

Github Analyzer

Github repository statisztika vizualizációs szoftver

Rendszerterv

Szoftverarchitektúrák házi feladat

Készítették:

Németh Marcell és Morvai Dániel

Tartalomjegyzék

Követelményspecifikáció.....	3
Feladatkiírás	3
Fejlesztői csapat	3
Részletes feladatleírás.....	3
Technikai paraméterek.....	3
Szótár.....	4
Fontosabb use-case diagrammok.....	4
1. A rendszer célja és funkciói	5
1.1. Feladatkiírás	5
1.2. A rendszer által megvalósított funkciók.....	5
2. Felhasznált technológiák	6
2.1. Spring Boot Kotlinban.....	6
2.2. Typescript, Angular.....	6
2.3. IntelliJ IDEA.....	6
2.4. Egyéb felhasznált eszközök	6
3. Tervezés.....	8
3.1. A rendszer architektúrája	8
3.2. Adat réteg.....	9
3.2.1. GitHub API részletek.....	9
3.2.2. Adatbázis	10
3.3. Üzleti logikai réteg.....	11
3.3.1. Kielemzett válasz entitások	12
3.3.1. Megvalósítandó végpontok és feladataik	14
3.4. Megjelenítési réteg.....	15
3.4.1. Tervezett komponensek.....	15
3.4.2. Komponens diagram	15
4. Megvalósítás.....	17
4.1. Felhasznált architektúrális minták.....	17
4.1.1. Frontend pollozása	17
4.1.2. Szerver válaszána feldarabolása	18
4.1.3. Szerver válaszána gyorsítótárba töltése	18
4.3. A megvalósított rendszer	18
5. Telepítési leírás.....	22
6. Felhasznált eszközök	23
7. Összefoglalás	24
8. Továbbfejlesztési lehetőségek	25
9. Hivatkozások.....	26

Követelményspecifikáció

Feladatkiírás

A rendszer egy nyilvános webalkalmazás, amely GitHub repository-k statisztikai adatait tudja kinyerni vagy GitHub API-ról, vagy a repository tartalmából. A hozzáféréshez nyilvános repository alapján nincs szükség, csak egy URL-re. Privát repository esetén egy GitHub-on készített access token megadás szükséges. A repository statisztikáit szövegesen és grafikusán mutatja az alkalmazás, amelyek közül pár példa: commitok száma, commitok időbeli eloszlása, commitok szerzőinek hozzájárulása, sorok számának alakulása stb.

Fejlesztői csapat

Csapattag neve	NEPTUN-kód	E-mail cím
Németh Marcell	F2AVXH	nmarci2008@gmail.com
Morvai Dániel	JG6L5H	morvai.daniel96@gmail.com

Részletes feladatléírás

A projekt során célunk egy olyan webalkalmazás készítése, amely képes egy felhasználó által választott GitHub *repository*-ról statisztikákat készíteni és vizualizálni. Az applikáció, mint kvázi BI eszköz, segítségével hatékonyabbá tehető a fejlesztői csapatok/egyének teljesítményének menedzselése, illetve esetleges javítása.

A kiválasztott repository életciklusa a webalkalmazás futása során a következő:

- Repository (továbbiakban tároló) URL kézi megadása (a kulcs validálása nem feladata a programnak, a felelősség a felhasználót terheli)
- A tároló felvételre kerül egy helyi metadata adatbázisba, amely egyfajta memóriaként, cache-ként működik egy adott időkereten belüli többszörös lekérdezések adatforgalom csökkentése érdekében.
- A feldolgozó motor (back-end) beolvassa a tároló tartalmát egy lokális memóriába és elkészíti a leíró statisztikákat, amelyek a következők:
 - Commitok száma: fejlesztők hozzájárulása a kódbázishoz
 - Legtöbb committal rendelkező fejlesztő és egy másik repositoryba legtöbbet commitoló fejlesztő összehasonlítása
 - Két repository fejlődésének üteme: hozzáadott commitok számának növekedése/időszak
 - Commitok időbeli eloszlása: év, hét, óra
 - Összes kód hozzáadás, törlés száma: év, hét
- Statisztikák eltárolása az adatbázisban
- A tároló feldolgozásának időigényessége miatt, a kliens aszinkron vár a szerver válaszára az eredmény pollozásával, a lekérdezés idejétől a válasz megérkezéséig.
- Statisztikák megjelenítése, amelyek a következő vizualizációkat jelenti:
 - Hisztogram az egyes fejlesztők commit számainak eloszlásáról
 - Pie-chart (kördiagram) a kód hozzáadás, törlés megoszlásáról
 - Line-chart (vonaldiagram) a kommitszámok időbeli változásáról

Technikai paraméterek

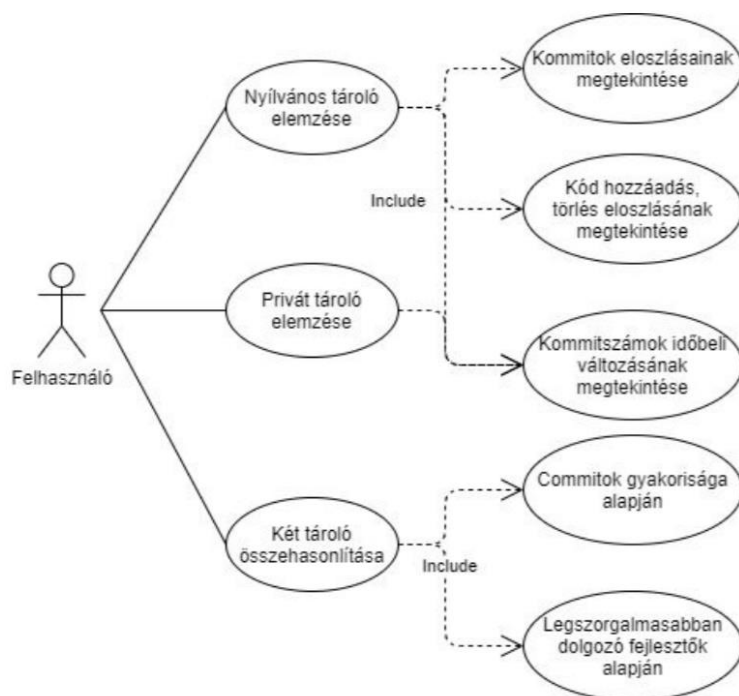
A definiált alkalmazást architektúrájából adódóan két részre osztottuk: back-end és front-end. A szerver feladata az üzleti logika megvalósítása: kapott URL alapján adatok biztosítása a webes kliens felé, amely ezeket megjeleníti. A két réteg egymással REST API hívásokkal kommunikál. A back-end

motort Spring Boot technológiával, Kotlin nyelven készítjük el annak érdekében, hogy több operációs rendszeren is lehessen használni. Az alkalmazás meta-adat adatbázisát egy Oracle SQL vagy hasonló adatbázis fogja tárolni. A front-end megvalósításához az Angular ketrendszert fogjuk használni.

Szótár

Entitás	Magyarázat
GitHub	Nyilvánosan elérhető kódbázis, amely fejlesztői csapatok kódmenedzsmentjét segíti
Repository	Tároló alkönyvtár a GitHub kódbázisban, amely különálló projektek forráskódját tárolja
BI eszköz	Adatok beolvasását, tárolását, feldolgozását, elemzését megvalósító eszköz
Github access token	Github felhasználók által készíthető karaktersorozat, amelyből információk nyerhetők ki a felhasználó tárolójáról.
Metadata adatbázis	Az alkalmazás adatbázisa melyben tárol cachelni kívánt adatokat a gyorsabb kiszolgálás végett.
Front-end	A webes kliens program, amely kérésekkel fordul a szerverhez.
Back-end	Az alkalmazás szerver, amely egy távoli gépen fut és kliensek kéréseit képes feldolgozni.
Polling	A kliens program folytonos szinkron kérdése a szerver felé.
Commit	Tárolóba a kód jelenlegi állapotának rögzítése adott felhasználó által.

Fontosabb use-case diagrammok



1. A rendszer célja és funkciói

1.1. Feladatkiírás

A rendszer egy nyilvános webalkalmazás, amely *GitHub repository*-k statisztikai adatait tudja kinyerni *GitHub API*-ról vagy a *repository* tartalmából. A hozzáférésre nyilvános *repository* esetén csak egy URL-re van szükség, privát tárolóhoz való hozzáféréskor egy *GitHub*-on készített *access token* megadás szükséges. A *repository* statisztikáit szövegesen és grafikusan is mutatja az alkalmazás, amelyek közül pár példa: commitok száma, commitok időbeli eloszlása, commitok szerzőinek hozzájárulása, sorok számának alakulása stb.

1.2. A rendszer által megvalósított funkciók

A program fő célja egy vagy több kódrendszer fejlődési ütemének vizsgálata leíró statisztikák segítségével. A készített statisztikák vizualizálása az emberi megértést és feldolgozást teszi könnyebbé diagrammokon keresztül. Az előbb nagyvonalakban felvázolt fő funkciók megvalósításához több, kisebb alfeladatot kell elvégeznie a rendszernek, amelyek a következők:

- Felhasználóval való interakció: *GUI* biztosítása, input adatok kezelése, parancsok fogadása.
- *Repository* metaadatok számolása és tárolása: egy olyan nagyobb lokális adatbázis hiányában, amely a tároló teljes tartalmát raktározna, egy leíró adatokat tároló, cache-ként működő memória került használatra, amelyben megtalálhatóak és lekérdezhetőek egy bizonyos időkereten belül a kódrendszer fejlesztői hozzájárulásai.
- Vizualizációk, további felhasználói interakciók: a számolt adatokat a felhasználó számára értelmezhető formátumban kell megjeleníteni a *frontenden*, azaz utolsó alfeladatként meg kell jeleníteni a diagrammokat, amelyeket a későbbiekben részletesebben is bemutatunk.

2. Felhasznált technológiák

A feladat megoldásához használtunk általunk már jól ismert, illetve újdonságnak számító technológiákat és eszközöket is. A szerver oldal *Spring Boot*-ot, a webes alkalmazás pedig az *Angular* keretrendszert használva készült el.

2.1. Spring Boot Kotlinban

Az alkalmazás *REST API* típusú architektúrán alapszik. A kliensek a szerverrel HTTP kérések segítségével kommunikálnak. Az egyes kéréseket a szerver feldolgozza és visszaküldi a kliensnek. A szerver végpontokat definiál, amiknek a címét ismerve szólítják meg őket a kliens alkalmazások. Kihhasználva a HTTP tulajdonságait, a kérések fejléc, törzs, URL részeiben utazhatnak adatok.

A szerver megírásához a *Spring Boot*[1] technológiát használtuk, ami egy *Java* alapú keretrendszer *micro-servicek* írásához. Azért ezt a keretrendszert választottuk, mert egyszerűen, annotációkat használva lehet adatbázis lekérdezéseket kezelni és egy *REST* végpontokkal rendelkező alkalmazást fejleszteni. Az egyes függőségek és külső könyvtárak használatára is egyszerű szintaktikát nyújt.

2.2. Typescript, Angular

Az *Angular*[2] egy Google által fejlesztett, *Typescript* alapú, nyílt forráskódú keretrendszer webes alkalmazások fejlesztéséhez. Kezdetben *AngularJS* néven indult, de az *Angular2*-vel (2014) sok mindent változtattak rajta, többek között inntől váltotta fel a *Javascript* alapú fejlesztést a *Typescript*. Mivel egy komplett keretrendszer, ezért számos beépített segédosztály áll rendelkezésünkre.

Az alkalmazásnál kihasználtuk, hogy az *Angular* komponens alapú, azaz, hogy az egyes oldalak modulokból épülnek fel. Próbáltuk úgy szervezni őket, hogy minél több komponens újra tudjunk használni esetleg pár kisebb dolog megváltoztatásával.

2.3. IntelliJ IDEA

Feladatunkhoz a diákoknak járó *Ultimate Edition*-t használtuk, mert a funkciók egy része csak itt működik, mint például az általunk használt *Spring Boot* is. Fejlesztés közben sokat használtuk a fejlesztői környezet kód kiegészítő és navigációs funkcióit. Beépített *Git* támogatás segítségével minden kisebb feladat elvégzése után commit-oltunk a *GitHub* szerverre.

2.4. Egyéb felhasznált eszközök

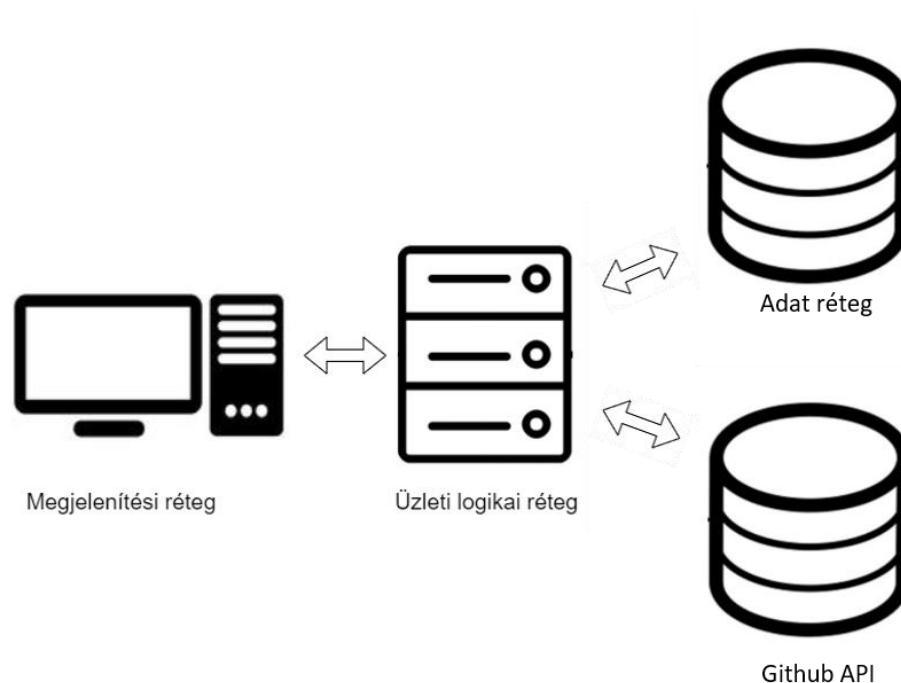
Az előző részekben felsorolt fő technológiák mellett a következőket is alkalmaztuk a fejlesztés során:

- *Kohsuke Github Api Wrapper*[5]: A *Github API Kotlin* kódból való eléréséhez az említett *open-source* könyvtárat használtuk.
- *Oracle SQL Developer*[3]: A fentebb említett *Github API*-tól nyert adatokat cachelési és célból az *SQL Developer* relációs adatbáziskezelő rendszer segítségével tároltuk.
- *Bootstrap és Angular material*: A UI szép megjelenéséhez használtuk a két könyvtárat. A *Bootstrap* navigációs fejlécét a menü elkészítéséhez, illetve *Grid* rendszerét az adott elemek elrendezéséhez és az oldal reszponzivitásához. Az *Angular material*-t az oldalon található elemekhez használtuk fel (gombok, választó gombok, kártyák, panelek)
- *Apexcharts.js*: A csomagot a kiszámolt statisztikák megjelenítéséhez használtuk. Segítségével egyszerűen tudtunk különböző diagrammokat rajzolni az egyes panelekbe.
- *Postman, Chrome DevTools*: A két eszközt a *backend*, illetve a *frontend* teszteléséhez, *debuggolásához* használtuk.

3. Tervezés

3.1. A rendszer architektúrája

Alkalmazásunk felépítése a háromrétegű architektúra mintájára készült el. Ez az elrendezés kiosztja és definiálja a kliens és a szerver feladatait. Esetünkben az architektúra adat-, üzleti logikai- és megjelenítési rétegekből tevődik össze, melyet az alábbi ábra szemléltet.



3.1. ábra: Háromrétegű architektúra felépítése

A réteges architektúra fő előnye, hogy minden réteg a saját, különálló feladatát végzi, függetlenül a további rétegek feladataitól. Csak a szomszédjaikkal kommunikálnak: vagy egy szolgáltatást nyújtanak nekik, vagy fordítva a szomszédok végeznek nekik feladatokat. Az alkalmazás rétegekre bontásával a fejlesztők könnyen karbantartható és fejleszthető rendszereket hozhatnak létre, mivel elegendő egy-egy réteget javítani, illetve bővíteni az egész alkalmazás módosítása helyett. A modell legnagyobb előnye, hogy lehetővé teszi az egyes rétegek egymástól függetlenül történő fejlesztését, sőt, akár teljes cseréjét is, lépést tartva így a folyamatosan változó követelményekkel és az egyre újabb technológiákkal.

Az alkalmazásunkhoz használt rétegek és feladataik:

- Az *adat réteg*: esetünkben az adatréteget a *GitHub API*[4]-ja, illetve egy relációs adatábázis valósította meg. Az *interface*-en keresztül a *merge* és üres commitokon kívül szinte minden fajta kódváltozásról tudtunk adatokat kinyerni. Az *API*-n keresztül a következő adatokhoz férünk hozzá:
 - fejlesztők azonosítói
 - commitok száma fejlesztőnként

- commitok időbélyege
- kód-hozzáadások száma
- kód-törlések száma

A *GitHub*-tól nyert adatokat és a generált statisztikákat cachelési célból egy relációs adatbázisban tároltuk.

- Az *üzleti logikai réteg* feladata a megjelenítési réteg és az adat réteg logikai összekötése. A *frontendtől* kapott *repository* cím alapján a *GitHub API*-tól adatok kérése, majd statisztikák előállítása.
- A *megjelenítési réteg* fő feladata az üzleti logikai rétegtől kapott adatok megjelenítése. Estünkben a réteget egy webes applikációval valósítottuk meg.

3.2. Adat réteg

Az adatréteg funkcióit a *GitHub Repository Statistics API* felhasználásával valósítottuk meg, amely lehetővé teszi különböző típusú tárolók aktivitásának elemzését.

Az *API*-n keresztül lekérdezett adatokat és az ezekből előállított statisztikákat egy relációs adatbázisban tároltuk el, amely egy gyorsítótár funkcióját töltötte be.

3.2.1. GitHub API részletek

Fontos megjegyzés, hogy a kódbázisokat jellemző mérőszámok lekérdezése/előállítása erőforrásigényes feladat. Annak érdekében, hogy a lekérdezési folyamat minél gördülékenyebben mehessen végbe az *API* többféle átmeneti memóriában való *cache*-elést valósít meg, amelyek aktuális állapotairól különböző *REST* hívás-válaszokkal tájékoztatja a fejlesztőket. A fejlesztés során a következő válaszokkal találkoztunk:

- *202*: az adat nem található meg a *cache*-ben, a statisztikák elkészítését egy háttérfolyamat megkezdte.
- *200*: jelzés, hogy a *202*-vel jelzett háttérfolyamat befejeződött, a statisztikák betöltődtek a gyorsítótárba és szabadon felhasználhatóak.

A következőkben pár példát mutatunk az *API* hívásokra és szintaxisukra:

- adott fejlesztő módosításainak száma:
GET :username/:repository , commits
- adott *repository* összes fejlesztőjének módosításai:
GET /repos/:owner/:repo/stats/contributors

Az összes fejlesztőre lekérdezett aktivitás *API* válaszáinak formátumát a következő részlet demonstrálja:

Status: 200 OK

```
[
  {
    "author": {
      "login": "octocat",
      "id": 1,
      "node_id": "MDQ6VXN1c2E=",
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "",
      "url": "https://api.github.com/users/octocat",
      "html_url": "https://github.com/octocat",
      "followers_url": "https://api.github.com/users/octocat/followers",
      "following_url": "https://api.github.com/users/octocat/following{/other_user}",
      "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/octocat/starred{/owner}/{/repo}",
      "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
      "organizations_url": "https://api.github.com/users/octocat/orgs",
      "repos_url": "https://api.github.com/users/octocat/repos",
      "events_url": "https://api.github.com/users/octocat/events{/privacy}",
      "received_events_url": "https://api.github.com/users/octocat/received_events",
      "type": "User",
      "site_admin": false
    },
    "total": 135,
    "weeks": [
      {
        "w": "1367712000",
        "a": 6898,
        "d": 77,
        "c": 10
      }
    ]
  }
]
```

3.2. ábra: GitHub API válasz formátuma

3.2.2. Adatbázis

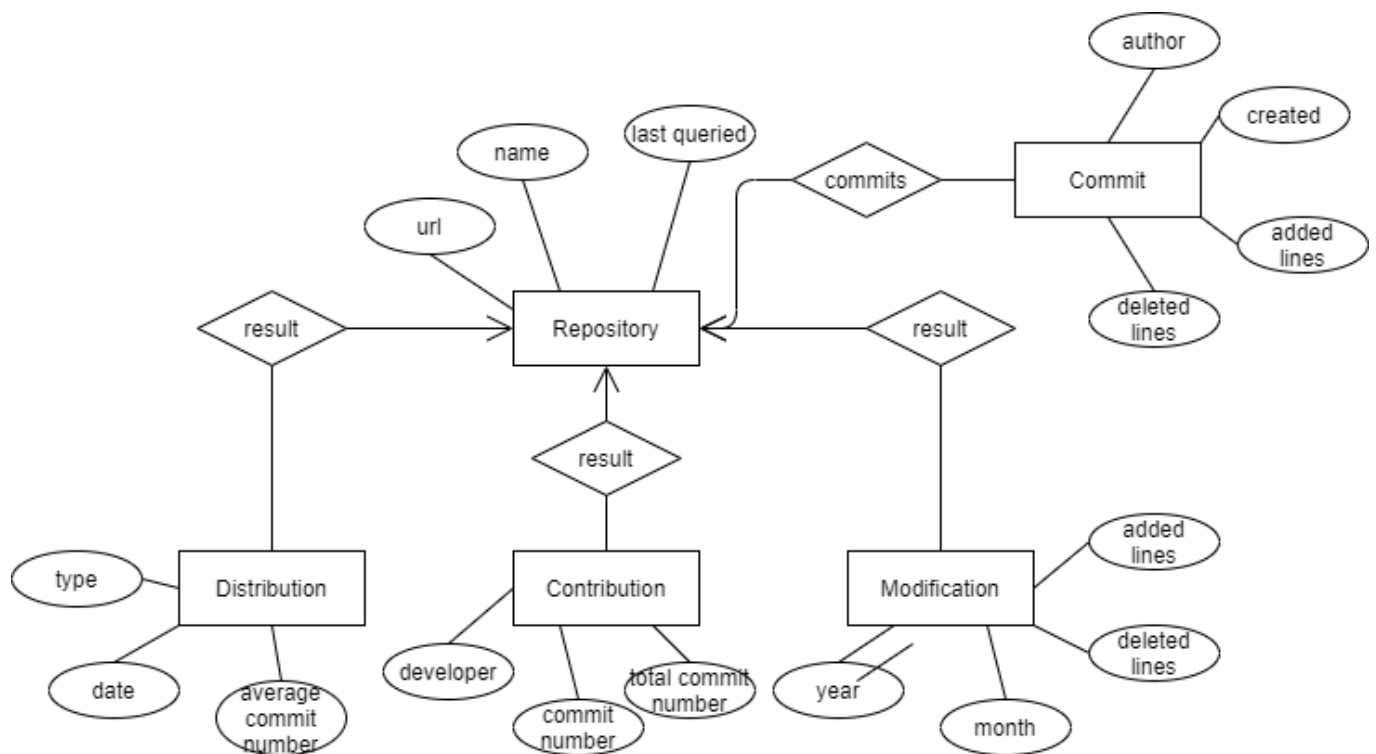
Az előző fejezetben bemutatott *API* hívások segítségével hozzáfértünk a *repository*-k aktivitásának alapszintű mérőszámaihoz. Következő lépésként, a magasabb szintű statisztikák elkészítéséhez, majd tárolásához egy adatbázist definiáltunk.

Az adatbázisban egyrészt tároltuk a már lekérdezett *repository*-khoz tartozó commitokat. Emelett tároltuk a commitok alapján kiszámolt válaszokat is. Ezen adatokat gyorsítótárként használtuk a jövőbeli statisztikák elemzéséhez, lekérdezések gyorsításához.

Az adatbázisba az alábbi entitások kerülnek mentésre:

- Commit
 - lista amit a *repository* neve alapján kérhetünk le
 - felhasznált tulajdonságok: *author*, *date*, *message*

- Repository
 - felhasznált tulajdonságok: *name, url, commits, last queried, results*
- Contribution Result
 - adott *repository*-hoz számolt lista, mely tartalmazza az egyes fejlesztők kommitjainak a számát
 - felhasznált tulajdonságok: *developer, commit number, total commit number*
- Modification Result
 - adott *repository*-hoz számolt lista, mely tartalmazza hónapok szerint csoportosítva a komitokkori törölt/hozzáadott sorokat
 - felhasznált tulajdonságok: *year, month, added lines, deleted lines*
- Distribution Result
 - adott *repository*-hoz számolt lista, mely tartalmazza típus szerinti idő periódusokhoz tartozó átlagos kommit számot. A típus lehet hét, nap, napszakasz.
 - felhasznált tulajdonságok: *type, date, average commit number*



3.3. ábra: Meta-adatbázis ER diagramja

3.3. Üzleti logikai réteg

Az alábbi fejezetben taglalt réteg feladata az üzleti logika, azaz rendszerünk esetében a leíró statisztikák kiszámolása. A rendszer két fő feladatot képes ellátni:

- kimutatások készítése egyetlen *repository* alapján (*single repository analysis*)

- kimutatások készítése két különböző *repository* összehasonlításával (*double repository analysis*)

A program működése két fő fázisra bontható, amelyek során az előbb felsorolt feladatokat a rendszer *backend* része valósítja meg a következő alfeladatok elvégzésével:

1. A *frontend* a beolvasott *repository* URL-t (privát tároló esetén az *access token*-t is) leküldi a feldolgozó motornak, majd a háttér folyamat elindítása után hagyja a backendet aszinkron módon dolgozni. A háttér folyamat során a motor kielemez az adatokat és előállítja a vizualizációkhoz szükséges választ.
2. Az eredmények elérése érdekében a *frontend* folyamatosan *poll* lekérdezéseket küld a *backend*-nek, egészen addig, amíg meg nem kapja a választ a statisztikákkal. Az adatok elérése után a *polling* leáll.

A rendszer továbbá nyilvántartja az egyszer már lekérdezett URL - *access token* párosokat, így képes az időigényes ismétlődő lekérdezéseket gyorsabban végrehajtani az adatbázis segítségével.

A *gyorsítótár* működése, az adatbázisban tárolt adatok segítségével, a következő: ha az adott *repository*-ba nem került új commit a legutóbbi elemzés óta, akkor a statisztika újra számolása nélkül, a számokat egyből visszaküldtük az adatbázisból a frontendnek. Ha már vannak korábbi eredményeink a *repository*-ról, viszont kerültek bele újabb commitok utolsó elemzésünk óta, csak ezeket az új commitokat kértük le a *Github API*-tól, lementettük őket adatbázisba és újraszámoltuk a statisztikákat. Az utolsó esetben, amikor nem voltak korábbi számításaink a *repository*-ról nem tudunk gyorsítótárat alkalmazni: lementjük a commitokat, generáljuk az eredményeket.

3.3.1. Kielemezett válasz entitások

Ahogy az már korábban említésre került, a *backend* számolja ki a kívánt válasz entitások tulajdonságait, a megjelenítendő számszerű statisztikákat. A következő táblázatban részletezésre kerülnek az egyes *feature*-ök (statisztikák) jellemzői.

Funkció	Név	Szükséges adatok	Adattípus	Megjelenítés
Fejlesztők hozzájárulása a kódbázishoz	<i>Contribution Response</i>	Fejlesztők listája	<i>list</i>	hisztogram
		Fejlesztők commitjainak száma	<i>list</i>	
		Repositoryban történt összes commit száma	<i>int</i>	

Legtöbb committos fejlesztők összehasonlítása	<i>Developer Compare Response</i>	Repository_1 legtöbb commitos fejlesztő	<i>string</i>	vonal-/oszlopdiagram
		Repository_2 legtöbb commitos fejlesztő	<i>string</i>	
		Repository_1 legtöbb commitos fejlesztő commitszámai havonta	<i>list</i>	
		Repository_2 legtöbb commitos fejlesztő commitszámai havonta	<i>list</i>	
		Repository_1 vizsgált időszak éve	<i>date</i>	
		Repository_2 vizsgált időszak éve	<i>date</i>	
Két repository fejlődésének üteme	<i>Development Compare Response</i>	Repository_1 hozzáadás commitok száma havonta	<i>list</i>	Vonaldiagram
		Repository_2 hozzáadás commitok száma havonta	<i>list</i>	
		Repository_1 fejlesztők száma	<i>int</i>	
		Repository_2 fejlesztők száma	<i>int</i>	
Commitok időbeli eloszlása	<i>Distribution Response</i>	Commitok átlagos száma havonta	<i>list</i>	vonal-/oszlopdiagram
		Commitok átlagos száma a hét napjain	<i>list</i>	
		Commitok átlagos száma egy napon belül (8-12, 12-16, 16-20, 20-24, 0-8)	<i>list</i>	
Összes kód hozzáadás, törlés	<i>Modification Response</i>	Vizsgált év	<i>date</i>	kördiagram
		Vizsgált hét/hónap	<i>date</i>	
		Törlések száma	<i>int</i>	
		Hozzáadások száma	<i>int</i>	

Az átláthatóság kedvéért az egyes statisztikák előállításának módja:

Feature	Számítás
<i>ContributionResponse</i>	Kommitok csoportosítása fejlesztő szerint majd csoportok méretének lekérdezése.
<i>DistributionResponse</i>	Kommitok csoportosítása hónap/nap/napszak szerint, majd csoportok méretének lekérdezése, ezek átlagolása.

<i>ModificationResponse</i>	Kommitok csoportosítása hónap szerint, majd adott csoporthoz tartozó kommitok törölt, illetve hozzáadott sorainak összegzése.
<i>DevelopmentCompareResponse</i>	Kommitok utolsó évének számolása. Kommitok csoportosítása hónap szerint, majd utolsó évbeli kommitok méretének lekérdezése.
<i>DeveloperCompareResponse</i>	Legtöbb kommittal rendelkező fejlesztő meghatározása. Kommitok utolsó évének számolása. Kommitok csoportosítása hónap szerint, majd utolsó évbeli kommitok méretének lekérdezése melyek az fentebb kiszámolt fejlesztőhöz tartoznak.

3.3.1. Megvalósítandó végpontok és feladataik

A következőkben az *API* hívásokon keresztül mutatjuk be az elkészült végpontokat és a hozzájuk tartozó funkciókat.

1. Egy tároló elemzése

Feladata a fent említett válaszok előállítása egy tároló elemzéséhez.

- *POST /repository/single/analyze*

Kiértékelendő tároló URL címének, illetve privát tároló esetén a felhasználó *GitHub access token*-ének küldése. Hatására a szerver lekéri a *GitHub*-tól a tárolóról tárolt adatokat és statisztikákat készít belőlük.

- *POST /repository/single/contribution*

Kiértékelendő tároló URL cím és a felhasználó *access tokenje* alapján hozzájárulási statisztikák lekérése. Ha az adat nem áll még rendelkezésre üres választ kapunk.

- *POST /repository/single/modification*

Kiértékelendő tároló URL cím és a felhasználó *access tokenje* alapján módosítási statisztikák lekérése. Ha az adat nem áll még rendelkezésre üres választ kapunk.

- *POST /repository/single/distribution*

Kiértékelendő tároló URL cím és a felhasználó *access tokenje* alapján eloszlási statisztikák lekérése. Ha az adat nem áll még rendelkezésre üres választ kapunk.

2. Két tároló összehasonlítása

Feladata a fent említett válaszok előállítása két tároló összehasonlításához.

- *POST /repository/double/analyze*

A két kiértékelendő tároló URL címeinek, illetve privát tároló esetén a felhasználó *GitHub access token*-jének küldése. Hatására a szerver lekéri a *GitHub*-tól a tárolókról tárolt adatokat és statisztikákat készít belőlük.

- *POST /repository/ double /development*

A két kiértékelendő tároló URL címei és a felhasználó *access tokenje* alapján fejlődési statisztikák lekérése. Ha az adat nem áll rendelkezésre üres választ kapunk.

- *POST /repository/ double /developer*

A két kiértékelendő tároló URL címei és a felhasználó *access tokenje* alapján legjobb fejlesztői statisztikák lekérése. Ha az adat nem áll még rendelkezésre üres választ kapunk.

3.4. Megjelenítési réteg

A megjelenítés architektúráját az *Angular* keretrendszerbe beépített *MVVM*-nek nevezett tervezési mintával terveztük megvalósítani, melynek három fő komponense van: a *Model*, a *View* és a *ViewModel*.

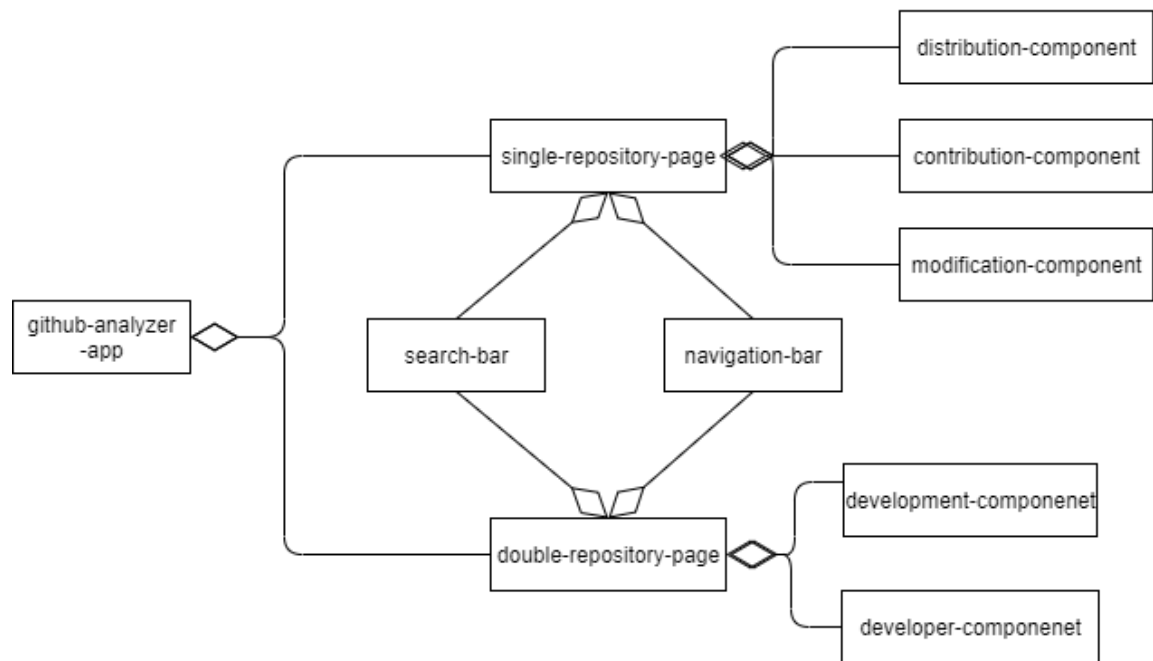
3.4.1. Tervezett komponensek

A *frontend*, azaz a vizuális megjelenítésért felelős felület tervezése során a következő komponensekből építettük fel a rendszert:

- Két különálló oldal, amelyek a különálló és a két különböző *repository* összehasonlítását jelenítik meg.
- Az oldalak közötti navigációt egy *navbar* menüsor, fejléc segíti.
- Az egyes oldalakon található több kereső mező, amelybe a felhasználó megadhatja a keresett *repository* URL-jét, illetve privát tároló esetén az *access token*-t.
- A statisztikák megjelenítése különböző, később részletesen bemutatott diagrammok segítségével történik, amelyek a nagy felbontásuk miatt az átláthatóság kedvéért összecukhatóak az alkalmazott kártyás nézetnek köszönhetően.

3.4.2. Komponens diagram

Mivel az *Angular* komponens alapú, a UI tervezésekor komponensekben kell gondolkodni. Az előző oldalakban tervezett megjelenési tervek alapján az alábbi ábrán látható komponens diagramot készítettük el.



3.4. ábra: Frontend komponens diagram

4. Megvalósítás

A tervezés után az alkalmazás implementálása következett. A fejlesztést *bottom-up* módon végeztük: először megértettük a *GitHub API*-ja által biztosított adatokat, majd folytattuk az üzleti logikával, a statisztikák elkészítésével, végül elkészítettük hozzá a webes frontendet.

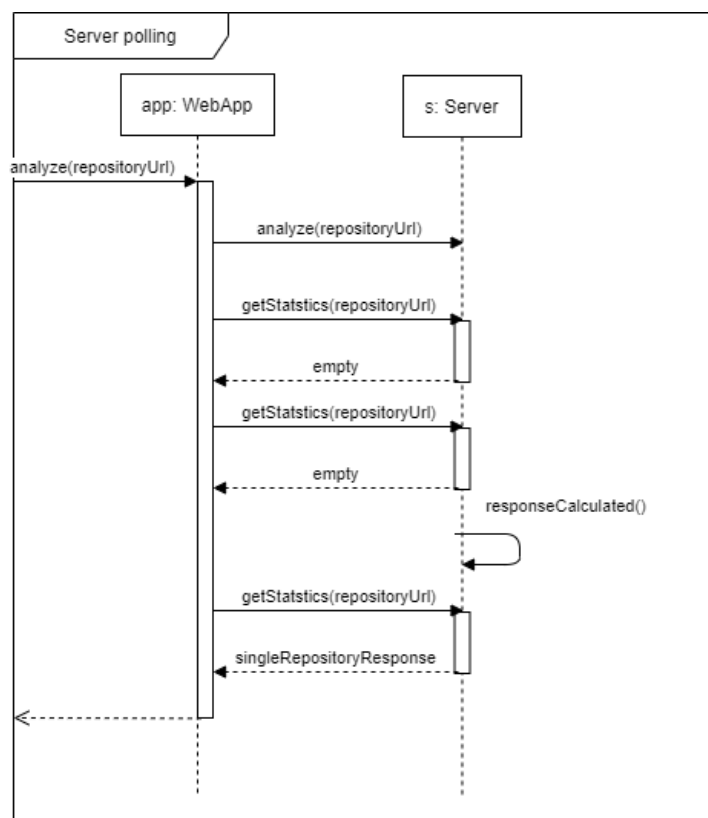
4.1. Felhasznált architektúrális minták

Az alkalmazás fejlesztése során a következőkben részletezett architektúrális mintákat használtuk a konzulensünk javaslatára.

4.1.1. Frontend pollozása

Abból következően, hogy egy tároló statisztikai elemzése, összes commitjának bejárása egy nagyobb kódbázis esetén több időbe is telhet, úgy döntöttünk, hogy a *frontenden* nem fogunk várakozni arra, hogy a szerver befejezze a bejárást. Egy ilyen rosszul implementált várakozás rossz felhasználói élményt tud nyújtani.

A szerver *pollozását* az alábbi szekvencia diagram szemlélteti.



4.1. ábra: Polling folyamata szekvencia diagramon ábrázolva

Amikor a felhasználó egy tároló elemzését kéri, a webalkalmazás küld egy aszinkron kérést a szervernek az adott URL-hez tartozó tároló elemzésére. A *frontend* ezután nem vár a szerver válaszára, hanem bizonyos időközönként lekérdezi tőle, hogy meg van-e már a szükséges adat. Az ilyen kéréseknek a feldolgozása rövid időbe telik és gyorsan visszatérnek üresen

vagy a válasszal. Amint a webalkalmazás megkapta a választ, befejezi a szerver *pollozását*, majd kirajzolja a kapott adatokat a megfelelő diagrammal.

4.1.2. Szerver válaszána feldarabolása

A megvalósítás elején úgy gondoltuk, hogy az egyedüli-, illetve a két tároló elemzéséhez egy-egy darab választ fog a rendszer visszaadni, mely magába foglalja az összes adott oldalhoz tartozó statisztika adatait. Ezt a nagy válasz objektumot a *frontend* fogadja és osztja szét a megfelelő diagramoknak. Ezzel ellentétben később kiderült, hogy az egyik részfeladat választ sokkal több időbe telik kiszámítani, mint a többi alfeladat eredményeit. Ebből kifolyólag úgy döntöttünk, hogy a frontenden lévő *Angular* diagram kirajzoló komponensek csak a saját kiszámított statisztikájukat kérik le a szervertől, ami pedig készen áll erre azáltal, hogy biztosítja, hogy az egyes válaszok külön-külön is lekérdezhetőek.

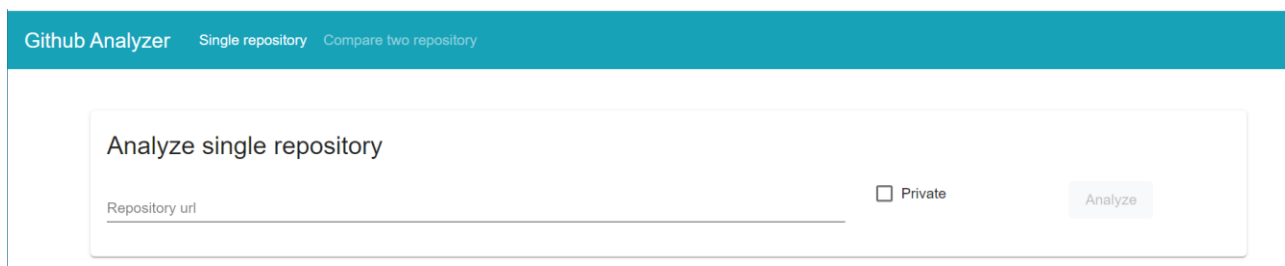
4.1.3. Szerver válaszána gyorsítótárba töltése

Ahogy az már korábbi fejezetekben többször is említettük, egyes tárolók adatainak lekérdezése és statisztikáiknak kiértékelése jelentős futási idő növekedést eredményez bizonyos esetekben. A felhasználói élmény javítása és az erőforrásszükségletek csökkentése érdekében bevezetésre került egy átmeneti memória alapú URL nyilvántartás, amely bizonyos időkereteken belül megőrzi az egyszer már kiszámolt *repository*-ra jellemző statisztikákat.

Egy, már korábban elemzett kódbázis újbóli lekérdezése esetén ebből a *cache* memóriából tölti be a számolások értékeit, ezáltal jelentősen lerövidítve a futási időt. A fent említett időkeret meghatározása fontos feladat, ugyanis egy nagy aktivitást mutató kódbázis esetén a túl hosszú keret pontatlan statisztikákat eredményezhet, a túl rövid pedig felesleges erőforrásallokációt kíván.

4.3. A megvalósított rendszer

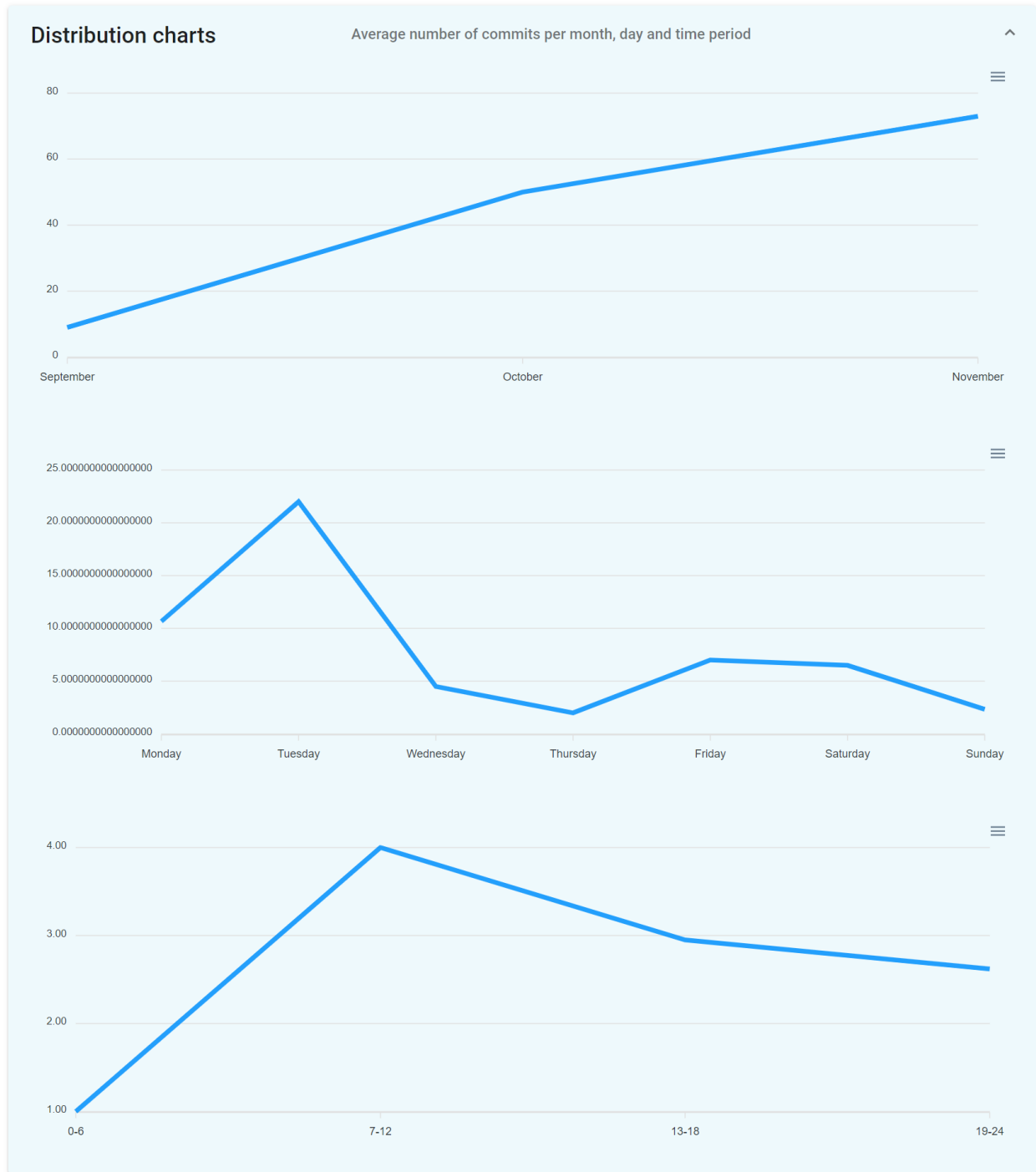
Ebben a fejezetben a kész implementáció bemutatása következik működés közben készített ábrákkal illusztrálva. A rendszer indítása után a *landing page* felület fogadja a felhasználót, amely alapbeállítás szerint a különálló, egyetlen *repository* elemzésének lapjára mutat.



4.2. ábra: Landing page és menüsor

1. Az elemezni kívánt kódbázis adatainak megadása után 5 különböző diagrammon kerülnek a statisztikák megjelenítésre:

- *Distribution chart*: a repository commitjainak időbeli eloszlása hónapokra, napokra és napokon belül több napszakra átlagolva.



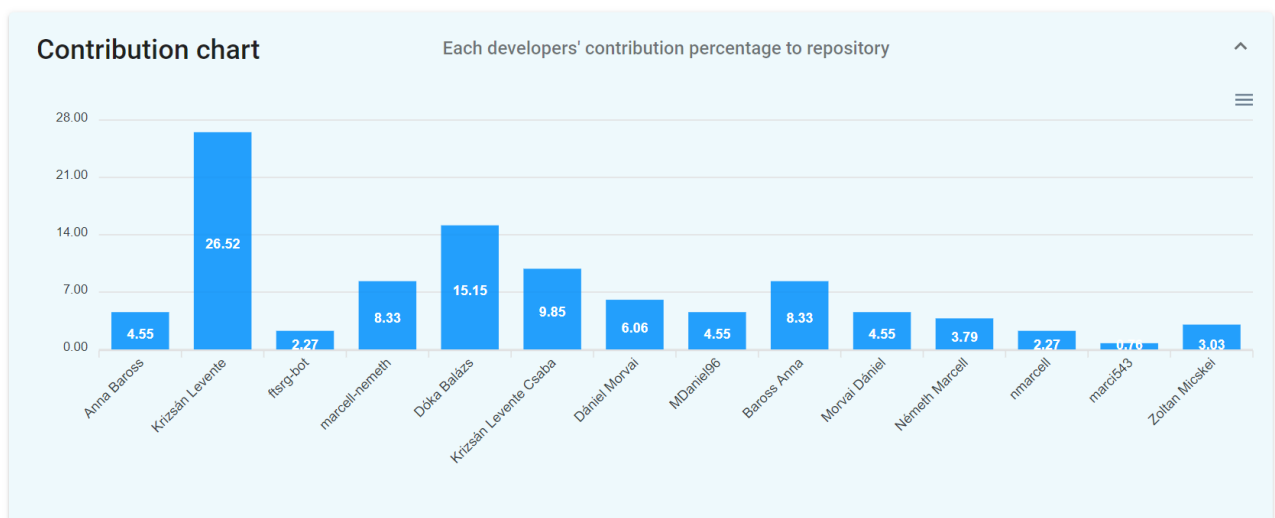
4.3. ábra: Kommitok időbeli eloszlása

- *Modification chart*: egy adott, a repository fennállása óta eltelt időszakból kiválasztott év/hónap kód törléseinek és hozzáadásainak aránya.



4.4. ábra: Kód hozzáadások és törlések aránya egy kiválasztott időszakban

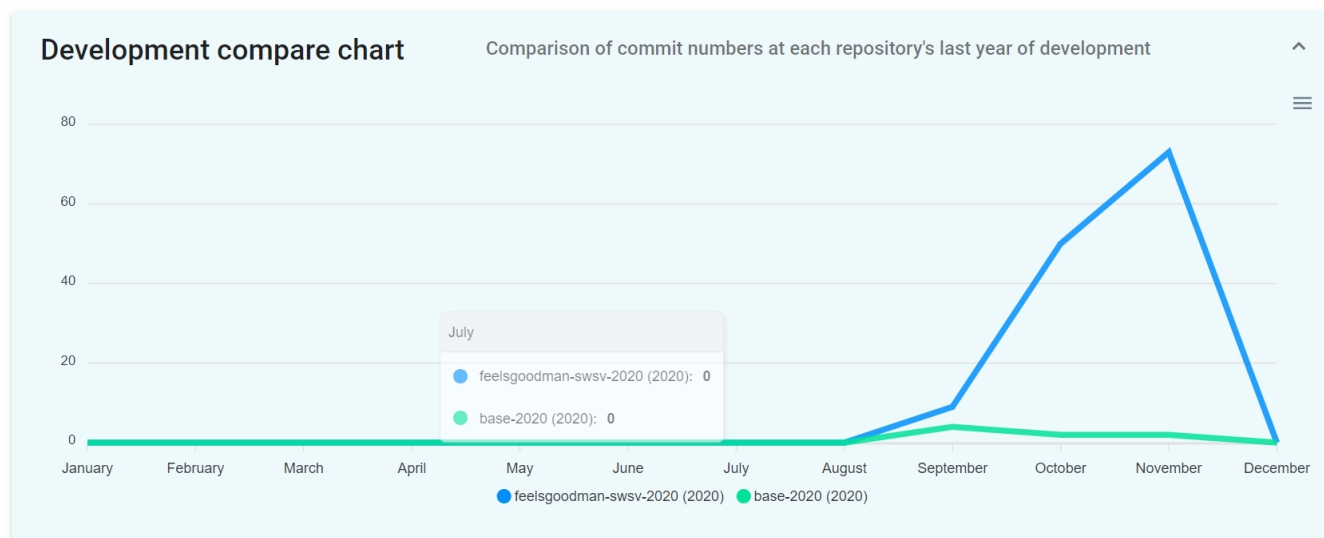
- *Contribution chart*: normalizált hozzájárulás a *repository*-hoz az összes fejlesztőre vetítve.



4.5. ábra: Fejlesztők százalékos hozzájárulása a tárolóhoz

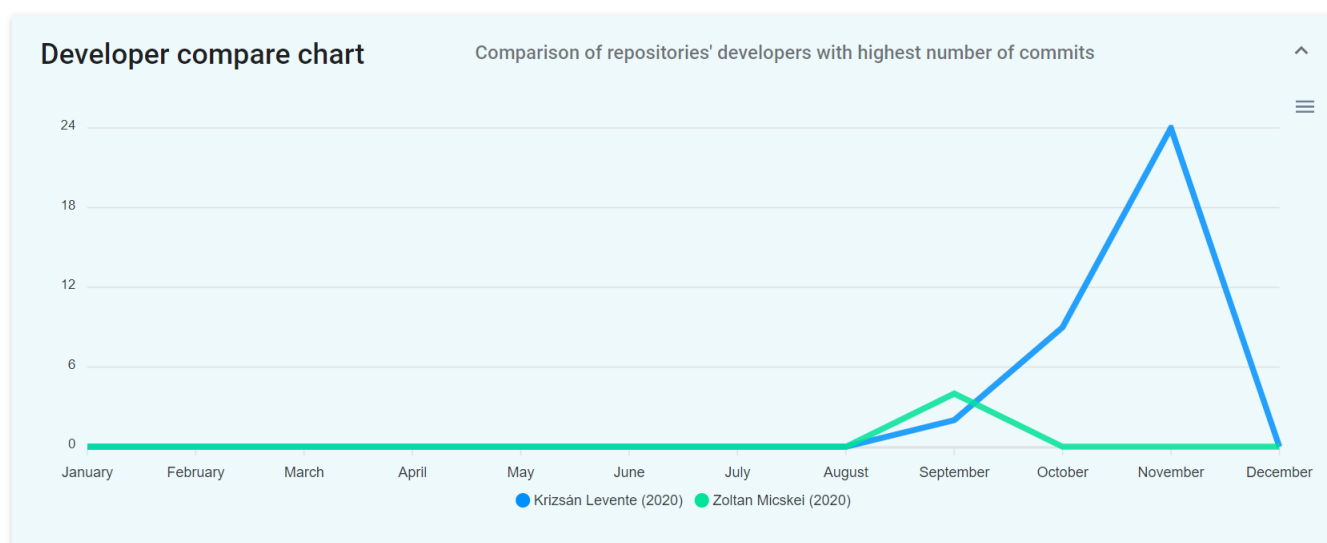
2. A menüsor segítségével a két különböző *repository* összehasonlítására szolgáló fülön tekinthetők meg a közös diagrammokon ábrázolt fejlesztői aktivitások.

- *Development compare chart*: a két összehasonlított kódbázis fejlődésének üteme



4.6. ábra: Összehasonlított tárolók fejlődése

- *Developer compare chart*: a két összehasonlított kódbázis legtöbbet kommitoló fejlesztőjének aktivitása



4.7. ábra: Összehasonlított tárolók legaktívabb fejlesztői

5. Telepítési leírás

Az alkalmazás futtatásához az alábbi eszközök szükségesek:

- JDK 11+
- Maven 3.2+
- Npm
- Angular CLI
- Oracle Sql Developer

Az adatbázis létrehozása:

- *Sql Developer* segítségével lokális adatbázis létrehozása *admin/admin* felhasználónév és jelszóval
- A *db* mappában található *create.sql* fájl futtatása

A szerver futtatása:

- Navigálás a *github-analyzer-server* mappa gyökerébe
- Alkalmazás buildelése: *mvn install*
- Alkalmazás futtatása: *mvn spring-boot:run*

A web alkalmazás futtatása:

- Navigálás a *github-analyzer-app* mappa gyökerébe
- Függőségek letöltése: *npm install*
- Alkalmazás futtatása: *ng serve*

Ezek után az alkalmazás elérhető a böngészőből a <http://localhost:4200/> címen.

6. Felhasznált eszközök

A fejlesztés során a következő eszközöket használtuk a munka megkönnyítése érdekében:

- *IntelliJ Idea Ultimate*: programozói környezet a *front/backend* kódolásához
- *GitHub*: a párhuzamos fejlesztői munka és verziókövetés eszköze
- *draw.io*: diagrammok, ábrák készítése a dokumentációhoz
- *Microsoft Word*: dokumentációk készítéséhez
- *Google Drive*: tárhely a projekt állományainak tárolására

7. Összefoglalás

A féléves munka során megterveztük, implementáltuk, illetve dokumentáltuk a GitHub Analyzer nevű statisztikai kódrendszer elemző rendszert. Az elkészített alkalmazás segítségével GitHub repository-k számszerűsített statisztikai aktivitás adatait tudjuk elemezni és vizualizálni.

A megvalósított alkalmazás háromrétegű architektúrát használ: adatbázis réteg, üzleti logikai réteg és felhasználói felület. Az alkalmazás a GitHub API segítségével lekérdezett adatokat egy adatbázis tárolja. A grafikus megjelenítést az Angular keretrendszer felhasználásával, platformfüggetlen megvalósítással oldottuk meg.

Munkánk során részletes terveket készítettünk – részük jelen dokumentum tartalmát képezik – és jelentős mennyiségű implementációs munkát is végeztünk. Ennek eredményeképpen egy jól működő és megbízható alkalmazást készítettünk el, amely az elvárt alapvető igényeknek megfelel, feladatát képes ellátni.

8. Továbbfejlesztési lehetőségek

A rendszer készítése során folyamatosan dokumentáltuk a potenciális továbbfejlesztési lehetőségeket, amelyeket későbbi félévek során lehetőség szerint meg is valósítunk.

Néhány ilyen feljegyzett ötlet:

- További statisztikák bevezetése: a cél az, hogy egy minél teljesebb körben működő üzleti intelligencia rendszer készüljön el, amely hatékonyan, számszerű adatokkal alátámasztva lenne képes a projektek fejlődését meggyorsítani a „gyenge pontok” felkutatásával.
- Mesterséges intelligenciát használva jövőbeli statisztikák, trendek kimutatása
- Felhasználók kezelése, így nem kell többszöri access token megadása, időzített elemzések futtatása emailben kiküldve, stb.
- További Github funkciók integrálása, adott commitok megtekintése, repository-k szerkesztése.

A fejlesztés folyamán is odafigyeltünk ezekre a továbbfejlesztési irányokra és igyekeztünk olyan tervezői döntéseket hozni, amelyek segítik a program fejlesztésének folytatását.

9. Hivatkozások

- [1] Spring Boot
<https://spring.io/projects/spring-boot>

- [2] Angular
<https://angular.io/>

- [3] Sql Developer
<https://www.oracle.com/database/technologies/appdev/sqldeveloper-landing.html>

- [4] Github API
<https://developer.github.com/v3/>

- [5] Kohsuke Github API
<https://github-api.kohsuke.org/>