The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume L-4/W1-2026
Free and Open Source Software for Geospatial (FOSS4G) 2026 – Academic Track "Bridging Geospatial Technology and Humanity",
30 August–5 September 2026, Hiroshima, Japan

Making OGC IndoorGML 2.0 Web-Ready: API – IndoorFeatures with IndoorJSON

Dong Gwon Kang[1]*, Seongmin Choi[1]*, Taehoon Kim[2]†, Kyung-Sook Kim[2], Ki-Joune Li[1]

[1] Pusan National University, Busan, South Korea - (kangdg2001, csm010311, lik)@pnu.edu
[2] National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan - (kim.taehoon, ks.kim)@aist.go.jp

Keywords: IndoorGML, IndoorJSON, OGC API, Indoor Navigation, Indoor Spatial Data, pygeoapi

Abstract

The infrastructure for indoor spatial data remains fragmented compared to its outdoor counterpart, lacking the interoperable and developer-friendly tools that have made outdoor mapping so prevalent. This paper presents API-IndoorFeatures, an open-source RESTful API crafted to bridge this gap by delivering IndoorGML 2.0 data in the IndoorJSON format through standardized HTTPS endpoints. This framework grants developers access to OGC indoor spatial standards without requiring in-depth knowledge of the underlying data models.

Built on pygeoAPI, the system enhances its architecture with a specialized backend for indoor spatial data that complies with the OGC API – Features standard. Spatial data is managed in PostgreSQL using PostGIS, enabling efficient geometric queries, while pgRouting supports indoor navigation. A notable contribution of this work is the adoption of IndoorJSON, a lightweight, web-native JSON serialization of the IndoorGML information model, which is currently in the process of standardization by OGC. This format replaces the parsing complexities associated with XML-based encodings and integrates seamlessly into contemporary development environments.

API-IndoorFeatures offers hierarchical data access across both the primal and dual space layers of IndoorGML 2.0, providing comprehensive Create, Read, Update, and Delete (CRUD) support, as well as geometric bounding box queries that enable progressive floor-by-floor rendering and routing from origin to destination. Distinct among open-source indoor spatial systems, it enforces IndoorGML 2.0 schema validation and referential integrity on all write operations. To our knowledge, this may be one of the first systems to operationalize the revised IndoorGML 2.0 conceptual model introduced in August 2025. This work expands previous contributions beyond mere data construction to encompass serving, validation, and consumption, thereby offering a reusable open-source platform for the next generation of indoor location services.

1. Introduction

As indoor environments grow increasingly complex, the demand for accurate and accessible indoor spatial data has expanded across a wide range of domains. Smart buildings, airports, shopping malls, and university campuses increasingly rely on indoor location-based services for navigation. In contrast to outdoor geospatial data, such as GPS and OpenStreetMap, which benefit from decades of investment in mature infrastructure and a rich ecosystem of standardized APIs, indoor spatial data remains fragmented, difficult to access, and challenging to integrate into modern applications.

OGC IndoorGML 2.0 (Zlatanova et al., 2025) is the most comprehensive international standard for indoor spatial information, offering a semantically rich data model that captures both the physical geometry of indoor environments through its primal space layer and their navigable connectivity through a dual space graph. Its multi-layered space model also supports multiple thematic perspectives, such as topographic layout, accessibility constraints, or emergency routes, within a single building model. Despite this expressiveness, IndoorGML's reliance on XML-based serialization introduces significant practical barriers. XML requires constructing a full document tree before individual elements can be accessed, incurs comparatively high parsing costs, and depends on domain-specific libraries that complicate integration into modern software stacks. These limitations have restricted the adoption of IndoorGML primarily to specialized GIS toolchains, leaving it largely inaccessible to the broader community of web and mobile developers.

In contrast, JSON has become the de facto standard for data exchange in web-based systems. Its lightweight structure maps naturally to objects in virtually all modern programming languages, supports incremental parsing, and integrates seamlessly with JavaScript frameworks, mobile platforms, and contemporary GIS clients. To bridge this gap, IndoorJSON (Kim et al., 2026) has been proposed as an OGC-standardized JSON-based encoding for the IndoorGML 2.0 conceptual model, preserving its full semantic structure, including `CellSpace`, `CellBoundary`, `Node`, `Edge`, `ThematicLayer`, and `InterLayerConnection`, while providing a web-friendly representation suitable for modern development environments.

However, a standardized encoding alone is insufficient. Prior work on IndoorGML has focused predominantly on the data construction phase of the indoor spatial data lifecycle, producing generators, editors, and conversion tools such as InFactory (Jeong et al., 2018). Comparatively little attention has been given to how IndoorGML data can be efficiently served, queried, validated, and consumed by downstream applications. This gap between the semantic expressiveness of IndoorGML and its practical accessibility in production systems remains a key challenge for the indoor spatial data community.

To address this, we present API – IndoorFeatures, a RESTful web API for indoor spatial data that operationalizes IndoorGML 2.0 using IndoorJSON encoding. Built by extending pygeoapi (The pygeoapi team, 2026), a Python-based OGC API framework, the API inherits the resource and endpoint conventions

* These authors contributed equally to this work
† Corresponding author

of OGC API – Features (Portele et al., 2022) and extends them with indoor-specific functionality, including multi-layered space management, spatial and semantic filtering, and network-based indoor navigation. In doing so, it extends the indoor spatial data lifecycle beyond construction to encompass serving, validation, sharing, and consumption, providing a reusable open-source platform for next-generation indoor location-based services.

The main contributions of this paper are as follows:

- A RESTful API design for accessing and manipulating IndoorGML 2.0 data, aligned with OGC API – Features and encoded in IndoorJSON.
- A demonstration of how IndoorGML 2.0 concepts — Thematic Layers, Primal Space, Dual Space, and Inter-Layer Connections — can be exposed through a standard-compliant web API.
- An example of the system using a real building dataset, validating its effectiveness for practical indoor navigation and querying tasks.

The remainder of this paper is structured as follows. Section 2 reviews background and related work, including IndoorGML 2.0, IndoorJSON, and prior systems. Section 3 presents the overall system architecture, API design, and database schema. Section 4 details the implemented endpoints, with Section 5 addressing the key implementation challenges encountered during development. Section 6 demonstrates the API using a real building dataset from Pusan National University, and Section 7 concludes the paper.

2. Background

Indoor spatial data has traditionally been represented using IndoorGML, a standard defined by the Open Geospatial Consortium (OGC) that employs an XML-based serialization format. While XML provides a structured and semantically rigorous representation, it introduces verbosity and complexity that can hinder adoption in modern web and mobile development contexts. To address this, IndoorJSON (Kim et al., 2026) has been proposed as a lightweight, JSON-based encoding profile of the IndoorGML 2.0 conceptual model. It should be noted that IndoorJSON is still a work in progress and has not yet been ratified as an OGC standard; the mapping between IndoorJSON constructs and their IndoorGML 2.0 (Zlatanova et al., 2025) counterparts remains under active development. Nevertheless, its simpler structure and natural compatibility with web technologies make it significantly more accessible for developers building location-aware applications. At the same time, it retains the semantic foundations of IndoorGML 2.0, including elements such as `ThematicLayer`, `PrimalSpace`, `DualSpace`, and `InterLayerConnection`, ensuring that the two representations are conceptually equivalent.

The design of the service interface for the API – IndoorFeatures conforms to the standards established by OGC API – Features (Portele et al., 2022), a RESTful standard for accessing geospatial feature data over the web. Rather than defining an entirely independent API, this work deliberately inherits from OGC API – Features, a decision grounded in conceptual alignment: an IndoorFeature is fundamentally a specialized form of a geospatial feature, carrying geometry, identity, and spatial relationships in the same manner as any other feature type. This inheritance enables the implementation to be built on top of compliant servers without modifications to the core API layer, and ensures that the resource paths and HTTP interaction patterns, including the `/collections` and `/collections/{collectionId}/items` endpoints, remain consistent with established OGC conventions.

To implement this design, we built the framework on top of pygeoapi (The pygeoapi team, 2026), a Python-based server that complies with the OGC API – Features and other relevant OGC API standards. This allows the development effort to be focused entirely on indoor-specific functionality rather than on recreating foundational API conventions.

The closest prior work to this study is InFactory (Jeong et al., 2018), a RESTful API server designed to support the construction of IndoorGML data. It allows users to generate and manage indoor spatial information without needing a deep understanding of the IndoorGML schema or XML encoding. However, InFactory was developed based on IndoorGML 1.0 and does not support the updated IndoorGML 2.0 resource model. While it accepts a custom JSON-based input format, it outputs IndoorGML as raw XML and does not conform to any standardized JSON encoding schema. Furthermore, its scope is limited to the data construction phase of the IndoorGML ecosystem, lacking support for spatial querying, level-based filtering, or indoor routing. Additionally, it does not conform to any OGC API standards. These limitations reveal a significant gap between indoor data construction and the broader lifecycle of IndoorGML data provision, querying, and consumption. The API – IndoorFeatures is designed to bridge this very gap.

3. System Design

3.1 RESTful API Design Goals

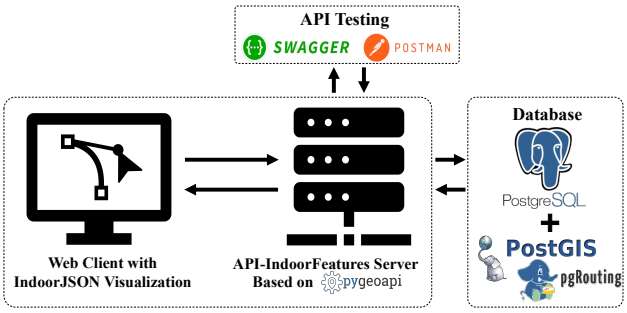

Figure 1. The architecture for API – IndoorFeatures

The primary objective of this work is to develop a RESTful API that provides standardized access to indoor spatial data. Existing solutions, while semantically rich, impose a high integration burden on developers due to their reliance on complex data formats and non-web-native interfaces. The proposed API addresses this by using IndoorJSON as its serialization format, reducing the overhead of consuming indoor spatial data in web and mobile applications. A secondary goal is to demonstrate that IndoorJSON can be used in a production API context without operational issues, thereby validating its suitability as a web-facing profile for IndoorGML 2.0.

3.2 Overall Architecture

The API – IndoorFeatures, illustrated in Figure 1, is built on top of pygeoapi (The pygeoapi team, 2026), an open-source OGC API – Features compliant server framework. pygeoapi follows a provider-based plugin architecture in which data sources are decoupled from the API layer through a standardized provider interface. By implementing a custom IndoorFeatures provider that conforms to this interface, API – IndoorFeatures integrates seamlessly into the pygeoapi request lifecycle without modifying the core framework, and indoor-specific endpoints that extend beyond the OGC API – Features core, such as `layers`, `primal`, `dual`, and `interlayerconnections`, are registered as additional route extensions within the same pygeoapi server instance, keeping the entire service under a single unified deployment. The backend is powered by PostgreSQL (The PostgreSQL Global Development Group, 2025), with PostGIS (PostGIS Project Steering Committee, 2025) providing geometric query support and pgRouting (pgRouting Project, 2024) enabling network-based routing operations on the dual-space connectivity graph. A web client is provided for direct interaction with the API server, allowing users to query, visualize, and navigate indoor spatial data through a browser-based interface. This architecture was chosen to maximize compliance with OGC standards while minimizing the need for custom server-side infrastructure.

3.3 API Resource Model

The API resource model follows a hierarchical structure grounded in the IndoorGML 2.0 data model. The entry point for the service is the landing page, which provides links to the OpenAPI documentation (OpenAPI Initiative, 2018) and the available collections of spatial data. All indoor spatial data is accessed through the `/collections/{collectionId}/items` endpoint, where each collection contains a set of `IndoorFeature` objects. Each `IndoorFeature` object can be represented as an IndoorJSON, typically encapsulating a single indoor entity such as a building or floor complex. `IndoorFeature` may maintain `InterLayerConnection` entries that describe topological relationships across different `ThematicLayer` instances. Each `IndoorFeature` is composed of one or more `ThematicLayer` members, each of which is further subdivided into a `PrimalSpaceLayer`, representing the geometric state of the space, and a `DualSpaceLayer`, representing its topological graph structure.

3.4 Database Schema

Indoor spatial data in API – IndoorFeatures is persisted in PostgreSQL with the PostGIS and pgRouting extensions enabled. The schema is designed to reflect the resource model mentioned.

The schema consists of six core tables as visible in Figure 2. The `collection` table serves as the top-level container, storing collection-level metadata as JSON binary (jsonb). The `indoorfeature` table references its parent collection and stores the GeoJSON geometry and properties of each IndoorFeature. The `thematiclayer` table represents each thematic layer within an IndoorFeature, capturing theme type, directionality, logical flags, and temporal validity attributes for both primal and dual spaces. Spatial elements(`CellSpace` and `CellBoundary`) are unified in the `cell_space_n_boundary` table, distinguished by a `celltype` enumeration attribute. A deliberate split is applied to geometry storage: 2D geometry is stored as a native PostGIS `geometry` column, enabling GiST spatial indexing for bounding box filtering via `bbox` query parameters, while 3D geometry is stored as jsonb. This split reflects the limited native support PostGIS has for the volumetric geometry types required by IndoorGML 2.0; specifically, the solid types which represent three-dimensional indoor spaces and are not included in the OGC simple features geometry (Herring, 2011). Storing 3D geometry as jsonb preserves the full IndoorJSON geometry object exactly as received and returned, with no lossy data conversion. Since 3D geometry is used primarily for visualization and export rather than spatial querying, the absence of spatial indexing on the jsonb column is an acceptable trade-off.

In the same manner, network elements(`Node` and `Edge`) are unified in the `node_n_edge` table, with edge connectivity represented in a dedicated `connects` join table holding source node, target node, and edge references. This structure is directly compatible with pgRouting graph traversal functions.

Inter-layer relationships are stored in the `interlayerconnection` table, referencing connected layers, cells, and nodes along with a topological expression type drawn from the `topo_type` enum (`CONTAINS`, `OVERLAPS`, `EQUALS`, `WITHIN`, `CROSSES`, `OTHERS`) following the 9-intersection model (Egenhofer, 1989).

A dual identifier strategy is applied uniformly across all tables. Each row carries both an auto-generated `BIGINT` primary key (`id`) used for all internal foreign key relationships and joins, and a user-supplied string identifier (`id_str`) exposed through the API. This design preserves the semantic identifiers defined in IndoorJSON while ensuring that all internal operations benefit from integer-based index lookups, as explained in more detail in section 5.

Several indexes are defined to support the query patterns of the API. Indexes on `duality_id` in both the `cell_space_n_boundary` and `node_n_edge` tables optimize duality resolution during feature retrieval. The `connects` join table is indexed on the `Node`, and `Edge` IDs in `BIGINT` to support efficient graph traversal, and the `interlayerconnection` table is indexed on both connected layer columns to accelerate inter-layer queries.

4. Implementation

Our implementation is publicly available online.[1]

4.1 IndoorFeature Collection and IndoorFeatures

The API resource model directly inherits the collection and feature conventions established by OGC API – Features, extending them with indoor-specific semantics and IndoorJSON encoding.

`Collections` accessible at `{root}/collections` serve as the top-level catalog, supporting GET to list available datasets and POST to register new ones. Collections declare `itemType` as `indoorfeature` to distinguish them from standard feature collections. Individual collections at `{root}/collections/{collectionId}` support GET for metadata retrieval and DELETE to remove the entire collection.

`IndoorFeatures` at `{root}/collections/{collectionId}/items` follow the same items/item pattern as OGC API – Features, with all representations conforming to IndoorJSON.

[1] github.com/STEMLab/API_IndoorFeatures

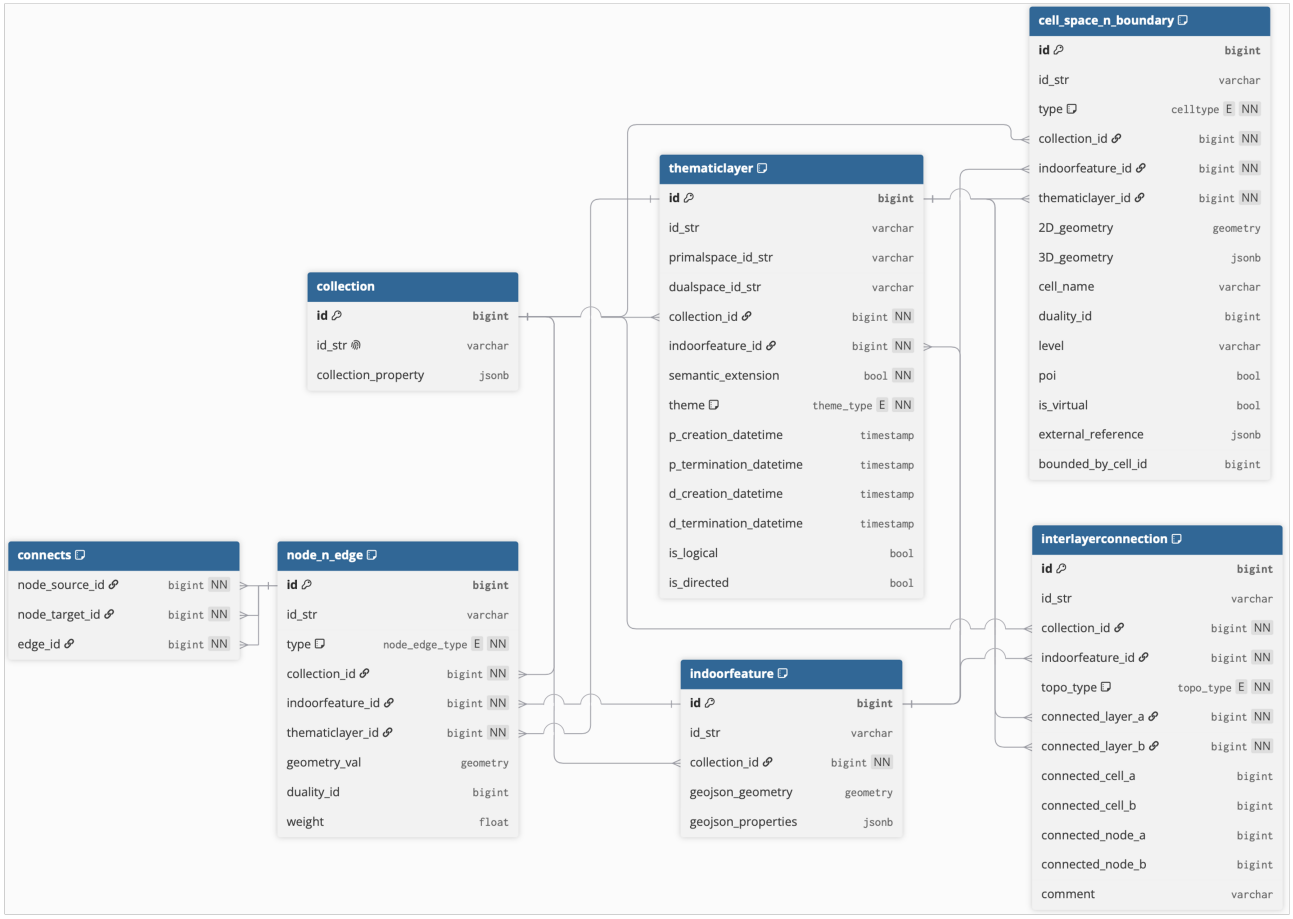

Figure 2. The database schema diagram for API – IndoorFeatures

The items endpoint supports GET with `bbox` and `limit` parameters for spatial filtering and pagination, and POST to create new indoor features that are validated against the IndoorJSON schema. Individual features at `{root}/collections/{collectionId}/items/{featureId}` support GET with optional `bbox` or `level` filtering for conditional detailing—returning only metadata by default and full IndoorJSON content when a spatial filter is applied—and DELETE for cascading removal.

4.2 ThematicLayer

ThematicLayers are exposed as sub-resources of an individual feature at `···/{featureId}/layers` reflecting the multilayered space model of IndoorGML 2.0. Each `ThematicLayer` provides a semantic or functional view of an indoor feature, such as a topographic or routing perspective. The layers endpoint supports GET, returning a lightweight summary with a `levels` array and optional filtering by `theme` and `level` query parameters, and POST for creating a new `ThematicLayer` whose `id` and all internal member identifiers must be unique within the parent feature. Individual layers at `···/layers/{layerId}` support GET with conditional `bbox` and `level` filtering that limits returned `PrimalSpace` members while leaving `DualSpace` members unaffected, and DELETE which cascades to all internal `PrimalSpace` and `DualSpace` members, automatically removing any referencing `InterLayerConnection` to preserve topological integrity.

4.3 InterLayerConnection

InterLayerConnections are exposed as sub-resources of an individual feature at `···/{featureId}/interlayerconnection`s, expressing topological relationships between thematic layers using one of the following types: `CONTAINS`, `OVERLAPS`, `EQUALS`, `WITHIN`, `CROSSES`, or `OTHERS`. The connections endpoint supports GET, returning all inter-layer connections conforming to the IndoorJSON schema with optional filtering by `typeOfTopoExpression` and `connectedLayerId`, and POST for creating a new `InterLayerConnection` between two thematic layers within the same feature, enforcing strict validation to ensure all referenced layers, nodes, and cell spaces exist and that any simultaneously provided `connectedNodes` and `connectedCells` maintain a valid duality relationship. Because inter-layer connections are lightweight relationship resources, individual connections at `{···/interlayerconnections/{connectionId}` support only DELETE, with updates handled by deletion and re-creation.

4.4 PrimalSpace

PrimalSpace is exposed as a sub-resource of a thematic layer at `···/{featureId}/layers/{tId}/primal`, managing the physical representation of an indoor environment through geometries which define the physical extent of indoor spaces in accordance with ISO 19107 (ISO/TC 211, 2019) `CellSpace` and `CellBoundary` members. The primal endpoint supports GET, returning the complete set of members conforming to the `PrimalSpaceLayer` schema with optional filtering by `bbox`, `level`, `poi`, `isVirtual`, and `cellSpaceName` query parameters, and POST for adding a new `CellSpace` or `CellBoundary` member strictly conforming to their respective IndoorJSON schemas, with exactly one type permitted per request. Individual members at `···/primal/{memberId}` support GET for retrieving

Resource	Method	Path
Collections	GET, POST	`/collections`
Collection	GET, PATCH, DELETE	`/collections/{cId}`
IndoorFeatures	GET, POST	`/collections/{cId}/items`
IndoorFeature	GET, DELETE	`/collections/{cId}/items/{fId}` ← `(base)`
InterLayerConnections	GET, POST	`(base)/interlayerconnections`
InterLayerConnection	DELETE	`(base)/interlayerconnections/{cId}`
ThematicLayers	GET, POST	`(base)/layers`
ThematicLayer	GET, DELETE	`(base)/layers/{tId}`
PrimalSpace	GET, POST	`(base)/layers/{tId}/primal`
PrimalSpaceMember	GET, DELETE, PATCH	`(base)/layers/{tId}/primal/{mId}`
DualSpace	GET, POST	`(base)/layers/{tId}/dual`
DualSpaceMember	GET, DELETE, PATCH	`(base)/layers/{tId}/dual/{mId}`
Geometric Query	GET	`(base)/layers/{tId}/geoquery`
Routing	GET	`(base)/layers/{tId}/dual/route`
Bounding Query	GET	`(base)/layers/{tId}/primal/{mId}/bounding`
Connected Nodes	GET	`(base)/layers/{tId}/dual/{mId}/connected`

Table 1. Summary of API — IndoorFeatures Endpoints

the detailed representation in either `geometry2D` or `geometry3D` form, DELETE for permanent removal, and PATCH exclusively for `CellSpace` members to partially update non-geometric metadata properties, such as `cellSpaceName`, `poi`, and `boundedBy`, without resubmitting large geometric payloads.

4.5 DualSpace

DualSpace is exposed as a sub-resource of a thematic layer at `···/{featureId}/layers/{tId}/dual`, managing the logical topological network of an indoor environment through `Node` members representing `CellSpace` entities and `Edge` members representing `CellBoundary` entities from the primal space. The dual endpoint supports GET, returning the complete dual graph with optional filtering by `minWeight` and `maxWeight` query parameters to retrieve edges within a specific traversal cost range, and POST for registering a new `Node` or `Edge` conforming to their respective IndoorJSON schemas, with exactly one type permitted per request and each `Edge` required to reference exactly two unique nodes in its `connects` property. Individual members at `···/dual/{memberId}` support GET for retrieving the detailed member representation, DELETE for permanent removal with cascading deletion of all connected `Edge` members and `InterLayerConnection` references when a `Node` is removed, and PATCH exclusively for `Edge` members to dynamically update the `weight` property — representing traversal cost or navigational impedance — without resubmitting the entire dual graph.

4.6 Services

The Services Requirements Class elevates the API beyond data management by defining specialized computational endpoints that execute complex spatial, topological, and routing algorithms directly on the server. While the preceding requirements classes handle the full CRUD lifecycle of IndoorGML entities, this class introduces four analytical services that leverage the primal-dual duality of IndoorGML 2.0 to deliver actionable spatial intelligence.

4.6.1 Geometric Query The Geometric Query endpoint, accessible at `···/{tId}/geoquery` has advanced spatial filtering capabilities beyond standard bounding box searches. Clients supply a mandatory `rel` parameter specifying an OGC Simple Features geometric predicate (e.g., `intersects`, `contains`, `within`), and a mandatory `geometry` parameter encoded in WKT (Herring, 2011) format. An optional `level` parameter restricts evaluation to a specific floor level.

4.6.2 Routing Service The Routing Service endpoint, accessible at `···/dual/route` computes the shortest navigable path between two topological nodes within a specified thematic layer. Clients provide the mandatory `sn` (start node) and `dn` (destination node) parameters that identify the origin and target `Node` members, respectively. The server calculates the path with the lowest cumulative `Edge` weight using a shortest-path algorithm, such as Dijkstra's, commonly implemented via spatial relational database routing engines like pgRouting.

The response contains an ordered `path_segments` array representing the sequential route, where each segment corresponds to a traversed `Edge`, with the final segment identifying the destination `Node`. A key property of this service is that while traversal occurs entirely within the abstract dual space, each node in the result includes its `duality` link back to the corresponding `CellSpace` in the primal space, enabling client applications to translate the computed graph path into actionable physical directions. If no valid path exists between the two nodes, the server returns `200 OK` with an empty `path_segments` array and a total cost of zero, rather than an error.

4.6.3 Bounding Space Query The Bounding Space Query endpoint, accessible at `···/primal/{memberId}/bounding` provides a reverse-topological lookup mechanism. Where the standard IndoorGML 2.0 model defines `CellBoundary` entities as the geometric limits of `CellSpace` entities, this endpoint answers the inverse question: given a specific boundary, which cell spaces does it define? The server validates that the provided `memberId` identifies a `CellBoundary` — requests targeting a `CellSpace` identifier are rejected with `404 Not Found` to enforce strict semantic usage. Each returned `CellSpace` explicitly includes its `duality` property, allowing clients to immediately resolve the corresponding `Node` in the dual space.

4.6.4 Connected Node Query The Connected Node Query endpoint, accessible at `···/dual/{memberId}/connected` provides network reachability analysis within the dual space. Given a source `Node` and a mandatory `hop` parameter specifying the maximum traversal depth, the server performs a recursive traversal of the topological graph. It returns all reachable `Node` members within the specified hop limit. Each discovered node includes a `hop` field indicating its traversal distance from the source, and a `duality` property linking back to the corresponding `CellSpace` in the primal space. This endpoint is particularly suited for identifying adjacent areas or determining spatial coverage scopes depending on the thematic layer in use. The server returns `404 Not Found` if the `memberId` identifies an `Edge` rather than a `Node`.

5. Implementation Challenges

5.1 Dual Identifier Strategy

To balance developer flexibility with query performance, each indoor feature is assigned two identifiers. A string-based ID is accepted at creation time via the POST request payload, allowing users to provide semantically meaningful identifiers that are consistent with their source data. In parallel, the server automatically generates an integer ID. Since integer-based joins and indexed lookups are significantly faster than string comparisons in PostgreSQL, all internal foreign key constraints and spatial queries are resolved using the integer ID. In contrast, the string ID is exposed through the API for client-facing interactions.

5.1.1 Experimental Validation To validate the efficiency of this dual-identifier approach, we conducted comprehensive benchmarks on PostgreSQL 18, which supports UUIDv7. The test environment was configured with `shared_buffers` = 2GB, `work_mem` = 64MB, and `effective_cache_size` = 4GB.

Dataset Generation: Synthetic datasets were generated to enable controlled scaling across the three identifier variants. Each schema variant — INT, STRING, and UUIDv7 — was populated with an identical structure consisting of one collection, one `IndoorFeature`, and one `ThematicLayer`, under which 1,000,000 `CellSpace` rows of type `space` were inserted into `cell_space_n_boundary`, each paired with a dual `Node` in `node_n_edge`, resulting in 1,000,000 node rows per variant. Connectivity was established via approximately 999,999 `connects` rows, forming a chain graph that links all nodes sequentially. Batch insertion was performed throughout using `execute_values` to ensure consistent loading performance across variants. This structure enabled Q1–Q4 to be executed on datasets with identical topology and scale, making identifier type the sole variable.

Query Descriptions The four benchmark queries were designed to evaluate performance across increasing levels of join complexity, from elementary point lookups to full dataset export, reflecting the most common access patterns of the API.

Q1 (Room Lookup) performs single-entity retrieval via a primary key join between `cell_space_n_boundary` and `node_n_edge` on `duality_id`, testing primary key index performance and corresponding to the Feature and PrimalSpace endpoints.

```
SELECT cs.id, cs.cell_name, cs.level, cs.type,
      n.id AS node_id, n.weight
FROM cell_space_n_boundary cs
JOIN node_n_edge n ON n.duality_id = cs.id
WHERE cs.id = %s;
```

Q2 (Neighbor) traverses the `connects` join table from a given `Node` to retrieve directly connected neighbors and resolve their dual `CellSpace` via a left join on `duality_id`, testing foreign key join performance and corresponding to the `services/connected` endpoint.

```
SELECT n2.id, n2.weight, cs.cell_name, cs.level
FROM node_n_edge n1
JOIN connects c ON c.node_source_id = n1.id
JOIN node_n_edge n2 ON n2.id = c.node_target_id
LEFT JOIN cell_space_n_boundary cs ON cs.id =
    n2.duality_id
WHERE n1.id = %s;
```

Q3 (Full IndoorJSON) is a multi-table aggregation joining `cell_space_n_boundary`, `node_n_edge`, and `connects` to retrieve all `CellSpace`, `Node`, and `Edge` members of a collection. This is the most demanding operation in the API—full IndoorJSON export—and the most sensitive to the identifier strategy.

```
SELECT cs.id, cs.cell_name, cs.level, cs.type,
      n.id AS node_id, n.weight,
      c.node_source_id, c.node_target_id, c.edge_id
FROM cell_space_n_boundary cs
JOIN node_n_edge n ON n.duality_id = cs.id
JOIN connects c ON c.node_source_id = n.id
WHERE cs.indoorfeature_id = %s;
```

Q4 (Edge Lookup Performance) isolates edge retrieval for pathfinding, matching the internal query pattern of `services/routing` during graph traversal and `services/bounding` for spatial `CellSpace` lookups. Unlike Q1–Q3, which use a fixed dataset, Q4 scales edges and nodes from 186 edges (269 total nodes and edges) up to 1,000,000, enabling direct comparison of how INT, STRING, and UUIDv7 identifiers degrade as the graph size increases.

```
SELECT c.* FROM node_n_edge n
JOIN connects c ON n.id = c.node_source_id
WHERE n.id_str = %s;
```

5.1.2 Key Findings The results demonstrate that integer-based identifiers consistently outperform string-based identifiers across all query patterns, with performance gains becoming more pronounced at larger dataset scales.

Figure 3 summarizes the benchmark results for Q1–Q3 across all three identifier variants at the 1,000,000-row scale. For Q1 and Q2, all three variants perform comparably at sub-millisecond latency, reflecting the effectiveness of primary key and foreign key indexes for simple lookups and 1-hop traversals. However, Q3 reveals a stark divergence: STRING identifiers incur an average execution time of 5,451.601 ms compared to 1,249.269 ms for INT, representing a $4.36\times$ speedup in favor of integer IDs for complex multi-table aggregation. UUIDv7 falls between the two at 2,857.736 ms. While UUIDv7 shows competitive performance in simple lookups (Q1), it exhibits measurable degradation under complex join and bulk read workloads due to its non-sequential nature reducing index locality.

Figures 4 and 5 present the Q4 edge lookup results across the full scaling range, from 2696 edges and nodes to 999,918. At small dataset sizes, all three variants perform within a comparable range, but divergence becomes significant beyond 10,000 edges. INT identifiers achieve substantially lower execution times and shared buffer hit counts than STRING at a large scale. While UUIDv7 did not deviate far from the execution times of the INT identifiers, differences were evident in larger-scale experiments in shared buffer hit counts; in Figure 5, the shared buffer hit counts confirm that INT identifiers require the lowest buffer accesses, indicating more efficient index utilization during graph traversal.

Together, these results validate the dual-identifier strategy described in Section 5.1: integer IDs are used exclusively for internal joins and graph operations to maximize query performance, while string identifiers are preserved at the API layer to maintain developer-facing flexibility and compatibility with IndoorJSON.

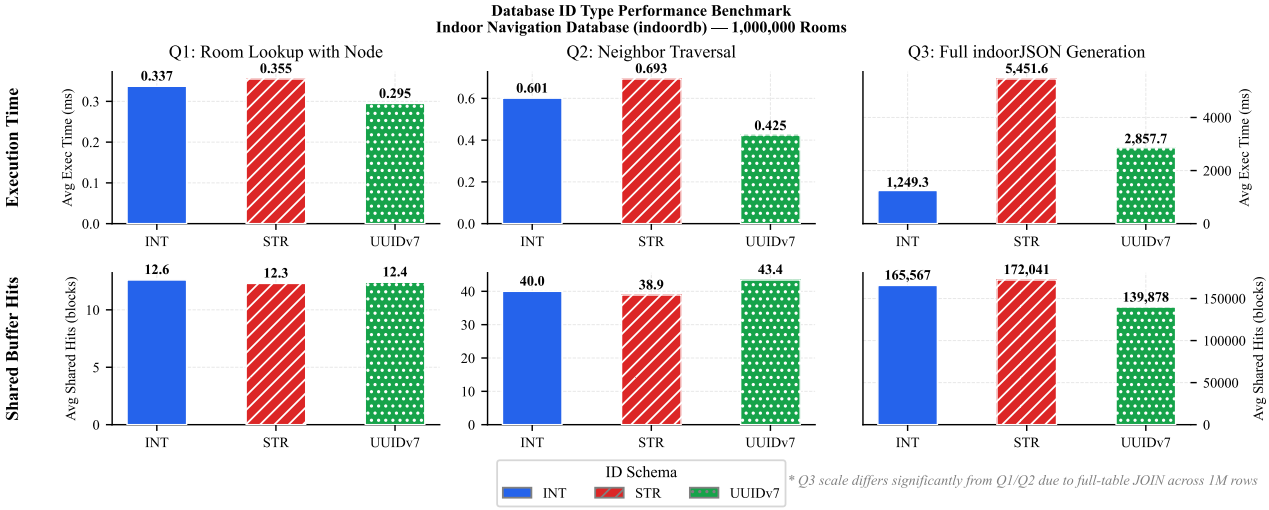

Figure 3. The experiment results performed on Q1, Q2, and Q3 for different Indexes

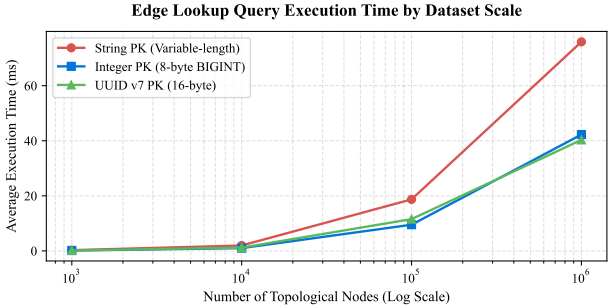

Figure 4. The execution time (ms) for different indexes

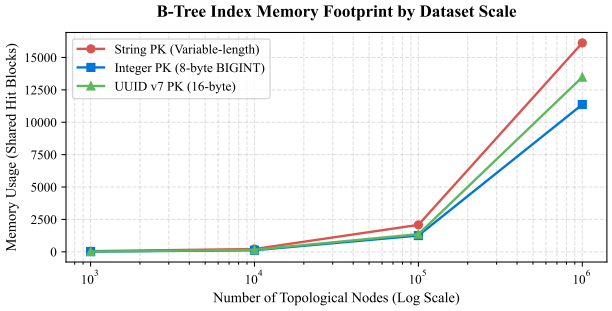

Figure 5. The shared buffer hits for different indexes

5.2 Cascading Delete and Update

Not only was the database schema sensitive to its ID typing. Maintaining data integrity across the hierarchical and topological structure of IndoorGML 2.0 required careful design of cascading rules for create, update, and delete operations, with particular attention to the duality relationships between `CellSpace`–`Node` and `CellBoundary`–`Edge` pairs.

For deletions, any removal of a higher-level resource must cascade to all subordinate members. Deleting a `ThematicLayer`, for instance, triggers cascading removal of its entire `PrimalSpace` and `DualSpace` contents. The same applies at higher levels: removing a collection or feature propagates deletion downward through all associated layers and their members. Other considerations made are as follows.

5.2.1 InterLayerConnection On creation, the API validates that all `connectedLayers` reference existing `ThematicLayer` IDs, and that any referenced `Node` or `CellSpace` identifiers are valid within the specified feature.

5.2.2 PrimalSpace On creation, if a `duality` field is given for a `CellSpace`, the API verifies that the referenced dual `Node` exists. The `boundedBy` field is similarly validated to ensure all referenced `CellBoundary` IDs are present. On deletion of a `CellSpace`, its dual `Node` is cascadingly deleted, and any associated `InterLayerConnection` referencing that node is also removed. When a `CellBoundary` is deleted, its dual `Edge` is not mandatorily removed, but any `CellSpace` referencing it in its `boundedBy` field must be updated accordingly, along with any affected `InterLayerConnection` entries. This is because an `Edge` can exist on its own without its `CellBoundary` duality, but not vice versa for the `Node`.

5.2.3 DualSpace On creation, a `Node` must declare a duality relationship — for the same reason mentioned above — and the API updates the corresponding `CellSpace` duality field upon successful creation. For `Edge` members, duality is optional, but if declared, the same validation and back-update process applies to the referenced `CellBoundary`. The `connects` field must reference at least two valid `Node` IDs, and the API updates the `connects` field on each referenced `Node` accordingly. On deletion of a `Node`, its associated `Edge` members and `InterLayerConnection` entries are cascadingly removed, and the duality field of its corresponding `CellSpace` is cleared. Deleting an `Edge` triggers an update to the `connects` field of its referenced `Node` members. If a duality exists, the corresponding `CellBoundary` is updated as well.

6. Use case: PNU building 201

To validate the practical applicability of API – IndoorFeatures, a prototype client application was developed using a real dataset captured from Building 201 at Pusan National University (PNU). Rather than relying on synthetic data, the dataset captures the building's actual spatial structure. It is encoded in IndoorJSON, making it a representative testbed for evaluating the API's core functionality.

The dataset comprises a single `ThematicLayer` (`TL1`) with a Physical theme that spans three floor levels. The `PrimalSpace`

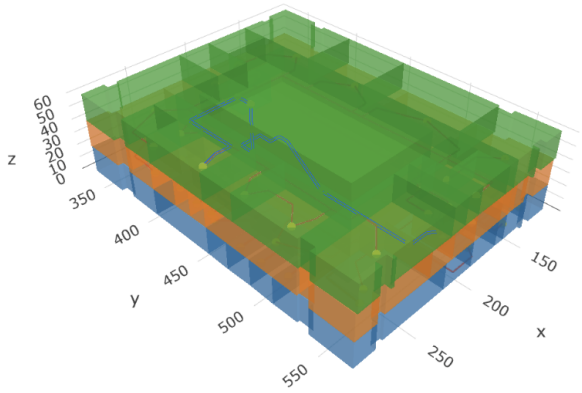

Figure 6. Visualization of the routing service result for PNU Building 201

contains 80 `CellSpace` members distributed across the floors — 28 on Level 1, 31 on Level 2, and 21 on Level 3 — along with 184 `CellBoundary` members representing the physical boundaries between spaces. The textttDualSpace encodes the building's navigational topology as a graph with 80 `Node` members and 186 `Edge` members. Note that not all PrimalSpaceMembers necessitate a duality, reflecting the duality relationship between the primal and dual representations defined in IndoorGML 2.0.

All endpoints defined in Section 4 were tested against this dataset with additional datasets. As demonstrated in Figure 6, the routing result confirms that the DualSpace graph and pgRouting integration operate correctly on real building data. Furthermore, not only the base IndoorGML 2.0 endpoints, but the endpoints defined in 4.6 were verified and optimized.

7. Conclusion

This paper presented the API — IndoorFeatures, a RESTful API designed to provide standardized and developer-friendly access to indoor spatial data as summarized in Table 1. By adopting IndoorJSON as its primary serialization format — a lightweight JSON-based profile of IndoorGML 2.0 — the API significantly lowers the barrier for developers seeking to integrate indoor mapping capabilities into web and mobile applications, without sacrificing the semantic richness of the underlying IndoorGML 2.0 data model.

The API is designed in conformance with the OGC API — Features standard, inheriting its resource patterns and HTTP interaction conventions. This deliberate alignment ensures compatibility with existing OGC API — Features clients and allows implementations to be built on top of conformant server frameworks such as pygeoapi without modification to the core API layer. The resource model faithfully preserves the fundamental IndoorGML 2.0 duality between primal and dual space across all requirements classes, from the management of `CellSpace` and `CellBoundary` entities in the PrimalSpace, to the topological graph of `Node` and `Edge` members in the DualSpace, through to the inter-layer relationships expressed via `InterLayerConnection`. Beyond data management, the Services Requirements Class further extends the API with specialized computational endpoints for geometric querying, indoor routing, reverse-topological lookup, and network reachability analysis.

Despite these contributions, several limitations remain. The mapping between IndoorJSON constructs and their IndoorGML 2.0 counterparts is still undergoing validation, and full semantic equivalence has not yet been formally verified. Furthermore, the current specification does not address authentication, authorization, or multi-user conflict resolution, which would be necessary for production deployments.

Future work will focus on broader adoption of the API by developers and industry practitioners, to gather real-world implementation feedback to refine and mature the specification. Community engagement through open-source tooling and developer documentation will be a priority to drive uptake and establish IndoorJSON as a practical standard for web-based indoor spatial data exchange.

References

Egenhofer, M. J., 1989. A formal definition of binary topological relationships. W. Litwin, H.-J. Schek (eds), *Foundations of Data Organization and Algorithms (FODO 1989)*, Lecture Notes in Computer Science, 367, Springer, 457–472.

Herring, J. R., 2011. OpenGIS Implementation Standard for Geographic Information - Simple Feature Access - Part 1: Common Architecture. OGC Implementation Standard OGC 06-103r4, Open Geospatial Consortium.

ISO/TC 211, 2019. ISO 19107:2019 Geographic information — Spatial schema. International Standard ISO 19107:2019, International Organization for Standardization, Geneva, Switzerland.

Jeong, H., Ryoo, H., Li, K.-J., 2018. INFACTORY: A RESTful API Server for Easily Creating IndoorGML. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-4/W8, 77–84. doi.org/10.5194/isprs-archives-XLII-4-W8-77-2018.

Kim, T., Diakite, A. A., Zlatanova, S., Li, K.-J., 2026. IndoorJSON schema and sample data repository. github.com/opengeospatial/IndoorGML-SWG. OGC IndoorGML Standards Working Group, draft in progress.

OpenAPI Initiative, 2018. OpenAPI Specification 3.0.2. Version 3.0.2.

pgRouting Project, 2024. pgRouting 3.7.3 Manual. Version 3.7.3.

Portele, C., Vretanos, P. A., Heazel, C., 2022. OGC API - Features - Part 1: Core corrigendum. OGC Implementation Standard 17-069r4, Open Geospatial Consortium.

PostGIS Project Steering Committee, 2025. PostGIS 3.5 Manual. Version 3.5.

The PostgreSQL Global Development Group, 2025. PostgreSQL 17.5 Documentation. Version 17.5.

The pygeoapi team, 2026. pygeoapi 0.23.4 Documentation. Version 0.23.4.

Zlatanova, S., Diakite, A. A., Kim, T., Li, K.-J., 2025. OGC IndoorGML 2.0 Part 1 – Conceptual Model. OGC Standard OGC 22-045r5, Open Geospatial Consortium.