GEO5019 - Assignment 1

Building Footprints RESTful API **Student:** Heiko Rotteveel (5480787)

1 Introduction

For this assignment, I developed a small Python-based RESTful API that serves building footprint data for a part of Zuid-Holland. The system downloads open datasets, stores them in a spatial database, and exposes a set of endpoints for querying buildings by municipality or bounding box. This report briefly summarises how the system works, the main challenges, and the extent to which the implementation follows the specification.

For this assignment, I created a small repository containing Python code for a RESTful API that serves building footprint data from the Zuid-Holland region in the Netherlands. The Python code downloads open geospatial data, stores it in a spatial database, and creates a web API to query the data with various filtering options. This report briefly summarises how the system works, the main challenges, and the extent to which the implementation of the endpoints is aligned with the *OGC API Features* specification.

2 System Overview

The system consists of two parts: (1) preparing a spatial database with DuckDB and (2) developing an API using FastAPI.

2.1 Backend

The backend workflow starts by creating a database file called *buildings-database.db* and loading the duckDB *Spatial* extension. The database stores building and municipality datasets covering a subsection of Zuid-Holland with the following bounding box (EPSG:28992):

$$\text{minx} = 78600, \quad \text{miny} = 445000, \quad \text{maxx} = 85800, \quad \text{maxy} = 450000.$$

Building data is retrieved from *Overture* (in EPSG:4326), so the bounding box is transformed before ingestion. The data is loaded into the TABLE `buildings`, after which geometry and BBOX columns are reprojected to EPSG:28992.

The municipality data of Zuid-Holland is collected from 'PDOK' in EPSG:28992 and loaded into the database (TABLE *municipalities*). A spatial join assigns each building to a single municipality based on the first detected intersection. If a building thus lies on the border of multiple municipalities, it will be assigned to the first municipality. Finally, a `building_count` column is added to the municipalities table, describing how many buildings fall inside each municipality

2.2 API

The API exposes four primary endpoints:

1. `/collections`: returns all municipalities,
2. `/collections/{municipality}/items`: buildings within a municipality,

3. `/collections/{municipality}/items/{building-id}`: single building lookup,
4. `/buildings/bbox`: buildings within a user-provided bounding box.

Most endpoints support a limit (default 50, max 1000) and offset parameter for pagination. Endpoint three is the exception, as it always only returns a maximum of one building. The limit and offset are both provided in the metadata of the JSON return following a standard schema. Schemas were created for buildings, features, metadata, and geometries to standardise and validate output. A paginator class also generates links to next/previous pages within the metadata of each JSON response to quickly walk through the dataset. Multiple error-handling functions were also added to inform users of the types of mistakes; Two functions to check if the input values do not contain failed constraints (Error code 400), an error handling for if the request cannot be queried or there is some internal server (Error code 500), an error is returned if there are no results (Error code 404) and an error is given if the query parameters are not correct (Error code 422).

3 Challenges

Using DuckDB was at times challenging due to limited documentation. For example, the `BOX2D` datatype may have simplified CRS transformations of the bbox, but because the documentation was lacking, there was no explanation of this datatype and how to tranform to it to be found. FastAPI on the other hand, was a breath of fresh air with how well it was documented. There were clear examples, it was well structured, and everything necessary for this project could be found on the FastAPI site itself. Though this article from Medium was also really helpful in creating the pagination and schemas: https://lewoudar.medium.com/fastapi-and-pagination-d27ad52983a.

Another issue when using DuckDB in the API was that it didn't always properly close after terminating the process. Therefore, when trying to restart the API, this was not possible due to an already active session, which could only be terminated by restarting the computer completely. Luckily, this could be fixed by creating a lifeline for FastAPI that explicitly closes the connection when shutting the API down.

One hard neck problem was the correct writing of features to a GeoJSON. The https://geojsonlint.com/ site kept giving right-hand rule errors even though the coordinates were correctly oriented. After an hour of trying to fix it, trying other GeoJSON validators ended up proving that the GeoJSONs were, in fact, correct. There seems to be some problems with these validators.

Lastly, some DuckDB SQL conventions differ from PostgreSQL, which made it difficult to create some queries. LLMs that were asked to help formulate these queries were also not useful. The suggestions were almost always inefficient, made for other SQL languages, or simply did not work. It ended up costing more time to fix their errors than simply writing them myself.

4 OGC API compliancy

The API is currently not fully OGC compliant. Endpoint one misses the information about the geospatial data collection, like its description and the spatial and temporal extents of all the data contained. For all the endpoints, there is also currently no information on what CRS the data is stored in (which normally would mean people assume it to be in EPSG:4326, even though it is EPSG:28992), nor is there the possibility to transform it to another CRS or to provide a bbox in another CRS.