions are depending on the time space which can be described as follows:

- f: A x T → B
- b = f (a,t) example
- $b^{m}{}_{n,tp} = f_{m}(a_{n,tp})$ — the property `m` in area `n` at time `p`

5.2.2. Sensor view

In the view of sensors in the real-world, they are observing the properties(f) of phenomenon space (A) at a time(t). For example, the sensors observing `the number of people` and `CO₂ concentration` in room `1` at time `p` can be described as follows:

- f: A x T → B
- $b^0{}_{1,tp} = f_0(a_1,t_p)$ — the number of the people in room a_1 at time t_p
- $b^1{}_{1,t0} = f_1(a_1,t_p)$ — the CO_2 concentration in room a_1 at time t_p

5.2.3. Feature view

A feature is an abstraction of real world phenomena (OGC 08-126). Within the context of this pilot, a feature or object can further be considered to be an element of the phenomenon space which is described by several properties. These properties can be either time dependent or time independent.

- F: $(A, f_0(a,t), f_1(a,t),\ldots, f_{n-1}(a,t), g_0(a),\ldots g_{m-1}(a))$
- f — time-dependent properties
- g — time-independent properties
- n — number of time-dependent properties
- m — number of time-independent properties

5.2.4. Relation to the SensorThings

In SensorThings, there are two links to features in the world. Thing entities refer to those physical objects or systems in the real-world directly occupied and monitored by sensors (corresponding to Platform entities in the Observations and Measurements formalism). SensorThings observations are also connected to FeatureOfInterest (FoI) entities that have one or more properties or attributes whose values can be estimated as a result of a sensor or IoT device observation. These attributes are listed and described in the *ObservedProperties* entity. The sensors and IoT devices are listed and described in the *Sensors* entity. The distinction between Thing and FeatureOfInterest is critical for remote sensing situations, however, for the in-situ sensors utilized in this Pilot, it is not necessary to distinguish Things and FeatureOfInterest entities. For Pilot purposes, information about the real-world entity being observed was largely maintained in the Thing entity.

The SensorThings API provides an open and unified way to interconnect IoT devices over the Web as well as interfaces to interact with and analyze their observations. The foundations of the SensorThings API are the relational connections between entities in the system and the way they are used to model systems in the real world. The entities have a natural relationship which enables any IoT sensing device from any vertical industry to be modelled in the system. An IoT device or system is modeled as a *Thing*. A *Thing* has a *Location* with one or more *Datastreams*. Each *Datastream* observes one *ObservedProperty* with one *Sensor* and has many *Observations* from the *Sensor*. Each *Observation* read by the *Sensor* estimates the value of an observed / observable property for one particular *FeatureOfInterest*. Together, these relationships provide a flexible standard way to describe and model any sensing system (OGC 15-078r6).

According to the conceptual model, as we want to integrate the building data and IoT data together, the *Thing* entity (or equally FoI) refers to a set of objects or physical systems (A — {a1, a2, …, a_n}) which are systems of buildings or building parts such as room, wall, roof etc. Each *Thing* contains a link referring to their associated building objects in the CityGML models containing all the building information. The time-independent results ($g_{0..n}(a_{0..m})$) can be stored as properties of the Thing / FoI while the time-dependent results ($f_{0..n}(a_{0..m},t_{0..p})$) are stored in the Datastream entity collecting the raw time-series sensor data. For example, a CO_2 sensor system installed in room A measuring the CO_2 concentration can be modeled in a SensorThings UML diagram as in Figure 2.

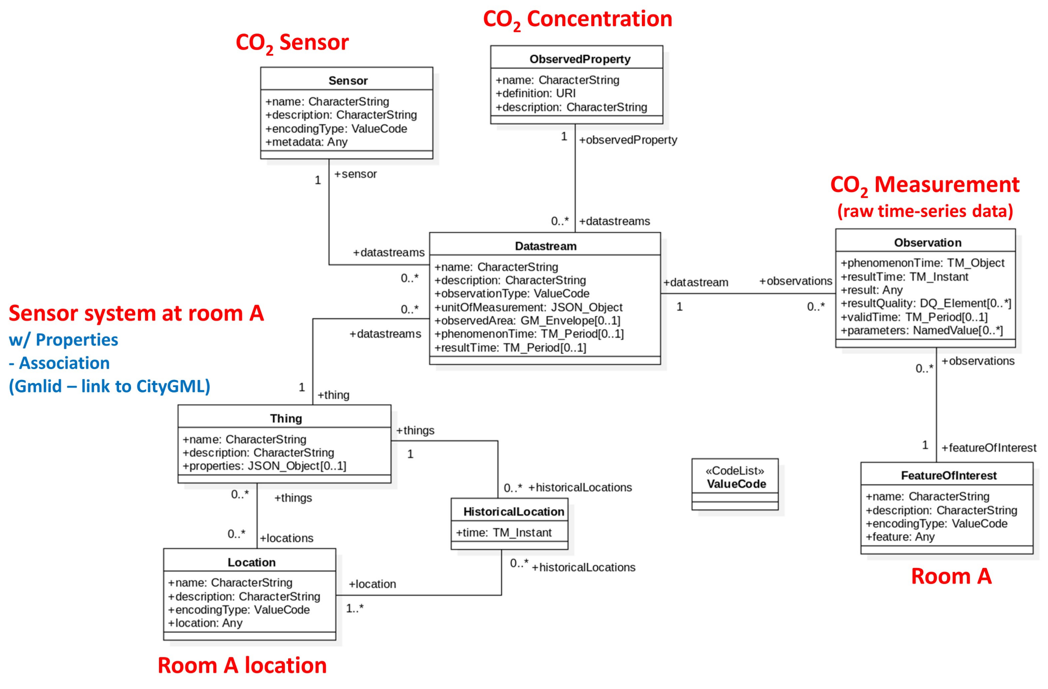


Figure 2. Example modeled SensorThings UML

If the sensor *Observations* data is post-processed by a process tool resulting in a new set of computed *Observations*, then the SensorThings entities have to be updated as follows. The *Sensors* resource is updated to describe the process tool. The *ObservedProperty* resource can either remain the same or be updated. The *Thing* and *FeatureOfInterest* resource are updated to collect the process tool information with association links to the 3D city model and the original sensor system. For example, if the WPS tool is used to compute the air quality index (AQI) or CO_2 concentration of floor B based on data from all rooms in floor B. This system can be modeled in a SensorThings UML diagram as in Figure 3.

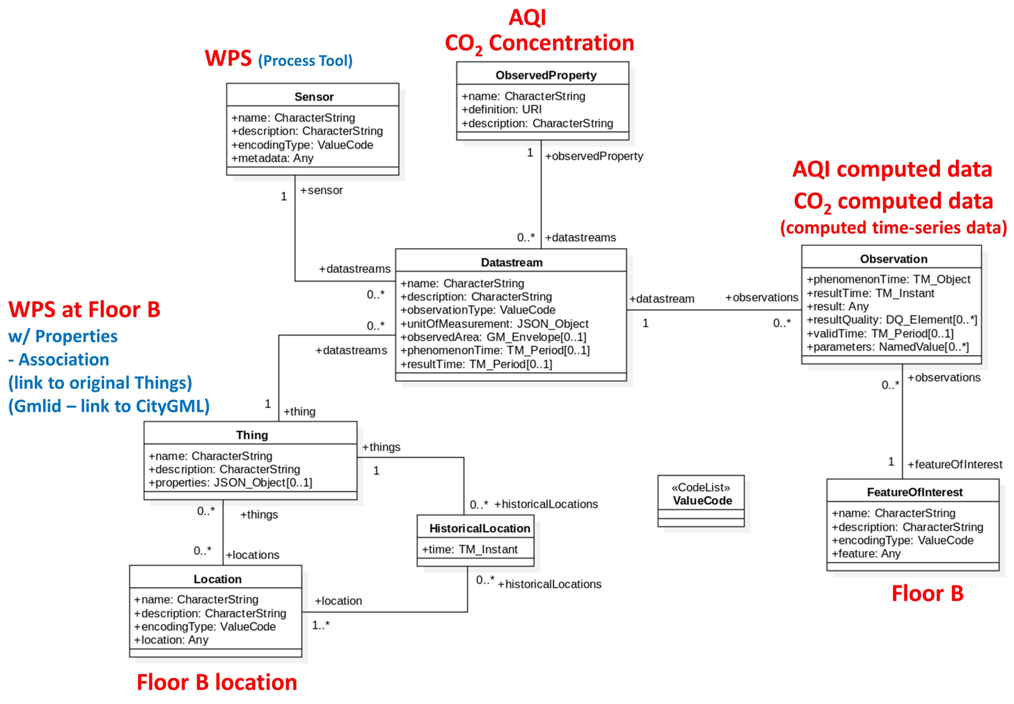


Figure 3. Example modeled SensorThings UML with post-processed data

5.2.4.1. Deriving Higher-Level Observations

SensorThings API also enables the continuous processing of *Observations* from multiple *Datastreams* to generate a higher-level *Observation*. Conceptually it can be described as follows:

- B_0 — Set of observations from the phenomenon space A_0
- B_1 — Set of observations from the phenomenon space A_1
- B_2 — Set of higher-level observations describing the phenomenon space A_2, and derived from other observations
- f: $B_0 \times B_1 \times T \rightarrow B_2$
- $b_2 = f(b_0, b_1, t), b_2$ □ B_2

For example, `dew point temperature` can be calculated with observed `air temperature` and `relative humidity`.

- B_0 — Set of air temperature observations
- B_1 — Set of relative humidity observations
- B_2 — Set of dew point temperature observations
- $b_2 = f(b_0, b_1, t)$ — dew point temperature can be approximated by using the following formula: `dew point temperature` = `air temperature` - ((100 - `relative humidity`)/5)

In many cases, a higher-level *Observation* is derived from one set of *Observation*, i.e., from the same *Datastream*. For example, the `speed` of a moving feature can be derived from the moving feature's `location_observation`$_1$ in `t`$_1$ and `location_observation`$_2$ in `t`$_2$.

In practice, higher-level *Observations* can be derived through the chaining of multiple SensorThings API implementations through the publish/subscribe-based MQTT protocol. The following diagram Figure 4 describes the concept.

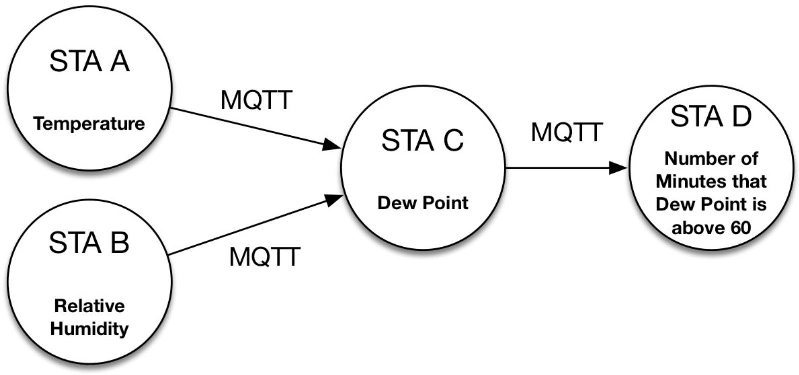


Figure 4. Higher-level observations derived from multiple SensorThings Datastreams

5.2.5. Summary

In the pilot, the measured data was provided as raw data in both use cases without chaining of multiple SensorThings API services. In the case of air quality measurement, the data was either visualized as a diagram in the 3D GeoPortal as measured, or an interpolation was calculated using all measurements in a given area by means of a WPS. One example of chaining SensorThings API implementations could be the aggregation of air quality measurements over time, e.g. to provide an hourly average of the air quality measurement. In the case of the occupancy sensor, the data is not based on real observations, but synthetic ones by an algorithm. In a real-world application, sensing the number of people in a given area typically requires chaining implementations of the SensorThings API that process several sensors such as camera systems and interpretation of the imagery. Since the focus of the pilot was on the integration and validation of the interfaces, a simplified synthetic occupancy sensor approach was chosen.

In both Pilot use cases, given the in-situ nature of the sensors, *FeatureOfInterest*, and the *Things* could be considered to refer to the same physical entity. One way to link the *Thing* and *Observation* with the feature and its properties would be by way of the CityGML Dynamizer extension that will be part of CityGML 3.0. It extends the *CityObject* in CityGML and introduces links to the relevant SensorThings API. An alternative approach adds the unique identifier of the *CityObject* to the *Thing* in the SensorThings API and maps the *Observation* to the attributes of the *CityObject*. This approach does not require an extension of CityGML. As the CityGML model was not accessible outside South Korea, this extension of the CityGML was more or less not doable. The integration of SensorThings API and the 3D City model has been implemented by linking the *Thing* to the *CityObject* via a unique identifier. It is important to notice that this approach requires a stable identifier in case of an update of the model.

Chapter 6. Architecture of the 3D IoT Smart City Platform

The software architecture adopted to implement a 3D IoT Smart City Platform is described in this chapter.

6.1. Use Cases and Activity diagrams

6.1.1. Use Case 1: air quality

The aim of this use case is the 3D visualization of a 3D city model together with outdoor air quality measurements. It includes two stakeholders:

- the facility manager who has an interest in monitoring the outside air quality, and
- the administrator of the 3D Geo-Portal

Other stakeholders such as citizens are represented by the facility manager use case in this 3D IoT pilot. The facility manager is also responsible for the maintenance of the sensors. However, in this pilot it is assumed that the sensors do not need maintenance.

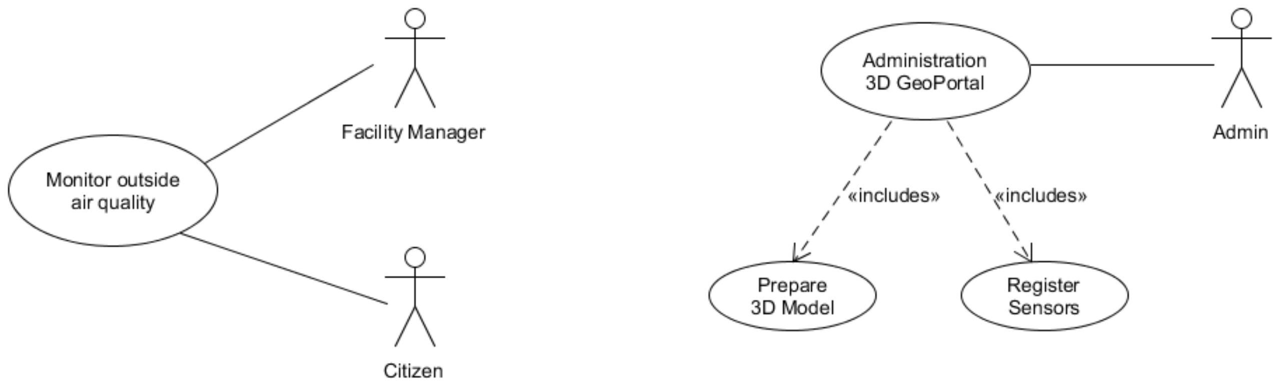


Figure 5. Use Case diagram

6.1.1.1. Use Case 1-1: Monitor the outside air quality using the 3D Geo-Portal

The facility manager uses a web-based system to monitor the current air quality. The air quality is measured by a number of sensors distributed in the Sejong area, for example in the lamp posts along the main streets. The measurements are visualized in real time in a 3D environment together with a 3D city model to get a better understanding of the air quality in along the streets and the built environment. In the pilot, only fine dust $PM_{2.5}$ is taken into account as an indicator of air quality for reasons of simplicity. As air quality is a continuous phenomenon, the sensor measurement is interpolated in real time to get a distribution of the air quality in space. The air quality distribution is simplified by using an Inverse Distance Weighted (IDW) interpolation. The facility manager can get access to the current and historic measurements of every single sensor. The historic measurements are then shown as a line chart diagram in the 3D viewer.

6.1.1.2. Use Case 1-2: Administration of 3D Geo-Portal

The administrator of the web-based service - the 3D GeoPortal - has to prepare the 3D model for web-based 3D visualization. In addition, the sensors have to be registered and integrated into the 3D model to ensure that the sensor readings are visualized in the correct 3D location.

6.1.1.3. Activity Diagrams Use Case 1-1

6.1.1.3.1. Visualizing a 3D scene with real time outdoor air quality using 3D Portrayal Service

The activity diagram in Figure 6 shows a general approach to get a 3D scene including a 3D building model and air quality data based on sensor measurements. The client selects a region of interest defined by a rectangular area. The 3D building model is retrieved via a 3D Portrayal Service for the specified area. The data delivery format can be specified in the request. In general, for larger 3D scenes, the data delivery format has to support streaming of a tiled data set such as those conforming to the OGC community standards Indexed 3D scenes (I3S) or 3D-Tiles. In this 3D IoT pilot, 3D-Tiles was used in all implementations. In this Activity Diagram, the terrain model of the 3D globe is used. No additional terrain data is provided. The client may need to project the 3D building model to the terrain model of the globe just to avoid a visual mismatch of the building geometry, and the terrain model that might appear due to different source and resolution of the building model and terrain ("flying buildings").

In a second request, the air quality sensors within the specified area are selected from a SensorThings server. Each Sensor supports multiple data streams of different measures of phenomena. In this Use Case, each sensor provides one data stream on fine dust $PM_{2.5}$. Time resolution may differ from sensor to sensor.

The client visualizes the data streams of the air quality sensors. Each data stream can be visualized as a line chart to show the exact sensor readings over time. In addition, the measured data of all sensors at a given point in time can be interpolated to show the spatial distribution of the air quality. Within the scope of the 3D IoT Pilot, an IDW interpolation is sufficient for this purpose.

To summarize the flow of information:

1. select a region of interest by using a 3D Portrayal Service
2. fetch data streams of all air quality sensors within this region
3. interpolate and visualize air quality measures in real time at client

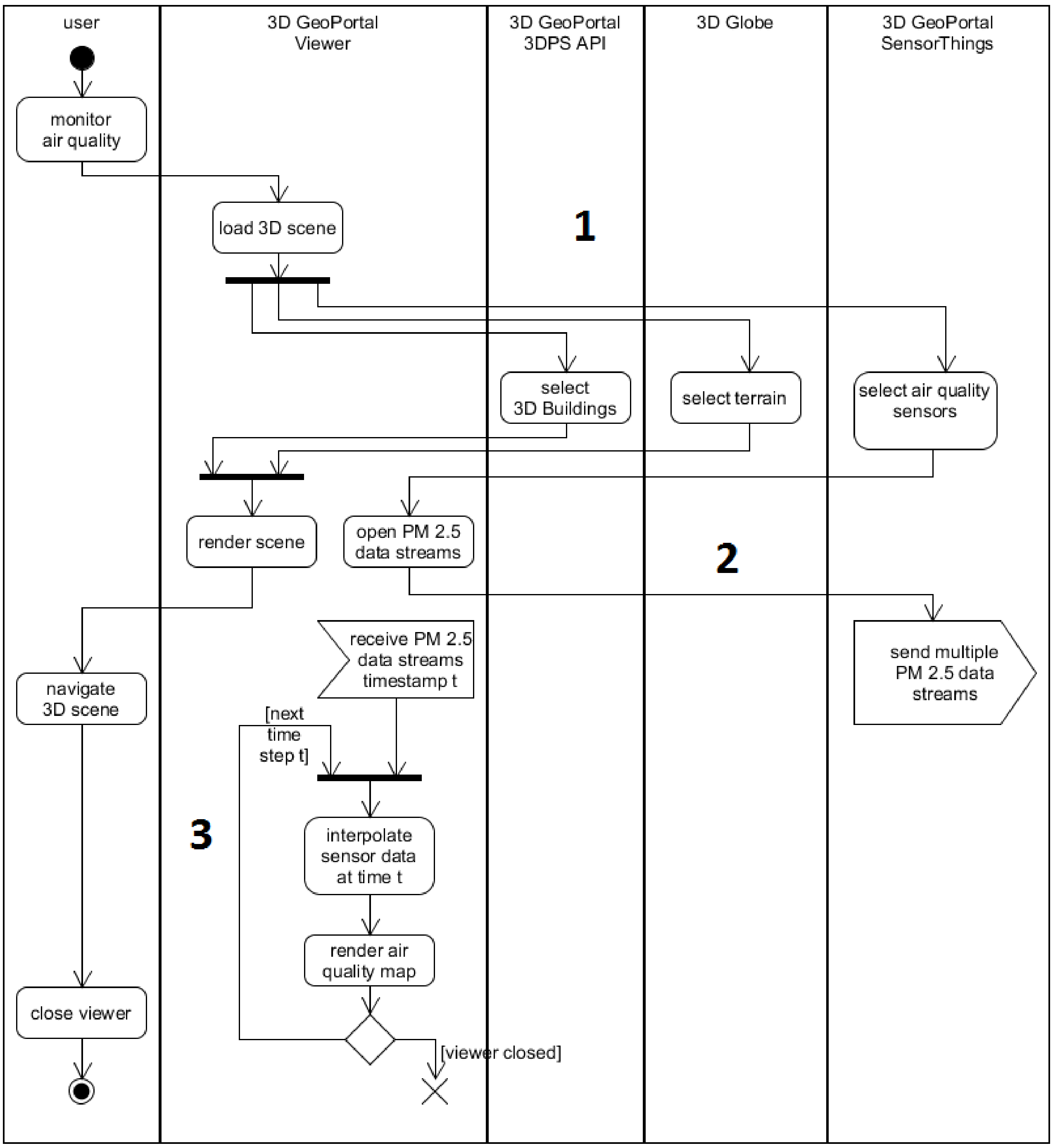


Figure 6. visualize 3D scene with real time outdoor air quality using 3D Portrayal Service

6.1.1.3.2. Visualizing 3D scene with real time outdoor air quality using predefined area

Th Activity Diagram in Figure 7 is similar to the above one. The only difference is that it loads a predefined area - Sejong - by a URL instead of using the 3D Portrayal Service API to query a region from the 3D GeoPortal. All participants worked with the same 3D model of the Sejong area and used 3D Tiles for content delivery in this pilot. In this case it is sufficient to provide a predefined 3D Tileset for web-based visualization.

The main difference is the request to fetch the 3D building geometry:

The 3D Portrayal Service retrieves the 3D Buildings of the area of Sejong (with CRS:84 bounding box coordinates of 127.238631,36.478551,127.260261,36.492008) through a request that looks like:

```
http://193.196.37.89:8092/service/v1?service=3DPS&acceptversions=1.0&request=GetScene&
layers=building&format=application/json&boundingbox=127.238631,36.478551,127.260261,36
.492008
```

The request returns the JavaScript Object Notation (JSON) key pair value with the URL to the tileset.json.

```
http://193.196.37.89:8092/Assets_3diot/sejong/tileset.json
```

This tileset can be retrieved directly if it is generated in a batch process and persistently stored on the 3D GeoPortal using the same URL: "http://193.196.37.89:8092/Assets_3diot/sejong/tileset.json"

To summarize the flow of information:

1. get predefined 3D scene (Sejong area) by URL
2. fetch data streams of all air quality sensors within this region
3. interpolate and visualize air quality measures in real time at client

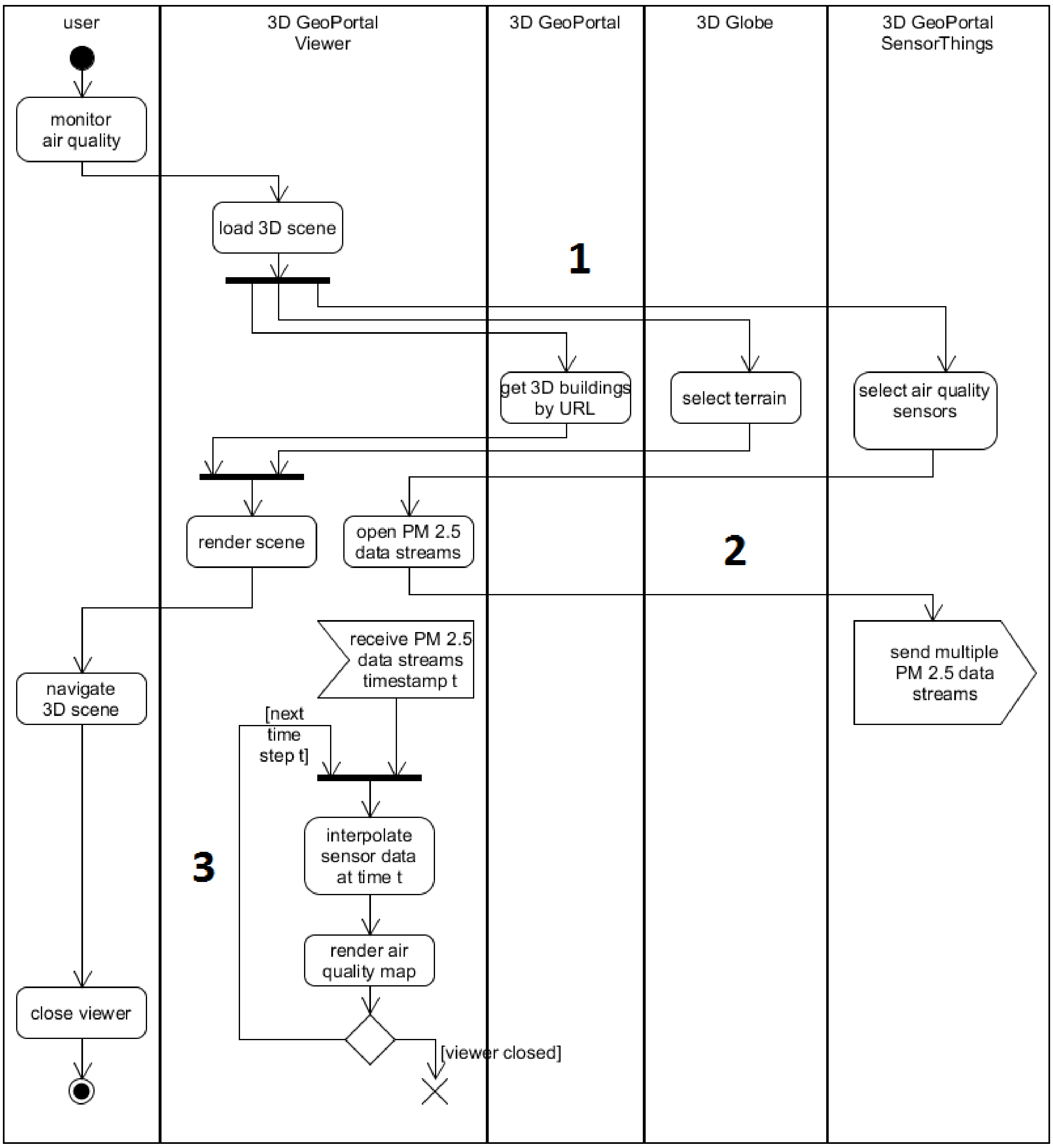


Figure 7. visualize 3D scene with real time outdoor air quality using predefined region

6.1.1.3.3. Using a web processing service to interpolate air quality data

Instead of doing the interpolation of the air quality measurements at the client, it can be done using a WPS at the back end. This will shift computational load from the client to the server and may allow more complex interpolation methods later.

To summarize the flow of information:

1. get predefined 3D scene (Sejong area) by URL
2. fetch data streams of all air quality sensors within this region
3. WPS receives air quality sensor data from Sensor Things API

4. WPS interpolates sensor data and sends a coverage / map to the client
5. client visualizes air quality "map" in real time

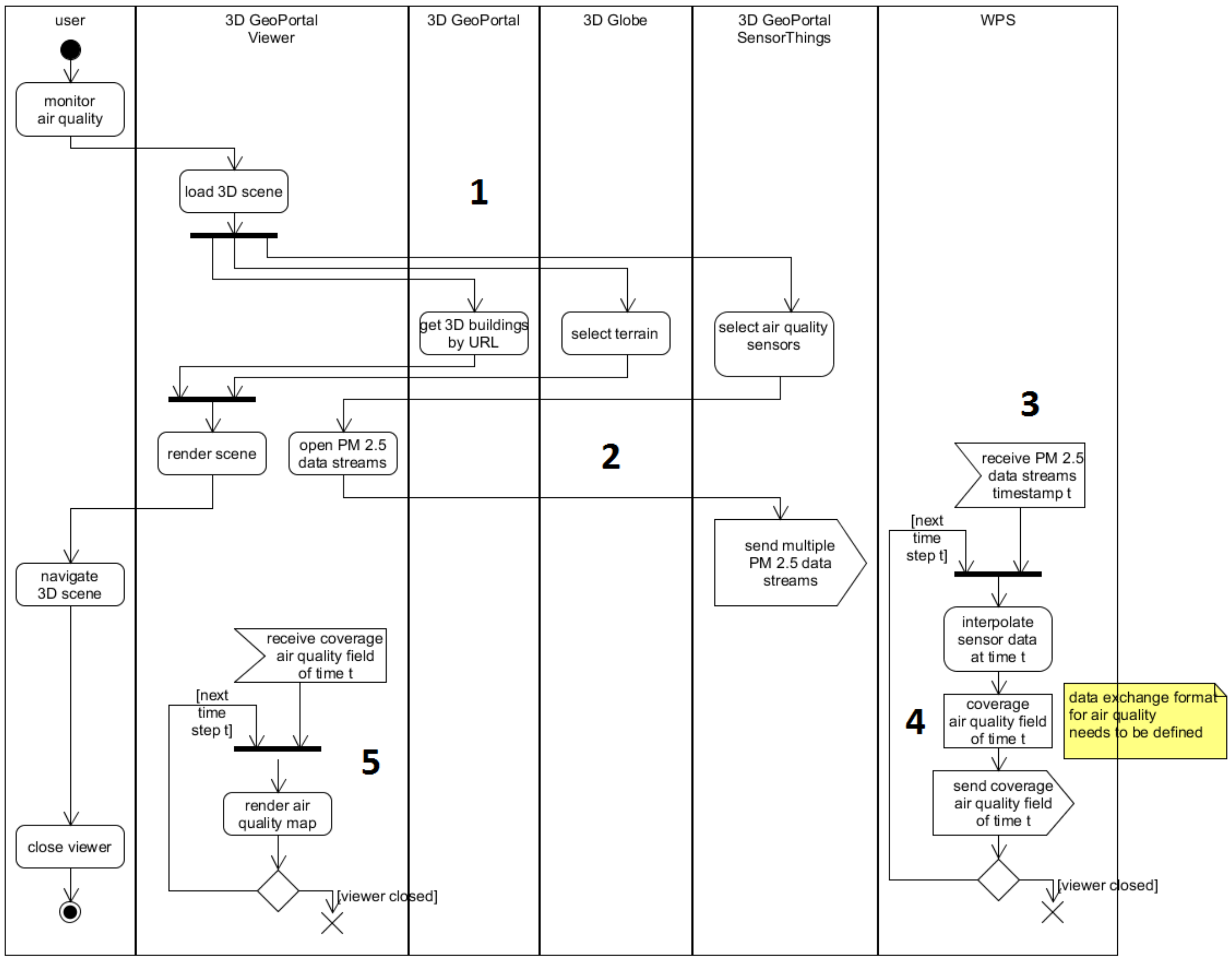

user
3D GeoPortal Viewer
3D GeoPortal
3D Globe
3D GeoPortal SensorThings
WPS
monitor air quality
load 3D scene
1
get 3D buildings by URL
select terrain
select air quality sensors
render scene
open PM 2.5 data streams
2
3
receive PM 2.5 data streams timestamp t
send multiple PM 2.5 data streams
[next time step t]
navigate 3D scene
interpolate sensor data at time t
receive coverage air quality field of time t
4
coverage air quality field of time t
data exchange format for air quality needs to be defined
[next time step t]
5
render air quality map
send coverage air quality field of time t
close viewer
[viewer closed]
[viewer closed]

Figure 8. Activity Diagram using a WPS to interpolate air quality data

6.1.1.4. Activity Diagrams Use Case 1-2

6.1.1.4.1. Registration of Air Quality Sensors

The air quality data shall be measured by sensors. However, to set up a test environment, it is helpful to generate synthetic air quality measurements by a "property estimator" process. It should be taken into account that measurements of different sensors usually do not have the same time stamp nor the same time interval. In addition, it is most likely that different sensors from different manufacturers are used in a real-world scenario. Due to the lack of standards, these sensors usually use different data formats for the sensor readings. The SensorThings API collects and stores all these different sensor datasets. The Sensor Things API enables a unified access to all the sensor readings, in this case $PM_{2.5}$ fine dust measurements. Besides the data stream, the sensors need to be registered with a fixed 3D location and a unique identifier. In this pilot, the location of a sensor is referenced to the World Geodetic System 1984 (WGS 84) datum, but with height above ground.

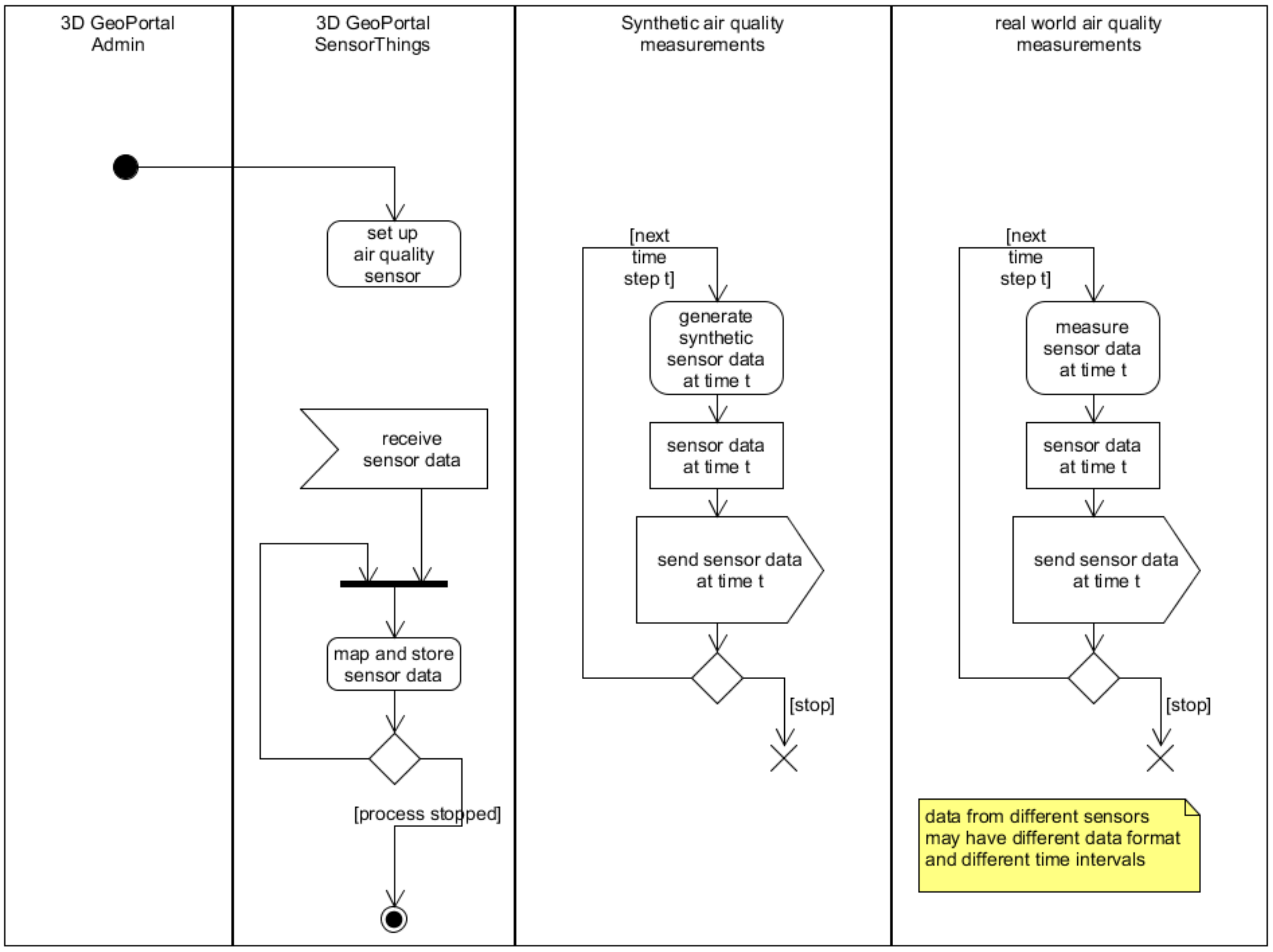

3D GeoPortal Admin
3D GeoPortal SensorThings
Synthetic air quality measurements
real world air quality measurements
set up air quality sensor
receive sensor data
map and store sensor data
[process stopped]
[next time step t]
generate synthetic sensor data at time t
sensor data at time t
send sensor data at time t
[stop]
[next time step t]
measure sensor data at time t
sensor data at time t
send sensor data at time t
[stop]
data from different sensors may have different data format and different time intervals

Figure 9. Activity Diagram using both synthetic air quality data as well as measured one

6.1.1.4.2. Preparation of the 3D model

Under the assumption that the 3D city model is available in some kind of 3D data store, the region of interest is retrieved from the data store by a region query. The resulting city model, in this case a 3D building model only, is stored as a CityGML document. This CityGML document is then converted to a tile-based format to support streaming within the format as well as a non-tiled format. In this case, the client needs to come up with a streaming strategy by itself.

In this Pilot, all implementations are based on the Cesium globe. The CityGML building models will be converted to 3D Tiles and to glTF. The entire process is straight forward, different tools can be used to convert the CityGML document to 3D Tiles and to glTF. The resulting glTF models, and tileset.json together with the hierarchical tile set will be stored at the 3D GeoPortal with a unique URI. The Activity diagram in Figure 10 shows the generation of the tiled data set. The generation of the glTF model follows the same process.

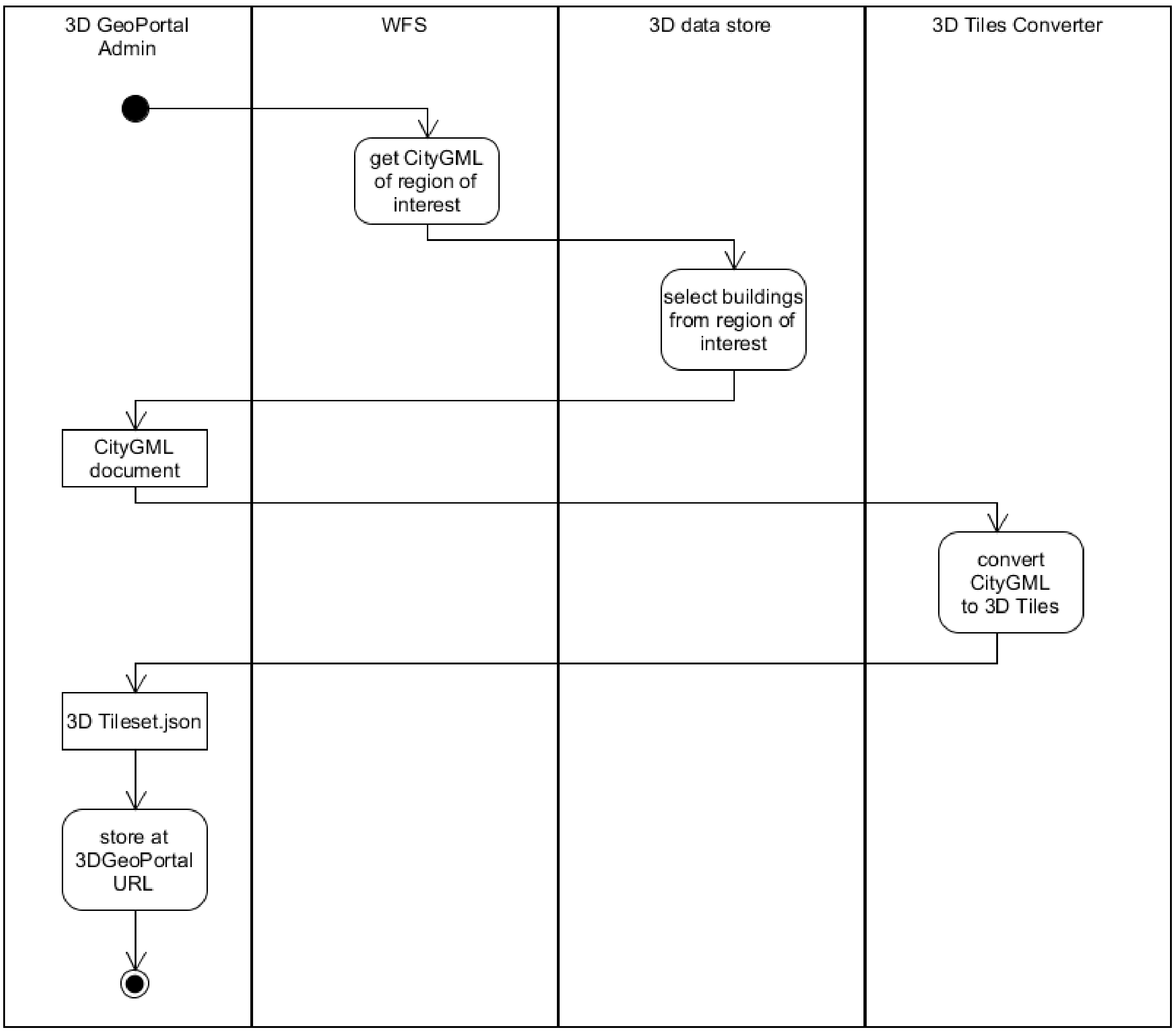


Figure 10. Activity Diagram Preparation of 3D model

6.1.2. Use Case 2: IndoorGML and Occupancy

The second use case is similar to Use Case 1, but with a focus on building interior and the observation of the occupancy of an interior space. The aim of this use case is the 3D visualization of a 3D building model including an indoor model together with occupancy observation. It includes two stakeholders:

- the facility manager who has an interest in monitoring the occupancy of an interior space such as a shop, and
- the administrator of the 3D Geo-Portal

Other stakeholders such as citizens are represented by the facility manager use case in this 3D IoT pilot. The facility manager is also responsible for the maintenance of the sensors. However, in this pilot it is assumed that the sensors do not need maintenance.

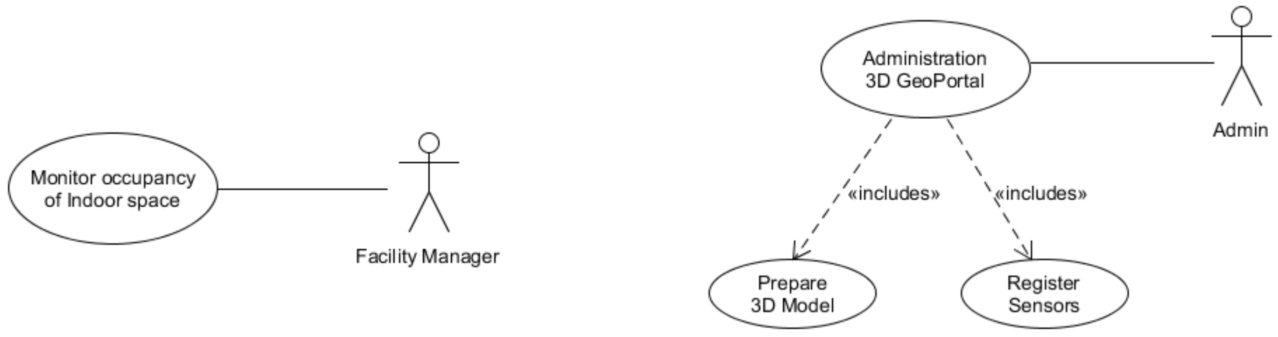


Figure 11. Use Case diagram 3D GeoPortal to Monitor Occupancy of an Indoor Space

6.1.2.1. Use Case 2-1: Monitor occupancy of an interior space

The facility manager uses a web-based system to monitor the occupancy of interior spaces. On an abstract level, this use case is similar to Use Case 1-1. However, the interactive visualization of the building interior is an additional challenge. Location based Augmented Reality shall be used in addition to a virtual reality 3D globe. Another challenge is the use of sensors to observe the occupancy of an interior space. A virtual sensor gives the number of people in a given space. However, this is the result of a cascading sensor system, for instance one sensor measures the CO_2 in ppm and a set of camera sensors observe the space. Based on an algorithm, the number of people can be derived from these measurements. In this pilot, the virtual sensor generated synthetic data. Consequently, the architecture took the cascading sensor system into account, as it is an interesting use case for the use of the SensorThings API.

6.1.2.2. Use Case 2-2: administration of 3D Geo-Portal

The administrator of the web-based service - the 3D GeoPortal - has to prepare the 3D model for web-based 3D visualization. In addition, the sensors have to be registered and integrated into the 3D model to ensure that the sensor readings are visualized in the correct 3D space. The occupancy is measured not at a 3D location, but in a space such as a room. The observed space in represented with a 3D geometry in IndoorGML. The sensor and the observed space have to be linked in this use case.

6.1.2.3. Activity Diagrams Use Case 2-1

6.1.2.3.1. Visualizing a 3D scene with real time occupancy of an interior space

The activity diagram in Figure 12 shows a general approach to get a 3D building model including building interior in IndoorGML and occupancy data based on sensor measurements. In this Activity Diagram, the terrain model of the 3D globe is used. No additional terrain data will be provided. The client may need to project the 3D building model to the terrain model of the globe just avoid a visual mismatch of the building geometry and the terrain model that might appear due to different source and resolution of building model and terrain.

In a second request, the occupancy sensors within the specified part of the building will be selected from a SensorThings server. Time resolution may differ from sensor to sensor.

The client visualizes the data streams of the occupancy sensors. One occupancy sensor counts the number of people in a room or a predefined cell of the building Each data stream can be visualized as a line chart or a color encoding to show the exact sensor readings over time.

To summarize the flow of information:

1. select the building of interest by using OGC API - Features interface of a WFS model server / IndoorGML
2. fetch data streams of all occupancy sensors within this building
3. visualization of occupancy of each room and/or cell in real time

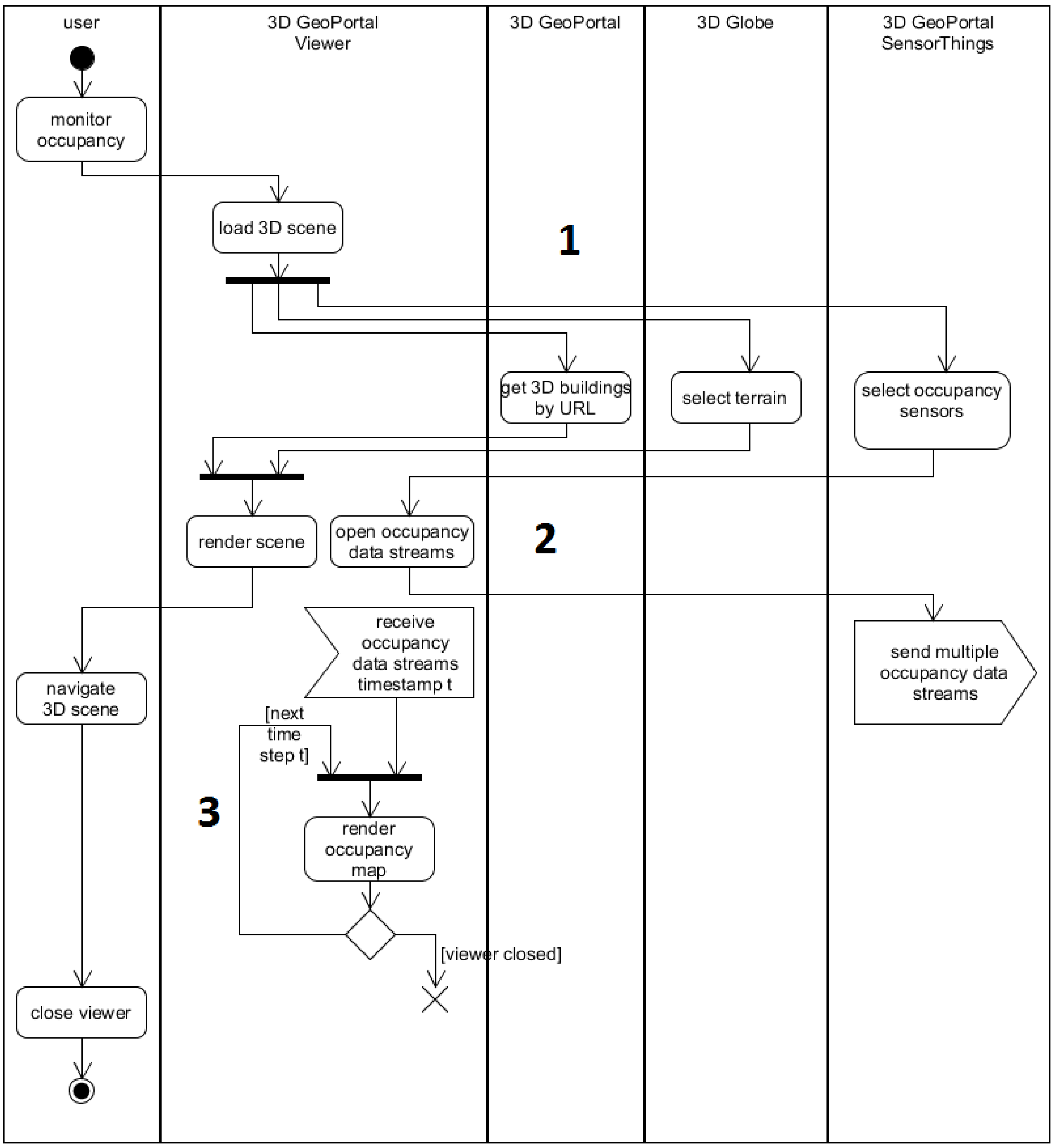


Figure 12. visualize 3D scene with real time occupancy using a predefined building model

6.1.2.4. Activity Diagrams Use Case 2-2

6.1.2.4.1. Registration of Occupancy Sensors

The occupancy of a given space - let it be a single room, or a large open space divided into a set of cells - is monitored by a virtual sensor. This virtual sensor is based on a cascading sensor system that observes the space. An algorithm calculates the resulting occupancy based on these observations. The virtual sensor can be accessed via the SensorThings API. It has to be linked to a representation of the same space in IndoorGML by the administrator of the 3D GeoPortal.

6.1.2.4.2. Preparation of the 3D model

Under the assumption that the 3D building model is available in some kind of 3D data store, the region of interest is retrieved from the data store by a region query. The resulting 3D building model is stored as an IndoorGML document. This IndoorGML document is then converted to glTF as a non-tiled compressed data format for 3D visualization.

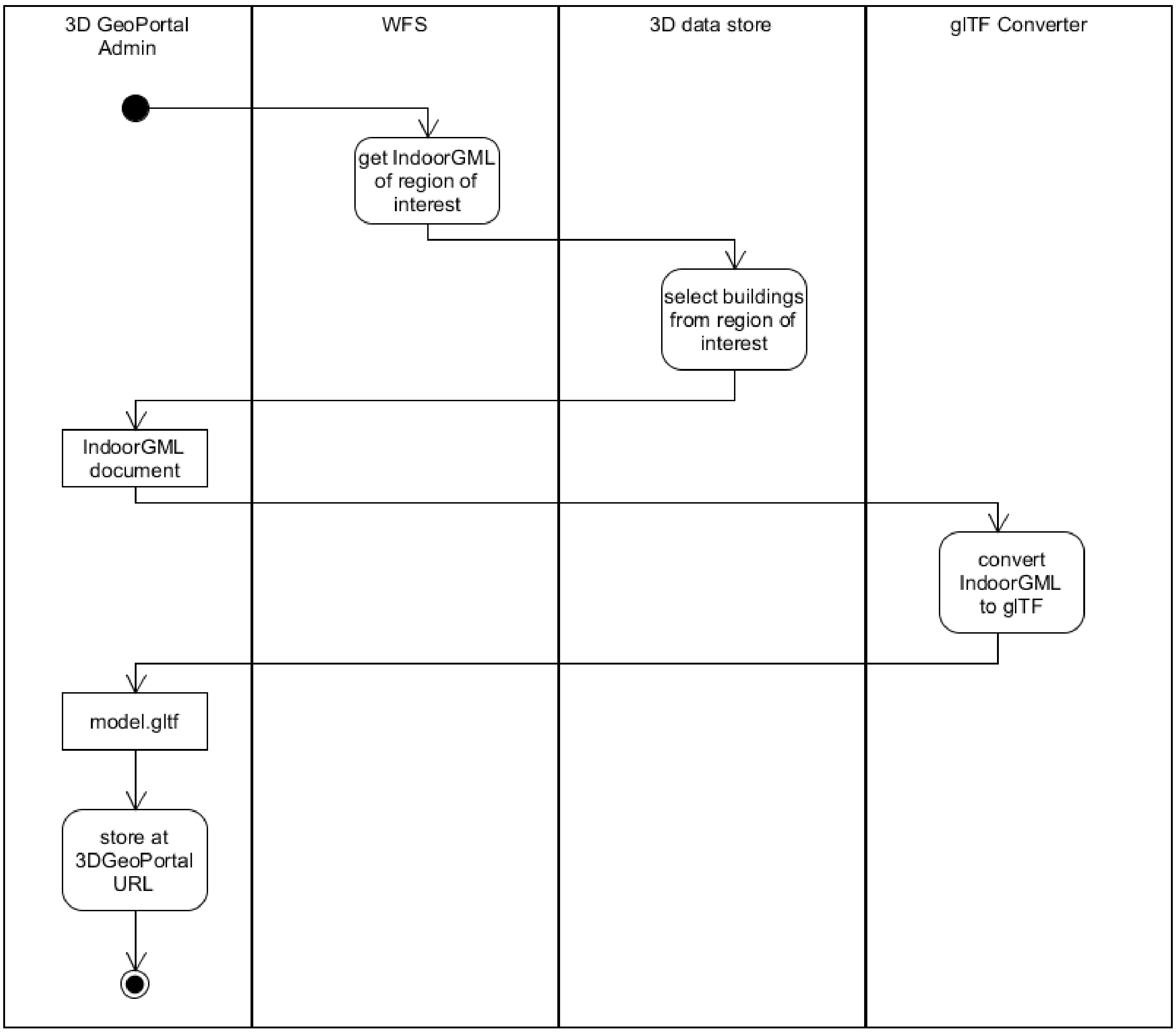

3D GeoPortal Admin
WFS
3D data store
glTF Converter
get IndoorGML of region of interest
select buildings from region of interest
IndoorGML document
convert IndoorGML to glTF
model.gltf
store at 3DGeoPortal URL

Figure 13. Activity Diagram Preparation of 3D model

6.2. Overall Architecture

Based on the Use Cases and activity diagrams, three components of the overall architecture have been identified. The Data layer includes the data sources such as the 3D City Model in CityGML and 3D Tiles, the 3D Building model in IndoorGML, and Sensor data. The data can either be stored in a database or a file-based resource. On top of the data layer, services grant access to the data. These services can be data delivery services or processing services. A very simple API just grants access to a file resource such as direct access to a 3D tiles resource. The server component can be seen as the backend of a 3D GeoPortal. The client in this pilot is a web-based client in all use cases.

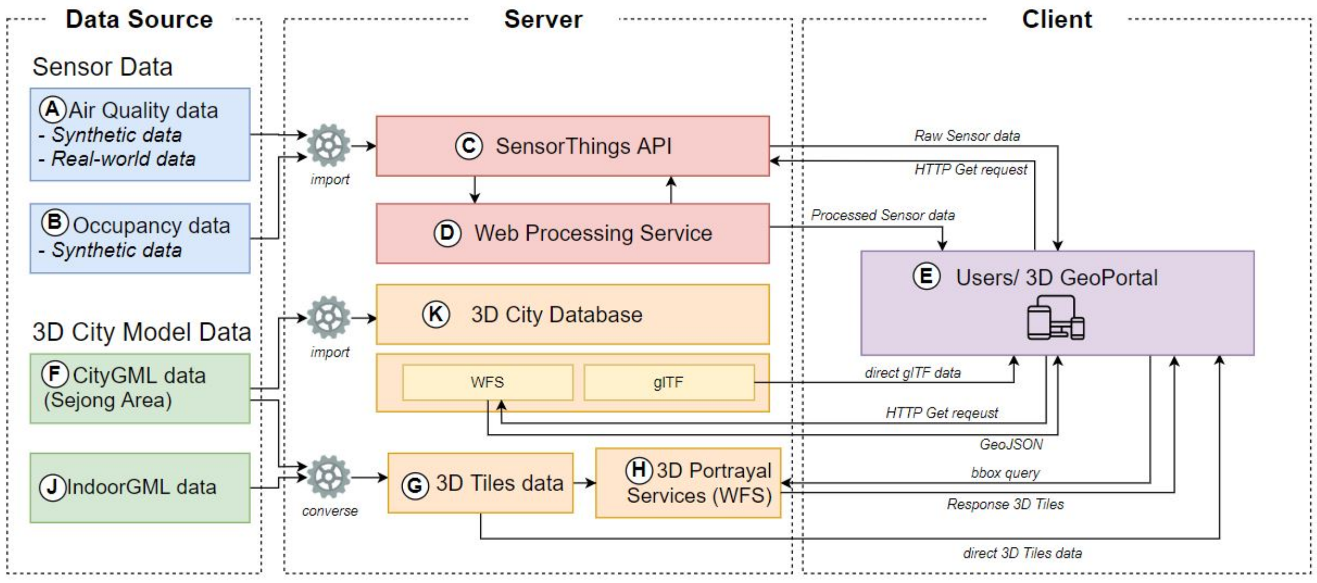


Figure 14. Overall architecture

Three alternatives have been discussed in the pilot within this architecture. In the *SensorThings service content enrichment* approach, an “Agent” invokes the Property Estimator (Web Processing Service) with sensor readings and relevant city / building objects and the results are posted back to SensorThings as derived observations on identified Thing / Features of Interest

In the second approach *City / Building model enrichment*, the “Agent” invokes the Property Estimator (WPS) with sensor readings and relevant city / building objects and the results are posted back to the WFS model server as observed (dynamic) properties of city / building objects

The third alternative *Geo-Portal enrichment* is Geo-Portal centric. The Geo-Portal fetches and visualizes both building / city objects and relevant sensor observations, and invokes the Property Estimator on this content (or references) and re-renders building / city objects according to the results

Most of the implementations follow the *Geo-Portal enrichment* approach in this pilot. In this pilot, several Geo-Portals have been implemented to test variants in the workflow as well as the relevant OGC APIs. The following sub-chapters give a short overview of the different approaches. Details on the implementation are documented in the next chapter.

6.2.1. Overview GeoPortal Cyient

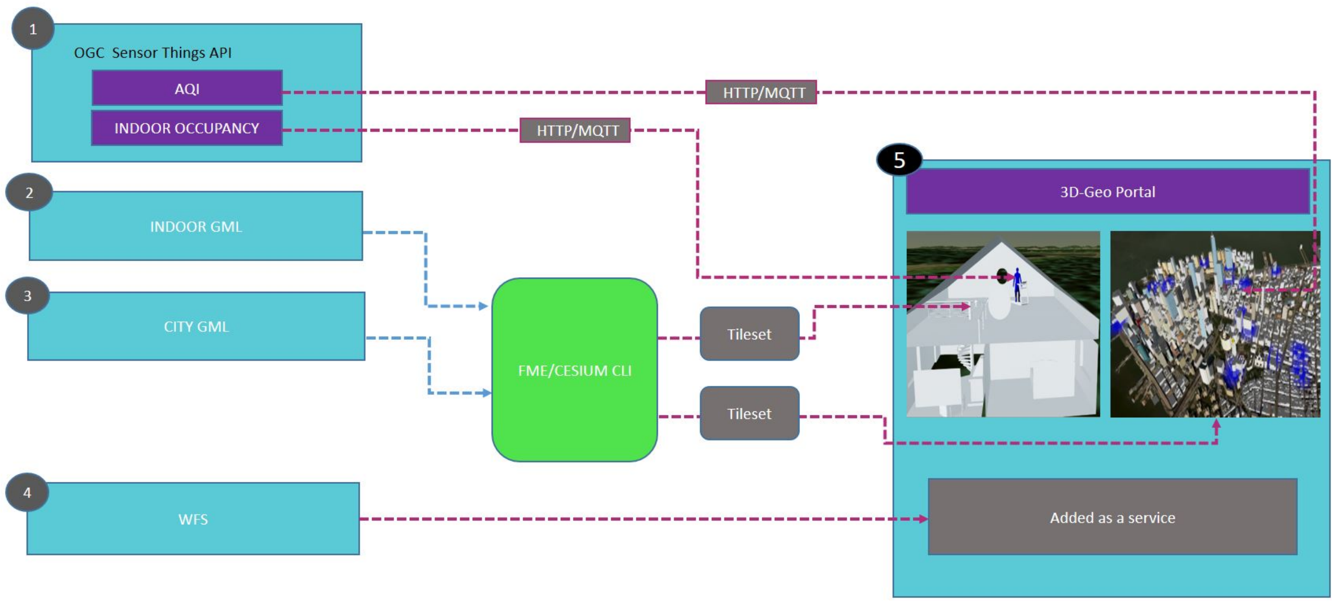


Figure 15. Workflow of things and GML

6.2.1.1. Air Quality Index and Indoor Occupancy

Air quality index data getting from the OGC SensorThings API, which include all components which will affect AQI like Carbon Monoxide (CO), Sulphur Dioxide (SO_2), Ozone (O_3) … etc, along with location details (latitude, longitude, height). In the same way occupancy data also works.

6.2.1.2. IndoorGML & CityGML

IndoorGML was converted into tileset data which is supported by FME (feature manipulation engine) or Cesium Command-line interface (CLI). In IndoorGML, all layers were separated depending upon data, like roof surface, floor surfaces, wall surface …etc. Then on and off layers also possible. CityGML was also converted into tileset data through FME, and visualized on a portal as well, with minor adjustments like terrain, offset.

6.2.1.3. 3D-GeoPortal

In the portal tilesets of IndoorGML and CityGML were visualized. On the top of tileset data serialized as CityGML AQI data, other data obtained from the SensorThings API was displayed, with devices color-coded based upon thresholds of individual parameters and AQI as well. Inside the tileset serialized as IndoorGML, occupancy sensor data was displayed. On and off control of IndoorGML layers was controlled through the portal.

6.2.2. Overview GeoPortal Gaia3D

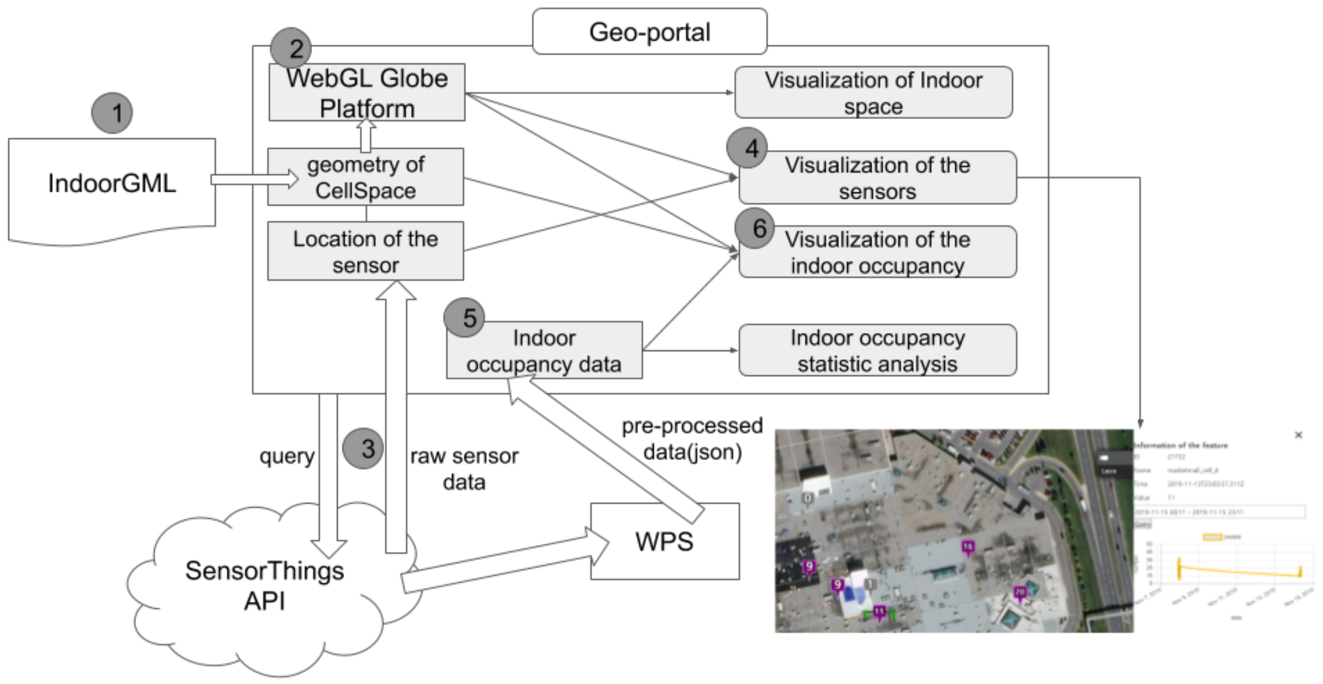


Figure 16. Workflow of GML and Sensor data, Indoor occupancy.

6.2.2.1. GML

The steps illustrated in Figure 16 are explained as follows and indicated through parenthesis. The IndoorGML model was parsed for the geometry and the attribute of the CellSpace of IndoorGML was saved at the local(1). After parsing, the indoor geometry was visualized using Gaia3D – a globe platform based on 3D WebGL (2). With parsed data, we calculate the middle point of the CellSpace geometry to use the point as the position for visualizing indoor occupancy data(5) later.

6.2.2.2. Sensor data visualization

A query was sent to the SensorThings API with the following API path (3) to get the location of the sensors, the phenomenon time of the observation and the result value of the observation to track the progress history of the sensor's value.

```
/Locations/Datastream/Observation
```

Per every 10 seconds the query was sent to refresh the data of the sensor. With the location data and the result value of sensor's observation, the sensor's position and value on the globe was visualized (4). The User can then review the history of the sensor data from the graph.

6.2.2.3. Indoor occupancy data visualization

At (1) is IndoorGML's data was parsed into two parts: the geometry and the data of CellSpace. Through GML ID of CellSpace, the Indoor occupancy data was requested from WPS server (5). Indoor occupancy data is then presented at the position that was calculated at (1). Pre-processed data in JSON was used for visualization(6) and statistical analysis. The workflow is similar to that of sensor data visualization.

6.2.2.4. 3D GeoPortal

At the 3D GeoPortal, the geometry of the IndoorGML including the network and the location of the occupancy is visualized. For intuitive sight, each room of the building is colored to represent the value of the occupancy. Visualization of the occupancy value history graph is supported on the portal.

6.2.3. Overview WPS Helyx

6.2.3.1. Overview of OGC API-Processes (WPS) Property Estimator Implementation

The position of the Property Estimator API within the architecture of the pilot is potentially a complex one. The API could be designed to support a range of implementations and use cases to supplement the SensorThings API readings.

For the purposes of this Pilot, the Property Estimator API can be considered an "aggregation model" sensor to support four identified use cases. The functionality of these four use cases ranges from standard request-response client-server interactions to purely server-to-server interactions. These are listed below from simple to complex.

- The Request-Response Client-Server process: This use case is the most traditional approach. The Client requests an aggregation from the Property Estimator API for a location or feature, which in turn fetches the relevant information from the Source SensorThings API (STAPI) services, calculates an aggregation and returns this to the client.
- The Pre-Calculated process: This use case is similar to the first, with the Property Estimator API subscribing to the Source STAPI services. However, once the data is retrieved the readings are incorporated into a more complex persisted space-time aggregation model. This runs as a separate process to the dissemination of the aggregation results. Client applications can then send a request to the Property Estimator API for an aggregation and the process can draw upon the persisted model to provide a result. This use case also requires a defined configuration from which the Property Estimator API can build the model.
- The Event-driven process: This use case treats the Property Estimator API as an ETL (Extract Transform Load) tool which sits between the Source STAPI services and the aggregation STAPI service. The API retrieves inputs from the Source STAPI services by subscribing to their MQTT endpoints. This can be considered the Export component of the API. The API then transforms the retrieved readings to a spatiotemporal aggregate. The API can then POST this aggregate to the STAPI aggregation service. This is can be considered the Load component of the event driven workflow. This use case requires a defined set of STAPI Features of Interest for the API to subscribe to.
- The "agent" driven circular STAPI process: This use case is an alternate event-driven process managed by an agent. The agent retrieves the defined STAPI Features of Interest from the aggregate STAPI service and then uses this to trigger the same process as use case two. The nature of this "agent" is at present abstract. It could be fulfilled by the client or by another scheduled or smart process within the Property Estimator API.

The technical implementation for this pilot covered the first two of these approaches, with the technical architecture in place to satisfy the third approach. In order to do so the implementation was flexible, light weight and only focused on the necessities. This prioritized the end utilization rather than the need to produce a full production ready implementation. Hopefully given the results of this implementation a number of decisions can then be made regarding which is the most appropriate use case(s) to support going forward.

The Property Estimator API component can be broken down into 3 parts:

- The API exposed to the client and the STAPIs. This includes how the API should be structured and what standards it should follow.
- The property estimation. This includes how the estimations are calculated, for both occupancy and air quality, and how the input data is incorporated.
- The output process. The nature in which the data is returned to the requester, potentially accounting for whether the requester is a service or client.

The API structure took inspiration from the Routing Pilot work conducted in 2019, which extended the draft OGC API – Processes standard. This is the most recent implementation of an OpenAPI inspired processing service, and is the closest the pilot participants had to a proto-standard processing service, along-side the draft OGC API - Common specification and OGC API - Features standard. That said, the scope of this pilot may not allow for a full HATEOAS (Hypermedia as the Engine of Application State) implementation given the flexibility required to support all of the other components. The API components exposed to the client should be sufficient to support the first two use cases. From the client perspective this is fairly straight forward. Only a few endpoints need be exposed to allow for the client to request an aggregation. This satisfied the client aspects of approaches 1 and 2, and the same endpoints could be used for approach 3.

The request JSON payload structure depended on the type of estimation required, either air quality or occupancy. The proposed example structure for air quality is below, for location estimate.

```
{
  "name": "airReadingRequest1",
  "air readings": [
    {
      "aggregation type": "on-the-fly",
      "default sensor type": [
        "PM2.5"
      ],
      "default radius": 100000,
      "default start time": "2019-11-28T04:28:58.704Z",
      "default end time": "2019-11-28T04:41:49.838Z",
      "default sensorThing API": "SensorUp",
      "centroids": [
        {
          "type": "Feature",
          "geometry": {
            "type": "Point",
            "coordinates": [
              127.3005228, 36.6296889
            ]
          }
        }
      ]
    }
  ]
}
```

The proposed example structure for air quality is below, for GMLID Feature estimate.

```
{
  "name": "string",
  "air readings": [
    {
      "aggregation type": "on-the-fly",
      "default sensor type": [
        "PM2.5"
      ],
      "default radius": 100000,
      "default start time": "2019-11-28T04:28:58.704Z",
      "default end time": "2019-11-28T04:41:49.838Z",
      "default sensorThing API": "SensorUp",
      "GMLIDs": [
        {
          "ID": "UUID_4552ad98-c383-48c2-a36b-fc517b2c3ffa"
        }
      ]
    }
  ]
}
```

The proposed example structure for occupancy is below.

```
{
  "name": "string",
  "occupancy readings": [
    {
      "default radius": 100000,
      "default start time": "2019-11-28T04:28:58.704Z",
      "default end time": "2019-11-28T04:41:49.838Z",
      "default sensorThing API": "SensorUp",
      "GMLIDs": [
        {
          "ID": "UUID_4552ad98-c383-48c2-a36b-fc517b2c3ffa",
          "start time": "2019-05-21T18:25:43-05:00",
          "end time": "2019-05-21T18:25:43-05:00"
        }
      ]
    }
  ]
}
```

Two methods for property estimation were required:

(1) Simple property estimation: This provided an estimation for a single point given the input parameters in the request payload. This single point was either chosen by the user or derived from the users chosen GMLID. The air quality estimation used the centroid x,y,z, the radius from centroid, start and stop times and a sensor type defining the particulate. This information was then

used to take an average through time, retrieving information from all of the sensors that fall within the chosen 3D spherical radius. For the occupancy estimation all of the ID elements within the chosen GML ID element were checked to see which elements have associated sensors. These sensors were then used to take an average number of people in the chosen element. This satisfied the simple asynchronous processing component.

(2) Persisted property estimation: This estimation created a persisted spatiotemporal model. The model was represented by raster data sets, each representing an area and height range (or bin) for a specific day. In order to create these raster datasets the persisted model accepted a configuration POSTed to the 'administer' endpoint of the API. This configuration provides information required by the model, including the chosen STAPI from which to build the model, the sensor type to base the model on and the minimum and maximum bounding box coordinates for the model extent. Once this configuration is received, the process checked that there were sensors for the chosen sensor type, on the chosen STAPI in the chosen area. If this is confirmed an MQTT client connection was made between the Processes API and the chosen SensorThings API, to observe all FeatureOfInterest updates. When an observation is received the process checked that the response matched the configuration criteria and if so added the data to the appropriate point dataset. Every time a data set was updated and had more than 3 point values, the IDW Interpolation was run to create the interpolated raster for that specific point dataset. Once the model had been running for a while successfully retrieving regular MQTT updates from the chosen STAPI, a catalog of raster datasets was generated. When an estimate request is made by the user specifying the "aggregation type" as "persisted" the user receives a link to a resource on the estimate endpoint. This JSON resource contains a list of links to the raster data sets that satisfy their estimate request. The user can then choose to download all or a subset of these raster data sets to display or interrogate on the client side. This satisfied the persisted model processing component and such a model could be drawn upon by any of the above approaches. Due to the scope of this pilot, this estimation process was constrained to just air quality.

These estimation processes relied on the SensorThings API storing a GML ID within either the FeatureOfInterest elements or Thing elements to associate sensors with parts of buildings. The result of the estimation request was stored as a resource at the estimation endpoint of the API, with the client being given the location of this resource in the 'Location' header of the POST response.

The implementation of the above approach is outlined in section 7.

6.2.4. Overview SensorThings API SensorUp

SensorUp’s role in the pilot was to provide two SensorThings API services with simulated observations. One SensorThings API implementation offered simulated indoor occupancy data and the other offered simulated $PM_{2.5}$ data. Figure 17 describes the components that SensorUp developed to create the simulated data.

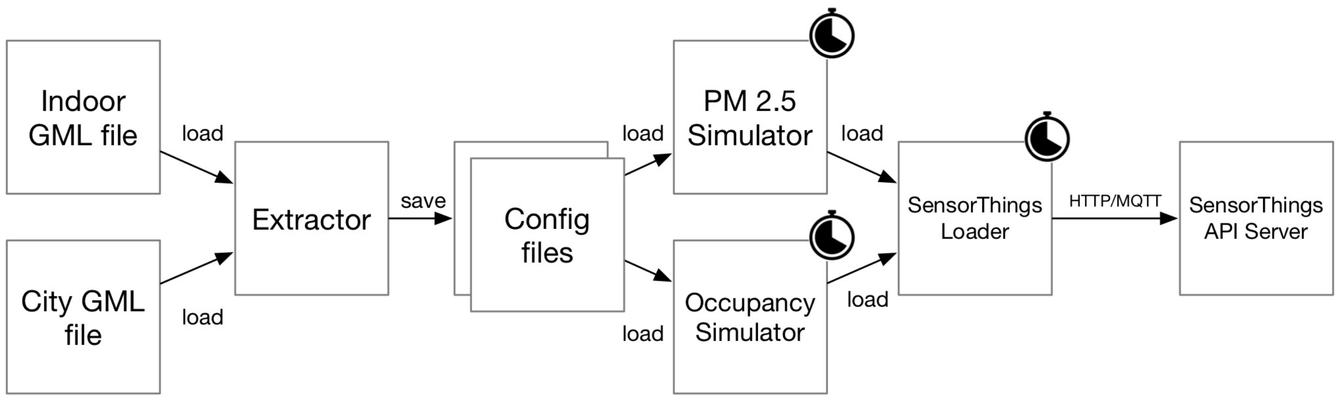


Figure 17. Architecture of the SensorUp $PM_{2.5}$ and Indoor Occupancy Simulator

6.2.5. Air Quality Simulator

SensorUp provided synthetic fine dust $PM_{2.5}$ observations for the five selected air quality stations. Four steps have been used to create the simulator:

- Selecting the location of five air quality stations;
- Considering 26 air quality stations around real stations;
- Appling Random Walk method to estimate synthetic observations for 26 air quality stations;
- Calculating the synthetic observation for each real station utilizing IDW interpolation and pushing calculated observations to the STAPI;

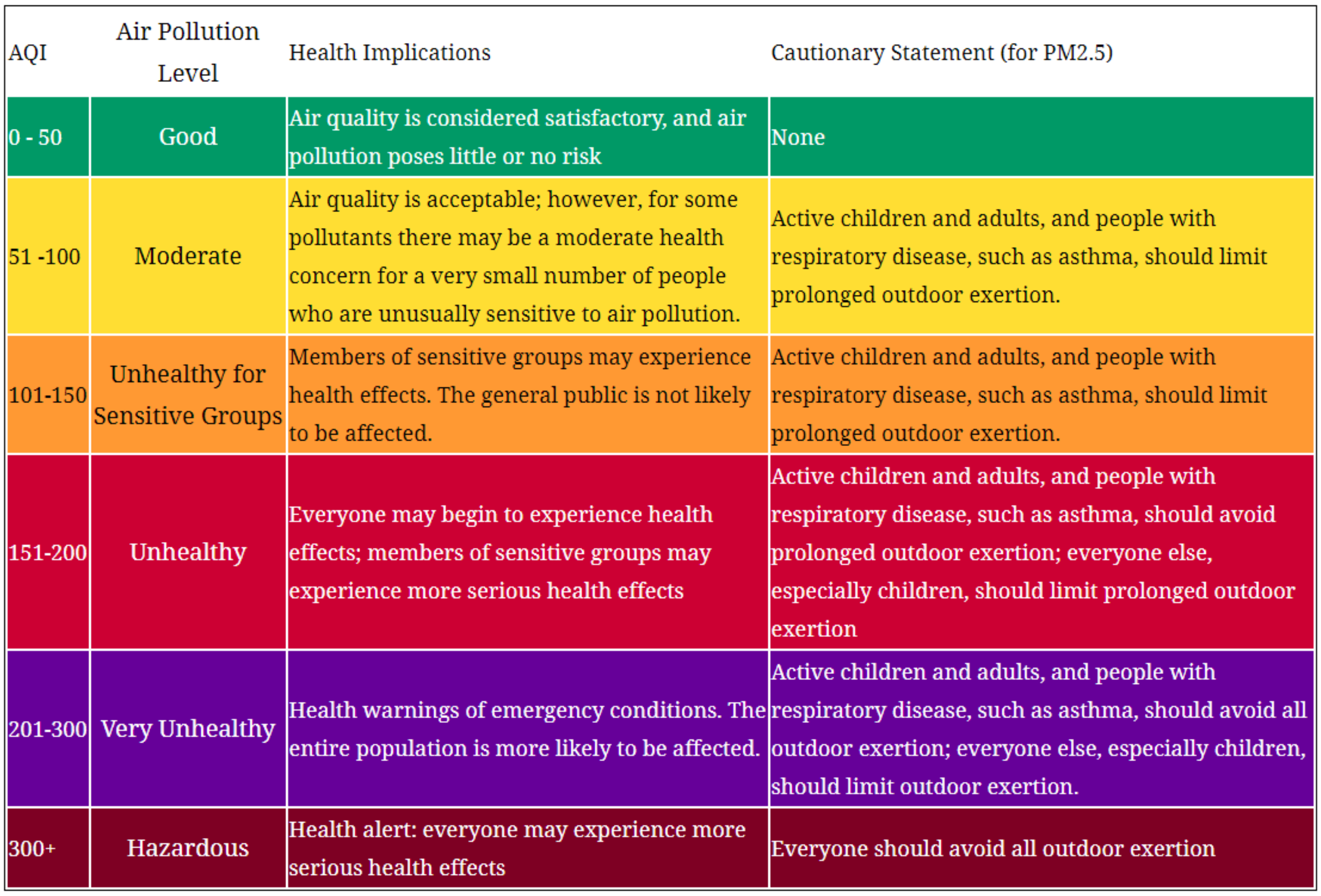
Step 1: Five air quality stations were selected. The criteria for selecting these five stations was that they were the closest stations to the area of study in this pilot provided by aqicn.org [http://aqicn.org/city/korea/sejong/bugang-myeon]. The five air quality stations were called "US" because their synthetic $PM_{2.5}$ observations are unknown and should be calculated by the simulator. Their names and locations are summarized in the following Table.

Table 1. Selected Air Quality Stations

Station name	Location	STA link to the Thing
Osong-eup	(127.3005228, 36.6296889)	link [https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(33978)]
Hansol-dong	(127.252529, 36.474172)	link [https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(33971)]
Areum-dong	(127.249653, 36.5177469)	link [https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(33964)]
Bugang-myeon	(127.370516, 36.527029)	link [https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(33957)]
Sinheung-dong	(127.292253, 36.592906)	link [https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(33950)]

Step 2: In this step, 26 stations have been randomly selected around real stations (i.e., “US” stations). They were called “KS” because it is assumed that their $PM_{2.5}$ observations are known. In other words, their $PM_{2.5}$ observations were estimated using the Random Walk method [https://en.wikipedia.org/wiki/Random_walk].

Step 3: In step 3, firstly, to start the process of simulating the $PM_{2.5}$ observations for “UK” stations, initial synthetic $PM_{2.5}$ observations for “KS” stations was needed. Six classes describing the index for the air quality were considered. Then, each of those 26 “KS” stations was randomly assigned to an air quality class. Air quality classes have been extracted by considering the Air Quality Index Scale and Color Legend provided by [aqicn.org]. The properties of each class are shown in Figure 18. Secondly, the random walk method was applied to estimate smoother synthetic $PM_{2.5}$ observations for each “KS” station. So, every 10 seconds, a new $PM_{2.5}$ value was estimated based on the random walk method for each of those 26 “KS” stations. It is worth mentioning that the walking value is set to be 20 ug/m3.



AQI	Air Pollution Level	Health Implications	Cautionary Statement (for PM2.5)
0 - 50	Good	Air quality is considered satisfactory, and air pollution poses little or no risk	None
51 -100	Moderate	Air quality is acceptable; however, for some pollutants there may be a moderate health concern for a very small number of people who are unusually sensitive to air pollution.	Active children and adults, and people with respiratory disease, such as asthma, should limit prolonged outdoor exertion.
101-150	Unhealthy for Sensitive Groups	Members of sensitive groups may experience health effects. The general public is not likely to be affected.	Active children and adults, and people with respiratory disease, such as asthma, should limit prolonged outdoor exertion.
151-200	Unhealthy	Everyone may begin to experience health effects; members of sensitive groups may experience more serious health effects	Active children and adults, and people with respiratory disease, such as asthma, should avoid prolonged outdoor exertion; everyone else, especially children, should limit prolonged outdoor exertion
201-300	Very Unhealthy	Health warnings of emergency conditions. The entire population is more likely to be affected.	Active children and adults, and people with respiratory disease, such as asthma, should avoid all outdoor exertion; everyone else, especially children, should limit outdoor exertion.
300+	Hazardous	Health alert: everyone may experience more serious health effects	Everyone should avoid all outdoor exertion

Figure 18. The air quality index provided by aqicn.org [http://aqicn.org]

Step 4: In this step, firstly the geographical distance between “KS” stations and “US” is calculated. Secondly, the value of $PM_{2.5}$ (calculated by Random Walk method) for all 26 “KS” stations and also their geographical distance to the “US” station are fed into the IDW interpolation. The output of applying the IDW interpolation is the calculated $PM_{2.5}$ observation for the “US” station. Finally, the calculated $PM_{2.5}$ values for all “US” stations will be pushed to the STAPI.

The process of simulating air quality observations is shown in Figure 19.

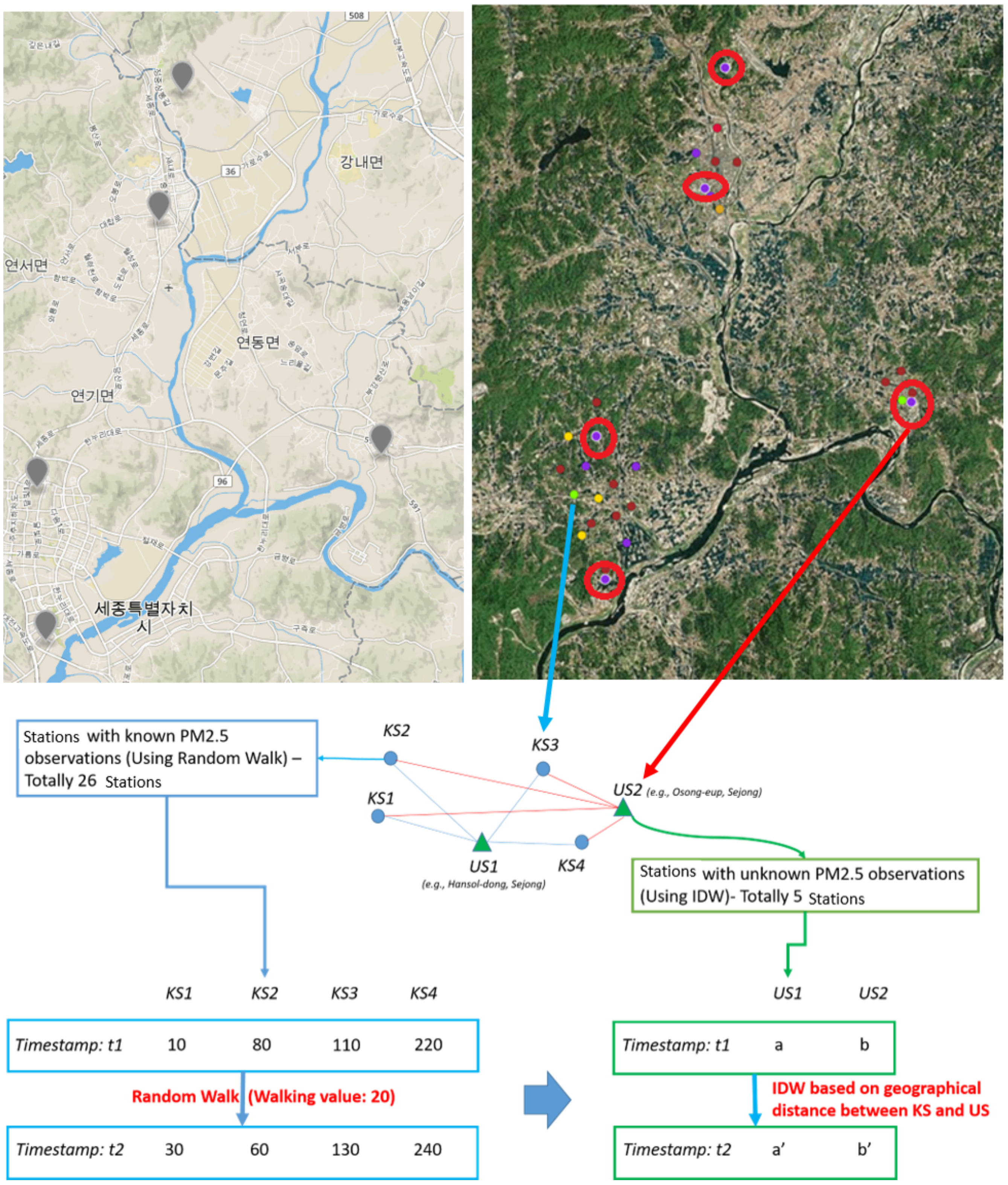

강내면
연서면
연동면
연기면
세종특별자치시
Stations with known PM2.5 observations (Using Random Walk) – Totally 26 Stations
KS2
KS3
KS1
US2 (e.g., Osong-eup, Sejong)
US1 (e.g., Hansol-dong, Sejong)
KS4
Stations with unknown PM2.5 observations (Using IDW)- Totally 5 Stations
KS1 KS2 KS3 KS4
Timestamp: t1 10 80 110 220
Random Walk (Walking value: 20)
Timestamp: t2 30 60 130 240
US1 US2
Timestamp: t1 a b
IDW based on geographical distance between KS and US
Timestamp: t2 a' b'

Figure 19. The methodology applied to simulate air quality

Figure 20 shows a visualization of the air quality simulator. All "US" and "KS" stations are shown, and color coded according to their $PM_{2.5}$ observations.

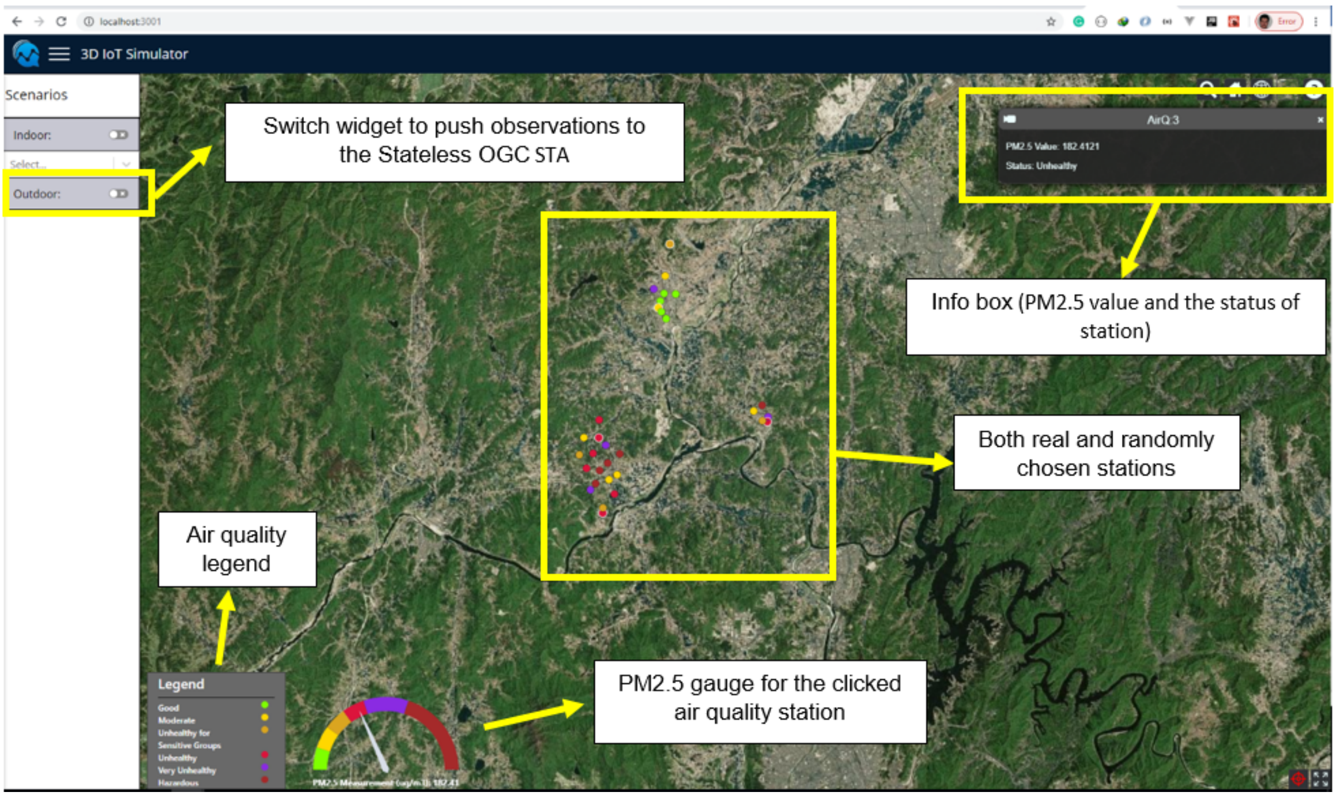

localhost:3001
3D IoT Simulator
Scenarios
Indoor:
Select...
Outdoor:
Switch widget to push observations to the Stateless OGC STA
AirQ:3
PM2.5 Value: 182.4121
Status: Unhealthy
Info box (PM2.5 value and the status of station)
Both real and randomly chosen stations
Air quality legend
Legend
Good
Moderate
Unhealthy for Sensitive Groups
Unhealthy
Very Unhealthy
Hazardous
PM2.5 Measurement (ug/m3): 182.41
PM2.5 gauge for the clicked air quality station

Figure 20. The air quality simulator

6.2.5.1. Occupancy Simulator

We use three steps to create the simulators:

- Loading an IndoorGML file and creating an adjacency graph
- Filtering unnecessary nodes/edges from the adjacency graph to create a connectivity graph
- Simulating the occupancy and pushing it to STAPI

Step 1: The occupancy simulator gets an IndoorGML dataset as input and extracts the adjacency graph with navigable and non-navigable nodes and edges. In this graph, nodes represent cells and edges are the connection between the cells. A weight is assigned to each edge based on the Euclidean distance between nodes.

Step 2: In step 2, nodes are classified into navigable and non-navigable cells by using semantic information provided in the IndoorGML, i.e., the description of each cell. In this pilot, non-navigable cells are areas where authorization is required for entering that area, such as electric room, fire pump room, etc. Navigable cells, on the other hand, include areas where they are accessible by public, for example, lobby, stairs, meeting rooms, etc. Then, all non-navigable cells and edges connected to them, as well as parking lots where this pilot is not concerned of its occupancy, need to be removed from the graph. The remaining nodes and edges form a connectivity graph that can be used for occupancy simulation. In the simulator, the number of people in each node is updated in a configurable frequency and the results are then pushed to STAPI Figure 21.

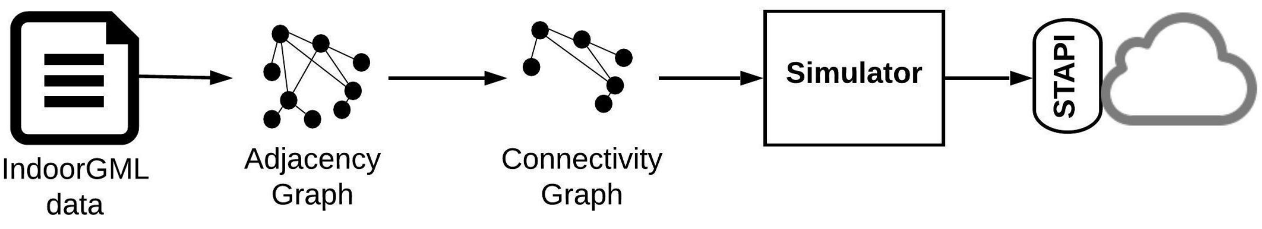


Figure 21. General scheme of occupancy simulator

Step 3: In the simulator, a random number (representing the number of people in a cell) is assigned to some random nodes. In every updating process of occupancy of a node, a person can either stay in the node or he can go to another node considering the weight value of each connected edge. The weight value has a direct relation with Euclidean distance between two nodes and it is calculated as:

Where node i is connected to another node j, w_{ij} is the weight value of the edge connecting nodes i and j, and d_{ij} is euclidean distance between two nodes i and j. Therefore, the longer the distance between two nodes, the lower the chance of choosing that edge and its connected node for transition. Figure 22 shows this process for a connectivity graph consisting of five nodes A, B, C, D, and E, in different timestamps. In this figure, numbers in the circles are occupancy, arrows show the transition of a person to a different or same node, and numbers on the edge of the connectivity graph represent the weight values.

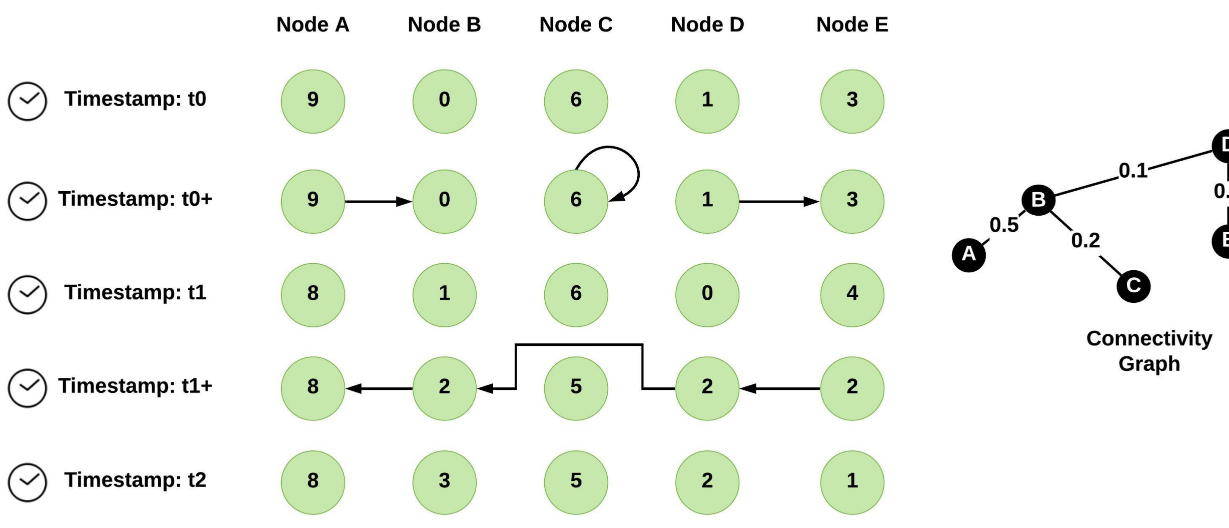
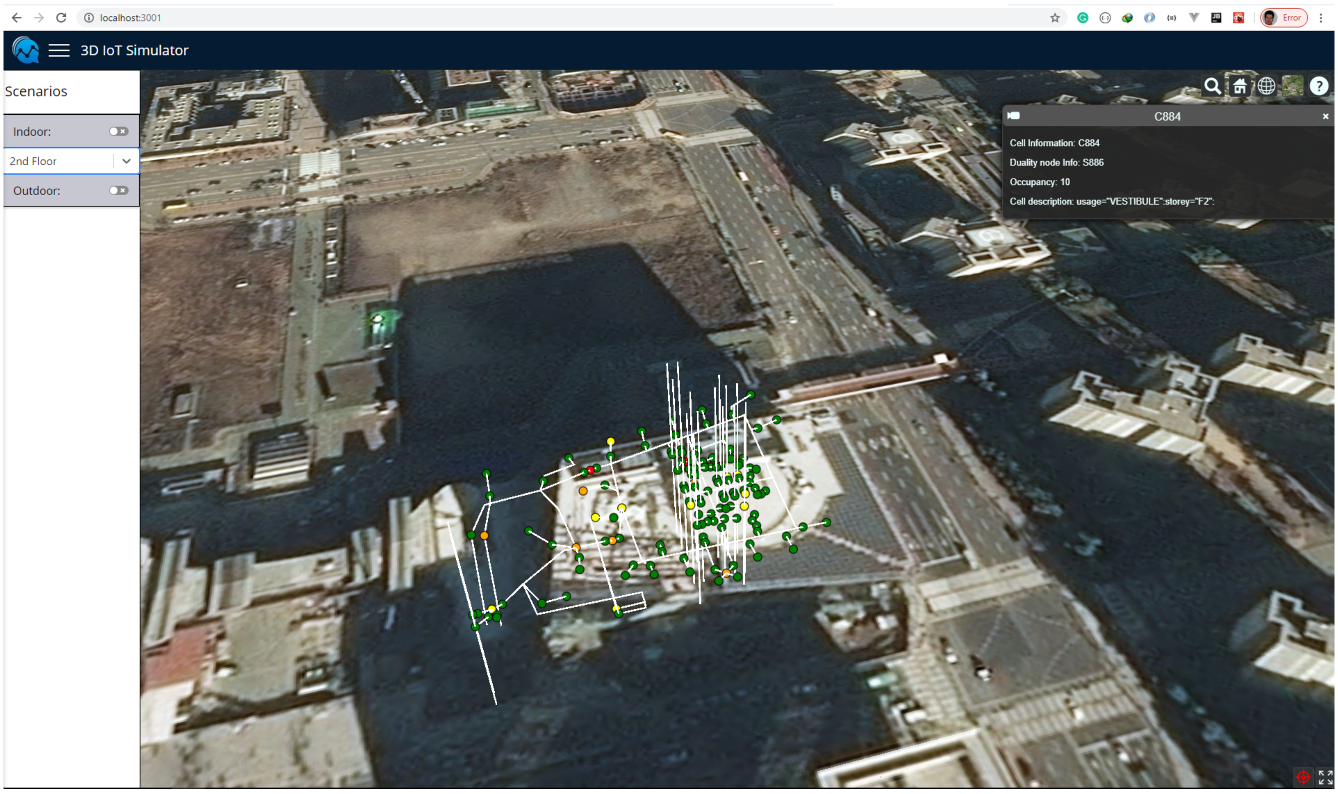


Figure 22. The occupancy updating process for five node A, B, C, D, and E in different timestamps. Numbers in the circles show the occupancy

Figure 23 shows the connectivity graph of the second floor of Alphadom building. In this figure, circles are nodes of the connectivity graph, and white lines are the edges. Occupancy is coded into different colors in each node. By the end of updating occupancy, the results are pushed to STAPI.



C884

Cell Information: C884

Duality node Info: S886

Occupancy: 10

Cell description: usage="VESTIBULE":storey="F2":

Figure 23. A visualization of occupancy simulator. Occupancy is color coded

6.2.6. Overview Augmented Reality GeoPortal Skymantics

6.2.6.1. IndoorGML

Two approaches were taken when trying to import and visualize either the Alphadom or Lotte World data. Note that Lotte World is a recreation complex. Initially a custom C# script was utilized to import the IndoorGML data and convert each feature to a game object for automatic propagation. This was limited by the way JSON would get parsed. Some of the features would not convert properly when parsing the data. The second approach was to use the STEMLAb InViewer desktop plugin for Unity. This did get the IndoorGML to visualize and run in Unity. The difficulty came when the end user wanted to quickly view the IndoorGML data again, as they have to input the IndoorGML source information every time. Given the file size of the IndoorGML data, storing it locally on the mobile device would not be an option.

Recommendations for further work: regarding this would be to have an indoor viewer inspired plugin created that would stream the data from a server.

6.2.6.2. 3D Tiles

Importing and Rendering 3D tiles into Unity posed a bit of a challenge. The first approach was to use the Mapbox Software Development Kit (SDK) for Unity – the 3D games engine (also referred to as Unity3D). It was the initial choice for rendering map data in Unity3D. Unfortunately, despite having a tileset.json file associated with the data, it was not possible to import the data into the studio using the SDK. Best guess is that the SDK does not support the B3DM file structure. A possible solution to this is to use the NASA-AMMOS /Unity3DTiles plug in for Unity 3D. This plugin allows you to visually display the data locally or by streaming it to Unity for rendering and visualization. When the participants tried to run it, there were several errors while compiling.

Recommendations for further work: despite having the ability stream 3D Tile data, the size of the dataset being streamed might cause performance issues if there is not a fast network connection available. The file size of the data might cause an issue performance on a mobile app.

6.2.6.3. SensorThings API

The approach that was used was to use the Unity3D network handler to make the GET request to the SensorThings API. This required that we create a code container gameobject that made the request and parsed the data and another script that made the button and text association that would display the final result retrieved from the api. First the URL was filtered to only request the $PM_{2.5}$ data and sorted so the most recent data was displayed at the top.

```
http://193.196.138.56:8080/frost-
airquality/v1.0/Datastreams(6)/Observations?$select=resultTime,result&$orderby=resultT
ime%20desc
```

Once that data was received, we had to parse the JSON. Unity3D does not have a natively built-in JSON handler, but a plugin was used.

Once the data is parsed it has to be handed off to a debugger.log in Unity3D that has a line associated with it to display the data being received in a gameobjected labeled text. This value text is associated with a button User Interface (UI) that is blended into the image that appears once the image target is triggered.

AR (Augmented Reality) was used as a part of this pilot. Vuforia SDK was installed into Unity3D for AR to work. To make this all come together a QR Code was created that would trigger an image. That image would house the button and text data spoken about above. The result of the API call would be displayed in the results section of the rendered image on the mobile device.

Recommendations for further work: Unity 3D can still be used for AR visualization of STAPI data. Vuforia SDK is only specific to Windows and Mac. Exploring ARCore and ARkit to handle the AR would be worth investigating as they run on Linux. The idea of markerless AR might also be a good idea for a future test. This is similar to Pokemon Go. When a user is in close proximity to the assets they want to view it would pop up on their screen.

6.2.7. Overview GeoPortal STT

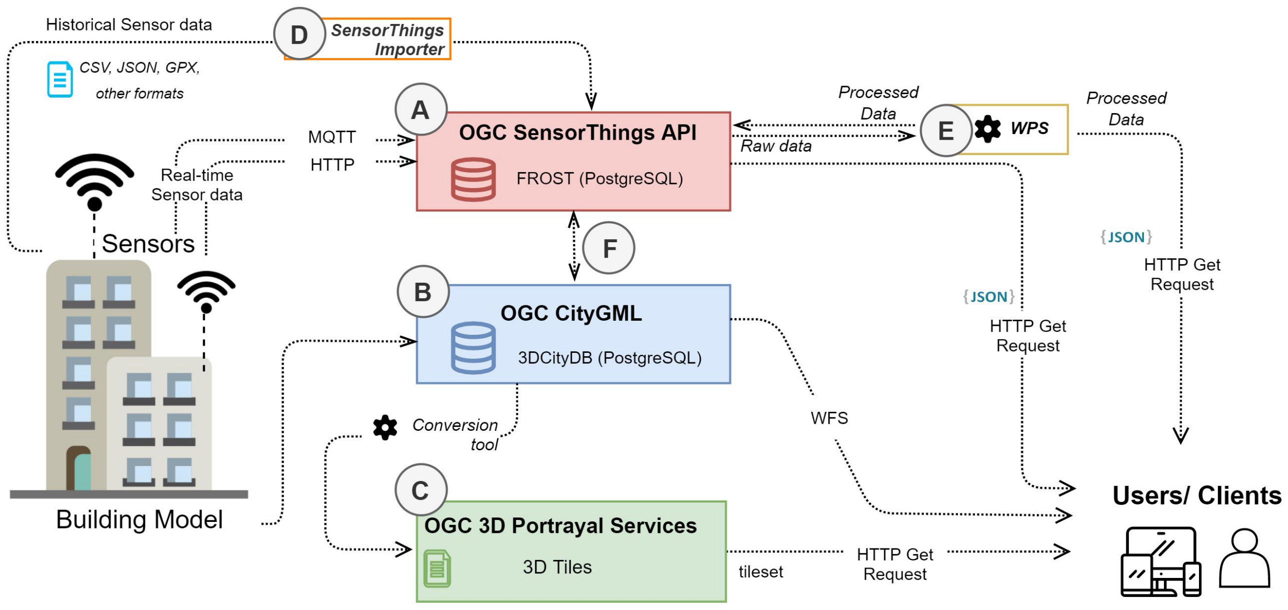


Figure 24. CityThings concept

6.2.7.1. Basic of CityThings (CityGML – SensorThings)

The CityThings concept is a simple approach to manage and integrate the CityGML 3D city model data (Figure 24A) and sensor or IoT data streams from the SensorThings (Figure 24B). The associations between the sensor systems and the 3D city models (Figure 24AF) are created and stored by the enrichment on the SensorThings at the properties of the physical sensor systems which are stored in as a Key Pair Value (KPV) in the Things's property array and must contain the unique CityGML ìdentifier (`gmlid`) of the building sections where the sensor systems belong to. This SensorThings server that has already been enriched with the linkage to the CityGML city model is referred to as `CityThings`. In addition, other important information or associations can be added to the Things's property array such as an association with the WFS model server. For example, the following JSON shows an example of the Things entity from the CityThings.

```
{
    "name": "Air Quality Sensor 1",
    "description": "Air Quality Sensor Station in Sejong",
    "properties": {
        "citygmlid":"[CityGML's gml-id here]",
        "deploymentCondition": "Deployed in the third-floor balcony",
        "wfsAssociations": [
            {
                "Type": "Feature",
                "Link":
"https://wfs.domain.local/collections/Sector5buildings/items/35"
            },
            {
                "Type": "Collection",
                "Link": "https://wfs.domain.local/collections/Sector5buildings"
            },
            {
                "Type": "Feature",
                "Link": "https://wfs.domain.local/collections/HeliPad/20"
            }
        ]
    }
}
```

6.2.7.2. The CityThings and the 3D Geoportal

The CityGML models were to be converted to the OGC 3D Tiles format and stored on the OGC 3D Portrayal Service (Figure 24C) to be visualized on the GeoPortal. The attribute information of the CityGML model was to be stored in the 3D Tiles dataset, which included the unique `gmlid` on each building or building part. With the CityThings setup, users or GeoPortals (clients) can use this `gmlid` to make a request for sensor or IoT datastreams through the SensorThings interface. For example, the following HTTP request shows a request for the SensorThings Datastream which observes temperature and locations recorded in a CityGML model.

```
http://<STA-Base>/Datastreams?$expand=Thing&$expand=ObservedProperty&
$filter=Things/properties/citygmlid eq □citygmlid-here□&
$filter=ObservedProperty/name eq □temperature□&
$filter=Datastream/unitOfMeasurement eq □Celsius□
```

6.2.7.3. The CityThings and the WPS – Property Estimator

There are two main approaches to handle the datastreams from the WPS (Figure 24E) which are:

1. Storing the processed data in the SensorThings API implementations, and
2. Sending the processed data to the client directly.

If the WPS process required high processing power and long processing time, the processed data should be stored in the SensorThings. In this case, a new set of Things, Sensors, FeaturesOfInterest and Datastreams entities that represent the WPS should be registered in the SensorThings API implementations to avoid confusion between the raw dataset and processed dataset. If the WPS process required low processing power and short processing time, the processed data can be transferred directly to the clients. The comparison and evaluation of these two approaches are needed to ensure better performance and scalability.

Chapter 7. Implementation of the 3D IoT Smart City Platform

The concept of the 3D IoT Smart City Platform described in section 6 has been implemented by the participants of this Pilot. The result is not one unified system but the implementation of the two main use cases on outdoor air quality and indoor occupancy. Some components are used in both use cases, others are specific for one use case only.

The developed components and the four GeoPortals implemented in the 3D IoT Pilot are available at the OGC 3D IoT Pilot persistent demo page

note

```
LINK TO BE ADDED, SEE ISSUE 101
```

The components used in both use cases, as well as the data preparation, are documented in section 7.1. It includes:

- 3D Models in CityGML and IndoorGML
- the WFS model server by GIS.FCU

Section 7.2 documents the implementation of the first use case on the Outdoor 3D City Model and outdoor air quality sensors, especially real-time monitoring of micro-dust. The section covers the following components:

- STT Geoportal
- WFS model server by GIS.FCU for 3D buildings
- WPS property estimator by Helyx SIS.

Section 7.3 focuses on the Indoor Building Model and indoor sensors such as real-time monitoring of indoor occupancy. The section covers the following components:

- Gaia3D Geoportal
- Cyient Geoportal
- Skymantics Geoportal (Augmented Reality)

7.1. Components / Services used in both air quality and indoor occupancy use case

7.1.1. Preparing 3D Models in CityGML and IndoorGML

7.1.1.1. CityGML and 3D Tiles

Due to the security situation in Korea, export of any digital maps to outside of Korea is forbidden by the law. For the 3D IoT pilot project, the pilot participants were obliged to use 3D tiles data of the

Sejong smart city area derived from CityGML data sets to avoid conflict with Korean Law. The derivation process is composed of the steps: first the conversion of CRS from the local reference system to EPSG:3857 and second, derivation of CityGML data to 3D tiles by FME. During the conversion process through FME, the gml ID of each feature in CityGML data was mapped to 3D Tiles as shown in Figure 25.

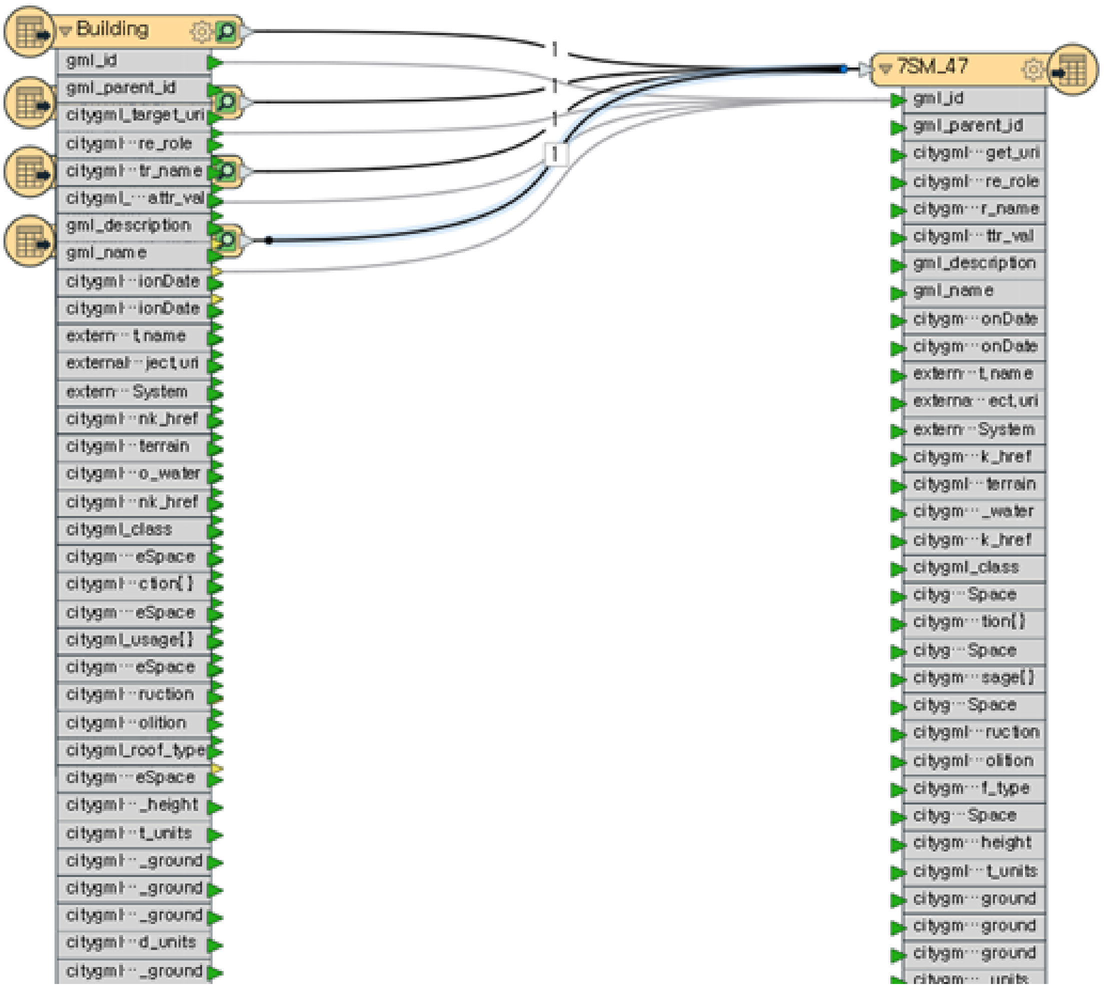


Figure 25. Mapping Attributes (GML ID) from CityGML to 3D Tiles by FME.

As the Sejong's CityGML data is not contained with real-world height information on each building, the pilot participants clamped the building models to fit the 3D ellipsoid on the web-client. In the Cesium web client, this is done by adjusting the height offset of the model matrix of the 3D Tiles with the following JavaScript code:

```
tileset.readyPromise.then(function () {
        var heightOffset = 99 //height offset here;
        var boundingSphere = tileset.boundingSphere;
        var cartographic = Cesium.Cartographic.fromCartesian(boundingSphere.center);
        var surface = Cesium.Cartesian3.fromRadians(cartographic.longitude,
cartographic.latitude, 0.0);
        var offset = Cesium.Cartesian3.fromRadians(cartographic.longitude,
cartographic.latitude, heightOffset);
        var translation = Cesium.Cartesian3.subtract(offset, surface, new Cesium
.Cartesian3());
        tileset.modelMatrix = Cesium.Matrix4.fromTranslation(translation);
    });
```

After the conversion process, the resulting 3D Tiles building model can be loaded in a Cesium web-client and clamped to the ellipsoid as shown in Figure 26.

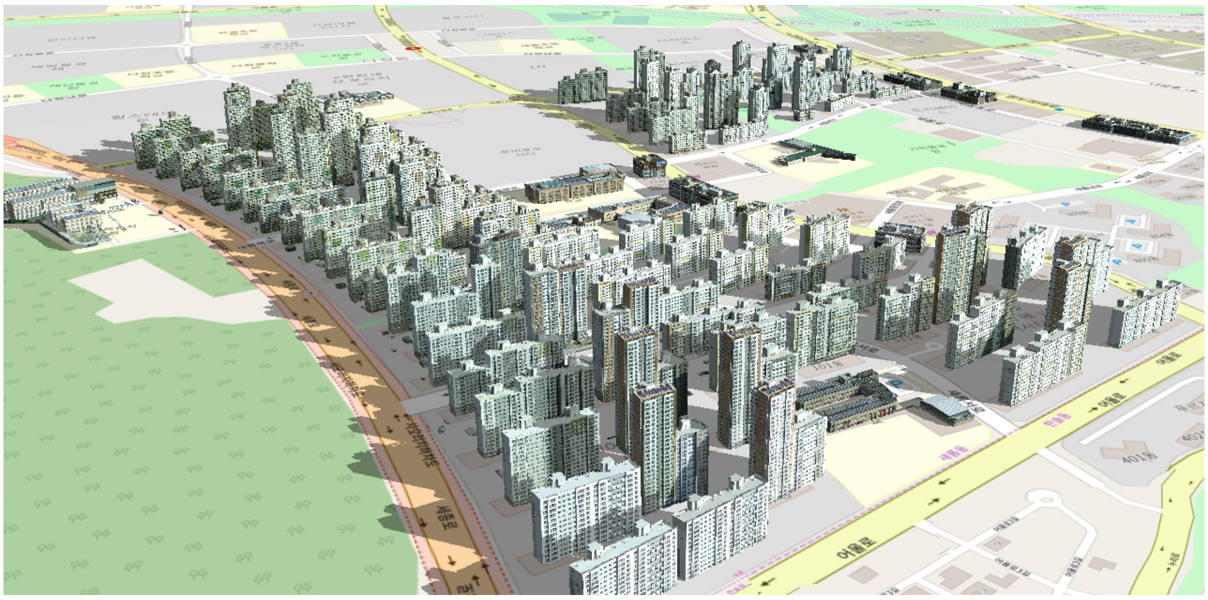


Figure 26. web-based visualization of Sejong 3D Building Model

7.1.1.2. IndoorGML

For the pilot project, the pilot participants prepared an indoor 3D model of a building called Alphadom, which is a complex shopping mall located in Seongnam city. The initial data was built using InEditor, which is an open source IndoorGML editing tool available at https://github.com/STEMLab/InEditor as shown in Figure 27.

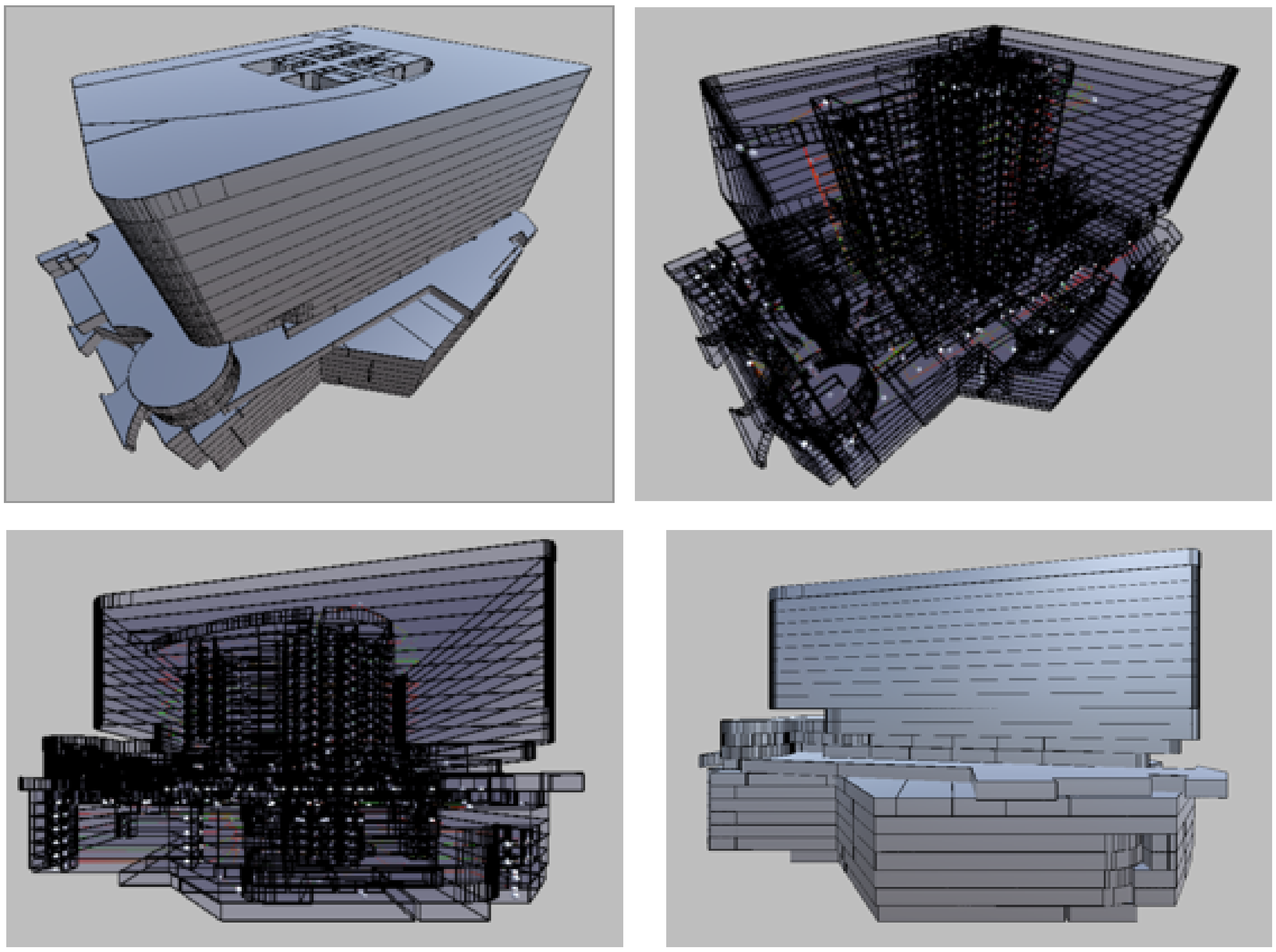
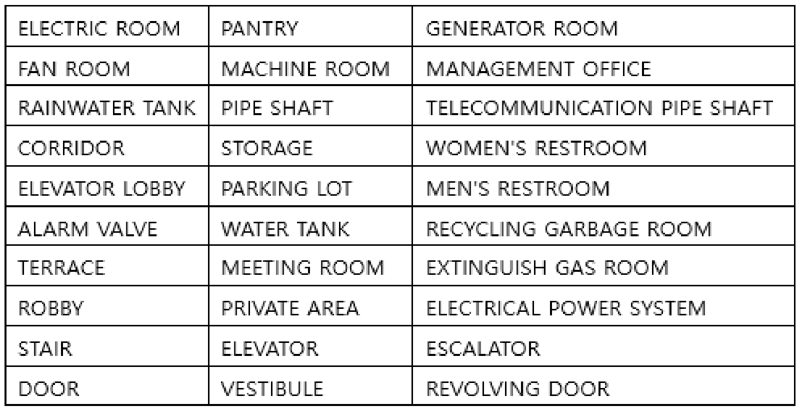


Figure 27. IndoorGML data for Alphadom.

This building is composed of 2115 cells with 22 floors (7 underground floors and 15 ground floors). While the initial IndoorGML data was given in a local reference system, it is converted to EPSG:3857 and the height in meters. Since the version of IndoorGML that was applied is v.1.0.3, the storey attribute is not explicitly defined. The pilot participants added the storey data as a gml description, as well as the usage of cell. The list of usages is given as below (Figure 28).



ELECTRIC ROOM	PANTRY	GENERATOR ROOM
FAN ROOM	MACHINE ROOM	MANAGEMENT OFFICE
RAINWATER TANK	PIPE SHAFT	TELECOMMUNICATION PIPE SHAFT
CORRIDOR	STORAGE	WOMEN'S RESTROOM
ELEVATOR LOBBY	PARKING LOT	MEN'S RESTROOM
ALARM VALVE	WATER TANK	RECYCLING GARBAGE ROOM
TERRACE	MEETING ROOM	EXTINGUISH GAS ROOM
ROBBY	PRIVATE AREA	ELECTRICAL POWER SYSTEM
STAIR	ELEVATOR	ESCALATOR
DOOR	VESTIBULE	REVOLVING DOOR

Figure 28. List of Cell Usages.

The data set also includes the navigation network to facilitate the path finding in indoor space. It is defined as a separate space layer from the topographic layer.

7.1.2. WFS model server by GISFCU

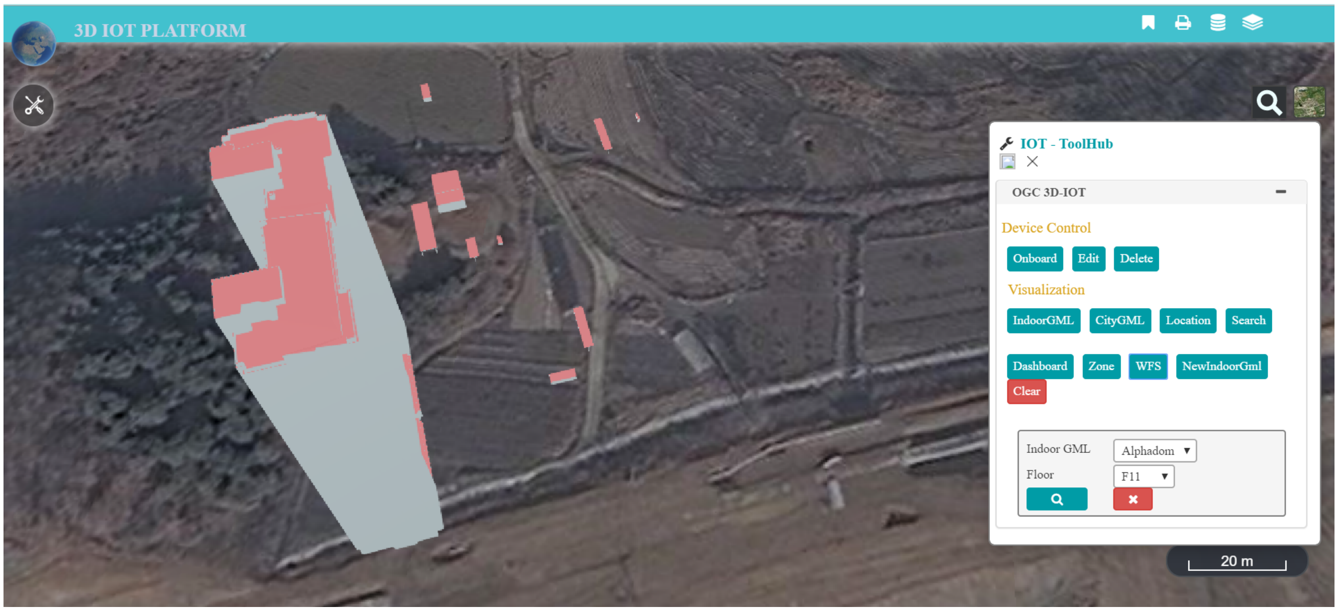


Figure 29. WorkFlow of GML Feature Queries.

In this pilot, the implementation of Web Feature Services follows the OGC API - Features version 1.0 standard, and there are two different datasets: one in CityGML and the other in IndoorGML.

7.1.2.1. CityGML Building Features

The client can send a request with a bounding box or the GML ID of the building to query building features of Sejong City. The API services will return the GeoJSON encoding building features. Since the support for 3D models is out-of-scope of GeoJSON in OGC API - Features, the pilot participants added the glTF model URL as property of building features, so after receiving the GeoJSON response, the client can also download 3D models directly by parsing the property `modelUrl` of the features in the response GeoJSON.

7.1.2.2. Preprocessed glTF

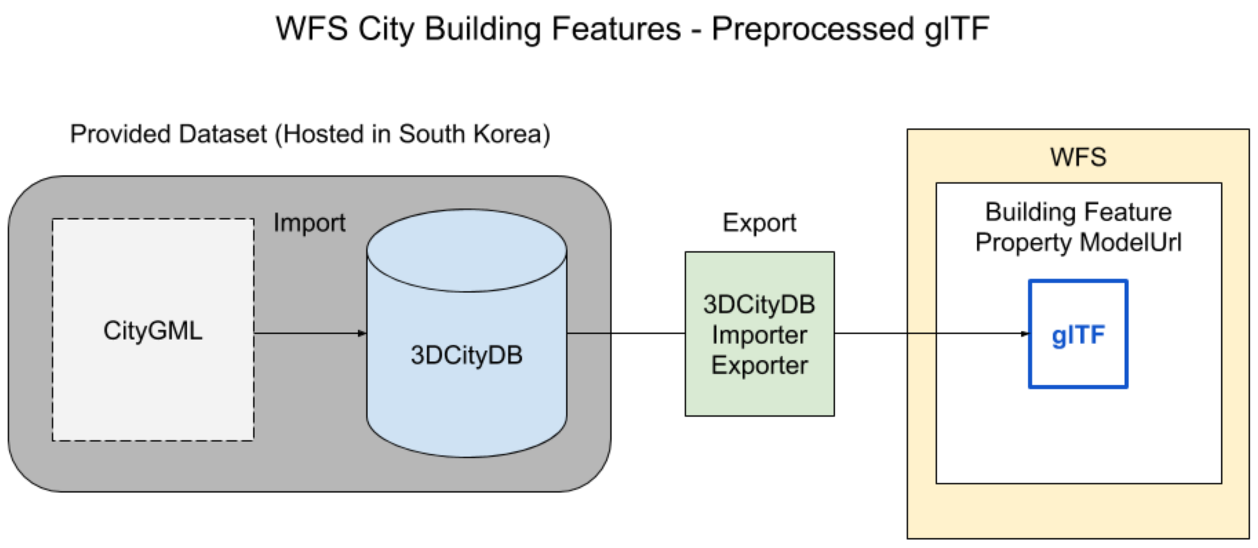

WFS City Building Features - Preprocessed glTF
Provided Dataset (Hosted in South Korea)
CityGML
Import
3DCityDB
Export
3DCityDB Importer Exporter
WFS
Building Feature Property ModelUrl
glTF

Figure 30. WorkFlow of preprocessing glTF.

Due to the data policy, the CityGML data cannot be persisted outside South Korea, so the CityGML dataset was imported to the 3DCityDB and hosted in South Korea. Therefore, participants only had access to 3DCityDB instead of the source data CityGML. For WFS glTF model provision, it was proposed to put the glTF building models in the properties of GetFeature responses. This would required all the building models to be preprocessed, the batch exporting process to be asynchronous and glTF models to be exported with 3DCityDB Importer Exporter. The exported building model had to be named as its GML ID, and then the feature could be connected to glTF model by setting the feature id as GML ID.

7.1.2.3. Sejong City Building Feature and Model Services

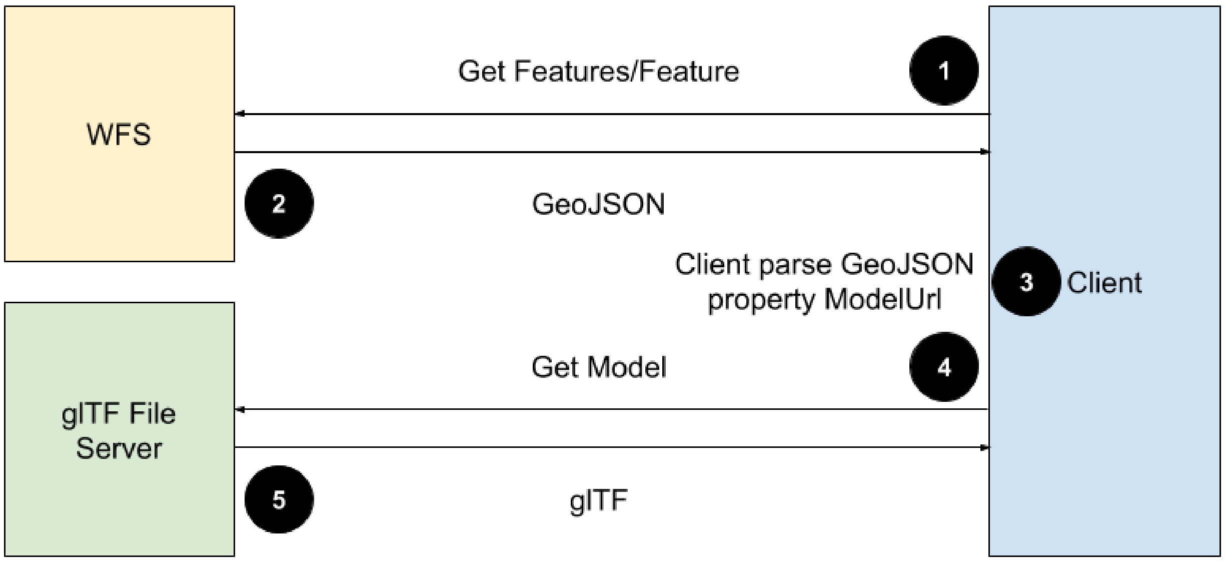


Figure 31. Interaction between client and OGC API - Features operations.

The client can query Sejong City Building features with a 2D bounding box using OGC API – Features capabilities, and query the specific building feature with GML ID via the same API. The service returns the query result in GeoJSON format, and each building feature has a property `modelUrl`. Additionally, after each query operation, the client can download the building model in glTF format instead of all the buildings by the GetModel operation. The same operation was implemented in the Testbed 14 CityGML and AR Service. The client can send separate requests with each of these `modelUrls` and retrieve the glTF models.

- OGC API - Features
  - GET /collections/SejongCityBuildings/items

Table 2. GetFeatures Parameter

Parameter	**Description**	**Remarks**
bbox (optional)	Only features that have a geometry that is within the bounding box are selected.	Provided as 4 numbers, WGS84 coordinate system.

The following example illustrates how to retrieve a feature:

https://tm.gis.tw/3dwfs/collection/SejongCityBuildings/items?

bbox=127.2515,36.4815,127.2525,36.4825

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "id": "UUID_4552ad98-c383-48c2-a36b-fc517b2c3ffa",
      "geometry": {
        "type": "Point",
        "coordinates": [
          127.251864591667,
          36.4819001943496,
          47.444417
        ]
      },
      "properties": {
        "gmlid": "UUID_4552ad98-c383-48c2-a36b-fc517b2c3ffa",
        "modelUrl":
"http://tm.gis.tw/3dwfs/static/citydbLoD2ExampleNoTiling/Tiles/0/0/1/UUID_4552ad98-
c383-48c2-a36b-fc517b2c3ffa.gltf"
      }
    },
    ...
  ]
}
```

- GetFeature
  - GET /collections/SejongCityBuildings/items/{featureId}

Table 3. GetFeature Parameter

Parameter	Description	Remarks
featureId (required)	Local identifier of a feature.	Using GML ID as local identifier.

The following example requests to retrieve feature:

https://tm.gis.tw/3dwfs/collection/SejongCityBuildings/items/UUID_dadf7c2b-dd5d-4e65-9709-e3f2ccef8655

```
{
  "type": "Feature",
  "id": "UUID_dadf7c2b-dd5d-4e65-9709-e3f2ccef8655",
  "geometry": {
    "type": "Point",
    "coordinates": [
      127.251977477379,
      36.4820200560248,
      46.357362
    ]
  },
  "properties": {
    "gmlid": "UUID_dadf7c2b-dd5d-4e65-9709-e3f2ccef8655",
    "modelUrl":
"http://tm.gis.tw/3dwfs/static/citydbLoD2ExampleNoTiling/Tiles/0/0/2419/UUID_dadf7c2b-
dd5d-4e65-9709-e3f2ccef8655.gltf"
  }
}
```

After parsing all the `modelUrl`, the client could download glTF models and visualize them in the glTF support client.

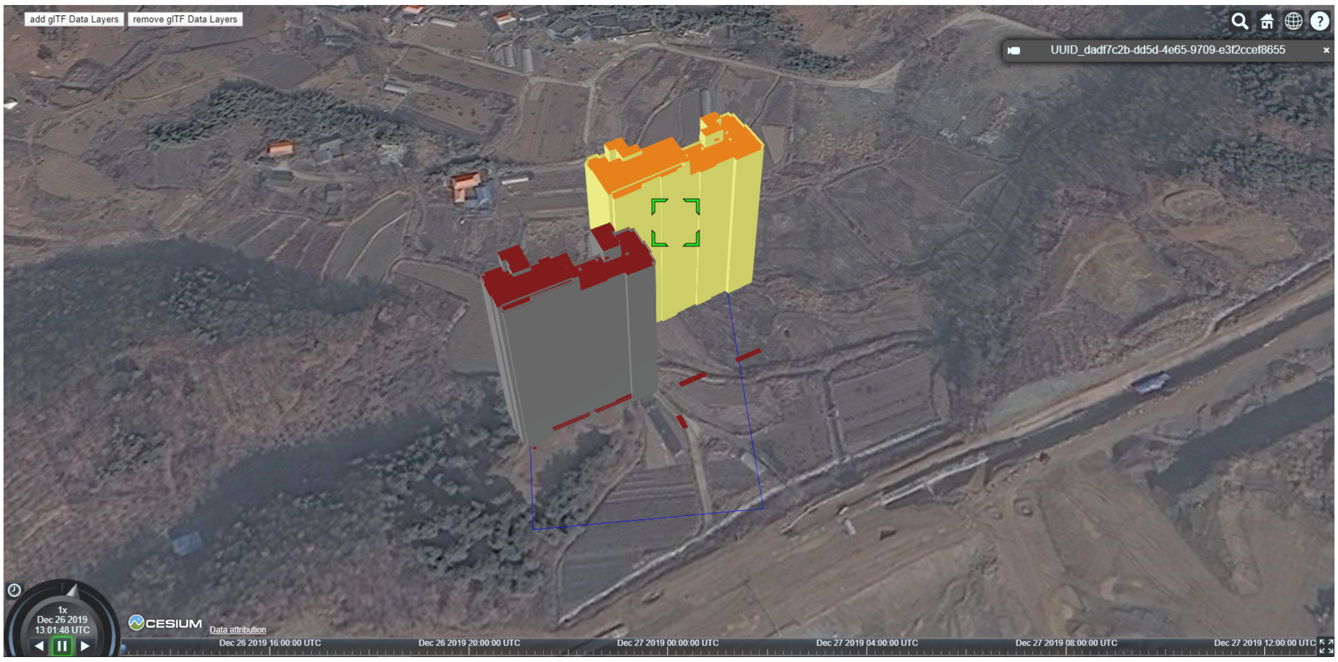


Figure 32. Showing glTF models of a query result on Cesium client.

7.1.2.4. Indoor Features

There would be several types of Indoor feature queries. A simple spatial query of feature intersection or containment within the bounding box of the 3D points, semantic query on classification of the cell or level, and more complex query like querying rest rooms on the optimal path from point A to point B. Currently there is no classification of the cellspace in the pilot demo's IndoorGML dataset, and the IndoorGML dataset is not connected to the Sejong CityGML dataset. The GeoPortal clients preprocessed the IndoorGML dataset and directly imported the whole Indoor data to the portal. Thus, the participants suggested that future work on the service should include the combination of Indoor and City Building Feature queries. The client scenario of the GeoPortal would be: First, query the City Building features, and after we get the GML IDs from the feature, we can query the Indoor features by the same GML IDs we retrieved from the CityGML feature request. The demo IndoorGML dataset in this pilot supports querying by bounding box and GML ID. The client can query the whole IndoorGML by GML ID of `IndoorFeatures` tags in IndoorGML. The response format is GeoJSON, the GML files can be downloaded by parsing the property `modelUrl` of the features in the response GeoJSON.

- GetFeatures
    - GET /collections/IndoorBuildings/items

Table 4. GetFeatures Parameter

Parameter	Description	Remarks
bbox (optional)	Only features that have a geometry that is within the bounding box are selected.	Provided as 4 numbers, WGS84 coordinate system.

The following example requests to retrieve features:

http://tm.gis.tw/3dwfs/collection/IndoorBuildings/items?bbox=127,36,129,40

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "id": "IFs",
      "geometry": {
        "type": "Point",
        "coordinates": [
          127.102431920709,
          37.5133338408155,
          3
        ]
      },
      "properties": {
        "gmlid": "IFs",
        "modelUrl": "http://tm.gis.tw/3dwfs/static/indoorgml/LWG_IndoorGML-
Absolute_CRS.gml"
      }
    },
    {
      "type": "Feature",
      "id": "nc82ac00-9610-7fe8-d6b2-27da02bcb53f",
      "geometry": {
        "type": "Point",
        "coordinates": [
          127.112537967655,
          37.3941675311279,
          -21
        ]
      },
      "properties": {
        "gmlid": "nc82ac00-9610-7fe8-d6b2-27da02bcb53f",
        "modelUrl": "http://tm.gis.tw/3dwfs/static/indoorgml/Alphadom_IndoorGML-
Absolute_CRS-meter-Network_.gml"
      }
    }
  ]
}
```

- GetFeature
  - GET /collections/IndoorBuildings/items/{featureId}

Table 5. GetFeature Parameter

Parameter	Description	Remarks
featureId (required)	Local identifier of a feature.	Using GML ID as local identifier.

The following example illustrates a request for retrieving a feature:

http://tm.gis.tw/3dwfs/collection/IndoorBuildings/items/nc82ac00-9610-7fe8-d6b2-27da02bcb53f

```
{
  "type": "Feature",
  "id": "nc82ac00-9610-7fe8-d6b2-27da02bcb53f",
  "geometry": {
    "type": "Point",
    "coordinates": [
      127.112537967655,
      37.3941675311279,
      -21
    ]
  },
  "properties": {
    "gmlid": "nc82ac00-9610-7fe8-d6b2-27da02bcb53f",
    "modelUrl": "http://tm.gis.tw/3dwfs/static/indoorgml/Alphadom_IndoorGML-
Absolute_CRS-meter-Network_.gml"
  }
}
```

7.2. Outdoor 3D City Model and outdoor air quality sensors

7.2.1. STT GeoPortal

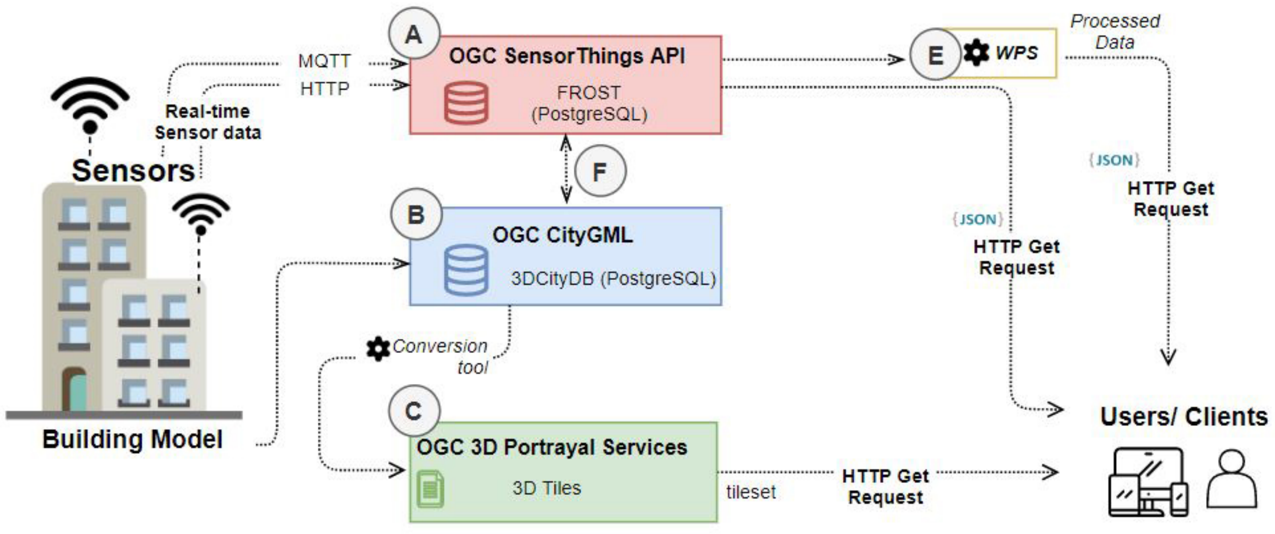
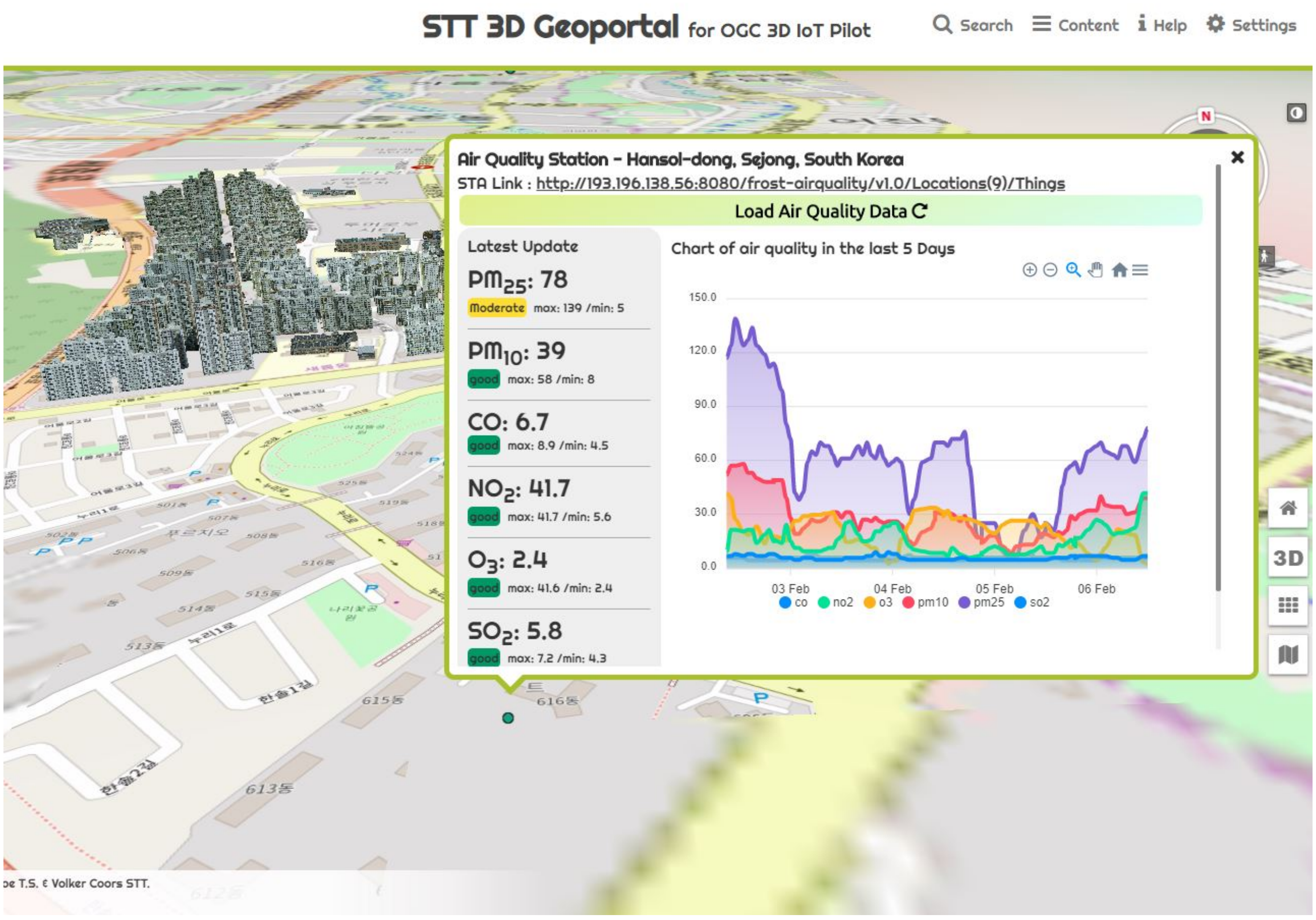


Figure 33. Overall system architecture of STT GeoPortal

7.2.1.1. The CityThings and the 3D Geoportal

The overall structure of the STT GeoPortal is shown in Figure 33 which aims to showcase the integration of air quality data from different sensor sources including the real-world and synthetic sensor data. In `part A` of Figure 33, the sensor data is collected and managed in the SensorThings API (STAPI) servers which are hosted by different parties (STT, GIS.FCU, Helyx, and SensorUP). Raw data collected by each sensor can be retrieved through these STAPI servers for presentation directly on the GeoPortal client as shown in Figure 34. Also, in `part E` of Figure 33, the WPS server is available for pre-processing the sensor data from STAPI server and visualize the data to the client as a heatmap layer as shown in Figure 35. In `part B and C` of Figure 33, the 3D city model layer is available in the CityGML format and converted to 3D Tiles format and hosted in the 3D Portrayal Services which is retrieved by the user client through the GeoPortal. As the attributes of the building are preserved in the 3D Tiles format, the CityGML and SensorThings API can be linked with the CityThings concept (`part F` of Figure 33). The GeoPortal is built based on the virtualcitySYSTEMS application which is based on the Cesium JavaScript library.


STT 3D Geoportal for OGC 3D IoT Pilot

Air Quality Station - Hansol-dong, Sejong, South Korea

STA Link : http://193.196.138.56:8080/frost-airquality/v1.0/Locations(9)/Things

Latest Update

PM_{25}: 78
Moderate max: 139 /min: 5

PM_{10}: 39
good max: 58 /min: 8

CO: 6.7
good max: 8.9 /min: 4.5

NO_2: 41.7
good max: 41.7 /min: 5.6

O_3: 2.4
good max: 41.6 /min: 2.4

SO_2: 5.8
good max: 7.2 /min: 4.3

Chart of air quality in the last 5 Days

Figure 34. STT GeoPortal with raw air quality sensor data at the sensor location

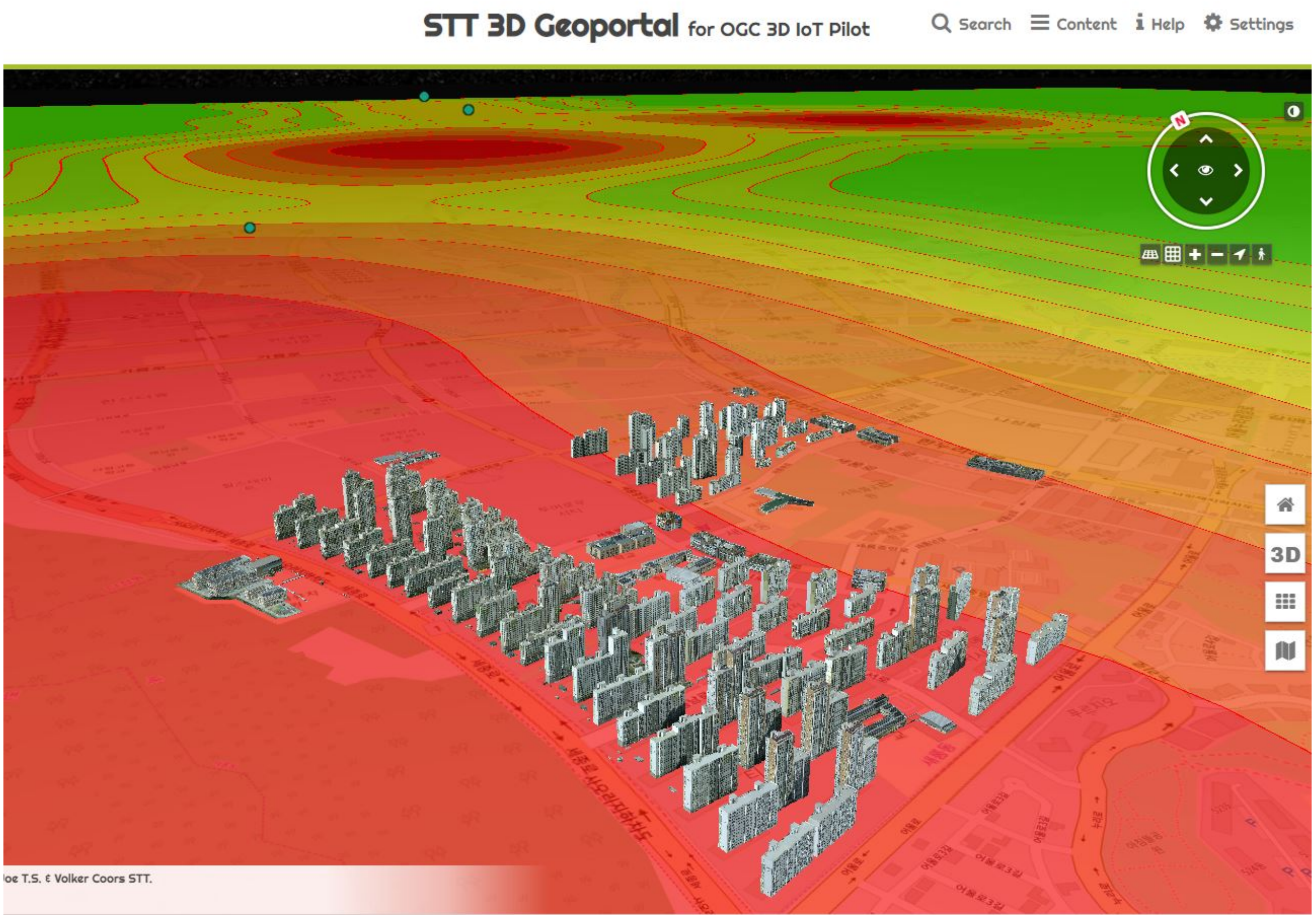

STT 3D Geoportal for OGC 3D IoT Pilot
Search
Content
Help
Settings
3D
loe T.S. & Volker Coors STT.

Figure 35. STT GeoPortal with processed sensor data from WPS services

7.2.1.2. Provide 3D Tiles building models as OGC 3D Portrayal Services

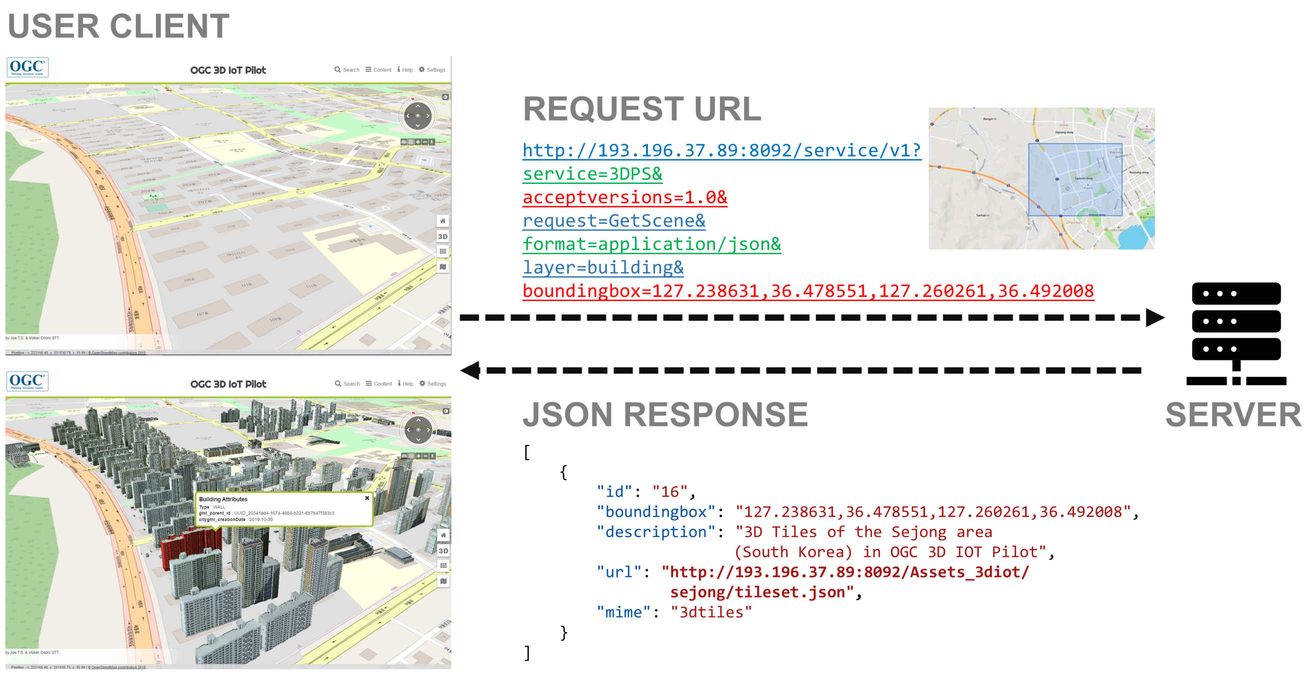
The converted building models in the 3D Tiles format are hosted on an implementation of the 3D Portrayal Service standard which is an OGC standard for 3D geospatial content delivery. Users can make a `GetScene` request to retrieve a 3D scene represented as 3D geometries. If the request is valid, users will get a document of the media type as specified in the request format parameter. The retrieved document contains a 3D scene assembled from the features of the selected layers within the specified bounding box, represented in the specified format.

Table 6. Example of the request properties in the 3D Portrayal Services

Property name	**Value**
service	`3DPS`
request	`GetScene`
layers	`building`
format	`application/json`
boundingbox	`127.238631,36.478551,127.260261,36.492008`

For example, in this pilot, user clients can retrieve and visualize the 3D building models with parameters shown in Table 2 and forms the request URL as `http://193.196.37.89:8092/service/v1?service=3DPS&acceptversions=1.0&request=GetScene&layer=building&format=application/json&boundingbox=127.238631,36.478551,127.260261,36.492008`. This request operation returns the KVP in

JSON format that contains the tileset.json file of the request 3D Tiles models hosted on the server. Then, users can use it for rendering the 3D building models on the web client. The overall steps of using 3D Portrayal Service is shown in Figure 36.



```
http://193.196.37.89:8092/service/v1?
service=3DPS&
acceptversions=1.0&
request=GetScene&
format=application/json&
layer=building&
boundingbox=127.238631,36.478551,127.260261,36.492008
```

```
[
    {
        "id": "16",
        "boundingbox": "127.238631,36.478551,127.260261,36.492008",
        "description": "3D Tiles of the Sejong area
                        (South Korea) in OGC 3D IOT Pilot",
        "url": "http://193.196.37.89:8092/Assets_3diot/
                sejong/tileset.json",
        "mime": "3dtiles"
    }
]
```

Figure 36. The 3D Portrayal Services request example for 3D building models in Sejong, South Korea

7.2.1.3. The CityThings and the WPS – Property Estimator

There are two main approaches for handling the datastreams from the WPS (Figure 24E) which are 1. Storing the processed data in the SensorThings and 2. Sending the processed data to the client directly. If the WPS process required high processing power and long processing time, the processed data should be stored in the SensorThings. In this case, a new set of Things, Sensors, FeaturesOfInterest and Datastreams entities that represents the WPS should be registered to the SensorThings to avoid confusion between the raw dataset and processed dataset. If the WPS process required low processing power and short processing time, the processed data can be transferred directly to the clients. The comparison and evaluation of these two approaches is needed to ensure better performance and scalability.

7.2.2. Web Feature Service to provide 3D building models in glTF

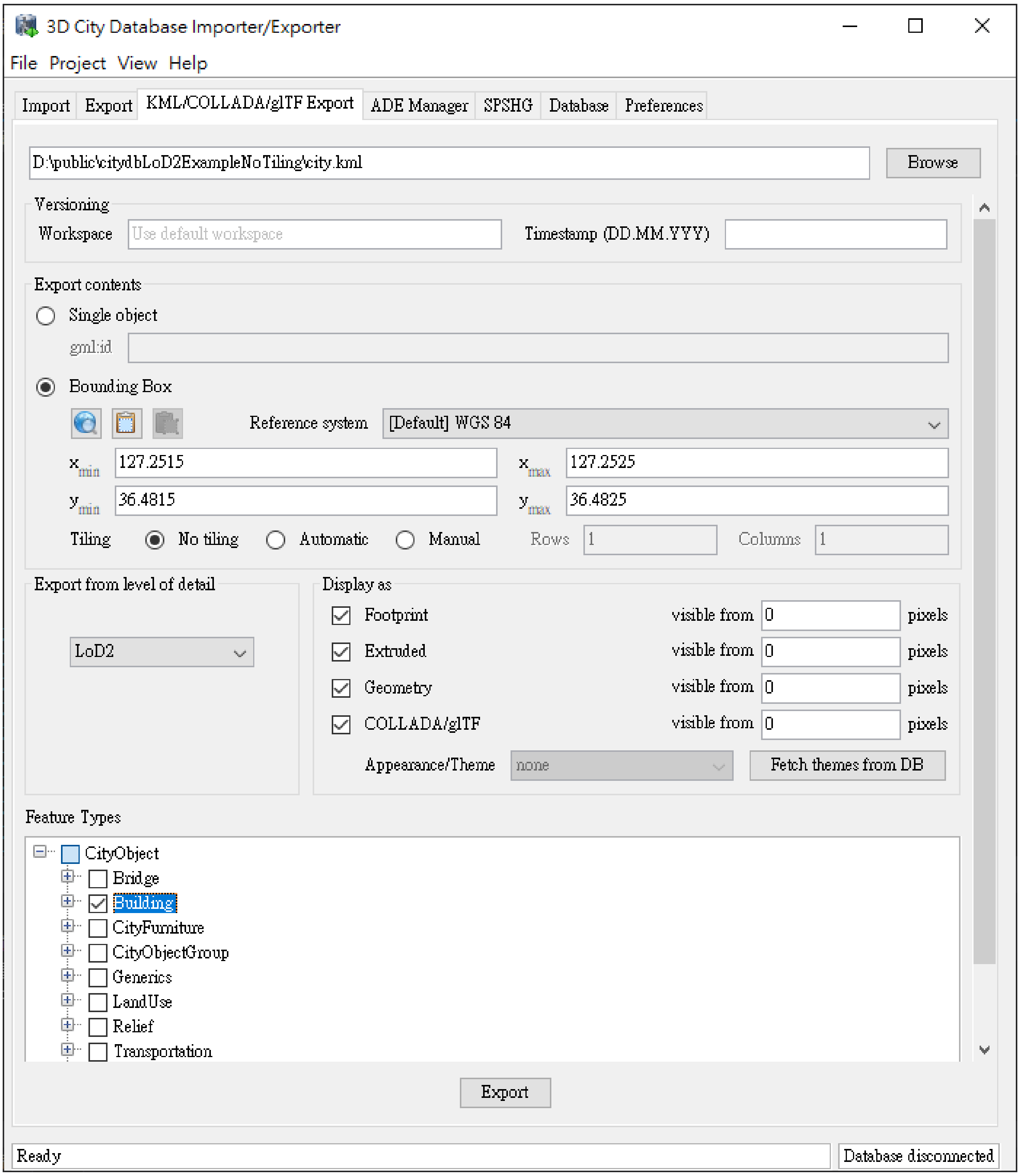
7.2.2.1. Generate glTF using 3DCityDB Importer/Exporter

In this pilot, the CityGML is imported to the 3DCityDB and the 3DCityDB is hosted on a VM machine in South Korea. For serving data encoded in the 3D model format glTF, the steps involved in generating glTF from 3DCityDB using the 3DCityDB Importer/Exporter tool are listed below:

1. Install 3DCityDB Importer/Exporter: Tested on version 4.2.0, Java 8 Runtime Environment.
2. Connect to the 3DCityDB: Select the `database` label and fill in the server connection information.
3. Select the `KML/COLLADA/glTF Export` label and set the output folder, the tool will save a KML file of the export configuration under the folder. Then we can choose to export objects in the bounding box or single objects by GMLID. The following screenshot shows the example settings

to implement the WFS.

4. Start to Export.


3D City Database Importer/Exporter
File Project View Help
Import Export KML/COLLADA/glTF Export ADE Manager SPSHG Database Preferences
Browse
Fetch themes from DB
Export

D:\public\citydbLoD2ExampleNoTiling\city.kml

Versioning
Workspace: Use default workspace
Timestamp (DD.MM.YYY):

Export contents
Single object
gml:id:
[✓ Bounding Box]
Reference system: [Default] WGS 84
x_{min} 127.2515 x_{max} 127.2525
y_{min} 36.4815 y_{max} 36.4825
Tiling: [✓ No tiling] Automatic Manual Rows 1 Columns 1

Export from level of detail: LoD2

Display as
- ✓ Footprint visible from 0 pixels
- ✓ Extruded visible from 0 pixels
- ✓ Geometry visible from 0 pixels
- ✓ COLLADA/glTF visible from 0 pixels

Appearance/Theme: none

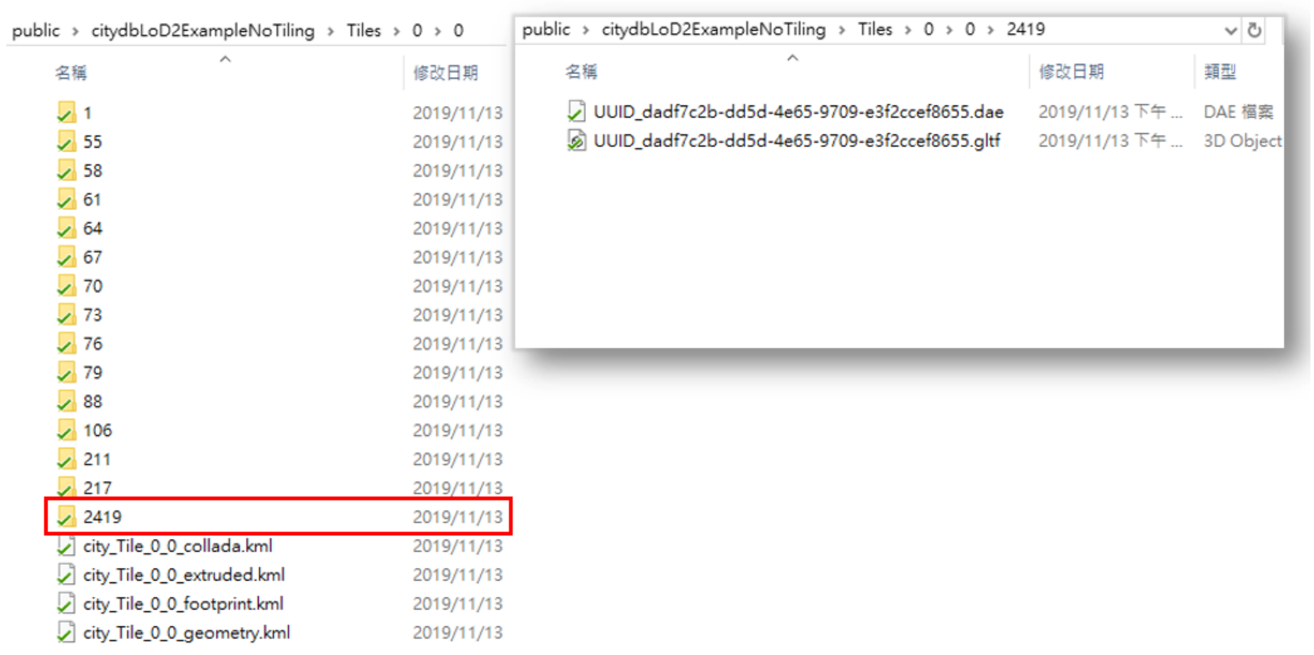
Feature Types
- CityObject
  - Bridge
  - ✓ Building
  - CityFurniture
  - CityObjectGroup
  - Generics
  - LandUse
  - Relief
  - Transportation

Ready | Database disconnected

Figure 37. Settings of exporting glTF served by GIS.FCU WFS model server.

7.2.2.2. Mapping glTF file by GML IDs

After exporting, all the glTF and DAE 3D models can be found in folders under `[ExportFolderName]\Tiles\0\0`. Each folder name is the primary key column `id` in the table `cityobject` in 3DCityDB, and each 3D model file name is the column `gmlid` in the same table. So when we query the buildings and their 3D models, select the column `id` and `gmlid`, then combine them together with the format like .gltf, as result we can get the file paths of each building's 3D model: `[ExportFolderName]\Tiles\0\0\[id]\[gmlid].gltf`.



public › citydbLoD2ExampleNoTiling › Tiles › 0 › 0

名稱	修改日期
1	2019/11/13
55	2019/11/13
58	2019/11/13
61	2019/11/13
64	2019/11/13
67	2019/11/13
70	2019/11/13
73	2019/11/13
76	2019/11/13
79	2019/11/13
88	2019/11/13
106	2019/11/13
211	2019/11/13
217	2019/11/13
2419	2019/11/13
city_Tile_0_0_collada.kml	2019/11/13
city_Tile_0_0_extruded.kml	2019/11/13
city_Tile_0_0_footprint.kml	2019/11/13
city_Tile_0_0_geometry.kml	2019/11/13

public › citydbLoD2ExampleNoTiling › Tiles › 0 › 0 › 2419

名稱	修改日期	類型
UUID_dadf7c2b-dd5d-4e65-9709-e3f2ccef8655.dae	2019/11/13 下午 ...	DAE 檔案
UUID_dadf7c2b-dd5d-4e65-9709-e3f2ccef8655.gltf	2019/11/13 下午 ...	3D Object

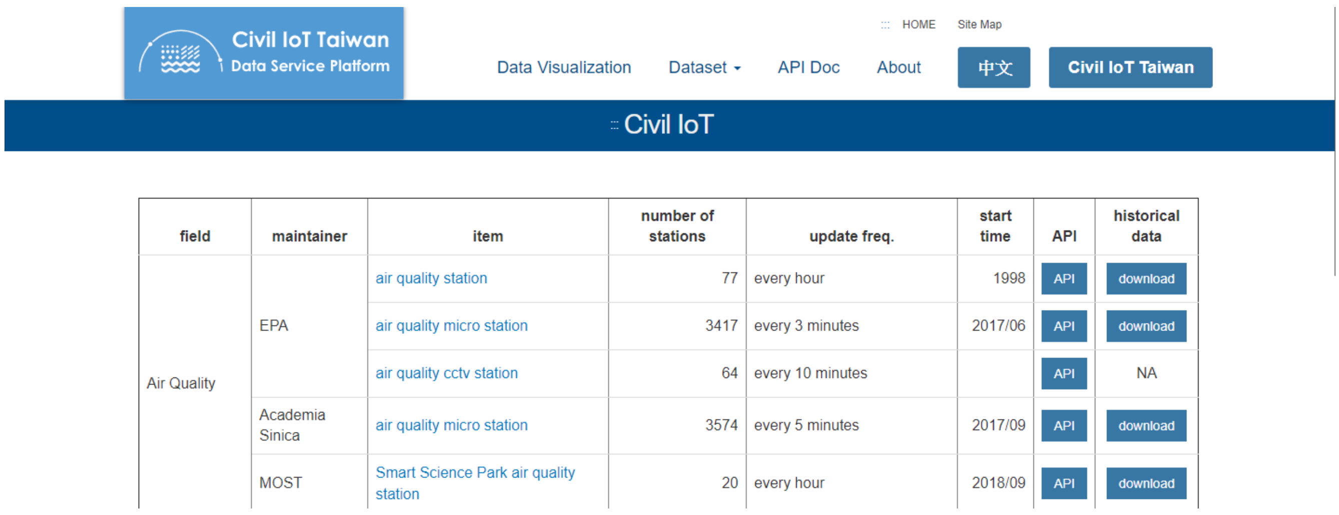
Figure 38. 3D models exported from 3DCityDB.

7.2.3. SensorThings API for air quality measurement

7.2.3.1. Connecting real world air quality Sensor Measurements

7.2.3.1.1. GIS.FCU Implementation - Air synthetic data from Civil IOT Taiwan Data Service Platform

The air synthetic data is the real $PM_{2.5}$ data from Taiwan Civil IOT Taiwan Data Service Platform, the participants collected the data from 19 stations in the air quality station dataset and imported the data into the database. The GIS.FCU SensorThings API server is developed using ASP.NET C# OData and MSSQL, the whole server implementation is focused on replacing the Taiwan location identification by Sejong City and creating a connection between the real-world air quality and CityGML dataset.



field	maintainer	item	number of stations	update freq.	start time	API	historical data
Air Quality	EPA	air quality station	77	every hour	1998	API	download
		air quality micro station	3417	every 3 minutes	2017/06	API	download
		air quality cctv station	64	every 10 minutes		API	NA
	Academia Sinica	air quality micro station	3574	every 5 minutes	2017/09	API	download
	MOST	Smart Science Park air quality station	20	every hour	2018/09	API	download

Figure 39. Taiwan Civil IOT Taiwan Data Service Platform. (Link: https://ci.taiwan.gov.tw/dsp/en/environmental_en.aspx)

The following steps describe the implementation of the GIS.FCU SensorThings API server:

1. Historical Data Import: The supplied historical data is in Comma-Separated Values (CSV) format. The data can be imported into a database like MSSQL or PostgreSQL, as both management systems offer spatial function support.
2. De-identification Sensor Data: Duplicate the sensor data from SensorThings API provided by the same dataset on Taiwan Civil IOT Taiwan Data Service Platform. Moving all coordinates of `Locations` to Sejong City.
3. SensorThings API Development: Build a SensorThings API interface using ASP.NET C# OData. The following data entities were implemented: Things, Locations, HistoricalLocations, Datastreams, Sensors, ObservedProperties, Observations and FeaturesOfInterest. Support all built-in filter operators: Comparison Operators, Logical Operators and Grouping Operators. Also support built-in query functions: String Functions, Date Functions and Math Functions.
4. Sejong City Building Connection: In order to connect with the CityGML building, property `gmlid` is put in the properties of Things, and property array `Associations` links to the related city building features.

The following example shows a sensor station that connects a building in Sejong City CityGML and the associated city building features from GIS.FCU Web Feature Service.

```
{
  "@odata.context":
"http://59.120.223.169/SensorThingsAPI/AirQuality/v1.0/$metadata#Things",
  "value": [
    {
      "@iot.id": 1,
      "@iot.selfLink":
"http://59.120.223.169/SensorThingsAPI/AirQuality/v1.0/Things(1)",
      ...
      "properties": {
        "gmlid": "UUID_dadf7c2b-dd5d-4e65-9709-e3f2ccef8655",
        "Associations": [
          {
            "Type": "Feature",
            "Link":
"https://tm.gis.tw/3dwfs/collection/SejongCityBuildings/items/UUID_dadf7c2b-dd5d-4e65-
9709-e3f2ccef8655"
          }
        ]
      },
      "Locations@odata.navigationLink":
"http://59.120.223.169/SensorThingsAPI/AirQuality/v1.0/Things(1)/Locations",
      "HistoricalLocations@odata.navigationLink":
"http://59.120.223.169/SensorThingsAPI/AirQuality/v1.0/Things(1)/HistoricalLocations",
      "Datastreams@odata.navigationLink":
"http://59.120.223.169/SensorThingsAPI/AirQuality/v1.0/Things(1)/Datastreams"
    }
  ]
}
```

7.2.3.1.2. STT Implementation - Air quality data from South Air Korea Environment Corporation

STT managed to get the air quality data from air quality sensors provided by the World Air Quality Index project (http://aqicn.org/) and collected it in the SensorThings API server which including several data types of the air quality such as PM_{10}, $PM_{2.5}$, NO_2, CO, SO_2, NO, CO_2, pressure, humidity, wind speed, wind direction, temperature and humidity. STT used the Fraunhofer Opensource SensorThings-Server (FROST: https://github.com/FraunhoferIOSB/FROST-Server) as the SensorThings server. As Sejong city is the study area, the participants selected 5 air quality stations to implement in this pilot including stations at Bugang-myeon, Sinheung-dong, Areum-dong, Hansol-dong and Osong-eup. ATT wrote the Node.js program to establish the connection from these 5 air quality stations to update to the FROST server in hourly based.

Table 7. STT SensorThings Server for air quality data

SensorThings URL	Description
http://193.196.138.56:8080/frost-airquality/v1.0/Things	`Things` Entity includes 5 sensor systems

SensorThings URL	Description
http://193.196.138.56:8080/frost-airquality/v1.0/Sensors	`Sensors` Entity includes 5 sensor information
http://193.196.138.56:8080/frost-airquality/v1.0/ObservedProperties	`ObservedProperties` Entity includes 12 data types of air quality
http://193.196.138.56:8080/frost-airquality/v1.0/Datastreams	`Datastreams` Entity includes 120 datastreams (5 sensor system x 12 data types)
http://193.196.138.56:8080/frost-airquality/v1.0/Observations	`Observations` Entity collects the sensor data results in hourly based
http://193.196.138.56:8080/frost-airquality/v1.0/FeaturesOfInterest	`FeaturesOfInterest` Entity of each sensor systems (Thing)
http://193.196.138.56:8080/frost-airquality/v1.0/Locations	`Location` Entity for storing current location of each sensor systems (Thing)
http://193.196.138.56:8080/frost-airquality/v1.0/HistoricalLocations	`HistoricalLocations` Entity for storing historical location when the sensor systems are moved.

The information of each air quality station is collected in the `Things` Entity including name, description, properties and links to other associated entity in SensorThings server. If the air quality station is connected to any available CityGML, the connection of each `Things` to `CityGML` will be included in the `properties` Key pair value as shown in the following `json` response.

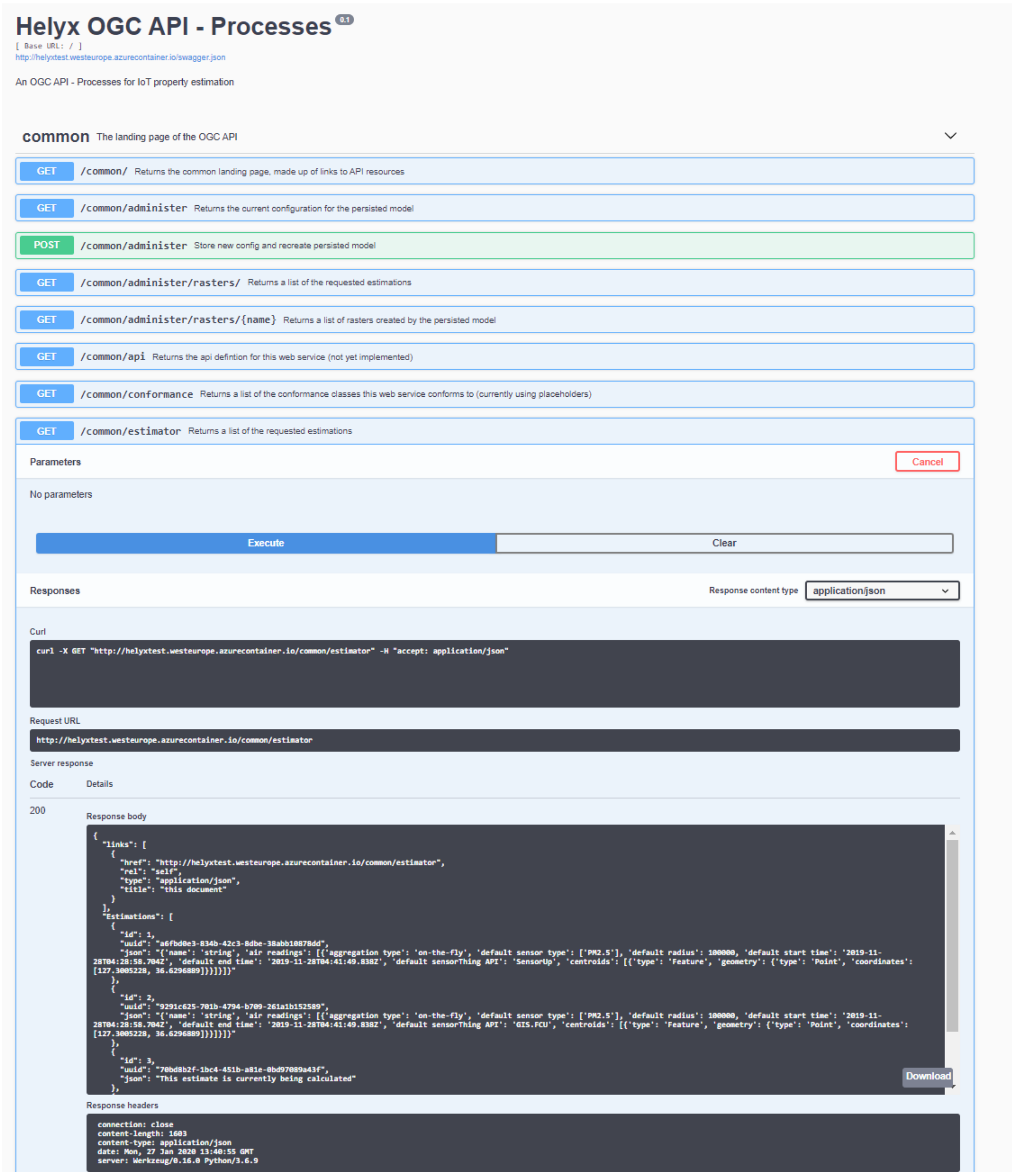
```
{
      "name": "Air Quality Station - Bugang-myeon, Sejong, South Korea",
      "description": "Air Quality data from https://aqicn.org/json-api/doc/",
      "properties": {
        "gmlid": "<gml_id here>",
        "Associations": [
          {
            "Type": "Feature",
            "Link": "<wfs_url here>"
          }
        ]
      },
      "Datastreams@iot.navigationLink": "http://193.196.138.56:8080/frost-
airquality/v1.0/Things(1)/Datastreams",
      "Locations@iot.navigationLink": "http://193.196.138.56:8080/frost-
airquality/v1.0/Things(1)/Locations",
      "HistoricalLocations@iot.navigationLink": "http://193.196.138.56:8080/frost-
airquality/v1.0/Things(1)/HistoricalLocations",
      "@iot.id": 1,
      "@iot.selfLink": "http://193.196.138.56:8080/frost-airquality/v1.0/Things(1)"
}
```

7.2.4. Helyx Implementation regarding out door air quality sensors

The Helyx Processes API is written in Python 3.6 using the flask-restplus framework and adheres to the OpenAPI specification 3.0. Details can be found in the Appendix API Specification

The API has been deployed to the following URL:

http://helyxtest.westeurope.azurecontainer.io/



Helyx OGC API - Processes 0.1

[Base URL: /]
http://helyxtest.westeurope.azurecontainer.io/swagger.json

An OGC API - Processes for IoT property estimation

common The landing page of the OGC API

GET /common/ Returns the common landing page, made up of links to API resources

GET /common/administer Returns the current configuration for the persisted model

POST /common/administer Store new config and recreate persisted model

GET /common/administer/rasters/ Returns a list of the requested estimations

GET /common/administer/rasters/{name} Returns a list of rasters created by the persisted model

GET /common/api Returns the api defintion for this web service (not yet implemented)

GET /common/conformance Returns a list of the conformance classes this web service conforms to (currently using placeholders)

GET /common/estimator Returns a list of the requested estimations

Parameters

No parameters

Responses — Response content type: application/json

Curl

```
curl -X GET "http://helyxtest.westeurope.azurecontainer.io/common/estimator" -H "accept: application/json"
```

Request URL

```
http://helyxtest.westeurope.azurecontainer.io/common/estimator
```

Server response

Code: 200

Response body

```
{
  "links": [
    {
      "href": "http://helyxtest.westeurope.azurecontainer.io/common/estimator",
      "rel": "self",
      "type": "application/json",
      "title": "this document"
    }
  ],
  "Estimations": [
    {
      "id": 1,
      "uuid": "a6fbd0e3-834b-42c3-8dbe-38abb10878dd",
      "json": "{'name': 'string', 'air readings': [{'aggregation type': 'on-the-fly', 'default sensor type': ['PM2.5'], 'default radius': 100000, 'default start time': '2019-11-
28T04:28:58.704Z', 'default end time': '2019-11-28T04:41:49.838Z', 'default sensorThing API': 'SensorUp', 'centroids': [{'type': 'Feature', 'geometry': {'type': 'Point', 'coordinates':
[127.3005228, 36.6296889]}}]}]}"
    },
    {
      "id": 2,
      "uuid": "9291c625-701b-4794-b709-261a1b152589",
      "json": "{'name': 'string', 'air readings': [{'aggregation type': 'on-the-fly', 'default sensor type': ['PM2.5'], 'default radius': 100000, 'default start time': '2019-11-
28T04:28:58.704Z', 'default end time': '2019-11-28T04:41:49.838Z', 'default sensorThing API': 'GIS.FCU', 'centroids': [{'type': 'Feature', 'geometry': {'type': 'Point', 'coordinates':
[127.3005228, 36.6296889]}}]}]}"
    },
    {
      "id": 3,
      "uuid": "70bd8b2f-1bc4-451b-a81e-0bd97089a43f",
      "json": "This estimate is currently being calculated"
    },
```

Response headers

```
connection: close
content-length: 1603
content-type: application/json
date: Mon, 27 Jan 2020 13:40:55 GMT
server: Werkzeug/0.16.0 Python/3.6.9
```

Figure 40. Screenshot of Helyx Processes API

As well as the standard endpoints for OGC API - Common, the API supports two primary endpoints "estimator" and "administer".

The "estimator" endpoint exposes the estimation request functionality, for both air quality and occupancy. This endpoint supports:

- GET on /estimator: returns a list of previously requested estimates.
- GET on /estimator/{uuid}: returns a previously requested estimate, designated by the path parameter uuid.
- POST on /estimator: A POST request with a JSON body to this endpoint for requesting an air quality or occupancy estimate for a specific point or feature. The URL in the "Location" Header of the POST response sends the user to the estimation resource created for this request at /estimation/{uuid}.

The "administer" endpoint exposes the functionality for managing the air quality model being created by the server. This endpoint supports:

- GET on /administer: returns the configuration currently being used by the model.
- POST on /administer: A POST request with a JSON body to this endpoint sets the configuration for the model, and restarts the model.

The "administer/rasters" endpoint exposes the resources created by the air quality model. This endpoint supports:

- GET on /rasters: returns a list of Inverse Distance Weighting raster datasets created by the model, whose names identify which date and height the raster datasets pertain to.
- GET on /rasters/{name}: returns a download of a specific Inverse Distance Weighting raster.

The POST request on the "/estimator" endpoint controls the primary functionality for the API. These POST requests run the estimation process. In the case of air quality estimates the server interacts with the outdoor air quality sensors on the STAPIs to provide either derived point-based estimations or links to interpolated raster datasets from the persisted model. If the user has requested an estimate for a GML Feature rather than a coordinate point the server will derive the centroid to use for the estimation from the GML feature on the OGC-Collections API.

In the case of occupancy estimates the server queries the OGC-Collections API by the chosen GMLID to find all the nested elements inside that chosen GMLID. These are then checked for sensors in the STAPIs. If sensors exist for these the nested elements readings are retrieved and an average is taken across sensors for the chosen time frame, to provide an average occupancy for the chosen GMLID.

7.2.4.1. Challenges

The challenges encountered whilst implementing the Estimator are as follows:

- The link between occupancy sensors, features and elements: The cross-API challenge of matching sensors to features and elements was necessary in order to calculate occupancy values for parts of GML features. Discussions were had as to whether sensor IDs stored in the GML or GMLIDs should be stored in the Things or FeatureOfInterest resources of the STAPIs. As the GML features in the Collections API describe static features, whereas sensors may move or change it was decided it would be inappropriate to store sensor identifiers in the GML Features.

As the sensors in the pilot were stationary and there was a one-to-one relationship between the sensors and the objects they described, the decision between storing the object IDs on the Thing resource of the FeatureOfInterest was easy, as they are the same thing. Therefore, the GMLID of elements in the Collections API were stored in the corresponding Thing resource in the STAPI service. However, A more complex approach could have been taken to group Thing sensors under FeatureOfInterest to allow for faster identification of parent features, ideal for calculating derived aggregate estimates. In such a case the FeatureOfInterest may contain the ID of a building floor in a GML Feature. The Things may contain the ID of a room or volume in a GML Feature. In such a case, a filter on FeatureOfInterest would return all of the Things for a single building floor, at which point each Thing can be inspected for their reading about a specific room on the floor. This could be described as "Cascading IDs", where the FeatureOfInterest describes the parent element and the Things describe child elements.

- The time frame of sensor readings: The sensor readings across the STAPIs were not standardized in time, meaning one sensor may output a reading every 4 minutes and another might output a reading every 1 minute. This means certain sensors have a large effect on time range averages than others. This was accepted as a limitation of the pilot.

7.2.4.2. Further development

There are two major improvements that could be made to this OGC API-Processes Estimator

- Voxels: The persisted air quality model should use voxels instead of raster datasets to represent bounding volumes for recording air quality. This would be far more appropriate for a 3D environment. The only reason this was not implemented was due to time constraints.
- Multiple STAPIs: Although the API allows uses to request estimates from a range of STAPIs the estimates are only calculated against the chosen STAPI. This was partially due to practicality of conducting technology integration experiments against each STAPI in turn to test varying functionality. The functionality could be extended to allow estimates to be calculated across multiple STAPIs at once. In the case of the persisted air quality model, the server would have to run an MQTT client connection for each STAPI simultaneously in order to account for updates to any STAPI in the model. In the pilot’s case 4, MQTT client connection would have to be maintained.

7.3. Indoor Building Model and indoor sensors

7.3.1. Gaia3D Implementation of the GeoPortal(only occupancy reading)

The overall structure of the implementation is shown in Figure 41.

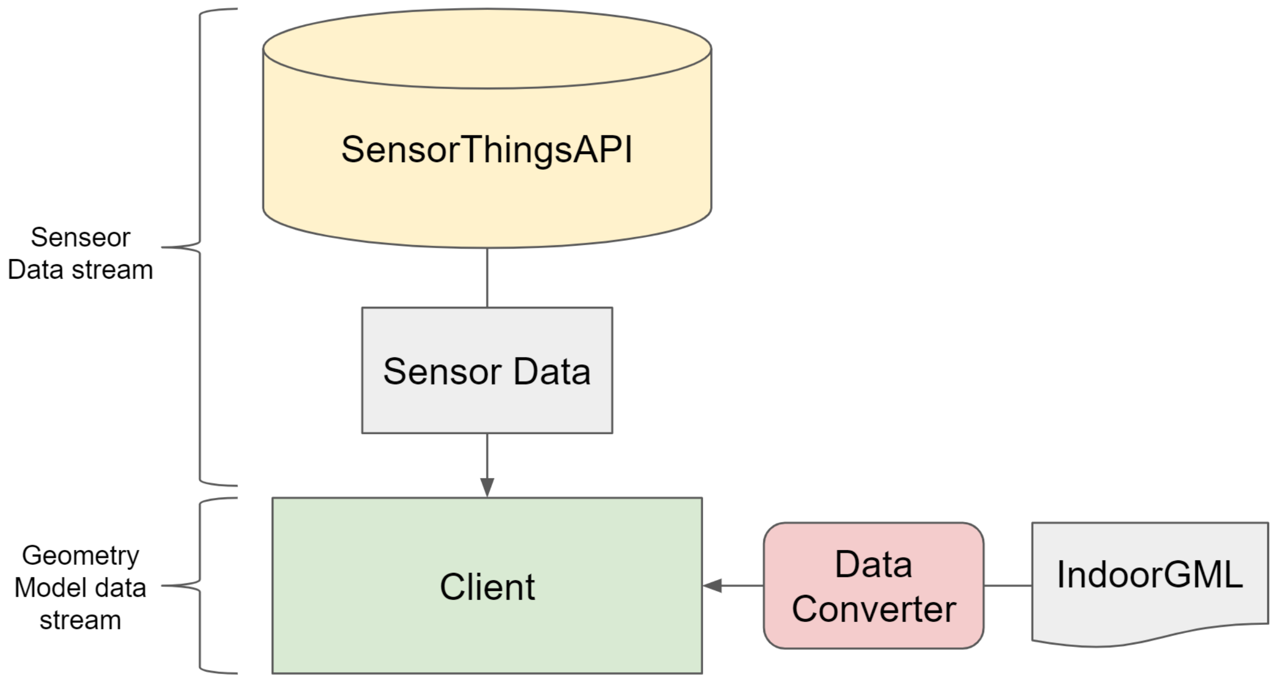


Figure 41. Process structure of the GeoPortal

7.3.1.1. Sensor Datastream

It is expected that an occupancy sensor has real location. Also, for visualization, the location of the sensor is needed. After expanding the data of Things and Datastream resources, the Datastream is filtered with ObservedProperty 'occupancy'.

```
Locations -> Things -> Datastream -> Observations
                            ^
                            |
                       ObservedProperty : occupancy
```

Per every 10 sec window, the client sends a request to the server and then refreshes the value of the indoor occupancy data.

ex) https://ogc-3d-iot-pilot.sensorup.com/v1.0/Locations?$expand=Things/Datastreams/Observations&$filter=Things/Datastream/ObservedProperty/name%20eq%20%27occupancy%27

```
...
{
    @iot.id: 21778,
    @iot.selfLink: "https://ogc-3d-iot-pilot.sensorup.com/v1.0/Locations(21778)",
    description: "Telus",
    name: "marketmall_cell_12",
    encodingType: "application/vnd.geo+json",
    location: {
        coordinates: [
            -114.15561497211456,
            51.08518410671183,
        ],
        type: "Point",
    },
    Things: [
        {
            @iot.id: 21779,
            @iot.selfLink: "https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(21779)",
            description: "Telus",
            name: "marketmall_cell_12",
            properties: { },
            Datastreams@iot.navigationLink:
            "https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(21779)/Datastreams",
            HistoricalLocations@iot.navigationLink:
            "https://ogc-3d-iot-
pilot.sensorup.com/v1.0/Things(21779)/HistoricalLocations",
            Locations@iot.navigationLink:
            "https://ogc-3d-iot-pilot.sensorup.com/v1.0/Things(21779)/Locations",
            Datastreams: [
                {
                    @iot.id: 21782,
                    @iot.selfLink: "https://ogc-3d-iot-
pilot.sensorup.com/v1.0/Datastreams(21782)",
                    description: "*",
                    name: "marketmall_cell_12_occupancy",
                    observationType:
                    "http://www.opengis.net/def/observationType/OGC-
OM/2.0/OM_CountObservation",
                        unitOfMeasurement: {
                            symbol: "*",
                            name: "*",
                            definition: "*",
                        },
                        Observations@iot.nextLink:
                        "https://ogc-3d-iot-
pilot.sensorup.com/v1.0/Datastreams(21782)/Observations?$top=100&$skip=100",
                    Observations: [
                    {
                        @iot.id: 27946,
                        @iot.selfLink:
                        "https://ogc-3d-iot-
pilot.sensorup.com/v1.0/Observations(27946)",
                        phenomenonTime: "2019-11-13T23:02:38.138Z",
                        resultTime: null,
                        result: 1,
                        Datastream@iot.navigationLink:
                        "https://ogc-3d-iot-
pilot.sensorup.com/v1.0/Observations(27946)/Datastream",
                        FeatureOfInterest@iot.navigationLink:
                        "https://ogc-3d-iot-
pilot.sensorup.com/v1.0/Observations(27946)/FeatureOfInterest",
                    },
                    ...
                    ]
                }
            }
        ],
        HistoricalLocations@iot.navigationLink:
        "https://ogc-3d-iot-pilot.sensorup.com/v1.0/Locations(21778)/HistoricalLocations",
}
...
```

This json is the part of the response body from SensorThingsAPI.

7.3.1.2. Geometry Model Data stream

In the case of Gaia3D, the local IndoorGML data is used because it is needed to be converted to a geometry model. After converting, it is visualized with WebGL library.

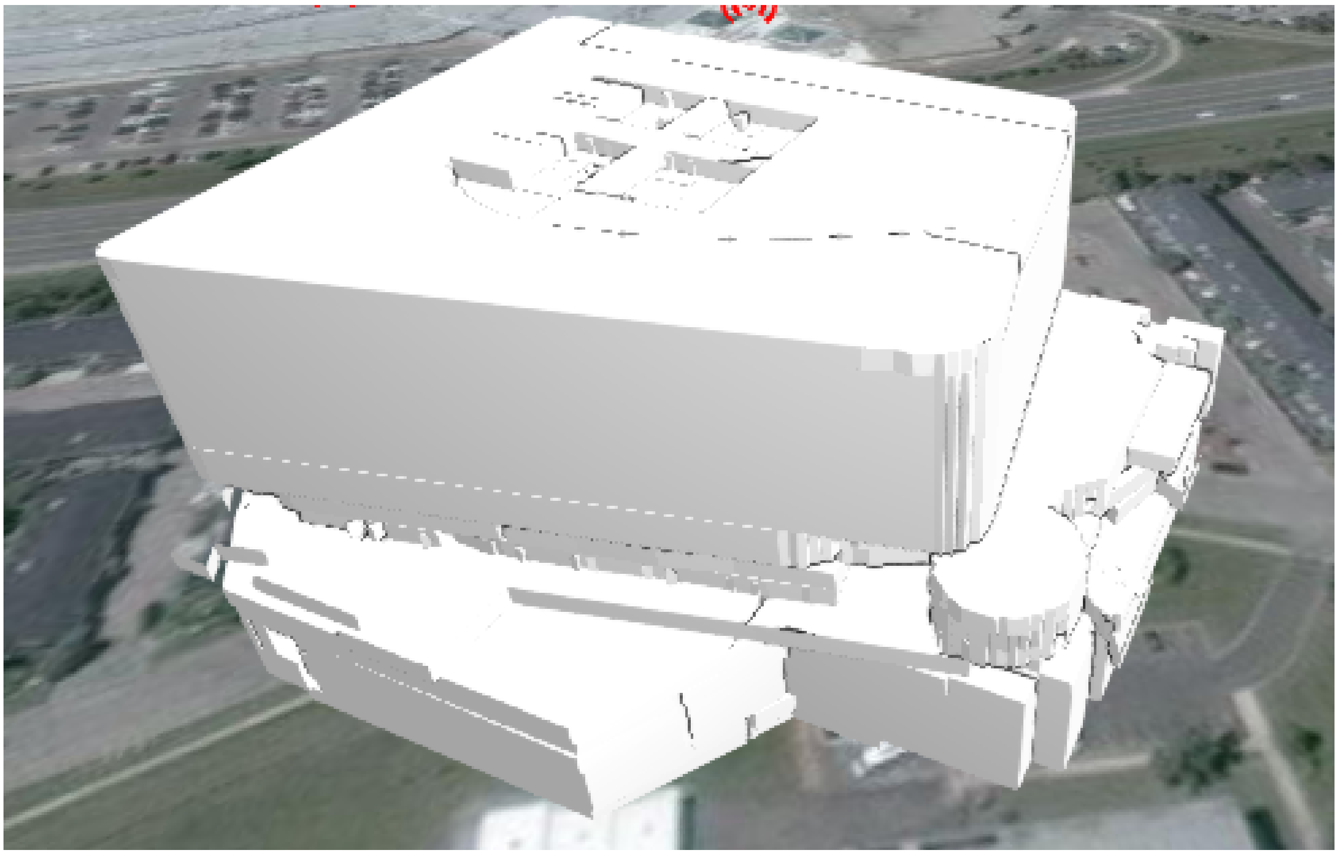
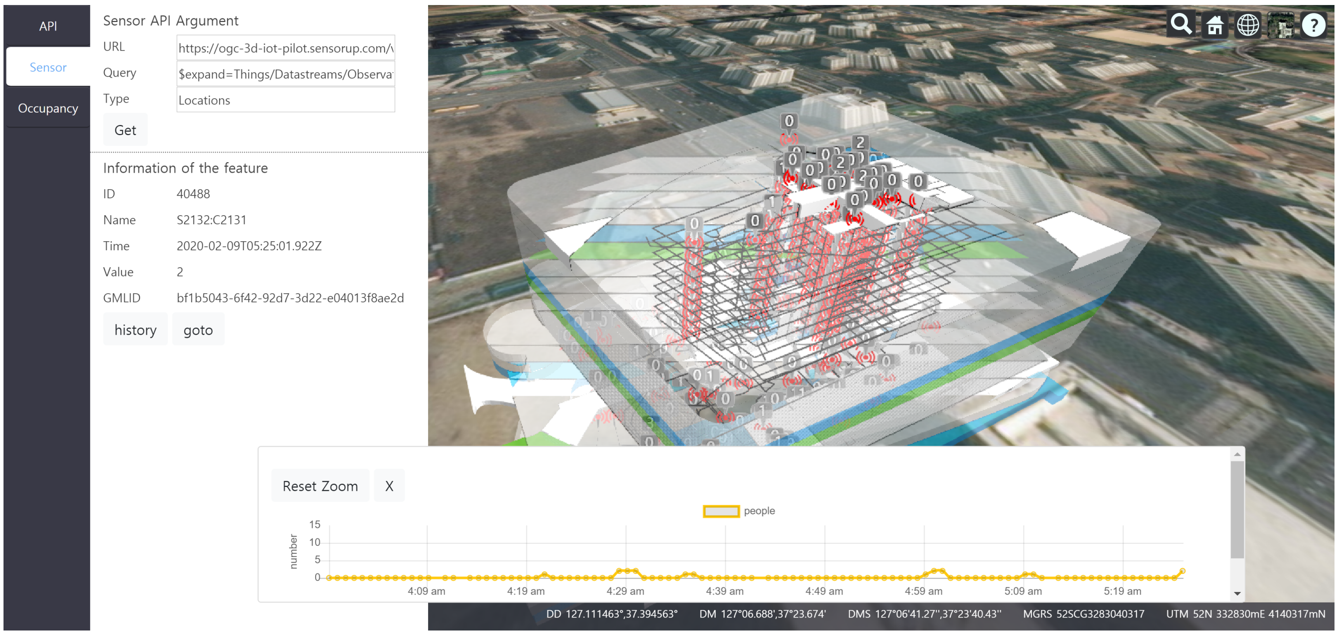


Figure 42. IndoorGML modeling(Alphadom)

7.3.1.3. Sensor Visualization

With the location value from the SensorThingsAPI server, the sensor is visualized using pin board. '1 sensor per 1 room' is assumed. The Sensor icon is attached at the bottom of the pin board for intuitive sight. The real time value of the pin board is shown at the pin board. If the user clicks the pin board, then the information of the sensor is printed at the left navigation bar and the history graph is shown at the bottom side separately. Also, through the ‘history’ button the history graph can be shown. The latest 100 values are shown on the graph.



Sensor API Argument

URL	https://ogc-3d-iot-pilot.sensorup.com/\
Query	$expand=Things/Datastreams/Observa
Type	Locations

Information of the feature

ID	40488
Name	S2132:C2131
Time	2020-02-09T05:25:01.922Z
Value	2
GMLID	bf1b5043-6f42-92d7-3d22-e04013f8ae2d

DD 127.111463°,37.394563° DM 127°06.688',37°23.674' DMS 127°06'41.27",37°23'40.43" MGRS 52SCG3283040317 UTM 52N 332830mE 4140317mN

Figure 43. Overall UI and sensor value history graph

About the IndoorGML geometry, each room is colored with the value of the occupancy sensor. 0 : white, 1-2 : blue, 3-5 : green, 6- : orange. The network of the IndoorGML is drawn with the geometry. Also, for convenient viewing, the transparency is applied at the geometry.

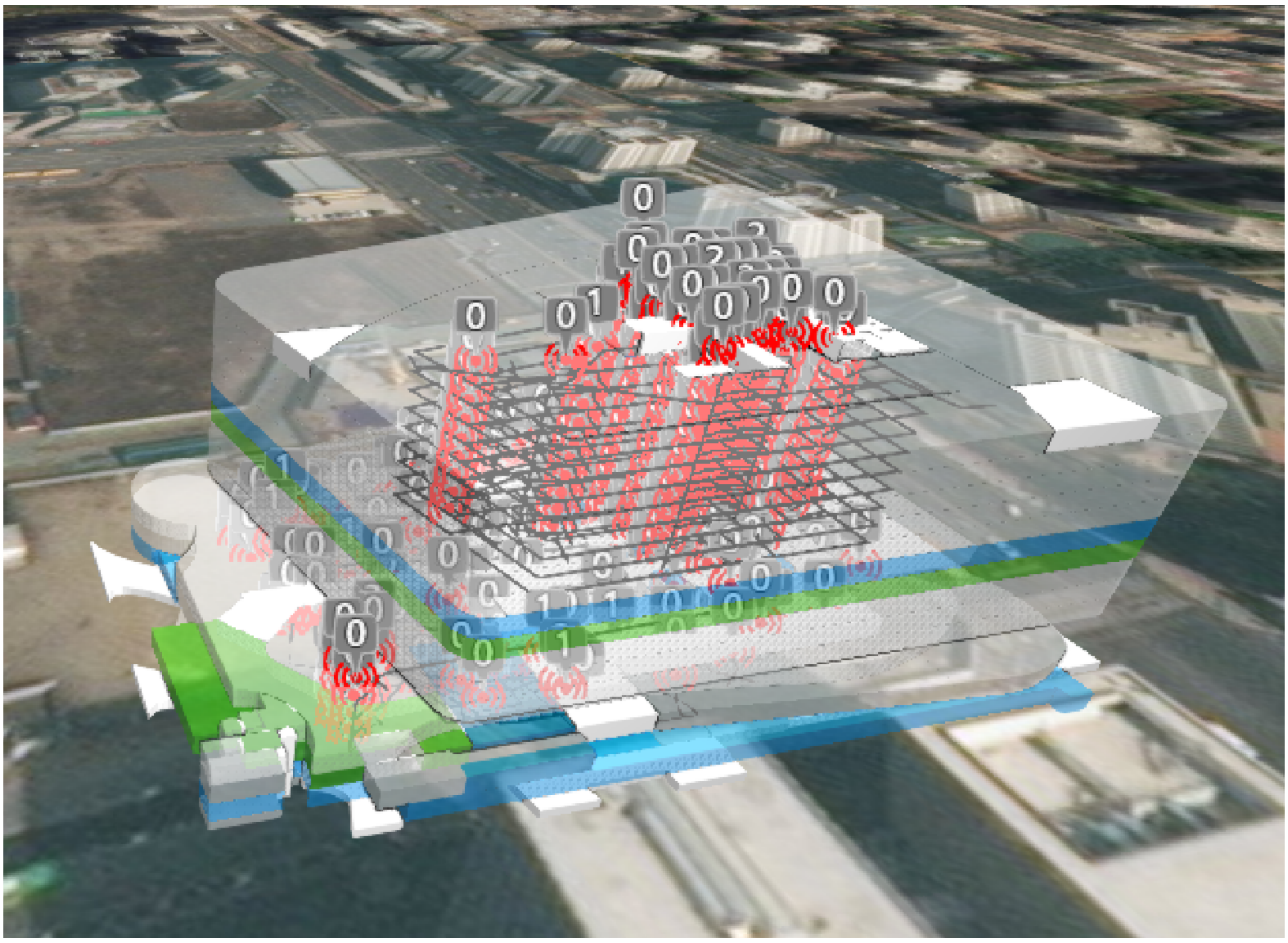


Figure 44. The implementation of the Geoportal

For the user’s convenience, the geometry model is separated by the floors.

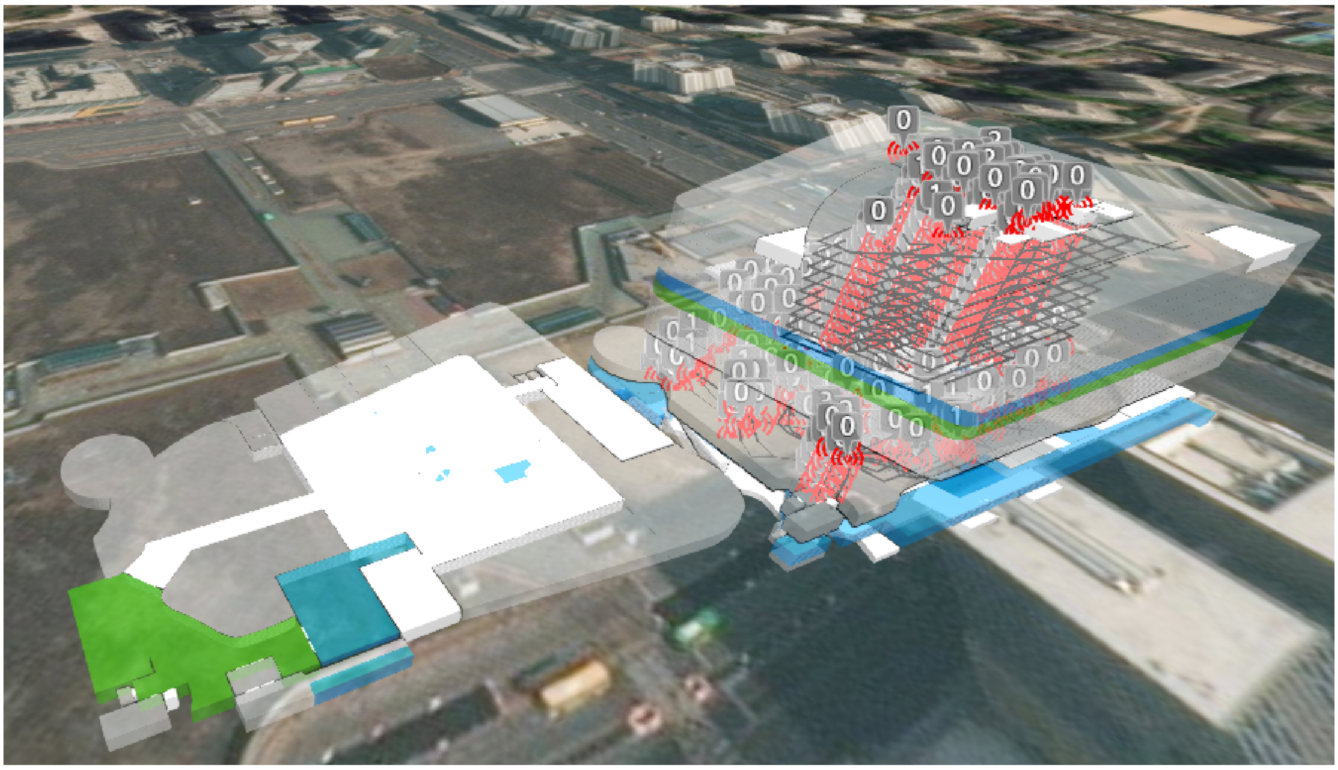


Figure 45. The floor division of IndoorGML modeling

7.3.1.4. Challenge

- Connecting with the WPS server for Indoor occupancy : Due to constraints of this pilot project, the assumption that '1 sensor per 1 room' is selected. This was because the previous assumption of 'multiple sensors per 1 room' was outside of the scope of this pilot. Furthermore, multiple sensors calculating per building, floors, etc. were also outside of the scope of this pilot. So, the visualization of the calculated value appeared to be difficult to implement.

7.4. Overview and Architecture by Cyient

The architecture of the Cyient component is shown in Figure 46.

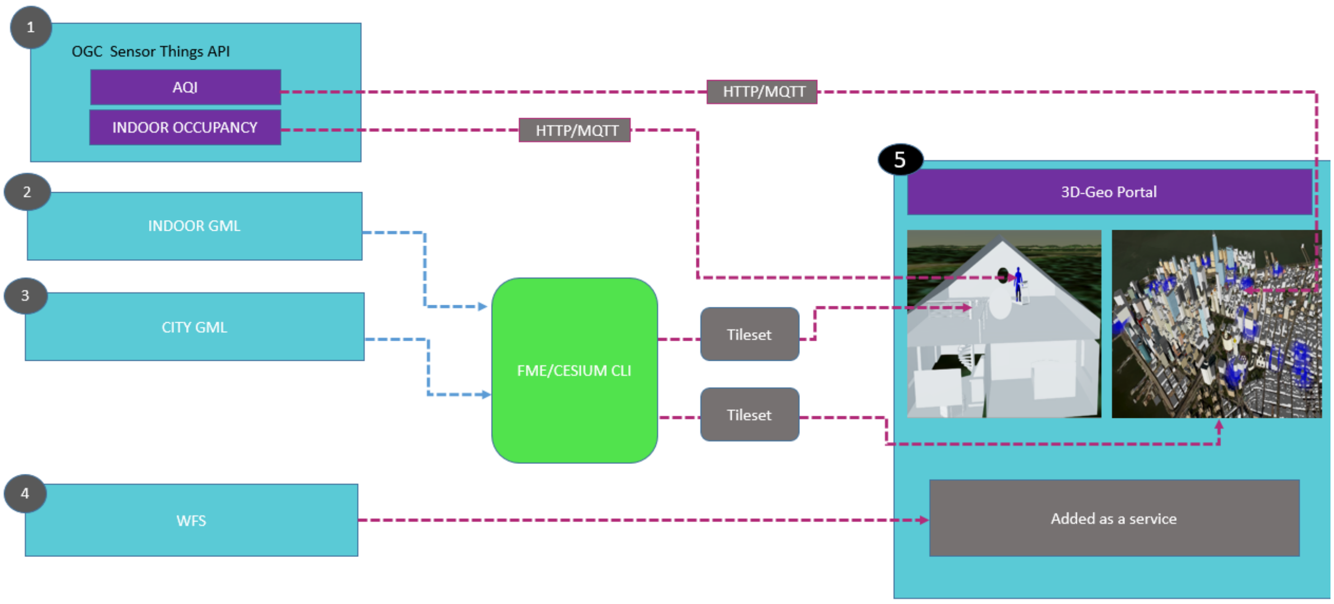


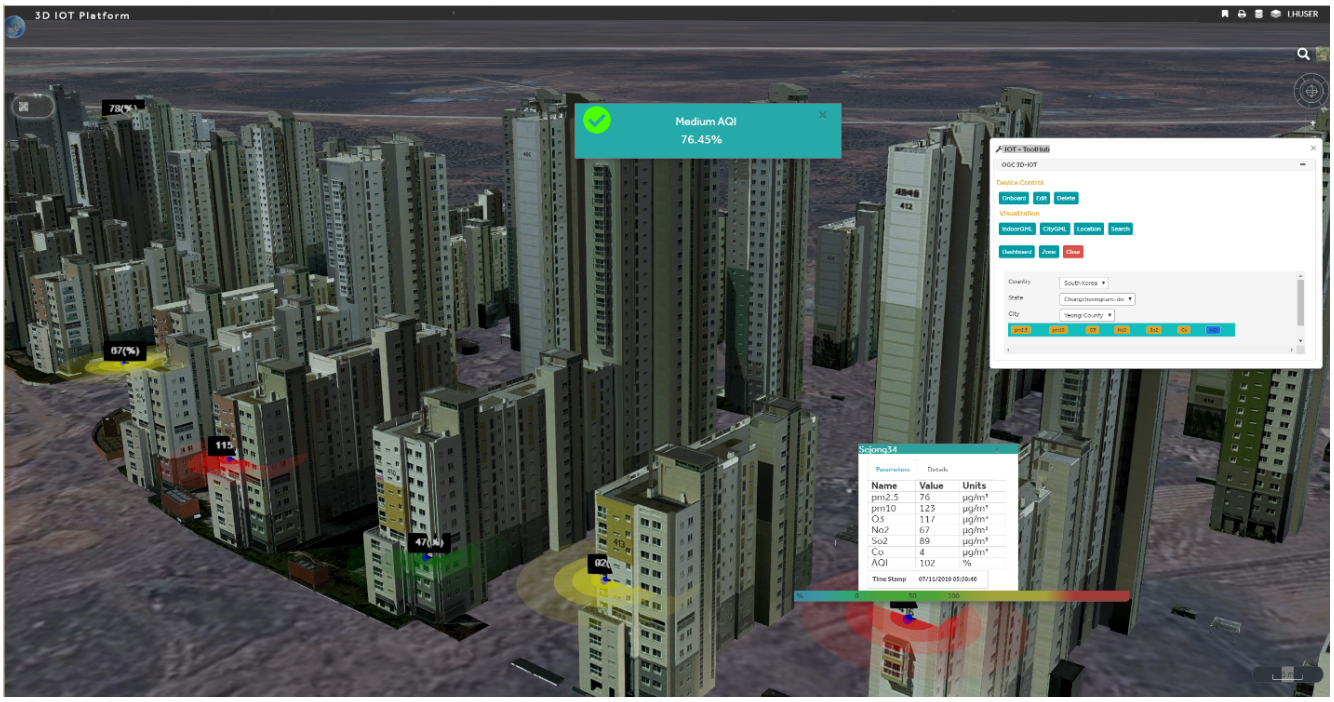
Figure 46. Architecture

The Primary motivation for addressing the topic of a 3D IOT model is based on the real-world scenario where there is concern about environmental issues. City Management need to monitor and take preparatory actions. Mainly environmental issues of air quality, water and solid waste etc. are monitored as they can lead towards severe health issues in human beings. To accelerate research and development for this model, the requirements of this initiative were to conduct the following prototyping and activities:

- The scenario is majorly dependent on 3D buildings of cities and air quality, synthetic occupancy data.
- 3D buildings, which are available in the formats that include IndoorGML and CityGML, so we need to convert these files to tile set data in a Cesium readable format using FME.
- Visualize the tile set data on Cesium, which is an open-source JavaScript library for 3D globes and maps.
- Fetch/get air quality and synthetic occupancy data from the Sensor Things API, Sensor Things API is format of OGC.
- Algorithm will read location information from the data provided by the API, and GML IDs from the building data integrate both and visualize in a 3D in a GeoPortal-appropriate manner.
- Another algorithm analyzes and classifies data, and gives proper analytics and alerts to the end user.

7.4.1. Cyient 3D GeoPortal

The 3D GeoPortal Platform allows the interaction of users with the Web-enabled GIS and provides spatial and non-spatial information display. Internet browsing software products (like Mozilla/Chrome) provide the service requesting and result visualization 3D platform to the users in client side. The 3D Platform interface includes the Cesium WebGL virtual globe, which is capable of performing the rotation, zoom, pan and fly operations. On the top right, the in-built buttons are available to geo-locate, refresh, changing the 3D mode and selecting the base layer. The web base interface is the initial page/Landing Page of the 3D Platform, which is accessed by the client through a call of HTML page on AWS cloud server.



Country	South Korea
State	Chungcheongnam-do
City	Yeongi County

Sejong34

Parameters | Details

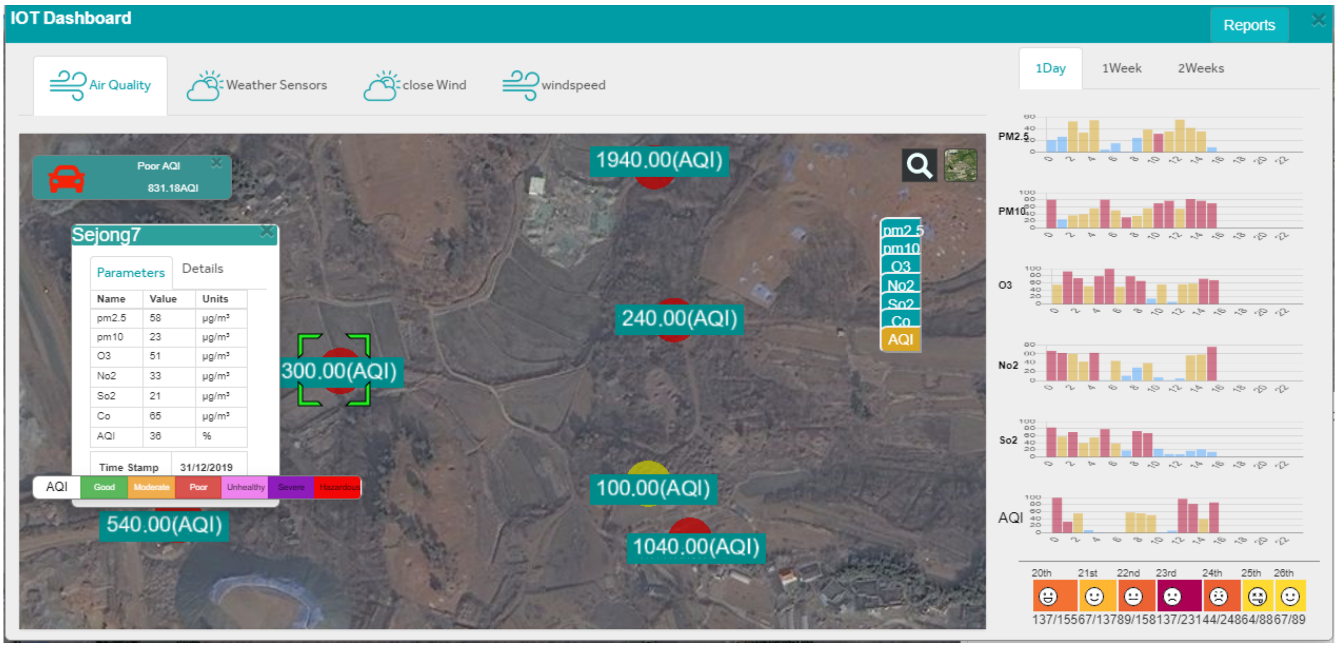
Name	Value	Units
pm2.5	76	μg/m³
pm10	123	μg/m³
O3	117	μg/m³
No2	67	μg/m³
So2	89	μg/m³
Co	4	μg/m³
AQI	102	%

Time Stamp	07/11/2019 05:59:46

Figure 47. GeoPortal With 3D Data

The integration algorithm for integrating sensors and building is based on geospatial information. The algorithm collects latitude, longitude, and altitude details from a SensorThings API implementation and places a symbol at the position. Based on the thresholds of AQI devices are color coded too. The following figure shows Air quality data on 3D buildings.

7.4.2. Air Quality Dashboard



Sejong7

Parameters | Details

Name	Value	Units
pm2.5	58	μg/m³
pm10	23	μg/m³
O3	51	μg/m³
No2	33	μg/m³
So2	21	μg/m³
Co	65	μg/m³
AQI	36	%

Time Stamp	31/12/2019

Figure 48. Dashboard

At the time of reporting, Cyient were working on IndoorGML support for the collected node and edges data and had made a connectivity graph.

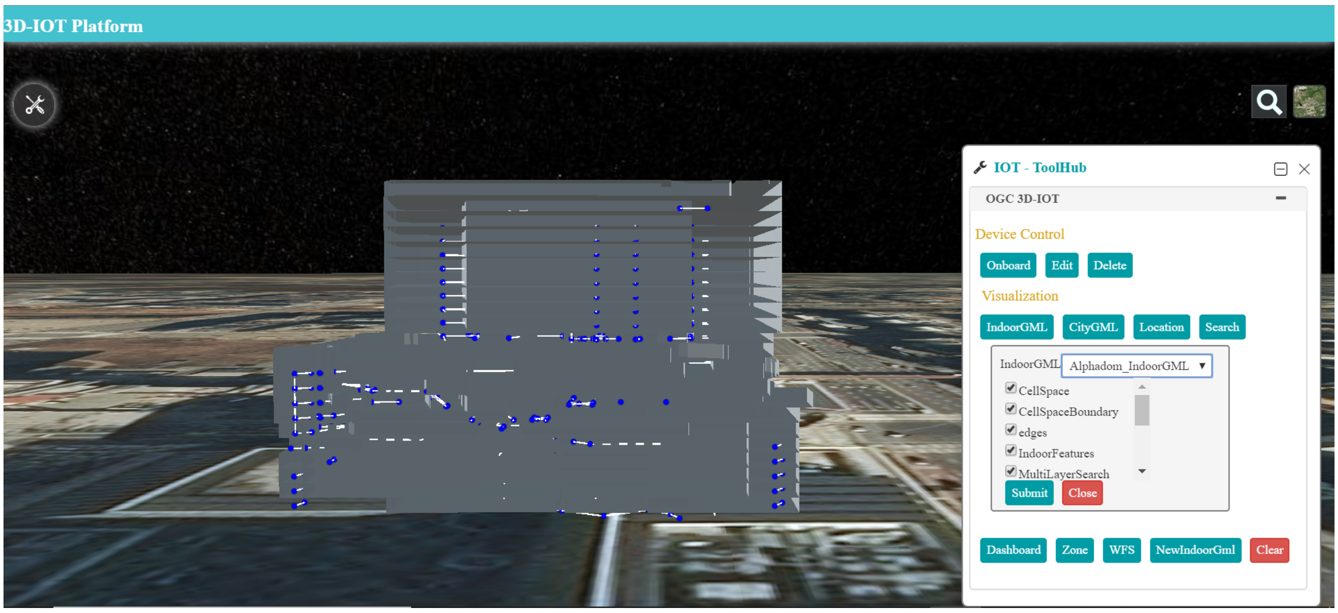


Figure 49. IndoorGML 3D

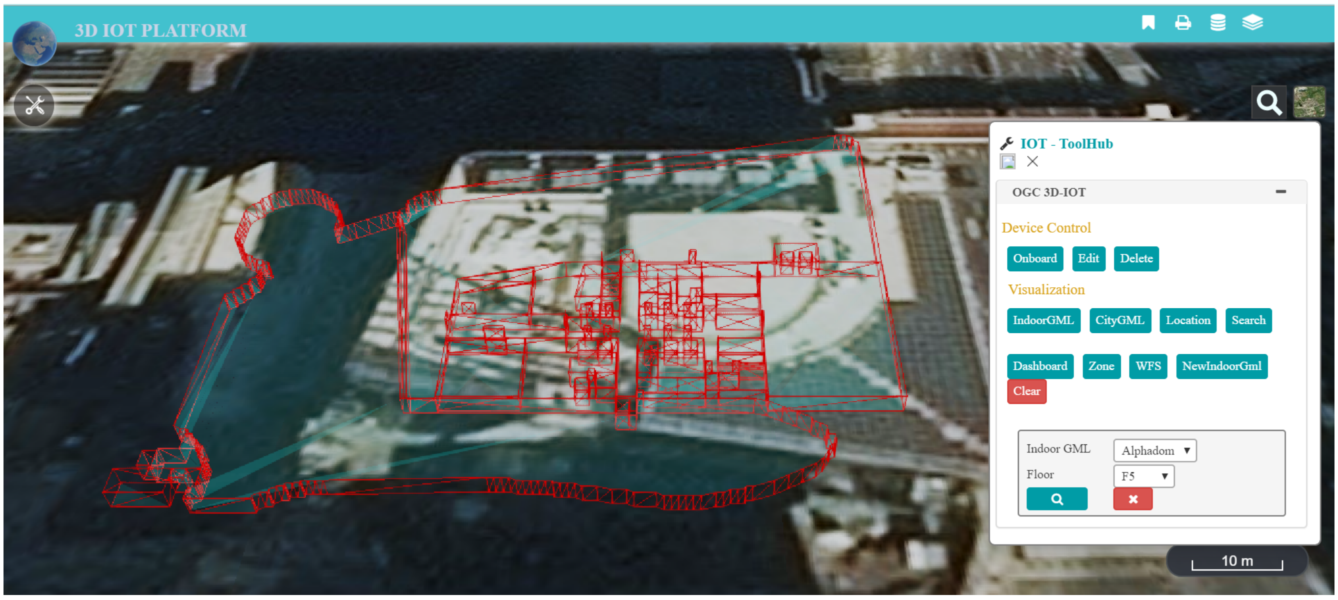


Figure 50. Floor 5 Alphadom

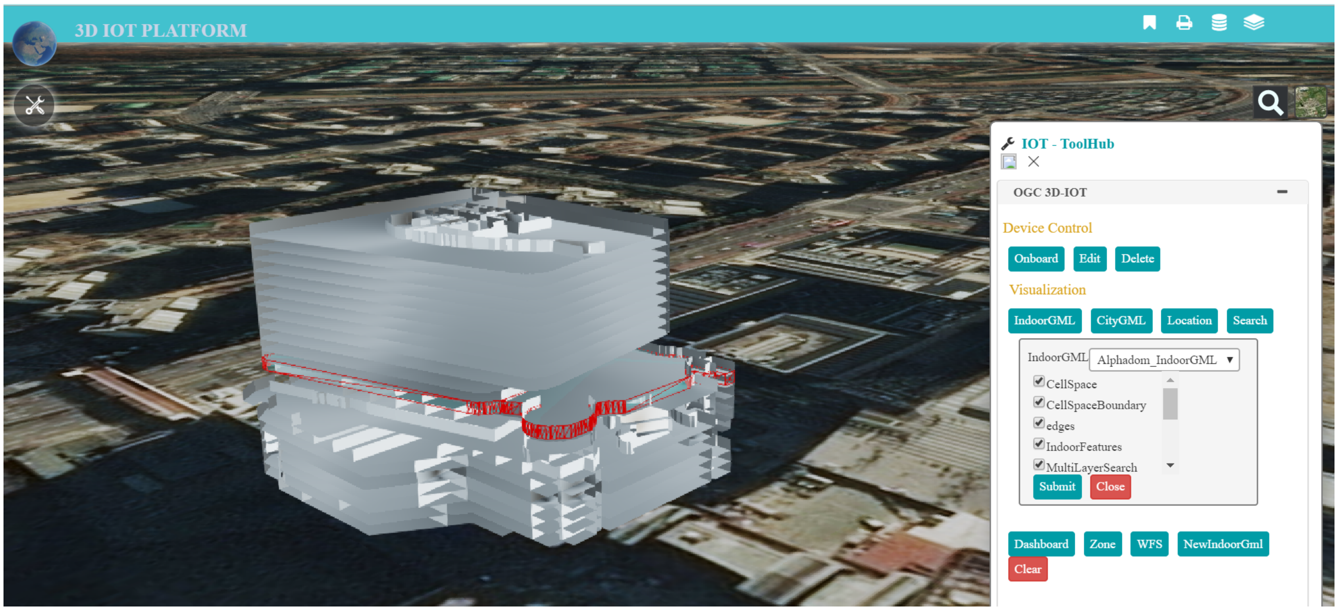


Figure 51. Floor 5 Alphadom with Tile

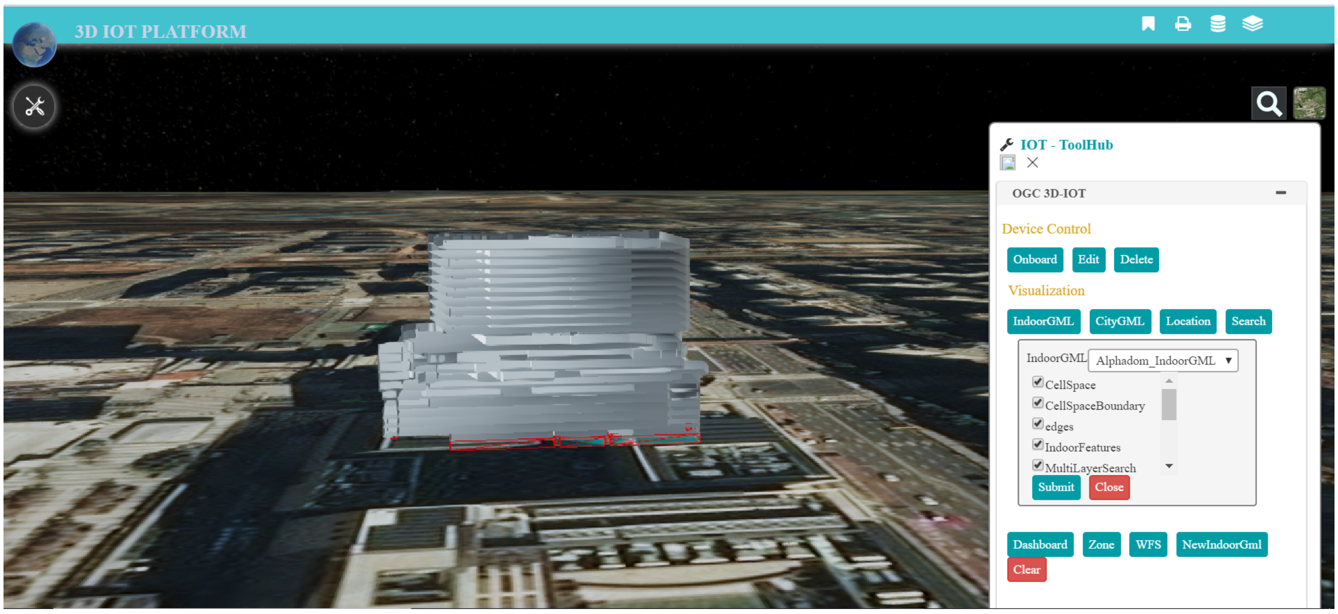


Figure 52. Floor b6 Alphadom with Tile

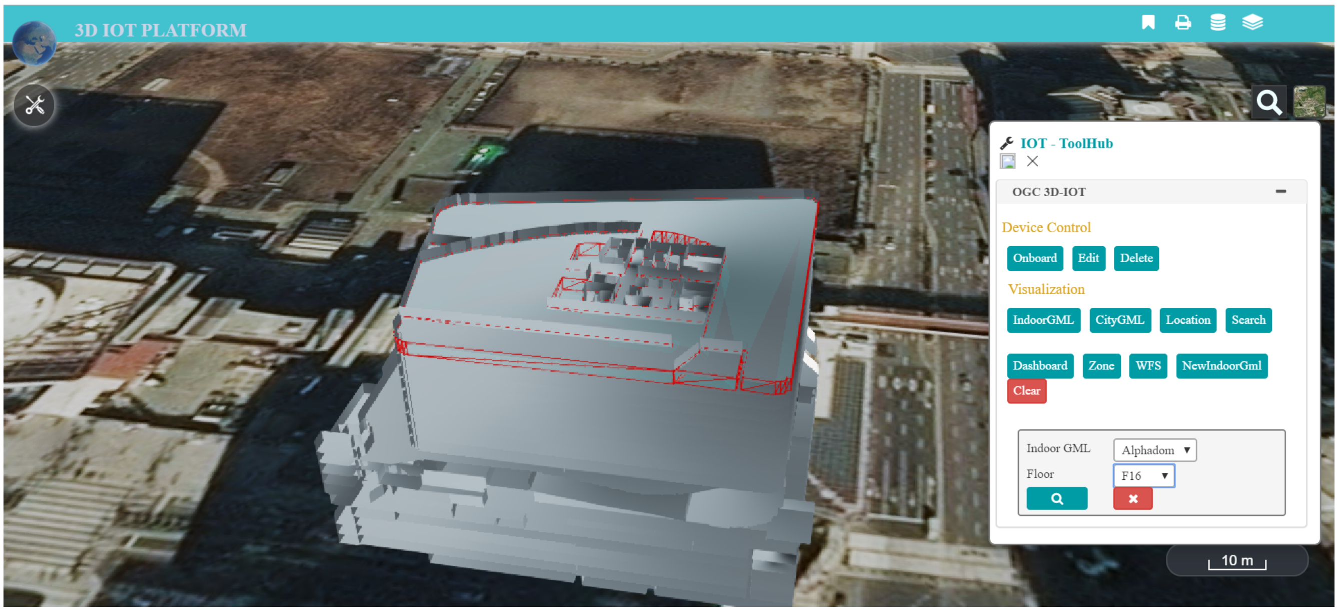


Figure 53. Floor 16 Alphadom with Tile

7.5. Skymantics 3D GeoPortal

The Skymantics GeoPortal is implemented in an Augmented Reality environment using MAIDEN, an application built using the Unity 3D engine. The Unity 3D engine is augmented using Vuforia SDK to provide the augmented reality enhancements. The concept proposed was to provide an overlay graphic depicting the sensor readings for occupancy and air quality measurements for a location in an augmented reality environment (see Figure 54).

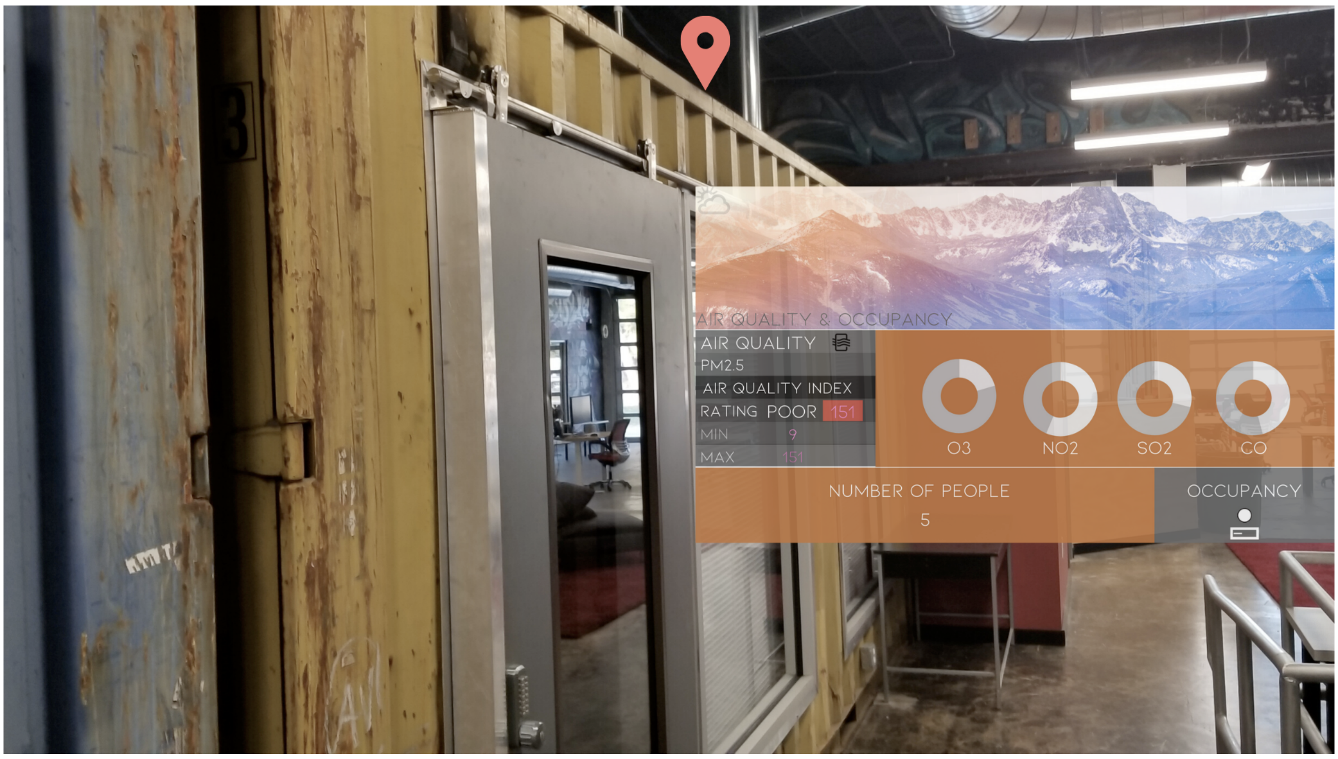


Figure 54. Mock-up of the Skymantics 3D GeoPortal for Air Quality and Occupancy

Additionally, the GeoPortal should be capable of displaying IndoorGML and CityGML for the SmartCities representation of Sejong City.

7.5.1. IndoorGML

Two datasets were provided in IndoorGML format for display within the Skymantics GeoPortal, Alphadom and Lotte World. These datasets were provided via an FTP server. Initially, a custom C# script was utilized to import the IndoorGML data and convert each feature to a game object. Game objects are the base class for displaying content within the Unity 3D engine. Some issues were encountered in the script during JSON parsing in which some of the features would not convert properly. This led to a secondary approach which was to import the STEMLab InViewer desktop plugin for Unity 3D.

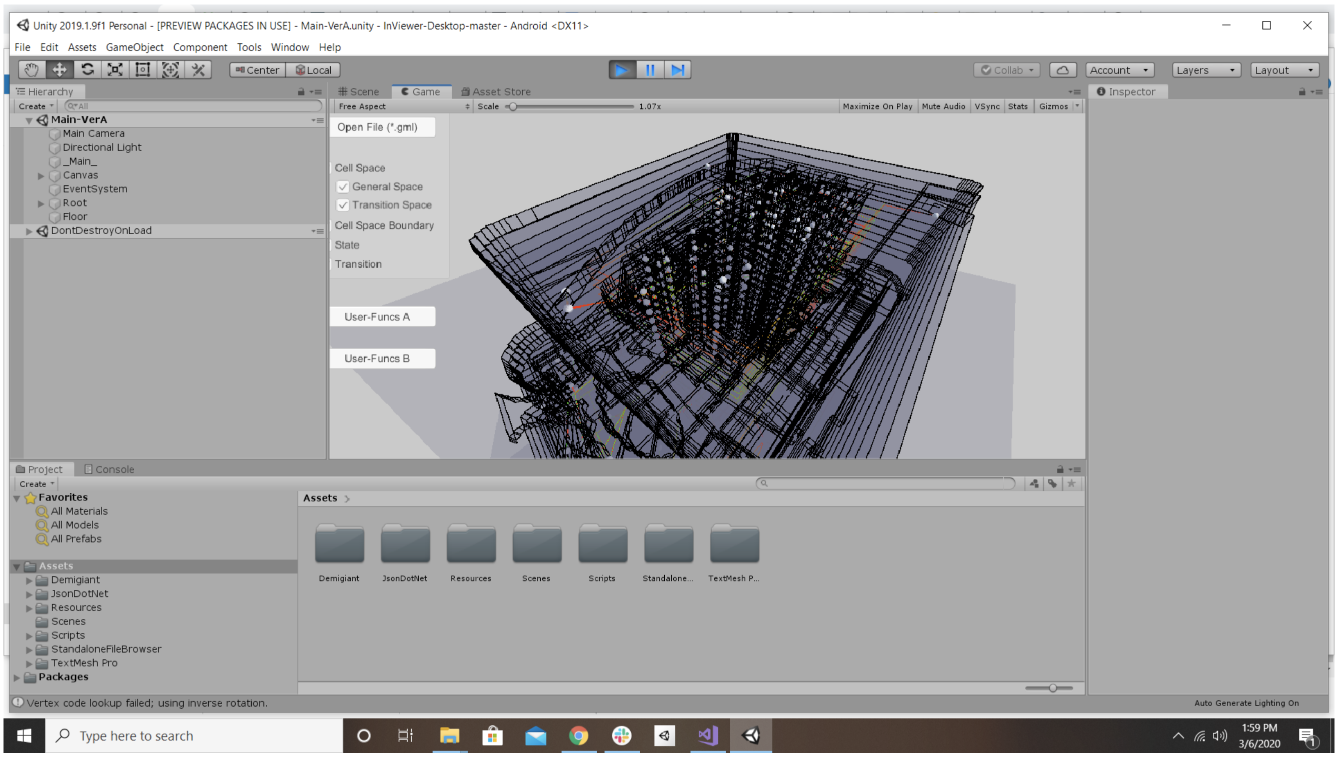


Figure 55. Unity 3D with STEMLab InViewer Plugin

One limitation in this approach is that the entire IndoorGML dataset must be downloaded locally in order to display the content. Thus, the dataset could not be loaded onto a mobile device. A larger effort would be required to custom develop a mobile application in Unity 3D to parse and display IndoorGML, possibly using GeoPackage or a WFS model server via the OGC API - Features accessible via the web. Integration of the IndoorGML plugin within a GeoPortal in the future could provide value to end users attempting to use IndoorGML for indoor navigation in an AR mobile application.

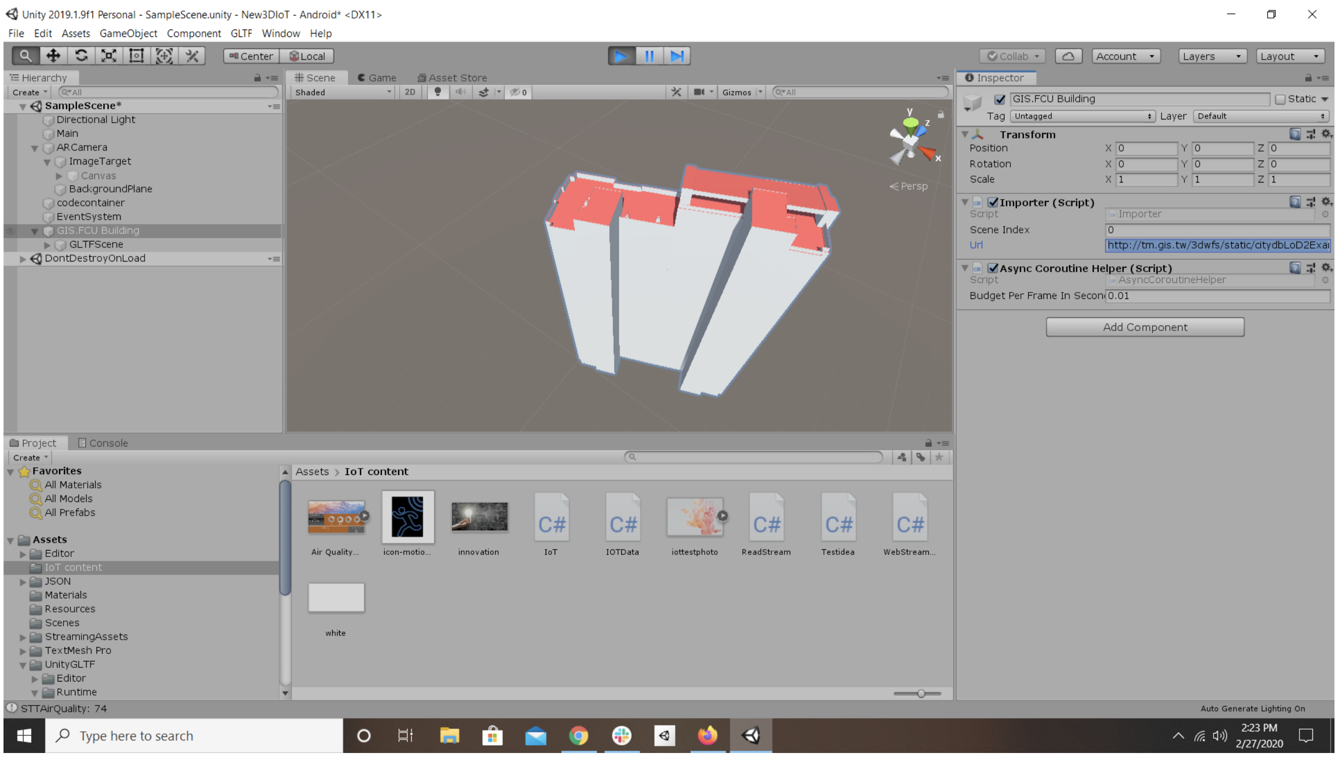
7.5.2. CityGML (3D Tiles)

Due to a policy constraint with the sponsor, the CityGML content had to be delivered using 3D Tiles. Thus, this content was consumed by Skymantics 3D GeoPortal using GL Transmission Format (glTF), a file format for 3D scenes and models using the JSON standard. The glTF dataset was provided via a WFS model sever by GIS.FCU.

Initially, Skymantics attempted to use the Mapbox SDK for Unity 3D to display the glTF content. However, Mapbox does not support the B3DM standard. Despite containing JSON, the file was incompatible with the Mapbox Studio service for serving map and tile content.

Skymantics also investigated use of the NASA-AMMOS Unity 3D Tiles plugin. This plugin visually displays the data locally or by streaming it to Unity for rendering and visualization. However, this software appeared to be at an early stage of development and had limitations that prevented its use. The application was slow to respond, but could be a viable solution in the future.

In the final implementation, Skymantics used the Khronos UnityGLTF plugin for Unity 3D engine to display the Sejong City Building (see Figure 56).



GIS.FCU Building Static
Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Importer (Script)
Script Importer
Scene Index 0
Url http://tm.gis.tw/3dwfs/static/citydbLoD2Exa
Async Coroutine Helper (Script)
Script AsyncCoroutineHelper
Budget Per Frame In Secon 0.01

Figure 56. Unity 3D with STEMLab InViewer Plugin

Some challenges were encountered because the plugin was out-of-date for the current version of Unity 3D, and significant effort was made to update the plugin to function in the latest version of Unity 3D used in MAIDEN. Additionally, the plugin contained several artifacts for use of Microsoft HoloLens. These packages had to be removed in order to display within the Unity 3D environment properly. Once again, integration of glTF within the AR environment needs significant effort, which can be considered in future work.

7.5.3. SensorThings API

Skymantics developed custom scripts to retrieve the air quality and occupancy data from various participant components. The first approach used the Unity 3D network handler to send an HTTP GET request to the Sensor Things API. This is encapsulated in a game object to make the request and parse the data. Then another script rendered the user interface for the button and text association that would display the final result retrieved from the STAPI. In order to retrieve the latest readings, the API request for the $PM_{2.5}$ data had to be sorted by time:

"http://193.196.138.56:8080/frost-airquality/v1.0/Datastreams(6)/Observations?$select=resultTime,result&$orderby=resultTime%20desc"

Once that data was received, the 3D GeoPortal parses the JSON content. Unity 3D does not have a native JSON handler. Therefore, a custom script was developed to parse the readings. Once the data is parsed, it is stored in a debug log in Unity 3D which is then converted to a gameobject text. The text value is associated with a UI button that is incorporated into an image that appears once the image target is triggered.

The Vuforia SDK was used to implement the AR portion of the 3D GeoPortal. A Quick Response (QR) code was used to trigger the event to retrieve the sensor reading and display the image of the result (see Figure 57).

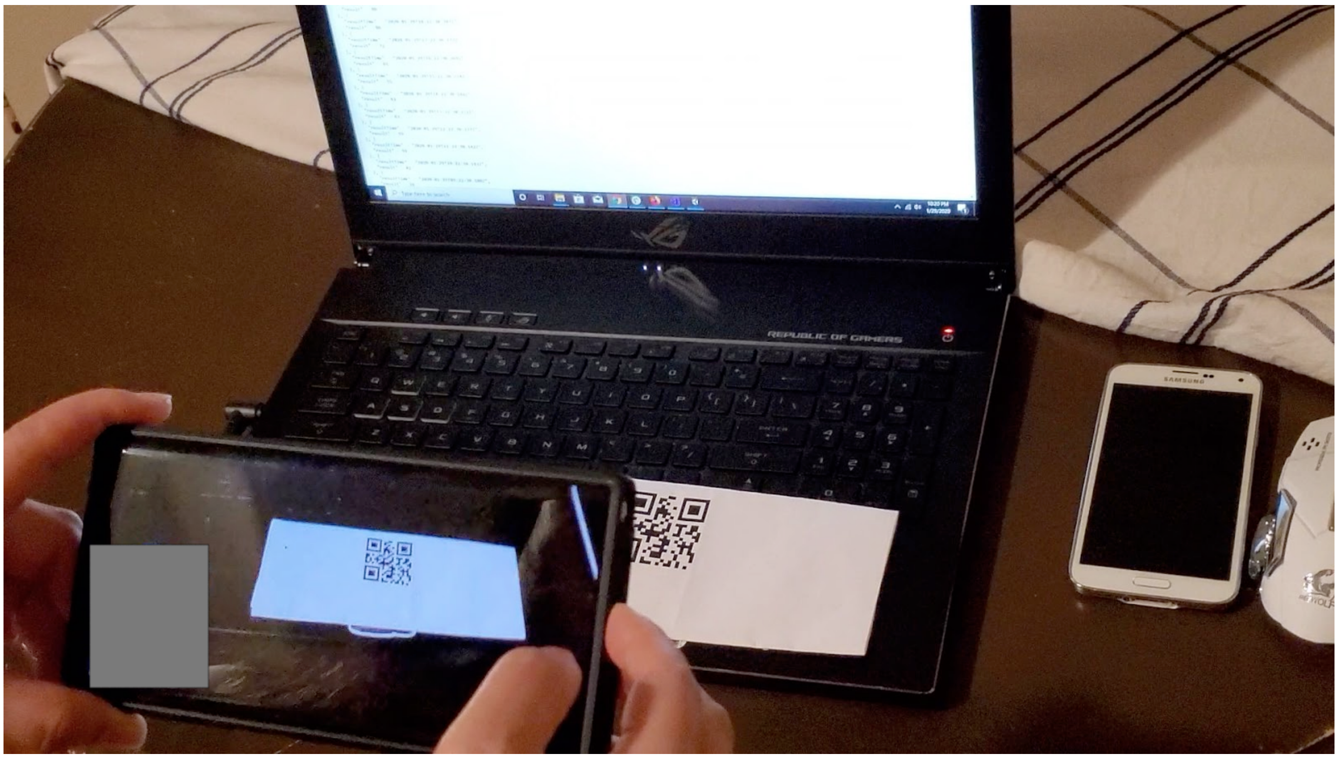

[redacted]

Figure 57. Mobile device scan of a QR code

The result of the STAPI query is displayed in the results of the image rendered on the mobile device (see Figure 58)

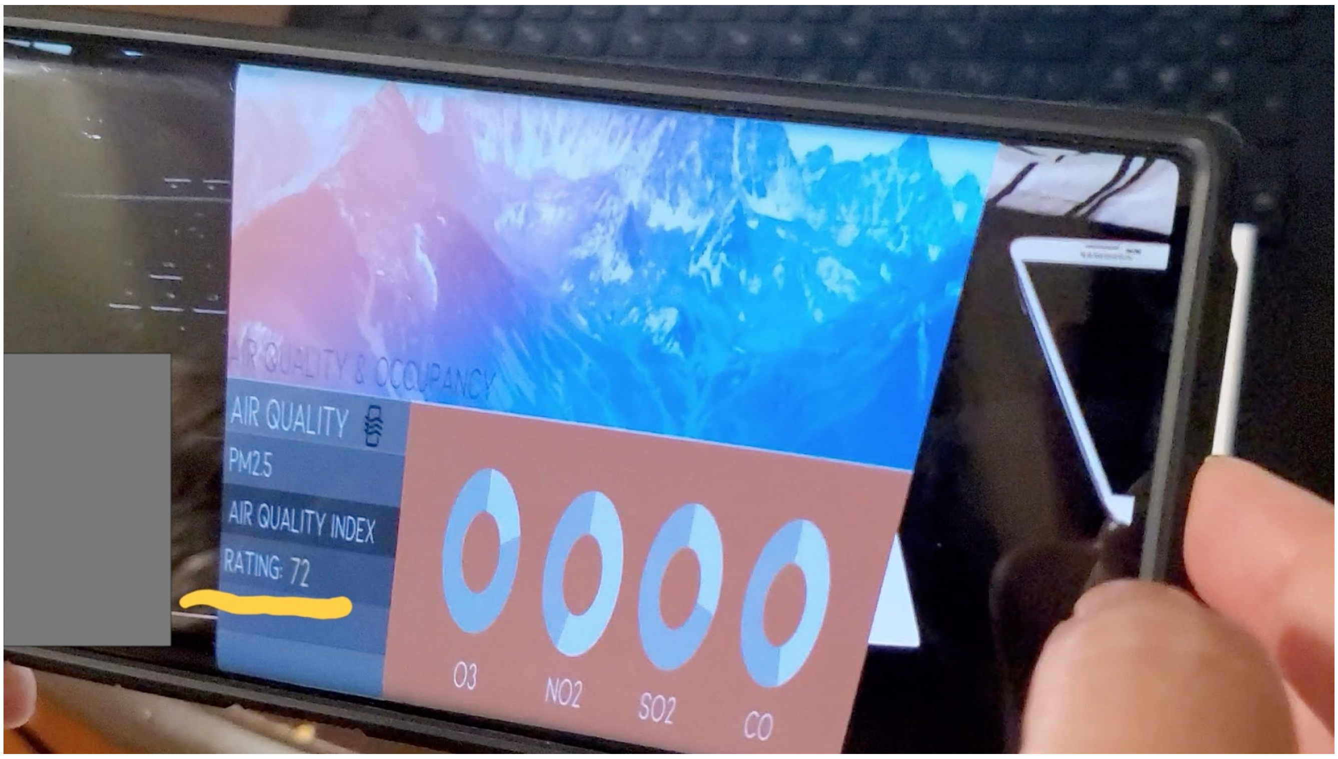

[redacted]

Figure 58. Augmented Reality Overlay

7.5.4. Recommendations for Future Work

- Skymantics recommends future development to include a mobile application for IndoorGML using a WFS model server to display indoor maps in augmented reality.
- The size of the glTF dataset being served by the OGC API - Features implementation might cause

performance issues if the network has limited bandwidth. The file size of the data might cause an issue performance on a mobile app. Skymantics also recommends considering improvements for serving the buildings based on zoom level via an implementation of the draft OGC API - Tiles specification.

- Unity 3D can be used for AR visualization of STAPI data. Vuforia SDK is only compatible with Windows and MacOS. Further exploration of ARCore and ARkit to handle the AR would be worth investigating as they run on Linux.
- The concept of Markerless AR currently used in video games (e.g. Pokemon GO) can provide interesting use cases for Smart Cities AR. When a user is in close proximity to geographic assets, it will trigger a pop up on their screen.

Chapter 8. Summary

8.1. Recommendations and change requests

8.1.1. Features of Interest and observed/inferred properties in the built environment

The core challenge of this initiative was to forge a close connection between the stable view of city features represented by building or infrastructure models, and the dynamic, shifting view of sensors, observation datastreams, and observed properties. The former may incorporate characteristic feature properties but those properties are generally expected to remain constant in both their types and their values. Sensor observed or model inferred properties, on the other hand, are likely to vary constantly in value and change in type with every new sensor or computational procedure.

Existing data models and API's do not integrate these perspectives very well. Neither CityGML 2 nor 3D Tiles have particular allowances for properties that change in value and number. Same with WFS or OGC API - Features. A new feature in CityGML 3, "dynamizers" does support varying property values, but it is still unclear what the best practice will be for adding and substracting properties on-the-fly.

On the observation side, SensorThings API connects each datastream to a "Feature of Interest" (FoI) as well as a "Thing" but it is left up to those respective feature representations, lampposts for example, to characterize their own places in the larger environment such as a building facades or city blocks.

The 3D-IoT pilot demonstrated that the most feasible solution is not to overload either the feature or the observed property perspectives with each other's capabilities, but to establish and maintain accurate and durable common identifiers (e.g. GML ID's) that align FoI's / Things of the latter perspective with the city model elements of the former perspective.

While this approach was successful in bringing sensor data together with model objects, it was not very durable. Any changes to one system needed to be carefully re-aligned with other systems on a system-by-system basis. One could imagine a more sustainable approach incorporating an authoritative service, a registry perhaps, to which other systems would be subscribed and which would maintain a current set of feature identifiers, and as well a current set of observed properties corresponding to those features with links to the sources of observations setting the values of those observed properties.

This still leaves the issue of the "hack" utilized in the pilot of incorporating what are essentially foreign key identifiers into the definitions of Things and FoI within SensorThings services. A potential SensorThings API change request will be to provide a more organized means of indicating where additional / authoritative information about those objects and their relationships to one another can be found.

8.1.2. Derived observations and virtual sensors O&M and the SensorThings API

Another challenge of the pilot was to derive observed properties for Features of Interest which are not directly the result of observations. In the case of building occupancy, the derivation procedure involved adding up the occupancies of rooms comprising a larger building unit such as a floor. In the case of air quality, the derivation procedure involved interpolating sampled air quality locations to a larger grid so that, for example, a grid subset might correspond to a neighborhood, street, or block. While the procedures were carried out by the Helyx web processing service and visualized in one or more of the client components, there was also interest in being able to load these "virtual" observations into a SensorThings API service.

Procedures themselves are concepts inherent in the Observations and Measurements model and in the SOSA-SSN ontology that corresponds to it. Related to this is the Semantic Sensor Network Ontology (SSN) which follows a horizontal and vertical modularization architecture by including a lightweight but self-contained core ontology called SOSA (Sensor, Observation, Sample, and Actuator) for its elementary classes and properties. In the SOSA-SSN ontology, sensors execute procedures, but procedures can also associated with sets of inputs and outputs and used by observations. Procedures are not, however, implemented separately from Sensors in the current SensorThings API model, and any observation (as part of a datastream) requires a Sensor to be associated with it. A "virtual" derived observation, therefore, requires a virtual Datastream and a virtual Sensor to go with a virtual Thing and presumably a non-virtual ObservedProperty and FoI. It was indeed possible, in this way, to load derived observations into a SensorThings API instance.

The relationship that could not easily be represented in the SensorThings API, though, was between input observations from real sensors, a derivation procedure, and output observations from virtual sensors. There are several possible approaches for this:

- Document this relationship outside of the SensorThings API instance in case that it is needed for some purpose other than day-to-day observation usage.
- Add a Procedure object with inputs and outputs to the SensorThings API as in SOSA-SSN. This is a possible extension to the SensorThings API but adds some degree of complexity to the design of an intentionally "simple" API.
- Add an "isInput" relation between Datastreams and Sensors. This is a relatively simple addition but contravenes the SensorThings API design principle of having at most one relation between each two types of resources.
- Include an optional input query string attribute for virtual sensors. This query could be used by a SensorThings API client to query and receive the observations used to compute the derived observation. Together with an attribute referencing or describing the procedure (or processing service request) used, the origin of a derived observation could then be recovered if needed without overly complicating SensorThings API. For example, a user might want to see the point observations that were used to construct an interpolated grid observation, in order to try to explain anomalous values. With this provenance information, it would be straightforward to do this.

Whichever approach is taken, the pilot made clear that processing and modeling of sensor observations to produce new ones has many useful applications and a SensorThings API model for representing this is a worthwhile endeavor.

8.1.3. Smart building / district design implications of 3D-IoT work

The 3D-IoT Pilot was focused on testing certain technical aspects of interoperability between sensor information and building information. The intent of a pilot, however, is always to consider questions of implementation and deployment feasibility and value. One useful question is: what are the implications of the Pilot work and outcomes for future design of smart buildings and districts that support comprehensive awareness of built environment observable properties based on IoT sensors and digital infrastructure?

One very clear answer to this question is the importance of keeping the design, physical, and digital representations of built environment and environment sensing elements aligned with each other. Awareness based on observable properties is only valid if the Things providing sensing capabilities and the building or district Features of Interest (FoI) that they observe (singly or in aggregate) are correctly linked in all three representational realms. If the sensor platform designed / constructed for Room 24-A is linked in the digital infrastructure to Room 26-A (or even worse, Room 24-A in another building), any data being provided by that sensor will provide an incorrect or even dangerous awareness of building state.

This has at least three design implications:

- Designs need to incorporate, maintain, and identify the Things supporting sensor devices, as well as to indicate which built FoI's they are associated with (if different). For the purposes of aggregation, it would even be worthwhile to catalog those feature hierarchies which should be used to derive observed properties for aggregate features such as floors. Design concerns might also include sensor coverage, redundancy, connectivity, and extensibility.
- Designs should include the Information and Communications Technology (ICT) as well as power infrastructure needed to connect, validate, and maintain sensor data streams.
- Smart building / smart district design needs to account not only for the form, type, positioning, and connections of construction elements, but also their physical and digital identities for linking to digital information. Even methods of validation and recovery should be considered, such as persisting those identities in bar / QR codes physically attached to building components.

A significant part of the challenge accompanying the transition to smart buildings / districts / cities is retrofitting existing structures with these sorts of sensing elements and digital "twin" model information. A comprehensive design approach will reduce the burden of this difficulty and cost at least for new construction or major refits. A useful role for this Pilot and subsequent initiatives will be to further delineate critical and useful design criteria for IoT-enabled built environments.

Appendix A: Open API Specification of Property Estimator WPS for synthetic air quality Sensor Measurements

This section further describes the Helyx Implementation, regarding outdoor air quality sensors. The Helyx Processes API is written in Python 3.6 using the flask-restplus framework and adheres to the OpenAPI specification 3.0. A fragment of the schema implemented by the API is presented below.

```
{
    "swagger": "2.0",
    "basePath": "/",
    "paths": {
        "/common/": {
            "get": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Returns the common landing page, made up of links to API
resources",
                "operationId": "get_common",
                "tags": [
                    "common"
                ]
            }
        },
        "/common/administer": {
            "get": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Returns the current configuration for the persisted model
",
                "operationId": "get_administer",
                "tags": [
                    "common"
                ]
            },
            "post": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Store new config and recreate persisted model",
                "operationId": "post_administer",
                "parameters": [
                    {
                        "name": "payload",
                        "required": true,
                        "in": "body",
                        "schema": {
                            "$ref": "#/definitions/configuration"
                        }
                    }
                ],
                "tags": [
                    "common"
                ]
            }
        },
        "/common/administer/rasters/": {
            "get": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Returns a list of the requested estimations",
                "operationId": "get_administerresources",
                "tags": [
                    "common"
                ]
            }
        },
        "/common/administer/rasters/{name}": {
            "parameters": [
                {
                    "name": "name",
                    "in": "path",
                    "required": true,
                    "type": "string"
                }
            ],
            "get": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Returns a list of rasters created by the persisted model",
                "operationId": "get_administerresource",
                "tags": [
                    "common"
                ]
            }
        },
        "/common/api": {
            "get": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Returns the api defintion for this web service (not yet
implemented)",
                "operationId": "get_ogcapi",
                "tags": [
                    "common"
                ]
            }
        },
        "/common/conformance": {
            "get": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Returns a list of the conformance classes this web service
conforms to (currently using placeholders)",
                "operationId": "get_conformance",
                "tags": [
                    "common"
                ]
            }
        },
        "/common/estimator": {
            "get": {
                "responses": {
                    "200": {
                        "description": "Success"
                    }
                },
                "summary": "Returns a list of the requested estimations",
                "operationId": "get_estimator",
                "tags": [
                    "common"
                ]
            },
            "post": {
                "responses": {
                    "201": {
                        "description": "Estimation successfully requested."
                    }
                },
                "summary": "Creates an estimate for either air quality or occupancy",
                "operationId": "post_estimator",
                "parameters": [
                    {
                        "name": "payload",
                        "required": true,
                        "in": "body",
                        "schema": {
                            "$ref": "#/definitions/estimation"
                        }
                    }
                ],
                "tags": [
                    "common"
                ]
            }
        },
        "/common/estimator/{uuid}": {
            "parameters": [
                {
                    "name": "uuid",
                    "in": "path",
                    "required": true,
                    "type": "string"
                }
            ],
            "get": {
                "responses": {
                    "404": {
                        "description": "estimate not found."
                    }
                },
                "summary": "Returns details of an estimate",
                "operationId": "get_estimate",
                "tags": [
                    "common"
                ]
            }
        }
    },
    "info": {
        "title": "Helyx OGC API - Processes",
        "version": "0.1",
        "description": "An OGC API - Processes for IoT property estimation"
    },
    "produces": [
        "application/json"
    ],
    "consumes": [
        "application/json"
    ],
    "tags": [
        {
            "name": "common",
            "description": "The landing page of the OGC API"
        }
    ],
    "definitions": {
        "estimation": {
            "required": [
                "name"
            ],
            "properties": {
                "name": {
                    "type": "string",
                    "description": "A name to give to your request"
                },
                "air readings": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/air readings"
                    }
                },
                "occupancy readings": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/occupancy readings"
                    }
                }
            },
            "type": "object"
        },
        "air readings": {
            "required": [
                "default sensorThing API"
            ],
            "properties": {
                "aggregation type": {
                    "type": "string",
                    "description": "The type of aggregation to use, either simple on-
the-fly aggregation or derived from a persisted model",
                    "example": "on-the-fly",
                    "enum": [
                        "on-the-fly",
                        "persisted"
                    ]
                },
                "default sensor type": {
                    "type": "array",
                    "items": {
                        "type": "string",
                        "description": "The type of estimation to be carried out",
                        "example": "PM2.5",
                        "enum": [
                            "PM2.5",
                            "CO"
                        ]
                    }
                },
                "default radius": {
                    "type": "integer",
                    "description": "The default spherical radius to be used in the
process (in meters)"
                },
                "default start time": {
                    "type": "string",
                    "format": "date-time",
                    "description": "The default start time for estimates"
                },
                "default end time": {
                    "type": "string",
                    "format": "date-time",
                    "description": "The default end time for estimates"
                },
                "default sensorThing API": {
                    "type": "string",
                    "description": "The SensorThing API to query",
                    "example": "Cyient",
                    "enum": [
                        "Cyient",
                        "GIS.FCU",
                        "SensorUp",
                        "STT"
                    ]
                },
                "centroids": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/centroid"
                    }
                },
                "GMLIDs": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/GMLIDs"
                    }
                }
            },
            "type": "object"
        },
        "centroid": {
            "properties": {
                "geometry": {
                    "$ref": "#/definitions/geometry"
                },
                "properties": {
                    "$ref": "#/definitions/properties"
                }
            },
            "type": "object"
        },
        "geometry": {
            "required": [
                "type"
            ],
            "properties": {
                "type": {
                    "type": "string",
                    "description": "The geometry type of sensor to be used",
                    "example": "Point",
                    "enum": [
                        "Point"
                    ]
                },
                "coordinates": {
                    "type": "array",
                    "description": "The coordinates for the centroid, either Lon, Lat
and Azimuth or just Lon, Lat",
                    "items": {
                        "type": "number"
                    }
                }
            },
            "type": "object"
        },
        "properties": {
            "required": [
                "sensorThing API"
            ],
            "properties": {
                "radius": {
                    "type": "integer",
                    "description": "The spherical radius to be used in the process (in
meters)"
                },
                "start time": {
                    "type": "string",
                    "format": "date-time",
                    "description": "The start time for estimates"
                },
                "end time": {
                    "type": "string",
                    "format": "date-time",
                    "description": "The end time for estimates"
                },
                "sensor type": {
                    "type": "string",
                    "description": "The type of sensor to be used",
                    "example": "PM2.5",
                    "enum": [
                        "PM2.5",
                        "CO"
                    ]
                },
                "sensorThing API": {
                    "type": "string",
                    "description": "The SensorThing API to query",
                    "example": "Cyient",
                    "enum": [
                        "Cyient",
                        "GIS.FCU",
                        "SensorUp",
                        "STT"
                    ]
                }
            },
            "type": "object"
        },
        "GMLIDs": {
            "required": [
                "ID"
            ],
            "properties": {
                "ID": {
                    "type": "string",
                    "description": "The ID of the feature for which to calculate an
aggregation"
                },
                "properties": {
                    "$ref": "#/definitions/properties"
                }
            },
            "type": "object"
        },
        "occupancy readings": {
            "required": [
                "GMLIDs"
            ],
            "properties": {
                "default start time": {
                    "type": "string",
                    "format": "date-time",
                    "description": "The default start time for estimates"
                },
                "default end time": {
                    "type": "string",
                    "format": "date-time",
                    "description": "The default end time for estimates"
                },
                "GMLIDs": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/GMLIDs"
                    }
                }
            },
            "type": "object"
        },
        "configuration": {
            "required": [
                "AOI",
                "API",
                "SensorType"
            ],
            "properties": {
                "API": {
                    "type": "string",
                    "description": "The SensorThing API to query",
                    "example": "Cyient",
                    "enum": [
                        "Cyient",
                        "GIS.FCU",
                        "SensorUp",
                        "STT"
                    ]
                },
                "SensorType": {
                    "type": "string",
                    "description": "The type of sensor to be used",
                    "example": "PM2.5",
                    "enum": [
                        "PM2.5",
                        "CO"
                    ]
                },
                "AOI": {
                    "$ref": "#/definitions/polygon"
                }
            },
            "type": "object"
        },
        "polygon": {
            "required": [
                "type"
            ],
            "properties": {
                "type": {
                    "type": "string",
                    "description": "The geometry type of sensor to be used",
                    "example": "Polygon",
                    "enum": [
                        "Polygon"
                    ]
                },
                "coordinates": {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": {
                            "type": "array",
                            "items": {
                                "type": "number"
                            }
                        }
                    }
                }
            },
            "type": "object"
        }
    },
    "responses": {
        "ParseError": {
            "description": "When a mask can't be parsed"
        },
        "MaskError": {
            "description": "When any error occurs on mask"
        },
        "NoResultFound": {}
    }
}
```

Appendix B: Revision History

Table 8. Revision History

Date	**Editor**	**Release**	**Primary clauses modified**	**Descriptions**
Feb 24, 2020	V. Coors	0.9	all	initial version
Mar 30, 2020	V. Coors	1.0	all	final version

Appendix C: Bibliography