Denodo GeoJSON Service - User Manual

Revision 20201111

Denodo Inc.: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA www.denodo.com

CONTENTS

1 OVERVIEW

GeoJSON is a geospatial data interchange format based on JavaScript Object Notation (JSON). It defines several types of JSON objects and the manner in which they are combined to represent data about geographic features, their properties, and their spatial extents. GeoJSON uses a geographic coordinate reference system, World Geodetic System 1984, and units of decimal degrees.

GeoJSON supports the following geometry types:

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon

Geometric objects with additional properties are `Feature` objects. Sets of features are contained by `FeatureCollection` objects.

The Denodo GeoJSON Service is a web service that can be used to query the views within a Virtual DataPort Server and return GeoJSON objects of the results. The endpoints of the service can be used by applications seeking GeoJSON objects from VDP results. If the view contains a designated field of geometric type, it will be incorporated in the GeoJSON object as a geometry type.

2 EXECUTION

The Denodo GeoJSON Service distribution consists of:

- A bin folder containing `denodo-geojson-service-<VERSION>.jar`
- A config folder containing the `application.properties` and the `log4j2.xml`
- A bin folder with the necessary scripts to launch the application.

For running the Denodo GeoJSON Service, you need to execute the `denodo_geojson_service.[bat|.sh]`

The Denodo GeoJSON Service has some default properties that can be reset:

- `default.rowlimit`: number of returned entries per request (see Pagination section for more information)
- `cors.allowed-origins`: List of origins to allow CORS access to
- `enable.adminUser`: false if access to the GeoJSON service is denied to the VDP admin user.

The configuration file `BOOT-INF/classes/application.properties` has the values for the properties explained above and can be edited as desired:

```
# Number of rows to return by default
default.rowlimit=10

# JNDI data source
spring.datasource.jndi-name=java:comp/env/jdbc/VDPdatabase

# Enable CORS by specifying a list of urls or * for all urls
cors.allowed-origins=*

# Enable or disable the use of ‘admin’ user when connecting to VDP
enable.adminUser=false
```

Once the application is deployed you can use the Denodo GeoJSON Service from a web client, using **HTTP Basic Authentication** with VDP-valid credentials. URLs are of the form:

```
http://localhost:8999/geojson/<DBNAME>
```

You can also configure the port of the service in the file `application.properties`.

3 ENABLING CROSS-ORIGIN RESOURCE SHARING (CORS)

Cross-origin resource sharing (CORS) is a mechanism whereby a browser and server interact to determine whether to allow a web page to perform HTTP requests to a domain other than the domain from which the web page is originated. I.e. so the browser can determine if it is safe that a page served from the domain `http://foo.com` can send requests to a server in the domain `http://bar.com`.

Once deployed the Denodo GeoJSON Service, follow these steps to enable the CORS support:

1. Edit the application.properties file which is at the `config` folder.
2. Change the following parameter's value:

```
cors.allowed-origins=*
```

***** represents that all origins are allowed.

You can specify a list of urls separated by comma to override this default value. For example:

```
cors.allowed-origins=http://foo.com,http://bar.com
```

4 FEATURES

The GeoJSON Denodo Service provides the following functionality:

- **Read-only** access to Denodo databases
- Query options
    - `$select`
    - `$filter`
    - `$orderby`
    - `$groupby`
    - `$start_index`
    - `$count`
- Pagination
- HTTP Authentication
    - Basic

5 QUERYING DATA: THE BASICS

There are three types of GeoJSON service endpoints:

1. Database Endpoints
2. View Endpoints
3. View Count Endpoints

5.1 DATABASE ENDPOINTS

You can receive a JSON object containing information from a VDP database by specifying a virtual database name after the root of the service with the following pattern:

```
.../geojson/<DBNAME>
```

The GeoJSON service returns a GeoJSON object that lists metadata of all the views accessible in the virtual database by the GeoJSON service. This metadata includes the folder the view is in, the name of the database it is in, the endpoint that can be appended to the URL to access the view data, the description of the view as set in the Virtual DataPort server metadata, and the view name.

Below there is an example of the returned GeoJSON object with the accessible views of a database, test is the name of this database: bv_corp_offices, bv_dist_centers, and bv_retail.

```
.../geojson/test
```

```
{
    "views": [
      {
           "folder": "/02 - base views",
            "database_name": "test",
            "end_point": "/views/bv_corp_offices",
            "description": "Base view created over a remote data
             source containing corporate office location info.",
            "view_name": "bv_corp_offices"
      },
      {
            "folder": "/02 - base views",
            "database_name": "test",
            "end_point": "/views/bv_dist_centers",
```

```
            "description": "Base view created over a remote data
            source containing distribution center location info.",
            "view_name": "bv_dist_centers"
      },
      {
            "folder": "/02 - base views",
            "database_name": "test",
            "end_point": "/views/bv_retail",
            "description": "Base view created over a remote data
                source containing retail seller location info.",
            "view_name": "bv_retail"
      }
  ]
}
```

5.2 VIEW ENDPOINTS

You will receive a GeoJSON `FeatureCollection` object containing data from a VDP view by specifying the view name after the root of the service and database with the following the pattern:

```
.../geojson/<DBNAME>/views/<VIEWNAME>
```

Each tuple of the results will be added as an individual `Feature` of the GeoJSON `FeatureCollection` returned and each field in the tuple will be added as an individual property of that `Feature`. Fields of the view suffixed with "`wkb`" or "`wkt`" (case insensitive), denoting `well-known binary` and `well-known text` respectively, will be automatically processed as geometric types in the GeoJSON output if no geometry field is denoted using the **`$geometry`** parameter described below.

For example, if a view called `bv_corp_offices` view has a field named "`location_wkt`" and we access the following URL:

```
.../geojson/test/views/bv_corp_offices
```

The response would be rendered as:

```
{
"features": [
      {
          "geometry": {
              "coordinates": [
                  -122.159285,
                  37.448861
```

```
                ],
                "type": "Point"
            },
            "type": "Feature",
            "properties": {
                "zipcode": "94301",
                "country": "United States",
                "city": "Palo Alto",
                "street": "525 University Avenue #31",
                "name": "Denodo HQ",
                "id": "1",
                "state": "California"
            }
        },
        {
        }
],
"type": "FeatureCollection"
}
```


5.2.1 Denoting a geometry field: $geometry

The **$geometry** parameter allows you to specify the name of the column containing the spatial data.

For example, continuing with the example in the **View endpoints section**, if the bv_corp_offices view has a field named "location" (without using the suffixes "wkb" or "wkt" in its name) containing geometric data and we access the following URL:

```
.../geojson/test/views/bv_corp_offices?$geometry=location
```

The response would be rendered as in the previous example where the field name was "location_wkt" instead of "location":


```
{
"features": [
        {
            "geometry": {
                "coordinates": [
                    -122.159285,
                    37.448861
                ],
                "type": "Point"
            },
            "type": "Feature",
            "properties": {
                "zipcode": "94301",
                "country": "United States",
                "city": "Palo Alto",
```

```
                "street": "525 University Avenue #31",
                "name": "Denodo HQ",
                "id": "1",
                "state": "California"
            }
        },
        {
        }
],
"type": "FeatureCollection"
}
```

Note that if there is no field referenced by the `$geometry` parameter or if the view does not have any columns whose name ends with "`wkt`" or "`wkb`" then the value for the geometry will be set as `null` and all the columns become part of the properties.

5.3 VDP VIEW COUNT ENDPOINTS

The number of rows returned by a view can be checked by adding a `/$count` to the view endpoint. For example, a request to the following URL:

```
.../geojson/test/views/bv_corp_offices/$count
```

will return:

```
{
    "count": 9
}
```

reflecting the number of results in the view.

5.4 OBTAINING ITEMS BY PRIMARY KEY

Each item could be identified using its primary key property:

```
.../geojson/<DBNAME>/viewName/<keyvalue>
```

Examples:

```
.../geojson/test/views/bv_corp_offices/1

.../geojson/test/views/bv_dist_centers/NE
```

The primary key can be a compound key, and in that case you must include all values separated by commas:

```
.../geojson/test/views/bv_retail/1,STo3
```

Note: When there is not a defined primary key, this option is unavailable.

Note: URL encoded characters should be used where necessary. For example, `%26` for $.

5.5 ACCESSING INDIVIDUAL PROPERTIES WITH $SELECT

Properties of an item can be accessed individually using the **`$select`** query parameter:

```
.../geojson/<DBNAME>/views/<VIEWNAME>?$select=<PROPERTYNAME>
```

Note: For the sake of simplicity we are removing the server, port, and GeoJSON service root from the example URLs.

Note: A `?` needs to be added to the end of the URL in order to start adding query parameters.

Example:

```
.../geojson/test/views/bv_corp_offices?$select=state
```

Response:

```
{
    "features": [
       {
            "geometry": null,
            "type": "Feature",
            "properties": {
                  "state": "California"
            }
       },
       {

       }
      ],
      "type": "FeatureCollection"
}
```

Multiple properties can be accessed by separating them with a comma. Example:

```
.../geojson/test/views/bv_corp_offices?$select=name,location_wkt
```

Response:

```
{
   "features": [
      {
            "geometry": {
                  "coordinates": [
                        -122.159285,
                        37.448861
                  ],
                  "type": "Point"
            },
            "type": "Feature",
            "properties": {
                  "name": "Denodo HQ"
            }
      },
      {

      }
      ],
      "type": "FeatureCollection"
}
```

6 ADVANCED QUERYING

GeoJSON defines some query options that allow refining the requests: **$filter**, **$select**, **$orderby**.

6.1 SELECTION: $FILTER

A URI with a **$filter** system query option identifies a subset of the entries from the collection that satisfy the **$filter** predicate expression.

Expressions can reference properties and literals. The latter can be strings (enclosed in single quotes), numbers, boolean or datetime values.

Denodo GeoJSON service supports the following operations and functions:

6.1.1 Operators

Operator	Description	Example
=	Equal	/bv_retail?$filter=state = 'WASHINGTON'
<>	Not equal	/bv_retail?$filter=state <> 'WASHINGTON'
>	Greater than	/bv_retail?$filter=store_id > 5
>=	Greater than or equal	/bv_retail?$filter=store_id >= 5
<	Less than	/bv_retail?$filter=store_id < 5
<=	Less than or equal	/bv_retail?$filter=store_id <= 5
and	Logical and	/bv_retail?$filter=store_id > 5 and store_i < 10
or	Logical or	/bv_retail?$filter=store_id < 5 or state = 'UTAH'
is null	Column value is NULL	/bv_retail?$filter=state is null
is not null	Column value is not NULL	/bv_retail?$filter=state is not null
like	Search for a specified pattern in a column. Available wildcard characters: % (percentage). Represents a segment of	/bv_retail?$filter=state like 'W%25'

	text of any length, including an empty text. Remember to use it as an URL encoded character (%25) _ (underscore). Represents any character (only one character).	
between	a between b AND c: true if a is greater than or equal to b and less than or equal to c	/bv_retail?$filter=store_id between 5 and 10

Note that when you want to filter by a datetime field, you have to provide the value with a specific format, depending on the type of the field:

Type	Format	Example
date	DATE 'yyyy-mm-dd'	/bv_retail?$filter=datecol = DATE '2020-10-27'
time	TIME 'hh:mm:ss[.[nnn]][<time zone interval>]'	/bv_retail?$filter=timecol = TIME '02:02:02'
timestamp	TIMESTAMP 'yyyy-mm-dd hh:mm:ss[.[nnn]][<time zone interval>]'	/bv_retail?$filter=tscol = TIMESTAMP '2020-10-27 11:29:48.104322'

6.2 PROJECTION: $SELECT

The **$select** system query option returns only the properties explicitly requested. **$select** expressions can be a comma-separated lists of properties or the star operator (*), which will retrieve all the properties.

Example:

```
.../geojson/test/views/bv_corp_offices?$select=zipcode,city,state
```

Response:

```
{
"features": [
      {
            "geometry": null,
            "type": "Feature",
            "properties": {
```

```
                "zipcode": "94301",
                "city": "Palo Alto",
                "state": "California"
            }
        },
       ...
```

6.3 ORDERING RESULTS: $ORDERBY

The **$orderby** query string option specifies the order in which items are returned:

```
.../geojson/<DBNAME>/views/<VIEWNAME>?$orderby=attribute [asc|desc]
```

To order the collection the resource path must identify a collection of entries, otherwise this option is unavailable.

The keywords **asc** and **desc** determine the direction of the sort (ascending or descending, respectively). If **asc** or **desc** are not specified items are returned in ascending order. Null values come before non-null values when sorting in ascending order and vice versa.

You can also sort by multiple attributes:

```
.../views/<VIEWNAME>?$orderby=attr1 [asc|desc],attr2 [asc|desc]
```

Example:

```
.../geojson/test/views/bv_denodo_offices?$orderby=zipcode

.../geojson/test/views/bv_denodo_offices?$orderby=city%20asc,zipcode
%20desc
```

6.4 GROUPING RESULTS: $GROUPBY

The **$groupby** option will let you perform a SQL group by on the view. It can be used as shown below:

```
.../geojson/<DBNAME>/viewname?
$select=field1,field2&$groupby=field1,field2
```

7 PAGINATION

Whenever the Denodo Geojson Service has to return a collection of entries which size exceeds that configured at the `default.rowlimit` property of its configuration file, it will split the response into pages (returning only the first n entries).

You can manage the pagination with the parameters **`$count`** and **`$start_index`** detailed below. The value of **`$start_index`** has priority over the value of `default.rowlimit`.

7.1 SPECIFYING MAXIMUM NUMBER OF RESULTS: `$COUNT`

The **`$count`** system query option selects the n first entries of the collection, with n being a non-negative integer:

```
.../geojson/<DBNAME>/views/<VIEWNAME>?$count=<POS_INT_VALUE>
```

Example:

```
.../geojson/test/views/bv_corp_offices?$count=1
```

Response:

```
{
"features": [
    {
      "geometry": {
            "coordinates": [
                  -122.159285,
                  37.448861
            ],
            "type": "Point"
      },
      "type": "Feature",
      "properties": {
            "zipcode": "94301",
            "country": "United States",
            "city": "Palo Alto",
            "street": "525 University Avenue #31",
            "name": "Denodo HQ",
            "id": "1",
            "state": "California"
      }
   }
],
```

```
"type": "FeatureCollection"
}
```

7.2 SPECIFYING START OF THE QUERY $START_INDEX

The **$start_index** allow to skip the first n rows of the result set, with n being a nonnegative
Integer:

```
.../geojson/<DBNAME>/views/<VIEWNAME>?$start_index=<POS_INT_VALUE>
```

Example:

```
.../geojson/test/views/bv_corp_offices?$star_index=2
```

This sample returns the query, starting the result in the third tuple.