UNIVERZITET U NOVOM SADU

FAKULTET TEHNIČKIH NAUKA U NOVOM SADU

Nikola Mijonić, PR49/2017

Migracija web aplikacije zadužene za nadgledanje radova distributivne mreže sa monolitne na mikroservisnu arhitekturu

PROJEKAT

- Primenjeno softversko inženjerstvo (OAS) -

Sadržaj

Opis rešavanog problema 2
Opis korišćenih tehnologija i alata 3
Opis rešenja problema 5
Monolitna arhitektura rešenja 5
Mikroservisna arhitektura rešenja 8
Postupak migracije sa monolitne na mikroservisnu arhitekturu 9
Komunikacija između mikroservisa 11
Predlozi za dalja usavršavanja 13
Literatura 14

OPIS REŠAVANOG PROBLEMA

Prvobitna monolitna implementacija web aplikacije zadužene za nadgledanje radova nad distributivnom mrežom imala je određene nedostatke koje je bilo neophodno rešiti. Adekvatnom analizom postojećih problema ustanovljeno je da bi migracija postojeće monolitne arhitekture na mikroservisnu arhitekutru rešila ključne probleme kao što su skalabilnost, održivost i međusobna zavisnost delova sistema.

Monolitna arhitektura predstavlja arhitekturu pogodnu za razvoj aplikacija do određenih figurativnih granica. Načini na koje se aplikacije razvijaju primenom ove arhitekure je lako prihvatljiv i znatno jednostavniji u odnosu na mikroservisnu arhitekturu. Najčešće se implementira kao višeslojna aplikacija u okviru koje se vrši komunikacija između slojeva. Jedna od glavnih karakteristika monolitne arhitekture je postojanje jedne obimne baze podataka koju direktno ili indirektno koristi cela aplikacija. Ovaj pristup nam omogućava relativno jednostavno očuvanje konzistentnosti podataka, ali potencijalno stvara prostor za nove probleme kao što su međusobna zavisnost delova sistema. Drugim rečima rečeno, ukoliko jedan deo sistem ne funkcioniše na predviđen način postoji velika verovatnoća da će to uticati na ostatak sistema. Osnovni zadatak web aplikacije je da zadovolji potrebe i zahteve korisnika. Ti zahtevi se izuzetno dinamičnim tempom menjaju pa je potrebno adekvatno odgovoriti na zahteve u vidu implementacije traženih zahteva. Potreba za konstantnim razvojem aplikacija vremenom postane jedan od glavnih ograničavajućih faktora monolitne arhitekture. Aplikacije bazirane na monolitnoj arhitekturu imaju potencijalan problem sa skaliranjem i samim tim vrlo često postanu neodržive i sa poteškoćama se odgovara na zahteve klijenata koji se nagomilavaju. [1]

Mikroservisna arhitektura predstavlja arhitekturu koja se postiže racionalnom podelom monolitne arhitekture na mikroservise. Svaki mikroservis prestavlja proces za sebe koji rukuje određenim specifičnim delom sistema. Prilikom odabira dela sistema monolitne arhitekture koji će se izmestiti u poseban mikroservis treba težiti ka što manjoj međusobnoj zavisnosti od postojećih mikroservisa. Na taj način rešava se međusobna zavisnost sistema koja prestavlja jedan od glavnih problema monolitne arhitekture. Svaki mikroservis ima svoju bazu podataka koja sadrži informacije neophodne samo za funkcionisanje tog mikroservisa. Mikroservisi međusobno komuniciraju i na taj način prestavljaju jednu celinu koja omogućava normalan rad aplikacije. Pored mnogih prednosti mikroservisa kao što su mnogo jednostavnija skalabilnost, održivost, dodavanje novih funkcionalnosti postoje i određene mane. Potrebno je obezbediti adekvatnu komunikaciju mikroservisa i sama migracija sa monolitne arhitekture može zahtevati mnoštvo vremenski skupih izmena. [2]

Kako bi se na adekvatan način izvršila migracija monolitne arhitekture prethodno implementirana web aplikacija zadužene za nadgledanje radova nad distributivnom mrežom na mikroservinsu arhitekturu izvršen je proces kontejnerizacije. Kontejnerizacija predstavlja proces u okviru kojeg se delovi monolitne aplikacije, koji će prestavljati posebne mikroservise, smeštaju u posebne kontejnere. Na taj način sve što je potrebno za normalno funkcionisanje jednog mikroservisa nalazi se u kontejneru. Postignut je visok nivo enkapsulacije i mogućnosti jednostavnog izvršavanja mikroservisa na različitim platformama što u velikoj meri može olakšati proces puštanja aplikacije u produkcioni rad. Nakon što je monolitna aplikacija podeljena u kontejnere izvršena je orkestracija kontejnera koja definiše način na koji se rukuje prethodno napravljenim kontejnerima. [3]

Detaljnom analizom postojeće arhitekture odlučeno je da se postojeća arhitektura podeli na tri mikroservisa čijom komunikacijom će biti omogućeno optimalnije i skalabilnije funkcionisanje sistema. Svaki od mikroservisa zadužen je za rad sa njemu svojstvenim resursima i na taj način u zadovoljavajućoj meri ukljonjena je zavisnost između delova sistema i postignuto je jednostavno dodavanje novih funkcionalnosti. Dodavanje nove funkcionalnosti u monolitnoj arhitekturi uglavnom zahteva testiranje delova aplkacije koji nisu menjani kako bi se utvrdilo da nova funkcionalnost nije negativno uticala na te delove.

OPIS KORIŠĆENIH TEHNOLOGIJA I ALATA

Prilikom izrade projektnog zadatka korišćeni su sledeći alati i tehnologije:

- ***Visual Studio Code*** **–** predstavlja kod editor koji je razvijen od strane *Microsoft*-a. Veliki broj korisnih ekstenzija znatno može ubrzati pisanje koda što ujedno predstavlja i jedan od razloga izbora ovog alata. [4]

- ***Visual Studio 2019*** **–** predstavlja razvojno okruženje razvijeno od strane *Microsoft*-a. Poseduje podršku za mnoštvo programskih jezika, među kojima je najpopularniji *C#.* Omogućava razvoj aplikacija raznovrsnih arhitektura i karakteristika. Sadrži mnoštvo ugrađenih klasa i implementiranih funkcionalnosti koje znatno ubrzavaju i olakšavaju proces razvoja softvera. Pored svega navedenog poseduje i debugger koji omogućava pronalaženje uzroka neželjenog ponašanja softvera na izuzetno zavidnom nivou. [5]

- ***SQL Server Management Studio*** **–** predstavlja alat razvijen od stane *Microsoft*-a koji omogućava upravljanje, konfiguraciju i monitoring *SQL* instanci i baza podataka. [6]

- ***Docker Desktop*** **–** predstavlja aplikaciju koja se na *Windows* operativni sistem instalira u nekoliko jednostavnih koraka. Ovaj alat koristi se za kreiranje kontejnera i njihovo deljenje. Na ovaj način postiže se nezavnisnot od platforme na kojoj se naš softver, najčešće mikroservis, izvršava. [7]

- ***C#*** **-** prestavlja objektno-orijentisani programski jezik razvijen od strange *Microsoft*-a. Mnoštvo programera se odlučuje za ovaj programski jezik zbog širokog spektra mogućnosti koji pruža. [8]

- ***ASP .NET CORE 3.1*** **–** predstavlja framework javno dostupnog koda namenjen za razvoj web aplikacija. Poseduje mogućnost izvšavanja na raznim platformama i koristi se za razvoj zadnje strane web aplikacije. [9]

- ***Entity Framework Core*** **–** predstavlja proširivu, open-source verziju *Entity Framework* tehnologije za pristup i manipulaciju podacima baze podataka. Može se koristiti kao objektno-relacioni maper koji omogućava korišćenje *.NET* objekata za manipulaciju podacima. [10]

- ***REST*** **(*eng. Representational state transfer*)** **-** predstavlja stil softverske arhitekture koji definiše pravila kojima se treba voditi prilikom razvoja veb servisa. Doprinosi jednostavnosti korišćenja servisa zadnje strane zbog univerzalnog definisinja načina na koji se podacima pristupa. Komunikacija sa *REST* serverom se najčešće obavlja upotrebom *HTTP* ili *HTTPS* protokola, a podaci se razmenjuju u *JSON* formatu. [11]

- ***Swagger*** **–** koristi se za jednostavnu dokumentaciju, testiranje i pregled dostupnih funkcionalnosti koje izlaže *REST API* u formatu koji je čitljiv čoveku. [12]

- ***MailKit .NET*** **-** prestavlja open-source biblioteku koja je korišćena za slanje email-a. [13]

- ***AutoMapper*** **–** predstavlja malu biblioteku koja se koristi za mapiranje objekta jednog tipa u objekat drugog tipa. Korišćena je za transformaciju modela podataka u objekte koji se koriste za prenos između slojeva aplikacije i obrnuto. [14]

- ***DAPR .NET -*** predstavlja modul koji se koristi za komunikaciju između mikroservisa. Komunikacija se izvršava asinhrono upotrebom *HTTP* protokola. Omogućava komunikaciju između mikoroservisa napisanih korišćenjem različitih programskih jezika. Bitno je napomenuti da u velikoj meri olakšava kreiranje skalabilnih aplikacija zbog jednostavnosti komunikacija između mikroservisa koju *DAPR* pruža. [15]

- ***Angular 11 –*** prestavlja *framework* razvijen od strane *Google*-a koji omogućava razvoj *Single-page* aplikacija. Korišćenje ovog *framework*-a u velikoj meri olakšava razvoj klijentskih strana web aplikacija poznatijih pod nazivom *front-end*. Jedna od glavnih karakteristika ogleda se u mnoštvu developera koji koriste ovaj *open-source framework*. [16]

- ***Angular Material –*** predstavlja biblioteku koja sadrži gotove komponente koje se na vrlo jednostavan način integrišu u bilo koji *Angular* projekat. Komponente se mogu modifikovati kako bi zadovoljile potrebe same aplikacije koja se kreira. Korišćenjem ove bibloteke dolazi do uštede vremena koje bi bilo potrebno da se već gotove komponente samostalno implementiraju. [17]

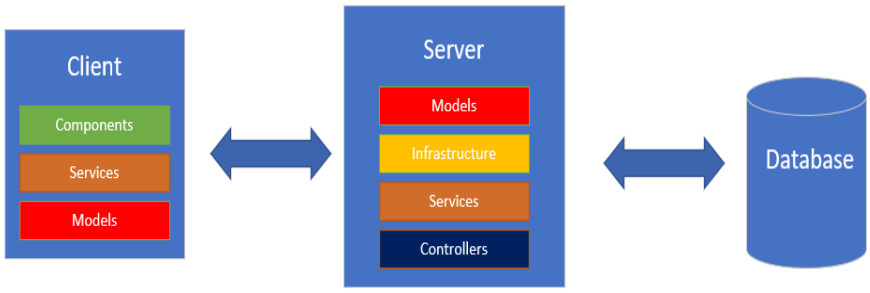
- ***HTML (eng. HyperText Markup Language) –*** predstavlja opisni jezik namenjen za opis prednje strane veb aplikacije. [18]

- ***CSS (eng. Cascade Style Sheet) -*** predstavlja jezik čijom upotrebom se definiše izgled prethodno kreiranih *HTML* tagova. Pruža širok spektar mogućnosti kada je dizajn u pitanju. [19]

OPIS REŠENJA PROBLEMA

Monolitna arhitektura rešenja

Monolitna implementacija web aplikacije zadužene za nadgledanje radova nad distributivnom mrežom imala je određene nedostatke kao što je već rečeno. Rešenje tih problema pronalazi se u mikroservisnoj arhitekturi koja je zahtevala određene izmene u postojećem sistemu. Izmene su zahtevale reorganizaciju i refaktorisanje pojedinih delova sistema kako bi funkcionalnosti ostale prisutne pri prelasku na mikroservisnu arhitekturu. Sledi opis monolitne arhitekture kako bi se proces migracije na mikroservisnu arhitekturu jednostavnije opisao.

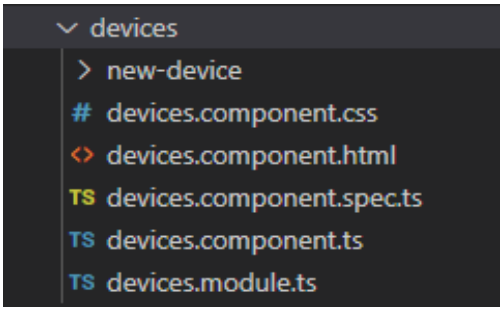


Slika 1 – Arhitektura monolitnog rešenja

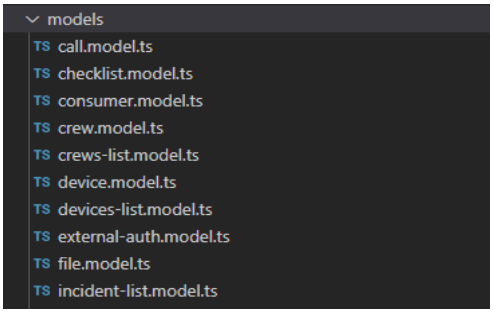
Monolitna implementacija može se podeliti na određene celine kako bi se sistem i njegova problematika mogli jasnije predstaviti (Slika 1) :

- Klijentska aplikacija - omogućava korisnicima jednostavno korišćenje funkcionalnosti sistema
- Server - predstavlja *API* korišćen za obradu pristiglih korisničkih zahteva
- Baza podataka - korišćena za smeštanje informacija od značaja za posmatrani sistem

Klijentska aplikacija predstavlja *Single-Page* aplikaciju implementiranu korišćenjem popularnog *Angular framework*-a. Kada je reč o dizajnu pored *CSS* korišćen je i *Angular Material* koji je u velikoj meri uticao na pozitivno korisničko iskustvo prilikom korišćenja same aplikacije. Kako bi se ispoštovali određeni paterni koje *Angular* nalaže srodne komponente su grupisane u određene foldere i sadrže sopstvene module neophodne za njihovo normalno funkcionisanje (slika 2). Na taj način postignut je visok stepen mogućnosti ponovnog korišćenja jednom napravljenih komponenti. Kako bi se zadovoljile potrebe korisnika koji aplikaciju mogu otvoriti na uređajima koji poseduju ekrane različitih rezolucija aplikacija je implementirana tako da bude potpuno prilagodljiva svim rezolucijama. [20]

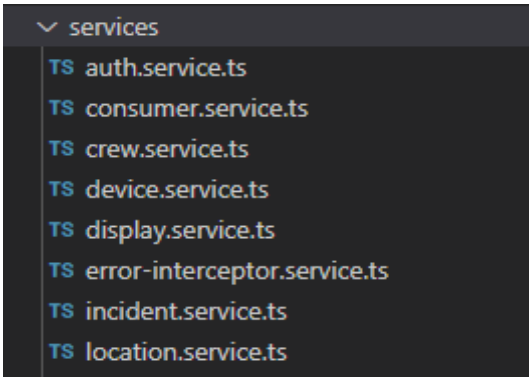


Slika 2 – Modularnost komponenti

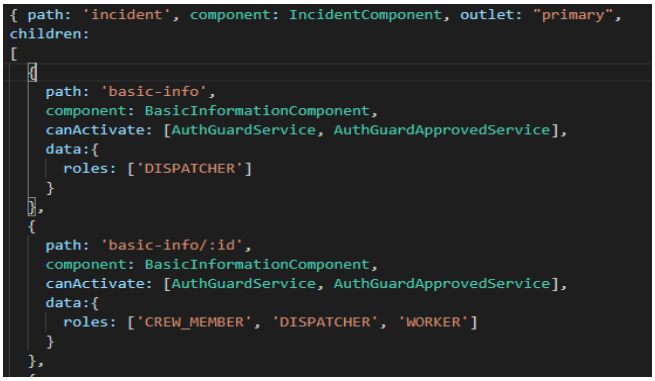


Slika 3 – Modeli podataka

Klijentska aplikacija sadrži modele koji se koriste kako bi se podaci pristigli sa servera mogli adekvatno sačuvati i koristiti po potrebi (slika 3). Za sve podatke koji pristižu ili se šalju ka serveru kreiran je poseban model podataka. Servisi koji se nalaze u okviru klijentske aplikacije korišćeni su uglavnom za komunikaciju sa serverom putem *HTTP* protokola i njegovih metoda (slika 4). U zavisnosti od potrebe korišćene su različite metode *HTTP* protokola. Kako bi se prilikom dobavljanja potencijalno velike količine podataka sa servera rasteretila klijentska aplikacija i smanjilo vreme prenosa podataka implementirana je paginacija. Pored paginacije sve komponente koje su zadužene za prikaz podataka imaju implementiranu pretragu, filter i sortiranje kako bi korisnik na što jednostavniji način pronašao željene informacije. Validacija formi implementirana je korišćenjem jednostavnih reaktivnih formi i njihovih ugrađenih validatora koje se nalaze u skolopu *Angular*-a. Pored pethodno pomenutih delova bitno je napomenuti da se komponente renderuju po kriterijumu koji definišu rute (slika 5). [21]



Slika 4 - Prikaz pojedinih servisa



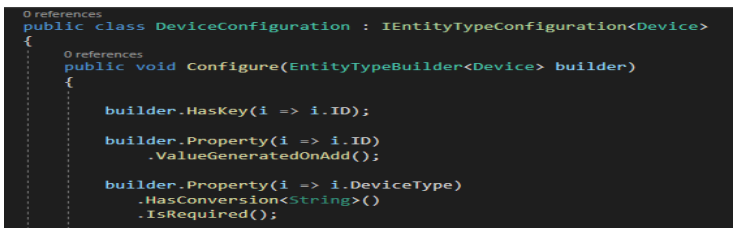
Slika 5 - Primer rutiranja

Serverska aplikacija predstavlja višeslojnu aplikaciju u okviru koje svaki sloj ima svoje zaduženje. Kreiranjem slojeva postignut je visok nivo enkapsulacije na nivou čitave aplikacije. U okviru sistema kreirani su sledeći slojevi:

- Sloj modela podataka
- Infrastrukturni sloj
- Servisni sloj
- Sloj kontrolera

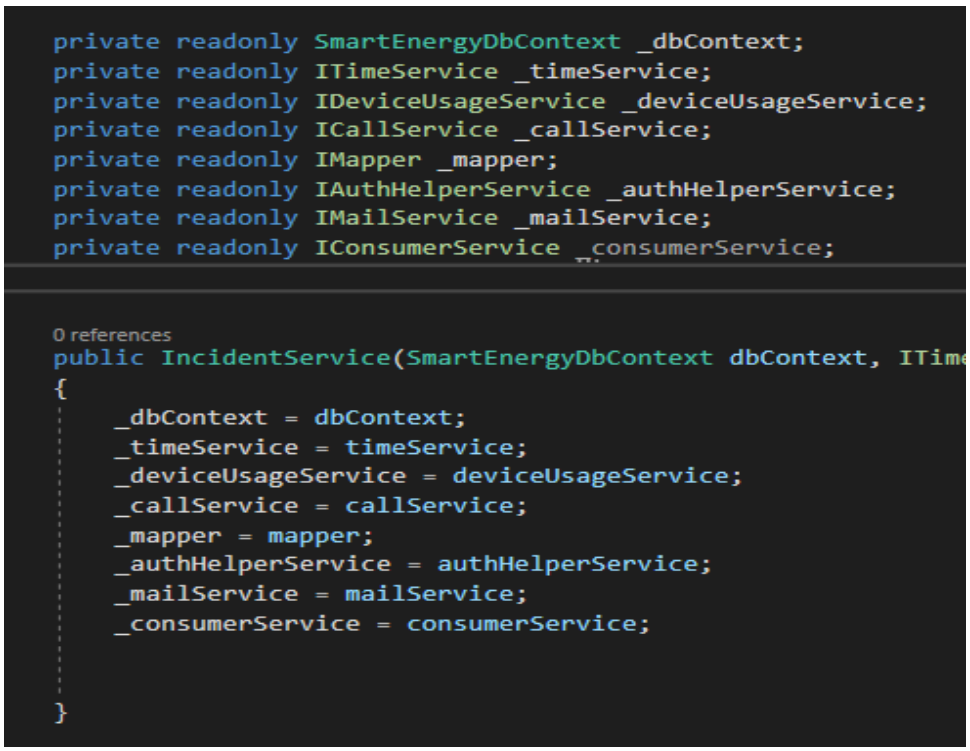
Sloj modela podataka korišćen je za kreiranje modela koji će se koristi na nivou čitave aplikacije. Pored ove svrhe modeli su korišćeni kako bi se kreirala baza podataka ugrađenim mehanizmima koje nudi *Entity Framework Core*.

Infrastrukturni sloj zadužen je za kreiranje konteksta baze podataka kao i za podešavanje relacija između tabela baza podataka koje nastaju na osnovu povezanosti prethodno pomenutih modela. Pored relacije modeluju se i odgovarajuća ograničenja koja zadovoljavaju potrebe sistema. Za konfiguraciju je korićen *Fluent API* (slika 6). [22]



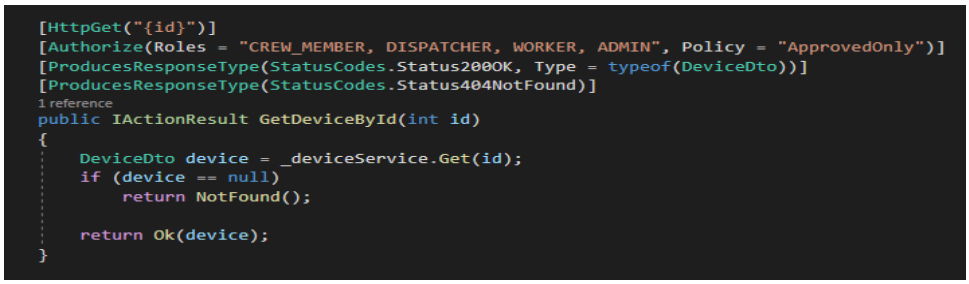
Slika 6 – Konfiguracija uređaja na mreži

Servisni sloj je sloj u okviru kojeg je implementirana biznis logika. Ovaj sloj sadrži kompletnu implementaciju svih funkcionalnosti koje serverska aplikacija pruža. U konstruktoru servisa se korišćenjem *Dependency Injection* paterna injektuju drugi servisi ukoliko je to potrebno (slika 7). Pored izvršavanja funkcionalnosti sistema servisni sloj je zadužen za validaciju pristiglih podataka sa klijentske aplikacije. [23]



Slika 7 – Injektovanje servisa

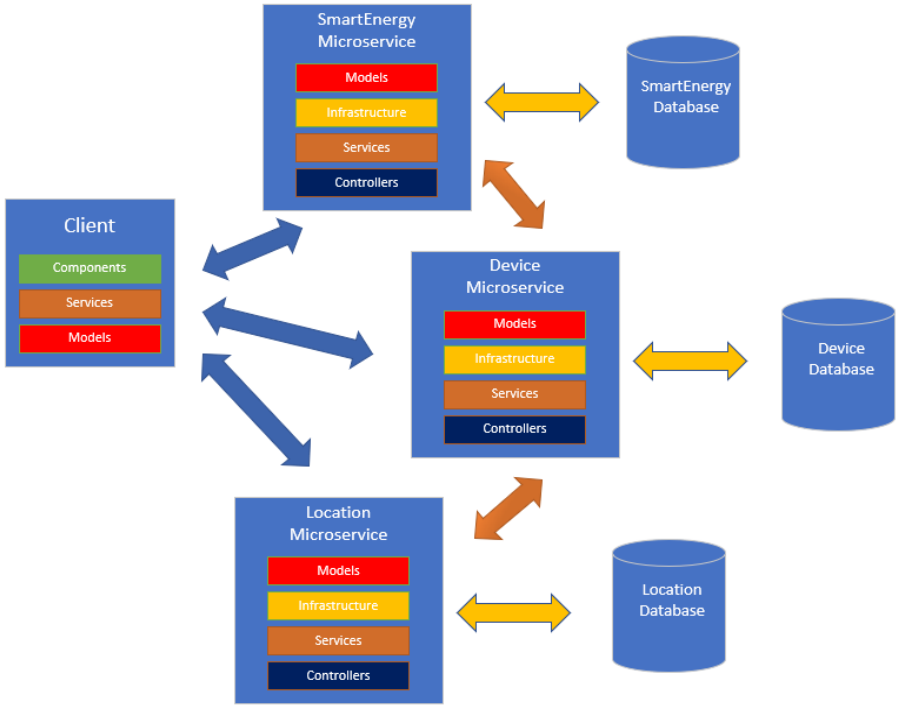
Sloj kontolera omogućava komunikaciju serverske i klijentske aplikacije. Podaci pristigli sa klijentske aplikacije u kontolerima se smeštaju u odgovarajuće *DTO* (*eng. Data Transfer Object)* objekte nakon čega se vrši poziv servisa koji pruža funkcionalnost neophodnu za izvršavanje klijentskog zahteva (slika 8). Komunikacija se vrši upotrebom metoda *HTTP* protokola.



Slika 8 – Metoda kontrolera

Mikroservisna arhitektura rešenja

Migracija monolitne arhitekture na mikroservisnu zahtevala je određeno vreme za planiranje i samu implementaciju. Mikroservisi su kreirani tako da samostalno predstavljaju smislenu celinu koja može nastati podelom već postojeće implementacije (slika 9). Sledi detaljan opis toka migracije.



Slika 9 – Mikroservisna arhitektura

U okviru mikroservisne arhitekture kreirana su tri mikroservisa :

- Mikroservis zadužen za manipulaciju lokacijama
- Mikroservis zadužen za manipulaciju uređajima na mreži
- Mikroservis koji predstavlja preostalu celinu nakon podele monolitne arhitekture

Postojanje mikroservisa namenjenog za manipulaciju lokacijama javilo se kao posledica činjenice da taj servis može predstavljati usko grlo u komunikaciji i jedan je od ključnih delova sistema. Izolacija ovog dela sistema u mikroservis omogućila je jednostavan proces instanciranja ovog dela sistema čime se obezbeđuje mogućnost sistema da odgovori na mnoštvo potencijalnih zahteva.

Mikroservis koji je namenjen za manipulaciju uređajima na mreži predstavlja deo sistema koji je izuzetno bitan za ispravno funkcionisanje ostatka sistema pa je neophodno omogućiti njegovu izolaciju. Kao i u slučaju mikroservisa namenjenog za manipulaciju lokacijama ovaj mikroservis komunicira sa mnoštvom drugih pa se za njegovu implementaciju utrošilo dodatno vreme kako bi se postigao zadovoljavajuć kvalitet.

Mikroservis koji predstavlja ostatak monolitnog rešenja sadrži sve preostale funkcionalnosti. Manipulaciju dokumentima u sistemu, korisnicima, ekipama i svime sto sistem predstavlja. Odluka da ovaj mikroservis ostane obimniji nego preostala dva donešena je kako bi grupe sličnih entiteta ostale u okviru istog mikroservisa. Ovaj mikroservis zadužen je i za manupulaciju korisnicima.

Postupak migracije sa monolitne na mikroservisnu arhitekturu

Postupak migracije će se opisati korišćenjem primera kreiranja mikroservisa namenjenog za manipulaciju uređajima na mreži. Ključni koraci ovog postupka su identični i za prestala dva mikroservisa pa iz tog razloga neće biti opisani.

Nakon donešene odluke o funkcionalnosti koju mikroservis pruža započinje proces refaktorisanja koda. Neophodno je model posmatranog dela sistema, u ovom slučaju model podataka uređaja na mreži izdvojiti u novi *ASP .NET CORE 3.1* projekat koji će predstavlja novi mikroservis. Kao što je već rečeno svaki mikroservis ima svoj model podataka koji mu omogućava normalno funkcionisanje. Uklanjanje određenog modela iz postojećeg sistema zahteva uklanjanje postojećih referenci ka tom modelu.

Neophodno je infrastrukturu postojećeg sistema prilagoditi novonastalim izmenama, njene delove izmestiti u novonastali mirkoservis i ukloniti ograničenja referencijalnog integriteta. Ista procedura primenjena je za servis, kontroler i maper. Na ovaj način postigli smo potpuno izmeštanje dela sistema u mikroservis. Bitno je napomenuti da svaki mikroservis u sistemu ima svoju bazu podataka kako bi u potpunosti bio izolovan od ostatka sistema.

Kontejnerizacija mikoroservisa izvršena je ugrađenim mehanizmom koji *Visual Studio 2019* razvojno okruženje nudi. U okviru *Dockerfile* (slika 10) nalaze sve potrebne informacije za pokretanje ovog kontejnera.

```
3  FROM mcr.microsoft.com/dotnet/aspnet:3.1 AS base
4  WORKDIR /app
5  EXPOSE 80
6
7  FROM mcr.microsoft.com/dotnet/sdk:3.1 AS build
8  WORKDIR /src
9  COPY ["SmartEnergy.DevicesAPI/SmartEnergy.DevicesAPI.csproj", "SmartEnergy.DevicesAPI/"]
10 COPY ["SmartEnergy.Contract/SmartEnergy.Contract.csproj", "SmartEnergy.Contract/"]
11 RUN dotnet restore "SmartEnergy.DevicesAPI/SmartEnergy.DevicesAPI.csproj"
12 COPY . .
13 WORKDIR "/src/SmartEnergy.DevicesAPI"
14 RUN dotnet build "SmartEnergy.DevicesAPI.csproj" -c Release -o /app/build
15
16 FROM build AS publish
17 RUN dotnet publish "SmartEnergy.DevicesAPI.csproj" -c Release -o /app/publish
18
19 FROM base AS final
20 WORKDIR /app
21 COPY --from=publish /app/publish .
22 ENTRYPOINT ["dotnet", "SmartEnergy.DevicesAPI.dll"]
23
```

Slika 10 – Docker fajl

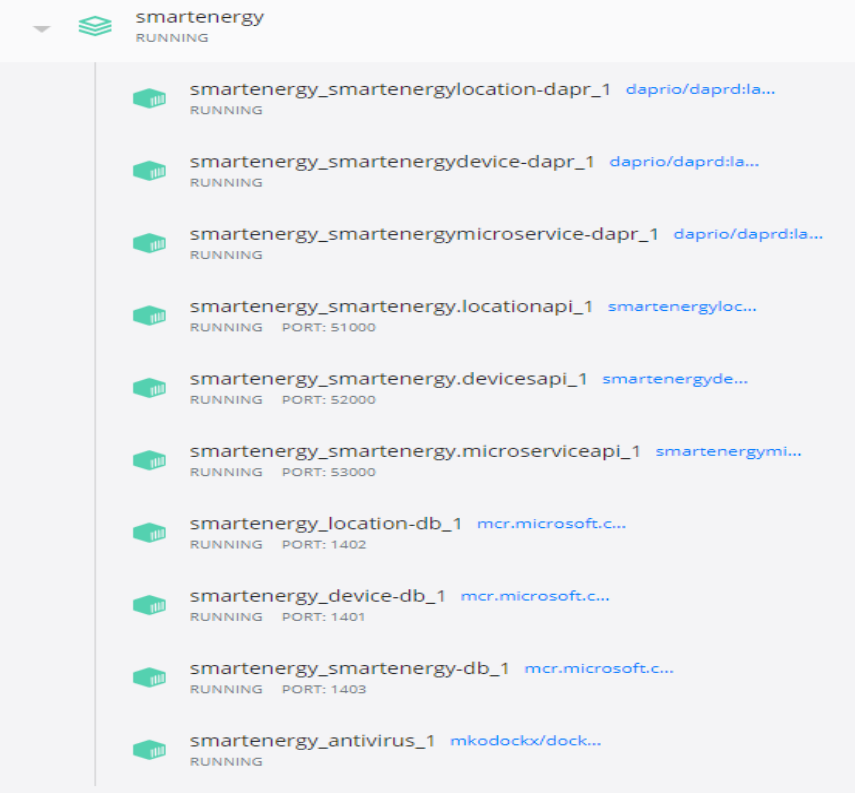
Nakon kreiranja pojedinačnih kontejnera izvršena je orkestracija kontejnera korišćenjem *Docker compose* orkestratora. Svaki kontejner sadrži lokalnu sliku servisa, sliku *SQL* servera koja se povlači sa *Microsoft* kontejnera i sliku *DAPR* – a korišćenu za komunikaciju mikroservisa koja se povlači sa *Docker* repozitorijuma. Može se zaljučiti da je za svaki mikroservis potrebno tri kontejnera (slika 12). Bitno je napomenuti da postoji jedan kontejner korišćen za smeštanje antivirusa koji je korišćen pri skeniranju fajlova koji se dodaju u sistem. [24]

```
#devices
smartenergy.devicesapi:
  image: ${DOCKER_REGISTRY-}smartenergydevicesapi
  build:
    context: .
    dockerfile: SmartEnergy.DevicesAPI/Dockerfile
  ports:
      - "52000:50001"
      - "44373:80"
  depends_on:
      - device-db

smartenergydevice-dapr:
  image: "daprio/daprd:latest"
  command: [ "./daprd", "-app-id", "smartenergydevice", "-app-port", "80" ]
  depends_on:
    - smartenergy.devicesapi
  network_mode: "service:smartenergy.devicesapi"


device-db:
  image: mcr.microsoft.com/mssql/server
  ports:
    - "1401:1433"
  environment:
    ACCEPT_EULA: "Y"
    SA_PASSWORD: "Your+password123"
```

Slika 11 – Deo Docker-compose fajla namenjenog mikroservisu za uređaje

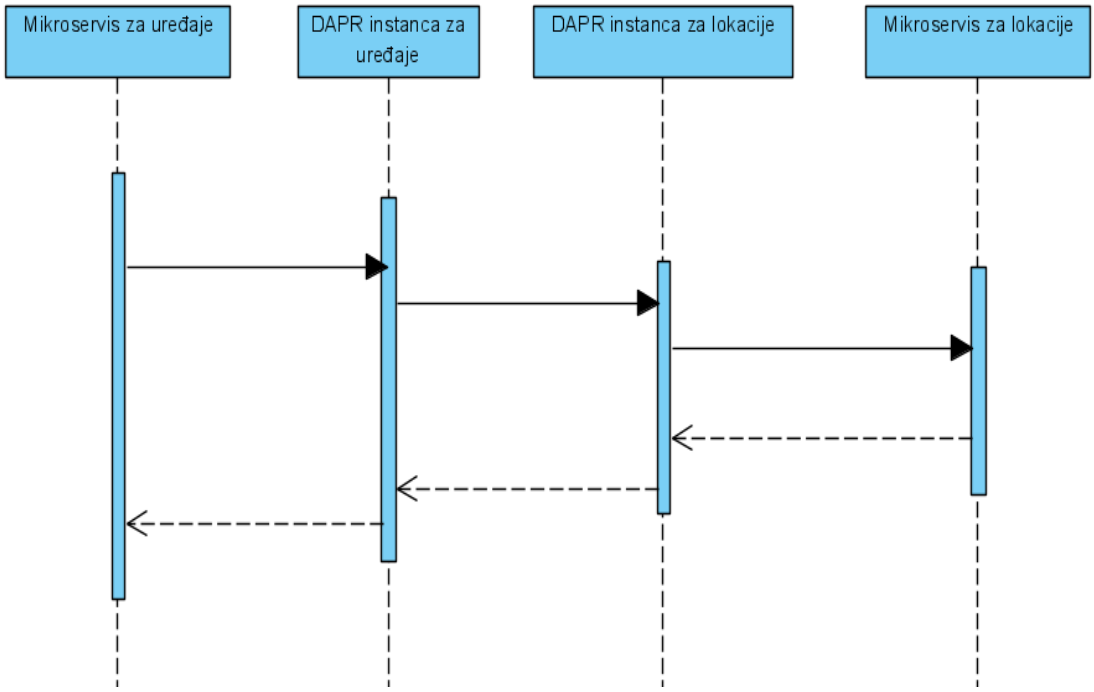


Slika 12 – Docker kontejneri

Komunikacija između mikroservisa

Komunikacija između mikroservisa predstavlja jedan od najbitnijih delova mikroservisne arhitekture. Bez adekvatne komunikacije smisao mikroservisa se gubi. Za komunikaciju mikroservisa korišćen je *DAPR .NET* koji koristi *sidecar* softverski patern. Zahvaljujući ovom paternu *DAPR* se veoma jednostavno integriše u bilo koji projekat. Ukoliko razrešavanje nekog klijentskog zahteva potražuje komunikaciju više mikroservisa ta komunikacija odvija se posredstvom *DAPR* instanci (slka 13). [25]

Pokretanja postojećih kontejnera izvršava se metodom *docker-compose up* u okviru *Terminal* – a koji pruža *Visual Studio 2019* razvojno okruženje. *DAPR* instanca će čekati servis u blokirajućem režimu rada sve dok servis za koji je vezana ne postane dostupan.



Slika 13 – Tok komunikacije nakon iniciranja klijentskog zahteva

Nakon iniciranja klijentskog *HTTP* zahteva pogađa se neka od postojećih akcija kontrolera u zavisnosti od strukture samog zahteva. Nakon toga mikroservis kojem pripadne zaduženje za rešavanje tog klijentskog zahteva preduzima potrebne mere kako bi ga što efikasnije rešio. Ukoliko je za rešavanje tog zahteva neophodno izvršiti komunikaciju sa drugim mikroservisom ta komunikacija izvršava se upotrebom *DAPR* instanci odgovarajućih mikroservisa. Na ovaj način vrši se enkapsulacija komunikacije koja prikriva složenost sistema. Prvobitno se *DAPR* instanca za uređaje obraća DAPR instanci za lokacije nakon čega u zavisnosti od samog zahteva *DAPR* instanca za lokacije zatraži potrebne informacije od servisa za lokacije (slika 13).

Komunikacija je implementirana korišćenjem ugrađene *C#* klase *DaprClient* koja predstavlja posrednika u komunikaciji. *DaprClient* se injektuje u konstruktoru servisa u okviru kojeg će biti korišćena kada to bude potrebno (slika 14). Komunikacija se izvršava asinhrono jer se na taj način sistem štiti od potencijalnog otkaza mikroservisa sa kojim je komunikacija ostvarena. Ukoliko mikroservis više nije dostupan ili je istekao predefinisani *Timeout* dolazi do izuzetka koji je adekvatno obrađen na aplikativnom nivou i sistem uz adekvatnu poruku obaveštava korisnika da zahtev nije moguće izvršiti (slika 15).

```
private readonly DeviceDbContext _dbContext;
private readonly IMapper _mapper;
private readonly DaprClient _daprClient;

0 references
public DeviceService(DeviceDbContext dbContext, IMapper mapper, DaprClient daprClient)
{
    _dbContext = dbContext;
    _mapper = mapper;
    _daprClient = daprClient;
}
```

Slika 14 – Injektovanje DaprClient klase u servis koji je zadužen za uređaje

```
Device device = _dbContext.Devices.FirstOrDefault(x => x.ID == id);
DeviceDto deviceDto = new DeviceDto();

if(device != null)
{
    deviceDto = _mapper.Map<DeviceDto>(device);

    try
    {
        LocationDto location = await _daprClient.InvokeMethodAsync<LocationDto>(
                                    HttpMethod.Get, "smartenergylocation", $"/api/locations/{device.LocationID}");
        deviceDto.Location = location;

    }
    catch(Exception e)
    {
        throw new LocationNotFoundException("Location service is unavailable right now.");
    }
}

return deviceDto;
```

Slika 15 – Primer komunikacije između mikroservisa

PREDLOZI ZA DALJA USAVRŠAVANJA

Komunikacija između mikroservisa predstavlja ključni deo mikroservisne arhitekture. Međusobna zavisnost mikroservisa je nešto što treba izbegavati ali vrlo često je ta zavisnost u realnim sistemima prisutna. Ova neželjena osobina mikroservisne arhitekture nije uspešno izbegnuta u postojećoj implementaciji web aplikacije zadužene za nadgledanje radova nad distributivnom mrežom. Realni sistem koji se modelovao po svojoj prirodi zavisi od pojedinih podsistema u okviru njega pa samim tim nije moguće izvršiti potpunu dekompoziciju sistema na nezavisne celine, ali i dalje postoji prostora za optimizaciju.

Postojeća implementacija mikroservisne arhitekture sastoji se od tri mikroservisa. Jedan od tih mikroservisa je mikroservis zadužen za manipulaciju dokumentima, korisnicima, potrošacima. Iz prethodno navedenih zaduženja mikroservisa jasno se vidi da postoji mnogo prostora za podelu tog mikroservisa na manje celine odnosno za kreiranje novih znatno manjih mikroservisa. Na ovaj način narušena je jedna od osnovnih ideja mikroservisne arhitekture, a to je težnja ka što manjim nezavisnim celinama koje predstavljaju mikroservise. Taj deo je namenski izostavljen zato što izlazi iz okvira specifikacije predmetnog projekta, ali je sigurno nešto što bi u budućnosti trebalo izmeniti.

Trenutna implementacija potencijalno može prouzrokovati nekonzistentnosti u sistemu. Neretko se u mikroservisnoj arhitekturi dešava da odgovor na klijentski zahtev podrazumeva komunikaciju i modifikaciju podataka više mikroservisa. Kako bi se adekvatno odgovorilo na kljienski zahtev neophodno je da se uspešno izvrši akcija više mikroservisa. Prethodno opisani proces ostavlja veliki prostor za nekonzistentnost u sistemu. Ukoliko se desi da jedan mikroservis modifikuje svoje podatke sa ciljem odgovora na klijentski zahtev, a drugi mikroservis iz nekog razloga ne uradi svoj deo zaduženja klijentski zahtev neće biti uspešno izvršen. Problem se javlja zato što je prvi mikroservis već izvršio modifikaciju svojih podataka, a nije implementirana inverzna funkcija koja bi omogućila vraćanje modifikovanih podataka u prethodno stanje. Ovo bi se jednostavno moglo rešiti primenom *Saga* paterna koji se primenjuje u mikroservisnoj arhitekturi kako bi se očuvala konzistentnost podataka. [26]

Klijentska aplikacija trenutno je svesna mikroservisne arhitekture serverske strane zato što se u zavisnosti od zahteva ciljano gađa različit izloženi *API* mikroservisa. Ovo je nešto što bi definitivno trebalo promeniti zato što klijentska aplikacija ne bi trebala da bude svesna detalja implementacije serverske strane. Ovo se vrlo jednostavno rešava dodavanjem *Kubernetes* – a koji za klijentsku stranu ima izložen *API* koji se ponaša kao *Load Balancer* i u zavisnosti od zahteva poziva određeni mikroservis. [27]

Postupak pronalazaka grešaka i praćenja dešavanja u sistemu baziranom na mikroservisima je poprilično komplikovan zbog asinhrone prirode komunikacije koja se obavlja između mikroservisa. Ovaj postupak znatno bi se uprostio implementacijom sistema za logovanje podataka. Na taj način mogao bi se ispratiti tok dešavanja i potencijalno lakše detektovati problem. Sistem za logovanje znatno bi unapredio stepen sigurnosti i bezbednosti zbog mogućnosti monitoringa celokupnog sistema i pravovremene reakcija na događaje.

Geografska distribuiranost korisnika je nešto što se može očekivati u sistemu kao što je trenutno modelovani sistem. Trenutno se svi mikroservisi izvršavaju lokalno što nije najbolja praksa u većini slučajeva. Kako bi se na što efikasniji način odgovorilo na zahteve klijenata neophodno je postojeće lokalne mikroservise izmestiti u *Cloud* bazirane sisteme kao što su recimo *Azure*. U zavisnosti od geografske lokacije sa koje stiže zahtev, taj zahtev bi se prosleđivao mikroservisu koji je najmanje opterećen i koji je u stanju da najbrže odgovori. [28]

LITERATURA

[1] ***https://microservices.io/patterns/monolithic.html, Monolitna arhitektura***
[2] ***https://microservices.io/patterns/microservices.html, Mikroservisna arhitektura***
[3] ***https://www.sumologic.com/glossary/application-containerization/, Kontejnerizacija aplikacije***
[4] ***https://code.visualstudio.com/, Visual Studio Code***
[5] ***https://visualstudio.microsoft.com/vs/, Visual Studio 2019***
[6] ***https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15, SQL Server Management Studio***
[7] ***https://www.docker.com/products/docker-desktop, Docker Desktop***
[8] ***https://docs.microsoft.com/en-us/dotnet/csharp/, C#***
[9] ***https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-aspnet-core?view=vs-2019, ASP .NET CORE 3.1***
[10] ***https://docs.microsoft.com/en-us/ef/core/, Entity Framework Core***
[11] ***https://en.wikipedia.org/wiki/Representational_state_transfer, REST***
[12] ***https://swagger.io/, Swagger***
[13] ***https://www.nuget.org/packages/MailKit/, MailKit***
[14] ***https://code-maze.com/automapper-net-core/, AutoMapper***
[15] ***https://docs.microsoft.com/en-us/dotnet/architecture/dapr-for-net-developers/getting-started, DAPR .NET***
[16] ***https://angular.io/, Angular***
[17] ***https://material.angular.io/, Angular Material***
[18] ***https://sr.wikipedia.org/sr-ec/HTML, HTML***
[19] ***https://sr.wikipedia.org/sr-ec/CSS, CSS***
[20] ***https://en.wikipedia.org/wiki/Single-page_application, Single-Page Application***
[21] ***https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol, HTTP***
[22] ***https://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx, Fluent API***
[23] ***https://en.wikipedia.org/wiki/Dependency_injection, Dependency Injection***
[24] ***https://docs.docker.com/compose/, Docker Compose***
[25] ***https://dzone.com/articles/sidecar-design-pattern-in-your-microservices-ecosy-1, Sidecar pattern***
[26] ***https://microservices.io/patterns/data/saga.html, Saga pattern***
[27] ***https://kubernetes.io/, Kubernetes***
[28] ***https://azure.microsoft.com/en-us/, Azure Cloud***