Министерство науки высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)

Факультет информационных технологий и программирования (ФИТиП)

Образовательная программа Программирование и интернет-технологии

Направление подготовки (специальность) 09.04.02 Информационные системы и технологии

ОТЧЕТ
О ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

по теме:

ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ВЕКТОРНОГО ТАЙЛОВОГО СЕРВЕРА

Обучающийся
Академическая группа M42051 — Степанов С. В.

Руководитель практики от профильной организации,
ООО «Цифровое проектирование»,
руководитель научной лаборатории — Ашихмин И. А.

Руководитель практики от университета,
Университет ИТМО,
факультет информационных технологий и программирования,
доцент — Зубок Д. А.

Практика пройдена с оценкой _____

Дата 01.02.2022

Санкт-Петербург
2022

Реферат

Данный отчет состоит из 15 страниц. Содержит в себе 3 иллюстрации, 4 листинга кода, 8 использованных источников. В данном отчете проводится проектирование и реализация векторного тайлового сервера для системы автоматического формирования генеральных планов площадных объектов капитального строительства. В рамках данной работы были проанализированы функциональные и нефункциональные требования, спроектирована архитектура сервиса и реализована запроектированная архитектура.

Содержание

Определения, обозначения и сокращения **4**

Введение **5**

Постановка задачи **5**

Требования **6**

Функциональные требования 6

Нефункциональные требования 6

Архитектура сервиса **7**

Реализация **9**

Примеры кода **10**

Заключение **14**

Список литературы **15**

Определения, обозначения и сокращения

В настоящем отчете применяют следующие термины с соответствующими определениями.

Генеральный план (генплан, ГП) в общем смысле — проектный документ, на основании которого осуществляется планировка, застройка, реконструкция и иные виды градостроительного освоения территорий. Основной частью генерального плана (также называемой собственно генеральным планом) является масштабное изображение, полученное методом графического наложения чертежа проектируемого объекта на топографический, инженерно-топографический или фотографический план территории.

Тайловая, плиточная или знакоместная графика — метод создания больших изображений (как правило, уровней в компьютерных играх) из маленьких фрагментов одинаковых размеров, подобно картинам из изразцов.

Векторные тайлы — это способ доставки географических данных небольшими порциями в браузер или другое клиентское приложение. Векторные тайлы похожи на растровые тайлы, но вместо растровых изображений возвращаемые данные являются векторным представлением объектов в листе. Например, векторный тайл GeoJSON может включать дороги как LineStrings и водоёмы как Polygons.

Mapbox Vector Tile (MVT) — формат передачи векторных тайлов.

Введение

При работе с картами на различных устройствах, мы очень редко задумываемся о том, как же, на самом деле, работает отображение геоданных. Какие технологии используются, чтобы мы смогли увидеть картинку у себя на экране монитора.

Способ отображения карт можно разбить на 2 больших класса: рендеринг на стороне клиента и рендеринг на стороне сервера. В первом случае, в качестве ответа приходят необработанные геоданные и рендеринг осуществляется на машине клиента. Во втором случае, клиенту уже приходят полностью подготовленные для отображения данных, которые клиенту следует только отобразить, никаких математических вычислений ему производить не нужно.

Эти два подхода используются в разных случаях, в зависимости от задач. При отображении большого объема геоданных следует выбирать подход с рендерингом на стороне сервера. Это позволит снизить количество данных, передаваемых по сети, снизить нагрузку на машину клиента, так как не требуется преобразования географических координат в координаты на экране.

Отрендеренные на сервере геоданные называются тайлы. А сам сервер – тайловым сервером. Тайлы можно разбить на 2 класса: растровые и векторные. Растровые тайлы используются в качестве подложки карты(спутниковые снимки и пр.). Их применяют там, где не нужна интерактивность взаимодействия данных с пользователем. Векторные тайлы – это подготовленный кусочек геоданных большого объема. Их используют там, где требуется интерактивность работы с пользователем. Например: пользователь навел на объект мышкой и этот объект подсветился.

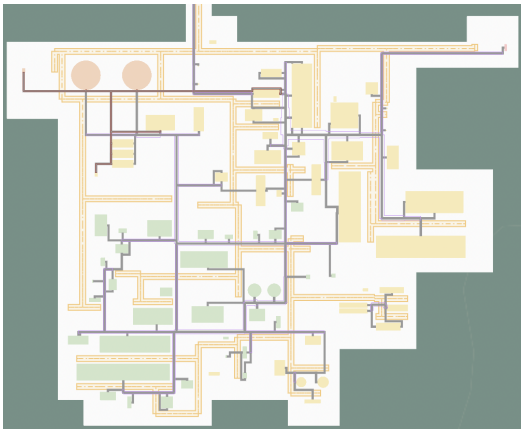

Рис. 1. Пример результата работы

У нас, как раз, случай, с отображением большого количества геоданных. Генеральный план площадного объекта содержит большое количество различных зданий, коммуникаций и других объектов необходимых для функционирования этого площадного объекта(см. рис1). И так как мы хотим дать пользователю взаимодействовать с геоданными, то векторные тайлы позволят решить нам нашу проблему.

Постановка задачи

Основной целью данной работы является спроектировать и реализовать сервис генерации векторных тайлов для набора геоданных.

Исходя из поставленной цели можно выделить следующие *задачи*:

1. Проанализировать функциональные и нефункциональные требования к сервису.
2. Опираясь на требования спроектировать архитектуру системы.
3. Реализовать спроектированную архитектуру.

Требования

В данном пункте кратко перечислим те функциональные и нефункциональные требования, с которыми пришлось столкнуться в рамках реализации тайлового сервера.

Функциональные требования

В рамках функциональных требований к сервису можно выделить следующее:

1. API должен предусматривать получение тайла по четырем параметрам:
   - **site_plan_id** – идентификатор отображаемого генплана.
   - **x** – координаты тайла по оси X.
   - **y** – координата тайла по оси Y.
   - **zoom_level** – уровень отдаления карты.
2. Формат векторных тайлов должен удовлетворять спецификации *Mapbox Vector Tile 2.1*
3. Предусмотреть механизм кэширования отрендеренных тайлов.
4. Предусмотреть механизм инвалидации кэша тайлового сервера.

Нефункциональные требования

Из нефункциональных требований выделим следующие:

1. Язык программирования **Python 3.8**
2. Получение данных из базы данных **PostgreSQL 12**
3. Развертывание сервиса осуществлять с помощью **Docker**

Архитектура сервиса

Для решения задачи генерации векторных тайлов для набора геоданных предлагается следующая архитектура.

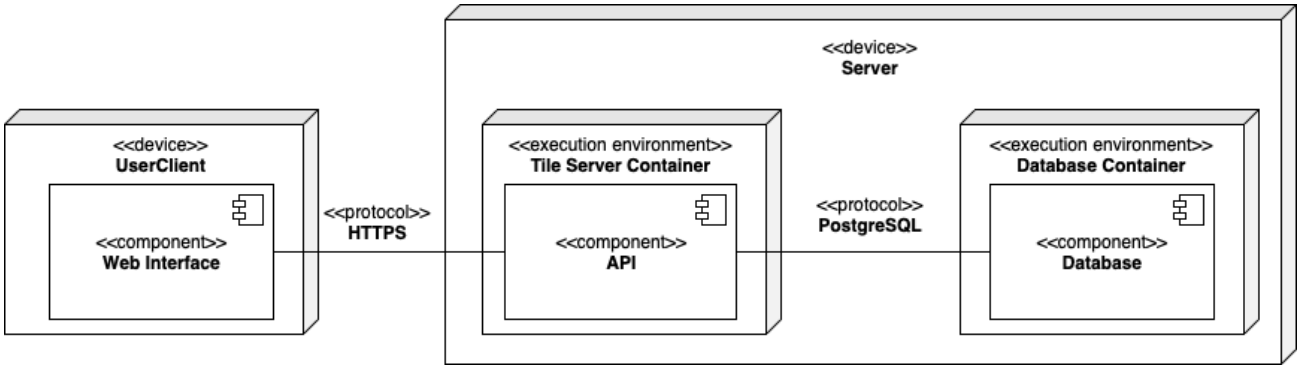

Рис. 2. Диаграмма размещения

Компоненты представленные на диаграмме размещения (см. рис 2):

- **Web Interface** – веб-интерфейс, позволяющий осуществлять взаимодействие с системой пользователю.
- **Tile Server Container**
  - API – веб-сервер, предоставляющий API для получения векторных тайлов.
- **Database Container**
  - Database – база данных (БД), используемая для хранения данных генеральных планов.

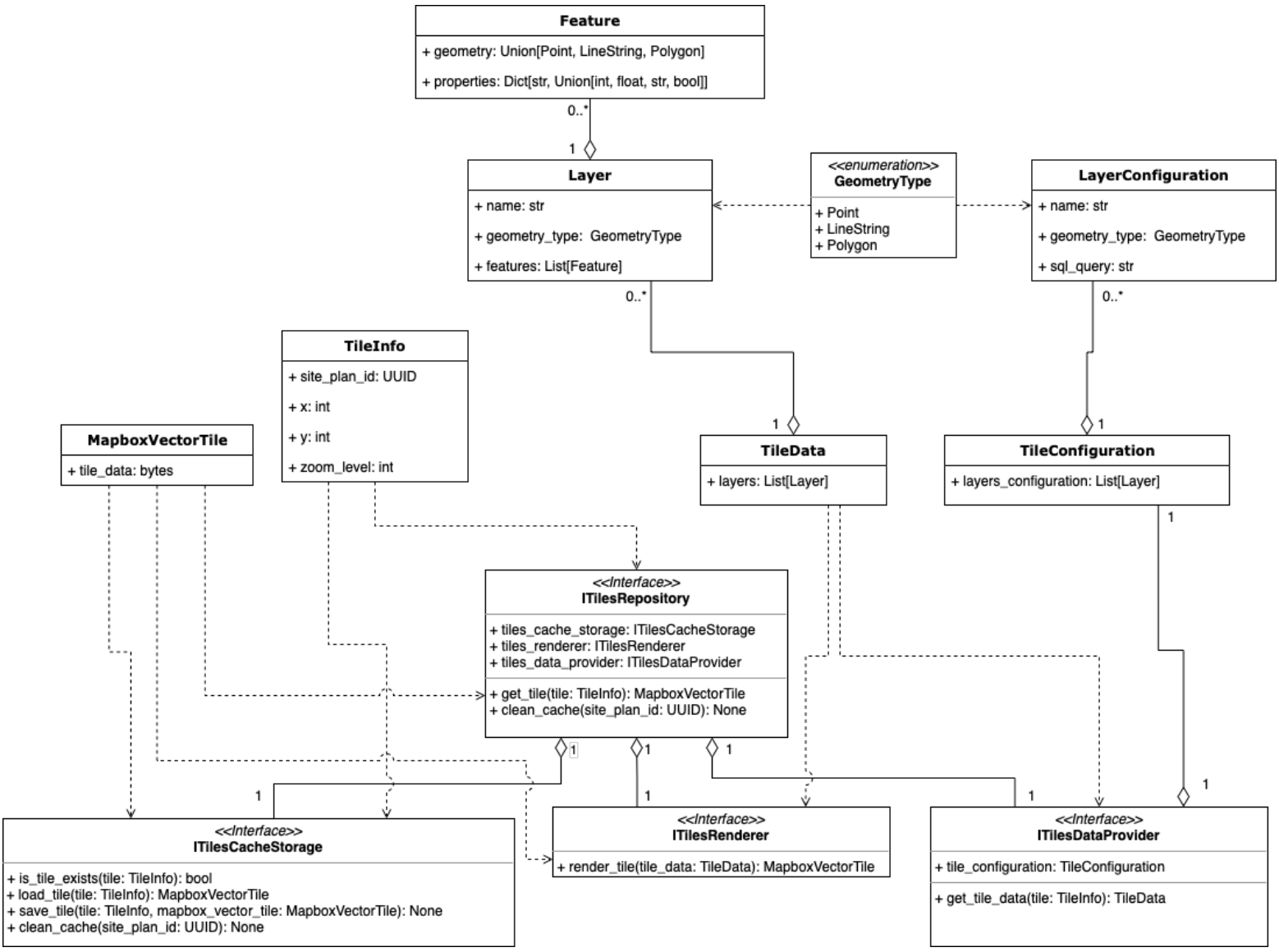

Рис. 3. Диаграмма классов

Основное ядро бизнес-логики проекта представлено на диаграмме классов(см. рис 3). Основные классы и интерфейсы:

1. *ITilesCacheStorage* – сервис, отвечающий за кэширование векторных тайлов.
2. *ITilesRenderer* – сервис, отвечающий за рендеринг векторных тайлов.
3. *ITilesDataProvider* – сервис, отвечающий за получение данных из внешних источников для рендеринга.
4. *ITilesRepository* – оркестратор, отвечает за получение отрендеренных тайлов. Если тайла нет в кэше, то он с помощью *ITilesDataProvider* получает данные для рендеринга, потом с помощью *ITilesRenderer* рендерит тайлы, а после сохраняет их в кэш *ITilesCacheStorage*.

5. **TileInfo** – хранит в себе координаты запрашиваемого тайла, а также уникальный идентификатор генплана.
6. **TileData** – хранит в себе сырые данные, запрошенные по **TileInfo**. Из сырых данных генерируется **MapboxVectorTile**.
7. **TileConfiguration** – хранит в себе параметры слоев, которые требуется получить из источника данных.
8. **MapboxVectorTile** – хранит в себе данные, соответствующие спецификации *Mapbox Vector Tile 2.1*.

Реализация

При реализации описанной архитектуры была получена следующая структура проекта.

```
/
├── app
│   ├── api
│   │   ├── __init__.py
│   │   ├── contexts.py
│   │   ├── controller.py
│   │   ├── model.py
│   │   ├── responses.py
│   │   └── views.py
│   ├── domain
│   │   ├── __init__.py
│   │   ├── cache_storage.py
│   │   ├── data_provider.py
│   │   ├── model.py
│   │   ├── renderer.py
│   │   └── repository.py
│   ├── execution
│   │   ├── __init__.py
│   │   ├── executor.py
│   │   ├── handler.py
│   │   ├── listener.py
│   │   ├── queue.py
│   │   ├── result.py
│   │   └── storage.py
│   ├── repository
│   │   ├── __init__.py
│   │   ├── cache_storage.py
│   │   ├── data_provider.py
│   │   ├── renderer.py
│   │   └── repository.py
│   ├── __init__.py
│   ├── app.py
│   ├── exceptions.py
│   └── logger.py
├── Dockerfile
├── README.md
└── main.py
```

```
└── requirements.txt
```

Само приложение находится в директории **app**. Всего получилось 4 python-пакета, в которых и скрыта основная логика работы.

1. **api** – реализует API тайлового сервера через HTTP. То есть endpoint-ы, модели запросов/ответов к серверу, а также взаимодействие с *ITilesRepository*
2. **domain** – пакет, в котором определены только интерфейсы взаимодействия между модулями программы. Именно здесь и реализована диаграмма классов (см. рис 3). Непосредственная реализация обозначенных интерфейсов находится в других пакетах.
3. **execution** – пакет, в котором вызывается код, отвечающий за рендеринг в отдельном процессе. Рендеринг является CPU-bound задачей, поэтому правильнее запускать его через отдельный процесс, чтобы он не воздействовал на главный процесс приложения.
4. **repository** – в этом пакете находится реализация интерфейсов из **domain**.

Примеры кода

```python
from app import startup_app

if __name__ == '__main__':
    from aiohttp import web

    web.run_app(startup_app(), host="0.0.0.0", port=8080)
```

Листинг 1. main.py

```
FROM python:3.8@sha256:4c4e6735f46e7727965d1523015874ab08f71377b3536b8789ee5742fc737059

WORKDIR /app

ENV LC_ALL C.UTF-8
ENV LANG C.UTF-8
ENV N_WORKERS 8

COPY requirements.txt .
RUN pip3 install --no-cache-dir -r requirements.txt

RUN pip3 check

COPY main.py .

COPY /app ./app

ENTRYPOINT /bin/bash -c "gunicorn run_app:app --workers=${N_WORKERS} --bind 0.0.0.0:8080 --worker-class
```

Листинг 2. Dockerfile

```python
from enum import Enum
from typing import Dict
from typing import List
from typing import Type
from typing import Union
from uuid import UUID

from dataclasses import dataclass
from shapely.geometry.base import BaseGeometry


@dataclass
class TileInfo:
    site_plan_id: UUID
    x: int
    y: int
    z: int

class GeometryType(Enum):
    POINT = "point"
    LINESTRING = "line"
    POLYGON = "polygon"

@dataclass
class Feature:
    geometry: Type[BaseGeometry]
    properties: Dict[str, Union[int, float, str, bool]]

@dataclass
class Layer:
    name: str
    geometry_type: GeometryType
    features: List[Feature]

@dataclass
class TileData:
    layers: List[Layer]


@dataclass
class LayerConfiguration:
    layer_name: str
    geometry_type: GeometryType
    sql_query: str

@dataclass
class TileConfiguration:
    layers_configurations: List[LayerConfiguration]
```

Листинг 3. domain/model.py

```python
import traceback
from pathlib import Path
from uuid import UUID
```

```
4
5  from aiohttp import web
6
7  from app.exceptions import TileDoesNotExistException
8  from app.domain.model import TileInfo
9  from app.repository import create_tiles_repository
10
11
12 async def index(request):
13     static_dir = request.config_dict['static_dir']
14     return web.FileResponse(static_dir / Path('index.html'))
15
16
17 async def vector_tiles(request):
18     tile = TileInfo(
19         site_plan_id=UUID(request.rel_url.query['task_id']),
20         x=int(request.rel_url.query['x']),
21         y=int(request.rel_url.query['y']),
22         z=int(request.rel_url.query['z'])
23     )
24
25     vector_data_service = request.config_dict['vector_data_service']
26     try:
27         tile_data: bytes = tiles_repository.get_tile(tile)
28         return web.Response(
29             body=tile_data,
30             status=200,
31             content_type="application/x-protobuf",
32             headers={
33                 'Access-Control-Allow-Origin': '*',
34             }
35         )
36     except TileDoesNotExistException as e:
37         return web.Response(
38             status=404,
39             text=str(e),
40             headers={
41                 'Access-Control-Allow-Origin': '*'
42             }
43         )
44     except Exception as e:
45         tb = traceback.format_exc()
46         print(tb)
47         return web.Response(
48             status=500,
49             text=tb,
50             headers={
51                 'Access-Control-Allow-Origin': '*'
52             }
53         )
54
55
56 async def clean_cache_for_task(request):
57     task_id = UUID(request.rel_url.query['task_id'])
58
```

```
59      tiles_repository = request.config_dict['tiles_repository']
60      tiles_repository.clean(task_id)
61      return web.Response(
62          status=200,
63          headers={
64              'Access-Control-Allow-Origin': '*',
65          }
66      )
67
68
69  def startup_app():
70      app = web.Application()
71
72      tiles_repository = create_tiles_repository()
73      static_dir = Path(__file__).parent / Path('static')
74
75      app['tiles_repository'] = tiles_repository
76      app['static_dir'] = static_dir
77
78      app.add_routes([
79          web.get("/", index),
80          web.get("/index", index),
81          web.get("/vector", vector_tiles),
82          web.get("/clean", clean_cache_for_task),
83      ])
84      return app
```

Листинг 4. domain/model.py

Заключение

В данной работе был спроектировать и реализовать сервис генерации векторных тайлов для отображения генерального плана площадного объекта. С учетом функциональных и нефункциональных требований была спроектирована архитектура системы, а так же эта архитектура была реализована.

Список литературы

1. Python 3.8 Documentation [Интернет ресурс]:
   URL:https://docs.python.org/3.8/(дата обращения: 28.01.22)

2. PostGIS Reference [Интернет ресурс]:
   URL:https://postgis.net/documentation/ (дата обращения: 28.01.22)

3. Glenford J. Myers, Corey Sandler, Tom Badgett "The Art of Software Testing 3rd Edition, 16 Dec 2011

4. Docker Reference[Интернет ресурс]:
   URL:https://docs.docker.com (дата обращения: 28.01.22)

5. Sphinx Python Documentation [Интернет ресурс]:
   URL:https://www.sphinx-doc.org/en/master/ (дата обращения: 28.01.22)

6. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition

7. Martin Robert C. "Clean Code: A Handbook of Agile Software Craftsmanship"Aug 1, 2008