Licență

1. Algoritmul Dijkstra

Algoritmul Dijkstra este un algoritm clasic de calea cea mai scurtă și a fost adoptat în multe sisteme ca bază teoretică. În general, este folosit pentru a calcula calea costului minim de la primul nod la toate celelalte noduri. Principiul algoritmului Dijsktra, a fost propus de EWDijkstra în 1959. [1 time]

Algoritmul Dijkstra și variantele acestuia sunt utilizate în mod obișnuit în aplicații precum Google Maps și alte sisteme de rutare a traficului.

Algoritmul Dijkstra este cel mai utilizat algoritm pentru planificarea traseelor. Acesta găsește calea optimă dintre un nod inițial și un nod destinație. Această abordare a fost dezvoltată pentru grafuri ponderate; ponderea fiecărei muchii trebuie să fie egală cu zero sau mai mare decât zero.

Algoritmul Dijkstra funcționează similar cu algoritmul Breadth First Search (BFS), doar ca funcționarea celui din urmă este de a folosi principiul cozii, dar algoritmul folosit de Dijkstra este coada prioritară. Deci numai nodurile care au cea mai mare prioritate vor fi căutate. În determinarea nodului prioritar, acest algoritm compară fiecare valoare (greutate) a fiecărui nod care este stocată pentru a fi comparată cu valoarea care trebuie găsită de la următoarea rută și așa mai departe până când un nod este găsit în căutare.2

Aplicarea algoritmului

Algoritmul Dijkstra menține două liste, lista deschisă și lista închisă. Lista deschisă păstrează nodurile care trebuie examinate, în timp ce lista închisă păstrează nodurile care au fost deja examinate. Algoritmul Dijkstra construiește cea mai scurtă cale de la nodul sursă la toate celelalte noduri. Inițial, cel sursa este atribuit listei deschise. La fiecare pas, nodul cu cel mai mic cost (costul minim de la nodul de pornire la nodul selectat) este selectat din lista deschisă, nodurile atinse sunt adăugate la lista deschisă, iar nodul selectat este transferat din lista deschisă în lista închisă. Când lista deschisă este goală sau nodul destinație este adăugat la lista închisă, atunci algoritmul se termină. Originalul Algoritmu Dijkstra are o complexitate de 2 O (n) , unde n reprezintă numărul de noduri din graf. Atunci când se utilizează o structură de date de cozi prioritare, complexitatea algoritmului poate fi redusă la O( ) m nl  ogn , unde m reprezintă numărul de muchii din graf.

Pseudocod Dijkstra

crează o listă goală C pentru closed nodes

crează o listă O pentru open nodes și valorile f respective containing care conține nodul de start

cât timp O nu este goală:

extrage nodul n din O cu cea mai bună valoare pentru f

Dacă n este target:

return soluție

pentru fiecare m care este vecin cu n:

Dacă (m nu este în C) și (m nu este în O):

adaugă m la O, setează n ca părinte a lui m

calculează g(m) și f(m) și salvează-le

altfel:

Dacă f(m) de la ultima iterație este mai bun decât g(m) de la iterația curentă:

setează n ca parinte a lui m

actualizează g(m) și f(m)

Dacă m este în C:

mută m în O

mută n din O în C

return nu există soluție

Implementarea algoritmului Dijkstra cu Java

public static Node aStar(Node start, Node target){

PriorityQueue<Node> closedList = new PriorityQueue<>();

PriorityQueue<Node> openList = new PriorityQueue<>();

start.f = start.g + start.calculateHeuristic(target);

openList.add(start);

while(!openList.isEmpty()){

Node n = openList.peek();

if(n == target){

return n;

}

for(Node.Edge edge : n.neighbors){

Node m = edge.node;

double totalWeight = n.g + edge.weight;

if(!openList.contains(m) && !closedList.contains(m)){

m.parent = n;

m.g = totalWeight;

m.f = m.g + m.calculateHeuristic(target);

openList.add(m);

} else {

if(totalWeight < m.g){

m.parent = n;

m.g = totalWeight;

m.f = m.g + m.calculateHeuristic(target);

if(closedList.contains(m)){

closedList.remove(m);

openList.add(m);

}

}

}

}

openList.remove(n);

closedList.add(n);

}

return null;

}

Explicare functionare

2. Android Studio

Android Studio este un mediu de dezvoltare integrat (IDE- Integrated Development Environment) pentru dezvoltarea de aplicații Android. Este bazat pe IntelliJ IDEA, un mediu de dezvoltare java pentru software și încorporează editarea codului și instrumente pentru dezvoltatori. Sprijină dezvoltarea aplicațiilor în cadrul sistemului de operare Android. Android Studio utilizează a Sistem de building bazat pe Gradle, emulator, șabloane de cod și integrare GitHub. Fiecare proiect din Android Studio are acces la cod sursă și fișiere de resurse. Acestea includ module de aplicații Android, module de biblioteci și module Google Application Engine.-FLUTER 1

Aplicația Sistem de recomandare a fost realizată în Android Studio 11.

3. Flutter

Flutter este un nou SDK pentru aplicații mobile multi platforms oferit de Google, care ajută dezvoltatorii să construiască diverse aplicații pentru sistemul de operare Android și iPhone (iOS), folosind o bază de cod comună. Aplicațiile create folosind Flutter sunt scrise în limbajul Dart. Flutter folosește un cadru reactiv modern permițând dezvoltatorilor să construiască o interfață cu utilizatorul. –FLUTER.

Framework-ul lui Flutter se bazează pe Dart, ca limbaj de programare de bază, Flutter Engine pentru rendering support layer, biblioteci fundamentale pentru furnizarea de clase și funcții de bază pentru aplicație, widget-uri și blocuri vizuale, inclusiv seturi de elemente specifice designului, cum ar fi Material Design de la Google și ghid pentru interfața om/sistem.-Flutter

Instalarea pe mobil

Extensia de fișier Android Application Package (APK) în Open Handset Alliance, care este o platformă de telefonie mobilă, bazat pe Linux, creat atunci când este utilizat Android Kit de dezvoltare software (SDK) a lui Google. Conține fișierele de cod, fișiere de resurse și extensia de fișier XML Manifest Android. Pentru a implementa APK-ul pe un smartphone Android, modul dezvoltator de pe telefon a fost activat mai întâi, urmat de activarea a USB debuging în setările de dezvoltare pentru a permite comunicarea cu computerul. Alternativ, build number poate fi atins de șapte ori, apoi telefonul este conectat la computer folosind USB aplicația poate fi accesată din Android Studio.

Arhitectura sistemului aplicatiei

Explicate fiecare pas din aplicatie

Rularea aplicatiei

Pentru a rula aplicația, Android Studio a fost folosit pentru a începe un nou proiect Flutter, făcând clic pe butonul de start. Proiectul a fost numit Sistem de recomandare, apoi un exemplu de aplicație (schaffold) a fost generat pentru a vedea cum arăta aplicația, urmând apăsarea butonului de terminare. Codul de pornire pentru a crea o aplicație de contor simplu a fost executată, folosind Emulatorul Android. Apoi a fost dezvoltată o aplicație materială și conectat cu widget-ul acasă pentru a insera programarea cod.

API-uri Google maps

Directions Api

Cu API-ul Directions este posibilă:

• Căutarea indicațiilor pentru mai multe moduri de transport, inclusiv tranzit, conducere, mers pe jos sau cu bicicleta.

• Specificați originile, destinațiile și punctele de referință ca șiruri de text (de exemplu, „Chicago, IL” sau „Darwin, NT, Australia”), ca ID-uri de locație sau ca coordonate de latitudine/longitudine. API-ul returnează cele mai eficiente rute atunci când calculează direcțiile.

Timpul de călătorie este factorul principal optimizat, dar API-ul poate lua în considerare și alți factori, cum ar fi distanța, numărul de viraje și mulți alții atunci când decide care rută este cea mai eficientă.

Parametri necesari:

Destinaţie: ID-ul locului, adresa sau valoarea textului latitudine/longitudine către care doriți să calculați indicațiile de orientare. Opțiunile pentru parametrul destinație sunt aceleași ca și pentru parametrul origine.

Origine: ID-ul locului, adresa sau valoarea textului de latitudine/longitudine din care doriți să calculați indicațiile de orientare.

Places Autocomplete Api

Serviciul de completare automată a locației este un serviciu web care returnează predicții de locație ca răspuns la o solicitare HTTP. Solicitarea specifică un șir de căutare textual și limite geografice opționale. Serviciul poate fi folosit pentru a oferi funcționalitate de completare automată pentru căutări geografice bazate pe text, prin returnarea unor locuri precum companii, adrese și puncte de interes.

Parametri necesari

input

Șirul de text pe care se caută. Serviciul de completare automată a locurilor va returna potrivirile candidatului pe baza acestui șir și va ordona rezultatele în funcție de relevanța lor percepută.

Obținerea datelor din Google maps Apis

Pentru a utiliza Google Maps Platform, trebuie creat un proiect pentru a gestiona serviciile, acreditările, facturarea, API-urile și SDK-urile.

Produsele Google Maps Platform sunt protejate împotriva utilizării neautorizate prin restricționarea apelurilor API la cele care oferă acreditări de autentificare adecvate. Aceste acreditări sunt sub forma unei chei API - un șir alfanumeric unic care asociază contul dvs. de facturare Google cu proiectul dvs. și cu API-ul sau SDK-ul specific. Este necesară includerea cheii API cu fiecare solicitare API Directions sau API Places.

Amadeus for Developers Api

API-urile Amadeus Travel oferă cele mai bogate informații din industria călătoriilor.

Pentru a integra un api in aplicație este necesar sa fie urmați urmatorii pași pentru autorizare:

Configurarea mediului

La crearea unei aplicații pe portalul Amadeus for developers va fi atribuit contului o pereche de chei: API key și API secret.

În prima parte a autorizației trebuiesc introduse grant type, API key și API secret. Aceasta returnează token-ul de acces necesar pentru a returna raspunsul API-ului.

var resultsHotels = await client.post(  
 Uri.parse('https://test.api.amadeus.com/v1/security/oauth2/token'),  
 body: {  
 "grant\_type": "client\_credentials",  
 "client\_id": "zHmPH2go7aCsH6qAigzfbvSjNj2EvaA1",  
 "client\_secret": "rIJW2hknmn7g4o5w",  
 },  
);

bearerToken este o variabilă folosita pentru a stoca access\_token-ul provenit de la apelul de autorizarea API concatenat cu token type și adăugarea acesteia la antetul fiecărui apel API. Access\_token-ul este valabil timp de 30 de minute, este salvat și injectat automat în antetul apelurilor API.

if (resultsHotels.statusCode == 200) {  
 try {  
 var security = jsonDecode(resultsHotels.body);  
 if (security != null) {  
 var tokenType = security['token\_type'];  
 var token = security['access\_token'];  
 var bearerToken = '$tokenType ' + '$token';  
 var response = await client.get(Uri.parse(airUrl),  
 headers: {  
 "Authorization": bearerToken,  
 });

Firebase

Firebase a fost folosit pentru stocarea și sincronizarea datelor.

Baza de date Firebase Realtime este o bază de date non-relațională baza de date care funcționează sub schema Platform-As-A-Service și permite stocarea informațiilor în format JSON.-FP383 Aici lista de dispozitive vor fi stocate, fiecare identificat prin „Activo Cod Fijo". La fel, pentru fiecare va fi salvat:

• Locația fizică a dispozitivului

• ID-ul calendarului de afișat pe dispozitiv

• Anunțul, care este opțional

• Calea imaginii de afișat, care este stocată în Stocare Firebase.

Spring Boot

Spring Boot este un framework care se bazează pe limbajul java, care este în esență un container de dependency injection. Spring Framework oferă un model cuprinzător de programare și configurare pentru aplicațiile de întreprindere moderne bazate pe Java - pe orice tip de platformă de implementare. Spring se bazează pe conceptul de „beans” sau „componente”, care sunt practic singleton-uri fără dezavantajele modelului singleton tradițional. Cu injecția de dependență, fiecare componentă doar declară colaboratorii are nevoie, iar Spring le oferă în timpul rulării. Cel mai mare avantaj este că se poate injecta cu ușurință diferite instanțe pentru diferite implementări ale aplicației.

Conectarea Spring boot cu firebase

Urmatoarea dependință este adăugată în pom.xml. Pom.xml definește configurația Meaven a aplicației.

<dependency>  
 <groupId>com.google.firebase</groupId>  
 <artifactId>firebase-admin</artifactId>  
 <version>6.11.0</version>  
 </dependency>

Urmează să fie inițializată baza de date Firebase prin urmatorul cod java:

package com.example.demo.firebase;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Service;

import com.google.auth.oauth2.GoogleCredentials;

import com.google.firebase.FirebaseApp;

import com.google.firebase.FirebaseOptions;

@Service

public class FirebaseInitialization {

@PostConstruct

public void initialization() throws FileNotFoundException {

FileInputStream serviceAccount =null;

try {

serviceAccount = new FileInputStream("./serviceAccount.json");

FirebaseOptions options = new FirebaseOptions.Builder()

.setCredentials(GoogleCredentials.fromStream(serviceAccount))

.setDatabaseUrl("https://fir-db-for-spring-boot-5a0ff-default-rtdb.firebaseio.com")

.build();

FirebaseApp.initializeApp(options);

}catch(Exception e) {

e.printStackTrace();

}

}

}

@Service și @PostConstruct sunt două adnotări din Spring Boot.

@PostConstruct arată că metoda adnotată trebuie să ruleze înainte ca service să fie injectat în context.

A fost folosit @Service pentru a indica faptul în clasă este scrisă logică bussines într-un layer diferit (service layer). Pe lângă faptul că este utilizat în service layer, nu există nicio altă utilizare specială pentru această adnotare.

Prima linie citește configurațiile din fișierul JSON și apoi inițializează conexiunea pentru baza de date specificată.

A urmat crearea clasei model. Modelul funcționează ca un container care conține datele aplicației, geterii și seterii necesari.

public class Product {

private String name;

private String weight;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getWeight() {

return weight;

}

public void setWeight(String weight) {

this.weight = weight;

}

}

Clasa service conține funcții pentru operațiile de GET (getAllProductsDetails), POST (saveProduct), UPDATE (updateProduct) și DELETE (deleteProduct) aplicate bazei de date.

FirestoreClient oferă acces la Google Cloud Firestore. Acest API este utilizat pentru a obține o instanță Firestore, care oferă metode pentru actualizarea și interogarea datelor în Firestore. Funcția getFirestore() returnează instanța Firestore asociată cu aplicația Firebase default.

public String saveProduct(Product product, String COLLECTION\_NAME) throws InterruptedException, ExecutionException {

Firestore dbFirestore=FirestoreClient.getFirestore();

ApiFuture<WriteResult> collectionApiFuture = dbFirestore.collection(COLLECTION\_NAME).document(product.getName()).set(product);

return collectionApiFuture.get().getUpdateTime().toString();

}

EXPLICATIE

**public** String deleteProducts() **throws** InterruptedException, ExecutionException {

ArrayList<String> allDocuments = **new** ArrayList();

allDocuments.add("carNodes");

allDocuments.add("trainNodes");

allDocuments.add("hotelNodes");

allDocuments.add("flyNodes1");

allDocuments.add("flyNodes2");

allDocuments.add("flyNodes3");

allDocuments.add("flyNodes4");

allDocuments.add("flyNodes5");

allDocuments.add("activitiesNode1");

allDocuments.add("activitiesNode2");

allDocuments.add("activities3Node");

allDocuments.add("activities4Node");

allDocuments.add("activities5Node");

allDocuments.add("activities6Node");

allDocuments.add("activities7Node");

allDocuments.add("activities8Node");

allDocuments.add("activities9Node");

allDocuments.add("activities10Node");

allDocuments.add("products");

Firestore dbFirestore=FirestoreClient.*getFirestore*();

**for**(**int** j=0;j<allDocuments.size();j++) {

List<Product> pList;

pList=getAllProductsDetails(allDocuments.get(j));

**for**(**int** i=0;i<pList.size();i++) {

ApiFuture<WriteResult> collectionApiFuture = dbFirestore.collection(allDocuments.get(j)).document(pList.get(i).getName()).delete();

}

}

**return** "The products was deleted successfluy";

}

În continuarea am realizat operațile CRUD, am dezvoltat controlerul REST care va ajuta în interacțiunea cu service layer, folosind adnotarea @RestController.

Funcția saveCarNodes este o funcție care salvează în baza de date toate nodurile pentru locațiile la care se ajunge cu mașina. Ca parametri ai funcției apelată din productService, saveProduct, am introdus nodul provenit de la post requestul trimis de flutter si un String asociat cu numele colecției din care face parte nodul în baza de date. În aceeași manieră am salvat toate nodurile.

@RestController

@RequestMapping("/api")

public class ProductController {

@Autowired

ProductService productService;

@Autowired

Node n;

@RequestMapping("/aStar")

public List<Integer> getPath() throws InterruptedException, ExecutionException {

return n.alg();

}

@PostMapping("/carNodes")

public String saveCarNodes(@RequestBody Product product) throws InterruptedException, ExecutionException {

return productService.saveProduct(product,"carNodes" );

}

Funcția deleteProduct este accesata printr-un delete request în flutter înainte de introducerea în baza de date a altor date specifice unei origini și destinații diferite alese de utilizator.

@DeleteMapping("/deleteProducts")

public String deleteAllProduct() throws InterruptedException, ExecutionException {

return productService.deleteProducts();

}