Verslag grote opdracht databanken 1

Databank en API voor indoor kaartgegevens (Android app backend)

Ik koos als grote opdracht om een backend te schrijven voor de Android-applicatie die ik voor het OPO Mobiele Toepassingen ontwikkel. Deze app, genaamd “Andin”, voor “Android Indoor” heeft als belangrijkste functie het weergeven van indoor kaarten. Een indoor kaart is te vergelijken met een traditionele stafkaart, maar dan binnenshuis. Zo kunnen er bijvoorbeeld lokalen, gangen, trappen of zelfs tafels of vuilbakken op worden aangeduid.

Om dit uiteindelijk voor bijvoorbeeld alle grote gebouwen in België te bereiken kunnen de kaartgegevens voor die gebouwen niet allemaal in de app worden opgeslagen, tenminste bij realistische opslag-beperkingen. Daarom is er nood aan een externe service die op aanvraag kan worden geraadpleegd. De rol van die externe service tracht ik met dit project te vervullen.

Specificatie

Zoals ik eerder duidde is het doel van dit project om een service te maken die op aanvraag gegevens kan bieden voor een mobiele applicatie. Vooraleer ik daarvoor de architectuur kan bepalen is het belangrijk om vast te leggen welk type data de service moet leveren en hoe die data gebruikt zal worden.

Type data

Een indoor kaart kan - in het algemeen - op twee manieren worden weergegeven; met zogenoemde tiles (kleine samengenaaide afbeeldingen van de kaart) of met vectordata (een geometrische beschrijving van de vorm van elk kaartelement).

De belangrijkste voordelen van tile-kaarten zijn simpliciteit en performantie van de app (of website) die de kaart weergeeft.

Met vectordata kan men de kaart makkelijker interactief maken, bijvoorbeeld met extra info over een bepaald lokaal of met indoor navigatie. Daarnaast is het detail op een vectorkaart meestal ook groter. Dit is in het bijzonder belangrijk voor kleine kaartelementen zoals lokalen. Daarom kies ik voor vectordata i.p.v. tiles.

In de eerste fase van de ontwikkeling focus ik enkel op lokalen. Het type data dat de service moet leveren is dus de vectordata (geometrie) van lokalen binnen gebouwen. Daarnaast moet er per lokaal en gebouw randdata zoals bijvoorbeeld de naam of het adres kunnen worden opgevraagd.

Gebruik van de data

Voor een mobiele app zoals Andin is het belangrijk dat data in een makkelijk consumeerbaar formaat toegankelijk is, mobiele toestellen hebben namelijk niet de performantie om data te parsen of om complexe filter- of sorteeroperaties te doen.
De te ontwikkelen service moet dus een API (Application Programming Interface) aanbieden in een bekend en makkelijk te verwerken formaat.

Architectuur

Uit de specificatie is gebleken dat de backend service uit twee delen moet bestaan; een databank om geografische en metadata op te slaan en een API om die data makkelijk op te vragen. Ik zal van hieraf respectievelijk als ‘andin-db’ en ‘andin-api’ naar deze delen refereren.

Andin-db

Na een productonderzoek (zie opdracht permanente evaluatie) bleek dat PostgreSQL[1] in combinatie met PostGIS[2] de meest geschikte databank is voor dit project is. Verder moet dus het datamodel en de bron van de data bepaald worden.

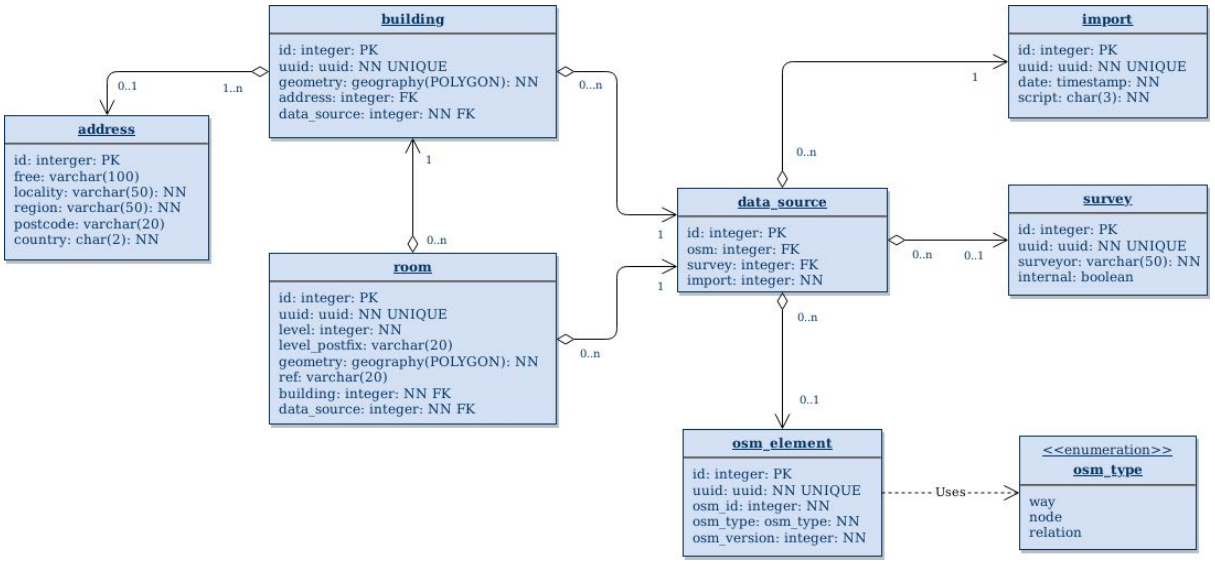
Data

Voor beperkte tests kan manueel ingevoerde data worden gebruikt. Maar voor een grotere schaal is er een bron van bestaande data nodig. Hiervoor is OpenStreetMap[3] perfect geschikt. OpenStreetMap (van hieraf ‘osm’ genoemt) is een open databank met geografische gegevens voor de hele wereld. *Open* in deze context betekent dat iedereen data kan toevoegen en wijzigen en dat die data vrij bruikbaar is in andere applicaties (onder bepaalde voorwaarden). Gegevens uit osm kunnen gedownload en verwerkt worden voor het datamodel van andin-db, zie het deel ‘Uitwerking’.

Datamodel

De twee belangrijkste entiteiten zijn gebouwen (‘building’) en lokalen (‘room’). Verder zijn er nog relaties met onder andere ‘survey’ en ‘osm_element’ om de bron van de data aan te duiden. Entiteiten die publiek toegankelijk moeten zijn geef ik automatisch een UUID (Universally Unique Identifier). Zo is een bepaald element niet gelinkt aan een interne en dus database-specifieke id (‘id’ in elke entiteit). Een extra voordeel hiervan is dat een eventuele gebruiker met slechte bedoelingen via de API moeilijker informatie kan vergaren over de interne werking van het systeem. Hieronder volgt het functioneel diagram.

[1] https://www.postgresql.org
[2] https://postgis.net
[3] https://welcome.openstreetmap.org



Andin-api

De API moet een gebruiksvriendelijke kijk geven op de data in andin-db. Voor deze fase van het project beperk ik de API tot enkel lezen.

Protocolkeuze

Een veelgebruikte optie voor moderne API's is HTTP REST (Representational state transfer). Dit type API houd in essentie in dat een programma objecten via standaard HTTP methoden zoals GET, PUT etc. kan raadplegen en manipuleren. Dit model wordt meestal gecombineerd met het JSON (JavaScript object notation) dataformaat. Voordelen van dit model zijn onder andere simpliciteit voor de consumerende applicatie (HTTP en JSON zijn alomtegenwoordig) en een simpele spiegeling van een relationeel datamodel.

Een bekend nadeel van REST API's is over-fetching. Een consument van de API beslist namelijk (meestal) niet welke data er voor een bepaald endpoint (entiteit) wordt teruggekeerd. Zo krijgt de consument meestal veel meer data dan eigenlijk nodig was. Bij under-fetching krijgt een consument in dezelfde geest te weinig data.

Bijvoorbeeld als een consument alle lokalen van het dichtstbijzijnde gebouw wil ophalen moet die: eerst een lijst van gebouwen in de buurt ophalen, dan alle kamers in het dichtstbijzijnde gebouw en uiteindelijk info over elk van die kamers. Dit zijn voor 20 lokalen al minstens 2+20=22 verschillende oproepen, wat al snel in tijd kan oplopen. Daarnaast wordt er ook veel onnodige data meegestuurd, bijvoorbeeld de vorm van het gebouw, wat geen deel was van de originele vraag.

GraphQL[4] is een protocol dat deze en nog andere problemen van REST API's tracht op te lossen. Over-fetching is bij GraphQL niet mogelijk aangezien de consument specifieert

[4] https://graphql.org

welke data nodig is. Ook under-fetching is geen probleem omdat men data kan opvragen over objecten die men nog niet heeft ontvangen. Bijvoorbeeld “haal gebouw A op en als dat succesvol is ook alle lokalen in dat gebouw”. In een REST-model zou dit meerdere API oproepen vereisen. Met GraphQL kan dit in één enkele oproep.

Ook de twee besproken voordelen van een REST-model blijven bij GraphQL behouden. GraphQL is namelijk protocol-agnostisch (en dus HTTP compatibel) en geeft resultaten terug in JSON.

GraphQL model

Een textuele versie van het graphql-model is toegankelijk als ‘model.gql’ in de andin-db repository (zie implementatie).

Keuze programmeertaal

Ik koos ‘go’[5] als programmeertaal voor andin-api. Dit omdat het een modern alternatief is op performante gecompileerde talen zoals C++ of C#. Een belangrijk voordeel dat go van hogere programmeertalen leent is garbage collection. Dit betekend dat de programmeur zich geen zorgen moet maken over bv. geheugen-allocatie. Go is daarnaast ook een functionele taal en daardoor uitermate geschikt om complexe modellen met simpele pure functies uit te drukken.

Implementatie

De implementatie van beide delen van het project, (db en api) zijn respectievelijk te vinden op https://github.com/ubipo/andin-db en https://github.com/ubipo/andin-api. In dit hoofdstuk geef ik een hoog-niveau beschrijving van de werking van beide delen.

Andin-db

Zie ook https://github.com/ubipo/andin-db met commentaar bij scripts en relevante ‘.md’ (README) bestanden.

Migrations

Om het besproken model in PostgreSQL te implementeren maakte ik gebruik van de ‘golang-migrate[6]’ migration manager. Zie het bestand ‘MIGRATE.md’ voor informatie over het gebruik van deze tool. De verschillende stappen die ik ondernam om het uiteindelijke model te bekomen zijn gedefinieerd als ‘.sql’ bestanden in de folder ‘migrations’.

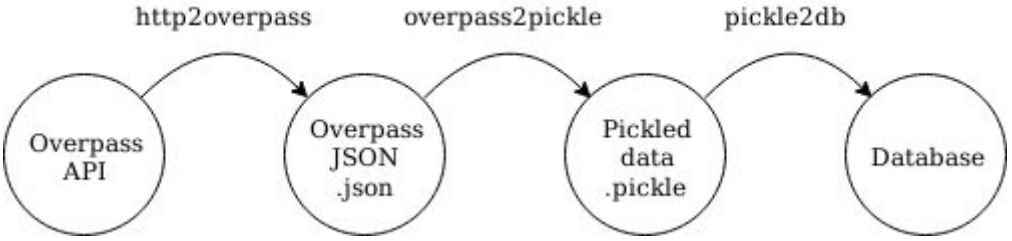
[5] https://golang.org/
[6] https://github.com/golang-migrate/migrate

Testdata

Om de API te testen implementeerde ik een script om een beperkt aantal lokalen in de database toe te voegen (‘insert.py’). Dit script maakt gebruik van de psycopg2 python library om met de database te connecteren. Zie ook ‘TESTDATA.md’

Osm-data

Om osm-data in de database te importeren implementeren ik een reeks scripts die gebruik maken van de Overpass API[7]. Dit is een service die met behulp van een speciale taal (Overpass Query Language, OQL) gefilterde osm data biedt (zoals bijvoorbeeld alle gebouwen in Leuven). Hieronder een overzicht van deze scripts.



- http2overpass.py download osm data van de Overpass api naar een json bestand
- overpass2pickle.py converteerd dit json bestand naar een geserialiseerd .pickle bestand met enkel de relevante data en de adressen van elk gebouw (met behulp van de Nominatim API[8])
- pickle2db upload het .pickle bstand naar de databank met behulp van psycopg2

Andin-api

Zie ook https://github.com/ubipo/andin-api met commentaar bij scripts en relevante ‘.md’ (README) bestanden.

Voor de databaseconnectie maakte ik gebruik van de excellente ingebouwde database library van go samen met sqlx[9] om sommige operaties iets simpeler te maken. Om de GraphQL functionaliteit te implementeren maakt ik gebruik van graphql-go[10].

Structuur

De algemene structuur van de code is als volgt:

- cmd/andin-api/main.go - Startcode van de applicatie
- internal/api - Code i.v.m. de geimplementeerd API
    - api.go - Startcode API
    - gql.go - GraphQL types en schema

[7] https://wiki.openstreetmap.org/wiki/Overpass_API
[8] https://nominatim.openstreetmap.org
[9] https://github.com/jmoiron/sqlx
[10] https://github.com/graphql-go/graphql

    - sql.go - SQL queries

API Functionaliteit

Het is via de API mogelijk om:

- Elk object met een UUID op te halen (met bv. ‘building(uuid: “xxx”)’ voor gebouwen)
- Lokalen of gebouwen op te halen rond een bepaald punt (met coördinaten of met een vrij ingegeven locatie)
    - Van een bepaalde grootte
    - Gesorteerd op grootte of op afstand tot het punt
- Vanaf elk object de relaties (zoals beschreven in het datamodel) met andere objecten op te halen, met specifiek:
    - Voor gebouwen: lokalen op een bepaald niveau of met een bepaalde naam
    - Voor lokalen: lokalen die grenzen aan het lokaal op hetzelfde of op andere niveaus of die overlappen met lokalen op een ander niveau

API Demo

Hieronder een demonstratie van een query die: alle gebouwen in Leuven ophaalt, voor elk van die gebouwen het adres, de UUID, en de bron van de data geeft, en elke kamer binnen die gebouwen op niveau 2 en met “pc” in de naam geeft.

```
{
  buildings(distanceFrom: {
    coordinates: {
      lon: 4.71581,
      lat: 50.87850
    },
    place: "Leuven",
    max: 2000,
  },
    area: {
      min: 1000,
      max: 1500
    }
    sort: AREA
  ){
    area
    building {
      uid,
      address {
        free,
        locality,
        region,
        country
      },
      rooms(level: 2, name: "pc") {
        name,
        level
      },
      dataSource {
        import {
          script,
          date
        },
        osm {
          id,
          type,
          version
        }
      }
    }
  }
}
```



```
{
  "data": {
    "buildings": [
      {
        "area": 1420.6289,
        "building": {
          "address": {
            "country": "be",
            "free": "Tiensevest 60",
            "locality": "Leuven",
            "region": "Vlaanderen"
          },
          "dataSource": {
            "import": {
              "date": "2019-12-08T12:43:19.879024Z",
              "script": "osm"
            },
            "osm": {
              "id": 78465129,
              "type": "way",
              "version": "20"
            }
          },
          "rooms": [
            {
              "level": 2,
              "name": "C203 / PC"
            },
            {
              "level": 2,
              "name": "C201 / PC"
            }
          ],
          "uid": "d107c4e6-728d-4fac-b65b-64a90014e32a"
        }
      }
    ]
  }
}
```


Erratum

Ik gebruikte in veel van mijn code verkeerdelijk de afkorting UID ipv UUID (voor Universally Unique Identifier).