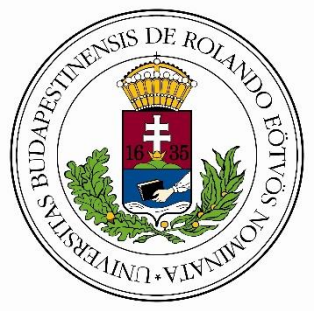
EÖTVÖS LÓRÁND TUDOMÁNYEGYETEM INFORMATIKAI KAR
TÉRKÉPTUDOMÁNYI ÉS GEOINFORMATIKAI TANSZÉK

Térképes szabad asztal kereső

DIPLOMAMUNKA

KÉSZÍTETTE:
Veress Orsolya
térképész mesterszakos hallgató

TÉMAVEZETŐ:
Dr. Gede Mátyás
adjunktus
ELTE Térképtudományi és Geoinformatikai Tanszék



Budapest, 2018

Tartalomjegyzék

1. Bevezetés

Diplomamunkaként egy olyan dologgal szerettem volna foglalkozni, ami valós problémákat old meg, de ugyanakkor engem is foglalkoztat, és kapcsolódik a tanultakhoz. Ezért döntöttem úgy, hogy egy webalkalmazást készítek, ami az emberek hasznára válhat, és ami egyszerűbbé teheti az életüket.

A mindennapokban folyamatosan keressük a lehetőséget arra, hogy az elvégzendő dolgainkat gyorsabban, könnyebben, egyszerűbben tudjuk megoldani. Nagyon sok ilyen lehetőség van, kezdve az online vásárláson keresztül egészen az online szórakozásig, és ha már szórakozás, akkor mindig egy sarkalatos kérdés, hogy hol is szórakozzunk. Ez az alkalmazás tematikus szűrők, információk által segít megtalálni a megfelelő helyet. Sok ehhez hasonló alkalmazás van, bármelyik térképes applikáció szóba jöhet, amely segítségével megtalálhatjuk a legközelebb lévő borozót, a még nyitva tartott éttermet, adatokhoz juthatunk egy általunk kiválasztott kávézóval kapcsolatosan. Ilyenek például az OpenStreetMap és a Google Maps szolgáltatásai.

Ellenben, ami különlegessé teszi ezt az alkalmazást, az a valós idejű információ, egész pontosan a valós időben jelzett telítettsége egy adott helynek. Ez azért fontos, mert az ember adott pillanatban tudhatja, hogy hol, és hogy mennyi szabad hely található, így ennek függvényében választhat, és nem találja magát szemben azzal a kellemetlenséggel, hogy az adott hely teljes mértékben foglalt, és hogy hirtelen új, alkalomhoz illő helyet kell keresnie, amivel ismételten előfordulhat ez a dolog. Ha egy csendes helyre vágyunk, ahol teázás közben nyugodtan olvasgathatunk, akkor könnyen megtalálhatjuk azt a teaházat, ahol csak néhányan vannak, és élvezhetjük a magányt. Egy másik nézőpontból megközelítve, annak is hasznos ez az alkalmazás, aki nagy tömegben szeretne szórakozni, bulizni, hiszen ebben az esetben a lényeg a minél telítettebb bárokon, szórakozóhelyeken van. Annak érdekében, hogy egy konkrétabb képet kapjunk a következőkben több szempont alapján próbáljuk meghatározni a honlapot.

Kinek szól?

A célcsoport főként a fiatalabb korosztály, a tinédzserek és az egyetemisták, fiatal felnőttek, mert ők a legaktívabbak ezen a területen. Ennek ellenére ezen a korcsoporton kívül álló emberek számára is hasznos, hiszen ezzel az alkalmazással bárki megtalálhatja a számára legmegfelelőbb,

a saját elvárásaihoz, hangulatához legjobban illő helyet, kortól függetlenül, legyen szó egy csendes vacsoráról, visszafogott borozásról vagy egy hatalmas buliról.

Miért hasznos?

Olyan információkat nyújt, amelyek így együtt nem találhatóak meg egy helyen, könnyen nem elérhetőek vagy nem teljesek. Egy település lakosságának jó része használja valamilyen formában a szórakozóhelyeket, ami nagyon sok embert jelent. Ezen emberek jó része keres és használ olyan technológiákat, újításokat, szolgáltatásokat, amelyek megkönnyítik az életüket, ami által kényelmesebben, könnyebben, gyorsabban érhetik el céljaikat, ezért is lenne sikeres az oldal.

2. Technikai rész

Az alábbi néhány alfejezetben röviden bemutatom a használt háttértechnológiákat, környezeteket, architektúrákat, amelyekben dolgoztam, amelyeket alkalmaztam a projekt elkészítése során. Mivel a munka végeredménye egy webalkalmazás, először ennek a fogalomnak a definíciójával és leírásával kezdem a bemutatást.

Webalkalmazás

A webes alkalmazás egy kliens-szerver közötti kommunikációt megvalósító alkalmazás, egy speciális program, amelyet egy webszerver futtat. A kliens a böngészőn keresztül kérést intéz a webszerver felé, ezt a kérést a szerver továbbítja a program felé, majd a program választ generál, amit visszaküld a webszervernek, ő pedig tovább a böngészőnek, ami megjeleníti.

Webszerver

A webszerver egy program, ami azokat a fájlokat tartalmazza, amelyekkel megjeleníthető egy weboldal. Ezeket a fájlokat a böngésző (Chrome, Safari, Firefox, Internet Explorer) a kliens számára értelmezhető képekké és szöveggé alakítja. A böngésző a webszerverrel kommunikál, hogy adatokat kérjen az internetről. A webszerverek több böngészővel kommunikálnak, ugyanazokat a fájlokat több felhasználónak küldik el ugyanabban az időben. A céljuk, hogy a kliens által kért fájlokkal térjenek vissza (Shklar és Rosen, 2003).

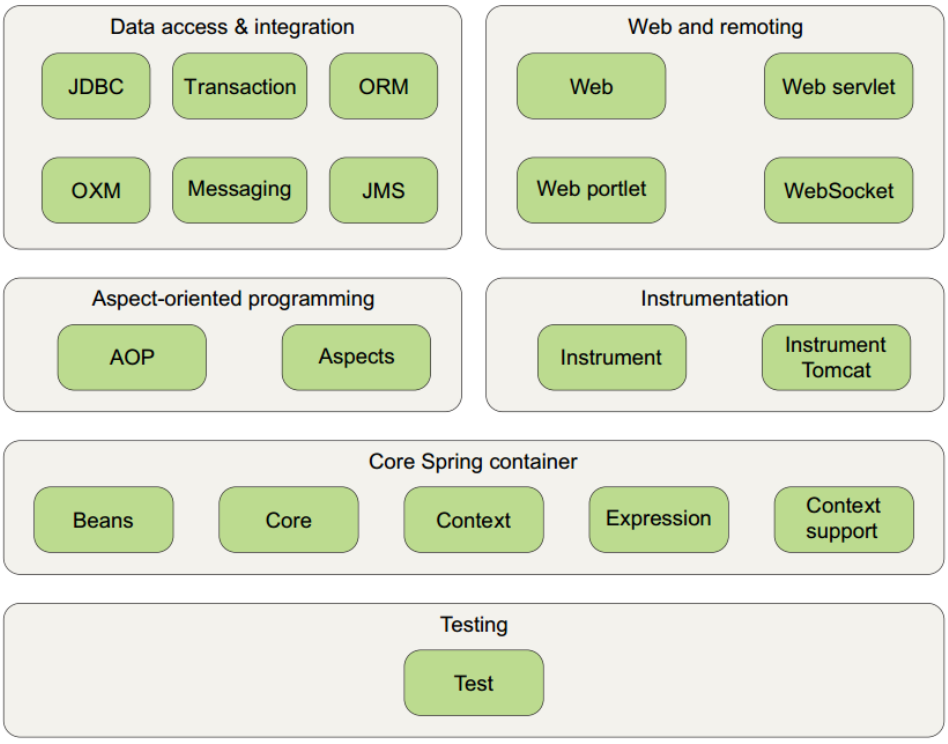
Példa erre az Apache Tomcat, ami egy nyílt forráskódú, Java nyelven készült webszerver. Első verzióját a Sun Microsystems-nél dolgozó James Duncan Davidson készítette, a Java Servlet (Java objektum, amely HTTP kéréseket dolgoz fel és HTTP választ generál) specifikáció referenciaimplementációjaként. Webes szolgáltatásokat nyújt az általunk készített alkalmazáshoz, miután kitelepítettük azt.

Spring Boot

Ahhoz, hogy az egész projektet egy nagy egészként tudjam kezelni, szükségem volt egy keretrendszerre. Ebben elhelyezve a programomat, lehetőségem nyílt különböző szolgáltatásokat igénybe venni, amelyek megkönnyítik és felgyorsítják számomra a fejlesztést.

Elsőként a Spring-et mutatom be, majd tovább haladva beszélek a Spring Bootról is, hiszen az alkalmazásomat ennek a keretrendszernek a használatával valósítottam meg.

A Spring egy nyílt forráskódú keretrendszer, amelyet Rod Johnson alkotott és írt le először az *Expert One-on-One: J2EE Design and Development* (Wrox, 2002) című könyvében. Általános megoldásokat kínál a fejlesztés során felmerülő problémákra, a gyors és hatékony alkalmazásfejlesztésre, illetve rétegelt alkalmazások készítésére. A Spring rétegelt kialakítású, ami azt jelenti, hogy a különböző típusú feladatok elvégzésére különböző modulok vannak. Ezek a modulok hat kategóriába sorolhatók (1. ábra). Ilyen például az adatbáziskezelés, tranzakciókezelés, webes szolgáltatások, tesztelési megoldások, aspektus orientált programozási lehetőségek, amelyek mind külön modulokban találhatók (Walls, 2015).



1. ábra: Spring Boot modulok csoportosítása

A modulok kategorizálása

A modularitás jelentősége abban nyilvánul meg, hogy nem kell az alkalmazásunkat teljes mértékben a Spring keretrendszertől függővé tenni, nem kell hozzákötnünk magunkat. Szabadon választhatunk a nekünk szükséges modulok közül, csak azt kell használnunk, ami konkrétan kell belőle, illetve, ha valamelyik modul nem megfelelő a számunkra, akkor egyszerűen egy másik keretrendszer könyvtárát használjuk a Spring által kínált megoldás helyett.

Igaz, hogy a Spring megkönnyíti a fejlesztés folyamatát a modulok megalkotásával, de a szoftver különböző beállításait, mint például a Maven függőségek hozzáadását és az XML fájlok konfigurálását a fejlesztőnek kell megcsinálnia. Erre a problémára kínál megoldást a Spring Boot.

Ez a keretrendszer a Spring család egy tagja, amely olyan plusz funkciókkal rendelkezik a Springhez képest, amelyekkel még hatékonyabbá és könnyebé teszi a fejlesztést. Célja, hogy a szoftverfejlesztőknek ne a program konfigurálással kelljen foglalkozniuk, hanem tudjanak a megírandó kódra koncentrálni, és hogy azt az időt, amit a beállítások szerkesztésével töltenének, azt az üzleti logika kigondolására és implementálására fordíthassák (Walls, 2016).

A fent említett Maven, teljes nevén Apache Maven, egy olyan szoftver, amelyet Jason van Zyl készített 2002-ben. Ezt az alkalmazást szoftverprojektek menedzselésére és a *build* folyamat automatizálására lehet használni. Alapja a POM (Project Object Model), azaz a Projekt Objektummodell, ami egy XML fájl, és leírja, hogyan kell az adott programot megépíteni, illetve tartalmazza a kódban használt függőségeket. A *build* folyamat alatt értjük a források lefordítását, a különböző állományok megfelelő helyekre történő másolását, és az alkalmazásunk összecsomagolását. A Maven *build* folyamat kimenetele az *artifact*, azaz a termék. Ezek a termékek lehetnek JAR állományok, webes alkalmazás esetén WAR állományok, nagyvállalati alkalmazások esetén EAR állományok. Ezek az állományok tulajdonképpen összecsomagolt fájlokat tartalmaznak. WAR, EAR és JAR kiterjesztésüket átírva ZIP-re megnézhetjük a bennük található fájlokat.

MVC (MNV)

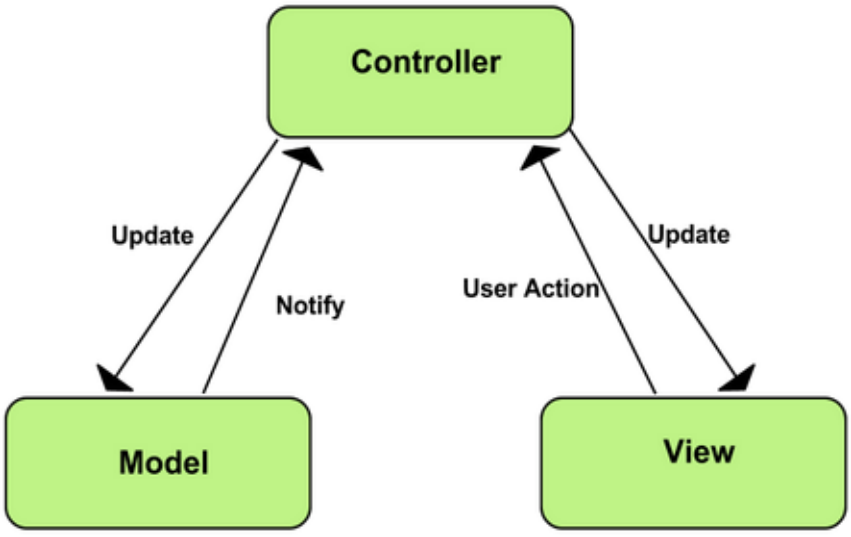
Az alkalmazás elkészítésekor az MVC mintát követve hoztam létre az osztályokat és a köztük lévő kapcsolatokat.

Az MVC betűszó a Model-View-Controller tervezési minta rövidítése, magyar megfelelője az MNV, Modell-Nézet-Vezérlő. Célja, hogy a felhasználói felület megjelenítéséért felelős kódot szétválassza az alkalmazás üzleti logikáját megvalósító kódjától (2. ábra). Ebben a struktúrában, ha módosítjuk, vagy kiegészítjük a felhasználói felületet, akkor nem kell megváltoztatnunk az implementált logikát.

A vezérlő az üzleti logikát képviseli, ő a kapcsolat a modell és a nézet között. Ha a felhasználói felületen módosítás történik, akkor a vezérlő feladata eljuttatni az információt a modellhez.

A modell a felhasználói felületen megjelenítendő adatokat, objektumokat reprezentálja.

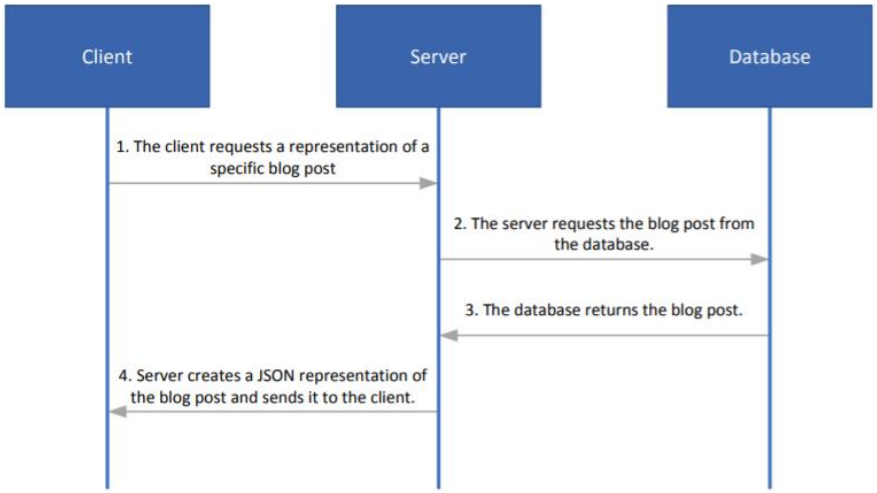
A nézet feladata a felhasználói felület megjelenítése, vagyis a modellben található adatok ábrázolása. A felhasználóval történő összes interakció a nézeten keresztül történik.



2. ábra: Modell-Nézet-Vezérlő tervezési minta

REST Web Szolgáltatás

A REST (REpresentational State Transfer), Reprezentációs Állapot Átvitel, egy architekturális stílus, egy tervezési szempont web-szolgáltatások készítéséhez. A kifejezést Roy Fielding vezette be és definiálta 2000-ben a doktori disszertációjában. Egy szerver-kliens kapcsolatot jelent, amelyben a szervernek vannak különböző erőforrásai, és a kliens kérheti ezeknek a megjelenítését valamilyen módon.



3. ábra: Rétegelt architektúra

A fenti példából kiindulva (3. ábra) láthatjuk, hogy a kliens nem tud arról, hogy a szerver hogyan tárolja az információt, számára az a fontos, hogy érthető módon megjelenjen a kért adat. A JSON (JavaScript Object Notation) a legtöbbször használt megjelenítési forma REST-es alkalmazások esetében, név-érték párokat tartalmaz (4. ábra).

```
{
  "title": "7 Things You Didn't Know about Star Wars",
  "content": "George Lucas accidentally revealed that..."
}
```

4. ábra: Egy JSON objektum tartalma, kinézete

Amikor a kliens megkapja a megjelenített választ, módosíthatja, törölheti, majd visszaküldheti a szerverhez ugyanabban a megjelenítési formában. A szerver ezt követően frissíti a saját forrásfájlját azzal az információval, amit a klienstől kapott.

Ahhoz, hogy egy alkalmazás REST-es legyen, eleget kell tennie néhány követelménynek:

1. Kliens – szerver architektúra: a rendszernek kliensekből és szerverekből kell állnia, amelyek jól elkülöníthetők egymástól. A szervereknek erőforrásaik vannak, amiket a kliens használni szeretne.
2. Állapot nélküli kommunikáció: a kéréseknek minden a kérés feldolgozásához szükséges információt tartalmazniuk kell, a szerveren nem kerül tárolásra semmilyen többletinformáció.
3. Cache: a szerver oldalról érkező válaszok megjelölhetők, hogy az adatok gyorsítótárazhatók vagy nem. Ha a válasz gyorsítótárazható, akkor a kliens gyorsítótára megadja a lehetőséget ezeknek az adatoknak a tárolására, hogy a későbbiekben azonos kérés esetén ez az adat újra felhasználható legyen.
4. Uniform interfész: az a legfőbb jellemvonás, ami megkülönbözteti a REST architektúra stílust más hálózati architektúra stílusoktól, hogy nagy hangsúlyt helyez a komponensek közötti egységes interfészre. Ennek a segítségével a teljes architektúra leegyszerűsödik, az implementációk szétválnak a szolgáltatásoktól.
5. Rétegelt rendszer: a kliensnek nem kell tudnia, hogy a kommunikáció közvetlenül a célszerverrel történik-e vagy egy a kliens és a szerver között elhelyezkedő közvetítő szerverrel. Ugyanakkor ez azt is jelenti, hogy a szerver több más szerverrel is kommunikálhat, hogy választ generáljon a kliens felé.

Annak függvényében, hogy egy alkalmazás mennyire „REST-es”, vannak fokozatok. Ha egy alkalmazás az összes egymásra épülő feltételt teljesíti, akkor az alkalmazás teljes mértékben „RESTful”-nak nevezhető. Dr. Leonard Richardson állította fel azt a négy szintes modellt (maturity modell), amelyen lentről felfele haladva, minden lépést egymás után teljesítve megalkothatjuk a tökéletes „RESTful” rendszerünket (Lange, 2016).

3. Adatforrások

Az alkalmazás elkészítéséhez szükségem volt azokra az adatokra, amelyek felhasználásával a grafikus felületen megjeleníthettem az információkat. Az adatok az alkalmazás felépítésének megfelelően két kategóriába sorolhatók. Az egyik adatforrást kizárólag a front-end használja, vagyis a felhasználói felület, míg a második a back-endhez, az üzleti logikát tartalmazó réteghez érkezik, ahonnan majd eljut a klienshez.

Az az adat, amit a front-end használ, a Mapzen által fenntartott, Metro Extract nevet viselő weboldalról származik, amely az OpenStreetMap ingyenesen hozzáférhető publikus adatait teszi elérhetővé, hetente frissítve őket. Ezek az adatok kategorizálva vannak, országokra lebontva, néhol ezen belül városokra is. Egy városon belül ismét osztályozva vannak, például az objektumok geometriája alapján letölthetjük csak a pontszerű, csak a vonalas, vagy csak poligon típusú objektumokra vonatkozó információkat. Az OSM-ről származó nyers adatok is letölthetők PBF vagy XML formátumban. Az adatok különböző formátumú fájlokként tölthetők le (SHAPEFILE, GeoJSON, PBF, XML), annak függvényében, hogy éppen mire van szükségünk.

Az alkalmazásom felhasználói felületén a szórakozóhelyeket jelenítem meg, ezért a pont típusú objektumokra volt szükségem SHAPEFILE formátumban. Ezt követően a QGIS használatával elkezdtem a fájlok feldolgozását.

A QGIS egy nyílt forráskódú térinformatikai szoftver, melynek első verzióját 2004-ben adták ki. A programmal különféle formátumban tárolt, különféle térképi vetületben ábrázolt, vektoros vagy raszteres fájlokat tudunk összeállítani egy egységes térképpé. Az általa támogatott fájlformátumok sokfélék, ide tartozik többek között az ESRI SHAPEFILE, a Mapinfo, GeoJSON, GML (GoogleEarth XML). Ezek mellett támogatja az összes raszteres formátumot, digitális terepmodelleket, légifényképeket és Landsat műholdfelvételeket (Szabó, 2010).

Először kiválogattam az adatokat, aztán külön SHAPE fájlokat hoztam létre a nekem szükséges kategóriáknak megfelelően: kávézók, éttermek, gyorséttermek, bárok. Az így keletkezett állományok adattábláinak objektumaihoz hozzárendeltem a koordinátákat a *Vector/Geometry Tools/Export/Add geometry columns* funkcióval, majd GeoJSON fájlokká alakítottam őket.

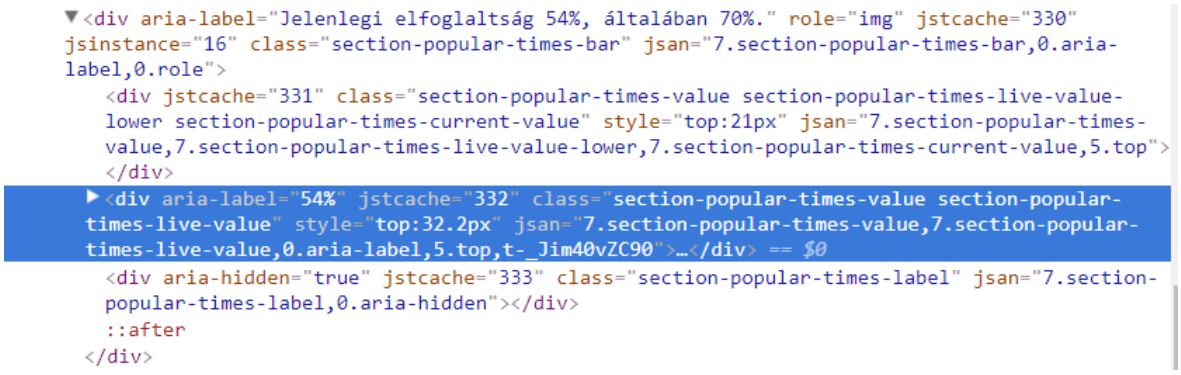
A back-end számára az adatokat a Google Places API szolgáltatja, ezen belül a Web Service. Ez egy olyan szolgáltatás, ami információkkal kínál intézményekkel, létesítményekkel, földrajzi pontokkal, kiemelkedő látnivalókkal kapcsolatosan HTTP kéréseken keresztül. Ezek a helyadatok, amelyek tartalmazzák például a hely azonosítóját, nevét, címét és más attribútumokat. Az információk megszerzéséhez először egy Google Places API kulcsot kellet szereznem, amelyhez elég volt egy egyszerű regisztráció. Ezt a kulcsot a lekérdezésekbe kell beilleszteni, mert enélkül nem elérhetők az adatok (példa URL + mellékletben a kapott válasz). A lekérdezés eredménye egy JSON fájl, ami könnyen kezelhető, és a továbbiakban gyorsan feldolgozható.

A Google Map minden helyének adatai között megjelenik egy grafikon *Népszerű időszakok* elnevezés alatt (5. ábra). A grafikon a hely átlag telítettségét mutatja órákra lebontva. Az értékek nem láthatók, az oszlopok magasságából olvashatjuk ki megközelítőleg, hogy milyen mértékben van tele az adott hely. Ezeket az adatokat százalékos formában szerzi meg a Python program.

5. ábra: A "Népszerű időszakok" grafikonja

Az alkalmazás lényege a valós adatokra épül. Ezek az adatok is a Google Map-ről származnak, de nem a Google Places API segítségével szereztem meg, hanem a Google Map használatával megnyitott hely DOM-jából bányászom ki (6. ábra). A DOM (Document Object Model), Dokumentum Objektum Modell, weboldalakat kapcsol össze script és egyéb programozási nyelvekkel. Az oldalak szerkezetéért felelős HTML elemeket JavaScript objektumként reprezentálja, így a dokumentum bármelyik apró részlete bejárható és manipulálható. A DOM a dokumentumot egy logikai faszerkezettel ábrázolja. A fa minden ága egy csomóponttal végződik, és mindegyik csomópont egy objektumot tartalmaz. Metódusait használva hozzáférhetünk a fa bármely eleméhez, megváltoztathatjuk a fa struktúráját, stílusát vagy tartalmát. Összességében a

weboldal egy objektumorientált megvalósítása, amelyet módosítható script nyelvekkel, általában JavaScript-tel.



6. ábra: A DOM-ban található valós idejű adat helye

4. A rendszer komponensei

Front-end

Ahogy az előző fejezetben említettem, az alkalmazásnak két része van: front-end és back-end.

A front-end a felhasználói felületet alkotó weboldalakból áll, feladata a rendszerből kijutó adatok prezentálása, illetve a bejövő adatok fogadása a felhasználó felől. Célja, hogy amikor a felhasználó megnyit egy weboldalt, akkor az információt könnyen olvasható és értelmezhető formában jelenítse meg.

Webes technológiák

Webes alkalmazások készítésekor különböző webes technológiákat használnak, mint például Hypertext Transfer Protocol (HTTP), Uniform Resource Locator (URL), Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript Programming Language, JavaScript Object Notation (JSON), Web API.

A HTML, vagyis a hiperszöveges jelölőnyelv egy leíró nyelv, egy internetes szabvány. A hiperszöveg jelenti az interneten található dokumentumokat, amelyek formázott vagy formázatlan szöveget, képet, videót, hanganyagot, vagy ezek tetszőleges kombinációját tartalmazzák. A HTML ezeknek a dokumentumoknak az elrendezését, formázását tartalmazza.

Az alkalmazásom *map.html* dokumentuma tartalmazza a weboldal külalakját alkotó elemeket. Elsőként magába foglalja a térkép megjelenítésére használt tároló ( ) elemet. Ezt követően a kategóriák felsorolásához egy új, lista (<ul>) típusú elemet hozok létre, amely magába foglalja a helyek adatainak megjelenítésére szolgáló tárolót. Ez utóbbi két elemet tartalmaz, az egyik egy egyszerű bekezdés (<p>) típusú elem, míg a másik egy táblázat (<table>), amely a kiválasztott hely nevét, címét, értékelését, az értékelők számát, illetve a telítettségi értékét jeleníti meg rendezett, szabályos, strukturált formában. A felsorolt elemek mellett tartalmaz hivatkozásokat több használt stíluslapra és a műveleteket végző JavaScript fájlra is. Az elemekhez tartozó stílusbeállításokat a *style.css* fájl tartalmazza.

A CSS, azaz az egymásba ágyazott stíluslapok írják le, hogy hogyan kell a HTML dokumentumnak megjelennie. Itt adható meg például a használt betűstílus, a betű és a háttér színe, a különböző elemek elhelyezkedése. A tervezése során az volt a legfontosabb szempont, hogy a dokumentum struktúráját elkülönítsék a megjelenítéstől.

Az alkalmazás által használt CSS fájl a *style.css*. A *map.html* dokumentum minden elemének stílusa, az elemek kinézetére vonatkozó beállítások mind ebben az állományban vannak leírva. A különböző célokra használt komponensek különböző tulajdonságokkal rendelkeznek. A *data* elem például a helyadatok megjelenítéséért felel, és a beállításai között olyanok szerepelnek, mint az elem elhelyezése, színe, szélessége, hosszúsága, a benne található szöveges elemekre vonatkozóan pedig megadásra került a betű méret, betűcsalád, betű vastagság, betű szín, szöveg behúzás.

A JavaScript egy magas szintű, dinamikus és interpretált programozási nyelv. A HTML és a CSS mellett a JavaScript az egyik leglényegesebb technológia webes alkalmazások fejlesztésénél. A legtöbb weboldal használja és minden modern böngésző támogatja.

A *load.js* felel a weboldal viselkedéséért, hogy mi és hogyan jelenjen meg. Ebben a fájlban az OpenLayers függvénykönyvtár használatával végzem el a térkép megjelenítéséhez szükséges beállításokat, illetve itt történik meg a kommunikáció a front-end és a back-end között. Az OpenLayers-ről szóló leírásnál bővebben ismertetem a kódot.

Egyéb technológiák

MapBox

A MapBox egy nyílt forráskódú térképszolgáltató, amely mobil- és webalkalmazások készítéséhez nyújt egyedi térképeket. A személyre szabható térképeket bárki könnyen elkészítheti, miután regisztrált a MapBox (www.mapbox.com) honlapján. Az alkalmazásomban használt térképhez a Cartogram funkciót (7. ábra) használtam. Ez a szolgáltatás lehetővé teszi a térkép színének a beállítását. Megadhatjuk a víz, a beépített terület, a kiemelt épületek, az utak és a névrajz színét egy speciális módon. Annyi a feladat, hogy kiválasztunk egy képet, amely azokat a színeket, színárnyalatokat tartalmazza, amelyeket használni szeretnénk a térképünkön. Ezt követően feltöltjük a képet a megadott helyen, és a különböző kategóriáknak megfelelő köröket arra a színre

húzzuk, amilyennel szeretnénk ábrázolni. A *Save* gombra kattintva elmenthetjük a saját stílussal ellátott térképünket, és végeredményként egy CSS fájlt kapunk, amelyre hivatkozhatunk a HTML dokumentumban.

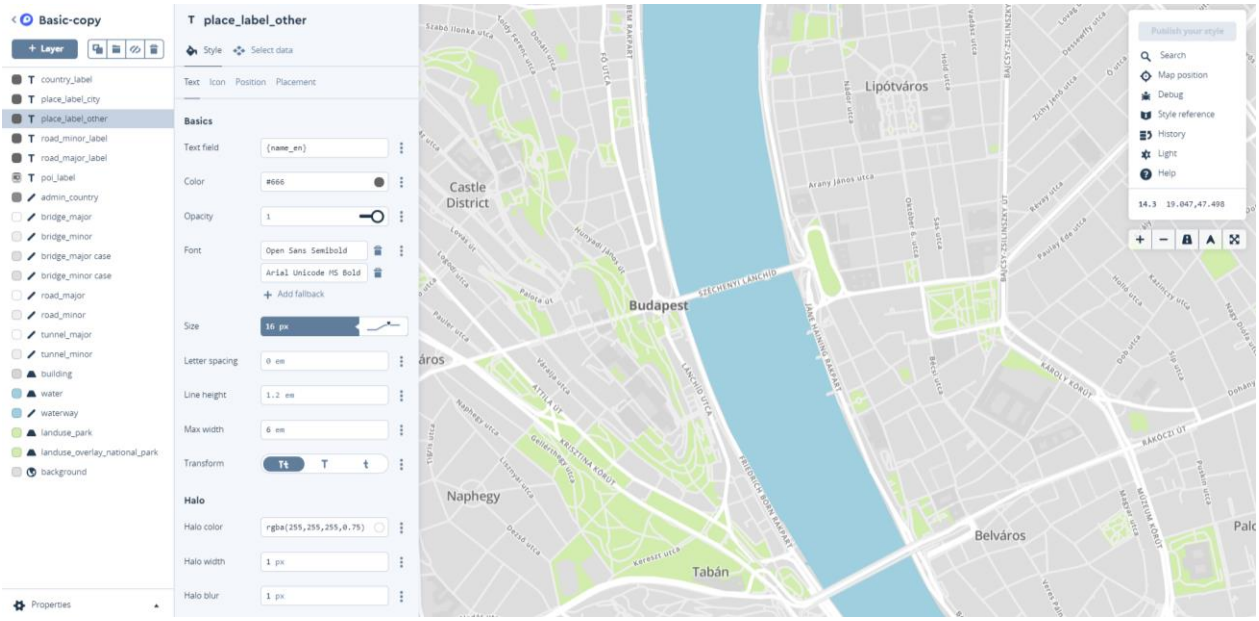
7. ábra: Térkép stílus létrehozása hozzáadott képet felhasználva

A *Products/Studio* menüpont alatt további igazításokat hajthatunk végre a térképen. Kiválasztjuk az előzőleg elmentett stílust, majd a bal oldalon megjelenő sávban található térképi elemek közül kiválasztjuk azt, aminek a színén, vonalvastagságán vagy betűstílusán változtatni szeretnénk. A térképi elemeket felsoroló sáv mellett megjelenik egy újabb sáv, amiben már a kiválasztott elem tulajdonságait látjuk, amelyeket kedvünk szerint állíthatunk.

OpenLayers

Az OpenLayers egy nyílt forráskódú kliens oldali JavaScript függvénykönyvtár, melynek segítségével különböző forrásokból származó térképi adatokat jeleníthetünk meg a legtöbb modern böngészőben térkép formájában, szerver oldali függőségek nélkül. A JavaScript API-t implementálja önmagában, hogy ezáltal web alapú földrajzi applikációkat, térképi adatok felhasználásával interaktív térképes weboldalakat készíthessünk. Első kiadását a MetaCarta nevű cég fejlesztette ki 2006-ban (Gede, 2014). Az OpenLayers csak az adatok megjelenítését végzi, ezért mindenképp szükség van a háttérben valamilyen adatforrásra: egy egyszerű raszteres képtől

kezdve az OpenStreetMap térképen és a különféle WMS, TMS szervereken keresztül különböző vektoros adatforrásokig rengeteg lehetőség akad.



8. ábra: A létrehozott stílus további beállításai

A kliens oldali technológia, másnéven vastag kliens, azt jelenti, hogy a felhasználó böngészője végzi el a feladatok nagy részét, képes nagy adatmennyiségek feldolgozására. Ennek párja a vékony kliens, egy minimális eszközökkel rendelkező kliens, feladata nagyrészt az alkalmazásszervertől érkező adatok grafikus megjelenítése.

Most, hogy röviden ismertettem az MapBoxot és az OpenLayerst, rátérek a JavaScript leírásánál említett *load.js* fájlra, ami az alkalmazáshoz tartozó weboldal működéséért felel, hiszen ebben a fájlban használom az OpenLayers függvényeit és objektumait, melyek segítségével könnyebben megjeleníthető és alakítható a térkép és a hozzá tartozó elemek. A térképre vonatkozóan, hogy mi és hogyan jelenjen meg, beállításra kerül a kezdeti zoom szintje, illetve a középpontja. A térkép megjelenítéséhez *basemap* néven létrehozok egy új csempét (Tile), amiben forrásként megadom az általam MaxBox-on készített térkép elérhetőségét, és az URL-be beleillesztem a regisztráció után kapott kulcsot is. Következő lépésként létrehozok *map* néven egy térkép (Map) objektumot, és ebben beállítom, hogy a *map.html* dokumentumból melyik tároló tartozik hozzá (target), és a megjelenítendő rétegekhez (layers) hozzáadom a korábban elkészített *basemap*-et. Ezzel megjelenítésre került a térkép.

Ezt követően beállítottam a *switchLayer* függvényben, hogy a rétegekre kattintva megjelenjenek a helyek egy-egy körként ábrázolva, és hogy egy időpontban egyszerre csak egy réteg legyen aktív. A kattintásra való megjelenítéshez egy *for* ciklust használva végig mentem a HTML dokumentum *category* listáján, és mindegyik elemhez hozzárendeltem egy *click* eseményfigyelőt.

A következő lépés a kategóriákhoz tartozó rétegek előállítása. Mindegyik kategóriához külön vektor réteg tartozik, hiszen az adatforrás változik. A rétegekben meg kell adni a forrást (source), ami ebben az esetben egy klaszter lesz, és a stílust (style), vagyis az adatok megjelenítésének módját. A forrást a *getClusterSource* függvény hozza létre, amely bemeneti paraméterként kéri annak a JSON fájlnak az elérési útvonalát, amely a megjelenítendő adatokat tartalmazza. A stílust a *clusterStyle* függvény készíti el, és ez a stílus mind a négy kategória esetén (kávézók, éttermek, gyorséttermek, bárok) ugyanaz marad, hiszen a megjelenítés nem változik, a helyeket ábrázoló körök színe és klaszter beállítása ugyanaz marad. Ebben a függvényben meg kell adni egy pont stílust, ez az objektumok kinézetére vonatkozik, és egy szöveg stílust a klaszterben található elemek számának a megjelenítése miatt. A pont stílus esetében fontos megjegyezni, hogy a kör mérete a benne található objektumok számával egyenes arányban változik.

A csoportosított objektumok megjelenítése után létrehoztam egy új *select* változót, ami egy *ol.interaction.Select* objektum. Ez az objektum teszi lehetővé, hogy a vektor rétegeken kiválaszthatók legyenek az elemek, vagyis, hogy ha egy helyre rákattintunk, akkor bekövetkezhessen egy esemény. Ehhez annyit kellett tenni, hogy a *select* változó rétegek (layers) attribútumánál felsoroltam azokat a rétegeket, amelyek esetén szükségem volt ennek a kiválasztási műveletnek az alkalmazására. Egy elemre történő kattintás esetén a jobb oldali sávban megjelennek a hely adatai. Először definiálni kell a kiválasztott elemet, majd, mivel klaszterről beszélünk, meg kell vizsgálni, hogy egy elem van-e benne. Ha több van, akkor nem történik semmi, ha viszont egy elemű, akkor kiolvasom a hely nevét és koordinátáit. Az így megszerzett elemek továbbítását egy *XMLHttpRequest* (XHR) objektum felhasználásával végzem. Először az *open* metódusával inicializálom a kérést, *setRequestHeader* metódusában beállítom a JSON formátumot, majd a *send* metódussal kérésként továbbítom a szerver fele a fentiekben megszerzett adatokat. Az *onload* metódusban visszakapom a választ a szervertől JSON formátumban, így egyszerűen kiolvashatom belőle a szükséges értékeket. Ezeket az értékeket beillesztem a HTML dokumentumban található

táblázat megfelelő soraiba. Az *onload* függvényben szereplő *if* szerkezetre azért van szükség, mert a JSON objektumban három féle értéket kaphatunk vissza a szervertől. Abban az esetben, ha nincs információ az adott helyről, akkor az attribútum *undefined* lesz, ha valós idejű érték van, akkor *realTimeValue,* ha pedig csak a grafikonról leolvasott érték található, valós idejű adat nem, akkor *graphValue* lesz az attribútum megnevezése. Ennek függvényében változik, hogy hogyan lesz megjelenítve az attribútumokhoz tartozó érték: *realTimeValue* esetén vörös színnel lesz ábrázolva a konkrét érték, *graphValue* esetén fekete színnel, míg *undefined* esetén a *Nincs adat!* felírat lesz feltüntetve, illetve az adatok megjelenítésére szolgáló táblázat sem lesz megjelenítve.

Az elemre történő kattintásra megjelenő adatok mellett egy másik fontos funkciója az alkalmazásnak, hogy meghatározott zoom szint fölött a képernyőn megjelenő térkép részletben megváltozik azoknak a helyeknek a szimbóluma, ahol már csak egy elem van a klaszterben. Az előző esetben használt lépéseken megyek keresztül ebben a függvényben is. Annyiban különbözik, hogy más belépési feltételnek kell teljesülnie, illetve az *onload* metódusban más események történnek. A feltétel teljesítéséhez három kritériumnak kell megfelelnie az elemnek. Az első, ahogy az előző funkcióban láthattuk, itt is fontos, hogy a klaszter csak egy elemű lehet, hiszen kiválasztáskor egy konkrét hely adatait szeretnénk megtudni. A második feltétel, hogy az adott objektumnak a képernyőn látható térkép részleten belül kell lennie. A harmadik feltétel, hogy a zoom szint 18 fölött legyen, mert így jobban lehatárolható és átlátható a terület, ahol szabad helyű szórakozóhelyet keresünk, illetve a programnak nem kell az összes klaszteren végig mennie és minden elemet megvizsgálnia, mert ez a zoom szint után már sokkal kevesebb elem esik a képernyőre jutó térkép részre. A szimbólumok változása itt is attól függ, hogy milyen attribútum érkezik a JSON objektumban: *undefined, realTimeValue* vagy *graphValue.* Ha az attribútum *undefined*, akkor a kör színe szürkére vált, ha *realTimeValue,* akkor pirosra, ha *graphValue*, akkor zöldre. A piros és zöld színek esetében ezek árnyalatai is változnak a telítettség értékének függvényében. A különböző kör stílusoknak a beállításait a *getCircleStyleForSelectedLocation* függvény végzi, ami paraméterként várja a telítettségi értéket, a zéró telítettségi értéknek megfelelő szín RGB értékeit, ebben a sorrendben, illetve a száznak megfelelő szín RGB értékeit. Ezen értékek alapján, a megfelelő műveleteket végrehajtva rajtuk, előállítható a két véglet között található átmeneti rész bármely színe.

Back-end

A back-end egy rendszerben a tényleges feldolgozást végző réteg. Feladata a front-end felől érkező adatok feldolgozása, majd a keletkezett eredmény visszajuttatása a front-end számára. Ebben a rétegben történik meg az adatok megszerzése, feldolgozása, továbbítása, illetve válasz generálása egy adott kérésre. A készített alkalmazás back-end-je két részből áll. Van egy Python kód, ami a Google-ről kérdezi le a helyek adatait, illetve van egy Java rész, ami ezeket az adatokat formázza, rendezi és műveleteket hajt végre rajtuk.

Python

A helyadatok lekérdezésére egy, már létező könyvtárat használok fel. Az általam írt python forráskód a datagetter.py nevet viseli, amely a *populartimes* könyvtárat felhasználva, ennek segítségével, külön fájlokba írja ki az adatokat. Mivel a Google lekérdezésekhez API kulcs szükséges, amely korlátozott számú lekérdezést engedélyez, ezért Budapest egy részét téglalapokra osztottam fel, és mindegyik téglalapra külön hajtottam végre a lekérdezéseket, így minden téglalaphoz egy külön szöveges állomány tartozik, amelyek a *data* mappába kerülnek. A téglalapok határait, illetve a keletkező fájlok nevét a dataLocation.csv fájlban adtam meg, a téglalapokat a bal alsó, és a jobb felső sarokpontok koordinátái határozzák meg. A program az előbb említett fájlból sorra kiolvassa az adatokat, majd a téglalap elhelyezkedése alapján használja a *populartimes* könyvtárat, és erre a területre vonatkozóan megszerzi a megadott feltételnek megfelelő helyeket, vagyis azokat, amelyek besorolhatók a kávézó, bár, étterem vagy gyorsétterem kategóriák egyikébe.

Java

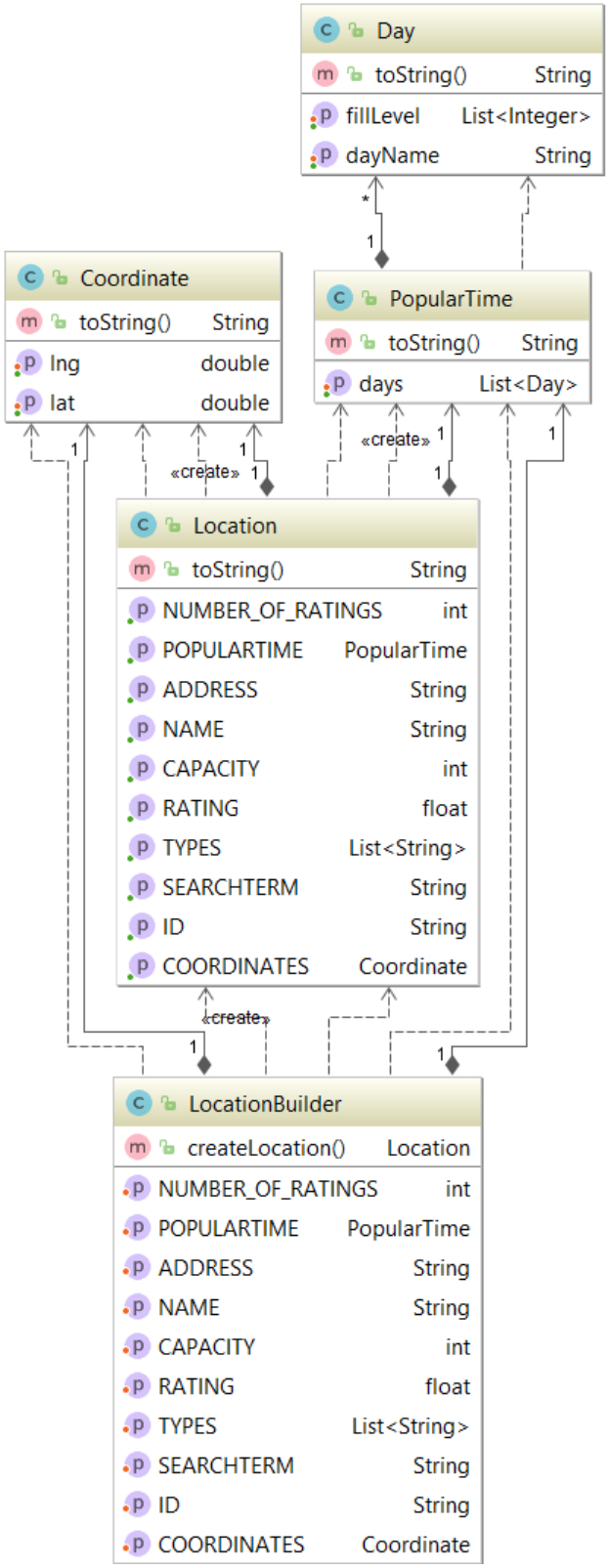
Az alkalmazásnak ez az a része, ahol a modell-nézet-vezérlő tervezési mintát alkalmaztam. Ennek megfelelően három különböző csomagot hoztam létre, amelyek az osztályokat tartalmazzák. A *model* csomagban a felhasználó felületén megjelenő objektumok találhatók. A csomag öt osztályt tartalmaz: *Coordinate, Day, Location, LocationBuilder* és *PopolarTime.* A *LocationBuilder* osztály az építő programtervezési mintát valósítja meg, amely a *Location* osztály létrehozását segíti. A *Location* osztály minden olyan adatot tartalmaz, amelyet egy helyről megtudhatunk. Az osztály négy *String*, azaz szöveg típusú változót tartalmaz: a hely azonosítója,

neve, címe, a keresési kifejezése, egy *float*, vagyis lebegőpontos értéket, amely a hely értékelése, két *int* típusú értéket: az értékelések száma, a hely maximális kapacitása, egy listát, amely *String* objektumokat tartalmaz, vagyis azokat a kategóriákat, ahova besorolható az adott hely. Ezek mellett tartalmaz egy *Coordinate* objektumot a koordináták tárolására, illetve a népszerű időszakok adatait magába foglaló *PopularTime* objektumot. A *Coodinate* osztály két értéket tartalmaz, a hosszúsági- és szélességi koordinátákat, a *PopularTime* osztály egy listát tartalmaz *Day* típusú objektumokkal. A *Day* osztály egy szöveg típusú változót tartalmaz, amely a hét egy napjának neve, illetve egy listát, amely a hely telítettségét mutatja százalékos formában minden órában.

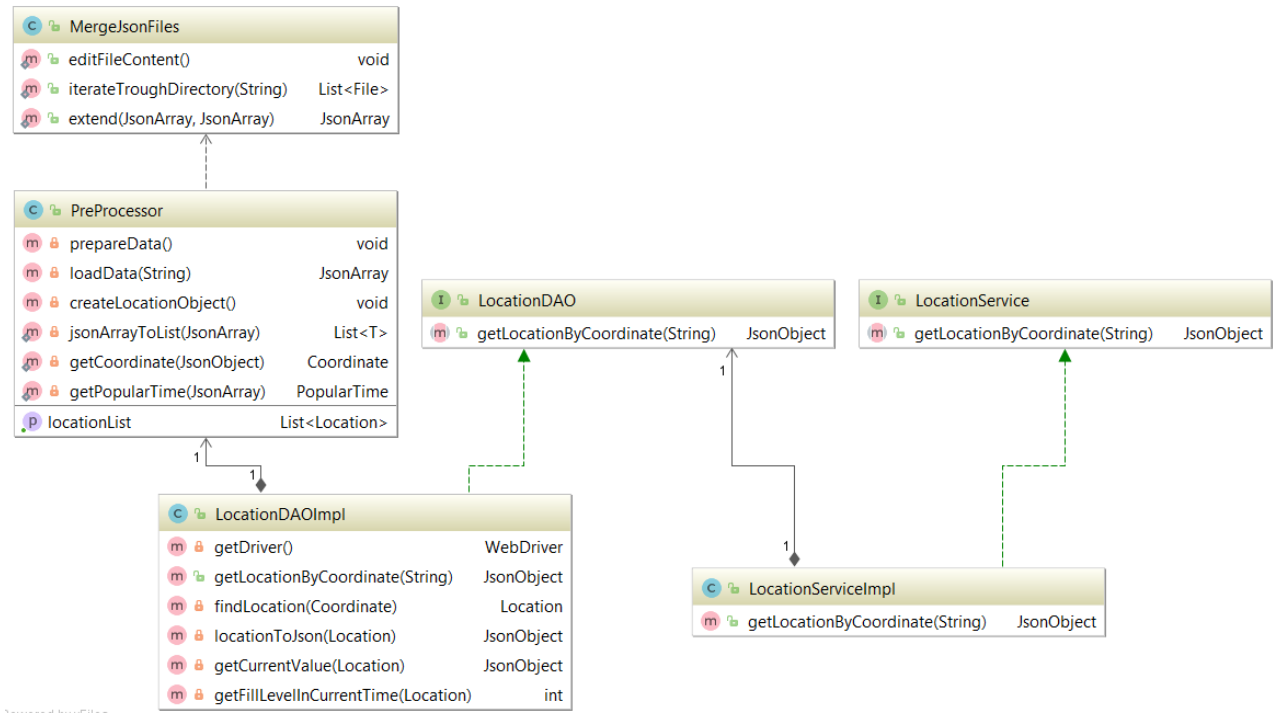
A vezérlő réteg osztályai a *service* csomagban találhatók. Ez a réteg képviseli az üzleti logikát, ő a kapcsolat a modell és a nézet között. Elsőként a *MergeJsonFiles* nevű osztályt mutatom be, ugyanis ő az első, aki módosítja a Python kód által generált adatokat, a szöveges dokumentumokat, annak érdekében, hogy a program további részei felhasználhassák. Az osztály *iterateThroughDirectory* függvénye végig iterál a *data* mappa tartalmán, és kiválasztja azokat a fájlokat, amelyek elnevezése a *part* szótaggal kezdődik, majd visszatér egy listával, amely a kiválogatott fájlokat tartalmazza. Következőként az *editFileContent* függvényben az előzőekben kapott lista minden elemét, vagyis a fájlok tartalmát egyenként szöveggé alakítjuk, aminek használatával egy *JsonObject*-et hozunk létre, és ezt az *extend* függvény segítségével hozzáadjuk az *allData* fájl tartalmához. Ez a JSON fájl egyesíti az összes fájl tartalmát egy nagy fájlban.

A *PreProcessor* osztályban a *prepareData* metódus hívja meg az előző osztály *editFileContent* metódusát, majd az *allData.json* fájlt felhasználva *Location* objektumokká alakítja az állományban található helyeket.

A *LocationDAO* egy adathozzáférési objektum (Data Access Object), amely egy interfész, és szerepe, hogy szigorúan elkülönítse a program két részét, azaz az adatbázis és az üzleti logika réteget. Ennek a két rétegnek nem kell tudnia egymásról, függetlenül fejleszthetőnek kell lenniük, az adatokkal kapcsolatos változások nem érintik a kliens számára elérhető funkciókat. Az interfész egy metódus fejlécet tartalmaz, a neve *getLocationByCoordinate*, amely egy *String* értéket vár paraméterül.



9. ábra: A „model” csomag osztálydiagrammja



10. ábra: A „service” csomag osztálydiagrammja

A *LocationDAOImpl* osztály implementálja a *LocationDAO* interfészt, vagyis kötelező módon meg kell írni a *getLocationByCoordinate* metódusát. Ez az osztály, ahogy a neve is mutatja, bemeneti paraméterként kap egy koordináta párt, és visszatér az azon a ponton található szórakozóhely adataival. A *getLocationByCoordinate* metódus a *String*-ként kapott értéket *Coordinate* objektummá alakítja, majd ez alapján a *findLocation* metódust felhasználva, a *PreProcessor* osztályból létrehozott *preProcessor* objektum *getLocationList* függvényének meghívásával kapott listát bejárja, és kiválasztja azt az objektumot, melynek koordinátái megegyeznek a beérkezett koordinátákkal. Ha nem talál a feltételnek megfelelő *Location* objektumot, akkor egy alap, üres objektummal tér vissza. A következő lépés a telítettségi adat megszerzése. Elsőként egy *ChromeDriver* objektum használatával megvizsgálom, hogy található-e valós idejű adat. A *ChromeDriver*-nek megadok egy URL-t, amely tartalmazza a hely azonosítóját, és ez alapján rákeres a helyre a Google térképen. Az így megnyitott weboldal DOM-jából a program kiolvassa a valós idejű értéket. Amennyiben nem található, a *Location* objektum *PopularTime* tulajdonságából olvassa ki az adott napnak és órának megfelelő átlagolt értéket. A metódus egy *JsonObject*-tel tér vissza, amely tartalmazza a *Location* objektumot és a telítettségi

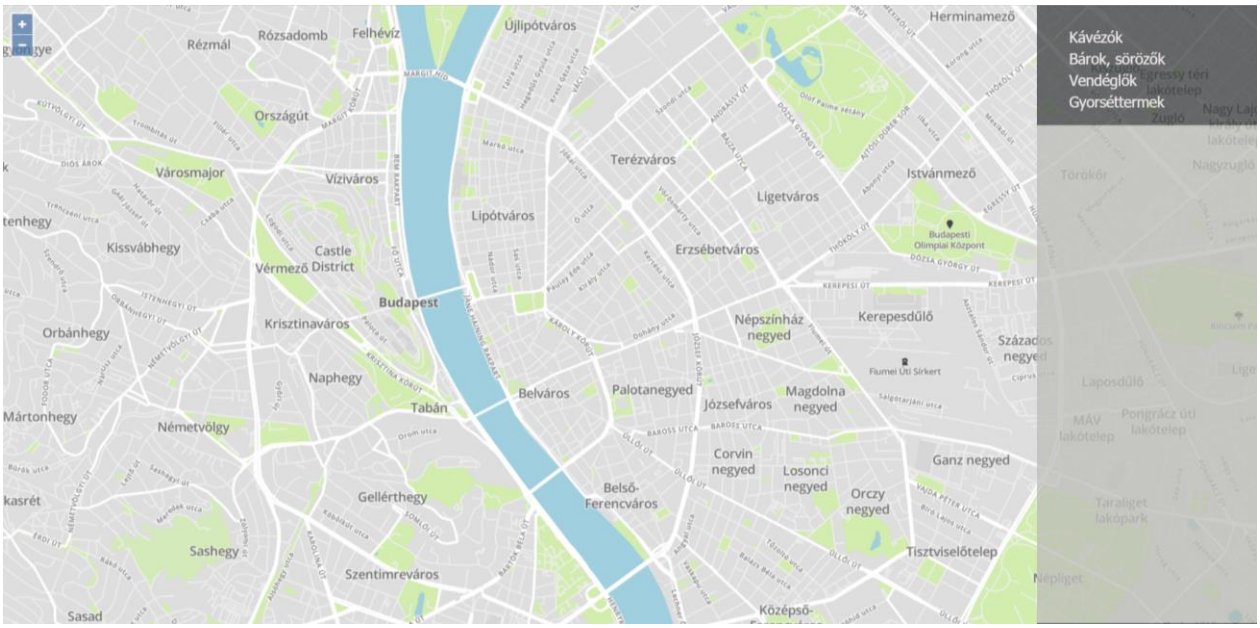
értéket. Attól függően, hogy a telítettségi érték valós, grafikonból kiolvasott, vagy egyáltalán nem található, három megnevezése lehet: *realTimeValue*, *graphValue* vagy *undefined*.

A *LocationService* interfész, akárcsak a *LocationDAO*, egy metódus fejet definiál. A *LocationServiceImpl* osztály az előbb említett interfészt valósítja meg önmagában úgy, hogy létrehoz egy objektumot, amely implementálja a *LocationDAO* interfészt, és azon keresztül meghívja az elérhetővé vált funkciókat, vagyis a *getLocationByCoordinate* metódust.

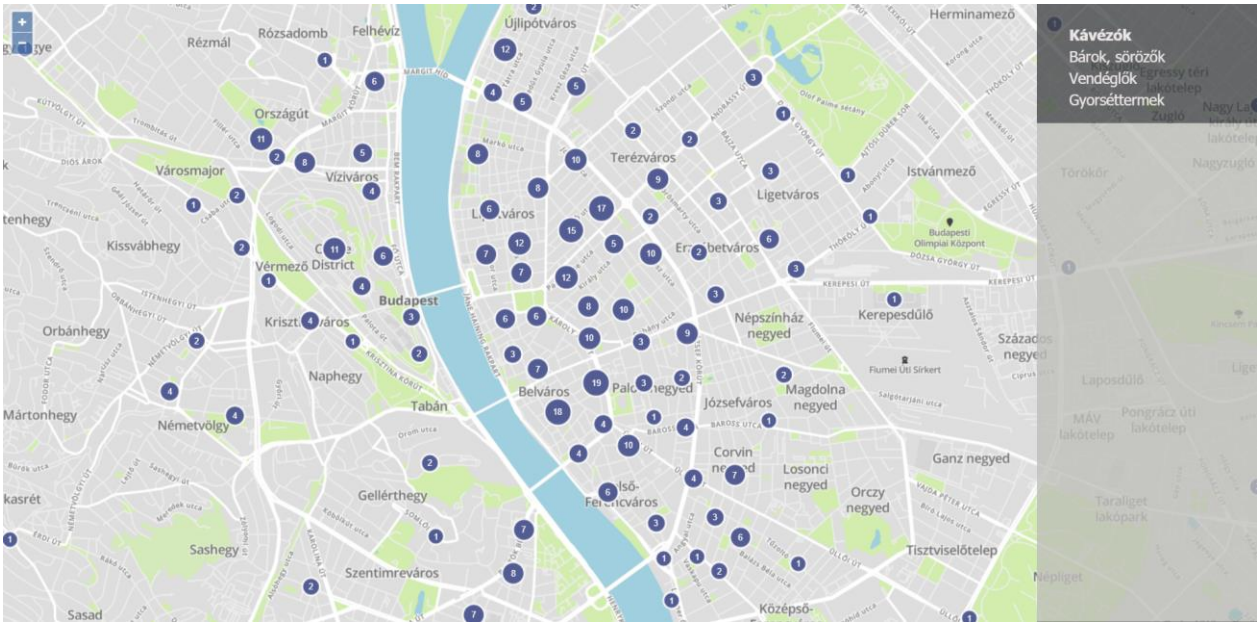
A web rétegben történnek a REST-es kérések. A „/map” elérési útvonalon megjelenik a *map.html*. Ugyanerre egy POST kérés is történik, vagyis a kliens oldalról kapunk információt. A *post* metódusban a *LocationService* interfészt implementáló osztály *getLocationByCoordinate* függvénye hívódik meg, amely egy *JsonObject*-tel tér vissza, és ezt a választ kapja meg a *load.js* JavaScript kód.

5. Az alkalmazás működése

Ha elindítjuk az alkalmazást, akkor a böngészőben egy térképes oldal jelenik meg, melynek jobb oldalán egy világosszürke sáv található, tetején egy sötét résszel. Itt vannak felsorolva a kategóriák, amelyek közül kiválaszthatunk egyet.



Ha egy kategória fölé visszük az egeret, akkor annak nevének betűstílusa megváltozik, jelölve, hogy melyik elem lesz kiválasztva. Rákattintva, különböző méretű kékes színű körök jelennek meg, bennük a klaszter által tartalmazott helyek számával.

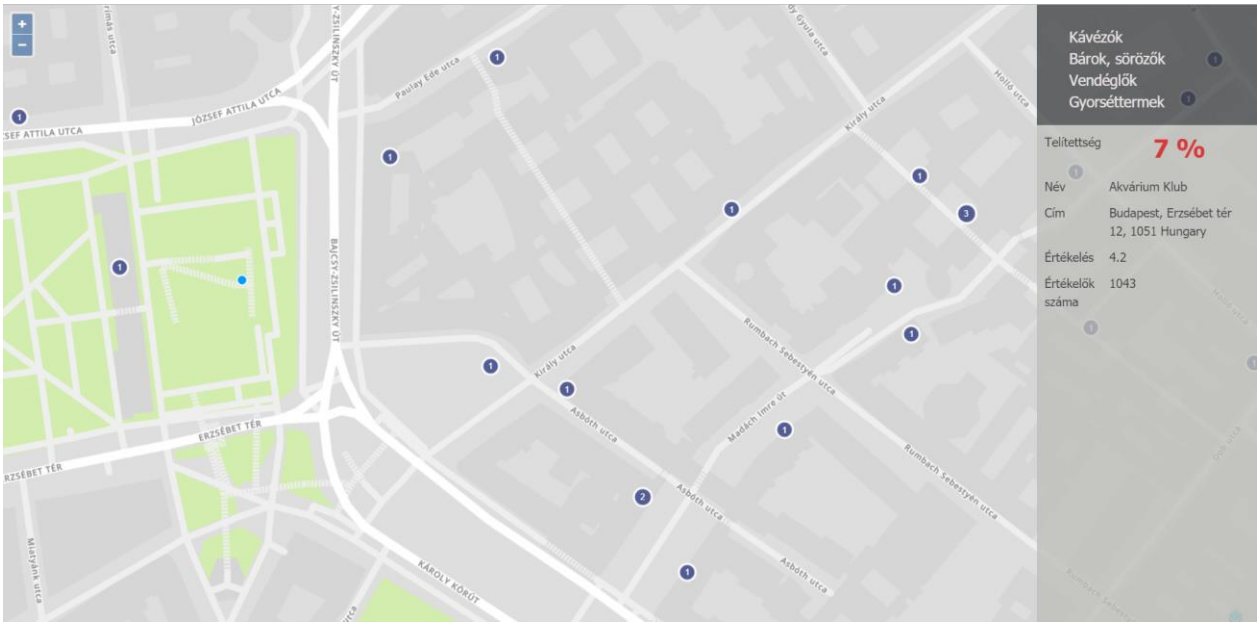


Ha belenagyítunk a térképbe, a klaszterek újabb, kisebb klaszterekre bomlanak mindaddig, amíg csak egy elemet tartalmaznak.

Azokra az elemekre, ahol már csak egy elem van, vagyis egyes jelenik meg a körben, rákattinthatunk, és információkat tudhatunk meg róla. Amennyiben az adott objektumról nem található információ a rendszerben, akkor a *Nincs adat!* felirat jelenik meg a jobb oldali világosszürke sávban.

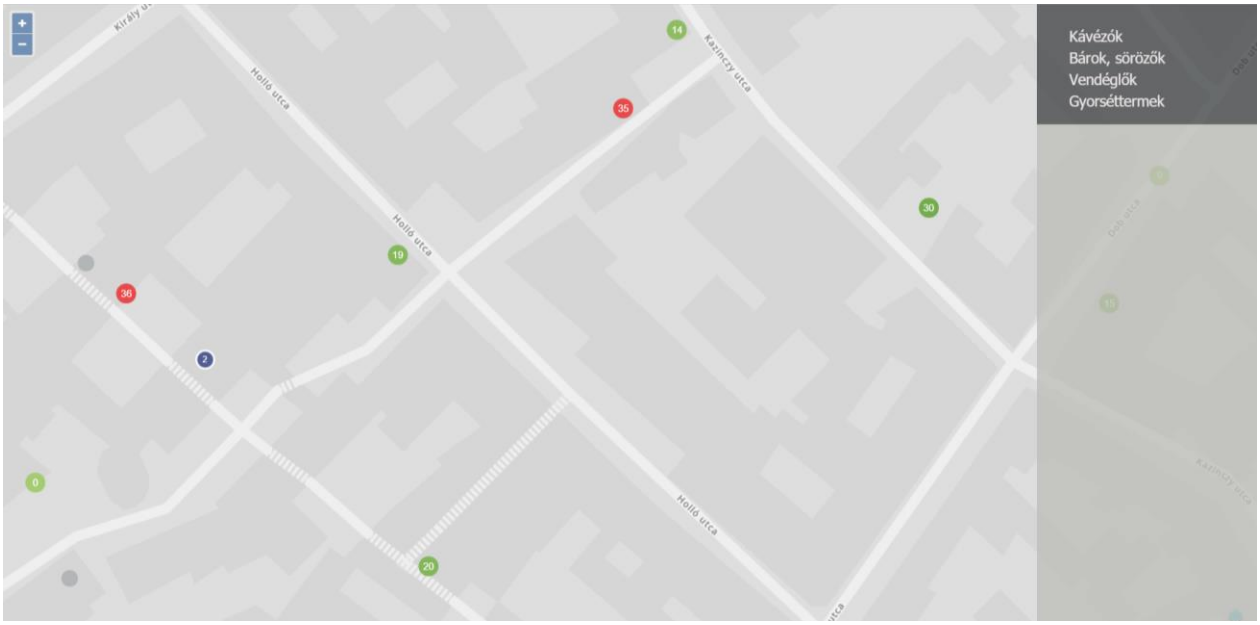
Ha a kiválasztott elem rendelkezik adatokkal, akkor az információk egymás alatt, táblázatos formában jelennek meg. A *Telítettség* tulajdonságnál az érték fekete színnel ábrázolva jelenik meg, amennyiben az adott helynek nincs valós idejű adata, de egy átlagolt érték kiolvasható a grafikonból.

Amennyiben a telítettségi érték piros színnel van ábrázolva, az azt jelenti, hogy az adott pillanatban a kiválasztott hely kapacitása abban a százalékban foglalt. Ez az érték nagyon rövid időn belül, akár percenként is változhat.

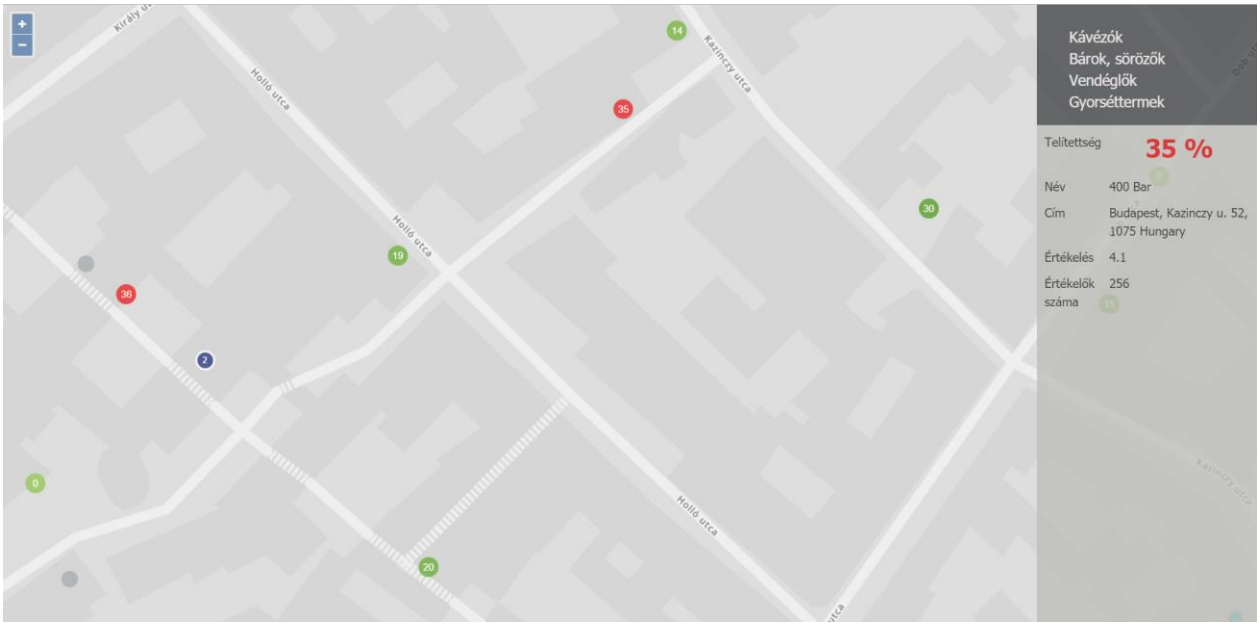


Elérve egy meghatározott nagyítási szintet, illetve ezen érték fölött, az egy elemű klaszterek megjelenítése megváltozik, és benne a telítettségi szint lesz látható. A piros szín árnyalataival kitöltött körök azon helyek esetén jelennek meg, ahol valós idejű adat található. Grafikonból kinyert telítettségi érték esetén a kör színe a zöld egy árnyalatával színeződik, illetve, ha nincs semmi adat, akkor szürkére vált. Az árnyalatok a hely telítettségi értékének függvényében változnak, minél nagyobb, annál intenzívebb, sötétebb, minél kisebb, annál világosabb.

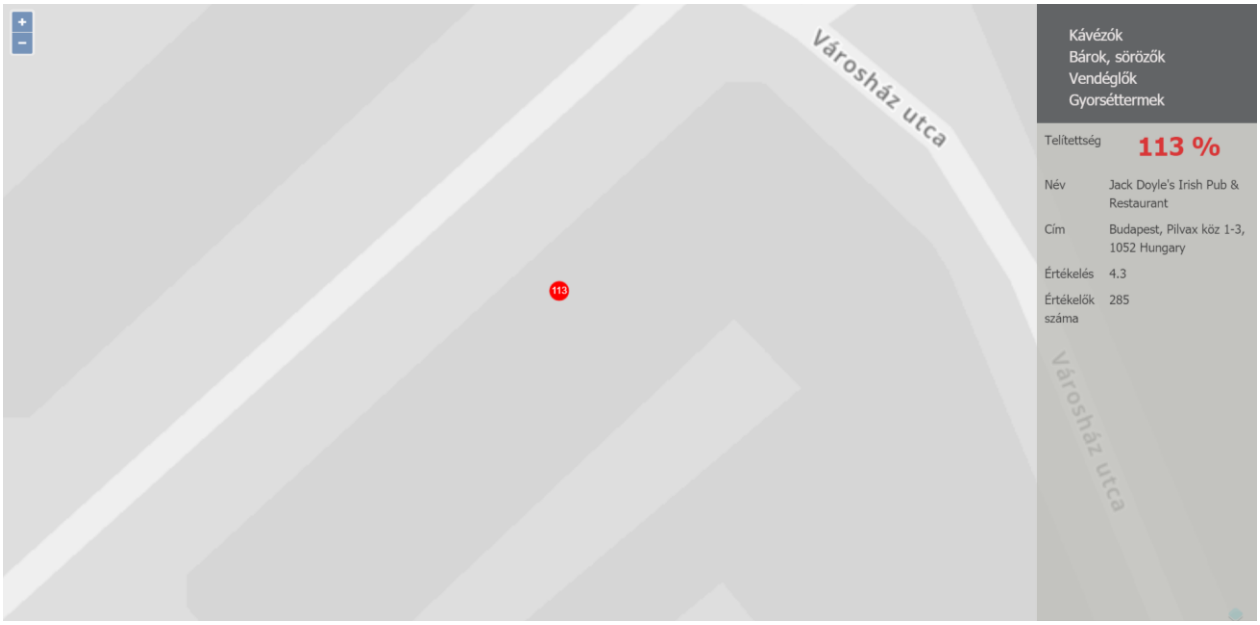
Ha a nagyítás eléri az elemek megváltozásához szükséges szintet, de a térképen nem minden klaszter egy elemű, akkor az adott klaszter kinézete változatlan marad, a többi elem viszont a saját adatainak megfelelően változik.



A megváltozott stílusú körök esetében is kattintásra megjeleníthetők a hely adatai.



Egy érdekesség, amelyre az alkalmazás tesztelése esetén akadtam, hogy a valós idejű érték túllépheti a 100%-ot, vagyis az is megfigyelhető, ha a kiválasztott hely túlzsúfolt.



6. Továbbfejlesztési lehetőségek

Egy alkalmazásra soha nem mondhatjuk, hogy teljesen kész, hiszen időről-időre változnak az igények. A felhasználás során szerzett tapasztalatok, vélemények alapján új ötletek merülnek fel, ezért soha nem lesz végleges állapota. Akár az alkalmazás külalakjára, akár a kód szerkezetére és az elvégezhető műveletekre gondolunk, folyamatosan fejlesztést igényel, új elemekkel, funkciókkal kell bővíteni. Fejlesztési ötleteket kaphatunk a felhasználóktól is, pontosan ezért lehetőséget kell biztosítani számukra is, hogy tudassák az ötleteiket. Ezt többféleképpen meg lehetne valósítani, mint például hozzászólási lehetőséggel az oldalon. A felhasználó az oldal böngészése során leírhatja a véleményét az alkalmazással kapcsolatban.

Egy új funkció lehetne, hogy a felhasználó megadja a keresett hely nevét, a program pedig visszaadja az adatokat. Ez egy hasznos eszköz lenne, hiszen az embernek vannak kedvenc szórakozóhelyei, ahova gyakran benéz, illetve kedvenc étkezői, és így egyszerűen juthatna hozzá az információhoz. Nem kellene a térképen keresgélnie, hogy kiválaszthassa a számára szükséges objektumot, hanem a neve alapján rákeresne, de akár lehetne koordináta, utca név, szolgáltatás, termékek alapján keresni.

A felhasználó helyzetének meghatározása egy másik funkció hozzáadását jelentheti, hiszen ennek függvényében lekérdezhetők a hozzá közel található helyek, majd ezek közül kiválaszthatók azok, amelyek egy előre meghatározott érték alatt vagy fölött vannak. A viszonyítási értéket akár a felhasználó is megadhatná, így az ő elvárásainak, igényeinek megfelelően ajánlhatna az alkalmazás helyeket.

Az előző funkcióból kiindulva, a következő lépés lehetne az útvonal tervezés lehetőségének a hozzáadása. Manapság minden térképes alkalmazáshoz hozzátartozik ez a funkció, hiszen a térkép fő rendeltetései közé tartozik az útvonal meghatározása két kiválasztott pont között, legyen az gyalogos útvonal, vagy tömegközlekedést használva a legrövidebb útvonal.

A jelenlegi formában négy kategóriába vannak sorolva az adatok: kávézók, bárok, vendéglők és gyorséttermek. A Google Places API még számos kategóriával szolgál, amelyek rendelkeznek a szükséges telítettségi adatokkal, és amelyeket az előzőekhez hasonlóan meg lehetne

jeleníteni: ruhás üzleteket, edzőtermeket, metró megállókat, élelmiszer bevásárlókat, múzeumokat, bevásárló központokat, kiemelt látnivalókat, fürdőhelyeket.

A program jelen állapotában JSON fájlban tárolja a helyek adatait, de ezek tartósítására egy jobb megoldási lehetőség az adatbázis, ahonnan egyszerűen lekérdezhetők, feltölthetők és módosíthatók az objektumok. A cél az adatok megbízható, hosszú távú tárolása lenne, illetve a keresett objektumhoz való gyors hozzáférhetőség.

A megjelenítéshez használt OSM adatok, és a Python program által letöltött információk megszerzését dinamikussá kell tenni. Az alkalmazás statikus adatokat használ, így azok frissítését kézzel kell végezni, újra és újra le kell futtatni a megfelelő programkódokat. Ezzel szemben, jó megoldás lenne, ha az információk letöltésére egy olyan alkalmazás készülne, amely automatikusan, előre meghatározott időközönként végrehajtaná a lekérdezéseket.

Igazán használhatóvá akkor válna az alkalmazás, ha telefonon is lehetne használni. A mobil applikációk napjainkban nagyon elterjedtek, mondhatni a mindennapjaink részét képezik, ilyen a Facebook, Messenger, Instagram, különböző okosórákhoz tartozó alkalmazások, táplálkozást megfigyelő programok, játékok. Az emberek többsége elsősorban telefont használ, csak másodsorban laptopot, számítógépet, tehát a jelenlegi webalkalmazás mobilra fejlesztve könnyebben népszerűsíthető lenne. Továbbá az applikáció több fejlesztési lehetőséget is nyújt, ilyen lenne mondjuk a jelzés opció. A felhasználó beállíthatná az általa preferált tulajdonságokat, és miközben sétál, az applikáció jelezne neki, ha az általa kiválasztott jellemzőkkel rendelkező vendéglátóhely mellett vagy közelében haladna el.

A telítettségi értékek százalékos formában érhetők el. Hasznos lenne, ha a helyek befogadóképességéről egy konkrét számértékkel rendelkezhetnék, hiszen nem mindegy, hogy egy 10 fős helyen van 50%-os telítettség, vagy egy 100 fős hely esetében. Az adott hely kapacitását, a férőhelyeinek számát a legegyszerűbben az ott dolgozó személyzettől lehet kideríteni, de ehhez el kell látogatni oda, ami kissé időigényes feladat, de megoldható, és cserébe pontos információval lehet szolgálni a felhasználónak. Akár a vendéglátóhely alkalmazottai is kaphatnak jogosultságot az adatok feltöltésére/változtatására. Minden szórakozóhely kaphat saját profilt, ahova belép, majd egyszerűen, valós időben változtat az adatokon.

Az adatok lekérdezése a Google API-ról egy megírt Python könyvtár felhasználásával történik. Mivel az alkalmazás lényegében Java nyelven íródott, fontos lenne, hogy ez a rész is ezen a nyelven legyen elérhető. A Python kód átírásával egységesíteni lehetne a program különböző részeit, és egy egységes webalkalmazássá válna.

Az általam felsorolt néhány opció csupán a kezdet, hiszen ezek megvalósításának a folyamatában, illetve az elkészült funkció használata közben szerzett tapasztalatok alapján újabb és újabb problémák, észrevételek jelennek meg, amelyekkel még jobbá, hatékonyabbá, a céljának megfelelőbbé tehető az alkalmazás.

7. Összegzés

Ahogy a bevezetőben írtam, a cél egy olyan webalkalmazás létrehozása volt, amely megkönnyíti nagyon sok ember életét, és olyan információkkal szolgál, amelyeket más alkalmazás nem nyújt. Ez a webalkalmazás valós igényeket elégít ki néhány olyan információval, amely más helyen is megtalálható, de olyan jellemzőt is tartalmaz, amely egyedi, és máshol nem fordul elő, ugyanakkor fontos, hogy az előbb említett információk mind egy helyen találhatók.

A dolgozat első felében ismertettem a felhasznált technológiákat, elsőként a webalkalmazás fogalmát, majd összetételét, ezt követően bemutattam a Spring Boot keretrendszert, amely segítségével egy nagy egészként tudom kezelni az alkalmazást. Az applikáció elkészítésekor a Modell-Nézet-Vezérlő tervezési mintát követve hoztam létre a Java osztályokat és a köztük lévő kapcsolatokat. Ennek célja, hogy a felhasználói felület megjelenítéséért felelős kódot szétválassza az alkalmazás üzleti logikáját megvalósító kódjától. A REST, Reprezentációs Adat Átvitel, architekturális stílus, szerver-kliens kapcsolatot jelent, amelyben a szervernek vannak erőforrásai, a kliens pedig kérheti ezeknek az erőforrásoknak a megjelenítését valamilyen módon. A kliens nem tud arról, hogy a szerver hogyan tárolja az adatot, számára az a fontos, hogy érthető módon megjelenjen a kért információ. A REST stílust az adatok közvetítésére használtam a front-end és a back-end között. Következő lépésként, az adatforrások megszerzéséről írtam, melyek kétfélék, egyik kizárólag a front-endhez szükséges, a másik a back-endhez. A front-end adatai közvetetten az OSM adatbázisából származnak, hetente frissülő információk. A back-end számára a Google szolgáltatja az adatokat, a helyek általános tulajdonságait egy Python könyvtár használatával szerzem meg, a valós idejű adatokat pedig Google Map oldalán megnyitott hely DOM-jából bányászom ki.

A következő fejezet a rendszer komponenseit tartalmazza, a megjelenítéshez szükséges HTML, CSS és JavaScript kódot, illetve a Java kódban található osztályokat, amelyek az alkalmazás üzleti logika részét képezik. A felsoroltak mellett bemutatásra került az OpenLayers függvénykönyvtár, amely felhasználásával egyszerűbben megjeleníthető, kezelhető és formázható a térkép, illetve megemlítésre került a MapBox térképszolgáltató, amellyel egyedi térképeket készíthetünk web- és mobilalkalmazásokhoz.

Az alkalmazás bárkinek segít, aki éppen tervezni szeretne egy találkozót, ebédet vagy szórakozási lehetőséget. A felhasználó egyszerűen fellép az oldalra, kiválasztja azt a kategóriát, ahova menni szeretne, aztán egyszerűen a találatokat végig böngészve kiválasztja a neki megfelelő helyet.

Ahogy bármilyen más applikáció esetén, itt is nagyon sok fejlesztési lehetőség van. Többek között lehet fejleszteni az adatok megszerzéséhez használt módot, például a jelenlegi Google API-ról való lekérdezések helyett a felhasználók is megadhatnának információkat, vagy saját adatbázist is készíthetünk. Az adatok típusát illetően, bővíteni lehet a megjelenítendő információkat az étlappal, nyitvatartási idővel, útvonal tervezéssel, asztalok és a hozzuk tartozó székek számával, azoknak a kategóriáknak a felsorolásával, amibe a kiválasztott hely besorolható, vagyis a hely stílusának megadásával. A keresési mód fejlesztése egy újabb lehetőség, akár nyitva tartás, szolgáltatás, termék, stílus vagy ár alapján.

Végezetül az alkalmazás, mint bármilyen új dolog, a kezdeti állapothoz képest még nagyon sokat változhat, a felhasználók visszajelzése, igénye vagy egyéb tényezők alapján.

8. Köszönetnyilvánítás

A dolgozat megírása során több oldalról kaptam segítséget, így köszönetet szeretnék mondani vezetőtanáromnak, dr. Gede Mátyásnak az iránymutatásért, illetve az Urban Institute [UI!] vállalkozásnál lévő munkatársaimnak, hogy szakmai észrevételekkel segítettek, és hogy biztosították a dolgozat elkészítéséhez szükséges időt.

9. Irodalomjegyzék

Felhasznált nyomtatott, és interneten elérhető kiadványok

Leon Shklar, Richard Rosen (2003): Web Application Architecture. Principles, protocols and practices, JohnWiley & Sons Ltd, West Sussex, England http://www.ce.uniroma2.it/courses/PRSI/WebApplication.pdf Utolsó elérés: 2017. november 02.

Craig Walls (2015): Spring in Action, Manning Publications Co., United States of America

Craig Walls (2016): Spring Boot in Action, Manning Publications Co., United States of America

Péntek István (2010): Webalkalmazásfejlesztés java technológiákkal. Spring Web MVC, Diplomamunka, Debrecen https://dea.lib.unideb.hu/dea/bitstream/handle/2437/95230/Szakdolgozat.pdf?sequence=1 Utolsó elérés: 2017. november 10.

Pótári Tamás (2011): RESTful web-szolgáltatások készítése WCF-ben, Diplomamunka, Szeged https://devportal.hu/download/Publikaciok/RESTful-web-szolgaltatasok-keszitese-WCF-ben-Potari-Tamas.pdf Utolsó elérés: 2017. november 12.

Leonard Richardson, Sam Ruby (2007): RESTful Web Services, O'Reilly Media, United States of America

Kenneth Lange (2016): The Little Book on REST Services, Copenhagen

Cody Lindley (2017): Front-End Developer Handbook.

Gede Mátyás (2014): Open Source rendszerek a térinformatikai gyakorlatban – Interaktív webtérképek készítése OpenLayers és MapServer használatával. http://tarsadalominformatika.elte.hu/tananyagok/opensource/lecke1_lap1.html Utolsó elérés: 2017. december 10.

Szabó Gergő (2010): Quantum GIS. Felhasználói kézikönyv.

Felhasznált internetes oldalak

http://tudasbazis.sulinet.hu/hu/informatika/informatika/informatika-9-12-evfolyam/programozas-c-kornyezetben/webes-alkalmazasok-keszitese Utolsó elérés: 2017. november 02.

http://whatis.techtarget.com/definition/Web-server Utolsó elérés: 2017. november 02.

http://tomcat.apache.org/ Utolsó elérés: 2017. november 02.

https://maven.apache.org/ Utolsó elérés: 2017. november 07.

http://www.jtechlog.hu/2010/04/28/maven-kezdolepesek.html Utolsó elérés: 2017. november 07.

http://web.progtanulo.hu/web-programozas-alapismeretek/3-szerver-oldali-mukodes/310-tervezesi-mintak/3103-mvc Utolsó elérés: 2017. november 10.

http://docs.openlayers.org/ Utolsó elérés: 2017. november 10.

http://tarsadalominformatika.elte.hu/tananyagok/opengis_qgis/lecke1_lap1.html Utolsó elérés: 2017. november 10.

https://platform.html5.org/ Utolsó elérés: 2017. november 17.

https://www.w3.org/Style/CSS/Overview.en.html Utolsó elérés: 2017. november 17.

https://www.w3.org/DOM/#what Utolsó elérés: 2017. november 25.

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model Utolsó elérés: 2017. november 25.

http://webprogramozas.inf.elte.hu/tananyag/wf2/lecke7_lap1.html Utolsó elérés: 2017. november 25.

Adatok forrása

Letöltés: https://mapzen.com/data/metro-extracts/ Utolsó elérés: 2017. október 03.

MapBox: https://www.mapbox.com/ Utolsó elérés: 2017. december 15.

NYILATKOZAT

Alulírott **Veress Orsolya** (NEPTUN azonosító: **X5TCRG**) az „Térképes szabad asztal kereső” című diplomamunka szerzője fegyelmi felelősségem tudatában kijelentem, hogy dolgozatom önálló munkám eredménye, saját szellemi termékem, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

A témavezető által elfogadott és elbírált diplomamunka elektronikus közzétételéhez (PDF formátumban a tanszéki honlapon) **hozzájárulok**.

Budapest, 2018. január 8. ……

a hallgató aláírása

Hozzájárulok a szakdolgozat benyújtásához:

Budapest, 2018. január 8. ……

a témavezető aláírása