Programsko rješenje poslužiteljskog dijela web sustava za potporu upravljanja projektima agilnog razvoja programske podrške zasnovano na višenitnom paralelizmu i arhitekturi mikrousluga

Arlović, Matej

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:690643

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij

PROGRAMSKO RJEŠENJE POSLUŽITELJSKOG DIJELA WEB SUSTAVA ZA POTPORU UPRAVLJANJA PROJEKTIMA AGILNOG RAZVOJA PROGRAMSKE PODRŠKE ZASNOVANO NA VIŠENITNOM PARALELIZMU I ARHITEKTURI MIKROUSLUGA

Diplomski rad

Matej Arlović

Osijek, 2021.

SADRŽAJ

1. UVOD

U posljednje vrijeme porasla je popularnost razvojnog okvira *Node.js* koji se koristi za poslužiteljske aplikacije. Ono što *Node.js* čini boljim u odnosu na druge razvojne okvire je njegova mogućnost korištenja za sve vrste poslužiteljskih aplikacija bez obzira na njihovu složenost i veličinu. Arhitektura zasnovana na mikrouslugama stavlja naglasak na male i samoodržive procese.

U teorijskom dijelu diplomskog rada bit će opisani pristupi, postupci i izazovi vođenja projekata agilnog razvoja programske podrške, te analizirane mogućnosti postojećih alata i okolina. Nadalje, bit će analizirana i predložena arhitektura mikrousluga prikladna za ostvarenje poslužiteljskog dijela web sustava za potporu projektima agilnog razvoja. U cilju brže obrade učitanih datoteka i ekonomične upotrebe memorije poslužitelja, bit će opisani i predloženi načini korištenja radnih niti u okolini *Node.js* radi ostvarivanja paralelizma. Nadalje, s gledišta učinkovitosti pohrane i učitavanja podataka, bit će analizirane i uspoređene relacijske i nerelacijske baze podataka, te će se predložiti način korištenja nerelacijske baze podataka u praktičnom dijelu rada. Na temelju predloženih pristupa, arhitektura i modela programski će se ostvariti poslužiteljska strana aplikacije za potporu pri upravljanju projektima agilnog razvoja, a ostvarena aplikacija bit će ispitana za različite scenarije korištenja. Ovaj diplomski rad bavi se poslužiteljskom stranom sustava, dok diplomski radom pod nazivom „Programsko rješenje korisničkog dijela web sustava za potporu upravljanja projektima agilnog razvoja programske podrške prilagođenog osobama s oštećenjima vida“ razvija klijentski dio sustava.

Diplomski rad strukturiran je tako da se u drugom poglavlju daje prikaz stanja u agilnom razvoju programske podrške, njegova osnovna načela, kao i okviri koji se koriste unutar agilnog razvoja. Analizirane su i uspoređene mogućnosti postojećih alata i okolina za vođenje projekata. U trećem poglavlju opisana je i predložena arhitektura zasnovana na mikrouslugama. U četvrtom poglavlju opisuje se i predlaže korištenje radnih niti unutar okoline Node.js s ciljem ostvarivanja paralelizma prilikom sažimanja učitanih slikovnih datoteka. U petom poglavlju postavljeni su zahtjevi za poslužiteljski dio sustava te je opisano idejno rješenje za ispunjavanje zadanih zahtjeva. U šestom poglavlju prikazano je i objašnjeno programsko rješenje poslužiteljske strane sustava. Sedmo poglavlje sadrži provedbu i analizu ispitivanja aplikacije, kao i analizu korištene arhitekture zasnovane na mikrouslugama i analizu učinkovitosti ostvarene višenitnosti.

2. PRIKAZ STANJA U AGILNOM RAZVOJU PROGRAMSKE PODRŠKE

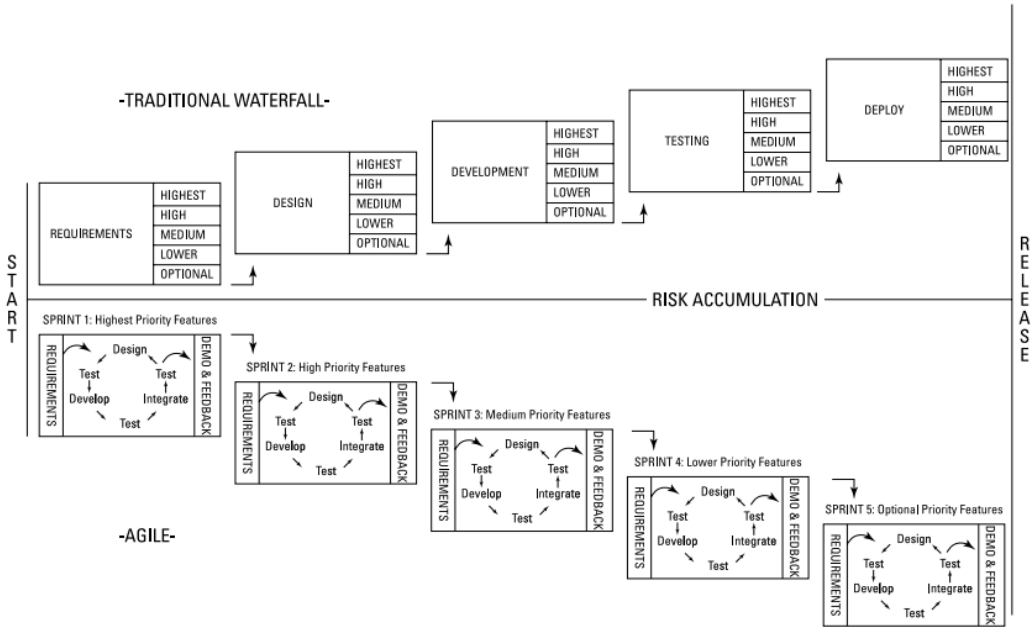
U ovom poglavlju je opisan razvoj programske podrške korištenjem agilnih metoda i prikaz trenutnog stanja razvoja programske podrške u praksi.

2.1. Agilni razvoj programske podrške

Agilni razvoj programske podrške odnosno agilno vođenje projekta je način vođenja projekta kojemu je naglasak dostavljanje proizvoda u što kraćem roku, te kontinuirano unapređenje proizvoda ali i njegovog razvojnog procesa. Ime je dobio po tome što se naglasak stavlja na brzom odgovoru na promjenu strategije umjesto da se slijedi prethodno stvoren plan što rezultira spremnim proizvodom prema klijentovim naputcima. Umjesto da se klijent nakon prikupljanja zahtjeva izolira od projekta, on je u Agilnom razvoju aktivno uključen u razvoj programske podrške.

Manifest, dokument kojeg je grupa programera i projektnih menadžera nazvala *Agile Manifesto*, sadrži sve vrijednosti koje agilni razvoj programske podrške treba vrednovati. Prema manifestu pojedinci i interakcije trebaju biti iznad procesa i alata, radna programska podrška treba biti iznad sveobuhvatne projektne dokumentacije, suradnja kupaca tijekom pregovora o ugovoru i tim treba biti podložan promjeni umjesto slijepo slijediti plan.

Za razliku od drugih načina vođenja projekta agilni način koristi empirijsku kontrolu projekta. To znači da: svi uključeni u projekt znaju njegovo trenutno stanje i napredak koji je postignut, često se vrše provjere samog proizvoda i tijeka razvoja istog, i da se projekt brzo adaptira promjenama kako bi se izbjegli problemi. Kako bi se postigle česte provjere stanja projekta razvoj projekta se dijeli u iteracije odnosno manje segmente. U svakoj iteraciji se ponavljaju isti koraci razvoja projekta kao kod drugih načina vođenja projekta: prikupljanje zahtjeva, dizajn proizvoda, razvoj proizvoda, dokumentacija razvoja i testiranje razvijenog proizvoda. Na slici 2.1 prikazana je razlika između agilnog i vodopadnog načina vođenja projekata [1].



Sl. 2.1 *Prikaz agilnog i vodopadnog načina vođenja projekata [1].*

U agilnom razvoju programske podrške najčešće su korišteni *Kanban* i *Scrum* okviri.

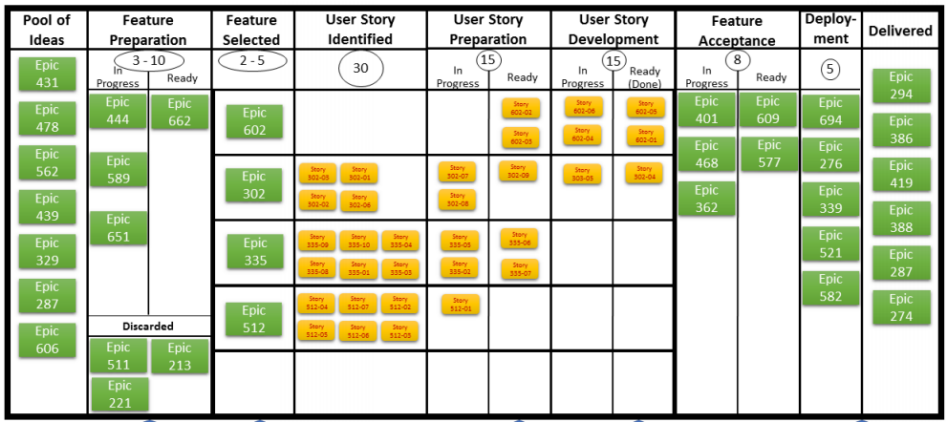
2.1.1. Razvojni okvir Scrum

Scrum je okvir za agilni razvoj programske podrške koji je razvijem devedesetih godina prošlog stoljeća. *Scrum* se temelji na empirizmu i jednostavnom razmišljanju. Empirizam je filozofski smjer koji koristi iskustvo kao osnovni izvor znanja, a jednostavno razmišljanje fokusira razvoj na važnije stvari. *Scrum* koristi iterativni pristup za optimizaciju predvidljivosti i kontrolu rizika. *Scrum* tim se sastoji od s*crum mastera*, *product ownera* i razvojnih inženjera. Unutar *Scruma* ne postoje kompliciranja s dodatnim timovima ili hijerarhijama. *Scrum* tim je kohezivan tim ljudi koji ima jedan cilj, a to je objavljivanje završenog produkta. Odnosno unutar tima se nalaze ljudi koji imaju dovoljnu stručnost da sami obavljaju zadatke i da sami odlučuju tko što radi i kada to radi. *Scrum* tim se sastoji od najviše deset ljudi, ali ta veličina je dovoljna da se obavi značajan posao unutar jednog *sprinta*. Preveliki timovi uzrokuju smanjenje komunikacije između članova, što automatski rezultira smanjenjem produktivnosti tima. *Scrum master* je zadužen da se tim pridržava pravila koja su definirana u *Scrum* priručniku. Oni su zaduženi za održavanje produktivnosti unutar *Scrum* tima tako što tim uče samoupravljanju i kako da svoj fokus stave na nadogradnje koje su visoke vrijednosti i koje su prethodno definirane. Osim tima, *scrum master* se brine da *product owner* efektivno definira ciljeve projekta i da efektivno upravlja *product backlogom*. Također pomaže *Scrum* timu da bolje definiraju unose u *product backlog. Product owner* se brine da se postigne najviše od *Scrum* tima i da se poveća vrijednost proizvoda. To se postiže tako da se dobro definiraju i iskomuniciraju zadani ciljevi za krajnji proizvod, pravljenje i

uređivanje stvari unutar *product backloga*. Razvojni inženjeri (eng. *developers*) su onaj dio *Scrum* tima koji je zadužen da napravi kvalitetno ažuriranje na kraju *sprinta* koje se može koristiti. Razvojni inženjeri su zaduženi da definiraju i prilagođavaju plan izvođenja *sprinta* kako bi se ispunili ciljevi postavljeni u planu. Svaki dan se izvodi petnaest minutni *Scrum* sastanak gdje se prilagođavaju ciljevi zadani na početku *sprinta* kako bi se tim prilagodio novonastalim problemima ili zahtjevima na proizvod. Na kraju svakog *sprinta* se radi revizija proteklog *sprinta* gdje se proizvod koji je nastao na kraju *sprinta* pokazuje svim sudionicima u projektu i gdje se diskutira o postignućima koja su postignuta i napreduje li dovoljno tim da se postignu svi postavljeni ciljevi. Na tome sastanku se također razgovara o povećanju kvalitete i efektivnosti tima odnosno što je bilo dobro, a gdje su bili problemi i mogu li se oni otkloniti ili izbjeći u idućem *sprintu* [2]. Planiranje novog *sprinta* trebalo bi trajati manje od osam sati, a dužina trajanja *sprinta* ne smije prelaziti četiri tjedna [30].

2.1.2. Kanban okvir

Kanban potječe iz tvrtke *Toyota* koja je razvila *Kanban* kako bi unaprijedila efikasnost svoga proizvodnog pogona i smanjila otpade. *Kanban* je postao učinkovit okvir za podršku pri vođenju proizvodnog sustava u cjelini. Mogu se lako identificirati vrijeme izvođenja i vrijeme ciklusa određenog procesa kao i njegovih potprocesa, ali se mogu i pronaći njihove nekompatibilnosti. Za razliku od *Scrum* okvira, *Kanban* okvir se bazira na vizualnim karticama i ploči gdje se te kartice slažu. Kartice se kreću s lijeva na desno kako bi se olakšalo timovima da koordiniraju svoje zadatke, ali i da lakše vizualiziraju napredak svih zadataka zadanih u projektu. Obično se ploča dijeli na tri stupca: *čekanje* (eng. *to-do*), *radi se* (eng. *in progress*), *završeno* (eng. *completed*). Stupac *radi se* može se dodatno podijeliti u više grupa kako bi se olakšala vizualizacija napretka. U programskom inženjerstvu se obično taj stupac dodatno dijeli u grupe: *analiza*, *dizajn*, *razvoj*, *testiranje* i *isporuka*. Na slici 2.2 prikazan je izgled *Kanban* ploče [3].



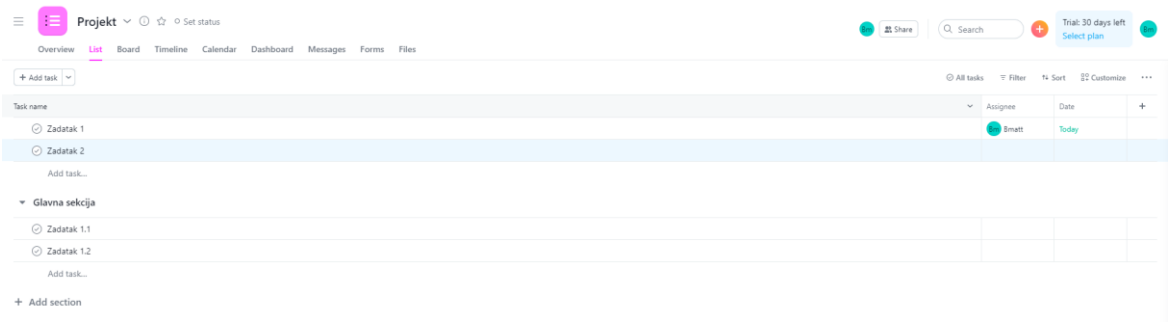
Sl. 2.2 *Prikaz Kanban ploče sa zadatcima [3].*

Glavni cilj *Kanbana* je da je cijeli razvoj projekta vizualiziran i da je razumljiv svima koji sudjeluju u njemu. *Kanban* nije nužno napravljen za razvoj programske podrške već i za druge poslove. Osim kod razvoja programske podrške *Kanban* se često koristi i u prodaji, marketingu, ali i kod upravljanja ljudskim resursima. Za vođenje projekata razvoja programske podrške najčešće se koriste programi i web aplikacije koje mogu putem Interneta vizualizirati svim članovima projekta pružiti uvid u njegov razvoj. Kako bi se podigla produktivnost razvojnog tima cilj je da tim obradi što više *Kanban* kartica, odnosno da ih iz stupca *čekanja* prebaci u stupac *završeno* i da vrijeme od kada se kartica stavi na ploču do njenog obrađenog stanja bude što kraće. Osim toga, potrebno je podijeliti jedan zadatak na više kartica koje su ograničene u stanju *razvijanje* kako bi se inkrementalno razvijalo rješenje i kako bi ono bilo što kvalitetnije. Također, vrlo je važno razviti i petlje povratnih informacija koji će davati povratne informacije iz *Kanban* procesa (makro razina) i njegovih pridruženih koraka (mikro razina), a koje se mogu iskoristiti za kontinuirano poboljšanje razvojnog procesa. Kako bi se povećala produktivnost razvojnog tima donose se *Kanban* pravila koja se mogu vidjeti na *Kanban* ploči. Njima se omogućuje da razvojni tim bude zaokupljen pravilnim aktivnostima koje će biti točno napravljene i u zadanom vremenskom roku. A projektni menadžer se mora potruditi da se ta pravila poštuju tijekom razvojnog procesa. Neka od tih pravila su: definiranje kada je zadatak gotov, kako rukovati sa zaostacima i koje aktivnosti treba prioritizirati i kako, kako ukomponirati nove zahtjeve u trenutni projekt i slično [3].

2.2. Programski sustavi za vođenje projekata u agilnom načinu

2.2.1. Asana

Asana je programska podrška kao usluga (eng. *service-as-a-service*) koja je dizajnirana za praćenje i organizaciju projekata. Njoj se može pristupiti putem web ili mobilne aplikacije. *Asana* se može koristiti za sve projekte, a ne samo za projekte razvoja programske podrške. *Asana* timovima nudi vođenje projekta pomoću okvira *Scrum* i *Kanban*. Prilikom kreiranja projekta korisnik može odabrati jedan od četiri izgleda projekta: *lista*, *vremenska crta*, *ploča zadataka*, *kalendar*. U bilo kojem trenutku korisnik može promijeniti izgled projekta gdje može vidjeti sve zadatke, zadane korisnike i krajnji rok za taj zadatak. Na slici 2.3 prikazan je izgled projekta kao liste [4].

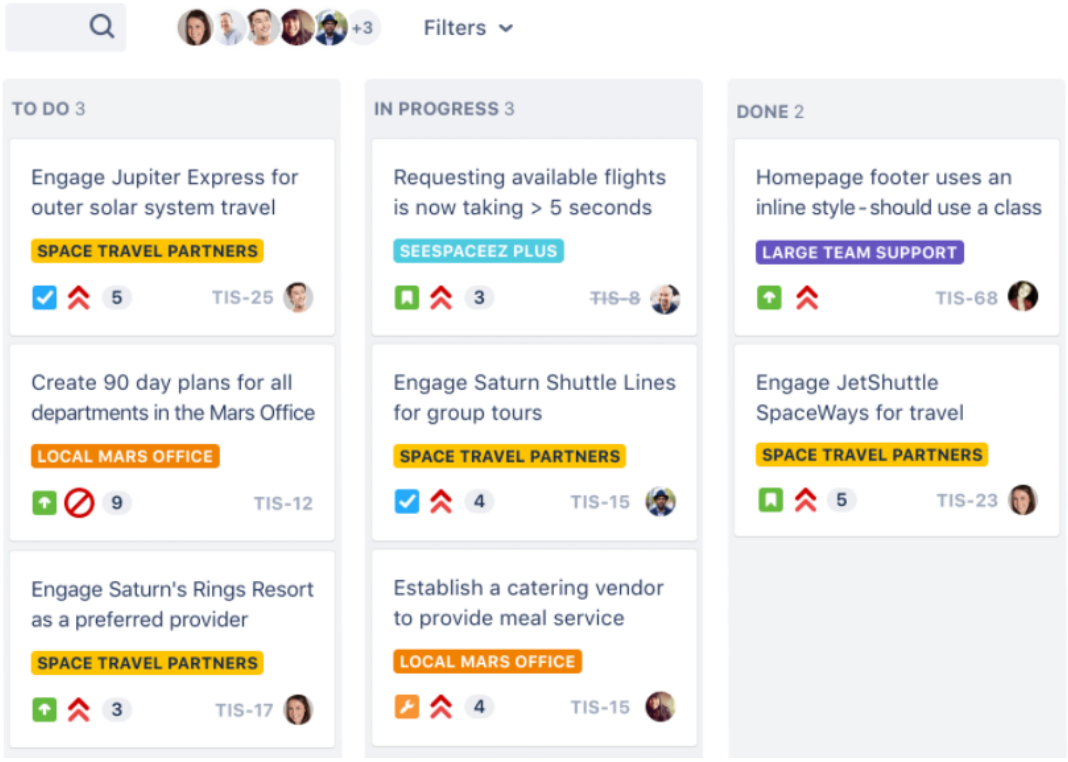


Sl. 2.3 *Prikaz izgleda projekta kao liste na Asani [4].*

Nakon postavljenog projekta, vlasnik projekta može dodati nove članove i dodavati ili mijenjati im rangove. Osim toga može dodati ili izmijeniti miljokaze i ciljeve projekta, ali i trenutni status projekta. Vlasnik projekta ili projekt manager može automatizirati određene stvari na projektu poput dodjeljivanja zadataka članovima tima, mijenjanje statusa zadatka, brisanje zadataka kada se određeni uvjeti zadovolje i slično. Članovi tima na projektu mogu pisati poruke u odjeljku za dopisivanje. Osim toga članovi tima mogu učitavati datoteke koje su važne za projekt. Također se *Asana* projekt može povezati s različitim aplikacijama poput *Zooma* gdje se može implementirati *Zoom* poziv s određenim zadatkom unutar projekta [4].

2.2.2. Jira

Jira je programska podrška koja se koristi za praćenje problema unutar programa i za vođenje projekata koji se razvijaju na agilni način. *Jira* omogućava timovima da planiraju detaljan razvoj programske podrške putem pisanja korisničkih priča i problema (eng. *issues*), planiranje *sprintova* i dodjeljivanje zadataka članovima tima. Uz pomoć *Jire* korisnik može koristiti *Scrum* ili *Kanban* okvir. Korištenjem *Scrum* okvira Jira ujedinjuje timove oko jednog cilja i promovira iterativnu i inkrementalnu isporuku programske podrške. *Scrum* ploča se koristi kako bi korisnici vizualizirali zadatke i sav posao u trenutnom razvojnom *sprintu*. *Jira* omogućuje korisnicima da prilagode *Scrum* ploču svojim potrebama. Na kraju svakog razvojnog *sprinta* korisnici mogu vidjeti sve zadatke koji su riješeni ili ipak nisu riješeni. Ne riješeni zadaci se automatski prebacuju u *backlog* kako bi se oni riješili u sljedećem razvojnom *sprintu* [5]. Korisnici mogu lako konfigurirati upravljačke ploče (eng. *dashboards*) i načine kako bi mogli lakše pratiti prioritete i kako bi se dobio stvaran uvid u trenutni napredak projekta u razvoju [31]. Osim toga *Jira* omogućava izvještavanje o trenutnom stanju projekta i njegovom napretku pomoću raznih agilnih izvještaja kao na primjer *burndown chart* (graf koji prikazuje količinu posla kojeg članovi tima obavlja u trenutnom projektu) [5]. Na slici 2.4 prikazan je izgled *Scrum* ploče unutar *Jire* [5].

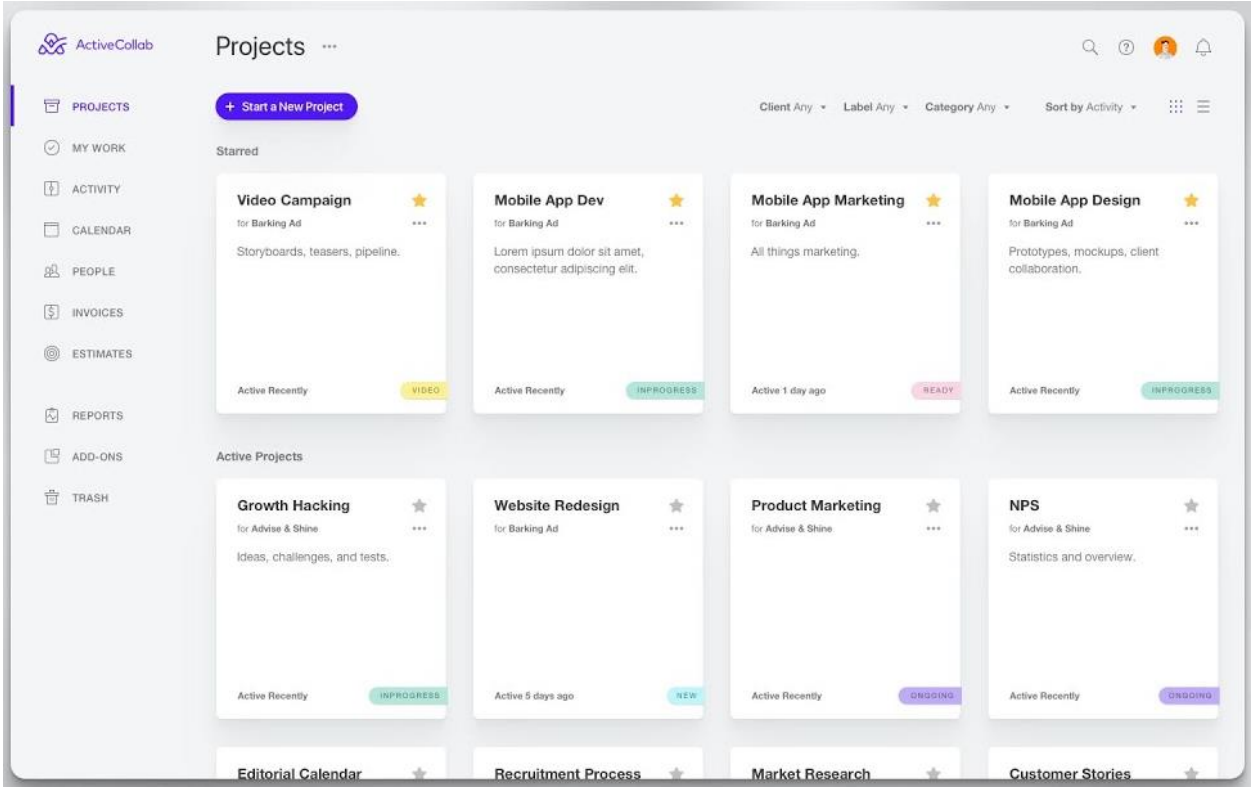


Sl. 2.4 *Prikaz izgleda Scrum ploče unutar Jire [5].*

Kanban ploče daju timovima potpuni uvid u razvoj projekta. Jedna *Kanban* kartica predstavlja zadatak koji kada se završi odmah se nastavlja na sljedeći zadatak. Klikom na karticu prikazuju se detalji o zadatku. *Jira* omogućuje ograničavanje broja kartica unutar stupca koji označava zadatke koji se trenutno razvijaju. Kao i kod *Scrum* ploča *Jira* omogućava izvješćivanje tima o napretku projekta. Osim *Kanban* i *Scrum* okvira korisnik ih može kombinirati u *Scrumban* i *Kanplan* okvire [5].

2.2.3. Active Collab

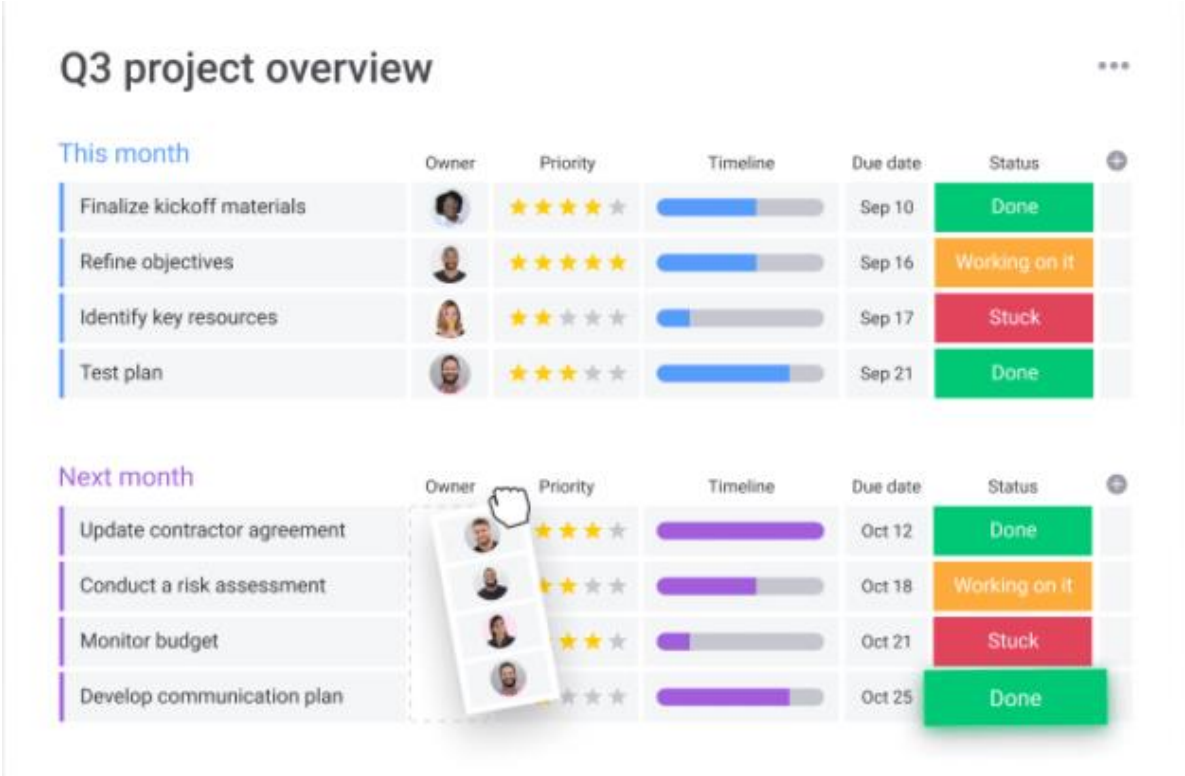
Active Collab je web aplikacija za upravljanje projektima dostupna kao program dostupan za vlastito posluživanje ili kao usluga na oblaku. Nudi usluge: upravljanje zadacima, fakturiranja i praćenja vremena, dijeljenje datoteka unutar zadataka, komunikaciju razvojnog tima i klijenata unutar projekta i podsjetnike. Projekti su prikazani pomoću *Kanban* ploče, a podijeljeni su u listu zadataka, zadatke i pod zadatke. Osim vođenja projekta *Active Collab* omogućuje predviđanje potrošnje budžeta projekta, izdavanje računa i naplaćivanje troškova razvoja projekta [6]. Na slici 2.5 prikazan je izgled vođenja projekta pomoću *Active Collab* aplikacije [6].



Sl. 2.5 *Prikaz izgleda vođenja projekta pomoću Active Collab aplikacije [6].*

2.2.4. Monday

Monday.com je platforma dostupna kao web i mobilna aplikacija za upravljanje projektima [7]. Cilj je timovima povećati produktivnost i suradnju pomoću praćenja projekata i vizualizacije dobivenih podataka. Aplikacija je intuitivna za korištenje bez potrebe za dodatnim treninzima. Unutar svakog projekta se mogu integrirati aplikacije koje olakšavaju komunikaciju unutar tima i praćenje razvoja projekta. Projekt se vizualno reprezentira na dva načina pomoću *Kanban* ploče ili tablice zadataka. Prilikom pravljenja projekta korisnik može odabrati jedan od dvjestotinjak ponuđenih predložaka vođenja projekta prema koji su raspoređeni u kategorije. Aplikacija će zatim postaviti projekt prema predlošku i integrirati aplikacije koje se koriste u istom. Sve unutar projektne ploče je interaktivno, korisnik može dodati ili brisati stupce unutar tablice, pomicati iste ili mijenjati podatke unutar tablice. Svaka interakcija unutar projekta se ažurira unutar stvarnog vremena i svi korisnici mogu odmah vidjeti izmjene. Trenutno stanje razvoja pojedinog zadatka se prikazuje pomoću status gumba. Korisnik može poslati obavijest drugim članovima tima, učitavati datoteke ili povezivati različite ploče zajedno kako bi se lakše prikazao status razvoja projekta. Na slici 2.6. prikazan je izgled vođenja projekta pomoću *Monday.com* aplikacije.



Sl. 2.6 *Prikaz vođenja projekta pomoću Monday.com aplikacije [7].*

Projekt se može vizualno prikazati još s kalendarom, vremenskom crtom, Ganttovim dijagramom, mapama, različitim formama i ostalim pogledima. Aplikacija će automatski projektne podatke transformirati i prikazati u odabranom pogledu. Pomoću vremenske crte i Ganttovog dijagrama korisnik može vidjeti uspješnost stizanja zadanih rokova. *Monday.com* omogućuje automatizaciju upravljanja projektima. Aplikacija može automatski: slati obavijestiti sve članove tima, ažurirati statuse zadataka, dodjeljivanje zadužene članove tima za pojedine zadatke, otvarati nove zadatke, itd. Integracijom ponuđenih alata članovi tima mogu lako povezati podatke potrebne za razvoj projekta s drugim članovima tima jer su vidljivi na *Monday.com* aplikaciji gdje korisnik može učitati nove verzije podatka ili spremiti ga na vlastito računalo. Tim može odabrati jednu od ponuđenih aplikacija u *Monday.com* trgovini ili pak stvoriti vlastitu aplikaciju koju zatim može integrirati u svoj projekt. Svi podatci su spremljeni na oblaku, a napredna tražilica korisnicima omogućava pretraživanje podataka prema tagovima, ljudima ili gdje je korisnik označen [7].

2.3. Usporedba programskih sustava za vođenje projekata u agilnom razvoju

U nastavku će se usporediti dva najpopularnija programska sustava za vođenje projekata u agilnom razvoju: *Asana* i *Monday.com*. Glavne funkcionalnosti oba programska sustava su iste, ali su krajnjem korisniku predstavljene na različite načine.

Kod *Asane* korisnik može: kreirati nove zadatke i dodijeliti ih članovima tima, postaviti rok izvršenja zadatka i učiniti ih ovisnima o drugim zadacima. Prilikom odabira zadatka prikazuju se njegovi detalji među kojima su i komentari na odabrani zadatak gdje se mogu učitati dokumenti vezani za isti ili se mogu označiti drugi članovi tima. Nakon što se korisnik prijavi u *Asanu* on će imati popis svih zadataka kojima se rok izvršenja bliži, a ako odabere odjeljak *my tasks* moći će vidjeti sve zadatke koji su mu dodijeljeni. Kod *Monday.com* zadaci se nazivaju *pulsi* koji se ne prikazuju nužno kao kartice (što je slučaj kod *Asane*). Korisnik *pulsima* može: dodijeliti ime, dodijeliti ih članovima tima, postaviti njihov trenutni status, postaviti ovisnost o drugim *pulsima*, pogledati pregled vremena do završetka s trakom napretka.

Tokovi rada u *Asani* se kategoriziraju različitim pogledima kojima se može vizualizirati trenutni napredak razvoja projekta, a to su: liste, *Kanban* ploča, Ganttov dijagram i kalendar. U većim projektima ovisnost zadatka o drugim zadacima pomaže da projekt ne ide previše ispred u odnosu na druge. Ovisnost zadatka o drugim zadacima se može korigirati pomoću pomicanja zadatka ispred ili iza drugih zadataka. Za razliku od *Asasne*, *Monday.com* omogućava laku vizualizaciju svojih *pulseva* bez potrebe za promjenom prozora ili kartica u pregledniku. Početni pogled je lista svih *pulseva* gdje korisnik može vidjeti sve osnovne informacije na jednom mjestu, a klikom na gumbove za promjenu pogleda automatski i instantno se mijenja način vizualizacije informacija projekta. Informacije o projektu se mogu prikazati pomoću: liste, *Kanban* ploče, Ganttovog dijagrama, mapa i grafova.

Asana i *Monday.com* imaju sustav portfelja za projekte gdje se može vidjeti šira slika trenutnog stanja razvojnog tima, a ne samo određenog projekta. Kod *Asane* taj sustav je napravljen tako da su svi projekti prikazani na jednom mjestu i korisnik može vidjeti osnovne informacije o pojedinom projektu poput: naziva, trenutnog statusa, trenutni napredak, početak i završetak razvoja, prioritet i koji je član tima zadužen za taj projekt. Dok kod *Monday.com* taj sustav se naziva grupe gdje se svaki *puls* može pridružiti određenoj grupi. Tako korisnik može birati hoće li u grupi gledati projekte ili zadatke (pulseve) ovisno o tome što je pridodano grupi.

Asana omogućuje integraciju projekta s preko stotinjak aplikacija poput Dropboxa, Slacka i *GitHuba* koje mogu povećati produktivnost razvojnog tima. Dok kod *Monday.com* projekti se mogu povezati sa svega četrdesetak aplikacija, ali *Monday.com* omogućava razvoj vlastitih integracija pomoću svoga *API*-ja što omogućava veću slobodu kod razvojnih timova [8].

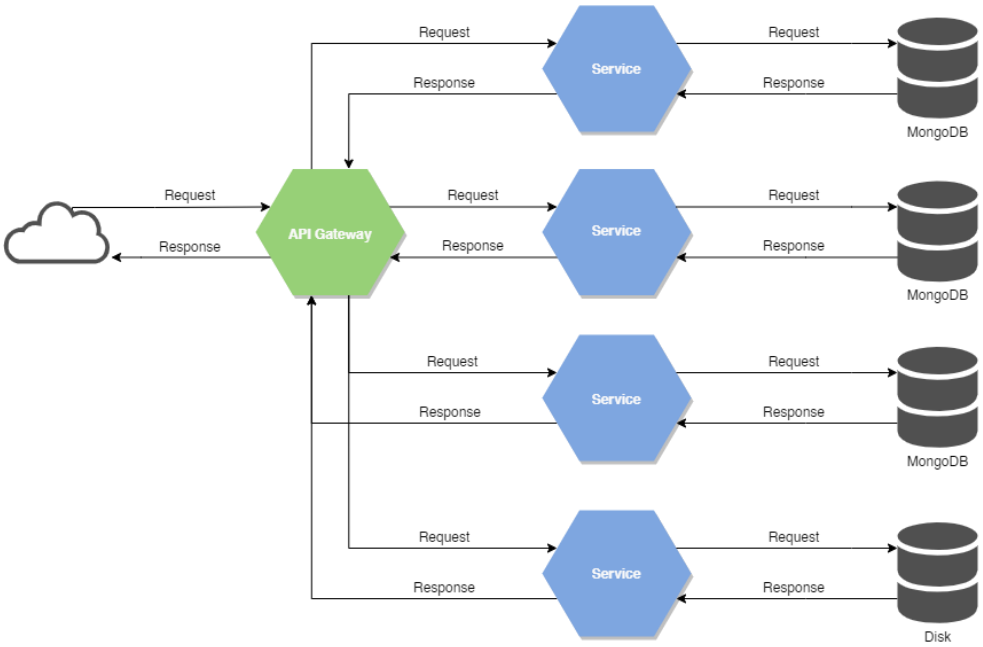
3. ARHITEKTURA MIKROUSLUGA

U ovome poglavlju će se opisati arhitektura mikrousluga, modeliranje pojedine usluge u arhitekturi koristeći *API Gateway*, osim toga će se navesti i primjeri korištenja tehnologije mikrousluga u praksi.

3.1. Arhitektura temeljena na mikrouslugama i stanje u području

Arhitektura temeljena na mikrouslugama je način izgradnje programske podrške sa skupom malih samoodrživih procesa. Iako ju je moguće koristiti i na korisničkom dijelu aplikacije (eng. *frontend*), arhitektura temeljena na mikrouslugama je namijenjena za poslužiteljski dio aplikacije. Svaka mikrousluga se razvija neovisno o drugim mikrouslugama i strogo su usmjerene na jednu stvar i njeno izvršavanje što znači da mikrousluge nastoje imati slabije međusobne veze i veću kohezivnost [9]. Svaka mikrousluga obuhvaća samo jednu poslovnu funkcionalnost [33]. Odnosno svaka mikrousluga treba imati zaseban model podataka s kojim radi i zasebnu logiku. Osim toga se svaka mikrousluga može temeljiti na različitim tehnologijama spremanja podataka pa čak i različitim programskim jezicima. Iako trebaju biti neovisne o drugim mikrouslugama one trebaju komunicirati s drugim procesima (*Gateway API*, baza podataka itd.), a ta komunikacija se odvija putem komunikacijskih protokola poput *HTTP/HTTPS-a, WebSocketa* ili *AMQP-a*.

Za razliku od monolitne arhitekture, arhitektura temeljena na mikrouslugama pruža dugoročnu agilnost projekta jer u velikim i kompleksnim programima omogućava proširenje funkcionalnosti programa uz dodavanje nove mikrousluge koja je neovisna od drugih mikrousluga, kao što je prikazano na slici 3.1.



Sl. 3.1 *Prikaz programske podrške pomoću arhitekture temeljene na mikrouslugama.*

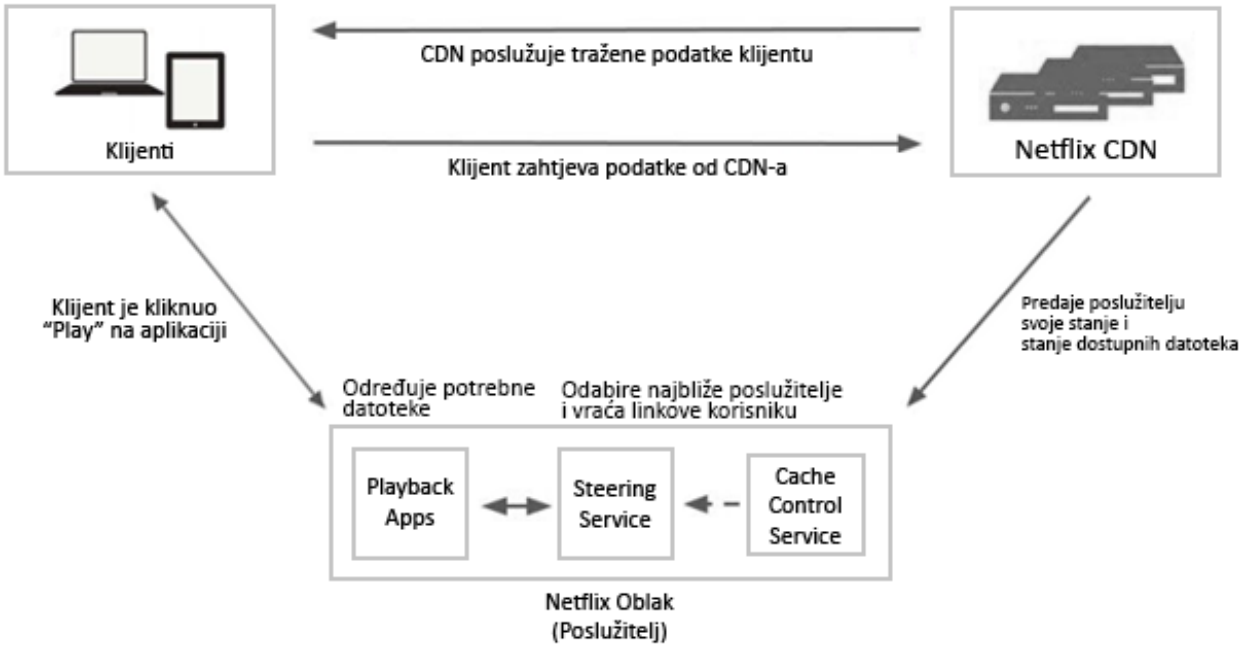
Također, s obzirom da su mikrousluge međusobno nezavisne, svaku je moguće mijenjati bez utjecaja na druge usluge u vrlo kratkom vremenu. Osim toga, arhitektura temeljena na mikrouslugama omogućava veliku skalabilnost uz pomoć *Docker* kontejnera ili možda pokretanja programa na različitim poslužiteljima. Umjesto da se cijela aplikacija proširi kao cjelina, ovako se prema potrebi proširuje ili smanjuje zasebna mikrousluga [9].

Skalabilnost je omogućena, jer za razliku od tradicionalnog pristupa programiranju programske podrške, u arhitekturi temeljenoj na mikrouslugama se važnost pridaje na suverenosti podataka prema mikrouslugama. Svaka mikrousluga mora biti jedna cjelina koja ima vlastitu domenu podataka i vlastitu logiku obrade tih podataka koja se vrši unutar autonomnog životnog ciklusa, ali pritom mora biti neovisna od drugih mikrousluga. Autonomnost se osigurava enkapsulacijom podataka koje pojedina mikrousluga obrađuje. U tradicionalnom pristupu programiranja programske podrške postoji jedna ili nekoliko baza podataka. Ako se taj pristup iskoristi na programsku podršku programiranu arhitekturom temeljenoj na mikrouslugama tada nekoliko mikrousluga može pristupiti jednom podatku u isto vrijeme čime se gubi njegov integritet (jer nije moguće svim uslugama u kratko vrijeme pružiti ažurirani podatak) i autonomnost samih mikrousluga. Stoga se u arhitekturi mikrousluga koriste različite baze podataka za svaku mikrouslugu [10].

3.1.1. Trenutno korištenje arhitekture temeljene na mikrouslugama u praksi – Netflix

Najbolji primjer korištenja arhitekture temeljene na mikrouslugama u praksi je *Netflix*. Iako prikazani primjer nema vezu uz zadatak rada on je uzet kao pokazni primjer korištenja arhitekture temeljene na mikrouslugama u praksi. *Netflix* je jedan od pokretača koji stoje iza arhitekture zasnovane na mikrouslugama. Netflix je transformaciju svoje streaming platforme započeo 2008. godine kada je jedan program temeljen na monolitnoj arhitekturi prebacio u skup mikrousluga poslužen na *AWS* oblaku. Netflixov streaming sustav se sastoji od: korisničke aplikacije, pozadinska aplikacije (eng. *backend*) i mreže za isporuku i distribuciju sadržaja (*CDN*-a). Korisnička aplikacija je *Netflix* aplikacija koja se pokreće na bilo kojem Internet pretraživaču ili pak mobilna aplikacija koja se može pokrenuti na *iOS-u* ili *Androidu.* Kontroliranjem vlastitih aplikacija mogu prilagoditi svoju uslugu prema bilo kojem korisniku na temelju različitih okolnosti. Pozadinska aplikacija se sastoji od usluga, baza podataka i spremnika podataka koji se nalaze na *AWS*-ovoj platformi. Mreža za isporuku i distribuciju sadržaja se sastoji od nekoliko poslužitelja koji su optimizirani za spremanje i prikaz velikih video datoteka. Ti poslužitelji ne nalaze se u standardnim podatkovnim centrima nego se nalaze unutar pružatelja internetskih usluga (eng. *internet service providers*).

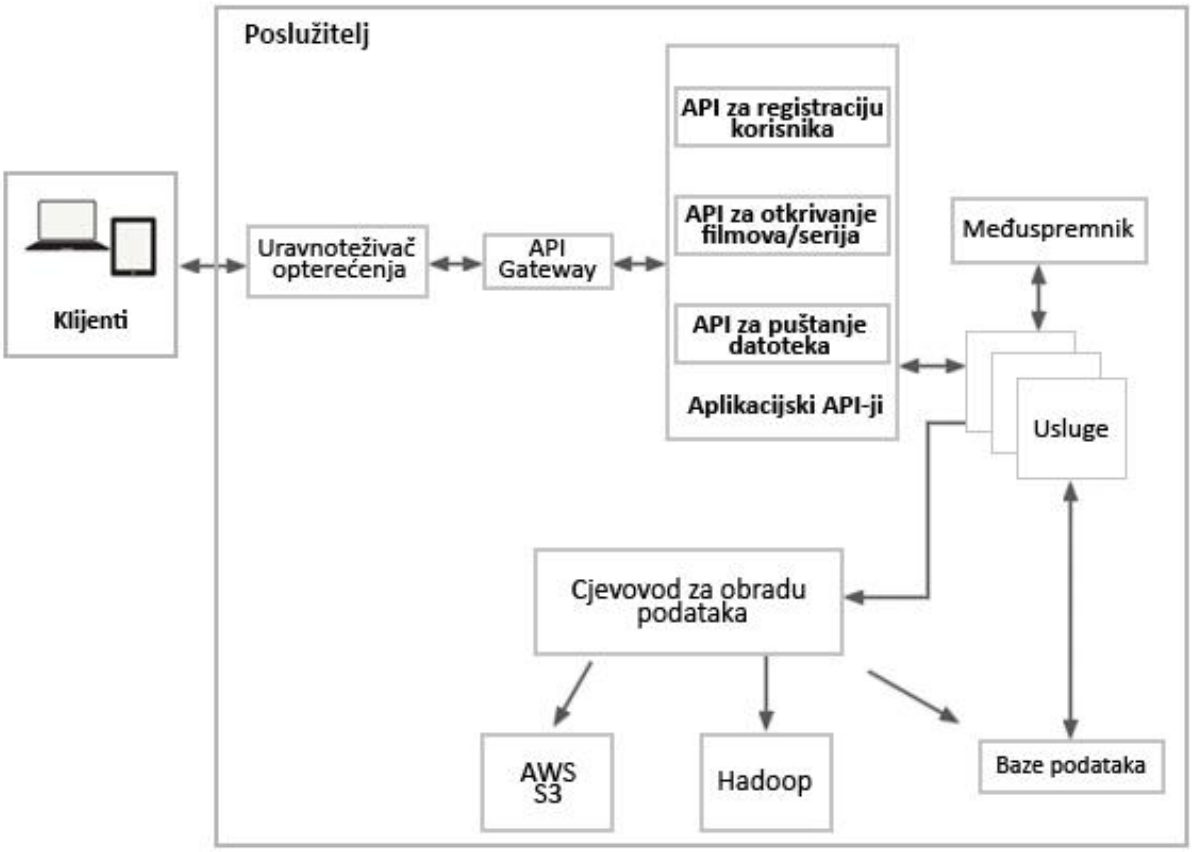
Arhitektura reprodukcije (eng. *playback architecture*) je programska podrška koja se aktivira kada pretplatitelj odabere video datoteku i pritisne gumb za njeno puštanje. Puštanje video datoteke se odvija tako da korisnik pošalje zahtjev pozadinskom programu na oblaku i *Netflixovom CDN*-u. Slika 3.2 napravljenoj po uzoru na [11], prikazuje dijagram toka procesa puštanja odabrane video datoteke pretplatitelju na *Netflixu*.



Sl. 3.2 *Prikaz dijagram toka procesa puštanja odabrane video datoteke pretplatitelju [11].*

Netflixov sustav za isporuku i distribuciju sadržaja konstantno šalje izvještaje o stanju video datoteka koje se nalaze u priručnoj memoriji kako bi se korisniku prikazale najsvježije datoteke. Nakon toga korisnik pokreće prikazivanje odabrane video datoteke na svome uređaju, te se šalje zahtjev *Netflixovom* oblaku za *URL* video datoteke kako bi se ona prikazala korisniku. Usluga pod nazivom *Playback Apps* validira korisnikov zahtjev kako bi se provjerilo smije li korisnik pogledati odabranu video datoteku (korisnik može biti u različitom platnom razredu ili drugoj zemlji gdje se ne prikazuje odabrana video datoteka itd.). Kada se korisnikov zahtjev validira, tada se kontaktira usluga pod nazivom *Steering service*. Koja na temelju korisnikove IP adrese pretražuje slobodne *Netflixove CDN*-ove koji su korisniku najbliži i koji će što bolje prikazati odabranu video datoteku korisniku. Zatim se lista slobodnih *CDN*-ova šalje natrag korisniku koji tada odabire najboljeg za sebe i odabranom *CDN*-u šalje zahtjev za prikaz odabrane video datoteke. Tada odabrani *CDN* poslužitelj prihvaća zahtjev i pruža korisniku odabranu video datoteku.

Pozadinska programska podrška vodi računa o svemu što je vezano uz *Netflix* platformu. Od prijavljivanja i registracije do naplate sadržaja i transkodiranja videa. Za odrađivanje svih tih zadataka *Netflix* koristi arhitekturu temeljenu na mikrouslugama koji se odrađuju na *Amazonovom oblaku računala* (*AWS*). Na slici 3.3 napravljenoj po uzoru na [11], prikazan je dijagram toka procesa obrade klijentovog zahtjeva za puštanje video datoteke.



Sl. 3.3 *Prikaz dijagrama toka procesa obrade klijentovog zahtjeva za puštanje video datoteke [11].*

Kada klijent putem uređaja odabere video datoteku za puštanje šalje zahtjev *Netflixovom* poslužitelju koji se obrađuje od strane *AWS*-ovog uravnoteživača opterećenja (eng. *load balancer, ELB*). Nakon obrade ELB zahtjev prosljeđuje ka *API Gateway* usluzi pod nazivom *Zuul* koja omogućava dinamično prosljeđivanje zahtjeva ka mikrousluzi, nadzor prometa i sigurnost na samome oblaku. U *API Gatewayu* se zahtjev dodatno obrađuje i filtrira prema zadanim pravilima i prosljeđuje aplikacijskom *API*-ju koji se sastoji od nekoliko različitih *API*-ja poput: prijave i registracije korisnika, istraživanje novih video datoteka, predlaganje korisniku određene video datoteke i puštanje video datoteka korisniku. Ovisno o zahtjevu koji dolazi s korisničke aplikacije koristi se pojedini *API*. U ovome slučaju je to *Play API*, odnosno *API* za reprodukciju video datoteka. Tada *Play API* poziva jednu ili lanac mikrousluga kako bi se na korisnikov zahtjev poslužitelj poslao odgovor. Mikrousluge u *Netflixovoj* arhitekturi su mali programi koji ne pamte stanja (eng. *stateless*), ali koji mogu pozivati jedan drugog. Kako bi se spriječili nagli ispadi iz rada i omogućila prilagodljivost sustava, sve mikrousluge su izolirane od procesa koji ih je pozvao putem biblioteke *Hystrix*. Rezultat poziva mikrousluga se može spremiti u pričuvnoj memoriji kako bi se mogli istom ili drugom korisniku brže dostaviti pri ponovnom upitu. Također svaka mikrousluga u arhitekturi može dohvatiti ili spremiti podatke u bazama podataka. Osim toga mogu

poslati različite događaje cjevovodu za obradu tokova podataka (eng. *stream processing pipeline*) kako bi se odradile razne statistike koje mogu pomoći pri poboljšanju poslovanja tvrtke, a koji se spremaju u spremnike podataka poput *Cassandra*, *AWS S3, Hadoop HDFS,* itd. Nakon završetka svih obrada korisniku se vraća rezultat koji je već opisan slikom 3.2.

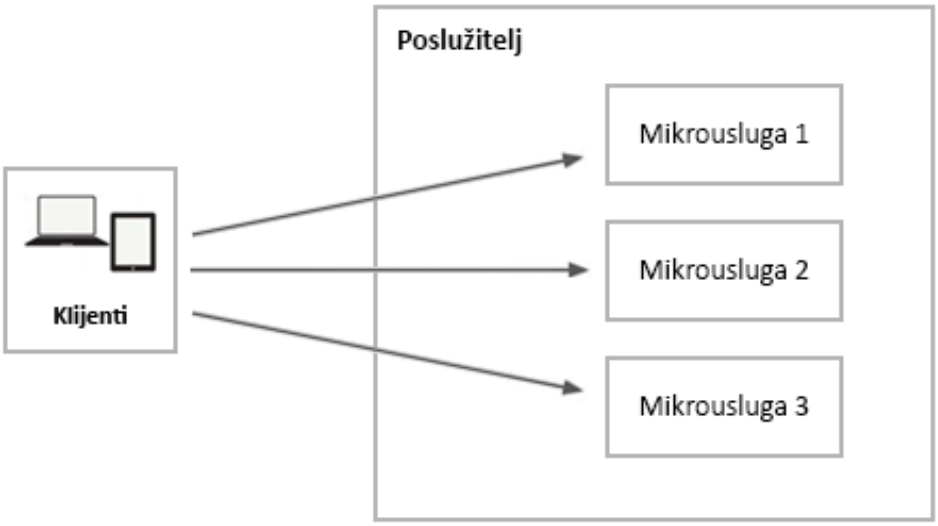
Kako *Application API* može pozvati jednu ili više mikrousluga, one komuniciraju između sebe putem *REST*-a ili *gRPC* biblioteke. *gRPC* biblioteka je sustav pozivanje udaljenih procedura. Svi zahtjevi prema mikrouslugama se spremaju u petlju mrežnih događaja (eng. *network event loop*), a rezultati drugih mikrousluga se asinkrono spremaju u red rezultata (eng. *result queue*) kako bi se na neblokirajući način moglo pristupiti određenim rezultatima i obraditi ih.

Svaka mikrousluga ima pristup zasebni način spremanja podataka, oni se (ovisno o potrebi) spremaju putem *SQL* i *noSQL* baza podataka. Relacijska baza podataka pogonjena *MySQL*-om se koristi za relacijske podatke poput informacija o pojedinim video sadržajima, podacima o korisnicima itd. *Hadoop* se koristi kako bi se spremili veliki tok podataka koji se dobije obično iz zapisnika unutar korisničke aplikacije ili poslužitelja. *ElasticSearch* se koristi prilikom pretraživanja svih sadržaja koji su dostupni na korisničkoj aplikaciji, a *Cassandra* se koristi kako bi se spremio veliki broj zahtjeva prema poslužitelju na siguran i brz način [11].

3.2. Modeliranje usluga

3.2.1. Izravno spajanje klijenta i mikrousluga

Svakoj mikrousluzi može se pristupiti putem krajnje točke (eng. *endpoint*) koja je unikatna za svaku mikrouslugu. Jedan od načina pristupa mikrousluzi je izravno spajanje klijenta i mikrousluge. U toj arhitekturi svaka mikrousluga ima javno dostupne krajnje točke koje mogu imati različite *TCP* portove, a kojima klijent može slobodno pristupiti. Na slici 3.4 napravljenoj po uzoru na [9], prikazan je način izravnog spajanja klijenta i mikrousluga.



Sl. 3.4 *Prikaz arhitekture izravnog spajanja klijenta i mikrousluga [9].*

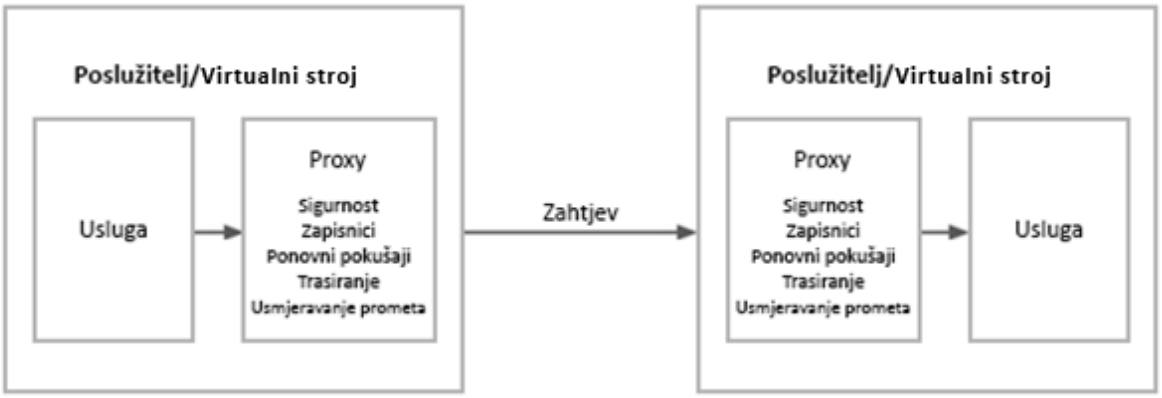
U produkcijskom okruženju gdje su mikrousluge podijeljene u nakupine (eng. *clusters*), krajnja točka ne vodi do mikrousluge nego do sustava za uravnoteženje opterećenja (eng. *load balancing*) za odabrani nakupine. U nekim slučajevima se može koristiti kontroler za dostavljanje aplikacija (eng. *application delivery controller*) koji se nalazi između klijenta i mikrousluga na poslužitelju, a osigurava sigurnost pri prijenosu zahtjeva i odgovora kriptiranjem istih putem *SSL*-a.

Arhitektura izravnog spajanja klijenta i mikrousluga se koristi u projektima koji su mali. Kada se klijent izravno spaja na pojedinu mikrouslugu tada se treba ograničiti broj zahtjeva prema poslužitelju kako bi se smanjila latencija i kompleksnost korisničkog sučelja. Poslužitelj bi trebao paralelno obraditi zahtjeve prema svim mikrouslugama i na efikasan način pridružiti njihove rezultate i vraća samo jedan rezultat klijentu. Također kompleksnost programske podrške na poslužitelju (koja koristi ovaj način programiranja mikrousluga) se dodatno povećava kada se žele implementirati validaciju i autorizacija svih zahtjeva i rezultata jer se mora za svaku mikrouslugu treba pisati kod za validaciju i autorizaciju zahtjeva i rezultata [9].

3.2.2. Mreža usluga

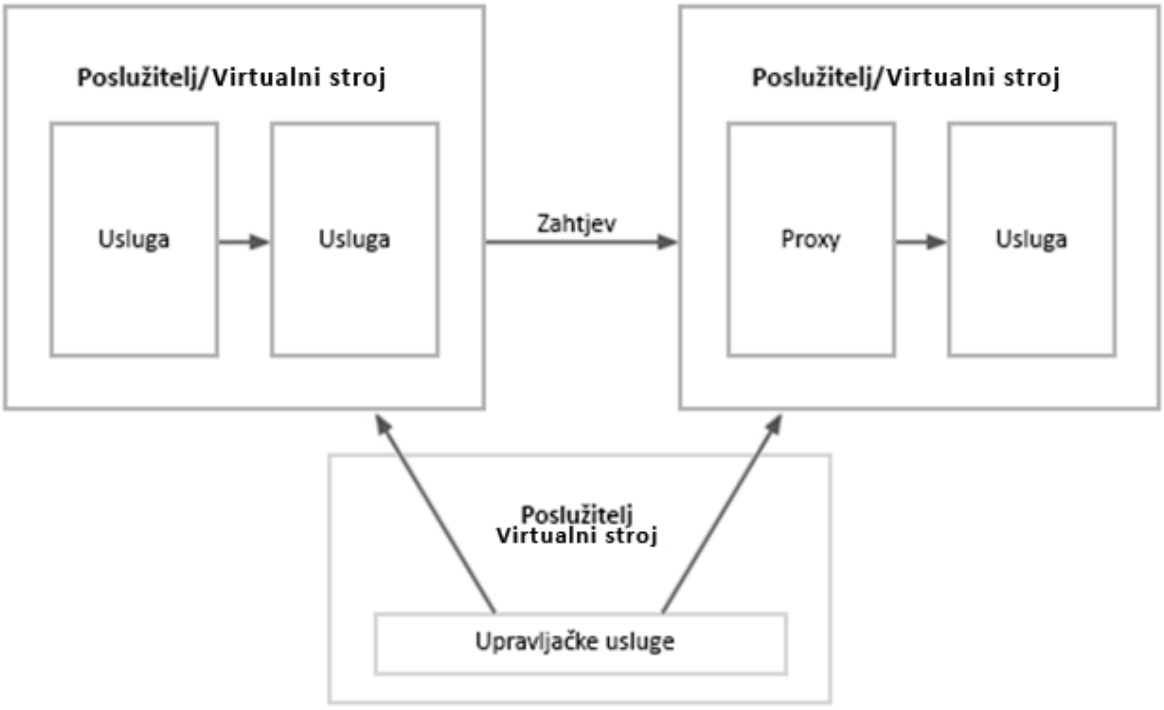
Mreža usluga je uzorak modeliranja mikrousluga gdje jedna mikrousluga šalje zahtjeve drugoj mikrousluzi. Za razliku od izravnog spajanja klijenta i mikrousluga gdje se za svaku mikrouslugu mora zasebno programirati validacija i autorizacija u mreži usluga se taj proces odvaja iz mikrousluga u posebni proces pod nazivom *proxy*. Svu unutarnju komunikaciju između mikrousluga pregledava taj proces koji se je autonoman i prijenosan jer se nalazi izvan mikrousluga. Osim autorizacije i validacije zahtjeva *proxy* se koristi za implementiranje šifriranje

zahtjeva od kraja do kraja (eng. *end-to-end*). Na slici 3.5 napravljenoj po uzoru na [12], prikazano je korištenje *proxya* unutar mreže usluga.



Sl. 3.5 *Prikaz korištenja proxyja unutar mreže usluga [12].*

Mreža usluga se smatra decentraliziranim uzorkom programiranja jer se *proxy* proces nalazi uz svaku mikrouslugu gdje kod *API Gatewaya* svi zahtjevi i rezultati prolaze kroz jedan centralni proces. *Proxy* i mikrousluga ne moraju biti na istom poslužitelju, ali se to preporuča jer će to smanjiti latenciju. Kako bi se olakšalo upravljanje postavkama *proxyja* uvodi se još jedan zasebna usluga koja se izravno spaja samo na *proxyje*. Njima zatim propagira prethodno definirane postavke. Slika 3.6 napravljenoj po uzoru na [12], prikazuje korištenje upravljačke usluge.

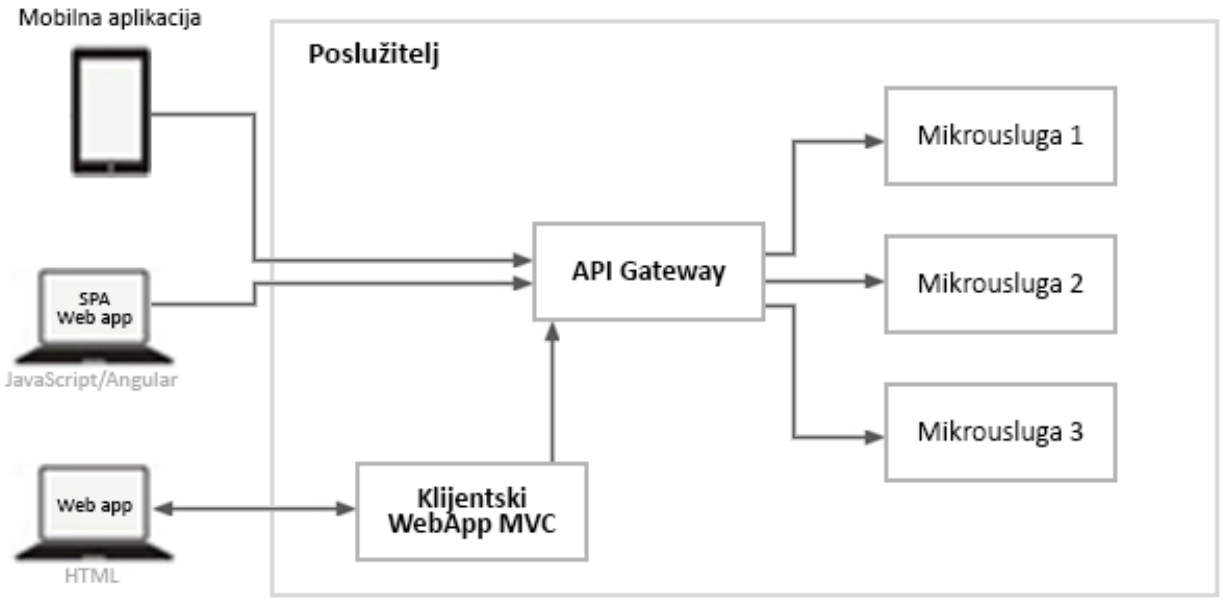


Sl. 3.6 *Prikaz korištenje upravljačke usluge za propagiranje postavki proxy procesima [12].*

Korištenje mreže usluga zbog proxy procesa zahtijeva prilagođavanje cjevovoda za integriranje programske podrške (*CI/CD*) jer se ispred svake mikrousluge mora postaviti proxy proces koji nadgleda komunikaciju od i prema mikrousluzi [12].

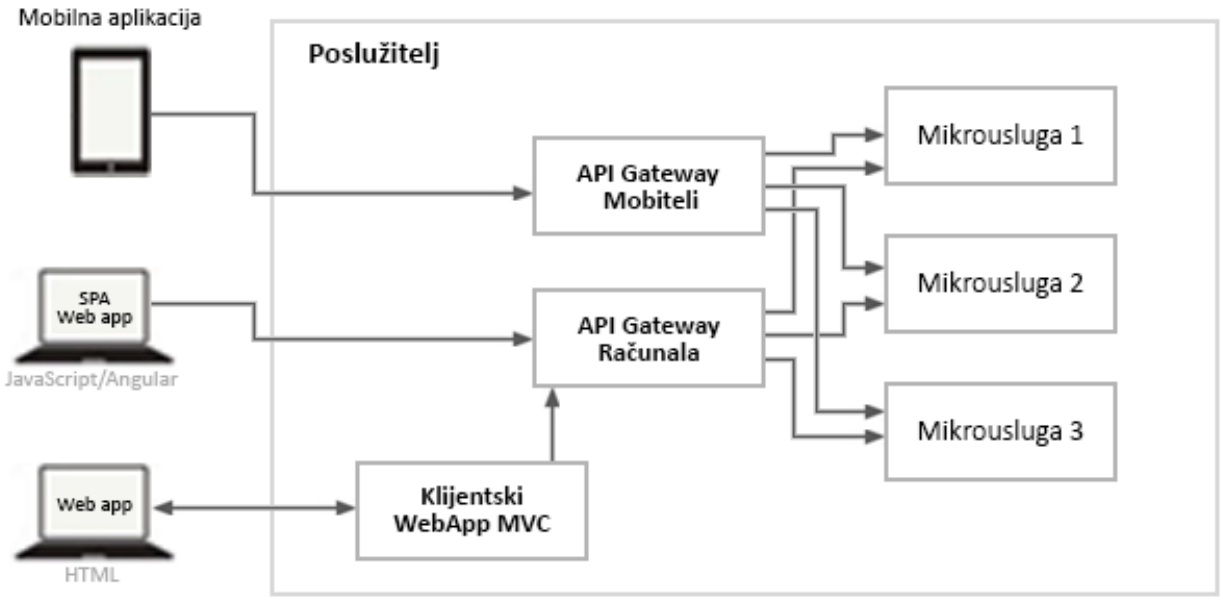
3.2.3. API Gateway

API Gateway je prema [9] programerski uzorak koji se koristi kada se dizajniraju i izrađuju velike i kompleksne aplikacije temeljene na arhitekturi mikrousluga. *API Gateway* uzorak se izgrađuje uzimajući u obzir potrebe korisničke strane, jer se *API Gateway* nalazi između korisničke i poslužiteljske programske podrške. Na slici 3.7 napravljenoj po uzoru na [9], nalazi se prikaz implementacije *API Gateway* uzorka unutar aplikacije bazirane na arhitekturi mikrousluga.



Sl. 3.7 *Prikaz implementacije API Gateway uzorka unutar aplikacije bazirane na arhitekturi mikrousluga [9].*

Korisničke aplikacije šalju zahtjeve na jednu krajnju točku na poslužitelju odnosno šalju zahtjeve na *API Gateway*. Zbog toga što više korisničkih aplikacija mogu komunicirati s *API Gatewayom* i to što jedan *API Gateway* mora udovoljavati velikom broju različitih korisničkih aplikacija može doći do toga da programski kod za *API Gateway* bude prevelik i jako kompleksan. Zato se preporuča da se tada *API Gateway* podijeli u više manjih mikrousluga ili više *API Gateway* usluga. *API Gateway* se treba podijeliti prema poslovnoj logici i jedan *API Gateway* ne smije objedinjavati sve mikrousluge u aplikaciji [9]. Na slici 3.8 napravljenoj po uzoru na [9], prikazana je implementacija više *API Gatewaya*.



Sl. 3.8 *Prikaz implementacije više API Gatewaya za posebne korisničke aplikacije [9].*

Prednosti korištenja *API Gateway* uzorka su:

- Izoliranje korisnika od načina podjele poslužiteljske aplikacije na mikrousluge
- Izoliranje korisnika od problema određivanja krajnjih točaka za pojedinu mikrouslugu
- Pruža optimalne *API* usluge za sve zahtjeve korisnika
- Smanjuje broj zahtjeva prema nizu mikrousluga jer *API Gateway* na jedan klijentski zahtjev može pozvati niz mikrousluga i predati jedan odgovor korisniku [9]
- Jednostavna implementacija autorizacije, potvrde autentičnosti, keširanja ili uravnoteživanje opterećenja jer je potrebno napisati jedan programski kod jer se *API Gateway* nalazi između korisničkih aplikacija i mikrousluga

Jedan od glavnih nedostataka *API Gateway* uzorka je povećanje kompleksnosti programske podrške što može dovesti do još jednog izvora pogrešaka i mogućih kvarova sustava. Još jedan nedostatak ovog uzorka programiranja je povećanje vremena odgovora korisniku na zahtjev jer je potrebno vrijeme da se prijeđe s *API Gatewaya* na odabranu mikrouslugu [9].

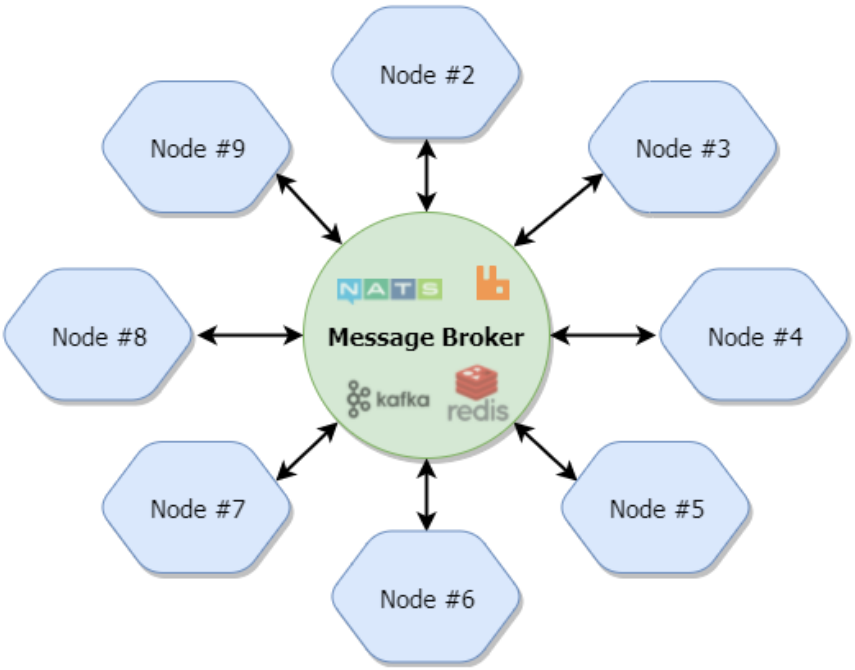
3.3. Mikrousluge u Node.js-u korištenjem Moleculer okvira za razvoj

Moleculer je programski okvir otvorenog koda za razvijanje programske podrške u arhitekturi mikrousluga u *Node.js-u* [13]. Pomoću njega se mogu izgraditi učinkovite, pouzdane i skalabilne programske podrške temeljene na mikrouslugama jer sadrži većinu od potrebnih značajki za razvoj mikrousluga. Neke od tih značajke su:

- Koncept komunikacije putem zahtjeva i odgovora

- Podržava arhitekturu programske podrške temeljene na događajima
- Podržava spremanje podataka u priručnu memoriju (*Memory* ili *Redis*)
- Ugrađeni načini prijenosa podataka pomoću *TCP*, *NATS*, *MQTT*, *Kafka*, itd.
- Ugrađeni programski kod za serijalizaciju podataka (*JSON*, *Avro*, *MsgPack*, itd.)
- Uravnoteživanje opterećenja
- Automatsko otkrivanje svih usluga i spremanje u registar usluga
- Ugrađeni sustavi za praćenje analitika korištenja usluga [13]

Za komunikaciju između usluga koje su na različitim čvorovima koristi se transporter. *Transporter* je komunikacijski modul unutar *Moleculera* koji se koristi kada se koriste usluge na više čvorova. Podaci poslani *Transporterom* se prvotno šalju u *Message Broker* mehanizam koji zatim prosljeđuje podatak ka krajnjem odredištu. Na slici 3.9 prikazano je spajanje čvorova s *Message Brokerom* preuzetoj iz [15].



Sl. 3.9 *Prikaz spajanje čvorova poslužitelja na Message Broker pomoću Transportera [15].*

Message Broker može prenositi drugim uslugama: događaje, zahtjeve, obrađivati odgovore na zahtjev i slično. Ako je pokrenuto više instanci usluga tada se zahtjevi i odgovori automatski uravnotežuju kako bi se opterećenje ravnomjerno rasporedilo po svim instancama usluge. Prilikom prijenosa podataka *Transporter* modul koristi jezike za prijenos podataka i prilikom njegovog korištenja mora se definirati način na koji se obrađuje jezik za prijenos podataka. Nekoliko je već preddefiniranih *Transportera*: *TCP*, *NATS*, *Redis*, *MQTT*, *AMQP*, *Kafka*, *NATS Streaming*. Mogu

se napisati i vlastiti moduli za komunikaciju između čvorova i usluga [14]. Detaljnije o pokretanju mikrousluga na više čvorova koristeći se *Dockerom* je opisano u potpoglavlju 6.3.2.

Usluge u *Moleculeru* mogu biti lokalne i udaljene. Lokalne usluge su dvije ili više usluga koje se nalaze na jednom čvoru. One dijele računalne resurse i koriste lokalnu sabirnicu koja omogućava brzu komunikaciju bez mrežne latencije (eng. *network latency*) jer se ne koristi *Transporter*. Udaljene usluge su usluge koje su smještene na različitim čvorovima i oni komuniciraju pomoću *Transportera*, ali komunikacija između njih nije brza, jer se zbog korištenja modula *Transporter* pojavljuje mrežno kašnjenje.

Moleculer koristi uzorak za izlaganje mikrousluga krajnjim korisnicima *API Gateway*. U praktičnom dijelu diplomskog rada poslužiteljska aplikacija koristi samo jednu *API Gateway* uslugu za komunikaciju između klijenata i ostatka mikrousluga. *API Gateway* usluga zadužena je za: autorizaciju, *CORS* zaglavlja, ograničivač brzine pristupa *API*-ju, posluživanje višestrukih ruta, posluživanje učitanih datoteka, parsiranje zahtjeva, itd. Za komunikaciju između usluga i klijentskih aplikacija, *API Gateway* usluga može koristiti: *HTTP* protoku, *WebSocketima*, itd. [15]. Zbog veličine poslužiteljske aplikacije i zbog toga što se ona neće koristiti u produkciji, koristi se jedan *API Gateway* usluga. Detaljnija analiza korištenja arhitekture zasnovane na mikrouslugama i korištenja *API Gateway* uzorka modeliranja mikrousluga će biti prikazana i objašnjena u potpoglavlju 7.3.2.

4. VIŠENITNA OBRADA PODATAKA

U ovome poglavlju opisana je upotreba višenitnosti prilikom paralelne obrade podataka unutar okruženja *Node.js* te razlika između obrade podataka koristeći se radnim nitima i asinkrone obrade podataka.

4.1. O višenitnosti i višenitnim računalnim programima

Višezadaćnost (eng. *multitasking*) kod računalnih operacijskih sustava je mogućnost pokretanja jednog ili više računalnih programa koji obavljaju svoju zadaću u isto vrijeme. Višenitnost je vezana uz višezadaćnost tako što omogućuje pokretanje dijelova istog programa u isto vrijeme. Višenitni program se dijeli na niti koje su zapravo potprocesi istog programa, a koje dijele zajedničke podatke u memoriji. Za razliku od procesa niti imaju manji utjecaj na performanse programa jer je *jeftinije* (u pogledu performansa) stvoriti i uništiti nit nego stvoriti novi proces. Niti mogu biti u četiri stanja: *nove, pokrenute, blokirane* i *mrtve*. Nit sa stanjem *nova* je nit koja je tek kreirana i nije još pokrenuta, odnosno programski kod unutar nje se nije još pokrenuo. Prije no što se nit pokrene operacijski sustav mora odraditi dodatne poslove zapisivanja informacija kako bi se ta nit kasnije mogla sinkronizirati i uništiti. Nit sa stanjem *pokrenuta* je nit kojoj je pozvana *start* metoda i trebala bi početi sa svojim izvođenjem u što kraćem vremenu. No to sve ovisi o operacijskom sustavu jer raspoređivač (eng. *scheduler*) dodjeljuje vrijeme izvođenja nitima. Postoje dva načina raspoređivanja izvođenja niti: prekidivi (eng. *preemptive*) i suradni (eng. *cooperative*). Odabir raspoređivanja izvođenja niti se razlikuje od uređaja do uređaja. Moderna računala koriste prekidni način raspoređivanja niti što znači da raspoređivač daje svakoj pokrenutoj niti dio vremena za izvođenje, ako je taj dio vremena prekoračen operacijski sustav pauzira trenutnu nit i daje drugoj niti dio vremena izvođenja. Time se sve niti izvode bez blokiranja. Prilikom odabira nove niti za izvođenje u obzir se uzimaju razina prioriteta koju nit ima. S druge strane manji uređaji (poput mobilnih uređaja) koriste suradno raspoređivanje niti gdje se nit izvodi do kraja sve dok se ne iskoriste *yield* ili *sleep* metode. Blokirana nit je nit koja je *pauzirana*: korištenjem *sleep* metode, korištenjem blokiranih IO poziva prema datotekama jer čeka izvršavanje nekog uvjeta ili jer želi pristupiti bravi (eng. *lock*) koju neka nit već koristi. Kada je nit blokirana druga nit dobiva pravo izvođenja, a nit koja je bila blokirana kada se odblokira mora ponovno čekati svoj red na izvođenje. Niti u stanju mrtva (eng. *dead thread*) je nit koja je završila svoje izvođenje na normalan način ili je prekinuta zbog neke iznimke (eng. *exception*).

Zbog toga što se jedan program istovremeno izvodi u više manjih dijelova, niti se moraju sinkronizirati kako bi mogle pristupati istim objektima i varijablama unutar programa [16].

Sinkronizacija niti se izvršava putem tri mehanizma koja su ugrađena u jezgri operacijskog sustava: *brave*, *semafori* i *barijere*.

Brave su mehanizam unutar operacijskog sustava koji omogućava da se nit (koja ima pristup bravi) izvodi bez ikakvog daljnjeg čekanja. Sve ostale niti koje se žele izvoditi moraju čekati svoj red i konstantno provjeravati jeli brava slobodna za pristup. Brava se oslobađa u dva slučaja: ako je nit završila svoje izvođenje ili automatski ako sustav primijeti da nit predugo čeka za svoje izvođenje. Ovaj mehanizam je pogodan za korištenje kada se očekuje da su niti blokirane na kraći rok zbog čega se ovaj mehanizam koristi u jezgrama operacijskih sustava. Inače je on težak za implementaciju jer se moraju koristiti posebne asemblerske naredbe kako bi se omogućio svim nitima fluidan pristup bravi.

Semafori su mehanizam koji je sličan bravama, ali su mnogo robusniji. Semafor je varijabla koja se koristi za kontrolu pristupa resursima procesora i kako bi se izbjegle situacije istovremenog pristupa više niti kritičnom osječku. Najjednostavnija implementacija ovog mehanizma je implementacija cjelobrojne varijable koja se povećava kada se neka nit već izvodi ili smanjuje kada nit završi svoje izvođenje. Drugi način implementacije semafora je cjelobrojna varijabla koja može biti: jedan (kada se već neka nit izvodi) ili nula (kada nit završi svoje izvođenje). Taj način implementacije semafora se koristi kako bi se implementirao mehanizam brava [17].

Barijere su mehanizam sinkronizacije niti koja omogućuje sinkronizaciju jedne ili grupe niti. Barijere imaju fiksni kapacitet (n) i kada nit dosegne *barijeru* ona tada pozove metodu za čekanje kako bi barijera znala da nit čeka izvođenje. Ako je broj niti koje čekaju *barijeru* manji od *n-1* tada se nit koja je pozvala metodu stavljena u red čekanja. U protivnome se sve niti koje čekaju i nit koja je pozvala metodu uništavaju [34]. Kada se barijera u potpunosti oslobodi druga grupa niti može pristupiti barijeri i započeti svoje izvođenje [17].

4.1.1. Primjer korištenja višonitnosti u praksi – poslužitelj Apache Tomcat od strane Netflixa

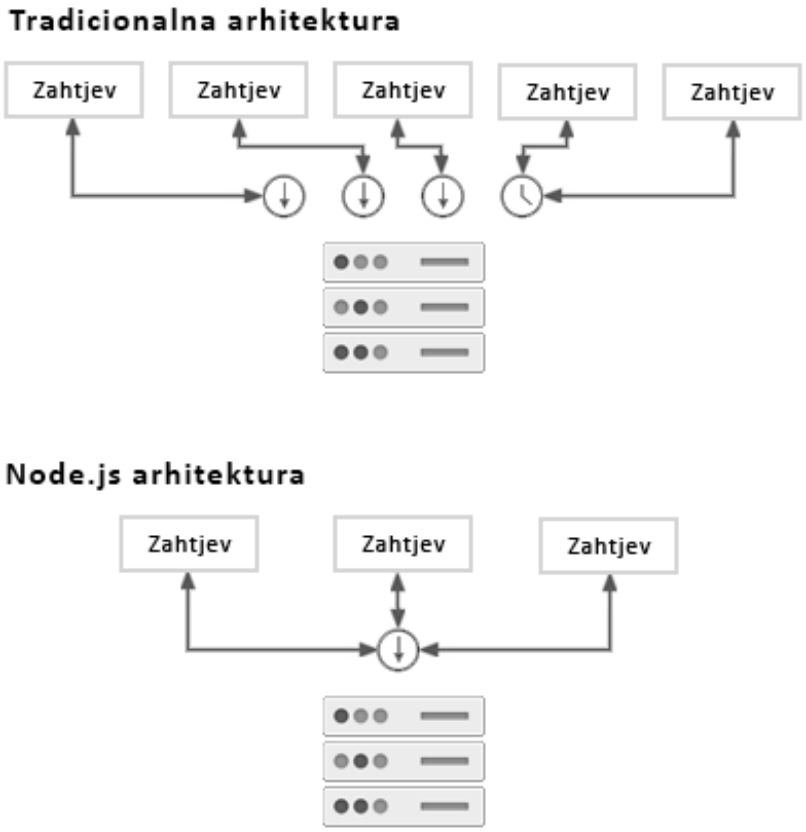
Ovdje je prikazan primjer korištenja višenitnosti iz prakse gdje *Netflix* koristi višenitnost za posluživanje svojih mikrousluga putem *Apache Tomcat* poslužitelja. *Apache Tomcat* je poslužitelj koji se koristi za posluživanje *Java* web aplikacija. Stoga se *Apache Tomcat* koristi za posluživanje dijelova *Netflixove* aplikacije. *Apache Tomcat* je višenitna aplikacija koja za svakog klijenta pokreće posebnu radnu nit. Kada se klijent spoji pokreće se *TCP* rukovanje između operacijskog sustava i njega kako bi se uspostavila veza. Ovisno o implementaciji *Tomcata*, veze se mogu spremati u jednom redu gdje se spremaju sve veze ili u dva reda gdje se u jedan spremaju

uspostavljene veze, a u drugi ne uspostavljene veze. Kada se uspostavi veza ona se iz reda veza za čekanje prebacuje u red s uspostavljenim vezama kako bi mogli klijent dalje mogao koristiti aplikaciju. *Tomcat* nit za prihvaćanje veza tada preuzima uspostavljenu vezu iz reda i provjerava postoje li radne niti unutar grupe niti, ako ona ne postoji *Tomcat* kreira novu (ako ima prostora u grupi) ili čeka da se oslobodi neka radna nit. Kada se pronađe radna nit tada nit za uspostavljanje veza predaje klijenta radnoj niti i vraća se osluškivanju krajnje točke za novim konekcijama. Radna nit tada čita i obrađuje zahtjev, te naposlijetku šalje odgovor klijentu. Ako veza više nije dostupna tada se radna nit oslobađa i stavlja na raspolaganju drugim klijentima u grupi radnih niti. Ukoliko je klijent i dalje aktivan radna nit čeka nove zahtjeve od klijenta određeno vrijeme. Ako to vrijeme istekne i radna nit nije u međuvremenu dobila daljnje zahtjeve od klijenta, tada se veza prekida i radna nit se stavlja na raspolaganje drugim klijentima u grupi radnih niti. Postavljanjem velikog broja veza koje se mogu prihvatiti od strane *Tomcata*, dovodi do uzrokovanja preopterećenosti poslužitelja i naposlijetku zagušivanja njegovog procesora s radnim nitima [18].

Ovaj primjer je također iskorišten kao misao vodilja prilikom implementacije višenitnosti unutar obrade slikovnih datoteka, a koji je objašnjen u potpoglavlju 6.1.

4.2. Višenitnost unutar Node.js-a

Node.js je *runtime* okruženje otvorenog koda koje je bazirano na programiranju pomoću programskog jezika *JavaScript* i *Googleovom V8 pokretaču* kako bi se stvorile brze i skalabilne web aplikacije. Arhitektura *Node.js-a* se bazira na jednonitnim događajima i na asinkronom ne blokirajućem unosu i ispisu podataka. Na slici 4.1 prikazana je razlika između tradicionalne i *Node.js* klijent-poslužitelj arhitekture [19]. Kod tradicionalne klijent-poslužitelj arhitekture za svakog klijenta se kreira nova nit ili čeka postojeću nit na poslužitelju koja zatim obrađuje njegove zahtjeve ka poslužitelju. Za rukovanje asinkronim događajima *Node.js* koristi *libuv* biblioteku kao apstrakcijski sloj kako bi se *Node.js* programi mogli izvoditi na svim operacijskim sustavima.



Slika 4.1 *Prikaz tradicionalne i Node.js klijent-poslužitelj arhitekture [19].*

Sve *Node.js* aplikacije se pokreću pomoću *V8 pokretača* koji tijekom izvođenja pretvara *JavaScript* kod u strojni kod, koji se zatim izvodi na poslužitelju. Kako je *JavaScript* programski jezik koji ne podržava višenitnost i *Node.js* se bazira na jednonitnoj obradi zahtjeva, dugo vremena je *Node.js* bio razvojni okvir za jednonitne aplikacije. *Libuv* osigurava korisniku jednonitnu petlju događaja, ali za sve ostale stvari koje zahtijevaju više vremena za obradu podataka (kao na primjer čitanje ili pisanje datoteka) *libuv* stvara nove niti iz grupe radnih niti [19]. Osim toga, neke nativne metode poput onih unutar *crypto* modula su također višenitne jer se nativni moduli pišu u C-u koji se preko *V8 pokretača* prevode u strojni jezik s napisanim *JavaScript* kodom. No, i dalje korisnik nije mogao napisati višenitnu web aplikaciju.

U šesnaestoj inačici, *Node.js* je dobio radne niti (eng. *worker threads*). *Worker_threads* modul omogućava pisanje višenitnog programskog koda koji se izvodi paralelno. Radne niti se koriste za izvođenje zadataka koji su intenzivni za procesor računala na kojem se izvodi *Node.js* aplikacija. Radne niti dijele memoriju tako što prebacuju *ArrayBuffer* instance između sebe ili dijele jednu *SharedArrayBuffer* instancu. Osim kreiranja jedne radne niti mogu se kreirati i bazen radnih niti (eng. *worker thread pool*) iz kojeg se tada uzimaju slobodne radne niti, a one koje završe svoje izvođenje se oslobađaju i stavljaju na raspolaganje za korištenje unutar istog bazena radnih niti.

Korištenje bazena radnih niti je pogodno kod obrade datoteka jer stvaranje i uništavanje radnih niti može imati negativan učinak na performanse obrade nego korištenje bazena radnih niti. Radne niti imaju dvosmjernu unutarnju komunikaciju porukama. Što znači da glavna nit može poslati i primiti poruke od drugih radnih niti. Prilikom stvaranja nove radne niti se stvara par portova zaduženih za razmjenu poruka između niti. Obično se radne niti deklariraju i definiraju unutar posebne datoteke, a onda se pozivaju u drugoj datoteci u programskom kodu gdje se trebaju koristiti radne niti. Pošto se sve definira unutar jedne datoteke glavna nit se identificira s metodom *isMainThread*. Unutar glavne niti se deklarira radna nit i podaci koji će se podijeliti s drugim radnim nitima. Također se definiraju slušatelji poruka (eng. *message listeners*) kako bi glavna nit mogla znati kada je neka radna nit normalno završila izvođenje ili je izašla s pogreškom ili nekom drugom iznimkom. Druge radne niti se obično definiraju unutar *else* bloka naredbi koji označava da ta radna nit nije glavna nit. Kada se završi obrada podataka tada se obrađeni podaci pomoću *postMessage* metode šalju na glavnu nit. Nakon završetka obrade radna nit se uništava ili ako je dio bazena radnih niti onda oslobađa svoju memoriju i vraća se u bazen slobodnih radnih niti [20].

Praktični dio rada ima opciju učitavanja datoteka na poslužitelj. Jedan od zahtjeva na poslužiteljsku aplikaciju je optimalno korištenje prostora na poslužitelju. Stoga se učitane slikovne datoteke sažimaju i smanjuju. Radne niti se koriste prilikom sažimanja i smanjivanja slikovnih datoteka, jer taj zadatak zahtjeva korištenje procesora, a raspoređivanjem datoteke na radne niti sprječavaju mogućnost blokiranja poslužitelja prilikom obrade velikih slikovnih datoteka. Cijeli programski kod i način korištenja radnih niti u praktičnom dijelu rada nalazi se u potpoglavlju 6.1.

5. MODELIRANJE PROGRAMSKOG RJEŠENJA SUSTAVA

U ovom poglavlju su opisani postavljeni zahtjevi na poslužiteljski dio sustava kao i alternative koje su moguće za odabrano rješenje postavljenog zahtjeva.

5.1. Zahtjevi na poslužiteljski dio sustava

Poslužiteljski dio sustava prije svega mora pružati podatke klijentskoj strani aplikacije za potporu pri upravljanju projektima agilnog razvoja. Poslužiteljska aplikacija treba moći u što kraćem vremenskom intervalu: primiti klijentske zahtjeve, obraditi ih i vratiti rezultat. Glavni zahtjev za funkcionalnost aplikacije je korištenje *Node.js* razvojnog okvira, ali aplikacija mora biti dizajnirana s arhitekturom zasnovanom na mikrouslugama. Između usluga na poslužitelju i klijentske aplikacije se treba nalaziti *API Gateway* usluga koja će obrađivati i preusmjeravati klijentske zahtjeve ka traženoj usluzi. Zbog male veličine aplikacije i zbog toga što se ona neće upotrebljavati u produkciji nema potrebe koristiti više *API Gateway* usluga kao što se koristi u stvarnim slučajevima. Svaka mikrousluga treba imati svoju instancu *MongoDB* baze podatka koja je nerelacijska baza podataka i unutar koje se moraju spremiti svi podaci vezani uz domenu rada usluge koja ju koristi. Usluge moraju imati jedinstvenu domenu rada, na primjer usluga *user* ne smije posjedovati programski kod koji nema veze s korisnicima. Kako bi se osigurala sigurnost korisničkih računa prilikom registracije računa generira se jedinstvena zaporka koja se šalje registriranom korisniku na njegovu e-poštu, s uputama da se ista mora odmah promijeniti. Zaporke u bazu podataka se spremaju kao kriptirani podatci s automatski generiranim *hashem* i *saltom* što omogućava da korisničke zaporke budu sigurne čak i ako dođe do proboja u bazu podataka. Kako bi se osigurala brzina obrade i prikaza podataka koriste se priručne memorije (eng. *cache*) kako bi se spremili podatci koji se rijetko mijenjaju. To mogu biti postavke korisničkog sučelja za svakog korisnika, projekata koji nemaju nekih promjena ili *MongoDB* dokumenti koji se učestalo koriste. Sljedeći zahtjev za funkcionalnost poslužiteljskog dijela sustava je mogućnost učitavanja datoteka koje će biti prikvačene uz pojedini zadatak. Da se brzo ne potroši prostor na poslužitelju i tako osiguraju dodatni ekonomski troškovi, potrebno je osigurati sažimanje (eng. *compression*) i smanjivanje (eng. *resizing*) slikovnih datoteka. Za brže sažimanje i prevenciju blokiranja cijelog poslužitelja tokom sažimanja, potrebno je koristiti radne niti koje će paralelno sažeti i smanjiti slike. Aplikacija nema ograničenja tipa i veličine slikovnih datoteka prilikom njihovog učitavanja.

5.2. Opis odabrane arhitekture mikrousluga

Kao što je objašnjeno u potpoglavlju 3.2, postoji više načina modeliranja usluga: izravno spajanje klijenta i mikrousluga, mreža usluga i *API Gateway*. U programskom rješenju sustava korišten je *API Gateway*. Klijenti se prvo spajaju na *API Gateway* uslugu koji se u arhitekturi poslužitelja nalazi između klijenata i ostalih mikrousluga. Glavna zadaća *API Gatewaya* je prosljeđivanje klijentskih zahtjeva prema pravoj mikrousluzi i vraćanje odgovora od mikrousluge ka pravom korisniku. Kako jedna usluga može imati više instanci, na *API Gatewayu* je da odabere instancu koja je slobodna i koja može brzo obraditi zahtjev. *Moleculer* razvojni okvir dolazi s *API Gatewayom* kao odabranim načinom modeliranja usluga. Unutar te usluge mogu se konfigurirati dodatne rute i dodatno definirati posebne postavke za te rute poput: autorizacije i provjere autentičnosti zahtjeva, *CORS postavki*, *parsera* za obradu tijela zahtjeva i slično. Što je korisno, ako se želi omogućiti paralelno korištenje *REST API*-ja i učitavanja datoteka na poslužitelj. *API Gateway* treba konfigurirati raščlambu klijentskih zahtjeva kako bi poslužitelj lakše obradio zahtjeve, ali raščlamba klijentskih zahtjeva nije potrebna prilikom učitavanja datoteka na poslužitelj. Prilikom svakog zahtjeva na *API Gateway* pokreće se posrednički program koji provjerava jeli korisnik autoriziran i ima li pravo pristupiti odabranoj ruti. Također se posrednički program koristi i kod provjere međusobnog dijeljenja resursa (eng. *Cross-Origin Resource Sharing*). Također se može postaviti limit slanja zahtjeva na poslužitelj od strane jednog klijenta. Obično se zahtjev poslan od klijenta sprema na deset sekundi i ako se dosegne broj od deset zahtjeva unutar deset sekundi tada se klijentu svi danji zahtjevi blokiraju. Korištenjem mehanizma limitiranja slanja zahtjeva poboljšava se dostupnost *API*-ja svim korisnicima bez da se izgladne resursi poslužitelja. U programskom rješenju sustava se koristio jedan *API Gateway* za sve vrste klijenata, ali se u stvarnoj primjeni koristi više *API Gatewaya* za različite tipove klijenata (jedan za klijente na računalima, a drugi za klijente na mobilnim uređajima).

5.3. Radne niti

Kao što je objašnjeno u poglavlju 4, radne niti koriste se kako bi se omogućila paralelna obrada podataka i stvaranje višenitnog programskog koda na poslužitelju. Preporučuje se korištenje radnih niti prilikom korištenja kompleksnih matematičkih operacija, ali ne i kod učitavanja datoteke na poslužitelj jer stvaranje i uništavanje niti može utjecati na ukupne performanse učitavanja nego da se ne koriste niti. U praktičnom dijelu diplomskog rada će se radne niti koristiti za smanjivanje veličine i sažimanje slika nakon što se one učitaju na poslužitelj. Učitana slika se unutar programskog koda prikazuje kao binarni kod i tada se ona smanjuje i sažima kako bi se spriječilo

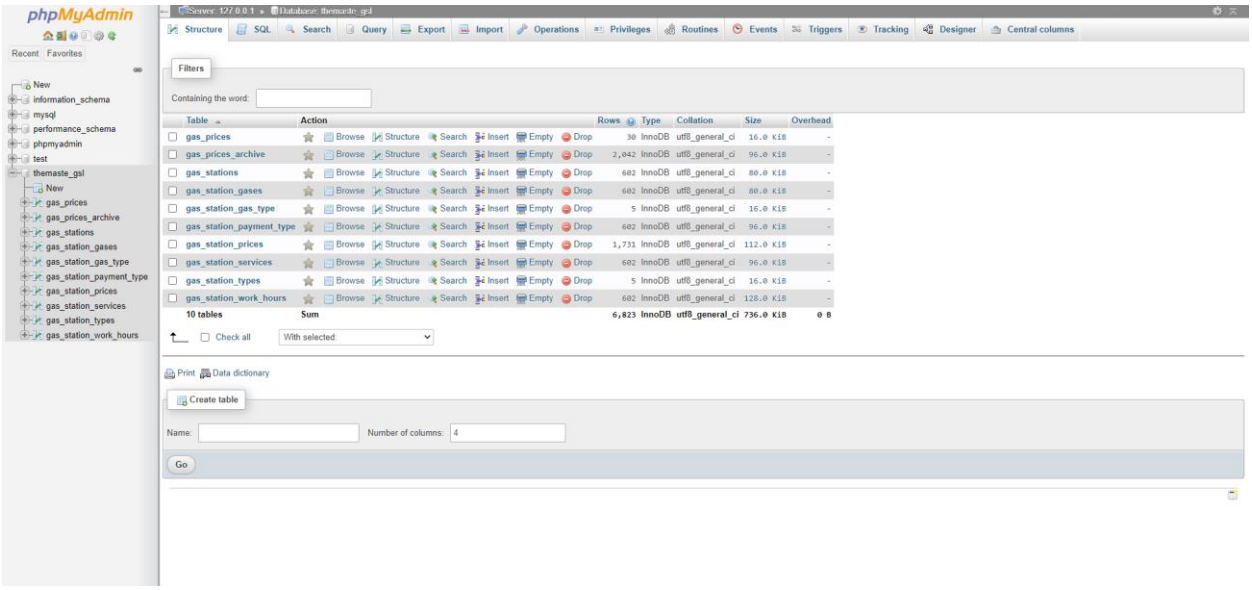
veliko zauzeće memorije na poslužitelju. Kod manjeg broja korisnika i učitavanja datoteka korištenje radnih niti ne dolazi toliko do izražaja, ali kod velikog broja onda se broj obrađenih zahtjeva povećava a vrijeme obrade smanjuje. Detaljnija analiza utjecaja višenitnosti na učinkovitost same poslužiteljske aplikacije je obavljena u potpoglavlju 7.3.1.

5.4. Usporedba relacijskih i nerelacijskih baza podataka

U praktičnom dijelu diplomskog rada koristi se nerelacijska ili *noSQL* baza podataka. Ovdje će se objasniti razlika između relacijskih i nerelacijskih baza podataka konkretno *MySQL* i *MongoDB*.

5.4.1. Relacijske baze podataka (MySQL)

MySQL je relacijska baza podataka bazirana na strukturnom jeziku upita (eng. *structured query language*). Baze podataka su strukturirana kolekcija podataka. Relacijske baze podataka organiziraju podatke unutar više tablica u kojima podaci mogu imati relacije između sebe. Umjesto da se podaci spremaju u jednu veliku datoteku. Tablice se sastoje od stupaca u kojima se definiraju podaci koji se mogu spremiti i redaka koji definiraju spremljene podatke. Korisnik može postaviti pravila koja uređuju odnose između različitih polja podataka: jedan-na-jedan, jedan-prema-više, jedinstveni podatak, obavezni podatak, izborni podatak i *pokazivači* između različitih tablica. Dobro definirana pravila osiguravaju da unutar baze podataka ne bude nedosljednih, dupliciranih, zastarjelih ili nepostojećih podataka i da baza podataka radi brzo. Strukturirani jezik upita (SQL) se koristi kako bi se pristupilo podacima unutar baze podataka, a korisnik može upisati *SQL* kod direktno u *MySQL* aplikaciju ili ugraditi *SQL* izraze u programski kod napisan u drugom programskom jeziku [21]. Na slici 5.1 prikazana je *MySQL* baza podataka s definiranim tablicama: *gas_prices, gas_prices_archive, gas_stations, gas_station_gases, gas_station_gas_type, gas_station_payment_type, gas_station_prices, gas_station_services, gas_station_types, gas_station_work_hours*.



Sl. 5.1 *Prikaz MySQL baze podataka s definiranim tablicama (napisati koje tablice).*

MySQL baze podataka su optimirane za visoke performanse prilikom velikog broja podataka unutar baze podataka. Koristi višenitni programski kod za brzo spajanje velikog broja podataka. Osim toga relacijske baze podataka omogućuju kreiranje procedura i funkcija za laku obradu podataka gdje se pokreće prethodno generiran *SQL* kod.

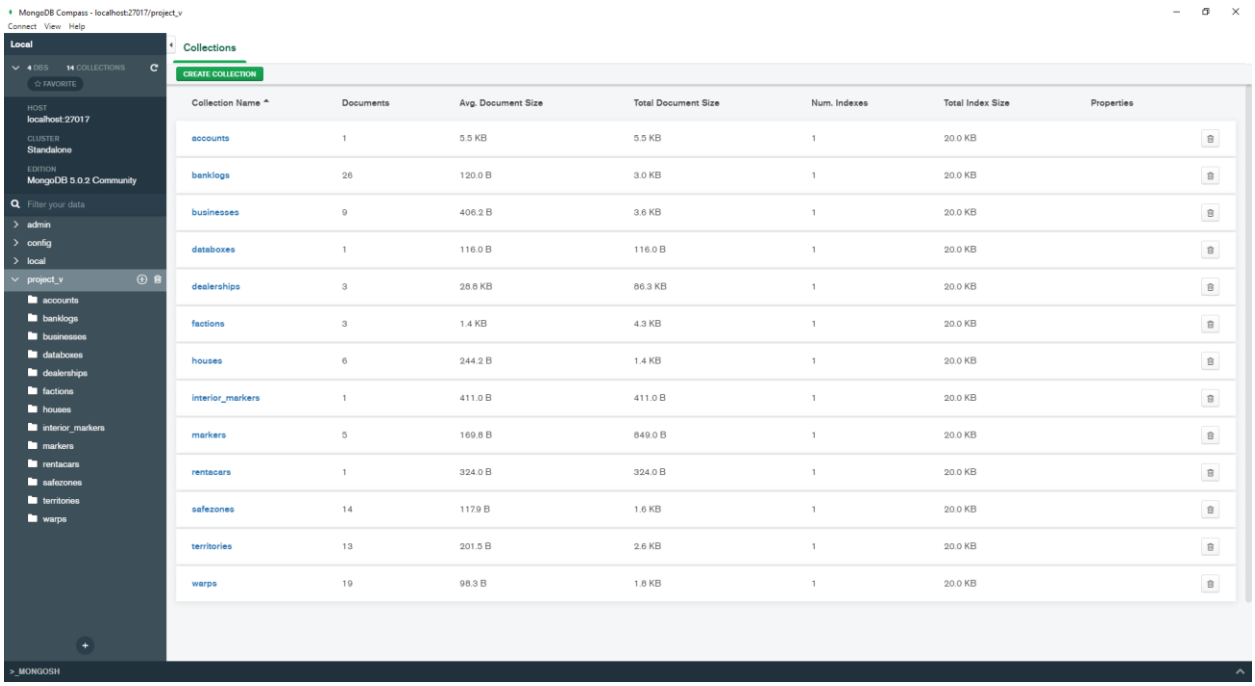
5.4.2. Nerelacijske baze podataka (MongoDB)

Nerelacijske odnosno *noSQL* baze podataka su baze podatka koje nemaju tablice i podatke spremaju drukčije od relacijskih. Nerelacijske baze podataka se razlikuju prema tipu podataka koji se žele spremiti unutar njih. Nerelacijske baze podataka koriste tipove podataka koje su prilagođenije specifičnim slučajevima upotrebe. Ključ/vrijednost nerelacijske baze podataka su pogodne za jednostavne upite dok su baze podataka u kojima su podaci spremljeni poput grafa su pogodne pri identificiranju složenih odnosa između zasebnih dijelova podataka. Nerelacijske baze podataka dolaze do izražaja kada korisnik želi spremiti više podataka u jedan dokument dok relacijske baze podataka će spremiti *pokazivač* na podatak u drugim tablicama i izvesti nekoliko upita kako bi došla do istog podataka kao i nerelacijske baze. Osim toga nerelacijske baze podataka se mogu skalirati na više različitih poslužitelja dok relacijske baze zahtijevaju korištenje jednog poslužitelja [22].

MongoDB je nerelacijska baza podataka koja podatke sprema u niz *JSON* dokumenata koje se nazivaju kolekcije. *MongoDB* dokument se sastoji od niza parova ključ/vrijednost koji mogu biti različitih tipova uključujući polja i ugniježdene dokumente. Važno je napomenuti da se parovi ključ/vrijednost mogu razlikovati od dokumenta do dokumenta unutar pojedine kolekcije jer se

dokumenti sami sebe opisuju i ne postoji shema po kojoj se definiraju podaci koji se spremaju unutar kolekcija. Zbog toga je *MongoDB* fleksibilnija baza podataka od *MySQL*-a jer nije potrebno poznavati principe normalizacije podataka i relacijskog dizajna baze podataka. Za dohvate podataka unutar *MongoDB* baze podataka se koristi *JavaScript* za razliku od *MySQL* koji koristi *SQL* jezik. *MongoDB* je optimiran za operacije masovnog unosa i ažuriranja podataka unutar baze podataka [23]. Primjer optimiranosti je: za učitavanje trideset tisuća podataka u *MySQL* bazu podataka potrebno je oko dvije tisuće četiristo sekundi, dok *MongoDB*-u treba dvije tisuće dvjesto sekundi. Drugi primjer je brisanje trideset tisuća podataka gdje *MySQL* bazi podataka treba oko tisuću dvjesto sekundi, a *MongoDB* bazi podataka je potrebno devetsto sekundi [32]. Na slici 5.2 prikazan je primjer *MongoDB* baze podataka s kolekcijama podataka.

MongoDB omogućuje kreiranje grodzova, gdje se više *MongoDB* baza podataka koristi na različitim poslužiteljima kao dijelovi baza podataka. Korisnik šalje upit nakupini koji ga usmjerava ka pravoj ljusci. *MongoDB* koristi *WiredTiger* skladišni motor za nerelacijsko spremanje i dohvaćanje podataka. Važno je napomenuti da *WiredTiger* koristi radnu memoriju računala (*RAM*) kako bi dijelove indeksa i dokumenata spremio u priručnu memoriju za brže obavljanje upita. *MongoDB* se najčešće koristi unutar *Node.js* okruženja zbog lakoće spajanja *MongoDB* baze podataka s aplikacijom putem paketa *mongoose*.



Sl. 5.2 *Prikaz MongoDB baze podataka s kolekcijama podataka*

6. PROGRAMSKO RJEŠENJE POSLUŽITELJSKOG DIJELA SUSTAVA ZA POTPORU UPRAVLJANJA AGILNOG RAZVOJA

U ovom poglavlju su opisana neka od programskih rješenja za postavljene zahtjeve na poslužiteljskom dijela sustava za potporu upravljanja agilnog razvoja.

6.1. Prikaz korištenja radnih niti prilikom smanjivanja i sažimanja slika

Vanjske datoteke se koriste kako bi pobliže objasnile ideju i viziju projektnog menadžera ili konkretno klijenta za kojeg se radi projekt u agilnom načinu. Analizirani programski sustavi objašnjeni u potpoglavlju 2.2, imaju veliki broj korisnika koji učitavaju veliku količinu datoteka na poslužitelj stoga je potrebno ekonomično iskoristiti memoriju na poslužitelju. Kako bi se ekonomično iskoristila memorija na poslužiteljskoj strani i kako bi učitane slike bile dostupne na svim uređajima odlučeno je da će se nakon učitavanja novih slikovnih datoteka vršiti smanjivanje, sažimanje i transformiranje datoteke u format *JPEG*. Format slika *JPEG* je format koji sadrži kompresiju slike koja prikazuje deset piksela kao jedan bez gubljenja kvalitete slike, osim toga je dostupan na svim uređajima i operacijskim sustavima. Na slici 6.1 prikazan je programski kod koji služi za definiranje krajnjih točaka na *API Gatewayu*.

Učitavanje datoteka se odvija slanjem toka podataka putem *POST* ili *PUT* zahtjeva na poslužitelj. Na krajnju točku za učitavanje podataka na poslužitelj se osim toka podataka (koji se koriste prilikom prijenosa podataka preko *HTTP* klijenta poput *AJAXa* i *cURLa*) može mogu učitati podaci i iz formi koji su označeni kao *HTML multipart* podatak. Za obradu (parsiranje) se koristi *busboy* modul dok se ugrađeni parseri zahtjeva unutar *Moleculera* isključuju. Modul *Busboy* koristi se jer na lak način omogućava konfiguriranje obrade svih zahtjeva od klijenta, a pogotovo zahtjeva za učitavanje podataka na poslužitelj. Nakon što se obrada zahtjeva za učitavanje datoteke završi tada se poziva *task.attachments* usluga koja je zadužena za spremanje datoteke na poslužitelj i sažimanje datoteke ako se radi o slici.

```
path: "/upload",

aliases: {
    "POST /": "multipart:task.attachments.save",
    "PUT /": "stream:task.attachments.save",
    "PUT /:id": "stream:task.attachments.save",
    "POST /file/:id": {
        type: "multipart",
        busboyConfig: {
            limits: {
                files: 1
            },
            onPartsLimit(busboy, alias, svc) {…
            },
            onFilesLimit(busboy, alias, svc) {…
            },
            onFieldsLimit(busboy, alias, svc) {…
            }
        },
        action: "task.attachments.save"
    },
    "POST /files": {
        type: "multipart",
        busboyConfig: {
            limits: {
                files: 3,
                fileSize: 1 * 1024 * 1024
            },
            onPartsLimit(busboy, alias, svc) {…
            },
            onFilesLimit(busboy, alias, svc) {…
            },
            onFieldsLimit(busboy, alias, svc) {…
            }
        },
        action: "task.attachments.save"
    },
    "GET /:task/:file": "task.attachments.get",
    "DELETE /:task/:file": "task.attachments.remove"
},

bodyParsers: {
    json: false,
    urlencoded: false
},
```

Sl. 6.1 *Prikaz programskog koda za definiranje krajnjih točaka za učitavanje podataka.*

Na slici 6.2 prikazan je isječak programskog koda koji sprema učitanu datoteku na poslužitelj. Datoteke se spremaju na poslužitelj kako bi mogle biti dostupne bez ikakvog kašnjenja i tako ispunile nefunkcionalni zahtjev postavljen u potpoglavlju 5.1. Nakon što se pozove usluga *task.attachments*, provjerava se postoji li zadatak za koji se učitava datoteka. Ako ne postoji, onda se klijentu vraća iznimka (eng. *exception*). Tok podataka dobiven od klijenta se prosljeđuje u metodu *saveFileToHost* koja vraća *Promise* što znači da je ta metoda asinkrona, odnosno njeno izvođenje ne blokira rad poslužitelja.

```
/**
 * Saves uploaded file to the disk.
 * @param {*} ctx
 * @returns
 */
saveFileToHost(ctx) {
    return new Promise((resolve, reject) => {
        const taskDir = path.join(UPLOAD_DIR, ctx.meta.fieldname);
        const fileName = this.randomName(ctx.meta.filename);
        const filePath = path.join(taskDir, fileName);
        const file = fs.createWriteStream(filePath);

        if(!fs.existsSync(taskDir)) {
            mkdir(taskDir);
        }

        file.on("close", async () => {
            resolve({ filePath, file, meta: ctx.meta });
        });

        ctx.params.on("error", err => {
            reject(err);
            file.destroy(err);
        });

        file.on("error", (err) => {
            reject(err);
            fs.unlinkSync(filePath);
        });

        ctx.params.pipe(file);
        this.broker.emit('task.attachment.uploaded', { task: ctx.meta.fieldname, file: fileName });
    });
}
```

Sl. 6.2 *Prikaz isječka programskog koda za spremanje učitane datoteke na poslužitelj.*

Kako bi se spremila datoteka na poslužitelj, potrebno je definirati direktorij u koji se želi spremiti i njezin naziv. Datoteke vezane uz pojedini zadatak se spremaju unutar direktorija čiji je naziv upravo naziv tog zadatka. Kada se obriše zadatak onda se briše i taj direktorij i sve datoteke unutar istog. Ako ne postoji direktorij što znači da je učitana datoteka prva datoteka učitana za odabrani zadatak, onda se direktorij kreira. Korištenjem metode *fs.createWriteStream* se definira tok podataka u koji se može zapisivati podaci. Svi tokovi podataka u *Node.js* koriste unutarnji međuspremnik (eng. *buffer*) u koji se spremaju podaci koji se trebaju iskoristiti (pročitati ili zapisati). Veličina međuspremnika se definira *highWaterMark* varijablom koja je obično 16 kilobajta. Kada se taj međuspremnik napuni, tok podataka se prekida sve dok se svi podaci iz tog međuspremnika ne iskoriste. Trenutni status toka podataka se saznaje iz događaja, ako dođe do pogreške prilikom čitanja datoteke tok podataka se briše, metoda *saveFileToHost* vraća pogrešku [24]. Metodom *ctx.params.pipe(file)* se podaci iz toka za čitanje zapisuju u tok za pisanje čime automatski započinje spremanje datoteke u memoriju poslužitelja. Kada se završi spremanje datoteke u memoriju okida se događaj *task.attachment.uploaded* koji drugim uslugama signalizira da je nova datoteka učitana u određeni zadatak. Nakon završetka spremanja datoteke u memoriju,

vrši se provjera *MIME* (eng. *Multipurpose Internet Mail Extensions*) informacija kako bi se saznalo o kojem tipu datoteke se radi. Ako se radi o slici tada se pokreće sažimanje i smanjivanje slike kako bi se ispunio zahtjev postavljen u potpoglavlju 5.1. Pozivom metode *imageCompress* se pokreću radne niti koje obrađuju učitanu sliku. Na slici 6.3 prikazan je modul za smanjivanje i sažimanje slike.

Kao što je spomenuto u potpoglavlju 4.1, modul u kojemu se nalazi programski kod za radne niti se poziva i za glavnu nit i za radne niti. Korištenjem varijable *isMainThread* programski kod radi razliku između glavne i radne niti. Ako se radi o glavnoj niti odnosno prvom pozivu *image-compress* modula onda se vrši deklaracija radnih niti. Deklaracija se sastoji od: definiranja naziva radnih niti, definiranja podataka koji će biti dostupni radnim nitima i osluškivanju događaja koje šalju radne niti. Radne niti će koristeći se *JIMP* modulom proslijeđeni podatak smanjiti na 75% od trenutne veličine slike i sažeti kvalitete slike na 60%. Nakon toga će se zapisati slika u *JPG* format i okinuti događaj glavnoj niti koji označuje kraj obrade slike. Završetkom obrade slike se vraća metoda *Promise.resolve()* koja označava da je definirani *Promise* uspješno izvršen, a ako se vrati *Promise.reject()* označava da se dogodila pogreška prilikom obrade unutar radnih niti. Detaljnija analiza korištenja višenitnosti u prikazanom primjeru je izvršena u potpoglavlju 7.3.1.

```
const { Worker, isMainThread, parentPort, workerData, threadId } = require("worker_threads");
const jimp = require("jimp");

if(isMainThread) {
    module.exports = (img) => new Promise(async (resolve, reject) => {
        const worker = new Worker(__filename, { workerData: img });
        worker.on("message", resolve);
        worker.on("error", reject);
        worker.on("exit", (code) => {
            if(code != 0) {
                reject(new Error(`Worker stopped with code ${code}`));
            }
        })
    });
}
else {
    (async () => {
        const imgObj = await jimp.read(workerData);
        imgObj.scale(0.75)
            .quality(60)
            .write(workerData);
        parentPort.postMessage(workerData);
    })();
}
```

Sl. 6.3 *Prikaz modula za smanjivanje i sažimanje slike.*

6.2. Ovjera autentičnosti uz pomoć JSON web tokena

Ovjera autentičnosti (eng. *authentication*) ili autorizacija (eng. *authorization*) je postupak ovjeravanja korisničkog identiteta unutar računalnog sustava [25].

6.2.1. JSON web token (JWT)

JSON Web Token (JWT) je prema [26] standard za sigurno prenošenje zahtjeva u različitim okruženjima. Njegovu arhitekturu odlikuju jednostavnost, kompaktnost i upotrebljivost. Zbog male veličine *JWT* se može na brz i efikasan način, prenositi putem *URL*-a ili zaglavlja i parametara unutar *HTTP* zahtjeva. Token se sastoji od dva dijela: zaglavlje i podatak (eng. *payload*). Zaglavlje se mora definirati prema *JWT* specifikacijama dok se podatak definira zahtjevima (eng. *claims*) koje opisuju korisnika. Ti zahtjevi mogu biti:

- *Registrirani zahtjevi* – Sastoji se od seta preddefiniranih zahtjeva čije se korištenje preporučuje, ali nije obavezno. Neki od njih su: *iss* (izdavatelj), *exp* (vrijeme isteka), *sub* (predmet), *aud* (publika) i drugi.
- *Javni zahtjevi* – Ne sastoje se od preddefiniranih zahtjeva i podaci se ovdje mogu definirati po volji, ali se oni trebaju registrirati u *IANA JSON Web Token Registry* ili se trebaju definirati kao *URI* kako bi se izbjegli sudari prilikom prijenosa podataka.
- *Privatni zahtjevi* – To su privatni lanci podataka koji služe kako bi se razmijenili podaci između dvije stranke.

Treći dio tokena je potpis koji je zapravo spoj šifriranog zaglavlja i šifriranog podatka koji se zatim dodatno kodiraju s *HMAC SHA256* algoritmom. Rezultat *JWT*-a su tri *Base64-URL* niza znakova koji su razdvojeni s točkama kako bi se lako prenosili putem *HTTP*-a [26].

Potvrda autentičnosti korisnika koristi se kako informacije o projektima i klijentima (koje su poslovna tajna) ne bi procurile u javnost. Time nefunkcionalni zahtjev o sigurnosti sustava ne bi bio ispunjen, a koji je postavljen u potpoglavlju 5.1.

6.2.2. Programsko rješenje ovjere autentičnosti u praktičnom dijelu

Za ovjeru autentičnosti klijenata i njihovih zahtjeva poslužitelj koristi posrednički program (eng. *middleware*) koji se pokreće čim zahtjev dođe do *API Gatewaya*. Ako posrednički program za ovjeru autentičnosti otkrije nepravilnosti u poslanom *JWT-u* poslužitelj klijentu vraća iznimku. Na slici 6.4 prikazan je programski kod za generiranje novog *JWT-a*.

```
/**
 * Generate a JWT token from user entity
 *
 * @param {Object} user
 */
generateJWT(user) {
    const today = new Date();
    const exp = new Date(today);
    exp.setDate(today.getDate() + 1);

    return jwt.sign({
        id: user._id,
        username: user.username,
        exp: Math.floor(exp.getTime() / 1000)
    }, this.settings.JWT_SECRET);
},
```

Sl. 6.4 *Prikaz programskog koda za generiranje novog JSON web tokena.*

JSON Web Token generira se prilikom klijentskog zahtjeva za prijavom ili registracijom. Poslužitelj provjerava može li klijent pristupiti željenoj ruti bez ovjere autentičnosti. Ako je klijent uspio pristupiti ruti bez ovjere autentičnosti poslužitelj provjerava ima li klijent generiran token, ako nema onda se on generira pomoću *generateJWT* metode. Poslužitelj podatke unutar *JWT-a* definira pomoću registriranih zahtjeva. Unutar podataka se definiraju korisnički podaci *ID* i *nadimak* kako bi se smanjio broj provjera i zahtjeva upućenih ka bazi podataka, osim toga se definira i vrijeme istjecanja *JWT-a* i nakon kojega će poslužitelj taj token smatrati nevažećim. Dok drugi argument *jwt.sign* metode označava kojom tajnom riječi će se šifrirati podatak. Zaglavlje i potpis tokena se automatski definiraju. Generirani token se zatim sprema u *HTTP* kolačić koji se zatim šalje prilikom svakog zahtjeva kojeg klijent šalje ka poslužitelju. Na slici 6.5 prikazan je programski kod za ovjeru autentičnosti korisnika koji se izvodi u posredničkom programu. U kontekstu agilnog razvoja *JWT* je iskorišten kako bi se spriječio hakerski napad koji bi mogao uzrokovati curenje informacija o projektima i klijentima u javnost.

```
/**
 * Authorize the request. Check that the authenticated user has right to access the resource.
 *
 * @param {Context} ctx
 * @param {Object} route
 * @param {IncomingRequest} req
 * @returns {Promise}
 */
async authorize(ctx, route, req) {
    let token;
    if (req.headers.authorization) {
        let type = req.headers.authorization.split(" ")[0];
        if (type === "Token" || type === "Bearer")
            token = req.headers.authorization.split(" ")[1];
    }

    if (req.$action.auth == "required" && !token) {
        return Promise.reject(new UnAuthorizedError(ERR_NO_TOKEN));
    }

    // Verify JWT token
    let user;
    if(token) {
        user = await ctx.call("user.resolveToken", { token });
        if (req.$action.auth == "required" && !user) return Promise.reject(new UnAuthorizedError(ERR_INVALID_TOKEN));
        ctx.meta.user = user;
        ctx.meta.token = token;
        ctx.meta.url = req.headers.referer;
    }

    if (req.$action.auth == "required" && !user)
        throw new UnAuthorizedError();
}
```

Sl. 6.5 *Prikaz programskog koda za ovjeru autentičnosti korisnika.*

Kada *API Gateway* od klijenta zaprimi zahtjev, on prvo provjeri rutu kojoj želi klijent pristupiti i treba li ovjeriti autentičnost korisnika za tu rutu. U slučaju da se treba vršiti ovjera autentičnosti, tada se pokreće *authorize* metoda. Na početku metode provjerava se ima li zahtjev zaglavlje u kojem se nalazi *JWT*. Ako nema tokena, a želi se pristupiti ruti koja zahtjeva važeći token tada se vraća iznimka koja opisuje nedostatak važećeg tokena. Zatim se pomoću metode *resolveToken* (koja se nalazi unutar *user* usluge) dohvaća korisnik kojemu pripada taj token. Na slici 6.6 prikazan je programski kod tijela *resolveToken* metode.

```
/**
 * Resolves and gets user by JWT.
 */
resolveToken: {
    cache: {
        keys: ["token"],
        ttl: 60 * 60, // 1 hour
    },
    params: {
        token: "string",
    },
    async handler(ctx) {
        const decoded = await new this.Promise((resolve, reject) => {
            jwt.verify(ctx.params.token, this.settings.JWT_SECRET, (err, decoded) => {
                if (err) return reject(err);
                resolve(decoded);
            });
        });
        if (decoded.id) return this.getById(decoded.id);
    }
},
```

Sl. 6.6 *Prikaz programskog koda tijela metode resolveToken.*

Unutar *resolveToken* metode se vrši provjera je li dani token valjan i ako je valjan, tada se dešifrira tijelo podatka unutar tokena koje sadrži korisničke informacije koje na kraju vraća. Ako metoda *resolveToken* vrati korisničke podatke, oni se spremaju u posebne globalne varijable unutar *Moleculer* u koje se spremaju podaci vezane uz klijente, a koji se zatim mogu koristiti unutar usluga. Ako se korisnički podaci ne vrate iz *resolveToken* metode to znači da *JWT* nije valjan i da poslužitelj treba odbiti zahtjev od klijenta.

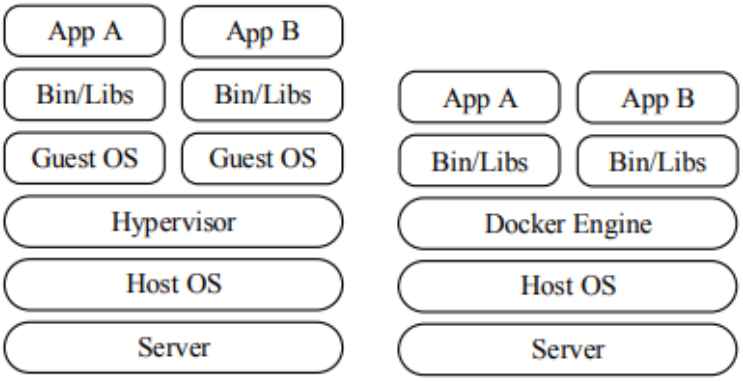
6.3. Skaliranje rada aplikacije uz pomoć Dockera

Skaliranje rada aplikacije uz pomoć tehnologije *Docker* se odrađuje kako bi se ispunio nefunkcionalni zahtjev o dostupnosti usluge jer skaliranjem poslužiteljske aplikacije osigurava bolju i bržu dostupnost aplikacije krajnjim korisnicima. Kako je glavni cilj agilnog načina upravljanja projektima brzo prilagođavanje problemima, potrebna je stalna dostupnost aplikacije kako bi se u svakom trenutku znao trenutni napredak razvoja projekta i mogući problemi pri razvoju.

6.3.1. Tehnologija Docker

Docker [29] je platforma koja se bazira na spremanju aplikacija u kontejnere, što je olakšana tehnologija virtualizacije. Kontejneri su viša razina apstrakcije od modela virtualizacije u virtualnim strojevima, a izoliraju okruženje za vrijeme izvođenja usluge. Kontejneri su efikasniji pri alokaciji resursa za izvođenje od virtualnih strojeva, jer ne pokreću zaseban operacijski sustav za svaku pokrenutu aplikaciju. Instance kontejnera su manje i brže se kreiraju i migriraju od

instanci virtualnih strojeva. Svi kontejneri unutar *Dockera* su izolirani od drugih kontejnera što znači da ako se napravi izmjena na jednom kontejneru ta promjena neće utjecati na druge *Docker* kontejnere. Tehnologiju kontejnera odlikuju sljedeće karakteristike: jednostavno pokretanje aplikacije, moguće pokrenuti kontejner na različitim operacijskim sustavima, visoki stupanj izolacije između kontejnera i visoki stupanj sigurnosti. Na slici 6.7 preuzetoj iz [29] prikazana je usporedba između arhitekture virtualnih strojeva i arhitekture *Dockera*.



Sl. 6.7 *Prikaz usporedba između arhitekture virtualnih strojeva i arhitekture Dockera [29].*

Zbog karakteristika *Docker* tehnologije, najčešće se koristi kada je potrebno brzo isporučiti veliki broj aplikacija. Osim toga, *Docker* se može koristiti za spremanje aplikacije u više kontejnera kako bi se aplikacija mogla pokrenuti na bilo kojoj platformi ili operacijskom sustavu [29].

6.3.2. Prikaz skaliranja aplikacije pomoću Dockera

Pokretanje aplikacije je olakšano, jer okvir *Moleculer* ima *moleculer-runner* modul koji pomaže pri pokretanju napisanih usluga kao *Docker* kontejnere. Podešavanjem datoteke *docker-compose.yml* može se odrediti broj kontejnera i što svaki kontejner sadrži. Ako se dva puta definira ista usluga, tada se putem *NATS-a* i *Traefik-a* odrađuje uravnoteženje opterećenja unutar usluge *API Gateway*. Na slici 6.8 prikazan je primjer datoteke *docker-compose.yml*.

```yaml
version: "3.3"

services:

  api:
    build:
      context: .
    image: formo-backend
    env_file: docker-compose.env
    environment:
      SERVICES: api
      PORT: 3000
    depends_on:
      - nats
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.api-gw.rule=PathPrefix(`/`)"
      - "traefik.http.services.api-gw.loadbalancer.server.port=3000"
    networks:
      - internal

  greeter:
    build:
      context: .
    image: formo-backend
    env_file: docker-compose.env
    environment:
      SERVICES: greeter
    depends_on:
      - nats
    networks:
      - internal

  products:
    build:
      context: .
    image: formo-backend
    env_file: docker-compose.env
    environment:
      SERVICES: products
    depends_on:
      - mongo
      - nats
    networks:
      - internal
  mongo:
    image: mongo:4
    volumes:
      - data:/data/db
    networks:
      - internal

  nats:
    image: nats:2
    networks:
      - internal

  traefik:
    image: traefik:v2.1
    command:
      - "--api.insecure=true" # Don't do that in production!
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
    ports:
      - 3000:80
      - 3001:8080
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - internal
      - default
  networks:
    internal:

  volumes:
    data:
```

Sl. 6.8 *Prikaz primjera docker-compose.yml datoteke za Moneculer.*

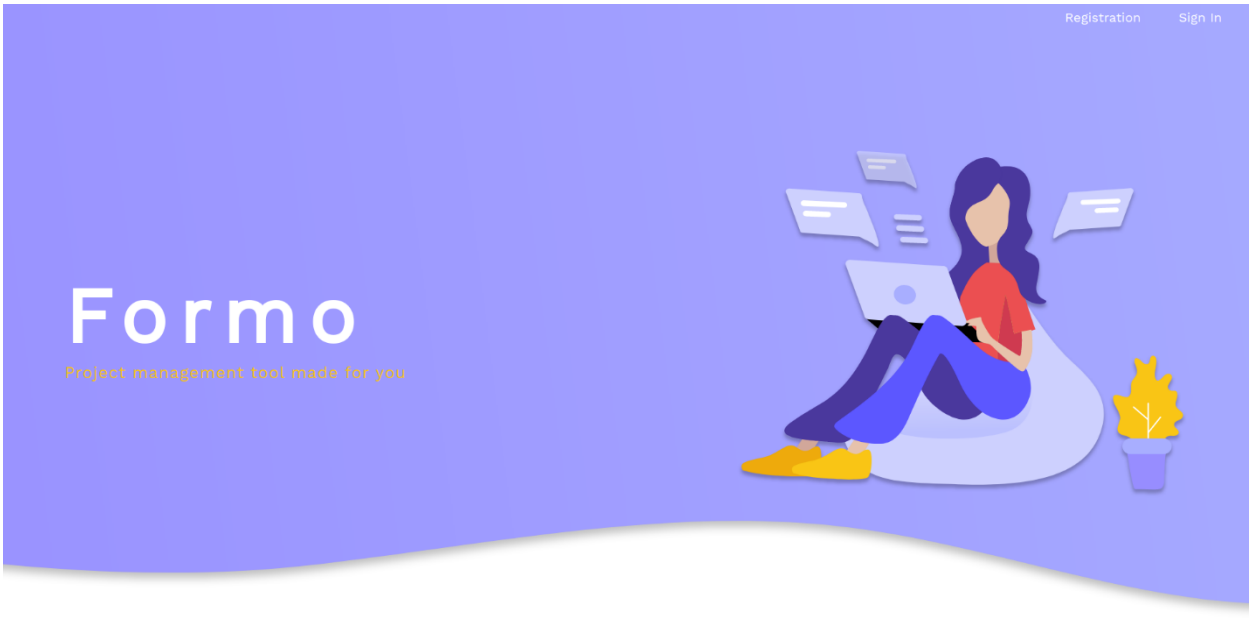
Pokretanjem jednostavne *NPM* naredbe *npm run dc:up* se pokreće *Docker* i iz *docker-compose.yml-a* se konfiguriraju *Docker* kontejneri kojima se može pristupiti kao da se čitava aplikacija pokreće kao jedan proces. Konkretno u praktičnom dijelu diplomskog rada sve mikrousluge su pretvorene u *Docker* kontejnere. Za usluge koje bi mogle imati veću opterećenost se napravio još jedan *Docker* kontejner kako bi unutarnji sustavi integrirani u *Moleculer* mogli automatski uravnotežiti opterećenje usluga. Ti sustavi su *NATS* i *Traefik*, a sprječavaju preopterećenje pojedine usluge tako što prenose dio opterećenja na njene replike. *Docker* kontejneri i čitava *Docker* tehnologija objašnjeni su u potpoglavlju 6.3.1. U ovome slučaju, *Traefik* se koristi isključivo na *API Gateway* usluzi kada je aplikacija skalirana putem *Docker* tehnologije jer *Docker* kontejneri mogu biti pokrenuti na različitim poslužiteljima kako bi se performanse aplikacije dodatno optimizirale. Dio teorije o *NATS*-u je opisan u 3.3 potpoglavlju, a detaljniji način korištenja *NATS*-a unutar praktičnog dijela rada će biti objašnjen unutar 7.3.2 potpoglavlja.

7. OPIS RADA APLIKACIJE I ISPITIVANJE

U ovom poglavlju opisane su upute za korištenje aplikacije na poslužiteljskoj strani kao i ispitivanje i analiza poslužiteljske strane aplikacije. U ovom poglavlju su također prikazane slike i objašnjeno korištenje klijentske strane aplikacije koja je razvijena u diplomskom radu pod nazivom „Programsko rješenje korisničkog dijela web sustava za potporu upravljanja projektima agilnog razvoja programske podrške prilagođenog osobama s oštećenjima vida“.

7.1. Opis rada aplikacije

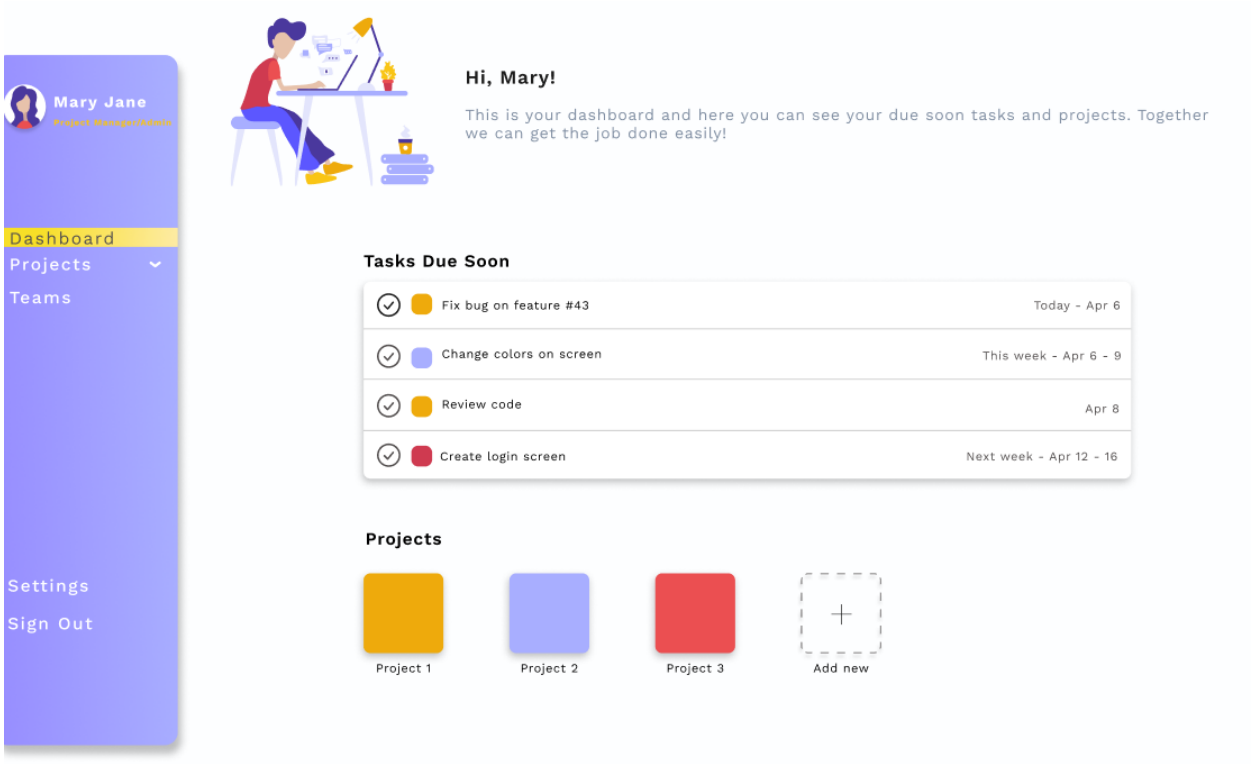
Uz poslužiteljski dio, praktični dio diplomskog rada se sastoji i od web aplikacije. Dolaskom na web aplikaciju korisnik može birati između registracije novog računa ili prijave u postojeći korisnički račun. Na slici 7.1 prikazan je izgled početne stranice projektnog dijela diplomskog rada.



Sl. 7.1 *Prikaz izgleda početne stranice praktičnog dijela diplomskog rada.*

Registracija novog korisnika sastoji se od dva dijela gdje se u prvome unose osobne informacije o korisniku, a zatim i informacije o organizaciji. Prvi korisnik je automatski i vlasnik novoregistrirane organizacije i ima administratorske ovlasti. Korisnik s administratorskim ovlastima zatim registrira račune ostalim zaposlenicima koji dobivaju nasumično generiranu zaporku koja se s ostatkom korisničkih informacija šalje novoregistriranom korisniku na elektroničku poštu. Prijava postojećeg korisnika se odvija tako da korisnik unese adresu elektroničke pošte i zaporku. Nakon prijave, korisnik se prosljeđuje nadzornoj ploči gdje (ovisno

o ranku) može vidjeti projekte na kojima je dodijeljen. Ako korisnik ima rang administratora ili projekt menadžera tada može vidjeti sve projekte unutar organizacije i sve postojeće timove. Na slici 7.2 prikazan je izgled kontrolne ploče.

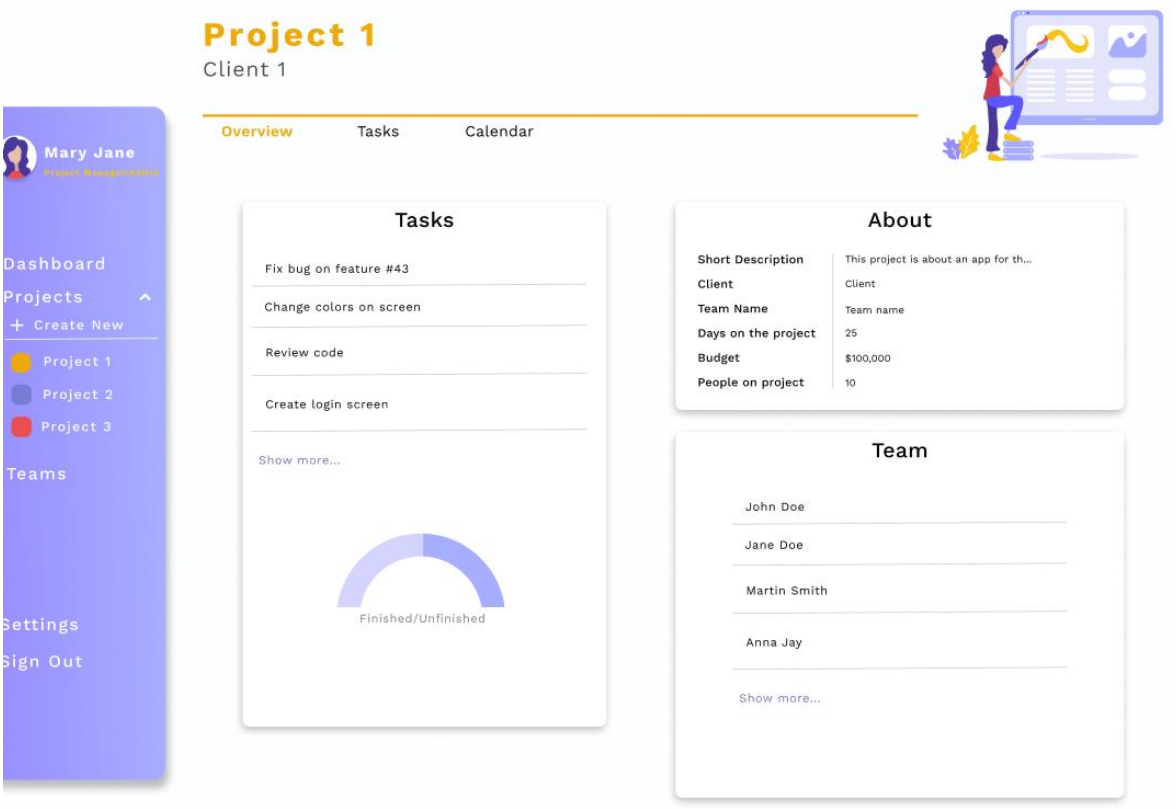


Sl. 7.2 *Prikaz izgleda kontrolne ploče.*

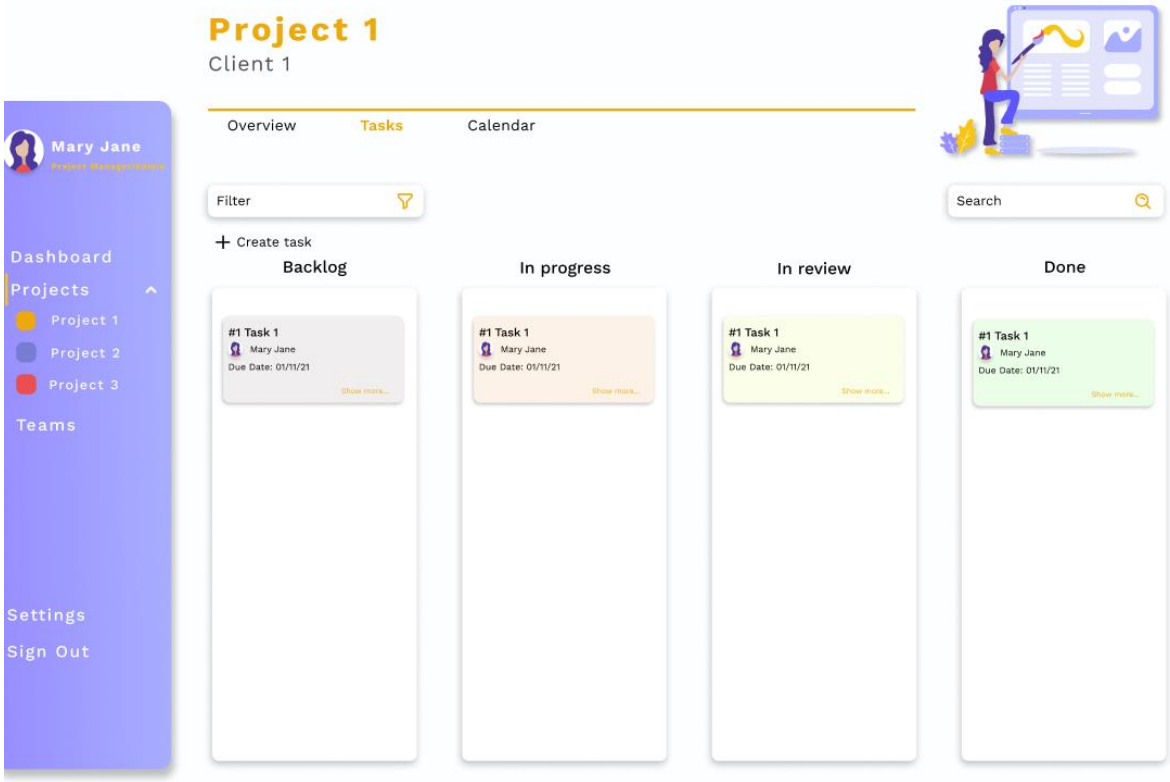
Korisnik može kliknuti na zadatke prikazane na zaslonu i tako dobiti uvid u detalje zadatka koji je njemu dodijeljen. Ispod zadataka korisnik može vidjeti projekte u kojima sudjeluje ili ako ima rang projekt menadžera ili administratora može kreirati nove projekte.

Klikom na projekt unutar liste projekata korisnik može vidjeti detalje odabranog projekta. Na slici 7.3 prikazan je izgled detalja odabranog projekta.

U pregledu odabranog projekta, korisnik može vidjeti zadatke, tim kojem je zadan taj projekt i općeniti podaci o projektu. Osim osnovnog pregleda detalja korisnik može vidjeti trenutno stanje razvoja projekta u *Tasks* tabu i tok razvoja u *Calendar* tabu. Na slici 7.4 prikazan je izgled zadataka i njihovo trenutno stanje unutar projekta.



Sl. 7.3 *Prikaz osnovnog pregleda odabranog projekta.*

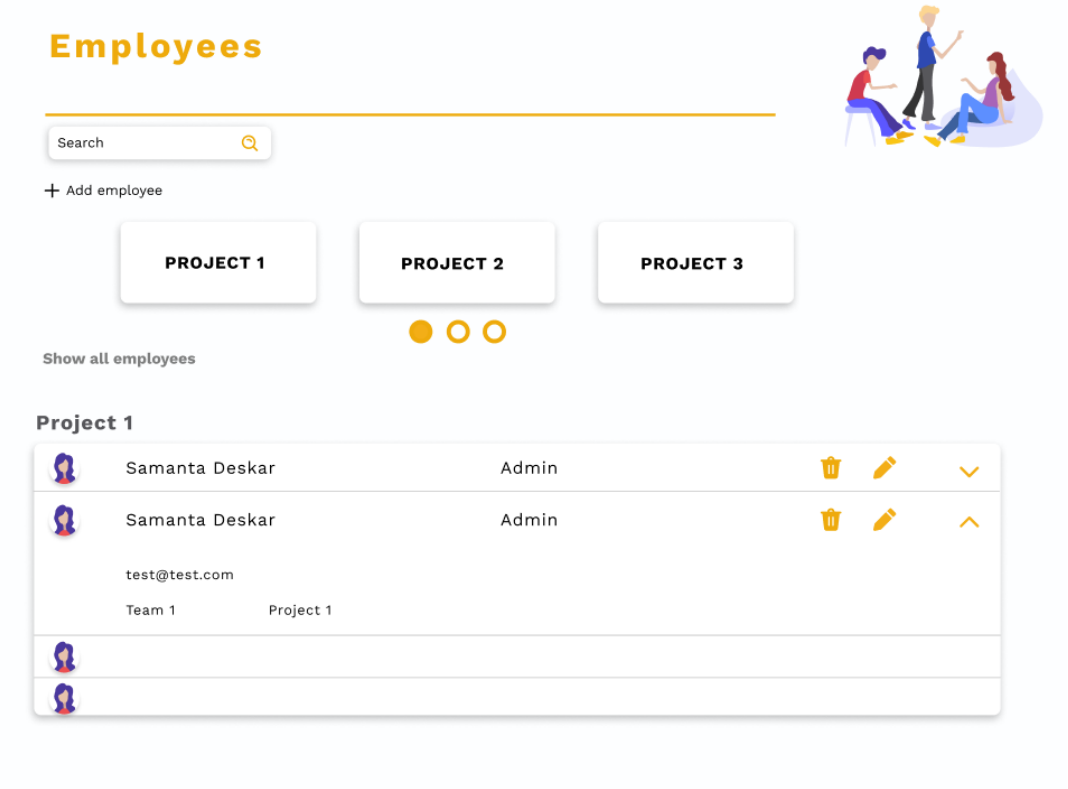


Sl. 7.4 *Prikaz zadataka unutar odabranog projekta.*

Zadaci su podijeljeni na *backlog*, *in progress*, *in review* i *done* te se mogu filtrirati prema korisnicima, prioritetu ili zadanom tipu. Pretraživanje se vrši prema imenu zadatka. Unutar istog pogleda, projekt menadžeri i administratori mogu kreirati nove zadatke.

Prilikom kreiranja novih zadataka korisnik mora unijeti naziv zadatka, datum do kojeg zadatak mora biti gotov, prioritet zadatka. Osim toga mogu se još dodati datoteke važne za taj zadatak i dodijeliti korisnik kojem je dodijeljen taj zadatak. Nakon njegovog kreiranja zadatak se automatski smješta u *backlog* stupac. Klikom na pojedini zadatak korisnik može vidjeti njegove detalje gdje je istaknut naslov i opis zadatka, prikvačene datoteke, prioritet i tip zadatka, komentari i korisnik kojem je dodijeljen zadatak. Svi korisnici mogu komentirati zadatak i prikvačiti datoteke vezane uz zadatak. Učitavanje datoteka na poslužitelj se odvija asinkrono. Po završetku procesa učitavanja okida se događaj nakon kojeg se učitana datoteka provjerava i ako je slika tada se pokreću radne niti i vrši se proces sažimanja i smanjivanja slike kao što je opisano u potpoglavlju 6.1. Komentiranje zadataka se odvija asinkrono gdje se spremaju informacije o autoru teksta, napisani tekst i kojem zadatku poruka pripada. Autori komentara ga mogu urediti ili obrisati, nakon čega se promjene spremaju unutar baze podataka. Administratori i projekt menadžeri mogu dodijeliti bilo kojem korisniku (koji je unutar njihove organizacije) dok korisnik može isključivo samog sebe dodijeliti odabranom zadatku (ako zadatak nema dodijeljenih korisnika). Trenutni status zadatka mogu mijenjati: administrator, projektni menadžeri ili korisnik kojem je dodijeljen zadatak.

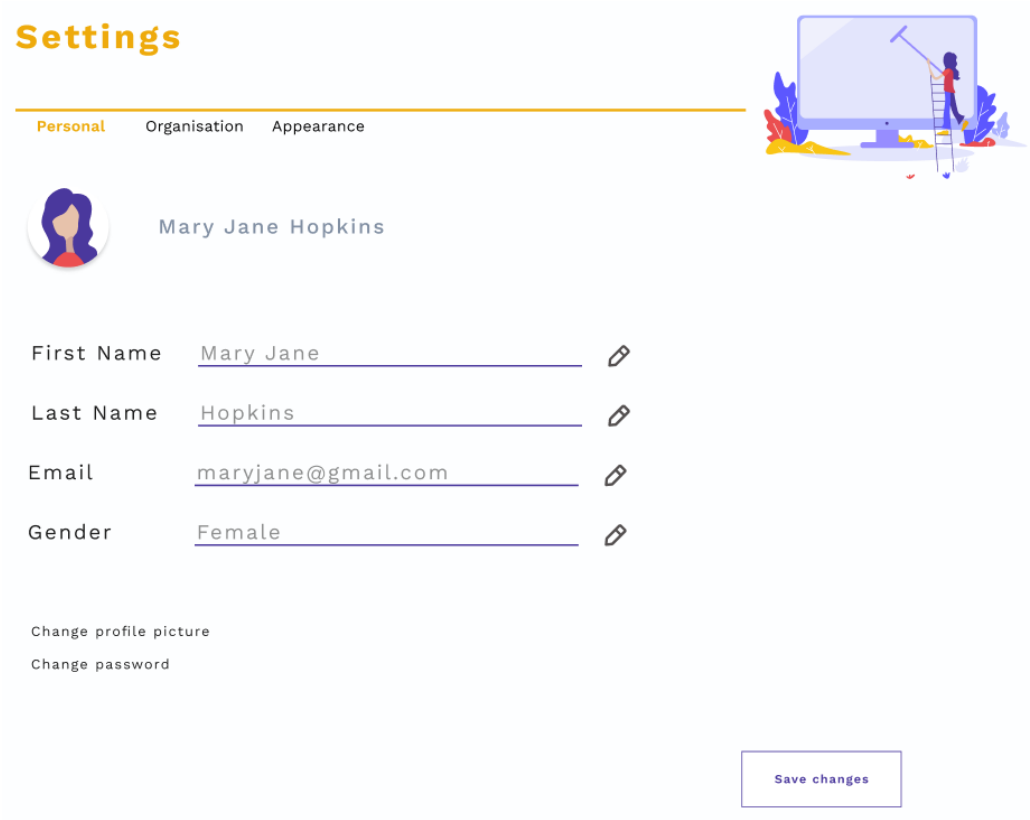
Projektni timovi se kreiraju tako da korisnik (projektni menadžer i administrator) odabere korisnike iz njegove organizacije. Odabirom pogleda svih zaposlenih unutar organizacije korisnik može vidjeti tablicu svih zaposlenika, a iznad koje su prikazane kartice s timovima (po projektima) i gumb za dodavanje novog zaposlenika. Zaposlenici se mogu pretraživati po imenu i prezimenu ili projektu kojem je dodijeljen. Tablica prikazuje osnovne informacije o zaposleniku poput imena i prezimena i ranga, a klikom na redak tablice se on proširuje i mogu se vidjeti ostale informacije o korisniku. Račun odabranog zaposlenika se također može obrisati ili urediti. Na slici 7.5 prikazan je pogled pregleda svih zaposlenih u organizaciji.



Sl. 7.5 *Prikaz pogleda pregleda svih zaposlenih u organizaciji.*

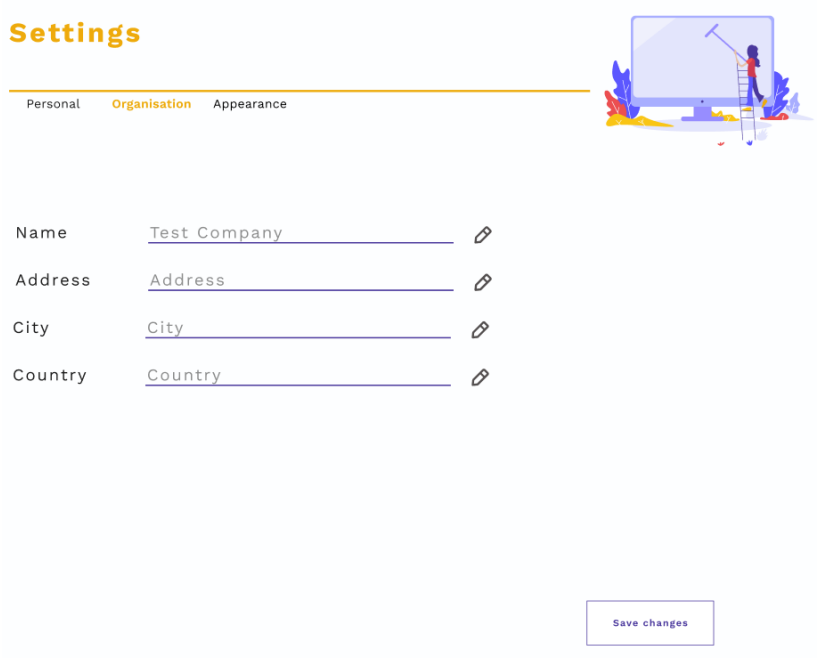
Prilikom kreiranja novog zaposlenika potrebno je unijeti ime i prezime, adresu elektroničke pošte, ulogu i spol. Nakon završetka kreiranja zaposlenika poslužitelj automatski šalje elektroničku poštu s nasumično generiranom zaporkom i drugim osnovnim informacijama računa. Također se unutar elektroničke pošte napominje da korisnik treba što prije promijeniti zaporku zbog dodatne sigurnosti.

Korisnik također može urediti postavke pregleda web aplikacije. On može urediti osnovne informacije, osnovne informacije organizacije (ako ima rang administratora) i izgled web aplikacije. Na slici 7.6 prikazan je pogled za uređivanje osnovnih informacija računa.



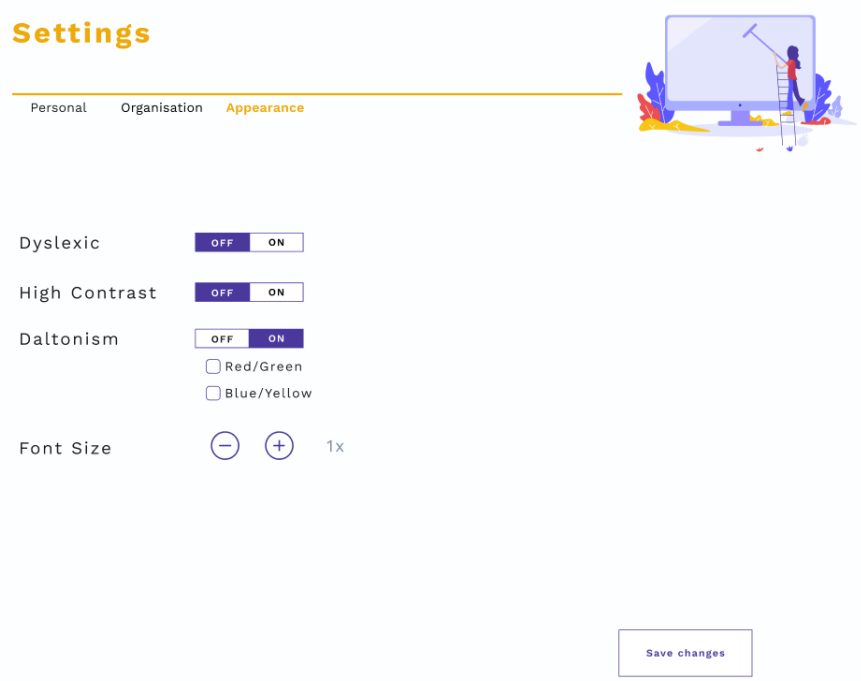
Sl. 7.6 *Prikaz pogleda za uređivanje osnovnih informacija korisničkog računa.*

Korisnik može promijeniti svoje ime, prezime, adresu elektroničke pošte, spol, zaporku i sliku profila. Klikom na gumb za spremanje informacija iste se odmah spremaju u bazu podataka. Slike profila spremaju se u poseban direktorij gdje su sažete i smanjene kako bi se osiguralo ispunjenje nefunkcionalnog zahtjeva koji zahtjeva ekonomično iskorištenje prostora na poslužitelju. Na slici 7.7 prikazan je pogled za uređivanja informacija organizacije.



Sl. 7.7 *Prikaz pogleda za uređivanja informacija o organizacijama.*

Informacije o organizacijama mogu uređivati samo korisnici koji imaju rang administratora u odabranoj organizaciji. Mogu se namjestiti informacije poput: imena organizacije, adrese, grada i države u kojoj se organizacija nalazi. Na slici 7.8 prikazan je pogled za uređivanje izgleda web stranice.



Sl. 7.8 *Prikaz pogleda za uređivanja izgleda web aplikacije.*

Unutar pogleda za uređivanje izgleda web aplikacije korisnik može urediti izgled prema svojim sklonostima. Korisnik može uključiti ili isključiti određene načine izgleda aplikacije za: disleksiju, visoki kontrast i daltonizam. Ti načini izgleda aplikacije olakšavaju korisnicima s određenim poteškoćama korištenje web aplikacije. Osim toga korisnik može povećati ili smanjiti veličinu teksta na web aplikaciji [27].

7.2. Ispitivanje rada aplikacije

7.2.1. Jedinično i integracijsko testiranje aplikacije

Testiranje je postupak koji osigurava da je proizvod napravljen po pravilima i da proizvod ispunjava sve korisničke zahtjeve. Testiranje se sastoji od testiranja prema bijeloj kutiji (eng. *white box*) ili crnoj kutiji (eng. *black box*). Prilikom testiranja prema bijeloj kutiji tester ima pristup izvornom kodu na temelju kojeg piše testove. Testovi su jedinično i integracijsko testiranje. Prilikom testiranja prema crnoj kutiji tester nema pristup izvornom kodu, a testovi su napisani kako bi testirali funkcionalnosti programskog koda. U praktičnom dijelu diplomskog rada obavljeno je testiranje prema bijeloj kutiji, odnosno jedinični i integracijski testovi.

Jedan modul (eng. *unit*) je najmanji dio aplikacije koji se može testirati, a izoliran je od drugih modula. Jedinično testiranje ispituje dali je modul razvijen prema zadanim specifikacijama prije no što je integriran s drugim dijelovima programa. Osim toga jedinično testiranje osigurava: da se programski kod unutar jednog modula ispravno izvodi u svim testnim slučajevima, povećanje kvalitete koda, izgradnju programskog koda koji se može više puta upotrijebiti i pojednostavljenu dokumentaciju programskog koda. Prilikom testiranja modula tester iz programskog koda modula piše testne slučajeve za testiranje tog modula. Tester koji provodi ovaj način testiranja je ujedno i programer koji je pisao izvorni kod za testirani modul. Stoga se pogreške otkrivene ne zapisuju već se odmah ispravljaju.

Integracija je proces spajanja jedne ili više jedinica programskog koda u jednu smislenu cjelinu koja tvori računalni sustav ili aplikaciju. Integracijsko testiranje se provodi kako bi se testirale interakcije između više modula i pokazuje sve pogreške unutar interakcije. Integracijsko testiranje se izvodi nakon jediničnog testiranja, a prije testiranja čitavog sustava. Postoji mnogo strategija integracijskog testiranja, no neke su: *big-bang*, *top-down* i *bottom-up*. Prilikom *big-bang* testiranja sve su jedinice odjednom povezane jedna s drugom. Ova strategija testiranja se koristi kako bi se smanjilo vrijeme trajanja integracijskog testiranja. No, ona nije dobar odabir jer je nemoguće pronaći pogreške unutar pojedinačnog modula jer se fokus stavlja na rad programskog koda u cjelini. Prilikom *top-down* testiranja cjelokupni programski kod se izgrađuje u etapama, prvo se

testiraju moduli koji se pozivaju druge module pa tek onda ostali moduli. Integracija pojedinih modula se testira korak po korak sve do kraja gdje se testeru omogućuje detaljno otkrivanje pogrešaka. Slično *top-down* ispitivanju, *bottom-up* ispitivanje ispituje prvo moduli najniže razine koji se postepeno integriraju s modulima više razine i testiraju. Uz pronalazak pogrešaka pri integraciji modula, ovaj pristup ispitivanja također pomaže pri određivanju razine razvijenog sustava i olakšava izvještavanje o napretku testiranja [28].

Za jedinično i integracijsko testiranje u se koristio okvir za testiranje u *JavaScriptu* pod nazivom *Jest*. Ovaj okvir je korišten prilikom testiranja jer je *Moleculer* već prethodno konfiguriran za korištenje *Jesta*. Svaki *Jest* test se pokreće kao zaseban proces koji se pokreću paralelno čime se smanjuje vrijeme trajanja testiranja.

7.2.2. Rezultati testiranja s analizom

Za prikaz ispitivanja praktičnog dijela diplomskog rada su se napravila dva integracijska testa i jedan jedinični test. Za integracijsko testiranje *API Gateway* modula i njegovih akcija prema usluzi vezanoj uz korisnike se koriste *Jest* i *supertest*. Na slici 7.9 prikazan je dio programskog koda koji testira stvaranje novog korisnika i nove organizacije.

```
it("POST '/api/user/first'", () => {
    return request(apiService.server)
        .post("/api/user/first")
        .send({
            "user": {
                "firstName":"Perica",
                "lastName": "Perić",
                "email":"pperic@gmail.com",
                "settings":[],
                "role": "admin",
                "sex":"male",
                "organisation": "",
                "projects":[]
            },
            "org": {
                "name":"Firmic855aaa",
                "address":"Adresa",
                "city":"Grad",
                "country":"Država",
                "members":[],
                "projects":[]
            }
        })
        .then(res => {
            expect(res.statusCode).toBe(200);

            userId = res.body.user._id;
            userToken = res.body.user.token;
            userOrg = res.body.user.organisation._id;
        });
});
```

Sl. 7.9 *Prikaz dijela programskog koda za testiranje stvaranje novog korisnika i nove organizacije.*

Novi testni slučaj se definira s ključnom riječi *it*, a unutar te funkcije se definiraju ulazi i očekivani izlaz ili izlazi. Zahtjevi ka poslužitelju se šalju putem *supertest* okvira. U ovom slučaju, poslužitelju se šalje zahtjev za registracijom novog korisnika (slanjem *user* objekta) i stvaranje nove organizacije (slanjem *org* objekta). Očekivanje je da će poslužitelj vratiti *HTTP 200* kod koji označava da su novi korisnik i organizacija uspješno stvoreni. Nakon ispunjenja očekivanja se popunjavaju globalne varijable koje se koriste u drugim testnim slučajevima. Napisani integracijski test pokriva 55% funkcija i 52.39% naredbi unutar *API Gateway* modula jer se ne pokrivaju akcije na *upload* ruti koja je vezana uz učitavanje datoteka na poslužitelj.

Drugi integracijski test pokriva testiranje metoda i događaja unutar usluge vezane uz korisnike. Na slici 7.10 prikazan je dio programskog koda za testiranje događaja koji se javlja kada se stvori novi projekt.

```
describe("Testing 'project.created' event:", () => {
    it("Should call event handler", async () => {
        usersService.adapter.updateById = jest.fn();

        await usersService.emitLocalEventHandler("project.created", {
            user: userId2,
            project: {
                color: "#fff",
                name: "Projekt",
                organisation: "123456",
                budget: 100,
                members: [],
                tasks: []
            }
        });

        expect(usersService.adapter.updateById).toBeCalledTimes(1);
        usersService.adapter.updateById.mockRestore();
    });
});
```

Sl. 7.10 *Prikaz dijela programskog koda za testiranje događaja koji se javlja kada se stvori novi projekt.*

Kako bi se testiralo javljanje događaja, potrebno je stvoriti novu metodu koja oponaša (eng. *mocking*) postojeću *updateById* metodu jer se ona javlja na kraju događaja ako su svi uvjeti ispunjeni. Nakon što se unutar programskog koda okine događaj od njega se očekuje javljanje prilagođene metode koja oponaša *updateById* metodu i to točno jedanput. Nakon što je testni slučaj uspješno izvršen, prilagođena metoda se mora vratiti na metodu iz *Moleculer* okvira, a to se obavlja putem *mockRestore* metode. Kada se trenutnim testom testiraju: akcije, metode i događaji iz usluge za korisnike dobije se pokrivenost koda od oko 80% (86.21% su pokrivene metode unutar usluge, a 80.54% je pokrivenost naredbi unutar usluge).

Za jedinično testiranje izabrana je usluga za generiranje verifikacijskih tokena, jer ne poziva druge usluge. Na slici 7.11 prikazan je dio programskog koda za testiranje validacije podataka poslanih ka akciji za generiranje verifikacijskog tokena.

```
it("Should return validation error on generation of new token.", async () => {
    expect.assertions(1);
    try {
        await broker.call("tokens.generate", {
            type: C.TOKEN_TYPE_VERIFICATION,
            owner: '1',
            expiry: 'ada'
        });
    } catch (err) {
        expect(err).toBeInstanceOf(ValidationError);
    }
});
```

Sl. 7.11 *Prikaz dijela programskog koda za testiranje validacije podataka poslanih ka akciji za generiranje verifikacijskog tokena.*

Korištenjem pokušaj-uhvati (eng. *try-catch*) se omogućuje *hvatanje* iznimke prilikom pozivanja *tokens.generate* akcije. Ako je akcija pravilno postavljena validacija će javiti pogrešku jer varijabla *expiry* mora biti cijeli broj, a ne skup znakova. Ako je javljena iznimka tipa *ValidationError* tada se zadovoljava testni slučaj. Svi napisani testni slučajevi su uspješno prošli, a pokrivenost koda unutar *tokens* usluge je oko 86% (82.86% je pokrivenost naredbi unutar usluge, a 90% je pokrivenost metoda unutar usluge). Na slici 7.12 su prikazani rezultati napisanih testova.

```
--------------------------|---------|----------|---------|---------|-------------------------------------------
File                      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------------------|---------|----------|---------|---------|-------------------------------------------
All files                 |   52.39 |    42.68 |   55.29 |   58.81 |
 api.service.js           |   66.67 |    81.82 |   14.29 |   69.57 | 88-113,202
 organisations.service.js |   34.23 |       18 |    38.1 |    40.7 | 112-121,144-170,213-245,260-301,317-319
 projects.service.js      |    8.57 |     3.64 |   22.22 |    9.64 | 86-222,237-313
 tokens.service.js        |   82.86 |    58.82 |      90 |   82.35 | 46,121-131,157
 user.service.js          |   80.54 |    64.71 |   86.21 |    87.5 | 252-254,363-379,463-467,479-483,579
--------------------------|---------|----------|---------|---------|-------------------------------------------

Test Suites: 3 passed, 3 total
Tests:       51 passed, 51 total
Snapshots:   0 total
Time:        3.554 s
Ran all test suites.
```

Sl. 7.12 *Prikaz rezultata napisanih testova.*

Neke akcije unutar usluge *user* zahtijevaju pozivanje akcija iz drugih usluga stoga se ne može izvesti jedinično testiranju već integracijsko testiranje. Kao što je objašnjeno u potpoglavlju 7.2.1, integracijsko testiranje izvodi kada se želi ispitati interakcija između jedne ili više modula. Izvođenjem integracijskog testa nad *user* uslugom se automatski testiraju *organisations* i *projects* usluge jer *user* usluga ovisi o akcijama iz tih usluga. No, kako one nisu ciljano obuhvaćene integracijskim testom, nisu važne prilikom iščitavanja rezultata testiranja pokrivenosti

programskog koda za sve testove. Od pedeset i jednog napisanog testa svi su uspješno izvršeni. Testni slučajevi su sadržavali ispitivanje pravilno postavljenih poziva pojedinih akcija unutar *user* usluge, ali su se testiralo i validaciju klijentskih zahtjeva tako što bi test poslao krivo zadan zahtjev gdje bi test trebao javiti pogrešku prilikom validacije tog zahtjeva.

Ako se prilikom izvođenja testa pojavila neka pogreška ona je automatski ispravljena unutar programskog koda. Greške koje su se pojavile pretežno su vezane uz krivo postavljene uvjete grananja unutar akcija iz usluga *organisations* i *projects*.

7.3. Analiza rada ostvarenog programskog rješenja

7.3.1. Analiza utjecaja višenitnosti na učinkovitost

Višenitnost u slučaju objašnjenom u potpoglavlju 6.1, se ne uvodi zbog ubrzavanja obrade digitalne slike, već kako bi se ta obrada prebacila na radne niti i kako bi poslužiteljev procesor bio slobodan za primanje i obradu drugih klijentskih zahtjeva. Obavljeno je ispitivanje usluge *task.attachments* tako da su se učitavale dvije slike različitih dimenzija. Mjerenje trajanja obrade slike mjerilo se na početku i na kraju poziva *imageCompress* metode unutar koje se vrši smanjivanje i sažimanje slike. U tablici 7.1 prikazani su rezultati testiranja sažimanja i smanjivanja dvije slike sa i bez upotrebe radnih niti.

Tab. 7.1. *Prikaz rezultata testiranja sažimanja i smanjivanja dvije slike sa i bez upotrebe radnih niti.*

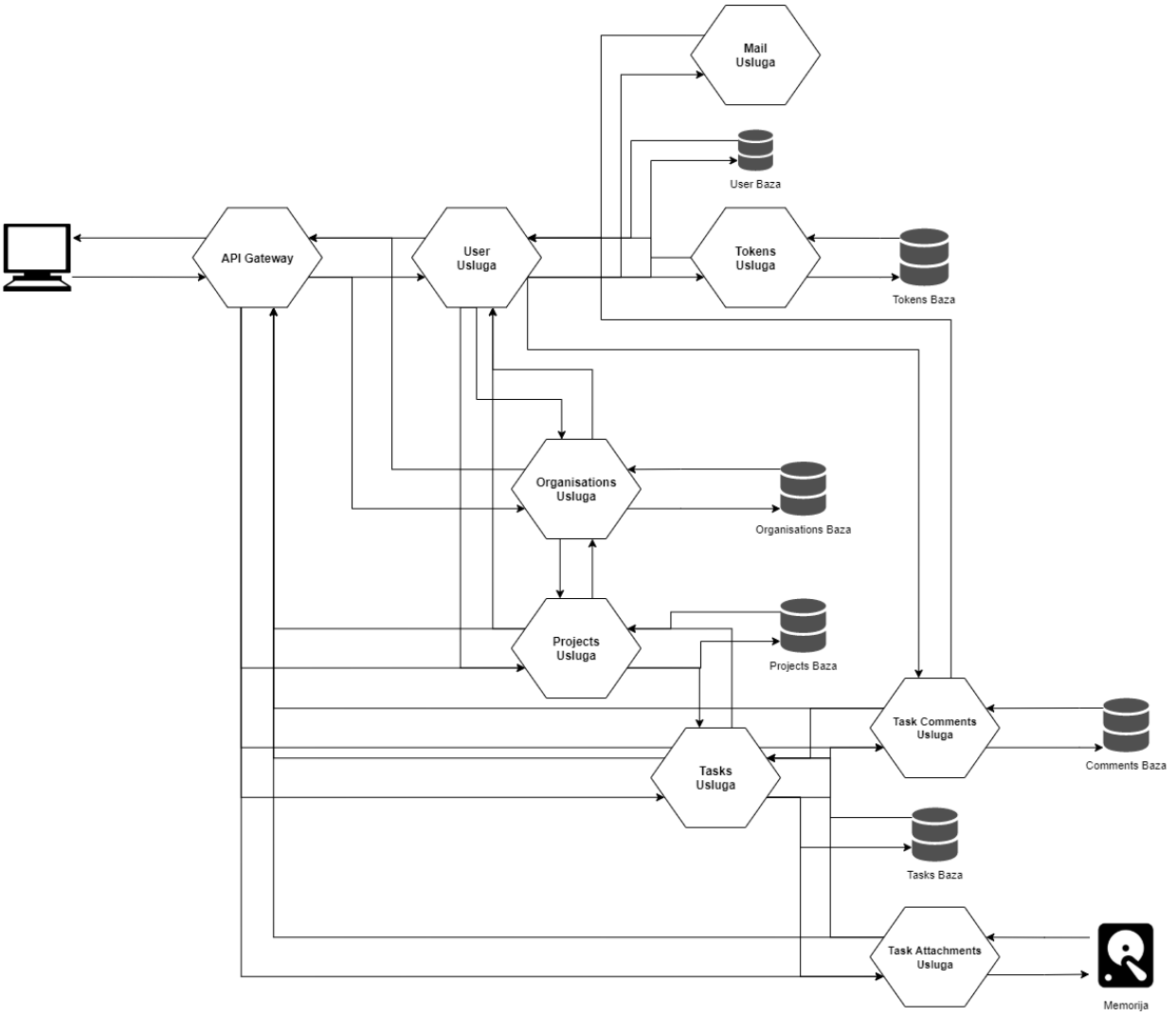
	Trajanje obrade 1920x1280 slike [ms]	**Trajanje obrade 4096x2733 slike [s]**
Bez višenitnosti	948.953	3.952
S višenitnošću	911.418	3.311

Iz rezultata u priloženoj tablici može se primijetiti da je ubrzanje obrade slike otprilike 1%, ali cilj uvođenja višenitnosti u ovome slučaju nije bio povećanje performansi u smislu brzine obrade, već u smislu prevencije blokiranja poslužitelja. Za vrijeme obrade slike bez upotrebe višenitnog programskog koda, ostatak poslužitelja bio je blokiran i nije mogao posluživati klijente podacima iz drugih usluga. Bilo kakvo blokiranje poslužitelja ili povećanje vremena odziva uzrokuje loše korisničko iskustvo na klijentskoj stranici. Za svaki proces obrade slike, alocira se jedna radna nit, jer se samo jednom poziva konstruktor klase *Worker*. U slučaju prikazanom u potpoglavlju 6.1

nije moguće dijeliti sliku na više jednakih dijelova i alocirati više radnih niti, jer se koristi gotovi paket koji ne podržava obradu slike na takav način.

7.3.2. Analiza arhitekture mikrousluga

U praktičnom dijelu rada korištena je arhitektura zasnovana na mikrouslugama, a za modeliranje mikrousluga izabran je uzorak *API Gateway*, koji je objašnjen u potpoglavlju 3.2.3. Usluga *API Gateway* ponaša se kao spona između klijentske aplikacije i ostatka usluga. Poslužiteljska aplikacija je raspodijeljena na osam mikrousluga koje imaju definiranu odgovornost i spojene su s vlastitom baze podataka koja čuva njihove podatke. Jedino usluga *task.attachments* nije spojena na bazu podataka, već svoje podatke sprema na čvrsti disk. Svaka usluga unutar okvira *Moleculer* mora biti instanca klase *ServiceBroker*. Okidanje događaja u svim uslugama, pozivi drugih usluga, validacija zahtjeva, keširanje odgovora, zapisivanje logova, zapisivanje metrika i komunikaciju s transporterom se odvijaju preko mehanizma *ServiceBroker*. Korištenje mehanizma *ServiceBroker* je olakšano putem *moleculer-runner* paketa koji je uključen unutar *Moleculer* razvojnog okvira. Taj paket automatski učitava preddefinirane postavke iz *moleculer.config.js* datoteke i pokreće *ServiceBroker* sa svim uslugama. Prije no što dođe klijentski zahtjev do mikrousluge, on se obrađuje na *API Gateway* usluzi. Svaki zahtjev mora imati svoj *JWT* jer sve akcije na uslugama zahtijevaju da klijentski zahtjev ima valjani *JWT token*. Također neke akcije vrše provjeru da li korisnik ima pravo pristupa na temelju svoga ranga ili nekih drugih uvjeta. Na slici 7.13 prikazana je arhitektura poslužiteljske aplikacije koja je zasnovana na mikrouslugama.



Sl. 7.13 *Prikaz arhitekture poslužiteljske aplikacije napravljene u praktičnom dijelu rada.*

Svaka usluga započinje s definiranjem: imena, *mixinsa* (odnosno klasa koje se kasnije automatski integriraju u programski kod) i postavki. U postavkama usluge se nalaze postavke validatora koje se koriste prilikom manipulacije podacima unutar baze podataka, ali je moguće i definirati rutu do usluge putem API Gatewaya. Sljedeće što svaka usluga posjeduje su:

- **Akcije** – Metode koje se izvode kada ih pozovu druge usluge ili sam *API Gateway*
- **Metode** – Privatne metode dostupne samo usluzi unutar koje se nalaze
- **Događaji** – Poseban sustav prijenosa događaja između usluga putem *Brokera*

Usluga *user* bavi se isključivo manipuliranjem i posluživanjem korisničkih podataka. Baza podataka za tu uslugu se naziva *user* i posjeduje isključivo podatke koji su definirani na početku usluge. Zbog lakšeg manipuliranja podacima spremaju se informacije o: organizaciji, projektima i postavkama. Unutar nje se nalaze akcije vezane uz registraciju i prijavu korisnika u sustav. Prilikom pravljenja novog korisnika usluga *user* zahtjeva od *tokens* usluge token za verifikaciju

korisnika, dok se od usluge *mail* zahtijeva slanje elektroničke poruke dobrodošlice zajedno s nasumično generiranom zaporkom. Nakon korisnikove prijave ili registracije, poslužitelj šalje *HTTP* kolačić koji sadrži *JWT token* s kojim korisnik potvrđuje svoju autentičnost. Proces potvrde autentičnosti je objašnjen u potpoglavlju 6.2. Sve usluge unutar okvira *Moleculer* imaju preddefinirane *REST API* akcije poput: *create* (*POST* zahtjev za pravljenje novog korisnika), *update* (*PATCH* zahtjev za ažuriranje korisničkih informacija), *get* (*GET* zahtjev za dohvaćanje informacija o korisniku), *list* (*GET* zahtjev za dohvaćanje liste registriranih korisnika), *remove* (*DELETE* zahtjev za brisanje korisnika). Preddefinirane akcije odmah rade s odabranom bazom podataka koju usluga koristi, ali se mogu dodatno urediti kako bi se dodale provjere dozvola ili se dodatno manipuliralo podacima. U slučaju dohvaćanja informacija o korisniku, dodatno se manipulira podacima tako što se pozivaju *organisations* i *projects* usluge gdje se dohvaćaju podaci o organizaciji i projektima kojima korisnik pripada. Uz to, postoje još dvije akcije koje dohvaćaju korisničke informacije, a to su: *getbyOrg* i *getBasicData*. Nekada nisu potrebne sve informacije o korisnicima koje dohvaća obična akcija *get* , pa se stoga koristi akcija *getBasicData* koja vraća samo korisnikov: id, ime, prezime i rang. Akcija *getbyOrg* dohvaća sve korisnike koji pripadaju zadanoj organizaciji. Osim toga, usluga *user* posjeduje akcije za slanje elektroničke poruke ukoliko je zaboravljena zaporka i ako je potrebno promijeniti zaporku (korisnik je kliknuo na link iz elektroničke poruke). Akcija *resolveToken* omogućava dohvaćanje informacija o korisniku iz *JWT* tokena ako je on valjan. Osim akcija, usluga *user* ima metode koje osluškuju okidanje određenih događaja u čitavom sustavu, a vezani su uz korisnike. Ako je korisnik dodan unutar neke organizacije okida se događaj *user.orgAdded* koji (ako je valjan korisnik) dohvaća tog korisnika i sprema nove podatke o organizaciji u njegov dokument. Kreiranjem novog projekta okida se događaj *project.created* koji dodaje id projekta u korisnički dokument kojeg sprema u bazu podataka. Događaj *project.removed* se okida kada se projekt obriše, a metoda tada pronalazi sve korisnike koji imaju *ID* obrisanog projekta u svojim dokumentima i ažurira ih s novim informacijama. Brisanje organizacije okida događaj *organisation.removed* koji tada pronalazi sve korisnike koji pripadaju obrisanoj organizaciji i briše njezin *ID* iz njihovog dokumenta, te ažurira bazu podataka s novim informacijama.

Glavna zadaća usluge *mail* je slanje elektroničkih poruka iz drugih usluga na poslužiteljskoj aplikaciji na definiranu elektroničku adresu. Za potrebe testiranja slanja elektroničkih poruka, one se šalju preko usluge Mailtrap. U produkcijskom okruženju koristio bi se neki drugi *SMTP* poslužitelj koji bi slao prave poruke. Usluga *Mail* koristi paket *moleculer-mail* koji sadrži preddefiniranu uslugu za slanje elektroničkih poruka, dok poslužitelj treba samo definirati adresu,

naslov i tekst poruke. Usluga *mail* služi kao spona između usluga na poslužiteljskoj aplikaciji i paketa *moleculer-mail* jer mijenja neke preddefinirane postavke iz paketa.

Glavna zadaća usluge *tokens* je generiranje i čuvanje tokena koji se koriste na poslužitelju. Tokeni mogu biti za verifikaciju ili resetiranje zaporki. Za generiranje tokena koristi se metoda *randomBytes* iz modula *crypto*, a generirani se bajtovi pretvaraju u heksadekadski kod koji se zatim kriptira algoritmom *sha256* i sprema u bazu podataka. Svakih sat vremena *CRON* posao provjerava bazu podataka i briše tokene koji su istekli ili su iskorišteni. Akcija *check* koristi se prilikom provjere postoji li odabrani token i je li on istekao. Usluga *tokens* se koristi isključivo unutar usluge *user*.

Usluga *organisations* manipulira i poslužuje podatke vezane uz organizacije. Usluga ima vlastitu instancu baze *MongoDB* koja u sebi sadrži isključivo podatke vezane uz organizacije. Unutar baze podataka se spremaju sljedeći podaci vezani uz organizaciju: naziv, adresa, grad, država, popis korisnika i popis projekata. Akcija *create* pravi novu organizaciju koju sprema u bazu podataka, a rezultat kreirane baze podataka vraća usluzi *user* koja poziva tu akciju. Samo korisnici s rangom *admin* mogu urediti informacije svoje organizacije ili ju pak obrisati. Putem akcije *addMember* korisnik ranga *admin* dodaje korisnika kao člana organizacije i okida događaj *user.orgAdded* kako bi se ažurirali podaci unutar drugih usluga. Nadalje, putem akcije *removeMember* korisnik ranga *admin* uklanja korisnika iz svoje organizacije. Putem akcije *projects* je moguće vidjeti osnovne informacije (naziv i budžet) o svim projektima koji su unutar zadane organizacije. Usluga *organisations* ima definirane tri metode koje se okidaju kada se istoimeni događaji dogode. Okidanjem događaja *user.removed*, usluga *organisations* će pronaći organizaciju koja u popisu svojih članova ima id obrisanog korisnika i ukloniti ga iz svoje evidencije. Okidanjem *project.created* dohvaća se organizacija kojoj pripada novootvoreni projekt i usluga ga sprema u dokument dohvaćene organizacije. Okidanjem događaja *project.removed* dohvaća se organizacija unutar koje se obrisao projekt i ažurira se njezin popis projekata.

Usluga *projects* manipulira i poslužuje podatke od projekata koji su vezani uz određenu organizaciju. Unutar instance baze podatka spremaju se: boja projekta (kako bi korisnik imao bolje korisničko iskustvo), naziv, id organizacije kojoj projekt pripada, budžet predviđen na projektu (opcionalan podatak), popis korisnika koji rade na projektu, popis zadataka koji su vezani uz projekt. Projekte mogu napraviti isključivo korisnici koji su ranga *admin* ili *project manager*, a naziv projekta mora biti unikatan. Informacije o samom projektu se dohvaćaju putem akcije *get* ,a zbog privatnosti jedino korisnici koji se nalaze unutar organizacije čiji je projekt mogu vidjeti

njegove informacije. Projekt se može obrisati putem akcije *remove*, ali pristup toj akciji imaju jedino korisnici koji su unutar organizacije i koji imaju rang *admin* ili *project manager*. Događaj *user.removed* se okida kada korisnik obriše svoj korisnički račun, a tada usluga *projects* pronalazi sve projekte koji su dodijeljeni obrisanom korisniku i brišu njegov id s liste korisnika. Događaj *task.created* se okida kada se kreira novi zadatak, tada se dohvaća projekt za koji je vezan otvoreni zadatak i dodaje ga na listu zadataka. Događaj *task.removed* se okida kada se neki zadatak obriše, tada se dohvaća projekt koji ima id obrisanog zadatka u listi zadataka i briše ga s liste. Događaj *organisation.removed* se okida kada se obriše organizacija, a onda se brišu svi projekti koji su vezani uz tu organizaciju.

Usluga *tasks* manipulira i poslužuje podatke od zadataka koji su vezani uz određeni projekt. Unutar svoje baze podataka sprema informacije vezane uz sam zadatak poput: njegovog statusa (*backlog*, *wip*, *review*, *done*), njegovog naziva, njegovog opisa, člana tima kojem je dodijeljen zadatak, krajnji rok do kojeg zadatak mora biti završen, id projekta za koji je vezan zadatak, prioritet zadatka, lista id-eva komentara vezanih uz zadatak i lista putanja do datoteka učitanih na poslužitelj. Zadatak mogu napraviti korisnici s rangom *admin* ili *project manager* putem akcije *create* nakon čega se odašilje *task.created* događaj svim aktivnim uslugama unutar *ServiceBroker-a*. Informacije vezane uz zadatak se ažuriraju putem akcije *update*, a tu akciju mogu pokrenuti svi korisnici koji su unutar projekta. Informacije o projektu koje su prilagođene prikazu na klijentskoj aplikaciji su dostupne putem akcije *get*. Korisnici s rangom *admin* ili *project manager* mogu odabrani zadatak obrisati putem akcije *remove*. Nakon završetka brisanja podataka vezanih u zadatak, okida se događaj *task.removed* kako bi i ostale usluge mogle ažurirati svoje podatke. Putem akcije *addMember* se korisnici dodjeljuju odabranom zadatku, a korisnici mogu sami sebe dodijeliti ili ih mogu nadređeni dodijeliti zadatku. Koristeći akciju *removeMember* akciju korisnici mogu maknuti korisnika sa zadatka. Događaj *project.removed* se okida kada se projekt obriše, a tada se brišu i zadaci koji su vezani uz taj projekt i okidaju *task.removed* događaj kako bi druge usluge ažurirale svoje podatke. Događaj *user.removed* se okida kada neki korisnik obriše svoj korisnički račun, a tada se on miče sa svih zadataka za koje je zadužen. Događaj *task.comment.create*d se okida kada korisnik postavi novi komentar na zadatku, a on se tada dodaje na listu id-eva komentara od tog zadatka. *Task.comment.removed* događaj se okida kada se obriše postavljeni komentar na određeni zadatak, a tada se id tog komentara briše iz liste postavljenih komentara. Doagađaj *task.attachment.uploaded* se okida kada se učita nova datoteka na poslužitelj i tada se putanja te datoteke dodaje u polje putanja ostalih datoteka koje su već prikvačene za

zadatak. Događaj *task.attachment.removed* događaj se okida kada se obriše prikvačena datoteka s poslužitelja, a tada se putanja do nje briše iz liste putanja prikvačenih datoteka.

Usluga *task.comments* manipulira i poslužuje podatke vezane uz komentare koji su ostavljeni na nekom zadatku. Unutar svoje baze podataka se spremaju informacije vezane uz komentare poput: autora (id korisnika), teksta komentara, zadatak na koji je ostavljen komentar. Novi komentar se sprema putem akcije *create*, a mogu ga ostaviti svi korisnici unutar projekta pod uvjetom da odabrani zadatak postoji. Nakon što se završi spremanje novog komentara, okida se događaj *task.comment.created* kako bi ostale usluge mogle ažurirati svoje podatke. Komentare mogu ažurirati korisnici s rangom *admin* ili *project manager* putem akcije *update*. Osnovne informacije, prilagođene za klijentsku aplikaciju, mogu dobiti svi korisnici na projektu putem akcije *get*. Komentar mogu obrisati autor komentara ili korisnici s rangom *admin* ili *project manager*. Događaj *task.removed* se okida kada se zadatak obriše, a tada se svi komentari brišu. Događaj *user.removed* se okida kada se korisnički račun obriše, a tada se svi komentari kojima je autor obrisani korisnik ažuriraju tako da se umjesto id-a autora stavlja autorovo ime i prezime.

Slanjem datoteke s klijentske strane na uslugu *API Gateway* putanjom *upload* se pokreće usluga *task.attachments*. Ona manipulira i poslužuje podatke vezane uz datoteke učitane na poslužitelj i koje su prikvačene uz određeni zadatak. Usluga nema direktnu vezu s bazom podataka već svoje podatke sprema na čvrsti disk samog poslužitelja. Nakon što korisnik završi učitavanje datoteke na poslužitelj, automatski se poziva *save* akcija iz usluge *task.attachments*. Nakon što se završi spremanje niza bajtova na poslužiteljev čvrsti disk vrši se provjera radi li se o slikovnoj datoteci ili običnoj. Provjera se odvija tako što se uspoređuju *MIME* informacije datoteke i ako je ekstenzija datoteke jednaka ekstenzijama slikovnih datoteka (koje su preddefinirane u listi). Tada se poziva modul za sažimanje i smanjivanje slika. Slikovna datoteka se može dohvatiti putem *get* akcije koja pretražuje poslužiteljev čvrsti disk i kao rezultat vraća otvoren niz bajtova koje internet preglednik korisniku pretvara u sliku. Događaj t*ask.removed* se okida kada se neki zadatak obriše, a tada se sve slikovne datoteke brišu s poslužiteljevog čvrstog diska.

8. ZAKLJUČAK

U ovome diplomskom radu istražene su programske tehnologije koje su korištene za razvoj programskog rješenja poslužiteljskog dijela sustava za potporu upravljanju projektima agilnog razvoja. Ovaj diplomski rad bavi se poslužiteljskom stranom, dok diplomski radom pod nazivom „Programsko rješenje korisničkog dijela web sustava za potporu upravljanja projektima agilnog razvoja programske podrške prilagođenog osobama s oštećenjima vida“ ima za cilj razvoj klijentskog dijela sustava. Nakon istraživanja i analize mogućnosti postojećih sustava za upravljanje projektima, istražene su i predložene tehnologije potrebne za ispunjavanje zahtjeva poslužiteljskog dijela sustava. Za modeliranje usluga unutar arhitekture mikrousluga izabran je uzorak *API Gateway* koji je podržan razvojnim okvirom *Moleculer*. S obzirom na to da aplikacija nije složena, koristi se samo jedna usluga *API Gateway* koja povezuje klijente i mikrousluge na poslužitelju. Višenitnost se koristi prilikom sažimanja i smanjivanja slika s ciljem ubrzavanja tih postupaka bitnih za agilni razvoj programske podrške. Također, predložene tehnologije koriste se za ostvarenje višenitnog načina rada poslužiteljske strane aplikacije i to unutar razvojne okoline *Node.js*.

Nakon izrade aplikacije obavljeno je testiranje aplikacije na razini jedinica koda i integracijsko testiranje ostvarenih usluga. Uz to, analiziran je utjecaj višenitnosti na učinkovitost rada web aplikacije i ostvarena arhitektura mikrousluga. Korištenje arhitekture zasnovane na mikrouslugama značajno je olakšalo ostvarenje programskog rješenja, jer je programski kod razdvojen u više manjih datoteka specifične zadaće. Nadalje, korištenjem uzorka *API Gateway* mogu se definirati načini obrade zahtjeva, ali i ispuniti neke od sigurnosnih zahtjeva na rad aplikacije. Analiza rada poslužiteljskog dijela sustava pokazala je da sustav ispunjava postavljene funkcionalne i nefunkcionalne zahtjeve. Dodatna poboljšanja ovog dijela web sustava podrazumijevala bi poboljšanje konfiguriranje postavki okvira *Moleculer* i njemu pripadajućeg *API Gatewaya*, dodatne mjere zaštite podataka unutar baza podataka, kao i mogućnost podizanja programskog rješenja u skalabilnu okolinu oblaka računala.

LITERATURA

[1] M. C. Layton, S. J. Ostermiller, Agile Project Managment For Dummies, John Wiley & Sons, Inc., Hoboken, New Jersey, 2017.

[2] K. Schwaber, J. Sutherland, Scrum Guide, https://scrumguides.org/scrum-guide.html [pristupljeno 9. srpnja 2021.]

[3] International Scrum Institute, The Kanban Framework, https://www.scrum-institute.org/contents/The_Kanban_Framework_by_International_Scrum_Institute.pdf [pristupljeno 10. srpnja 2021.]

[4] Asana, Features, https://asana.com/ [pristupljeno 14. srpnja 2021.]

[5] Jira, Agile Project Managment Tools For Software Teams, https://www.atlassian.com/software/jira/agile [pristupljeno 14. srpnja 2021.]

[6] Active Collab Wiki, https://everipedia.org/wiki/lang_en/Active_Collab [pristupljeno 15. srpnja 2021.]

[7] Monday.com, Work OS: The Visual Platform That Manages Everything, https://monday.com/product/ [pristupljeno 16. srpnja 2021.]

[8] Information Techonology, Asana vs. Monday: Comparing Two Big Names In Project Management, https://technologyadvice.com/blog/information-technology/asana-vs-monday/ [pristupljeno 18. srpnja 2021.]

[9] C. de la Torre, B. Wagner, M. Rousos, .NET Microservices: Architecture for Containerized .NET Applications, Microsoft Developer Division, 2020.

[10] M. Fowler, PolyglotPersistence, https://martinfowler.com/bliki/PolyglotPersistence.html, [pristupljeno 22. lipnja 2021.]

[11] C. D. Nguyen, A Design Analysis of Cloud-based Microservices Architecture at Netflix, https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f, [pristupljeno 23. lipnja 2021.]

[12] M. Pallidino, The Difference Between API Gateways and Service Mesh, https://www.cncf.io/blog/2020/03/06/the-difference-between-api-gateways-and-service-mesh, [pristupljeno 25. lipnja 2021.]

[13] Icebob, Moleculer, a Progressive Microservices Framework for Node.js, https://medium.com/moleculer/moleculer-a-modern-microservices-framework-for-nodejs-bc4065e6b7ba [pristupljeno 21. srpnja 2021.]

[14] Moleculer, Networking, https://moleculer.services/docs/0.14/networking.html [pristupljeno 21. srpnja 2021.]

[15] Moleculer, Core Concepts, https://moleculer.services/docs/0.14/concepts.html [pristupljeno 21. srpnja 2021.]

[16] C. S. Horstmann, Core Java 11th Edition, Pearson, London, Ujedinjeno Kraljevstvo, 2018.

[17] A. Silberschatz, P. B. Galvin, Operating System Concepts 8th Edition, Addison-Wesley, Boston, USA, 2009.

[18] S. Tandon, Tuning Tomcat For a High Throughput, Fail Fast System, https://netflixtechblog.com/tuning-tomcat-for-a-high-throughput-fail-fast-system-e4d7b2fc163f [pristupljeno 7. kolovoza 2021.]

[19] Simform, What is Node.js? Where, When & How To Use It (With Examples), https://www.simform.com/what-is-node-js/ [pristupljeno 7. kolovoza 2021.]

[20] Node.js Documentation, Worker Threads | Node.js v16.6.1, https://nodejs.org/api/worker_threads.html [pristupljeno 7. kolovoza 2021.]

[21] MySQL 8.0 Reference Manual, What is MySQL, https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html [pristupljeno 9. kolovoza 2021.]

[22] MongoDB, What is NoSQL? NoSQL Databases Explained | MongoDB, https://www.mongodb.com/nosql-explained [pristupljeno 9. kolovoza 2021.]

[23] MongoDB, Comparing the Differences - MongoDB vs MySQL | MongoDB, https://www.mongodb.com/compare/mongodb-mysql [pristupljeno 9. kolovoza 2021.]

[24] A. Nawo, Node.js File Streams Explained!, https://areknawo.com/node-js-file-streams-explained/ [pristupljeno 12. kolovoza 2021.]

[25] The Economic Times, What is Authentication? Definition of Authentication, https://economictimes.indiatimes.com/definition/authentication [pristupljeno 14. kolovoza 2021.]

[26] JWT.io, JSON Web Token Introduction, https://jwt.io/introduction [pristupljeno 14. kolovoza 2021.]

[27] S. Deskar, Programsko Rješenje Korisničkog Dijela Web Sustava za Potporu Upravljanja Projektima Agilnog Razvoja Programske Podrške Prilagođenog Osobama s Oštećenjima Vida, diplomski rad, Fakultet elektrotehnike, računarstva i informacijskih tehnologija, 2021.

[28] B. Hambling, P. Morgan, A. Samaroo, G. Thompson, P. Williams, Software Testing: Software Testing: An ISTQB-ISEB Foundation Guide 4th Edition, BCS, The Chartered Institute for IT, London, Ujedinjeno Kraljevstvo, 2019.

[29] Q. Ma, J. Han, Z. Xu, X. Yang and Q. Li, "Docker-based Simulation Training System on Dispatching and Control Cloud," 2019 IEEE Innovative Smart Grid Technologies - Asia (ISGT Asia), 2019, pp. 1216-1220, doi: 10.1109/ISGT-Asia.2019.8881708.

[30] O. Havazík and P. Pavlíčková, "How to design Agile game for education purposes in JIRA," 2020 7th International Conference on Control, Decision and Information Technologies (CoDIT), 2020, pp. 331-334.

[31] L. Filion, N. Daviot, J. Le Bel and M. Gagnon, "Using Atlassian Tools For Efficient Requirements Management: An Industrial Case Study," 2017 Annual IEEE International Systems Conference (SysCon), 2017, pp. 1-6.

[32] S. Chickerur, A. Goudar and A. Kinnerkar, "Comparison of Relational Database with Document-Oriented Database (MongoDB) for Big Data Applications," 2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA), 2015, pp. 41-47.

[33] S. Hassan, N. Ali and R. Bahsoon, "Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity," 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 1-10.

[34] M. J. Bedy, S. Carr, Xianglong Huang and Ching-Kuang Shene, "The design and construction of a user-level kernel for teaching multithreaded programming," FIE'99 Frontiers In Education. 29th Annual Frontiers In Education Conference. Designing The Future Of Science And Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011, 1999, pp. 13A3/24-13A3/29 Vol. 2.

SAŽETAK

U ovom diplomskom radu su analizirani i opisani postupci agilnog razvoja programske podrške, kao i mogućnosti postojećih alata i okolina. Opisana je i predložena arhitektura mikrousluga, za čiju implementaciju je korišten okvir Moleculer. Opisan je način ostvarivanja paralelizma unutar Node.js okruženja. Uspoređene su relacijske i nerelacijske baze podataka, s gledišta učinkovitosti pohrane i učitavanja podataka. Koristeći predložene metode i tehnologije, programski je ostvarena poslužiteljska strana aplikacije za pružanje potpore upravljanju projektima agilnog razvoja. Aplikacija manipulira i poslužuje podacima koji su potrebni za vođenje projekata (npr. spremanje i prikaz informacija o korisnicima, projektima i zadacima). Nakon izrade aplikacije obavljeno je njeno testiranje na razini jedinica koda i integracijsko testiranje ostvarenih usluga koristeći se okvirima Jest i Supertest. Koristeći tehnologiju Docker i arhitekturu mikrousluga, moguće je poslužiteljsku aplikaciju pokrenuti na različitim poslužiteljima tako da se usluge pretvore u Docker kontejnere čime se ostvaraju kvalitetno korisničko iskustvo.

Ključne riječi: agilni razvoj, arhitektura mikrousluga, relacijske i neralacijske baze podataka, testiranje, višenitnost.

ABSTRACT

Title: Backend Solution of Web System for Agile Project Management Based on Multi Thread Parallelism And Microservice Architecture

This thesis analysis and describes the procedures in the agile development of software, as well as the possibilities of existing tools and environments. The described and proposed microservice architecture, for which the Moleculer framework was used. The way of achieving parallelism within the Node.js environment is also described. Relational and non-relational databases were compared, from the point of view of data storage and data loading efficiency. The proposed methods and technologies were used to create a server-side application to support the management of agile projects. The application manipulates and serves the data needed to manage projects (storing and displaying information about users, projects, and tasks). After the development of the application, the unit and the integration tests were performed with the Jest and Supertest frameworks. Using Docker containers and microservice architecture, it is possible to run a server application on different servers thus achieving a quality user experience.

Keywords: agile development, API Gateway, Jest, microservice-based architecture, multithreading, relation and non-relation databases, testing.

ŽIVOTOPIS

Matej Arlović rođen je 5. Svibnja 1996. godine u Osijeku. U Osijeku završava „Osnovna škola Ljudevita Gaja“, nakon čega se upisuje u „Elektrotehnička i prometna škola Osijek“ zanimanja Mehatroničar. Po završetku srednje škole upisuje preddiplomski stručni studij Elektrotehnike, smjer Automatika kao redovan student na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. Na drugoj godini studija pohađa PHP Akademiju koju su pokrenuli tvrtka Inchoo i FERIT. Tijekom ljeta iste godine pohađa praksu u tvrtki Inchoo kao backend programer. 2018. godine stječe akademski naziv sveučilišni prvostupnik (lat. baccalaureus) inženjer elektrotehnike, smjer Automatika. Nakon završetka razlikovnih obveza, upisuje diplomski sveučilišni studij Programsko inženjerstvo.

_____

Potpis autora

PRILOZI (NA CD-U):

Prilog 1: Diplomski rad u .pdf formatu.

Prilog 2: Diplomski rad u .docx formatu.

Prilog 3: Izvorni kod projekta dostupnog na GIT repozitoriju.