Federated Database (OGBv4) API



Figure 1 – Storage model of the federated database

Figure 1 shows the storage model of the NoSQL ICN federated spatial database. The storage space is formed of the storage spaces offered by the differed sites of the federation (e.g. one site in EU and one site in JP).

The Storage model provides that the storage space is divided per tenant and per collection.

A tenant is a group-administrator that has a slice of the storage space. Each tenant has users (e.g. Bob) which can carry out CRUD operation only for the spatial objects of the tenant slice. Tenant data can be spread over all the sites of the federation. E.g. the tenant #1 can have Japanese users that store their spatial object in a Japanese site, and EU users that store their spatial object in an EU site. The system procedures, supported by ICN, will expose these two physically separated storage spaces as a single one.

Tenant slice is organized in collections, like in MongoDB. A collection is a logical partition created for storing spatial objects that have relationships with each other from the programmer point of view, e.g. a collection could contain spatial objects are used by the same applications. A collection is similar to a SQL table but can contain spatial objects with a different schema, i.e. having different properties. A proper design of the collection can accelerate query operations since they search within a collection.

The spatial objects stored in OGB must follow a GeoJSON schema (geojson.org), such as

{ "type": "Feature",

"geometry": {

"type": "Point",

"coordinates": [125.6, 10.1]

},

"properties": {

"name": "Dinagat Islands"

}}

Supported geometries are Point, Multipoint and Polygon. Type and Geometry keys have standardized keywords (including upper and lower cases characters). Within the properties key, the user can insert one or more property of interest, defining its keys.

OGB supports object and spatial queries that can be executed from any federation site and search in the whole federation storage space.

Object queries request for a single object identified by a unique object identifier (oid) that is chosen by the OGB system and sent back to the user after an insertion. Spatial objects are cached by ICN nodes, thus object queries can provide stale data. For instance after a deletion, an object query can still provide the deleted object because the deletion operation only cancels the data item in the origin site but not in the ICN nodes of the networks, whose caches can send back the object. Consequently object queries should not be used for production application.

Range queries search for spatial objects (of a given tenant and collection) which intersect a polygon area specified in the query statement. OGB mechanisms assure that responses to range queries are always fresh, i.e. will never contain stale data. Range query is the recommended query operation.

Note1: to speed up Insert/Delete operations, OGB follows a kind of Fire-and-Forget interaction model for which the system immediately provide you a positive answer closing the HTTP connection if the request is formally correct. Then the Insert/Delete operation is executed in background. Sometimes, it my happen that these background operations will not succeed and so the Insert/Delete operation actually fails. A range query can confirms the effective successfulness of the operation.

Note2: OGB carry out query routing which exploits a distributed spatial indexing mechanism based on ICN. After an Insert operation the spatial index could need an update and the inserted data item will be effectively available only after that the index update process terminates. Roughly, index update takes few tens of ms for computation plus **three round trip times** for index synchronization among sites.

## Identifiers

oid : unique name that identify an object in the system

tenantName: unique identifier of the tenant

userName: unique identifier of the user of a given tenant, Every user has a password to access the federation

cid: collection identifier, a name that identifies the collection

## OGB HTTP API

The OpenGeoBase federated database provides a HTTP interface.

### Login

This method allows the user to log in.

* **URL:** /OGB/user/login
* **Method:** POST
* **consumes:** application/json
* **produces:** application/json
* **POST body:** a JSON object with the following body :

{

"userName" : <the username of the user that attempts to login>,

"password" : <the password of the user>,

"tenantName" : <the tenant responsible for the user>

}

* **Response:** the **users** **authentication token** in case of success, an error otherwise with following code:
  + 407,"Error on login: Wrong credential provided!"

### Object insertion

This method allows the user to insert a GeoJSON object. Collection id must be the last path component in the method entry point

* **URL:** /OGB/content/insert/{cid}
* **Method:** POST
* **consumes:** application/json
* **produces:** application/json
* **POST header:** the following key:value :

Authorization : [the authorization token of the user]

* **POST body:** a GeoJSON object that describes the geospatial structure to be inserted in OGB. Supported geometry types are the following: Point, Polygon, MultiPoint. Geometric objects with additional properties are Feature objects.

The following text shows an example of the body for a **Point** GeoJSON object (remove the comments to have a valid json):

{  
 "geometry": {  
 "coordinates": [12.28,41.63], // coordinates in format longitude, latitude  
 "type": "Point" // supported geometry types are Point, Polygon, MultiPoint  
 },  
 "type": "Feature",  
 "properties": { // additional properties  
 "train\_speed" : "10",  
 "delay" : "0"  
 }  
}

The following text shows an example of the body for **MultiPoint** GeoJSON object (remove the comments to have a valid json):

{  
 "geometry": {  
 "coordinates": [ //For type "MultiPoint", the "coordinates" property is an array of positions.  
 [12.536976,41.950308],  
 [12.459226,41.937373],  
 [12.363410,41.914593]  
 ],  
 "type": "MultiPoint"  
 },  
 "type": "Feature",  
 "properties": {  
 "provider": "ICN2020 Project",  
 "format": "GTFS",  
 "URL" : "http:/myurl.it/gtfs.zip"  
 }  
}

The following text shows an example of the body for **Polygon** GeoJSON object which has a box shape (remove the comments to have a valid json):

{

"geometry" : {

"type" : "Polygon",

"coordinates" : [[

[12.41,41.01], //Polygon point 1 coordinate [latitude, longitude]

[12.51,41.01], //Polygon point 2 coordinate [latitude, longitude]

[12.51,41.11], //Polygon point 3 coordinate [latitude, longitude]

[12.41,41.11], //Polygon point 3 coordinate [latitude, longitude]

[12.41,41.11] //Polygon last point [latitude, longitude]. **Must be equal to point 1**

]]

},

"type":"Feature",

"properties" : { // object properties

"name": "Italy tile"

}

}

* **Response:** the method returns a string representing the Object Identifier (OID) of the inserted GeoJSON object, an error message otherwise with following codes:
  + 420, "Invalid authorization token"
  + 421, "Invalid permission type "
  + 407, "Failed to upload Content!"
  + 407, "Error on upload Content! Empty Content!"

### Object query

This method allows the user to retrieve a GeoJSON by its ObjectID. Collection id must be the last path component in the method entry point

* **URL:** /OGB/query-service/element/{cid}
* **Method:** POST
* **consumes:** application/json
* **produces:** application/json
* **POST header:** an HashMap with the following key:value :
  + Authorization : [the authorization token of the user retrieved by login procedure]
* **POST body:** a JSON object with the following body :

{

"oid": <the object identifier (OID) of the requested GeoJSON>

}

* **Response:** the requested GeoJSON object as string if it exists into database, an error otherwise with following codes:
  + 420, "Invalid authorization token"
  + 431, "Invalid oid in GeoJSON!"

### Range query

This method allows the user to find GeoJSON objects within a specified area. Collection id must be the last path component in the method entry point

* **URL:** /OGB/query-service/{cid}
* **Method:** POST
* **consumes:** application/json
* **produces:** application/json
* **POST header:** an HashMap with the following key:value :
  + Authorization : [the authorization token of the user]
* **POST body:** a JSON object that describes the geospatial query. Currently only "$geoIntersects" geospatial operator and $geometry.type Polygon is supported. More information at https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/

The following text shows an example of the body of a range query (remove the comments to have a valid json):

{

"geometry": {

"$geoIntersects": {

"$geometry": {

"type": "Polygon" , // supported geometry type is only "Polygon"

"coordinates":[[

[12.27,41.62], // first point longitude, latitude

[12.73,41.62], // second point longitude, latitude

[12.73,42.02], // third point longitude, latitude

[12.27,42.02], // fourth point longitude, latitude

[12.27,41.62] // last point longitude, latitude. **MUST be equal to first point**

]]

}

}

}

}

* **Response:** an array of GeoJSON objects (as string) within the specified area (if no GeoJSON is present in the area the method returns an empty array). In case of failure the following error codes are returned:
* 420, "Invalid authorization token"
* 430, "Invalid query params"
* 440, "Requested area size exceeds the maximum limit"

### Object removal

This method allows the user to delete a GeoJSON object

* **URL:** /OGB/content/delete
* **Method:** POST
* **consumes:** application/json
* **produces:** application/json
* **POST header:** an HashMap with the following key:value :
* Authorization : [the authorization token of the user]
* **POST body:** a JSON object with the following body :

{

"oid": <the object identifier (OID) of the requested GeoJSON>

}

* **Response:** the method returns an empty response with status code 200 in case of success, an error otherwise with following codes:
  + 420, "Invalid authorization token"
  + 431, "Invalid oid in GeoJSON!"
  + 403, "User unauthorized!"
  + 421, "Security issues: User grant not retrieved!"

### Register (NOT use)

This method allows to register a new user into the OpenGeoBase system.

* **URL:** /OGB/user/register
* **Method:** POST
* **consumes:** application/json
* **produces:** application/json
* **POST body:** a JSON object with the following keys:values :

{

“username” : <the username of the new user>

“password” : <the password of the new user>

“tenantName” : <the tenant responsible for the new user>

“permission” : <the permission type for the new user ("r" or "rw")>

}

* **Response:** an empty response with status code 200 in case of success, an error otherwise with following code:
  + 407, "Register Failed"

## OGB JAVA Library

This is a java library developed for OpenGeoBase frontend. To use this library in your JAVA project, export OgbJavaLibrary/src/com/ogb/clientlib/OgbClientLib.java  
 as runnable jar file and add it to the build path of your project. Then use

import com.ogb.clientlib.\*;

### Library object constructor

Create a new OgbClientLib with optional settings to manage communication with the OGB FrontEndServer.

**OgbClientLib(String serverURL)**

* **Parameters:**
* String serverURL : URL of the FrontEndServer (ip:port)

The following code shows an example of object allocation:

OgbClientLib ogbTestClient = new OgbClientLib(serverURL);

### Login

This method allows the user to log in.

*String* **login**(String userId, String tenant, String password)

* **Parameters:**
* *String* userId : the username of the user that attempts to login
* *String* tenant : the tenant responsible for the user
* *String* password : the password of the user
* **Response:**
* (*String*) secure token to be used for next operations with the FrontEndServer. This token identify the user. keep it secret.

The following code shows an example of login:

String uid = "myUserID";  
String tid = "myTenantID";  
String pwd = "myPassword";  
String token = ogbTestClient.login(uid, tid, pwd);

### Point insertion

This method allows the user to insert a **Point** GeoJSON object.

*String* **addPoint** (String token, String cid, HashMap < String, String > propertiesMap, final double[ ] location)

* **Parameters:**
* *String* token : the authorization token of the user
* *String* cid : collection identifier
* *HashMap < String, String >* propertiesMap : set of geo-json properties, hashmap < string properties, string property\_value >
* *double[ ]* location : [latitude longitude] double array
* **Response:**
  + (*String*) object identifier (OID) of inserted GeoJSON
* **Example**

The following code shows an example of **Point** GeoJSON object with additional properties insertion:

// point coordinates  
 double lat = 0.1;  
 double lon = 0.2;  
 double [] coordinates = {lat, lon};  
 // point properties  
 HashMap<String,String> prop = new HashMap<String,String>();  
 prop.put("train-name", "ice-374");   
 prop.put("train-speed", "170 km/h");  
 // db insertion, response is the object identifier (oid)  
 String oid = ogbTestClient.addPoint(token,cid, prop, coordinates);

### MultiPoint insertion

This method allows the user to insert a **MultiPoint** GeoJSON object.

*String* **addMultiPoint**(String token, String cid, HashMap < String, String > propertiesMap, ArrayList < double[ ] > coordinates)

* **Parameters:**
  + *String* token : the authorization token of the user
  + *String* cid : collection identifier
  + *HashMap < String, String >* propertiesMap : set of geo-json properties, hashmap
  + *ArrayList < double[ ] >* location : array list of [latitude longitude] double arrays
* **Response:**
  + (*String*) object identifier (OID) of inserted GeoJSON

The following code shows an example of **MultiPoint** GeoJSON object with additional properties insertion:

// multipoint coordinates  
 ArrayList<double[]> mcoordinates = new ArrayList<double[]>();  
 double lat\_point1 = 0.01; //point 1 latitude  
 double lon\_point1 = 0.01; //point 1 longitude  
 double lat\_point2 = 0.02; //point 2 latitude  
 double lon\_point2 = 0.02; //point 2 longitude  
 mcoordinates.add(new double[] { lat\_point1, lon\_point1 }); //point 1  
 mcoordinates.add(new double[]{lat\_point2, lon\_point2 }); // point 2  
 // point properties  
 HashMap<String,String> mprop = new HashMap<String,String>();  
 mprop.put("prop100", "value100");   
 mprop.put("prop200", "value200");  
  
 // DB insertion, response is the object identifier (oid)  
 String moid = ogbTestClient.addMultiPoint(token,cid, mprop, mcoordinates);

### Polygon insertion

This method allows the user to insert a **Polygon** GeoJSON object.

*String* **addPolygon**(String token, String cid, HashMap < String, String > propertiesMap, ArrayList < double[ ] > coordinates)

* **Parameters:**
  + *String* token : the authorization token of the user
  + *String* cid : collection identifier
  + *HashMap < String, String >* propertiesMap : set of geo-json properties, hashmap
  + *ArrayList < double[ ] >* location : array list of [latitude longitude] double arrays
* **Response:**
  + (*String*) object identifier (OID) of inserted GeoJSON

The following code shows an example of **Polygon** GeoJSON object with additional properties insertion:

ArrayList<double[]> pcoordinates = new ArrayList<double[]>();  
 double polygon\_lat\_point1 = 0.00; //point 1 latitude  
 double polygon\_lon\_point1 = 0.00; //point 1 longitude  
 double polygon\_lat\_point2 = 0.11; //point 2 latitude  
 double polygon\_lon\_point2 = 0.11; //point 2 longitude  
 double polygon\_lat\_point3 = 0.11; //point 3 latitude  
 double polygon\_lon\_point3 = 0.00; //point 3 longitude  
 pcoordinates.add(new double[] { polygon\_lat\_point1, polygon\_lon\_point1 });   
 pcoordinates.add(new double[] { polygon\_lat\_point2, polygon\_lon\_point2 });  
 pcoordinates.add(new double[] { polygon\_lat\_point3, polygon\_lon\_point3});  
 // point properties  
 HashMap<String,String> polygonProp = new HashMap<String,String>();  
 polygonProp.put("URL", "http:/myurl.it/gtfz.zip");   
 polygonProp.put("Type", "GTFS");  
 polygonProp.put("Provider", "ICN2020 Project");  
  
 // db insertion, response is the object identifier (oid)  
 String poid = ogbTestClient.addPolygon(token,cid, polygonProp, pcoordinates );

### GeoJSON insertion

This method allows the user to insert a GeoJSON object. Supported GeoJSON shape are Point, MultiPoint and Polygon.

*String* **addGeoJSON**(String token, String cid, String geoJSON)

* **Parameters:**
* *String* token : the authorization token of the user
* *String* cid : collection identifier
* *String* geoJSON : string representing geoJSON object

The following text shows an example of **Point** GeoJSON object with additional properties:

{  
 "geometry": {  
 "coordinates": [12.28,41.63], // coordinates in format longitude, latitude  
 "type": "Point" // supported geometry types are Point, Polygon, MultiPoint  
 },  
 "type": "Feature",  
 "properties": { // additional properties  
 "train\_speed" : "10"  
 "delay" : "0"  
 }  
}

Following text shows an example of **MultiPoint** GeoJSON object with additional properties:

{  
 "geometry": {  
 "coordinates": [ //For type "MultiPoint", the "coordinates" member must be an array of positions.  
 [12.536976,41.950308],  
 [12.459226,41.937373],  
 [12.363410,41.914593],  
 ],  
 "type": "MultiPoint"  
 },  
 "type": "Feature",  
 "properties": {  
 "Provider", "ICN2020 Project",  
 "Type", "GTFS",  
 "URL" : "http:/myurl.it/gtfs.zip"  
 }  
}

Following text shows an example of **Polygon** GeoJSON object with additional properties:

{  
 "geometry" : {  
 "type" : "Polygon",  
 "coordinates" :  
 [  
 [  
 [0.0, 0.0], //Polygon point 1 coordinate [latitude, longitude]  
 [0.11, 0.11], //Polygon point 2 coordinate  
 [0.0, 0.11] //Polygon point 3: if needed, the API method closes Polygon automatically connecting last and first points  
 ]  
 ]  
 },  
 "type":"Feature",  
 "properties" : { // object properties  
 "name": "null island"  
 }  
}

* **Response:**
  + (*String*) object identifier (OID) of inserted GeoJSON

### Object query

This method allows the user to retrieve a GeoJSON by its ObjectID.

*String* **queryObject**(String token, String cid, String oid)

* **Parameters:**
* *String* token : the authorization token of the user
* *String* cid : collection identifier
* *String* oid : unique identifier of the geoJSON object
* **Response:**
* (*String*) the requested GeoJSON object as string

The following code shows an example of object query:

String oid = "/OGB/000/000/21/00/GPS\_id/GEOJSON/icn2020/testCID/test/q37871900e8w1r69";  
 System.out.println("\n\n\*\*\*\* Object Query \*\*\*\*");  
 String response1 = ogbTestClient.queryObject(token, cid, oid);  
 System.out.println("query response: " + response1);

### Range query

This method allows the user to find GeoJSON objects within a specified **square** area. This method perform GeoJSON spatial query using square **box** shape and $geoIntersects geospatial operator. For more information visit [[1]](#footnote-1).

*String* **rangeQuery**(String token, String cid, double sw\_lat, double sw\_lon, double boxSize)

* **Parameters:**
* *String* token : the authorization token of the user
* *String* cid : collection identifier
* *double* sw\_lat : south west latitude
* *double* sw\_lon : south west longitude
* *double* boxSize : box edge size in degree
* **Response:**
* (*String*) a JSON array of GeoJSON objects within the specified area (if no GeoJSON is present in the area the method returns an empty array).

The following code shows an example of query:

double sw\_lat = 0.0; // south west latitude in degree   
 double sw\_lon = 0.0; // south west longitude in degree   
 double boxSize = 0.5; // box size in degree  
 String response = ogbTestClient.rangeQuery(token, cid, sw\_lat, sw\_lon, boxSize);  
 System.out.println("query response: " + response);

### Range query Box

This method allows the user to find GeoJSON objects within a spcified area. This method perform GeoJSON spatial query using **box** rectangular shape and $geoIntersects geospatial operator. For more information visit [[2]](#footnote-2).

*String* **rangeQueryBox**(String token, String cid, double sw\_lat, double sw\_lon, double ne\_lat, double ne\_lon)

* **Parameters:**
* *String* token : the authorization token of the user
* *String* cid : collection identifier
* *double* sw\_lat : south west latitude
* *double* sw\_lon : south west longitude
* *double* ne\_lat : north east latitude
* *double* ne\_lon : north east longitude
* **Response:**
  + (String) a JSON array of GeoJSON objects within the specified area (if no GeoJSON is present in the area the method returns an empty array).

The following code shows an example of box query:

double sw\_lat = 0.0; // south west latitude in degree   
 double sw\_lon = 0.0; // south west longitude in degree  
 double ne\_lat = 0.5; // north east latitude in degree   
 double ne\_lon = 0.5; // north east longitude in degree  
 String response = ogbTestClient.rangeQueryBox(token, cid, sw\_lat, sw\_lon, ne\_lat, ne\_lon);  
 System.out.println("query response: " + response);

### Range query Polygon

This method allows the user to find GeoJSON objects within a spcified area. This method performs GeoJSON spatial query using **Polygon** shape and $geoIntersects geospatial operator. For more information visit[[3]](#footnote-3).

*String* **rangeQueryPolygon**(String token, String cid, ArrayList < ArrayList < double [ ] > > coordinates)

* **Parameters:**
* *String* token : the authorization token of the user
* *String* cid : collection identifier
* *ArrayList < ArrayList < double [ ] > >* coordinates : ArrayList of ArrayList<[latitude longitude]> double arrays, at the moment supporting only polygon without holes (for more information visit http://geojson.org/geojson-spec.html#id4)
* **Response:**
  + (String) a JSON array of GeoJSON objects within the specified area (if no GeoJSON is present in the area the method returns an empty array).

The following code shows an example of Polygon query:

ArrayList<ArrayList<double[]>> polygon = new ArrayList<>();  
 ArrayList<double[]> subPolygon = new ArrayList<>();   
 double queryPol\_lat\_point1 = 0.00; //point 1 latitude in degree  
 double queryPol\_lon\_point1 = 0.00; //point 1 longitude in degree  
 double queryPol\_lat\_point2 = 0.15; //point 2 latitude in degree  
 double queryPol\_lon\_point2 = 0.15; //point 2 longitude in degree  
 double queryPol\_lat\_point3 = 0.15; //point 3 latitude in degree  
 double queryPol\_lon\_point3 = 0.00; //point 3 longitude in degree  
 subPolygon.add(new double[] {queryPol\_lat\_point1,queryPol\_lon\_point1});  
 subPolygon.add(new double[] {queryPol\_lat\_point2,queryPol\_lon\_point2});  
 subPolygon.add(new double[] {queryPol\_lat\_point3,queryPol\_lon\_point3});  
 polygon.add(subPolygon);  
 String response = ogbTestClient.rangeQueryPolygon(token, cid, polygon);  
 System.out.println("query response: " + response);

### Object removal

This method allows the user to delete a GeoJSON object

*boolean* **deleteObject**(String token, String oid)

**Parameters:**

* *String* token : the authorization token of the user
* *String* cid : collection identifier
* *String* oid : the object identifier (OID) of the GeoJSON to remove

**Response:**

* (*Boolean*) true/false if success/failure

The following code shows an example of object removal:

String oid = "/OGB/000/000/21/00/GPS\_id/GEOJSON/icn2020/testCID/test/q37871900e8w1r69";  
 boolean removalStatus = ogbTestClient.deleteObject(token, oid);

1. https://docs.mongodb.com/manual/reference/operator/query/geoIntersects [↑](#footnote-ref-1)
2. https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/ [↑](#footnote-ref-2)
3. https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/ [↑](#footnote-ref-3)