HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

MSE | MASTER OF SCIENCE IN ENGINEERING

A Minimal Feature and Tile Server written in Python

Projekt Arbeit 1

Labian Gashi

Information and Communication Technologies
Software and Systems

Advisor Prof. Stefan Keller

HSR - Hochschule für Technik Rapperswil

Fall Semester 2019

Abstract

This project serves as a ”Mini Feature And Tile Server” which is minimally compliant with one of the geospatial standards defined in the web.
These standards define how APIs should be built & developed in order to standardize the ways of retrieving geospatial data from the web using modern web development practices.

The standard that this project is minimally compliant with is the ”OGC API - Features” standard. It is a standard created by the Open Geospatial Consortium (OGC) which is an international consortium consisting of hundreds of institutions that encourage development and implementation of open standards for geospatial content and services. The OGC API - Features standard specifies the behaviour of Web APIs that provide access to geospatial data. Various requests to the API enable the user to retrieve information about the underlying data set that it contains.
This application receives one or multiple GeoJSON files as input, pre-processes them, stores them as *collections*, and then makes them available for querying and investigating to the API requesters.

The application is built using the Python programming language and is also containerized using Docker, which makes it simple & efficient to be setup in any host system.

Keywords: *OGC, OGC API, GDAL, OAPIF, XYZ, GeoJSON, Web Tiles*

Acknowledgements

I would like to express my gratitude towards my advisor Prof. Stefan Keller for suggesting this very neat project and assisting me throughout the whole journey. His great interest in geodata and spatial features made me (who didn't know anything about these topics before) very interested and engaged in them as well!

A huge thanks also goes to Nicola Jordan from the institute of HSR who helped me and gave me suggestions on how to code in Python and was just generally a great guidance for my first Python project.
A special thanks goes to all the authors of the Python libraries, Docker images and LateX templates that I used in order to complete this project.

I would also like to send some appreciation to the GitLab team for having such a nice and neat application lifecycle tool for me to utilize. It provided me with a code repository, CI/CD pipeline features and just generally a nice and consistent environment to have my application on.

Declaration of Authorship

I hereby declare that:

- This thesis and the work reported here was composed by and originated entirely from me unless stated otherwise in the assignment of tasks or agreed otherwise in writing with the supervisor.
- All information derived from the published and unpublished work of others has been acknowledged in the text and references are correctly given in the bibliography.
- No copyrighted material has been used illicitly in this work.

Place, Date Signature

Contents

Chapter 1

Introduction

The documentation for this project is split into four main parts. The first chapter gives an introduction to the work done in relation with the goals and requirements that this project was set initially to meet. It also explains about the standards that this application follows and also some important notations related to those standards.

The second chapter has to do with the architecture and the design of the application. It shows some design decision, some of the technologies that were used, some useful information regarding those technologies and the reason why they were used for this project.

After the things above have been clarified, we now delve into the third chapter which explains the actual implementation of the application. It mostly contains the main notations about the project, what they mean and how they were used in the benefit of the application. It also explains about the code repository that was used and a little bit of how the project was documented.

The fourth and final chapter presents the results that were achieved from this project work and the outlook of it. It wraps up with a conclusion about the project and my personal opinions about this project.

1.1 Goals and Requirements

The main objective for this project is creating a back-end application that is minimally compliant with the ”OGC API - Features” [1]. The application also has to support correct API responses for the OAPIF [2] driver (which is an acronym for OGC API Features). It also has to support a couple more API endpoints that return raster tiles. More technical explanation about those endpoints and how they work will be written below and in the implementation chapter.
The main tasks include: returning GeoJSON objects given a specific collection, with the option of limiting the features to a specific bounding box within a region or even limiting the result set to as many features as the request prefers. These are the tasks that also should be compliant with the ”OGC API - Features”.

This project not only serves as an API that is minimally compliant with the ”OGC API - Features” standard, but it also serves as a possible replacement for an application developed by the HSR Geometa Lab. This application, called Castle Map is an application that shows all the castles in Switzerland within a map layer.
This application utilizes the API’s methods to return all the features within a specific area (in this case, Switzerland) and adds links to those features from which users can access the Wikidata about that feature, the Wikipedia data or even open it on OpenStreetMap. It also uses some of the other API endpoints that this API offers in order to server map tiles (which will be explained more thoroughly below) as images on top of the map layer that displays the features, or in this case, the castles.
The features are sorted based on the views the castles receive (in Wikidata) and then are displayed in a sort-of prioritized way in the map layer.
The Castle Map application contains an MIT license and is contained here:
gitlab.com/geometalab/castle-map

Another requirement for this application is to be operable by the ogrinfo program, which is a program contained in the ”GDAL” library (more details about GDAL below) that lists information about a data source that is supported by GDAL.
In this case, the ogrinfo program must be able to make requests to this application and receive correct responses from it using the OAPIF driver, which is a driver for connecting to OGC API servers.

1.2 Open Geospatial Consortium API - Features

Open Geospatial Consortium (OGC) is an international consortium consisting of hundreds of institutions that encourage development and implementation of open standards for geospatial content and services.
OGC API - Features is a standard, created by OGC that is very important for this project since this project implements it's features and tasks the standard defined HERE.

Portion of text directly taken from the OGC API - Features Abstract: [1]
"OGC API standards define modular API building blocks to spatially enable Web APIs in a consistent way. The OpenAPI specification is used to define the API building blocks.

The OGC API family of standards is organized by resource type. This standard specifies the fundamental API building blocks for interacting with features. The spatial data community uses the term 'feature' for things in the real world that are of interest.

OGC API Features provides API building blocks to create, modify and query features on the Web. OGC API Features is comprised of multiple parts, each of them is a separate standard. This part, the "Core", specifies the core capabilities and is restricted to fetching features where geometries are represented in the coordinate reference system WGS 84 with axis order longitude/latitude." [1]

1.3 Geospatial Data Abstraction Library

Geospatial Data Abstraction library or GDAL is a very important notation for this project since a lot of the drivers and features in their library can be used in conjunction with this project to deliver geospatial data and different responses.

GDAL is a translator library for raster and vector geospatial data formats that is released under an X/MIT style Open Source License by the Open Source Geospatial Foundation. As a library, it presents a single raster abstract data model and single vector abstract data model to the calling application for all supported formats. It also comes with a variety of useful command line utilities for data translation and processing. [3]
GDAL also provides a lot of drivers, and the driver that is important for this project is the OAPIF driver (Short for OGC API - Features).
This driver provides us the opportunity to query OGC API servers and to retrieve Geo information from those APIs.

1.4 Tiled Web Map

A tiled web map is a map that is displayed is a map that is displayed by joining a lot of individually requested images or vector files over the internet. It is currently the most popular way to display maps, which replaces the old methods such as Web Map Service (WMS), which usually displayed a single but large map, that was navigable using different arrow buttons. The first technology for displaying tiles used raster tiles and then after that, vector tiles were introduced.
The reason why this standard is important for this project is that this application implements some of the methods that are mentioned there.

1.4.1 XYZ Specification

One of the most popular ways to serve these tiles is using the XYZ specification, Google was one of the first companies to majorly use this way of serving tiles. This request, then usually follows the de facto OpenStreetMap standard (known as Slippy Map [4]) for tiles that follows these rules:

- Tiles are 256 × 256 pixel PNG files
- Each zoom level is a directory, each column is a subdirectory, and each tile in that column is a file
- The request URL looks like *http://.../{z}/{x}/{y}.png*, where *Z* is the zoom level, and *X* and *Y* identify the tile's position.

Chapter 2

Architecture and Design

2.1 Overview

The application is developed and built using the Python language (v3.7.4) and it is also containerized into a Docker image which contains the Alpine Linux distribution, the operating system which runs the application using the Python language. The application uses a lot of geospatial libraries in Python, it also uses some libraries that were initially created in other languages (C++, Go etc), but were minimally converted to Python since they were very powerful and useful for the community. Some of these libraries are powerful (for example the drawing library) and require a lot of dependencies which do not exist in the Alpine Linux distribution out of the box for the docker image. As a result, the docker image is not very light as it requires those dependencies first before the libraries can be used (which are necessary for the project).
The application mostly deals with geometry objects, API requests, correct and precise HTTP responses and some mathematical calculations for the geospatial data.

The docker image of this application also contains the application's libraries and some dependencies that some of those libraries use. As soon as the docker image is built and ran, there is a command in that docker file that starts the application automatically.

```
FROM tiangolo/uvicorn-gunicorn:python3.7-alpine3.8
WORKDIR /app/wfs
COPY requirements.txt /app/wfs/
RUN apk add build-base python-dev py-pip jpeg-dev zlib-dev
ENV LIBRARY_PATH=/lib:/usr/lib
RUN pip install -r requirements.txt
COPY . /app/wfs
EXPOSE 8000
CMD ["uvicorn", "ogc_api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

FIGURE 2.1: How the docker file looks

That docker image is then also sent to the Docker Compose tool which is a tool for defining and running docker applications (it is more suitable for multi-container applications, but works in this case too), this was added for easier accessibility, configuration and running of the container.

```
version: '3'
services:
  ogcapi_server:
    build: .
    ports:
      - "80:8000"
    environment:
      - COLLECTIONS=castles=/app/wfs/osm-castles-CH.geojson
      - PORT=8000
```

FIGURE 2.2: How the docker compose file looks

The container architecture makes it very easy to build and start the container since it only requires changing the environment variables in the docker compose (if necessary) and then running the command:

```
$ docker compose up --build
```

2.2 Design Decision - Programming Language

This application was built using the Python programming language, version 3.7.4.

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. [5]

The reason why this project was chosen to be build in Python is because: it is powerful, runs everywhere, can be used in conjunction with other technologies too and has been getting a lot of attention lately. A big difference between Python and other programming languages is the syntax and the programming paradigm it uses. Usually, when programming in Python, it requires a different way of thinking, python programmers call it a ”pythonic” way of programming.

Here's a Python code & syntax example for writing/reading a file: [6]

```
#!/usr/bin/python

import sys

try:
# open file stream
file = open(file_name, "w")
except IOError:
print "There was an error writing to", file_name
sys.exit()
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
file_text = raw_input("Enter text: ")
if file_text == file_finish:
# close the file
file.close
break
file.write(file_text)
file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
```

```
if len(file_name) == 0:
print "Next time please enter something"
sys.exit()
try:
file = open(file_name, "r")
except IOError:
print "There was an error reading file"
sys.exit()
file_text = file.read()
file.close()
print file_text
```

As you can see, this is not the typical code syntax that people write in other programming languages. However, it is very convenient and efficient once you get used to it.

2.3 Design Decision - Containerization

This application is also shipped into containers, which are the new thing in the software engineering world. This project uses only one container, the application only, since that's the only required one, there is no database or front-end application which has to be separated into other containers.
The container architecture that this project uses is Docker.
Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating-system kernel and are thus more lightweight than virtual machines. [7]

Here's an illustration of the Docker architecture:

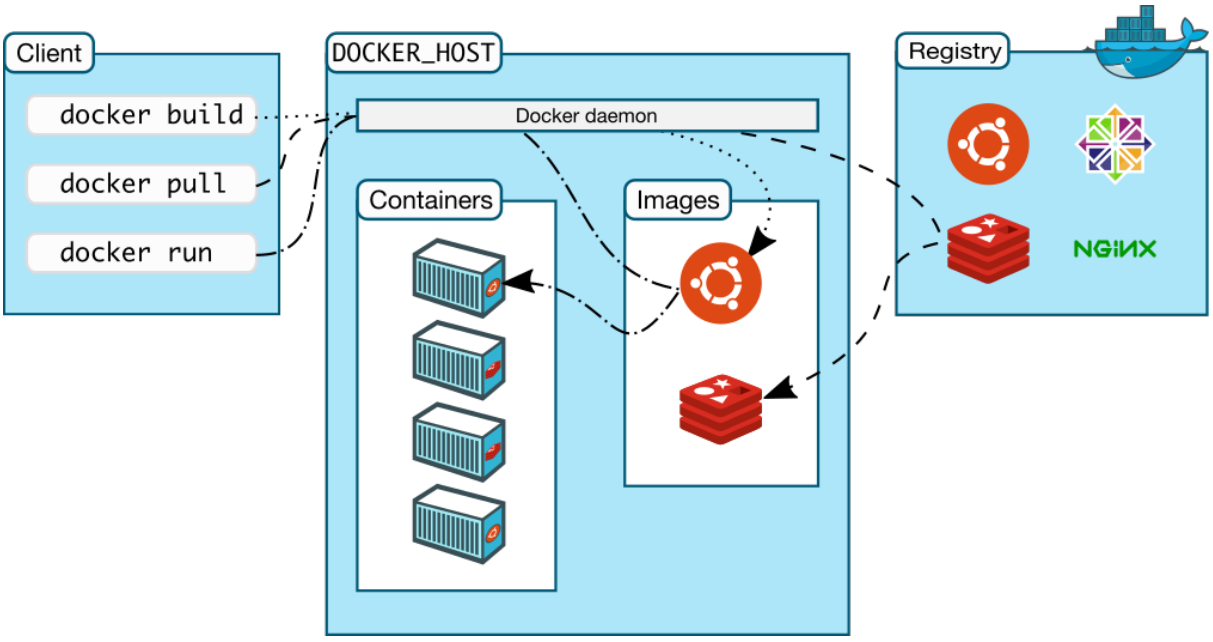

FIGURE 2.3: How the docker architecture looks like [8]

Docker images are read-only templates and can for instance contain already installed software such as Ubuntu, Python, various databases or in this case, combined image of Alpine Linux and Python. They simplify the process of creating Docker containers.

Docker registries are used for distributing existing Docker images. They can be either private or public in the Docker Hub. Docker images are downloaded from Docker registries.

Docker containers are built from the Docker images themselves. They contain everything needed for an application to run and have their own file system and networking.

Chapter 3

Implementation

3.1 Overview

This chapter shows how the project was implemented in a more-detailed way. It also shows some of the key points for this application, how they were implemented and what they are useful for. This chapter of the documentation is the technical part of it.
It will show how the data is inserted into the server, how it is processed and based on the request, how that data is used to return a response to that specific request.
This section will also cover some of the important techniques and tools that were used to test, document, generate artifacts etc. It will also cover the code repository and CI/CD part, which for this application, is GitLab.

3.2 API

This application serves as an API for OAPIF clients or the driver itself. In order to use it, one must point it to the URL where it is hosted under. All the paths and request handling are taken care by the application itself. The application has around 6 API endpoints which some of them are a requirement from the ”OGC API - Features” standard, and the others are a requirement for the web map tiles and also for one of the front-end applications (the castle map mentioned in the introduction goals), which is going to use this API to serve features, geometry objects but also PNG raster tiles which will appear as dots in the application where a user can click on them in order to view what castle is hidden under a dot and get more information about it.
The application uses a library called FastAPI, which was very convenient for this project and offers a lot of different ways of implementing an API, depending on the requirements and the nature of the application.

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

The key features are:

- Fast: Very high performance, on par with NodeJS and Go (thanks to Starlette and Pydantic). One of the fastest Python frameworks available.
- Fast to code: Increase the speed to develop features by about 200
- Fewer bugs: Reduce about 40
- Intuitive: Great editor support. Completion everywhere. Less time debugging.
- Easy: Designed to be easy to use and learn. Less time reading docs.
- Short: Minimize code duplication. Multiple features from each parameter declaration. Fewer bugs.
- Robust: Get production-ready code. With automatic interactive documentation.
- Standards-based: Based on (and fully compatible with) the open standards for APIs: OpenAPI (previously known as Swagger) and JSON Schema. [9]

3.3 GeoJSON & Features

This application works a lot with GeoJSON. It processes it and uses various libraries to serialize/deserialize it in order to re-format it and present it to the request response in the appropriate format.
The application needs a GeoJSON file (.geojson) to be supplied using the environment variables in the docker compose file (multiple files can be passed too). The GeoJSON file should contain a FeatureCollection with Features inside that the server will use to display and send responses using those features. These files, when parsed and formatted appropriately are called *collections* in the application and in the *OGC API - Features* standard.

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. [10]

A Feature object represents a spatially bounded thing. Every Feature object is a GeoJSON object no matter where it occurs in a GeoJSON text.

- A Feature object has a ”type” member with the value ”Feature”.
- A Feature object has a member with the name ”geometry”. The value of the geometry member SHALL be either a Geometry object as defined above or, in the case that the Feature is unlocated, a JSON null value.
- A Feature object has a member with the name ”properties”. The value of the properties member is an object (any JSON object or a JSON null value). [10]

Here’s an example of some minimal GeoJSON data:

```
{
  "features": [
      {
        "geometry": {
        "coordinates": [
          11.183468,
          47.910414
        ],
        "type": "Point"
      },
        "id": "N123",
        "properties": {
        "natural": "lake",
        "name": "Katzensee"
      },
      "type": "Feature"
      }
  ],
  "type": "FeatureCollection"
}
```

3.4 Raster Tiles

3.5 Testing

This application is tested using two methods. That is the Unit Testing which was done using pytest and the Load Testing which was done using Locust to measure the application's performance.
The Unit Testing was done to test some of the main features of the application, some HTTP requests & responses etc.
The Load Testing was done to test what happens if a lot of users access the server and make the same requests at the same time, what the performance of the server into giving responses would be.

3.5.1 Unit Testing

The Unit Testing, as said above, was done by using pytest.
Unit Testing is a software testing method where individual parts of the code, or modules, or usage procedures are tested in order to find out if they work as expected.
pytest is a framework that makes building simple and scalable tests easy. Tests are expressive and readable—no boilerplate code required. [11] pytest is a more "pythonic" way of unit testing in Python. It uses the native Python syntax and methods in order to assert results. After all the unit tests are created, they can be simply run with the command:

```
$ pytest <path of file or path of folder>
```

What this will do is will check all the files within that directory or the file itself if it is a file and test them all, if one of them fails it will show which one and return a failure error, if all of them succeed it will say all of them succeeded and will pass the unit tests.

An example of a unit test using pytest:

```
# content of test_class.py
class TestClass:
  def inc(x):
    return x + 1

  def test_answer():
    assert inc(3) == 5
```

This will return a failure, because 3 + 1 is equal to 4 and not to 5:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F                                                    [100%]

================================ FAILURES ================================
_____________________________ test_answer _____________________________

def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_sample.py:6: AssertionError
========================== 1 failed in 0.12s ===========================
```

And in this project's case, for example, where all the unit tests succeed, it shows this:

```
$ pytest tests/
========================== test session starts ==========================
platform win32 -- Python 3.7.4, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: C:path\to\directory\python-wfs-server
collected 22 items

tests\test_geometry.py ....
tests\test_index.py ...
tests\test_server.py ............
tests\test_tiles.py ..

========================== 22 passed in 1.36s ===========================
```

3.5.2 Load Testing

The Load Testing part, was done using Locust.
Load testing is a subset of performance testing, it puts demand on a system and measures its response time and performance. It is used to check how a system behaves under low, normal or peak load conditions.
Locust is an easy-to-use, distributed, user load testing tool. It is intended for load-testing web sites (or other systems) and figuring out how many concurrent users a system can handle.
The idea is that during a test, a swarm of locusts will attack your website. The behavior of each locust (or test user if you will) is defined by you and the swarming process is monitored from a web UI in real-time. This will help you battle test and identify bottlenecks in your code before letting real users in. [12]

A locust file looks something like this:

```
from locust import HttpLocust, TaskSet, between

def login(l):
  l.client.post("/login", {"username":"ellen_key", "password":"education"})

def logout(l):
  l.client.post("/logout", {"username":"ellen_key", "password":"education"})

def index(l):
  l.client.get("/")

def profile(l):
  l.client.get("/profile")

class UserBehavior(TaskSet):
  tasks = {index: 2, profile: 1}

def on_start(self):
  login(self)

def on_stop(self):
  logout(self)

class WebsiteUser(HttpLocust):
  task_set = UserBehavior
  wait_time = between(5.0, 9.0)
```

To run locust, we just simply run the command:

```
$ locust
```

After that, the locust server starts and can be accessed by default in http://127.0.0.1:8089 (if Locust is being ran locally). Then a page would open that would look something like this:

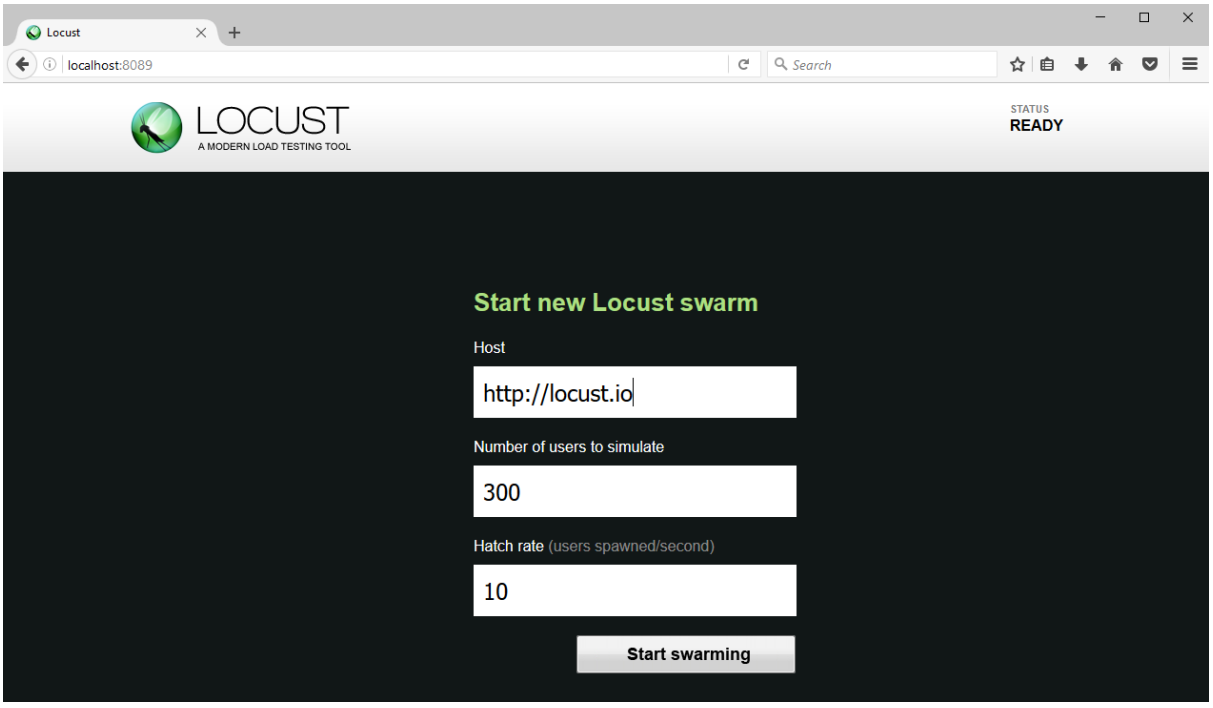

FIGURE 3.1: A look at the Locus web-interface

3.6 Code Repository and the CI/CD Feature

The code repository and the CI/CD pipeline features for this project were provided by GitLab.
GitLab is a DevOps lifecycle tool that provides a lot of things like: Git-repository manager providing wiki, issue-tracking and CI/CD pipeline features, it uses an open-source license and is developed by GitLab Inc. It has a very nice web interface and lots of different features that make application lifecycle much easier. The features GitLab offers that were used for this project were the project repository and the CI/CD feature.

CI/CD, or continuous integration and continuous delivery (aka continuous deployment), combines the values of these two practices in order to provide precise integration and delivery. GitLab has a feature of writing a CI/CD file (yaml), which does various operations for you each and every push, to make sure that the CI/CD configuration still works with the applied changes.
In this case, CI/CD was used to test the project (run the unit tests) and then generate and upload the documentation PDF into an artifact which can be downloaded from the CI/CD pipeline itself.

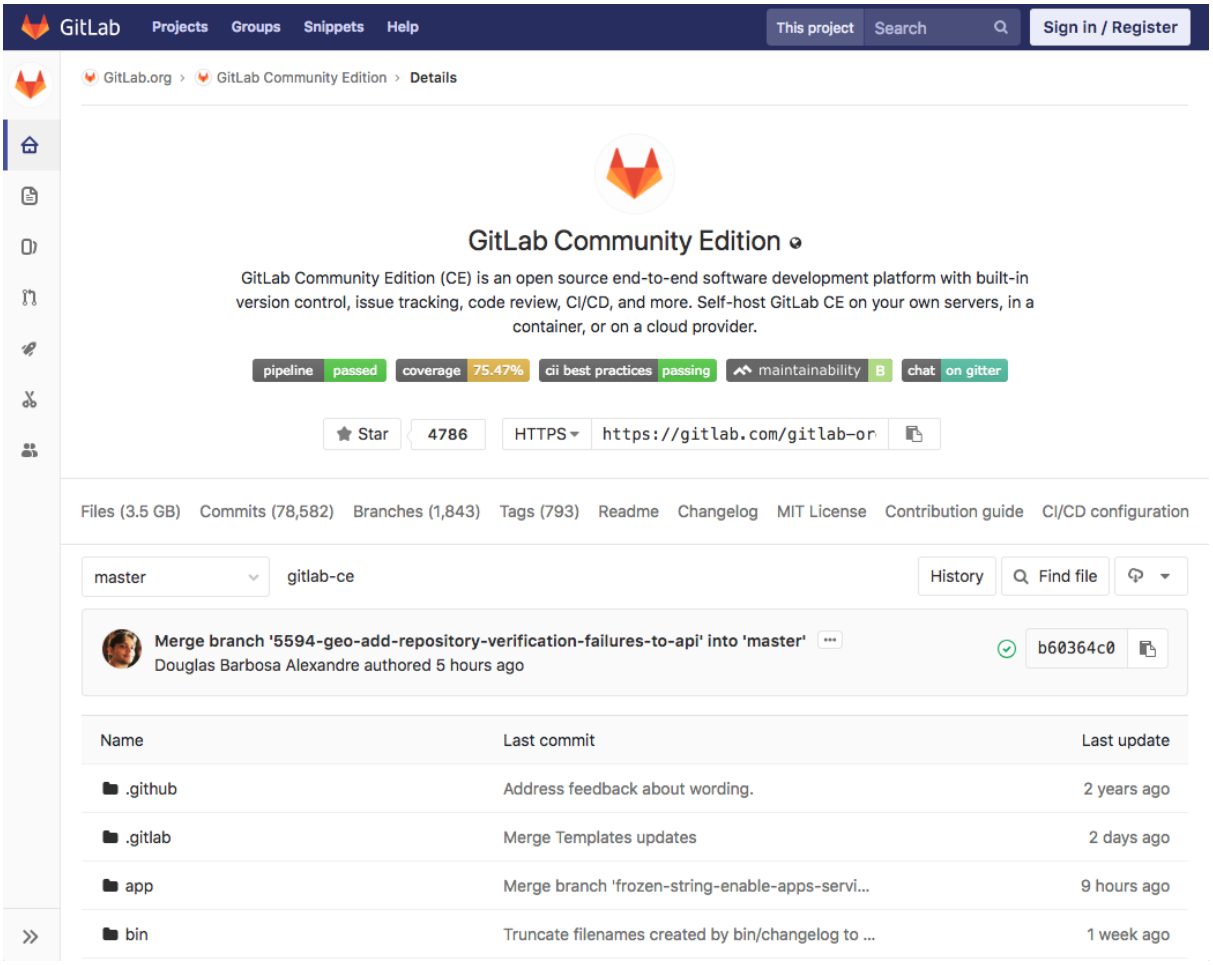

FIGURE 3.2: A peek at the GitLab interface

3.7 Document Preparation System

The system that was used in order to document this project was LaTeX.
LaTeX is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents. LaTeX is available as free software. [13]
The application that was used in conjunction with LaTeX was TeXstudio. TeXstudio is an integrated writing environment for creating LaTeX documents. Our goal is to make writing LaTeX as easy and comfortable as possible. Therefore TeXstudio has numerous features like syntax-highlighting, integrated viewer, reference checking and various assistants. [14]
The reason why LaTeX was used is that it is very convenient to developers, since they can get used to the syntax very quickly and are familiar with ”coding” approaches. Another big reason is that it provides a lot of packages for displaying code in a nice formatted way, it has the ability of displaying web links in a nice manner and also the figures and figure alignments are very appropriate and easy to create & maintain.

This is for example how a portion of this page looks like in LaTeX before being compiled:

```
1 \section{LaTeX}
2 The system that was used in order to document this project was
  ↪ \href{https://www.LaTeX-project.org/}{LaTeX}.\\
3 LaTeX is a high-quality typesetting system; it includes features designed
  ↪ for the production of technical and scientific documentation. LaTeX
  ↪ is the de facto standard for the communication and publication of
  ↪ scientific documents. LaTeX is available as free software.
  ↪ \cite{WhatIsLaTeX}\\
4 The application that was used in conjunction with LaTeX was
  ↪ \href{http://texstudio.sourceforge.net/}{TeXstudio}.
5 TeXstudio is an integrated writing environment for creating LaTeX
  ↪ documents. Our goal is to make writing LaTeX as easy and comfortable
  ↪ as possible. Therefore TeXstudio has numerous features like
  ↪ syntax-highlighting, integrated viewer, reference checking and
  ↪ various assistants. \cite{WhatIsTeXStudio}\\
```

Chapter 4

Results

4.1 Achievements

These were/are the goals for this project that have all been met.

- Implement a 'Mini-Feature and Tile Server' in Python.
  - Input big, sorted GeoJSON files
  - Preprocess and format those files containing features and return them to the HTTP response in the appropriate format (using the OGC API - Standards format), support limiting the results in a given bounding box.
  - Return a specific feature in the feature collection, it's metadata and it's bounding box.
  - Return a raster tile (256x256) which is then displayed on top of a map to display where the features are located in the map (used in conjunction with maptile for example).
  - Containerize the application so that it can easily be shipped between different hosts.
- Integrate this in the fork of Castle Dossier Map
- Integration tests with OAPIF (GDAL driver using the ogrinfo program)
- Integration tests with QGIS

Generally, the main idea was to have a project that is minimally compliant with the "OGC API - Features" and that is something that ultimately has been achieved by this project.

4.2 Outlook

The current version of this application is pretty robust and can be used to run a WFS/OGC client in tandem with.
It implements the OGC - API Features only minimally, there are some optional API endpoints that are mentioned in the standards but they have not been implemented here because the goal for this project was to only minimally support those standards, which means, only the required endpoints and rules.

Something that would maybe improve the application is to introduce async methods using the FastAPI framework, but that is not necessary for now.
Another thing would be implementing all the API endpoints (optional ones too) from the OGC - API Features standard but that is as well not needed now.
Another interesting extension for the application would be the implementation of the collection *SPATIAL EXTENT* in order to determine the correct location of the features in space. This extension would be very useful for clients who use the API and especially for QGIS.

4.3 Reflection

This project was a great experience for me and taught me a lot of things that I had never heard of!
First off, GeoJSON, GDAL, OGC, QGIS, these were all completely unknown terms for me, thanks to this project, I now know what they are and how important they are for geomatics.

Another thing that was very new for me was Docker, containers is something I have heard of before, but never really used. This is the first time I have used Docker and containers to ”ship” an application, even though it is a single-container application, I still learned a lot of new things from it.

And the most important part of this, is that the whole project is written in Python.
This was my first Python application and learning it was a big learning curve for me. I never worked with such syntax and such programming paradigm. I am mostly used to high level OOP programming languages like Java or C#. So this was a completely new, exciting but sometimes even confusing for me.

Working with such technologies, tools, libraries and standards was all around a new and great experience for me, although frustrating at times, it definitely taught me a lot of new things!

List of Figures

Appendix A

Installation

The process of setting up this server is very simple. This is made possible by docker and docker compose, which makes it even easier to set-up and run. The application, in order to be integrated with a front-end application, you just need to point it to the URL of the server and make the API calls in conformace with the ”OGC API - Features”. The server can also be hosted locally just by cloning the repository and using an ASGI server to run it.

A.1 Cloning and Containerizing

To checkout the source from GitLab, this command needs to be run:

```
$ git clone https://gitlab.com/labiangashi/python-wfs-server.git
```

After that, the application can be hosted using docker and docker-compose. Before doing that, first, the *docker-compose.yaml* file needs to be modified in order to include the correct environment variables for the *collections* files. The environment variables can be modified here in the docker compose file:

```
version: '3'
services:
  ogcapi_server:
    build: .
    ports:
      - "80:8000"
    environment:
      - COLLECTIONS=castles=/app/wfs/osm-castles-CH.geojson
      - PORT=8000
```

This docker compose file is a simple file which gives a name to the docker image, tells it what directory to build, what ports to expose and the environment variables required for the application.

After setting up the correct environment variables (the *collections* and the port), the next step is building that docker image and then running it. The Dockerfile already runs the server using uvicorn as soon as the image is run
This is how the Dockerfile looks:

```
FROM tiangolo/uvicorn-gunicorn:python3.7-alpine3.8
WORKDIR /app/wfs
COPY requirements.txt /app/wfs/
RUN apk add build-base python-dev py-pip jpeg-dev zlib-dev
ENV LIBRARY_PATH=/lib:/usr/lib
RUN pip install -r requirements.txt
COPY . /app/wfs
EXPOSE 8000
CMD ["uvicorn", "ogc_api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

A.2 Running the server

In order to build the image and then run it, the two following commands need to be executed in the terminal:

```
$ docker-compose build
```

and

```
$ docker-compose up
```

And that's it, the docker image with the running server is ready for production.

Appendix B

API Docs

Here are some of the API endpoints, provided by Swagger UI.
Disclaimer: This is all data within my environment, the response body can change if the environment or collections are different, but not the response's body or head structure.

B.1 OGC API Endpoints

B.1.1 Landing Page

Response body:


```
{
  "title": "OGC API - Features server",
  "description": "Web API that conforms to the OGC API Features specification.",
  "links": [
    {
      "href": "http://127.0.0.1:8000/",
      "rel": "self",
      "type": "application/json",
      "title": "This document"
    },
    {
      "href": "http://127.0.0.1:8000/api",
      "rel": "service-desc",
      "type": "application/json",
      "title": "The API definition"
    },
    {
```

```
      "href": "http://127.0.0.1:8000/collections",
      "rel": "data",
      "type": "application/json",
      "title": "Metadata about the feature collections"
    }
  ]
}
```


Response headers:


```
content-length: 609
content-type: application/json
date: Wed, 11 Dec 2019 08:02:43 GMT
server: uvicorn
```


B.1.2 /collections

Response body:


```
{
  "links": [
    {
      "href": "http://127.0.0.1:8000/collections",
      "rel": "self",
      "type": "application/json",
      "title": "Collections"
    }
  ],
  "collections": [
    {
      "name": "castles",
      "title": "castles",
      "links": [
        {
          "href": "http://127.0.0.1:8000/collections/castles",
          "rel": "item",
          "type": "application/geo+json",
          "title": "castles"
        }
      ]
    }
```

```
  ]
}
```


Response headers:


```
content-length: 294
content-type: application/json
date: Fri, 06 Dec 2019 13:42:33 GMT
server: uvicorn
```


B.1.3 /collections/{collection}

Response body:


```
{
  "name": "castles",
  "title": "castles",
  "links": [
    {
      "href": "http://127.0.0.1:8000/collections/castles",
      "rel": "item",
      "type": "application/geo+json",
      "title": "castles"
    }
  ]
}
```


Response headers:


```
content-length: 160
content-type: application/json
date: Fri, 06 Dec 2019 13:45:05 GMT
server: uvicorn
```

B.1.4 /collections/{collection}/items

Response body (with only 1 feature, which means a limit of 1):


```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "id": "N2306343684",
      "geometry": {
        "type": "Point",
        "coordinates": [
          10.749732,
          47.557561
        ]
      },
      "properties": {
        "castle_type": "stately",
        "description": "Die Schlosshöfe können kostenlos besichtigt werden. In die Sc
        "historic": "castle",
        "name": "Schloss Neuschwanstein   ",
        "name:ar": "(characters that can't be interpreted here)",
        "name:be": "(characters that can't be interpreted here)",
        "name:en": "Neuschwanstein Castle",
        "name:fr": "Château de Neuschwanstein",
        "name:ja": "(characters that can't be interpreted here)",
        "name:ja_kana": "(characters that can't be interpreted here)",
        "name:ru": "(characters that can't be interpreted here)",
        "name:zh": "(characters that can't be interpreted here)",
        "opening_hours": "Mo-Su 09:00-18:00",
        "phone": "+49 8362 93 98 80",
        "toilets:wheelchair": "yes",
        "tourism": "attraction",
        "website": "http://www.neuschwanstein.de",
        "wheelchair": "yes",
        "wikidata": "Q4152",
        "wikipedia": "de:Schloss Neuschwanstein"
      }
    }
  ],
  "bbox": [
```

```
    10.749732,
    47.557561,
    10.749732,
    47.557561
  ],
  "links": [
    {
      "href": "http://127.0.0.1:8000/collections/castles/items?limit=1",
      "rel": "self",
      "type": "application/geo+json",
      "title": "self"
    }
  ]
}
```


Response headers:


```
content-length: 1195
content-type: application/geo+json
date: Fri, 06 Dec 2019 13:46:28 GMT
server: uvicorn
```


B.1.5 /collections/{collection}/{feature_id}

Response body:


```
{
  "type": "Feature",
  "id": "N473471590",
  "geometry": {
    "type": "Point",
    "coordinates": [
      7.488973,
      45.736946
    ]
  },
  "properties": {
    "historic": "castle",
    "name": "Castello di Fénis",
    "wikidata": "Q607126",
    "wikipedia": "it:Castello di Fénis"
```

```
  }
}
```

Response headers:

```
content-length: 219
content-type: application/geo+json
date: Fri, 06 Dec 2019 13:47:36 GMT
server: uvicorn
```

B.2 Other Endpoints for Tiles

B.2.1 /tiles/{collection}/{zoom}/{x}/{y}.png

Response body:

FIGURE B.1: A raster tile PNG response, 256x256

Response headers:

```
content-length: 2342
content-type: image/png
date: Fri, 06 Dec 2019 13:49:50 GMT
server: uvicorn
```

B.2.2 /tiles/{collection}/{zoom}/{x}/{y}/{a}/{b}.geojson


```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "id": "W387544802",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [
              7.406668,
              46.649168
            ],
            [
              7.406333,
              46.649
            ],
            [
              7.406405,
              46.648945
            ],
            [
              7.406735,
              46.649107
            ],
            [
              7.406668,
              46.649168
            ]
          ]
        ]
      },
      "properties": {
        "historic": "castle",
        "name": "Festi",
        "wikidata": "Q67772651"
      }
    }
```

```
  ],
  "bbox": [
    7.406668,
    46.649168,
    7.406668,
    46.649168
  ]
}
```


Response headers:


```
content-length: 348
content-type: application/geo+json
date: Fri, 06 Dec 2019 13:51:08 GMT
server: uvicorn
```

Appendix C

Project Management

C.1 Organization

The work for this project has been done by me, but also with the help and supervision of Prof. Stefan Keller, and with the help of Nicola Jordan of the HSR Software Institute.
The source code is open in GitLab with an MIT license (for now).
The source code can be found here: gitlab.com/labiangashi/python-wfs-server

C.2 Planning and Coordination

Meetings have been held regularly every week in Zurich or in the HSR institute to discuss the week's achievements, obstacles and plans for the next week. An agile software development process was used and each sprint lasted 1 to 2 weeks. The project started on 15 Sep 2019 and lasted around 8 sprints.

The initial sprints were mostly information gathering, defining the functional requirements, the languages to be used, tools, libraries, software architecture & design etc.
After those decisions were made and the couple of initial sprints were finished, then began the phase of developing the application, this phase was preliminary finished in about 4 sprints.
After this phase, for the last two sprints, began the phase of testing (unit testing, load testing, integration testing etc) and making sure everything is fine before dockerizing the application.
The last phase was the phase of documenting the project.

C.3 Workflow

The application was coded using the PyCharm IDE, which is a Python IDE for professional Python developers, it offers a lot of features and is all around a great environment for developing in Python.
As mentioned above in the Implementation chapter, the code repository and CI/CD tools for this application was GitLab.
Here are some of the graphs that were taken from GitLab for this project:

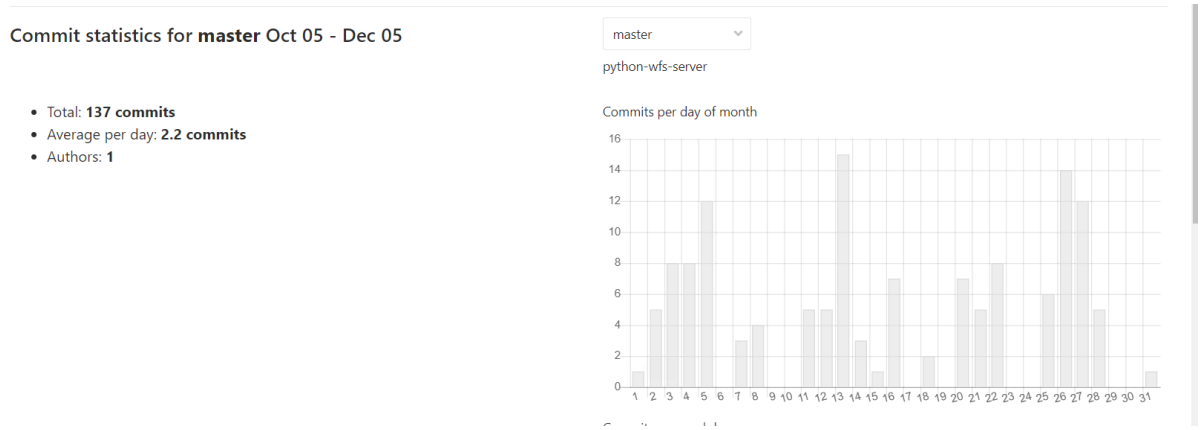

FIGURE C.1: Information about GitLab commits

FIGURE C.2: Information about CI/CD pipelines

Bibliography

[1] Open Geospatial Consortium. Ogc api - features - part 1: Core. 14.10.2019. URL https://docs.opengeospatial.org/is/17-069r3/17-069r3.html.

[2] GDAL. Ogc api - features driver, . URL https://gdal.org/drivers/vector/oapif.html. [Last Accessed: 10.05.2019].

[3] GDAL. Gdal, . URL https://gdal.org/. [Last Accessed: 10.05.2019].

[4] OpenStreetMap. lippy map tilenames. URL https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames. [Last Accessed: 12.11.2019].

[5] Dave Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. 22.04.2012. URL https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html.

[6] TutorialsPoint. Python - basic syntax. URL https://www.tutorialspoint.com/python/python_basic_syntax.htm. [Last Accessed: 12.05.2019].

[7] Docker Inc. What is a container? . URL https://www.docker.com/resources/what-container. [Last Accessed: 12.05.2019].

[8] Docker Inc. Docker overview. . URL https://docs.docker.com/engine/docker-overview/. [Last Accessed: 12.10.2019].

[9] Tiangolo Sebastián Ramírez. Fastapi. URL https://fastapi.tiangolo.com/. [Last Accessed: 12.05.2019].

[10] Multiple Authors. The geojson format. 08.2019. URL https://tools.ietf.org/html/rfc7946.

[11] Holger Krekel. pytest documentation. 03.12.2019. URL https://buildmedia.readthedocs.org/media/pdf/pytest/latest/pytest.pdf.

[12] Multiple Authors. What is locust? URL https://docs.locust.io/en/stable/what-is-locust.html. [Last Accessed: 12.05.2019].

[13] LaTeX3 Project. Latex – a document preparation system. URL https://www.latex-project.org/. [Last Accessed: 12.05.2019].

[14] Texmaker. Welcome to texstudio. URL http://texstudio.sourceforge.net/. [Last Accessed: 12.05.2019].