SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1462

SUSTAV ZA PRAĆENJE STANJA POSLUŽITELJA TEMELJEN NA JEZIKU GRAPHQL

Dominik Dejanović

Zagreb, lipanj, 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1462

Pristupnik:	**Dominik Dejanović (0036541578)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	izv. prof. dr. sc. Ivana Bosnić

Zadatak: **Sustav za praćenje stanja poslužitelja temeljen na jeziku GraphQL**

Opis zadatka:

Proučiti postojeće aplikacije za praćenje stanja poslužitelja s operacijskim sustavom Linux. Istražiti metode pristupa svojstvima računalnog i operacijskog sustava na nižoj razini, poput praćenja svojstava procesora, memorije, tvrdih diskova, stanja procesa i slično. Osmisliti i razviti sustav za udaljeno praćenje stanja poslužitelja koji s praćenim poslužiteljima komunicira putem aplikacijskih programskih sučelja (API), a za brzo dohvaćanje i pretragu veće količine podataka o stanju poslužitelja koristi jezik GraphQL. Sustav treba podržati pregled stanja više poslužitelja te omogućiti poruke upozorenja u slučaju problema na poslužitelju. U sklopu sustava potrebno je razviti i web-aplikaciju za vizualizaciju prikupljenih podataka u stvarnom vremenu i povijesnih podataka, koja primjenom responzivnog dizajna treba biti prikladna za uporabu i na uređajima manjih zaslona. Komentirati razvijeno rješenje.

Rok za predaju rada: 14. lipnja 2024.

Zahvaljujem mentorici izv. prof. dr. sc. Ivani Bosnić na stpljenju, savjetima i brzim odgovorima tijekom izrade rada.

Sadržaj

1. Uvod

Današnja tehnologija iznimno se oslanja na poslužitelje kako bi ostvarila razne funkcionalnosti koje su potrebne ljudima u svakodnevnom životu, od pretraživanja raznih web-stranica i videa na internetu do povezanosti brojnih računala u virtualno superračunalo.

Nažalost, nije dovoljno konfigurirati poslužitelja da obavlja određene zadaće i nadati se da će raditi zauvijek. Zbog raznih problema kao prirodna katastrofa, pogrešaka u kôdu, virusa i hakera, neispravnog rada komponenti, prevelikog broja zahtjeva i slično, može doći do usporenja poslužitelja, neispravnog obavljanja funkcionalnosti te čak i do potpunog prestanka rada poslužitelja.

Zbog tih razloga postoji puno aplikacija koje se koriste za praćenje rada poslužitelja i obavještavanje administratora u slučaju neispravnog rada. Problem koji se javlja kod većine ovakvih aplikacija je nemogućnost pregleda specifičnog perioda u kojemu se dogodila greška, potreba za plaćanjem naprednijih funkcionalnosti aplikacije te pohranjivanje podataka samo za zadnjih nekoliko dana ili tjedana.

Cilj ove aplikacije je omogućiti praćenje jednog ili više poslužitelja kroz neograničen period vremena, što omogućuje administratorima pregledavanje podataka o radu poslužitelja tijekom specifičnog perioda vremena u kojemu je nastala greška na njemu.

2. Specifikacija zahtjeva

2.1. Korisnički zahtjevi

Osnovna funkcionalnost rješenja je praćenje stanja poslužitelja te prikaz tih podataka na web-stranici. To uključuje pregled:

- *podataka o procesoru* - prikazuje se kartica s općenitim podacima o procesoru te grafovi iskorištenosti procesora i broja procesa za odabrani period
- *podataka o radnoj memoriji* - prikazuje se kartica s grafom na kojem su vidljivi podaci o raznim aspektima radne memorije kao iskorištenost memorije, iskorištenost *swap* particije te postotak priručne memorije
- *podataka o trajnoj pohrani* - za svaki disk prikazuju se općeniti podaci o disku kao naziv, veličina, proizvođač te se prikazuje graf na kojemu je vidljiva iskorištenost particija diska za odabrani period vremena
- *obavijesti i upozorenja* - klikom na "Alerts" otvara se tablica s prikazom obavijesti i upozorenja koje je programsko sučelje generiralo prilikom upisa podataka u bazu podataka (na primjer upozorenje za visoku iskorištenost particije tvrdog diska)

Svaku karticu moguće je minimizirati, nakon čega se one dinamično rasporede da stanu na zaslon (stranica je kompatibilna i za mobilne uređaje).
Također je moguće uvećati dio grafa nakon čega se šalje novi zahtjev na programsko sučelje kako bi se prikazalo više detalja za novi period. Nakon uvećanja grafa, moguće ga je vratiti na početne postavke čime se on umanjuje na originalni period vremena te se šalje novi zahtjev na programsko sučelje za dohvat podataka za taj period.
Korisnik također može odabrati globalni period: nakon promijene početnog ili krajnjeg datuma se svi podaci za odabrani poslužitelj osvježavaju za taj period. Prilikom počet-

nog otvaranja web-stranice, početni datum se postavlja na tjedan dana prije trenutnog vremena, a krajnji datum je neograničen.

2.2. Funkcionalni zahtjevi

Za pohranu podataka o poslužiteljima koristi se baza podataka.
Programsko sučelje procesira zahtjeve za pisanje i čitanje podataka o poslužiteljima te komunicira s bazom podataka. Sučelje ima razne parametre kojima se omogućava filtriranje podataka za određeni poslužitelj i period te parametre koji određuju način kompresije podataka (min/max/average). Aplikacije za praćenje stanja poslužitelja šalje podatke koji se pohranjuju u bazu podataka, a web-stranica šalje zahtjeve za čitanje podataka o poslužiteljima.
Aplikacija za praćenje stanja poslužitelja pokreće se na poslužitelju koji se prati. Moguće je aplikaciju pokrenuti na više poslužitelja. Također je moguće konfigurirati interval slanja podataka te odabrati koji se podaci šalju pomoću JSON konfiguracijske datoteke.
Svakom poslužitelju koji se prati dodjeljuje se jedinstveni identifikator čime se podaci poslani od više poslužitelja razlikuju. Nakon pokretanja aplikacije, šalju se podaci na centralni poslužitelj na kojem je pokrenuto programsko sučelje.
Web-stranica šalje zahtjeve za čitanje podataka na programsko sučelje te pomoću vraćenih podataka omogućava njihov strukturiran pregled.

3. Korištene tehnologije

U ovom projektu je korišten RDBMS, web razvojna okolina, GraphQL te brojne druge tehnologije, čiji opis slijedi u nastavku.

3.1. Linux

Linux je open-source operacijski sustav temeljen na Unixu koji je nastao 1991. godine. Postoje razne distribucije linuxa (Ubuntu, Mint, Arch, Fedora, CentOS i druge) koje uključuju jezgru te razne aplikacije i modifikacijama koje čine tu distribuciju jedinstvenom. Odabran je operacijski sustav *Linux* jer je poznat po svojoj stabilnosti, sigurnosti i fleksibilnosti, što ga čini vrlo popularnom opcijom za poslužitelje. Njegova otvorenost omogućava korisnicima i programerima da slobodno modificiraju i dijele kôd, ali unatoč tome su sve distribucije temeljene na istoj jezgri što omogućava ovom programu da ispravno radi na svim modernim *Linux* distribucijama. Koriste se razni linux programi za prikupljanje podataka kao:

- *lscpu* - podaci o procesoru
- *top* - podaci o trenutno pokrenutim procesima
- *systemctl* - podaci o određenom servisu kao trenutni status, lokacija i poruke
- *lsblk* - podaci o diskovima i particijama na računalu
- *journalctl* - poruke koje je određeni servis poslao

Za programiranje i testiranje programa korištene su *Arch* i *Mint* distribucije, ali program bi trebao raditi na svim *Linux* distribucijama, dokle god se na njih mogu instalirati potrebni programi za prikupljanje podataka i pokretanje aplikacija.

3.2. GraphQL

GraphQL je jezik za upite podataka, hibrid REST programskog sučelja i SQL jezika. Napravljen je da omogući klijentima da definiraju podatke koji su im potrebni, čime se izbjegava dohvaćanje previše ili premalo podataka što su česti problemi kod tradicionalnih REST programskih sučelja.
Odabran je GraphQL umjesto REST programskog sučelja zbog raznih karakteristika GraphQL-a, neke od važnijih su:

- *deklarativni podaci* - korisnici navode točno koji podaci im trebaju te se ne šalje ništa više od toga
- *jedan URL* - koristi se samo jedan URL za sve upite, od upita za dohvaćanje podataka do onih za slanje podataka, što znatno ubrzava razvoj programskog sučelja te olakšava njegovo održavanje
- *ugrađena validacija polja* - ovo omogućava GraphQL-u provjeru polja koja korisnik unosi tako da se ne treba ručno programirati provjeravanje polja (na primjer GraphQL će automatski izbaciti grešku ako se u brojčano polje unese znakovni niz). Također podcrtava pogreške prilikom korištenja nepoznatih parametara i polja:

```
# INVALID: hero is not a scalar, so fields are needed
{
  hero
}
```

```
{
  "errors": [
    {
      "message": "Field \"hero\" of type \"Character
         \" must have a selection of subfields. Did
         you mean \"hero { ... }\"?",
      "locations": [
        {
          "line": 3,
          "column": 3
        }
      ]
    }
  ]
}
```

Slika 3.1. Primjer krivog GraphQL upita [1]

- *upiti slični SQL-u* - za razliku od REST-a koji se oslanja na putanje, GraphQL ko-

risti *Query* objekt kako bi korisnik mogao pomoću određenih parametara filtrirati podatke te odabrati koje podatke želi dohvatiti

- *dohvaćanje više podataka u jednom zahtjevu* - koristeći REST, korisnik bi morao za dohvaćanje raznih podataka slati puno upita, dok se u GraphQL-u može dohvatiti proizvoljan broj nepovezanih podataka u jednom zahtjevu:

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

Slika 3.2. Primjer GraphQL upita [2]

- fleksibilnost razvoja programskog sučelja - stari upiti će raditi čak i ako se shema programskog sučelja promijeni, dokle god polja koja korisnik dohvaća još uvijek postoje

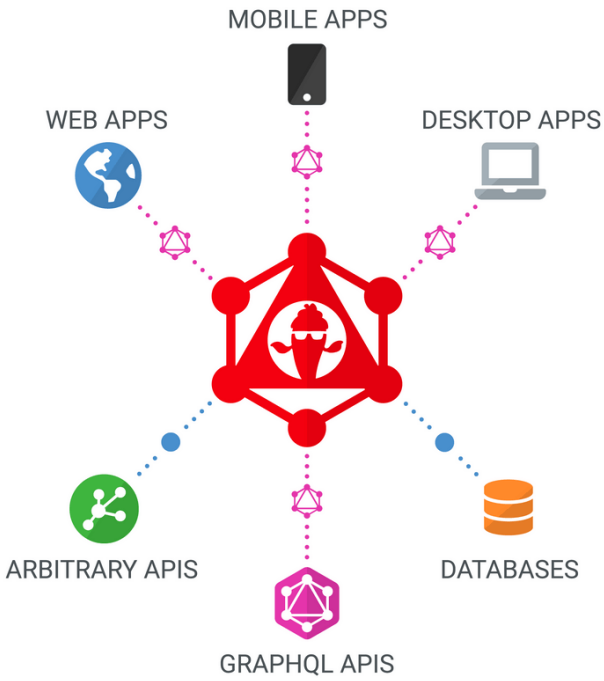
GraphQL se sastoji od četiri važna dijela:

- shema (*schema*) - glavni dio GraphQL sustava koji definira vrste podataka te upite koje korisnici mogu izvršavati
- upit (*query*) - omogućava slanje upita poslužitelju u kojem se definiraju podaci koji se vraćaju
- mutacija (*mutation*) - omogućava slanje podataka poslužitelju koji se koriste za dodavanje, izmjenu te brisanje podataka
- pretplata (*subscription*) - omogućava osvježavanje podataka u stvarnom vremenu

Navedene karakteristike GraphQL-a omogućavaju efikasno prenošenje velike količine podataka te smanjuju kompleksnost održavanja i dokumentiranja kôda, zbog čega je on odabran umjesto REST programskog sučelja.

3.3. HotChocolate

HotChocolate je C# biblioteka koja se koristi za izgradnju GraphQL poslužitelja. Ona omogućava korištenje svih funkcionalnosti GraphQL tehnologije pomoću .NET razvojne okoline bez da ih moramo ručno implementirati. Postoje razne druge biblioteke koje se koriste za interakciju s GraphQL poslužiteljem, no ova biblioteka je odabrana zbog opširne dokumentacije i podrške za moderne GraphQL funkcionalnosti.
Korištena verzija: 13.6.0



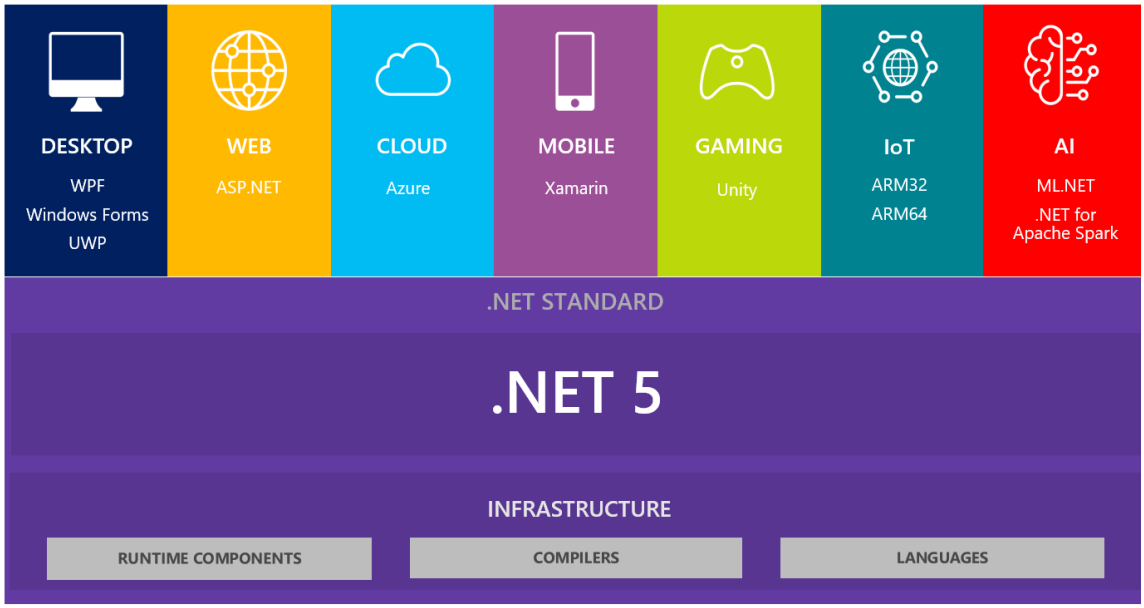
Slika 3.3. HotChocolate server [3]

3.4. .NET i C#

.NET je otvorena platforma za razvoj raznih tipova aplikacija (web-aplikacije, aplikacije za mobilne uređaje, video igre te drugo). Obuhvaća razne tehnologije i alate koji omogućavaju brzi razvoj aplikacija. Ključne komponente su *.NET Runtime* (zadužen za automatsko skupljanje smeća i upravljanje memorijom), *.NET SDK* (alati i biblioteke koje se

koriste za razvoj .NET aplikacija, alati za ispravljanje pogrešaka) i *.NET biblioteke* (osnovne funkcionalnosti potrebne za razvoj aplikacija).
.NET okruženje podržava C#, F# i VB.NET jezike.
Korištena .NET verzija: net8.0

C# je objektno orijentiran programski jezik dizajniran 2000. godine. Sintaksa C# jezika je jednostavna te jezik omogućava izgradnju paralelnog kôda što znatno ubrzava aplikacije. Često je uspoređivan s programskim jezikom Java.
Odabrane su ove tehnologije zbog jednostavnosti razvoja aplikacija u njima te zbog toga što .NET podržava izvođenje i razvoj aplikacija na više platformi, uključujući Windows i Linux operacijske sustave.
Korištena C# verzija: 12.0



Slika 3.4. Alati .NET platforme [4]

linq2db

linq2db je .NET biblioteka koja se koristi za pretvaranje LINQ kôda u SQL kôd kako bi se moglo lako pristupati bazi podataka.
Nakon instalacije biblioteke se pomoću određenih naredbi može spojiti na bazu podataka i iz nje generirati C# klase koje odgovaraju tablicama u bazi podataka:

```
using System;
using LinqToDB.Mapping;

[Table("Products")]
public class Product
{
  [PrimaryKey, Identity]
  public int ProductID { get; set; }

  [Column("ProductName"), NotNull]
  public string Name { get; set; }

  [Column]
  public int VendorID { get; set; }

  [Association(ThisKey = nameof(VendorID), OtherKey=
     nameof(Vendor.ID))]
  public Vendor Vendor { get; set; }

  // ... other columns ...
}
```

Slika 3.5. Primjer linq2db kôda [5]

Newtonsoft.Json

Newtonsoft.Json je C# biblioteka koja se koristi za serijalizaciju i deserijalizaciju JSON podataka.

3.5. Vue

Vue je Javascript razvojno okruženje koje se koristi za izradu korisničkog sučelja. Omogućava jednostavniji rad s prikazom podataka i bolje strukturiranje kôda od samog Javascript jezika.
Neke od prednosti Vue razvojnog okruženja:

- *progresivna arhitektura* - može se lako integrirati u postojeće projekte te se nakon toga postepeno proširivati
- *komponente* - koriste se komponente kako bi se aplikacija razdvojila na manje, ponovno upotrebljive dijelove
- *reaktivnost* - prikazani podaci se automatski osvježavaju kada se promijene varijable za koje su podaci vezani

- *direktive* - Vue pruža direktive kao v-model, v-if, v-for te druge kako bi se jednostavno manipuliralo DOM-om

Primjer Vue kôda za povećanje gumba:

```
<div id="app">
<button @click="count++">
Count is: {{ count }}
</button>
</div>
```

Slika 3.6. Primjer Vue kôda [7]

Korištena verzija: 3.2.13

primevue

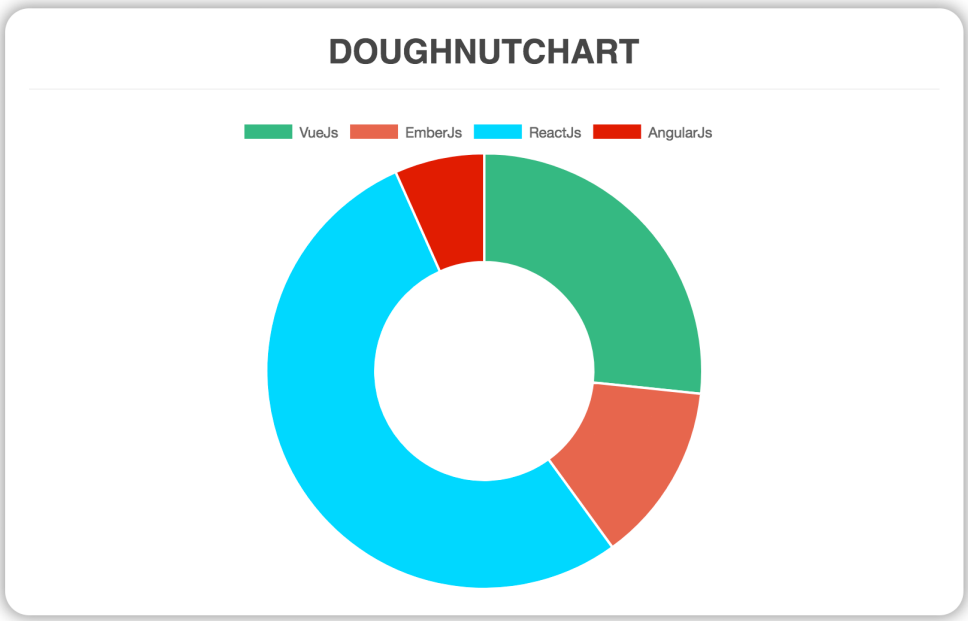
Primevue biblioteka sadržava razne Vue komponente koje se koriste za dinamički prikaz podataka na ekranu (prilikom promijene podataka se odmah mijenja i prikaz bez potrebe za dodatnim kôdom). Koristi se i *primeicons* biblioteka koja omogućuje korištenje raznih ikona.

Korištena *primevue* verzija: 3.52.0

Korištena *primeicons* verzija: 7.0.0

vue-chartjs

Vue-chartjs je biblioteka koja se koristi za vizualizaciju podataka ovisno o predanim parametrima. Omogućuje prikaz raznih tipova grafova kao linijski, stupčasti te točkasti graf koje se može konfigurirati kroz parametre komponente.



Slika 3.7. Primjer chartjs grafa [9]

Korištena verzija: 5.2.0

vue-datepicker

Vue-datepicker je Vue komponenta koja se koristi za odabir datuma i vremena.

Slika 3.8. Primjer datepicker komponente

Korištena verzija: 7.1.0

3.6. PostgreSQL

PostgreSQL relacijski je sistem za upravljanje bazama podataka (RDBMS) koji je otvorenog kôda i besplatan. Poznat je po svojoj stabilnosti i fleksibilnosti.
Važne karakteristike PostgreSQL-a:

- *otvoreni kôd* - PostgreSQL kôd je open-source zbog čega ga bilo tko može vidjeti, izmijeniti i distribuirati
- *moderni standardi* - implementira puno modernih karakteristika SQL jezika

- *podrška za JSON* - omogućava rad s JSON podacima, uključujući njihovu pohranu
- *razni tipovi podataka* - podržava integer, varchar, boolean, array, uuid, xml te razne druge tipove podataka
- *ACID (atomicity, consistency, isolation, durability)* - osigurava pouzdane transakcije i integritet podataka

Odabran je PostgreSQL kao RDBMS sustav za upravljanje bazom podataka zato što je otvorenog kôda te ima opširnu dokumentaciju i korisničku podršku.
Korištena verzija: 16.2

3.7. Git i Github

Git je sustav za upravljanje kôdom. Podržava verzioniranje datoteka što omogućava praćenje promjena kôda te suradnju sa drugim programerima. Napravio ga je Linus Torvalds 2005. godine te je danas jedan od najvažnijih alata za razvoj programa. Neke od važnijih karakteristika:

- *distribuiranost* - svaki korisnik ima kompletnu kopiju repozitorija na svom računalu što omogućava rad na projektu kada korisnik nema pristup internetu
- *verzioniranje datoteka* - pohranjuje stare datoteke datoteka što olakšava pronalazak kôda koji je uzrokovao novu grešku
- *grananje* - omogućava rad na odvojenim dijelovima projekta, neovisno o main/master grani
- *rad u timu* - olakšava suradnju više programera koji svi rade na istom projektu pomoću raznih mehanizama, jedan od kojih je spriječavanje nestanka kôda kojeg je jedan programer objavio s kôdom kojeg drugi programer pokušava objaviti

Github je platforma za razvoj aplikacija temeljena na Git-u. Koristi se za lakše korištenje Git-a te pruža razne alate koji se mogu koristiti za organizaciju zadaća programerima.

4. Opis rješenja

4.1. Programsko rješenje

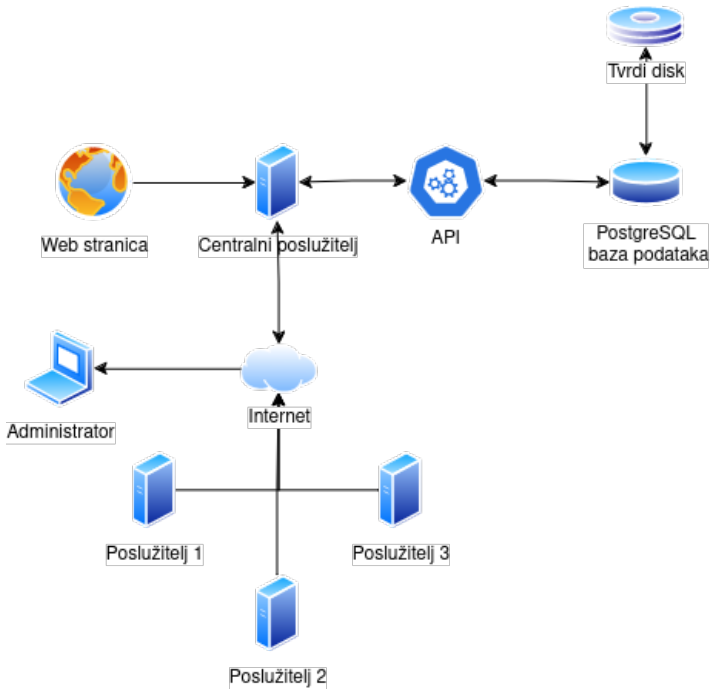
Napravljene su tri .NET projekta (API, Common i Monitor), web stranica te baza podataka.
API (programsko sučelje) je centralna aplikacija koja povezuje *Monitor* i web-stranicu s bazom podataka. To je ostvareno pomoću GraphQL-a, kod kojeg se koristi *Query* objekt za dohvaćanje podatka za web-stranicu te *Mutation* objekti koji se koriste kako bi *Monitor* mogao pisati podatke u bazu podataka.
Monitor je aplikacija koja se pokreće na poslužitelju koji se prati. Ona u specificiranom intervalu prikuplja razne podatke o poslužitelju te ih šalje programskom sučelju koji ih zapisuje u bazu podataka.
Web-stranica se zatim koristi za prikaz prikupljenih podataka o pojedinačnim serverima te za prikaz obavijesti i upozorenja koje je programsko sučelje generiralo prilikom upisa podataka u bazu podataka.

4.2. Topologija

Postoji jedan centralni poslužitelj na kojem su pokrenute dvije aplikacije: *Vue web-stranica* i *programsko sučelje*.

Web-stranica se dohvaća HTTP protokolom, nakon čega ona šalje zahtjeve programskom sučelju na centralnom poslužitelju radi dohvata podataka iz baze podataka.

Poslužitelji koji su konfigurirani za praćenje podataka također komuniciraju s programskim sučeljem, ali ne za dohvaćanje nego za slanje podataka sučelju koji onda te podatke pohranjuje u bazu podataka.



Slika 4.1. Dijagram rješenja

U nastavku slijedi detaljan opis funkcionalnosti i kôda svih projekata.

5. Baza podataka

Glavni direktorij rješenja sadržava datoteku db.sql koja se koristi za stvaranje PostgreSQL baze podataka. U nastavku je opisana struktura baze podataka.

5.1. Poslužitelj

Za praćenje poslužitelja koristi se tablica *server* koja pohranjuje samo ID poslužitelja. Ovu tablicu referencira većina drugih tablica.

Također je napravljen pogled koji se koristi za dohvat statusa poslužitelja. Poslužitelj se smatra aktivnim ako je od zadnjeg slanja podataka prošlo manje od n minuta (n = interval specificiran prilikom zadnjeg slanja podataka).

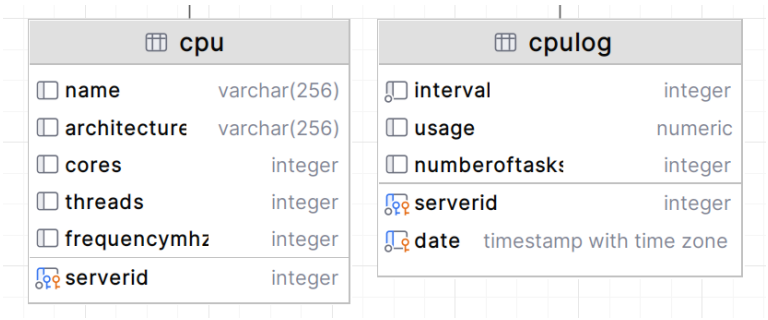
```
CREATE VIEW serverStatus AS
SELECT serverid, (SELECT CASE WHEN COUNT(*) > 0 THEN '
   online' ELSE 'offline' END
                  FROM serverlog
                  WHERE
                       serverlog.serverid = server.serverid
                          AND
                       EXTRACT(EPOCH FROM timezone('utc',
                          now())-serverlog.date) < (
                          serverlog.interval * 60)) as
                          status
FROM server;
```

Slika 5.1. serverStatus pogled

5.2. Procesor

Za praćenje podataka o procesoru koriste se tablice *cpu* i *cpulog*. Tablica *cpu* pohranjuje općenite podatke o procesoru: ime procesora (*name*), arhitektura (*architecture*), broj jezgri (*cores*), broj niti (*threads*), frekvencija (*frequency*) te poslužitelj kojem on pripada (*serverid*).

cpulog tablica pohranjuje podatke o procesoru koji se mijenjaju tijekom vremena kao iskorištenost procesora (*usage*) i broj pokrenutih procesa (*numberoftasks*).

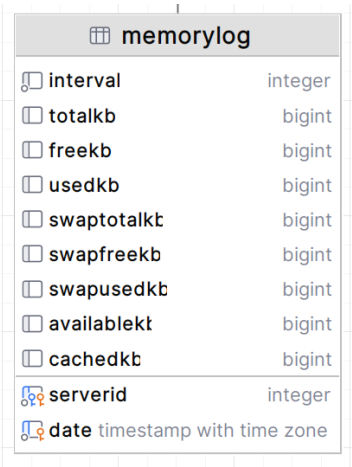


Slika 5.2. Struktura tablica za procesor

5.3. Radna memorija

Za pohranjivanje podataka o radnoj memoriji koristi se tablica *memorylog* koja pohranjuje podatke o datumu prikupljanja podataka (*date*) ukupnoj memoriji (*totalkb, freekb, usedkb*), *swap* memoriji (*swaptotalkb, swapfreekb, swapusedkb*), neiskorištenoj memoriji (*availablekb*), priručnoj memoriji (*cachedkb*) te ID poslužitelja kojem memorija pripada (*serverid*).

Za razliku od procesora, ne koristi se zasebna tablica za pohranu općenitih podataka o priručnoj memoriji kao proizvođač, serijski broj te drugo jer nije pronađen način da se ti podaci očitaju s poslužitelja bez administratorskih privilegija.

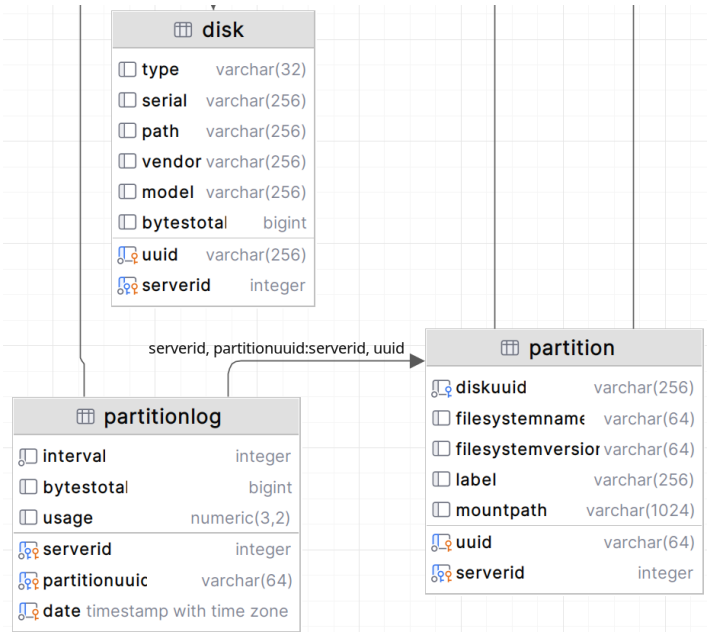


Slika 5.3. Struktura tablice *memorylog*

5.4. Trajna pohrana

Za pohranu trajne memorije koriste se tablice:

- *disk* - općenite informacije o tvrdom disku: jedinstveni ID (*uuid*), tip (*type*), serijski broj (*serial*), putanja na kojoj se disk nalazi (*path*), proizvođač (*vendor*), model, veličina diska (*bytestotal*) i ID poslužitelja na kojem se disk nalazi (*serverid*)
- *partition* - općenite informacije o jednoj particiji: jedinstveni ID (*uuid*), UUID diska kojem pripada (*diskuuid*), tip i verzija datotečnog sustava (*filesystemname, filesystemversion*), naziv particije (*label*), putanja na kojoj se nalazi (*mountpath*) te ID poslužitelja na kojem se particija nalazi (*serverid*)
- *partitionlog* - podaci o particiji koji se mijenjaju tijekom vremena kao veličina particije (*bytestotal*) i iskorištenost (*usage*)



Slika 5.4. Struktura tablica za pohranu

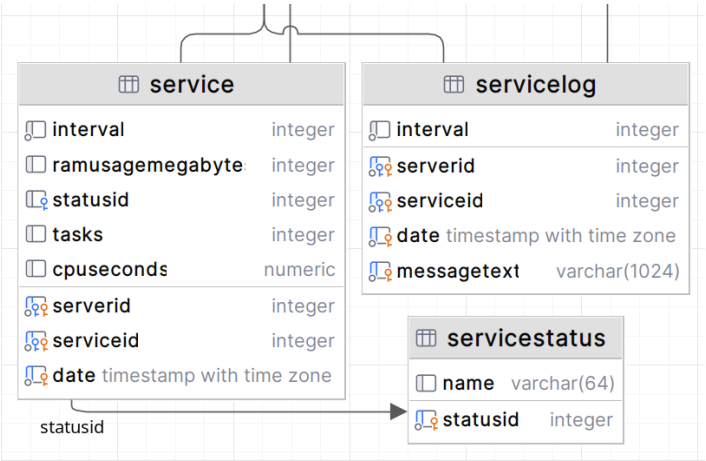
5.5. Servisi

Servisi nisu do kraja implementirani u aplikaciji, no podloga za praćenje podataka o servisima je implementirana u bazi podataka. Za praćenje servisa koriste se tablice:

- *service* - prikupljeni podaci o statusu servisa kao pohrana koju koristi (*ramusage-megabytes*), broj procesa koje je servis stvorio (*tasks*), procesorska snaga koju koristi

(*cpuseconds*) te status servisa (*statusid*)

- *servicename* - povezuje jedinstveni broj s nazivom servisa
- *servicelog* - pohranjuje poruke koje je servis poslao (*messagetext*) te vrijeme kada su poslane (*date*)
- *servicestatus* - povezuje ID statusa servisa (*statusid*) s tekstualnom reprezentacijom (*name*)

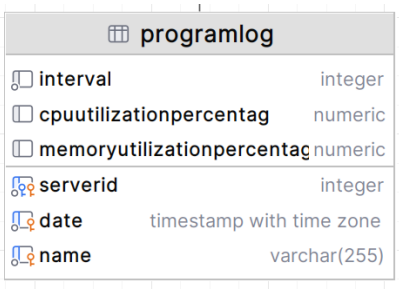


Slika 5.5. Struktura tablica za servise

5.6. Programi

Podška za programe nije do kraja implementirana u aplikaciji, no podloga za njihovo praćenje je implementirana u bazi podataka.

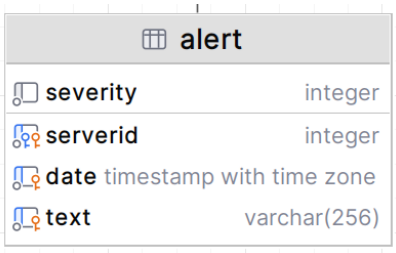
Za praćenje programa koristi se tablica *programlog* koja pohranjuje podatke kao ime programa (*name*) te postotak procesora (*cpuutilizationpercentage*) i memorije (*memoryutilizationpercentage*) koju program koristi.



Slika 5.6. Struktura tablica za programe

5.7. Obavijesti

Stvorena je tablica za obavijesti koja pohranjuje važnost poruke (*severity*), ID poslužitelja za koji je relevantna (*serverid*) te datum (*date*) i tekst poruke (*text*).



Slika 5.7. Struktura tablica za obavijesti

Uz tablicu je također napravljena i metoda *before_alert_insert_func()* koja se poziva prije unosa podataka u tablicu. Metoda je stvorena kako bi se onemogućio unos iste obavijesti ako je ona već poslana u zadnjih sat vremena (na primjer ako se proba unijeti "*Server 0 cpu load above 90%*" u razmaku od 30 minuta, ta poruka će biti zapisana samo prvi put u bazu podataka, a drugi put se odbacuje uz grešku).

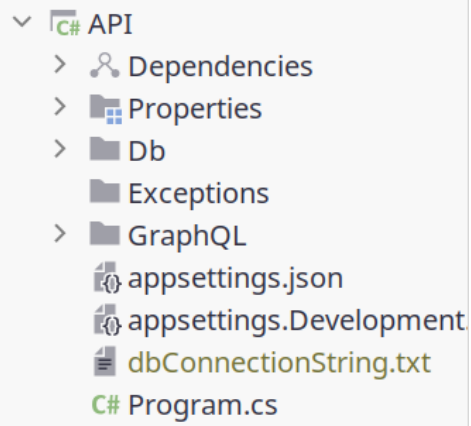
```
    SELECT EXTRACT(EPOCH FROM NEW.date::timestamp-date::
       timestamp)/60 FROM alert WHERE NEW.serverId =
       serverId AND NEW.text LIKE text ORDER BY date desc
       INTO mins;
    IF mins < 60 THEN
        RAISE EXCEPTION 'Same alert already raised less
           then an hour ago (% minutes ago)', mins;
    END IF;
    RETURN NEW;
```

Slika 5.8. Metoda *before_alert_insert_func()*

6. Programsko sučelje

Programsko sučelje je implementirano koristeći *HotChocolate server* biblioteku koja implementira funkcionalnosti GraphQL poslužitelja.

Struktura projekta:



Slika 6.1. Struktura API projekta

6.1. Klasa Program

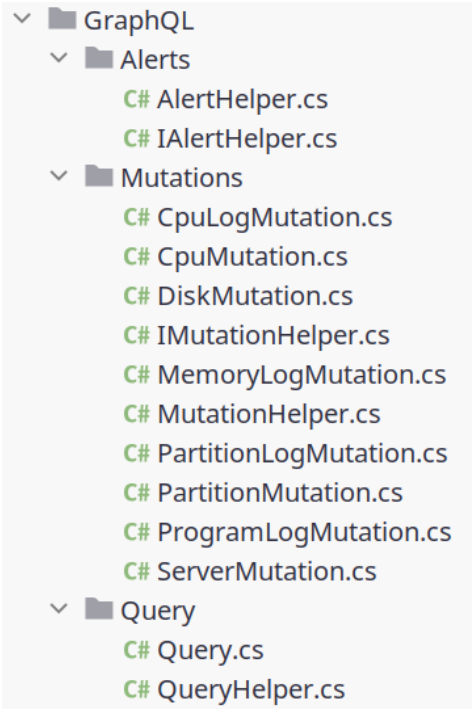
Klasa Program se pokreće prilikom pokretanja programa. Glavne funkcije koje ona izvršava su konfiguracija *Dependency Injectiona* te povezivanje putanja programskog sučelja.

Dependency Injection je način strukturiranja kôda kod kojeg se klase ne stvaraju direktno pomoću kodne riječi "*new*" nego se parametri konstruktora klase automatski predaju u nju. Odabran je ovaj način izrade kôda kako bi se omogućilo jednostavnije testiranje kôda u budućnosti jer se umjesto stvarne klase koja obavlja određenu funkcionalnost može predati lažna klasa koja emulira stvarnu klasu (na primjer kao "*IDb database*" sučelje se umjesto klase koja obavlja funkcionalnosti stvarne baze podataka predaje lažna

klasa koja samo provjerava je li određena metoda pozvana tri puta; ako da onda je test ispravan, a u suprotnom je neispravan).

6.2. Direktorij GraphQL

Direktorij GraphQL sadržava sav kôd potreban za ispravan rad GraphQL poslužitelja kao *Query* i *Mutation* kôdovi te kôd za rad s obavijestima.



Slika 6.2. Struktura GraphQL direktorija

6.2.1. Klasa Query

Koristi se za dohvat podataka iz baze podataka. Za svaki tip podatka (kao procesor, memorija i tvrdi disk) je napravljena metoda koja se poziva kada korisnik šalje zahtjev pomoću kojega se pokušava dohvatiti određeni podatak.

Svaka metoda predstavlja tip *query* objekta koji se pokušava dohvatiti, a svi parametri te metode predstavljaju argumente koje korisnik specificira tijekom slanja zahtjeva.

Neki od parametara koje gotova svaka metoda sadržava:

- *serverId* - dohvaćaju se samo podaci koje je poslao server čiji je ID jednak ovom argumentu

- *startDateTime* - dohvaćaju se samo podaci poslani nakon specificiranog datuma i vremena
- *endDateTime* - dohvaćaju se samo podaci poslani prije specificiranog datuma i vremena

Klasa se oslanja na *IDbProvider* sučelje za rad s bazom podataka te na metodu *GetLogs* za dohvat podataka iz baze podataka. Primjer jedne takve metode:

```
public async Task<CpuOutput?> Cpu(int serverId,
    DateTimeOffset? startDateTime, DateTimeOffset?
    endDateTime, double? interval, string? method)
{
    var getEmptyRecordFunc = () => new CpuLog();
    Func<IList<CpuLogDbRecord>, CpuLog> combineLogsFunc =
        logs =>
    {
        double? usageProcessed = (double?)QueryHelper.
            CombineValues(method, logs.Select(c => c.Usage).
            ToList());
        var numberOfTasksValues = logs.Where(c => c.
            NumberOfTasks != null).Select(c => c.NumberOfTasks
            !);
        int? numberOfTasks = (int?)QueryHelper.CombineValues(
            method, numberOfTasksValues.ToList());
        return new CpuLog { Date = logs.First().Date, Usage =
            usageProcessed, NumberOfTasks = numberOfTasks };
    };

    var cpuOutput = new CpuOutput(serverId);
    await using var db = dbProvider.GetDb();
    {
        var cpu = await (from c in db.Cpus where c.ServerId
            == serverId select c).FirstOrDefaultAsync();
        if (cpu == null) return null;

        cpuOutput.Name = cpu.Name;
        cpuOutput.Architecture = cpu.Architecture;
        cpuOutput.Cores = cpu.Cores;
        cpuOutput.Threads = cpu.Threads;
        cpuOutput.Frequency = cpu.FrequencyMhz;
        var query = from l in db.CpuLogs where l.ServerId ==
            serverId select l;
        await foreach (var log in QueryHelper.GetLogs(query,
            combineLogsFunc, getEmptyRecordFunc, startDateTime
            , endDateTime, interval))
        {
            cpuOutput.Logs.Add(log);
        }
    }
```

Slika 6.3. Metoda *Cpu*

6.2.2. Klasa QueryHelper

Klasa *QueryHelper* je pomoćna klasa koju koristi klasa *Query*. Sastoji se od raznih metoda od kojih je najbitnija metoda *GetLogs* koja sadržava algoritam koji se koristi za spajanje više podataka u jedan.

Algoritam radi tako da pomoću početnog i krajnjeg datuma izračuna interval unutar kojeg se podaci spajaju u jedan:

```
    return (int)((DateTime)endDate).Subtract((DateTime)
       startDate).TotalHours;
```

Slika 6.4. Izračun intervala

Nakon izračuna intervala dohvaćaju se podaci iz tablice koji su pohranjeni nakon specificiranog početnog datuma i prije krajnjeg datuma. Nakon toga se prolazi kroz svaki podatak te se koristi klasa *DatasetHelper* za spajanje više podataka u jedan te se na kraju vraća lista spojenih podataka.

```
  public static async IAsyncEnumerable<TLog> GetLogs<TDbLog
     , TLog>(
    IQueryable<ILog> table,
    Func<IList<TDbLog>, TLog> combineLogsFunc,
    Func<TLog> getEmptyLogFunc,
    DateTimeOffset? startDateTime,
    DateTimeOffset? endDateTime,
    double? interval) where TDbLog : ILog where TLog :
       LogBase
```

Slika 6.5. Parametri metode *GetLogs*

6.2.3. Klasa DatasetHelper

Klasa *DatasetHelper* je pomoćna klasa korištena u algoritmu spajanja više podataka u jedan podatak. Najvažnija metoda u klasi je metoda *AddLog* koja dodaje podatak u listu podataka koja se kasnije vraća. Broj podataka koji se spaja u jedan podatak ovisi o intervalu.

Na primjer ako je interval 30 minuta, a podaci se bilježe svakih 5 minuta, spaja se 6 podataka u jedan. Ako je došlo do prekida u slanju podataka (na primjer podaci se šalju svakih 5 minuta, ali jedan podatak se pošalje nakon 7 minuta), onda se za period između

ta dva podatka vraća "prazan" podatak koji signalizira prekid u slanju podataka. Algoritam radi na ovaj način kako bi se korisnike moglo obavijestiti ako je došlo do prekida slanja podataka.

```
public IEnumerable<TLog?> AddLog(TDbLog log, double?
    interval)
{
  //Combining logs and adding additional logs if there
     has been some kind of a break
  //between the last log and the current one so that the
     charts goes to 0 in case of break
  if (_nextDate != null && log.Date != _nextDate)
  {
    if (_logs.Count != 0) yield return CombineLogs();
    var emptyLog = getEmptyLogFunc();
    emptyLog.Date = _nextDate.Value;
    yield return emptyLog;
    _break = true;
  }

  //Returning a log if a break happened
  if (_break)
  {
    var emptyLog = getEmptyLogFunc();
    emptyLog.Date = log.Date.AddSeconds(-1);
    yield return emptyLog;
    _break = false;
  }

  //Adding log to the list of logs
  _logs.Add(log);
  _nextDate = log.Date.AddMinutes(log.Interval);
  _intervalSum += log.Interval;
  if (_intervalSum < interval) yield break;

  //Returning the combined log
  yield return CombineLogs();
}
```

Slika 6.6. Metoda AddLog

6.2.4. Direktorij Mutations

Direktorij *Mutations* sadržava klase i sučelja koji se koriste za dodavanje i osvježavanje podataka u bazu podataka.

Važnije klase i sučelja u direktoriju:

- *IMutationHelper* - sučelje koje definira funkcionalnosti koje su potrebne za umetanje, osvježavanje i brisanje podataka iz baze podataka

- *MutationHelper* - implementacija *IMutationHelper* sučelja

Primjer dijela jedne mutacije:

```
[ExtendObjectType(OperationType.Mutation)]
public class CpuMutation(IMutationHelper mutationHelper)
{
    private readonly Func<IDb, CpuIdInput, IQueryable<
       CpuDbRecord>> _getCpuQuery = (db, cpu) =>
        from c in db.Cpus
        where c.ServerId == cpu.ServerId
        select c;

    private readonly Func<CpuDbRecord, CpuIdInput>
       _getCpuId = cpu => new CpuIdInput(cpu.ServerId);

    public async Task<Payload<CpuOutputBase>> AddCpu(
       CpuInput cpu)
    {
        var model = InputToDbModel(cpu);
        return await mutationHelper.AddModelAsync<
           CpuIdInput, CpuDbRecord, CpuOutputBase>(model,
           _getCpuId(model), _getCpuQuery);
    }
```

Slika 6.7. Isječak mutacije *CpuMutation*

6.2.5. Sučelje IAlertHelper

Sučelje *IAlertHelper* koristi se za slanje obavijesti bazi podataka. Poruka se odbacuje ako je ista poruka za isti poslužitelj već poslana prije manje od sat vremena.

```
public interface IAlertHelper
{
  Task RaiseAlert(int serverId, DateTimeOffset date,
     AlertSeverity severity, string text);
}
```

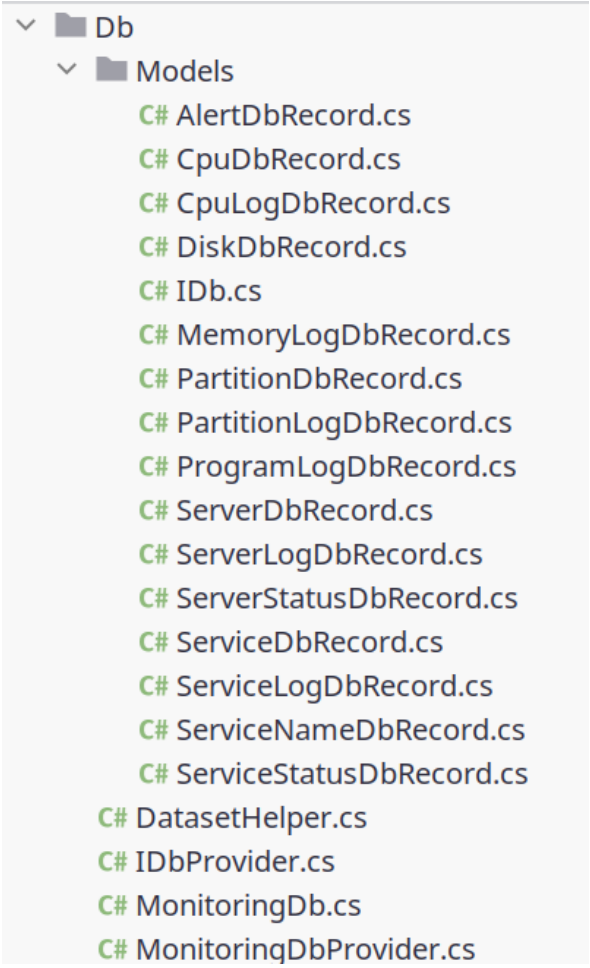
Slika 6.8. Sučelje *IAlertHelper*

6.3. Direktorij Db

Direktorij *Db* sadržava sve pomoćne klase i modele koji se koriste za interakciju s bazom podataka. Većina klasa u direktoriju *Models* su automatski generirane (i djelomično ručno promijenjene) korištenjem *linq2db* paketa pomoću naredbe "*dotnet linq2db scaf-*

fold -p PostgreSQL -c "Host=localhost; Username=[username];

Password=[password];Database=[database]""



Slika 6.9. Struktura direktorija *Db*

Neke od važnijih klasa i sučelja direktorija:

- *sučelje IDb* - sučelje koje sadržava popis svih tablica u bazi podataka

```
public interface IDb : IDataContext
{
  public ITable<CpuDbRecord> Cpus { get; }
  public ITable<CpuLogDbRecord> CpuLogs { get; }
  public ITable<DiskDbRecord> Disks { get; }
  public ITable<MemoryLogDbRecord> MemoryLogs { get; }
  public ITable<PartitionDbRecord> Partitions { get; }
  public ITable<PartitionLogDbRecord> PartitionLogs { get;
     }
  public ITable<ProgramLogDbRecord> ProgramLogs { get; }
  public ITable<ServiceDbRecord> Services { get; }
  public ITable<ServiceLogDbRecord> ServiceLogs { get; }
  public ITable<ServicenameDbRecord> ServiceNames { get; }
  public ITable<ServicestatusDbRecord> ServiceStatuses {
     get; }
  public ITable<AlertDbRecord> Alerts { get; }
  public ITable<ServerDbRecord> Servers { get; }
  public ITable<ServerLogDbRecord> ServerLogs { get; }
  public ITable<ServerStatusDbRecord> ServerStatus { get; }
}
```

Slika 6.10. Sučelje *IDb*

- *klasa MonitoringDb* - automatski generirana implementacija IDb.cs sučelja
- *klasa MonitoringDbProvider* - koristi se za generiranje nove veze s bazom podataka (potrebno jer se koristi *Dependency Injection*)

6.4. Konfiguracijske datoteke

U programu se nalaze tri konfiguracijske datoteke: *appsettings.json*, *appsettings.Development.json* i *dbConnectionString.txt* od kojih je samo zadnja namijenjena za konfiguraciju poslužitelja.

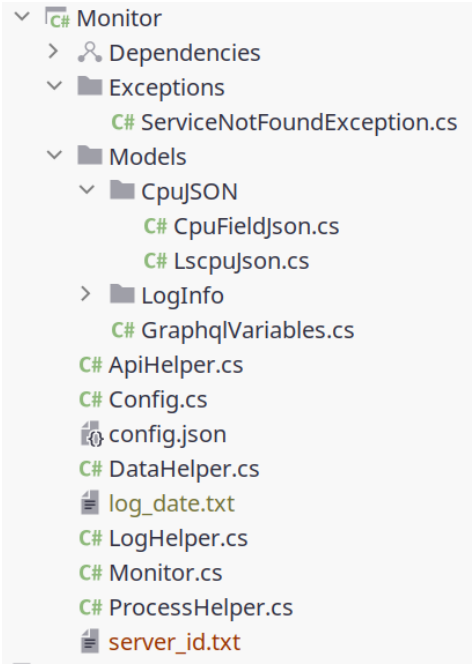
Datoteka *dbConnectionString.txt* sadržava niz znakova koji se koristi za spajanje na postojeću bazu podataka prilikom pokretanja programa:

```
Host=localhost;Username=[username];Password=[password];
   Database=[dbName];Include Error Detail=[true for
   debugging; false for deployment]
```

Slika 6.11. Primjer datoteke *dbConnectionString.txt*

7. Projekt Monitor

Aplikacija *Monitor* koristi se za prikupljanje i slanje podataka o poslužitelju programskom sučelju.



Slika 7.1. Struktura projekta *Monitor*

Objašnjenje važnijih dijelova aplikacije:

- *klasa Monitor* - sadržava metodu *Main* koja se pokreće prilikom pokretanja aplikacije. Na početku učitava konfiguracijsku datoteku i zatim generira nasumični broj za ID poslužitelja. Na kraju se u beskonačnoj petlji pokreće proces prikupljanja podataka te slanje tih podataka programskom sučelju

```
private static async Task Main()
{
  var config = JsonSerializer.Deserialize<Config>(await
     File.ReadAllTextAsync("config.json"));
  if (config == null) throw new Exception("Configuration
     invalid!");

  //Creating server ID if the program is running for the
     first time
  if (File.Exists(ServerIdFilename) == false)
  {
    await File.WriteAllTextAsync(ServerIdFilename,
       DateTimeOffset.UtcNow.GetHashCode().ToString());
  }

  int serverId = int.Parse(await File.ReadAllTextAsync(
     ServerIdFilename));
  var logHelper = new LogHelper(config,
     LastLogDateFilename);
  var apiHelper = new ApiHelper(config, serverId);

  while (true)
  {
    var log = await logHelper.Log();
    await apiHelper.SendLog(log);
  }
}
```

Slika 7.2. Glavna metoda programa Monitor

- *direktorij Models* - sadržava klase koje se koriste za serijalizaciju i deserijalizaciju podataka dohvaćenih s terminala
- *klasa ApiHelper* - koristi se za slanje podataka programskom sučelju. Prilikom poziva metode *SendLog* radi se GraphQL zahtjev koristeći prikupljene podatke te se šalje zahtjev programskom sučelju

```
        var request = new GraphQLRequest(requestString,
            variables: variables);

        try
        {
            //Sending the request and printing out result/
                errors
            var graphQlClient = new GraphQLHttpClient(
                _config.ApiUrl, new NewtonsoftJsonSerializer
                ());
            var payload = await graphQlClient.
                SendMutationAsync<Payload<CpuLogInput>>(
                request);
            if (payload.Errors != null && payload.Errors.
                Length != 0) throw new Exception(payload.
                Errors[0].Message);
            if (payload.Data.Error != null) Console.
                WriteLine($"ERROR: {payload.Data.Error}");
            Console.WriteLine(payload.Data.Data);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error in sending POST
                request to server: {ex.Message}");
        }
```

Slika 7.3. Isječak metode *SendLog*

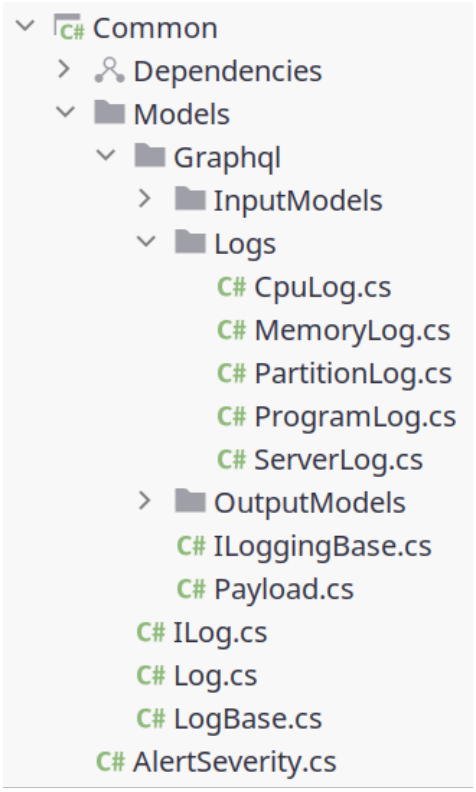
- *klasa DataHelper* - čita podatke s terminala te ih deserijalizira u klase koje se nalaze u direktoriju *Models*
- *klasa LogHelper* - pokreće proces dohvaćanja podataka svakin [n] minuta (n = definiran u datoteci *config.json*)
- *klasa ProcessHelper* - koristi se za pokretanje procesa u terminalu
- *datoteka config.json* - koristi ju administrator kako bi konfigurirao aplikaciju

```
    var process = new Process();

    //Checking if the program can be found on the system
    process.StartInfo.FileName = "which";
    process.StartInfo.Arguments = programPath;
    process.StartInfo.RedirectStandardOutput = true;
    process.Start();
    string path = await process.StandardOutput.
       ReadToEndAsync();
    if (string.IsNullOrEmpty(path)) throw new Exception($"
       Program {programPath} not found on the system!");
    await process.WaitForExitAsync();

    //Starting the process
    process = new Process();
    process.StartInfo.FileName = programPath;
    process.StartInfo.RedirectStandardOutput = true;
    process.StartInfo.RedirectStandardError = true;
    process.StartInfo.Arguments = arguments;
    process.Start();
    return process;
```

Slika 7.4. Isječak metode *StartProcess*

8. Projekt Common

Projekt *Common* sadržava klase i sučelja koje koriste više projekata. Stvoren je kako bi se izbjeglo ponavljanje kôda.



Slika 8.1. Struktura Common projekta

8.1. Sučelje ILog

Glavno sučelje je *ILog* i svaka *Log* klasa nasljeđuje polja definirana u njemu.

```
public interface ILog
{
  DateTimeOffset Date { get; }
  int Interval { get; }
}
```

Slika 8.2. Sučelje *ILog*

8.2. Direktorij Graphql

Direktorij *Graphql* sadržava klase i sučelja koji se koriste za komunikaciju *API* i *Monitor* programa.

Neke od važnijih komponenti direktorija:

- klasa *Payload* - sastoji se od *Data* polja koje sadržava podatke o poslužitelju koji se vraćaju korisniku te *Error* polja koje je prazno osim u slučaju pogreške prilikom obrade zahtjeva
- *direktorij InputModels* - sadržava klase koje se koriste prilikom dodavanja ili izmjene podataka. Primjer klase:

```
public class CpuInput : CpuIdInput
{
  public string? Name { get; set; }
  public string? Architecture { get; set; }
  public int? Cores { get; set; }
  public int? Threads { get; set; }
  public int? FrequencyMhz { get; set; }

  public CpuInput(int serverId) : base(serverId)
  {
  }
}
```

Slika 8.3. Klasa *CpuInput*

- *direktorij Logs* - sadržava klase koje opisuju *Log* podatke za određeni aspekt poslužitelja
- *direktorij OutputModels* - sadržava klase koje se koriste prilikom vraćanja odgovora programskog sučelja. Primjer klase:

```
public class AlertOutput(int serverId, DateTimeOffset date,
    AlertSeverity severity, string text)
{
  public int ServerId { get; set; } = serverId;
  public DateTimeOffset Date { get; set; } = date;
  public AlertSeverity Severity { get; set; } = severity;
  public string Text { get; set; } = text;
}
```

Slika 8.4. Klasa *AlertOutput*

9. Web-stranica

Web-stranica je napravljena pomoću *Vue* razvojne okoline. Sastoji se od raznih komponenti koje se koriste za prikaz i dohvaćanje podataka od programskog sučelja. Koristi se *Vue* razvojna okolina te se zbog toga upotrebljava samo jedna HTML stranica koja se nalazi u direktoriju "*public*". Stranica se koristi kao kostur web-stranice u kojemu se prikazuju renderirane komponente.

9.1. Komponenta App

Komponenta *App* je glavna Vue komponenta koja kontrolira prikazani sadržaj stvara se prilikom pokretanja programa.

Na početku komponente nalazi se glavni izbornik koji koristi *Vue router* komponentu koja renderira određenu komponentu, ovisno o tome na kojoj putanji se korisnik nalazi:

```
<Menubar :model="menu_items">
  <template #item="{item, props, hasSubmenu}">

    <!--creating a router link in case the item has a
        route-->
    <router-link v-if="item.route" v-slot="{ href,
       navigate }" :to="item.route" custom>
      <a :href="href" v-bind="props.action" @click="
         navigate">
        <span :class="item.icon" style="margin-right:
            10px;" />
        <span class="ml-2">{{ item.label }}</span>
      </a>
    </router-link>

    <!--creating a normal item in case the item has
       no route-->
    <a v-else :href="item.url" :target="item.target"
       v-bind="props.action">
      <span :class="item.icon" style="margin-right:
         10px;" />
      <span class="ml-2">{{ item.label }}</span>
      <span v-if="hasSubmenu" class="pi pi-fw pi-
         angle-down ml-2" />
    </a>

  </template>
</Menubar>
```

Slika 9.1. Kôd za prikaz izbornika [14]

Prilikom stvaranja komponente šalje se upit programskom sučelju kako bi se dohvatila i popunila lista poslužitelja, a zatim se korisnik prosljeđuje na putanju prikaza podataka prvog poslužitelja (osim ako nisu zabilježeni podaci za bilo koji poslužitelj).

9.2. Vue komponente

Vue razvojna okolina koristi komponente kako bi se dijelovi kôda mogli ponovno koristiti. Sve korištene komponente nalaze se u direktoriju *components*.

Komponenta Chart

Komponenta *Chart* koristi se za prikaz linijskog grafa.

Parametri komponente:

- *name* - naslov grafa

- *chartData* - točke grafa
- *scales* - konfiguracijsko polje za x i y osi

Koristi nekoliko pomoćnih polja za ispravan rad, od kojih je najvažnije *options* koje omogućava uvećanje grafa, postavlja x os kao vremensku os i y os kao brojčanu os te pokreće događaj *emitZoomChanged*.

Komponenta koristi *emitZoomChanged* kako bi roditelju javila da je graf uvećan, što je potrebno kako bi roditelj mogao osvježiti podatke za određeni period i poslati ih komponenti.

Ostale komponente

Komponente *CpuInfo*, *DiskInfo* i *MemoryInfo* koriste se za prikaz podataka o procesoru i pohrani. Komponente rade na sličan način:

- komponenti se mogu predati parametri *startDate* (početni datum grafa), *endDate* (krajnji datum grafa) i *serverId* (ID poslužitelja na kojeg se komponenta odnosi)
- nakon kreiranja komponente učitavaju se podaci pomoću klase *Api* te se ti podaci pretvaraju u podatke povoljne za prikaz grafom pomoću klase *ChartHelper*
- podaci se prikazuju pomoću komponente *Fieldset* na čijem početku se nalaze generalni podaci o komponenti te graf (ili grafovi) ispod toga
- nakon primanja događaja *zoomChanged* od grafa podaci za taj graf se ponovno učitavaju za novi period

Primjer dijela jedne takve komponente:

```
<template>
  <Fieldset legend="Memory" :toggleable="true">
    <Chart
      name="Memory"
      :scales="{ x: { type: 'time' }, y: { min: 0, max: 100
          }}"
      :chart-data="this.$data.memoryChartData"
      @zoom-changed="async (limits) => {
      $data.memoryChartConfig.startDate = limits.startDate
         ?? $props.startDate
      $data.memoryChartConfig.endDate = limits.endDate ??
         $props.endDate
      await this.refreshData($data.memoryChartConfig.
         startDate, $data.memoryChartConfig.endDate)
    }"
    />
  </Fieldset>
</template>
```

Slika 9.2. Isječak *MemoryInfo* komponente

9.3. Direktorij models

Direktorij *Models* sadržava razne klase koje se koriste za pohranjivanje podataka dohvaćenih s programskog sučelja.

Primjer modela:

```
export class Partition {
  uuid: string
  label: string
  filesystemName: string
  filesystemVersion: string
  mountPath: string
  logs: PartitionLog[]

  constructor(uuid: string, label: string, filesystemName:
     string, filesystemVersion: string, mountPath: string,
     logs: PartitionLog[]){
    this.uuid = uuid
    this.label = label
    this.filesystemName = filesystemName
    this.filesystemVersion = filesystemVersion
    this.mountPath = mountPath
    this.logs = logs
  }
}
```

Slika 9.3. Model particije

9.4. Klasa Api

Klasa *Api* zadužena je za komunikaciju s programskim sučeljem.

Primjer dohvaćanja podataka o poslužiteljima:

```
  static async getServers(){
    let queryString = `
    query {
      server {
        serverId
        online
      }
    }`

    let response = await this.executeQuery(queryString)
    if (response?.data?.server == null) return []
    let servers: Server[] = []
    response.data.server.forEach((s: any) => servers.push(s
       ))
    return servers
  }
}
```

Slika 9.4. Dohvaćanje podataka o poslužiteljima

9.5. Klasa ChartHelper

Klasa *CharHelper* pomoćna je datoteka koja se koristi za pretvorbu podataka koje programsko sučelje vraća u točke na grafu. Metode ove klase prolaze kroz parametar *logs* te za svaki podatak vraćaju podatke relevantne za graf.

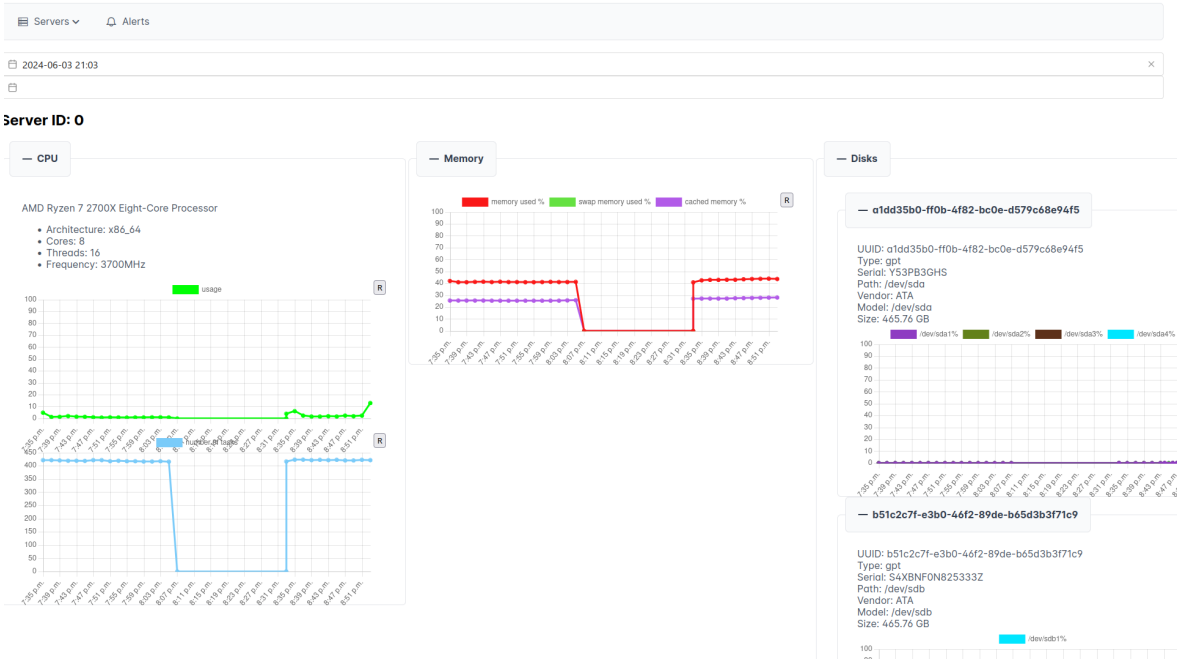
Primjer metode koja vraća točke za graf particije:

```
  static PartitionLogsToDataset(partitionLabel: string,
     color: string, logs: PartitionLog[]){
    if (logs == null || logs.length === 0) return {datasets
       : []}

    let usagePoints: object[] = []
    logs.forEach(l => {
      usagePoints.push({
        x: l.date,
        y: l.usedPercentage == null ? 0 : l.usedPercentage
           * 100
      })
    })

    return {
      showLine: true,
      label: partitionLabel + "%",
      borderColor: color,
      backgroundColor: color,
      data: usagePoints
    }
  }
}
```

Slika 9.5. Vraćanje točaka grafa za podatke particije

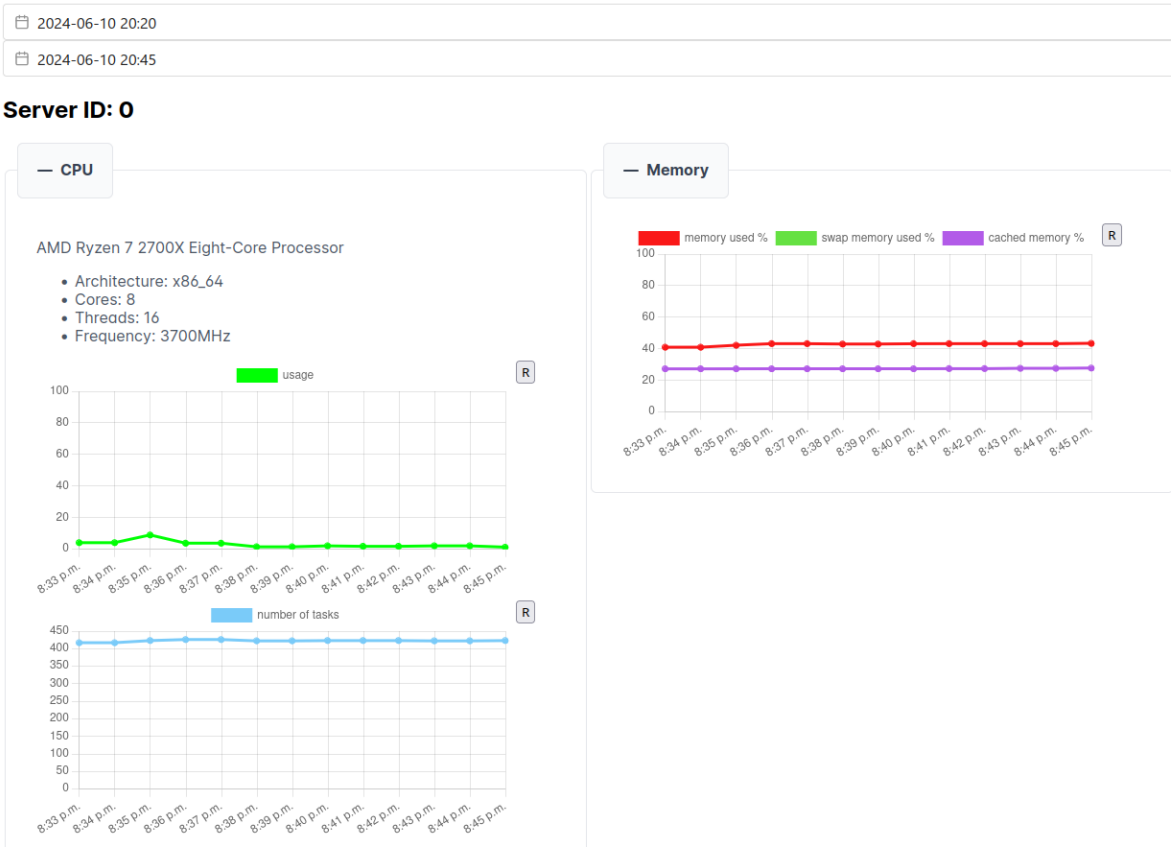
10. Korisničko sučelje i interakcija

Otvaranjem web-stranice prikazuje se glavna stranica na kojoj se nalazi izbornik s listom poslužitelja i karticom za obavijesti. Ako barem jedan poslužitelj postoji, prikazuju se podaci o prvom poslužitelju.



Slika 10.1. Početna stranica

Moguće je odabrati početni i krajnji datum, čijim se mijenjanjem osvježavaju svi podaci web-stranice.



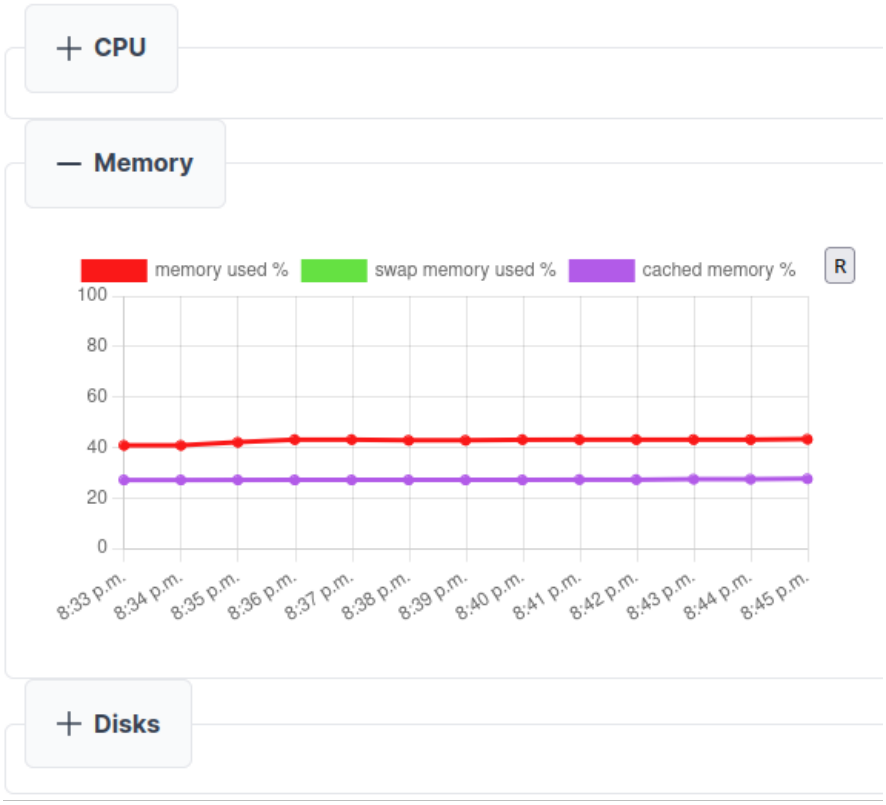
Slika 10.2. Početna stranica sa filtriranim datumom

Prilikom učitavanja podataka za graf prikazuje se poruka "*Loading...*", a u slučaju da podataka nema prikazuje se "*No data (try choosing a different date range)*"

Pojedinačne grafove moguće je uvećati čime se osvježavaju podaci. Graf je također moguće vratiti na početne postavke čime se osvježavaju podaci za originalni period.

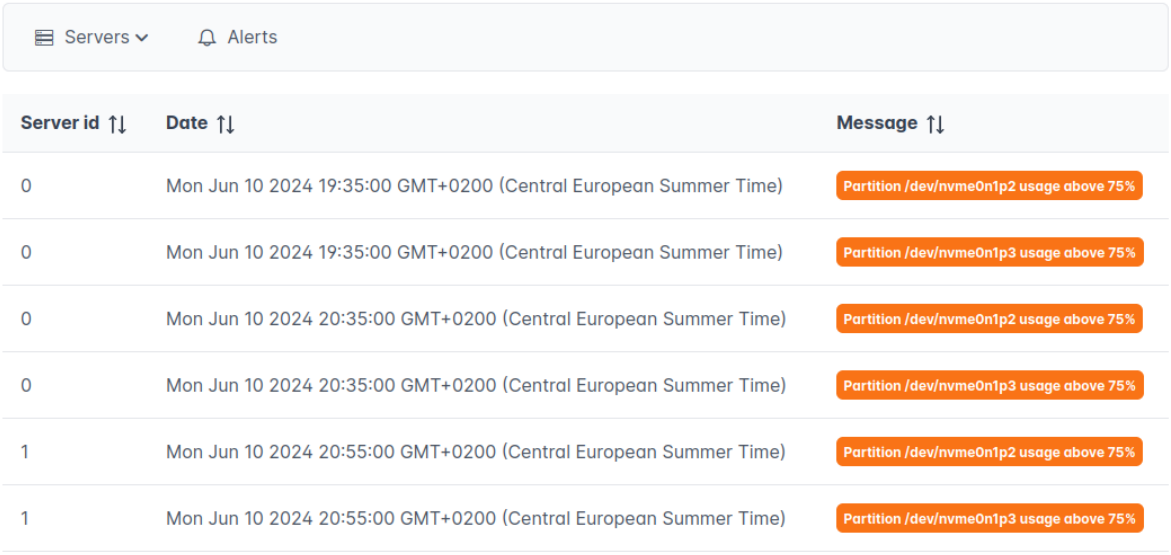
Slika 10.3. Primjer uvećanja grafa *Memory*

Kartice komponenti moguće je smanjiti čime se mijenja izgled stranice (proširuju se komponente kako bi stale na ekran).



Slika 10.4. Izgled web-stranice nakon umanjenja kartica

Klikom na karticu *Alerts* prikazuju se obavijesti i upozorenja.



Slika 10.5. Prikaz obavijesti i upozorenja

Web-stranica podržava široke ekrane i mobilne uređaje.

Slika 10.6. Web-stranica na mobilnim uređajima

11. Komentari

U trenutnom obliku, napravljeno rješenje nije bolje od raznih drugih rješenja za praćenje statusa poslužitelja. Svako od tih rješenja nudi neka poboljšanja od sigurnosti, optimizacije, bolje administracije pozlužitelja, osvježavanja podataka u stvarnom vremenu te brojne druge funkcionalnosti i poboljšanja.
Uz dovoljno vremena i truda moguće je implementirati te funkcionalnosti i brojne druge koje bi učinile ovaj alat vrlo korisnim za administraciju i praćenje poslužitelja. U ovom poglavlju bit će opisani razni aspekti aplikacije koji se mogu poboljšati ili dodati.

11.1. Sigurnost

Web sučelje i programsko sučelje nemaju implementiranu autentifikaciju što omogućava bilo kome da pristupi njima ako može pristupiti web mreži i vratima na kojeima se aplikacije izvršavaju. Ovo predstavlja veliki sigurnosni propust, pogotovo za velike mreže poslužiteljima o kojima bi se mogli pročitati povjerljivi podaci i dodati neispravni podaci u bazu podataka. Kako bi se ovo spriječilo potrebno je dodati određenu metodu autentifikacije kao OAuth2, čime bi se pristup omogućio samo korisnicima s ispravnom lozinkom ili tokenom.

11.2. Brzina rada

PostgreSQL baza podataka omogućava indekse koji nisu korišteni u ovom rješenju. Indeksi se koriste radi brzog pretraživanja podataka po određenom polju (na primjer po datumu) što može znatno ubrzati aplikaciju, pogotovo ako se nakupila velika količina podataka o više poslužitelja.

Također, prilikom dohvaćanja podataka iz programskog sučelja uvijek se učitavaju

svi podaci, bez obzira želi li ih korisnik dohvatiti. Ovo znatno usporava aplikaciju jer je najviše vremena potrebno za čitanje i procesiranje velike količine podataka. Potrebno je naći način da se iz baze podataka pročitaju samo podaci koje korisnik želi dohvatiti.

11.3. Administracija poslužitelja

Nakon konfiguracije i pokretanje *Monitor* aplikacije na poslužitelju, moguće je vidjeti podatke o tom poslužitelju na web-stranici, ali nije moguće preko programskog sučelja ili web-stranice mijenjati ikakve postavke poslužitelja. Rad s većom količinom poslužitelja bi znatno olakšala implementacija sučelja za administraciju unutar web-aplikacije preko kojeg bi se moglo zaustaviti slanje podataka od određenog poslužitelja te konfigurirati obavijesti i upozorenja koje poslužitelj šalje, interval u kojem se šalju podaci te razne druge postavke.

Implementacija ugrađenog *ssh terminala* bi također znatno olakšala rad s poslužiteljima jer bi se na jednom mjestu moglo iz daljine upravljati poslužiteljima i izvršavati naredbe na njima.

11.4. Više kategorija podataka

Trenutno se prate samo podaci o procesoru te trajnoj i radnoj memoriji. Djelomično je implementirana podrška za praćenje programa i servisa, no ona bi se trebala dovršiti.

Također se mogu dodati podaci o poslužitelju (vrijeme rada, status, obavijesti te drugo), mreži (IP adrese, praćenje stanja registriranih domena, graf slanja i primanja podataka) te razni drugi podaci koji bi bili korisni administratorima poslužitelja radi otkrivanja grešaka.

11.5. Mail obavijesti

Obavijesti su trenutno vidljive samo na web-stranici. Ovo je korisno za pregled velike količine obavijesti, ali je problem u tome što administratori moraju često posjećivati web-stranicu kako bi vidjeli je li došla nova obavijest. Moglo bi se osim praćenja obavijesti na web-stranici dodati opcija za slanje podatka na adresu e-pošte administratora kako bi

mogao na vrijeme vidjeti obavijest i popraviti grešku ako je došlo do problema s radom poslužitelja.

11.6. Osvježavanje podataka na web-stranici

Web-stranica omogućava pregled podataka za određeni period. Korisnik može polje za krajnji datum ostaviti prazno kako bi se dohvatili svi podaci do zadnje poslanih podataka, no nakon dohvaćanja tih podataka oni se neće osvježiti dolaskom novih podataka. GraphQL omogućava korištenje *subscription* komponente kako bi se korisnik mogao prijaviti na određeni događaj i dobiti nove podatke kada se oni pošalju. Implementacija *subscripiton* komponente bi omogućila osvježavanje podataka u stvarnom vremenu.

11.7. Podrška za Windows

.NET aplikacije se mogu izvršavati na Linux i na Windows operacijskom sustavu što omogućava ovom rješenju da se koristi i za praćenje podataka o Windows poslužiteljima uz određene promijene. Najveći problem podržavanja više operacijskih sustava je drugačije dohvaćanje podataka o računalu (na Linux operacijskom sustavu koriste se određeni alati kao *lsblk* koje Windows ne podržava) zbog čega bi se trebali koristiti drugi alati i drugačije čitanje i procesiranje podataka.

Jedno od potencijalnih rješenja je instalacija "*Windows Subsystem for Linux*" rješenja koje bi omogućilo korištenje Linux alata na Windows operacijskom sustavu, no trebalo bi testirati ovo rješenje jer je moguće da određeni alati ne bi radili ili bi neispravno radili na Windowsu.

Sažetak

Sustav za praćenje stanja poslužitelja temeljen na jeziku GraphQL

Dominik Dejanović

U sklopu ovog završnog rada implementirana je skupina programa koji omogućuju praćenje i pregled stanja raznih aspekata poslužitelja. Aplikacija za praćenje stanja poslužitelja i programsko sučelje napravljeni su koristeći ASP .NET Core, a za izradu web-aplikacije koristi se Vue razvojna okolina. Baza podataka implementirana je u PostgreSQL sustavu.

Aplikacija za praćenje stanja poslužitelja periodički šalje podatke programskom sučelju, koje ih zatim zapisuje u bazu podataka. Web-stranica slanjem zahtjeva programskom sučelju može dohvatiti i prikazivati podatke o procesoru, radnoj memoriji, diskovima i obavijestima za određeni poslužitelj.

Ključne riječi: Vue.js; ASP .NET Core; web-aplikacija; Linux; GraphQL; C#; poslužitelj

Abstract

GraphQL-based server monitoring system

Dominik Dejanović

As a part of this final thesis, multiple programs have been made which enable the monitoring of various aspects of a server. The monitoring application and the API have been made using ASP .NET Core, and the website has been made using the Vue framework. The database has been made using PostgreSQL RDBMS.

The monitoring application periodically sends data to the API which then writes them to the database. The website can then get and view the stored data about the processor, RAM, hard disks and alerts for a specific server by sending queries to the API.

Keywords: Vue.js; ASP .NET Core; website; Linux; GraphQL; C#; server

Privitak A: Literatura

[1] GraphQL validacija polja, https://graphql.org/learn/validation, 11.6.2024.

[2] GraphQL query, https://graphql.org/learn/queries, 11.6.2024.

[3] HotChocolate slika, https://chillicream.com/docs/hotchocolate/v13, 11.6.2024.

[4] .NET slika, https://auth0.com/blog/what-is-dotnet-platform-overview, 11.6.2024.

[5] linq2db dokumentacija, https://github.com/linq2db/linq2db, 11.6.2024.

[6] , Newtonsoft.Json repozitorij, https://github.com/JamesNK/Newtonsoft.Json, 11.6.2024.

[7] , Vue uvodna dokumentacija, https://vuejs.org/guide/introduction.html, 11.6.2024.

[8] , primevue dokumentacija, https://primevue.org, 11.6.2024.

[9] , chartjs repozitorij, https://raw.githubusercontent.com/apertureless/vue-chartjs, 11.6.2024.

[10] vue-datepicker dokumentacija, https://vue3datepicker.com, 11.6.2024.

[11] PostgreSQL dokumentacija, https://www.postgresql.org/docs, 11.6.2024.

[12] git stranica, https://git-scm.com, 11.6.2024.

[13] Github stranica, https://github.com, 11.6.2024.

[14] Primevue Menubar komponenta , https://primevue.org/menubar, 11.6.2024.

Privitak B: Pokretanje programa

Baza podataka

Konfiguraciju baze podataka potrebno je napraviti stvaranjem nove baze podataka i pokretanjem skripte *db.sql* datoteke koja se nalazi u korijenu repozitorija.

API

Prije pokretanja programskog sučelja potrebno je stvoriti datoteku *dbConnectionString.txt* i u nju upisati podatke za spajanje na prethodno stvorenu bazu podataka.

```
Host=localhost;Username=[username];Password=[password];
   Database=[dbName];Include Error Detail=[true for
   debugging; false for deployment]
```

Slika B1. Primjer datoteke *dbConnectionString.txt*

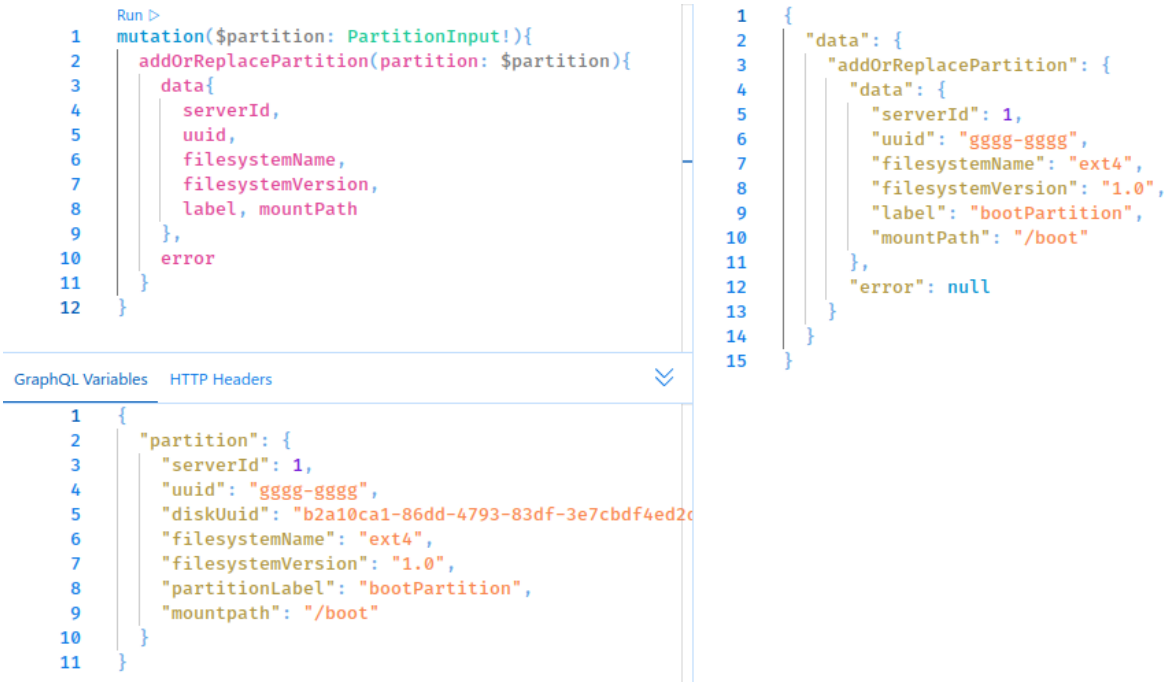
Program se pokreće iz komandne linije (terminala) pomoću naredbe "*dotnet run*" unutar direktorija *API*.

Nakon pokretanja može se pristupiti URL-u "*http://[URL aplikacije]:[PORT]//graphql*" čime se otvara web-stranica na kojoj se može vidjeti dokumentacija programskog sučelja te se mogu izvršavati upiti (prije konfiguriranje aplikacije *Monitor* upiti za dohvat podataka neće ništa vratiti):

```
Run ▷
{
    disk(
      serverId: 1,
      startDateTime: "2024-05-27 16:47",
      endDateTime: null,
      uuid: "b2a10ca1-86dd-4793-83df-3e7cbdf4ed2d")
    {
      serverId,
      uuid,
      type,
      serial,
      path,
      vendor,
      model,
      bytesTotal,
      partitions{
        uuid,
        label,
        filesystemName,
        filesystemVersion,
        mountPath,
        logs{
          date,
          bytes,
          usedPercentage
        }
      }
    }
}
```

```
{
  "data": {
    "disk": [
      {
        "serverId": 1,
        "uuid": "b2a10ca1-86dd-4793-83df-3e7cbdf4ed2d",
        "type": "gpt2",
        "serial": "S649NL0TA61676F",
        "path": "/dev/nvme0n1",
        "vendor": "ATA",
        "model": "Samsung SSD 980 1TB",
        "bytesTotal": 1000204886016,
        "partitions": [
          {
            "uuid": "gggg-gggg",
            "label": "bootPartition",
            "filesystemName": "ext4",
            "filesystemVersion": "1.0",
            "mountPath": "/boot",
            "logs": []
          }
        ]
      }
    ]
  }
}
```

Slika B2. Dohvat podataka o pohrani



```
Run ▷
mutation($partition: PartitionInput!){
  addOrReplacePartition(partition: $partition){
    data{
      serverId,
      uuid,
      filesystemName,
      filesystemVersion,
      label, mountPath
    },
    error
  }
}
```

```
{
  "data": {
    "addOrReplacePartition": {
      "data": {
        "serverId": 1,
        "uuid": "gggg-gggg",
        "filesystemName": "ext4",
        "filesystemVersion": "1.0",
        "label": "bootPartition",
        "mountPath": "/boot"
      },
      "error": null
    }
  }
}
```

GraphQL Variables | HTTP Headers

```
{
  "partition": {
    "serverId": 1,
    "uuid": "gggg-gggg",
    "diskUuid": "b2a10ca1-86dd-4793-83df-3e7cbdf4ed2c
    "filesystemName": "ext4",
    "filesystemVersion": "1.0",
    "partitionLabel": "bootPartition",
    "mountpath": "/boot"
  }
}
```

Slika B3. Dodavanje jedne particije

Monitor

Prije pokretanja aplikacije *Monitor* potrebno je stvoriti (ili izmijeniti) konfiguracijsku datoteku.

```
{
  "ApiUrl": "http://{API_URL}/graphql",
  "IntervalInMinutes": 1,
  "Cpu": true,
  "Program": true,
  "Storage": true,
  "Services": []
}
```

Slika B4. Primjer datoteke *config.json*

Također je potrebno instalirati aplikacije na koje se *Monitor* oslanja za ispravan rad. Pomoću naredbe "*which name*" može se provjeriti je li aplikacija instalirana. Aplikacije potrebne za ispravan rad programa:

- *lscpu*
- *top*
- *systemctl*
- *lsblk*
- *journalctl*

Nakon izmjene konfiguracijske datoteke i instalacije potrebnih programa, aplikacija se može pokrenuti pomoću naredbe "*dotnet run*".

Web-stranica

Prije pokretanja web-stranice potrebno je instalirati biblioteke na koje se aplikacija oslanja koristeći naredbu "*npm install*".

Nakon instalacije biblioteka, aplikacija se pokreće s naredbom "*npm run serve*".