Razvoj mikroservisa korištenjem jezika Go

Bračić, Darijo

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Darijo Bračić

Razvoj mikroservisa korištenjem jezika Go

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Darijo Bračić

Matični broj: 0016156370

Studij: Informacijski i Poslovni Sustavi

Razvoj mikroservisa korištenjem jezika Go

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Marko Mijać

Varaždin, Rujan 2024.

Darijo Bračić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Mikroservisna arhitektura je pristup razvoju softvera koji se oslanja na razbijanje velikih i kompleksnih sustava na manje, neovisne i samostalne servise. Svaki od ovih servisa funkcionira autonomno, ali je sposoban komunicirati s drugim servisima putem API-ja. Mikroservisna arhitektura omogućava veću fleksibilnost, lakše upravljanje i brži razvoj pojedinačnih komponenti softvera.

Jezik Go je popularan izbor za razvoj mikroservisa zbog svoje efikasnosti, jednostavnosti i sposobnosti da se lako integrira s različitim alatima i infrastrukturama.

U teorijskom dijelu rada detaljno su opisane ključne karakteristike mikroservisne arhitekture, uključujući nezavisnosti servisa. Također, objašnjene su osnovne funkcionalnosti i prednosti jezika Go u kontekstu razvoja mikroservisa, s posebnim naglaskom na njegovu sintaksu, tipove podataka, funkcije rutine i pakete.

U praktičnom dijelu rada razvijen je jedan ili više mikroservisa koristeći jezik Go. Prikazano je kako se teoretski koncepti mikroservisne arhitekture primjenjuju u stvarnim aplikacijama. Praktični primjeri uključuju dizajn API-ja, implementacija funkcija, upravljanje podacima, te metode osiguravanja sigurnosti i skalabilnosti.

Ključne riječi: API,Mikroservis,GO,Arhitektura,Softver

Sadržaj

1.Uvod

U ovom završnom radu naglasak je stavljen na mikroservisnu arhitekturu i jezik Go. Mikroservisna arhitektura postaje sve važnija u modernom razvoju softverskih aplikacija zbog svoje sposobnosti da razdvoji složene sustave na manje, nezavisne komponente poznate kao mikroservisi. Mikroservisna arhitektura predstavlja pristup u razvoju softverskih aplikacija koji omogućuje da se kompleksne sustave dekonstruira u manje.

Mikroservisi su dizajnirani tako da funkcioniraju neovisno jedni o drugima, ali su u stanju efikasno komunicirati putem dobro definiranih aplikacijskih programskih sučelja (API-ja), što dovodi do veće modularnosti, lakše održivosti i bolje skalabilnosti softverskih rješenja.

Jedan od ključnih izazova u implementaciji mikroservisa jest odabir odgovarajućeg programskog jezika i alata koji podržavaju njihove zahtjeve. U ovom kontekstu, programski jezik Go pokazao se kao vrlo efikasan alat za razvoj mikroservisa. Go je razvijen od strane Googlea i brzo je stekao popularnost među programerima zbog svoje jednostavne sintakse, visoke performanse i izvrsne podrške za konkurentno izvršavanje. Uz to, Go je jezik otvorenog koda (engl. open source), što znači da je dostupan širokoj zajednici programera i ima veliki broj alata i biblioteka koji olakšavaju razvoj softverskih rješenja. Njegova sposobnost da skalira u velikim sustavima i podržava paralelno izvršavanje procesa čini ga idealnim izborom za razvoj mikroservisa.

Programski jezik Go odabran je kao tema zbog svoje iznimne učinkovitosti u radu s mikroservisima. Uspoređujući ga s drugim jezicima kao što su Java, Python ili Node.js, Go se izdvaja zbog svoje brzine i jednostavnosti, kao i zbog snažne podrške za paralelno programiranje, što je od ključne važnosti u distribuiranim sustavima poput mikroservisa. Korištenjem jezika Go, moguće je razviti mikroservise koji su lagani, brzi i jednostavni za održavanje, što ga čini idealnim izborom za moderne aplikacije u oblaku i drugim okruženjima.

Ciljevi ovog rada su analizirati ključne karakteristike mikroservisne arhitekture i prikazati prednosti korištenja jezika Go u razvoju mikroservisa. Teoretski dio rada fokusirat će se na opisivanje osnovnih značajki mikroservisa, kao što su modularnost, skalabilnost i jednostavnost održavanja. Također, bit će objašnjene prednosti mikroservisnog pristupa u odnosu na tradicionalne monolitne arhitekture. U tom kontekstu, posebna pažnja bit će posvećena ulozi jezika Go, njegovim osnovnim funkcionalnostima, poput upravljanja paralelnim procesima, te njegovoj primjeni u razvoju distribuiranih sustava.

Praktični dio rada obuhvatit će implementaciju mikroservisa koristeći programski jezik Go. Kroz konkretne primjere bit će demonstrirano kako se koriste osnovni principi mikroservisne arhitekture, od dizajna API-ja do upravljanja podacima i osiguravanja skalabilnosti i sigurnosti sustava. Praktični primjeri će uključivati razvoj jednostavnog mikroservisa, njegovo povezivanje s bazom podataka te demonstraciju kako se koristi Go-ov ugrađeni alat za upravljanje paralelnim zadacima.

Struktura ovog rada podijeljena je na nekoliko ključnih dijelova. U prvom poglavlju, teorijski će biti obrađeni koncepti mikroservisne arhitekture, uključujući pregled njezinih ključnih značajki i prednosti. U drugom poglavlju, bit će analiziran programski jezik Go, s naglaskom na njegovu primjenu u razvoju distribuiranih sustava. Treće poglavlje rada bit će posvećen praktičnoj implementaciji mikroservisa koristeći Go, gdje će se kroz konkretne primjere demonstrirati kako teorijske koncepte primijeniti u praksi. Na kraju rada bit će prezentirani zaključci i razmatranja o efikasnosti Go-a u razvoju mikroservisnih rješenja, uz preporuke za daljnja istraživanja.

2.Mikroservisna arhitektura

Monolitna arhitektura predstavlja pristup u kojem se cijela softverska aplikacija gradi kao jedinstveni, nerazdvojivi sustav, u kojem su svi moduli međusobno povezani i ovisni jedni o drugima. Tradicionalna monolitna arhitektura često stvara probleme u samom razvojem softvera, pogotovo ako se u softveru javljaju greške. Tada je potrebno otkloniti grešku, ispraviti problematični segment i ponovno objaviti cijeli kod na poslužitelj [1]. Ako sustav koristi više klijenata (web klijent, mobilni klijent, itd.), rad na svim klijentima bit će obustavljen dok se problem ne riješi, što dovodi do smanjenja performansi sustava. Još jedan problem s ovakvom arhitekturom je teškoća podjele projekata na segmente koji bi omogućili članovima tima da razvijaju softver relativno neovisno jedni o drugima. Da bi se smanjila složenost velikih sustava, mnoge organizacije prelaze na mikroservisnu arhitekturu, koja predstavlja evoluciju servisno orijentirane arhiterkture (SOA) i sastoji se od skupa neovisnih servisa, svaki od kojih implementira jedan poslovni zahtjev [1]. Ovakav dizajn i pristup omogućuje brži i pouzdaniji razvoj softvera što poboljšava efikasnost i inovativnost [2].

Ključne prednosti ovoga stila arhitekture su:

- Neovisnost u razvoju i raspoređivanju
- Modularnost
- Mogućnost korištenja različitih tehnologija
- Skalabilnost
- Veća agilnost i specijalizacija

Jedan od glavnih izazova mikroservisne arhitekture je upravljanje složenošću distribuiranih sistema, gdje svaki servis može koristiti vlastiti skup tehnologija i baze podataka, što može dovesti do težeg održavanja dosljednosti podataka i upravljanja transakcijama.

2.1. Što su mikroservisi?

Mikroservisi predstavljaju arhitetonski stil u kojem se aplikacija sastoji od niza malih, neovisnih servisa koji međusobno komuniciraju putem dobro definiranih sučelja, obično API-ja. Svaki mikroservis je specijaliziran za jednu specifičnu funkcionalnost i može se razvijati, testirati, implementirati i skalirati neovisno o drugim servisima unutar sustava.

Karakteristike mikroservisa:

- Mikroservisi su mali, nezavisni, te mali tim programera može razvijati i održavati jedan servis
- Tim može razvijati ažurirati jedan servis bez da se nešto dogodi cijelom sustavu, i bez potrebe da se opet cijeli sustav ponovno implementira
- Servisi su odgovorni za čuvanje podataka, što ih razlikuje od monolitnih modela, odnosno tradicionalnih modela gdje odvojeni sloj upravlja čuvanjem podataka
- Servisi komuniciraju jedni s drugima koristeći API-je. Interne implementacije servisa su skriveni od drugih servisa
- Svaki servis može biti napisan u različitim tehnologijama.

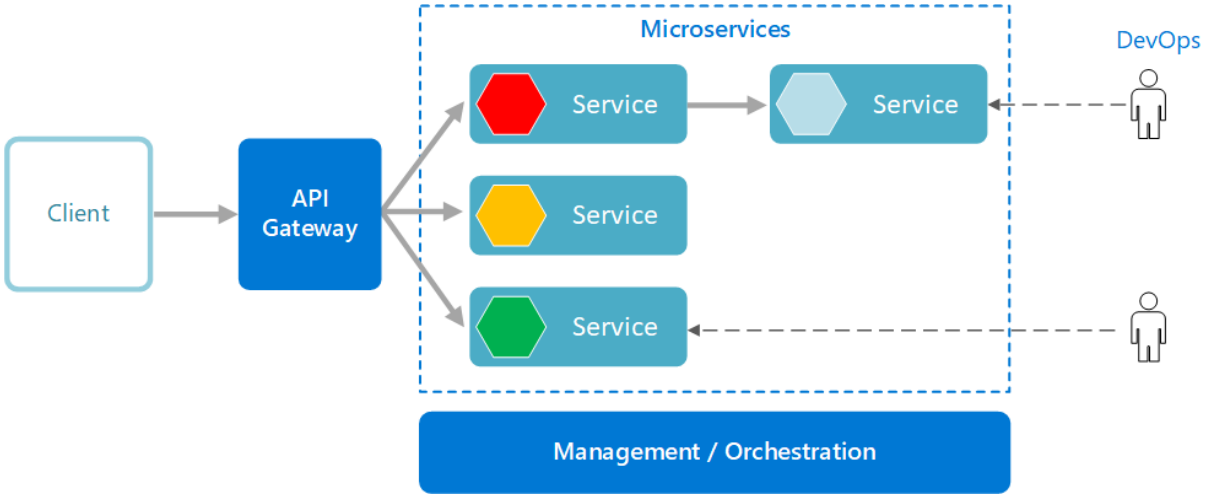
Mikroservisi se mogu implementirati koristeći različite programske jezike i infrastrukture. Najvažniji tehnološki izbori su načini komunikacije između mikroservisa (sinkrona, asinkrona, integracija korisničkog sučelja) i protokoli koji se koriste za komunikaciju (RESTful HTTP, messaging, GraphQL, itd.). Za razliku od tradicionalnih sustava, gdje izbor programskog jezika utječe na cijeli sustav, pristup odabiru tehnologija u mikroservisima je znatno drugačiji [2].

Implementacija mikroservisne arhitekture je složen proces. Postoji mnogo aspekata koje mikroservisna arhitektura mora riješiti. Na primjer, Netflix je razvio okvir za mikroservise kako bi podržao svoje interne aplikacije, a kasnije je objavio mnoge dijelove tog okvira kao izvorni kod [2].

Pored samih servisa, u tipičnoj mikroservisnoj arhitekturi pojavljuju se i druge komponente:

- **Upravljanje** je zaduženo za smještanje servisa na čvorove, identifikaciju kvarova, balansiranje opterećenja servisa i slično. Tipično ovu komponentu realizira tehnologija poput Kubernetesa.
- **API Pristupnik(Api Gateway)** je ulazna točka za klijente. Umjesto da klijenti direktno pozivaju različite servise, oni šalju zahtjeve API Gateway-u, koji zatim prosljeđuju te zahtjeve odgovarajućim servisima na pozadinskoj strani.

Jednostavni prikaz rada mikroservisa dan je na slici 1.



Slika 1 Prikaz rada mikroservisa

2.2. Prednosti i nedostaci mikroservisne arhitekture

Mikroservisna arhitektura donosi brojne prednosti u odnosu na tradicionalne monolitne sustave, što je i razlog njezine sve veće popularnosti u modernom razvoju softvera.
U nastavku ćemo razmotriti ključne prednosti mikroservisne arhitekture, koje je čine prikladnom za implementaciju.

Prednosti mikroservisne arhitekture:

- **Agilnost**: Zato što se mikroservisi raspoređuju nezavisno, lakše je upravljati greškama i implementacijom novih funkcija. Servis možemo ažurirati bez ponovnog raspoređivanja cijele aplikacije. U tradicionalnim aplikacijama kada se pronađu greške cijeli sustav se zaustavi.
- **Mali fokusirani timovi**: Mikroservis bi trebao biti dovoljno malen da je jedan tim zadužen za njegov razvoj. Male veličine timova promoviraju veću agilnost, dok su veliki timovi u mikroservisnoj arhitekturi manje produktivni, jer komunikacija teće sporije, a agilnost se smanjuje.
- **Mala baza koda**: Ne dijeljenjem koda ili skladišta podataka, arhitektura mikroservisa minimizira zavisnosti, što olakšava dodavanje novih funkcija.
- **Mješavina tehnologija**: Timovi mogu izabrati tehnologiju koja najbolje odgovara njihovom servisu, time koristeći skup tehnologija.
- **Skalabilnost**: Servisi se mogu skalirati nezavisno, omogućavajući skaliranjem podsistema koji zahtijevaju više resursa.

- **Izolacija podataka**: Mnogo je lakše izvoditi ažuriranja shema, jer je pogođen samo jedan mikroservis. U monolitnoj aplikaciji, ažuriranje shema može biti veoma izazovno, jer različiti dijelovi aplikacije moraju koordinirati izmjene bez rizika.

Nedostaci mikroservisne arhitekture :

- **Kompleksnost**: Aplikacija zasnovana na mikroservisima ima više pokretnih dijelova nego ekvivalentna monolitna aplikacija. Iako je svaki servis pojednostavljen, cjelokupni sistem je kompleksniji.
- **Razvoj i testiranje**: Pisanje malog servisa koji zavisi od drugih zahtjeva drugačiji pristup od pristupa prema monolitnoj aplikaciji. Postojeći alati često nisu dizajnirani za rad sa servisnim zavisnostima. Također, testiranje servisnih zavisnosti, posebice na većem softveru, može biti izazovno.
- **Nedostatak upravljanja**: Aplikacija ili softver s mikroservisnom arhitekturom može završiti s mnogo različitih jezika i okvira što aplikaciju čini teškom za održavanje.
- **Zagušenje mreže**: Koristeći mnogo malih, detaljno definiranih servisa može rezultirati većim brojem nepotrebnih međuservisnih komunikacija. Također, ako lanac zavisnosti servisa postane predugačak, dodatna latencija može postati problem. Da bi se ovo izbjeglo potrebno je pažljivo dizajnirati API-je, to jest potrebno je izbjegavat API-je koji previše komuniciraju.
- **Integritet podataka**: Svaki mikroservis je odgovoran za osobno čuvanje podataka, kao rezultat toga konzistentnost podataka može biti izazovno.
- **Verzioniranje**: Ažuriranje servisa ne smije poremetiti servise koji zavise od njega. Više servisa može biti ažurirano u bilo kojem trenutku, pa može biti problem s kompatlitabinošću kako unazad tako unaprijed.

2.3. Razlike između monolitne i mikroservisne arhitekture

Sljedeća tablica prikazuje glavne razlike između monolitne i mikroservisne arhitekture:

Tablica 1. Razlike između monolitne i mikroservisne arhitekture

Mikroservis	Monolitna
Mikroservisi dopuštaju neovisnu skalabilnost servisa rezultirajući boljim perfomansama i dostupnošću.	Monolitna je jednostavnija za razvoj,testiranje i implementaciju kao jedinstvenu jedinicu.
Dizajnirano za kontinuiranu implementaciju, omogućujući timovima da brzo izdaju nove funkcionalnosti.	Lakše je upravljati i nadzirati jer su svi dijelovi čvrsto povezani.
Zahtijeva dodatnu infrastrukturu i troškove za upravljanje različitim uslugama.	Nije tako skalabilna kao arhitektura mikroservisa jer su svi dijelovi dio istog sustava.
Testiranje svih usluga postaje složenije kako raste broj funkcionalnosti i api-ja.	Nije tako fleksibilan kao mikroservisi jer su svi dijelovi međusobno ovisni i koriste isti tehnološki stog.

2.4. Upravljanje mikroservisima

Upravljanje u mikroservisima je proces koji omogućava koordinaciju više mikroservisa u svrhu postizanja zajedničkog cilja [3]. Takav proces se može postići preko različitih alata i okvira, kao što je AWS Step Functions, Kubernetes i slični.

Ključne funkcije ovakvog procesa su :

1. **Raspoređivanje (engl. Deployment):** Upravljanje verzijama i implementacija novih servisa ili ažuriranje postojećih servisa. Alati poput Helm (za Kubernetes) ili Docker Compose olakšavaju upravljanje rasporedom aplikacija.
2. **Praćenje i Evidencija (engl. Monitoring and Logging)**: Kontinuirano praćenje performansi, otkrivanje grešaka i prikupljanje logova.
3. **Upravljanje Konfiguracijom**: Upravljanje konfiguracijskim datotekama i postavkama servisa. Alati poput Consul i Spring Cloud Config omogućuju centralizirano upravljanje konfiguracijom.
4. **Otkrivanje Servisa**: Automatizirano otkrivanje lokacija servisa unutar mreže. Alati kao što su Consul i Eureka omogućuju dinamičko otkrivanje servisa.
5. **Sigurnost:** Implementacija sigurnosnih politika, autentikacije i autorizacije. Alati poput Istio i Linkerd pružaju sigurnosne značajke kao što su mTLS (mutual TLS) i kontrola pristupa.
6. **Upravljanje Greškama (Fault Management)**: Otkrivanje, izolacija i oporavak od grešaka.

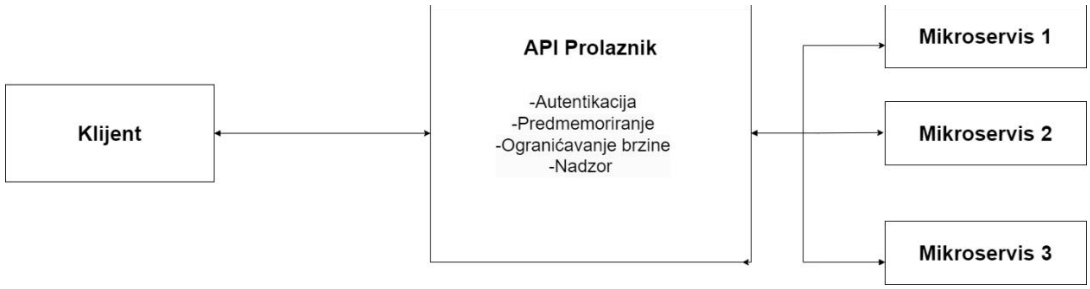
U kontekstu upravljanja mikroservisima bitan je odabir alata za upravljanje. Dvije poznate opcije su Kubernetes i Docker Swarm. Kubernetes je vodeća platforma za upravljanje kontejnerima razvijena od strane Googlea [3]. Pruža sustav za automatsko raspoređivanje, skaliranje i upravljanje kontejnerskim aplikacijama. Ključne značajke Kubernetesa su: Pods, Services i Deployments. Pods su najmanja jedinica koja se može rasporediti, a koja sadrži jedan ili više kontejnera. Services su apstrakcije koje omogućuju stabilnu mrežnu komunikaciju između različitih podova. Deployments služe za upravljanje verzijama aplikacija.

Druga opcija je Docker Swarm, Dockerova platforma za upravljanje mikroservisima [3], fokusirana na jednostavnost i integraciju sa Dockerom. Ključne značajke korištenja ovog alata su: Manager and Worker Nodes te Services and Tasks. Manager and Worker nodes služe za podjelu čvorova na upravljačke (manager) i radne (worker) za učinkovito raspoređivanje zadataka. Services and Tasks definiraju kako aplikacije trebaju biti raspoređene i koje zadatke

trebaju izvršavati. Svaki od ovih alata za upravljanje mikroservisima nudi specifične značajke i pogodnosti koje mogu značajno unaprijediti upravljanje i skalabilnost distribuiranih aplikacija.

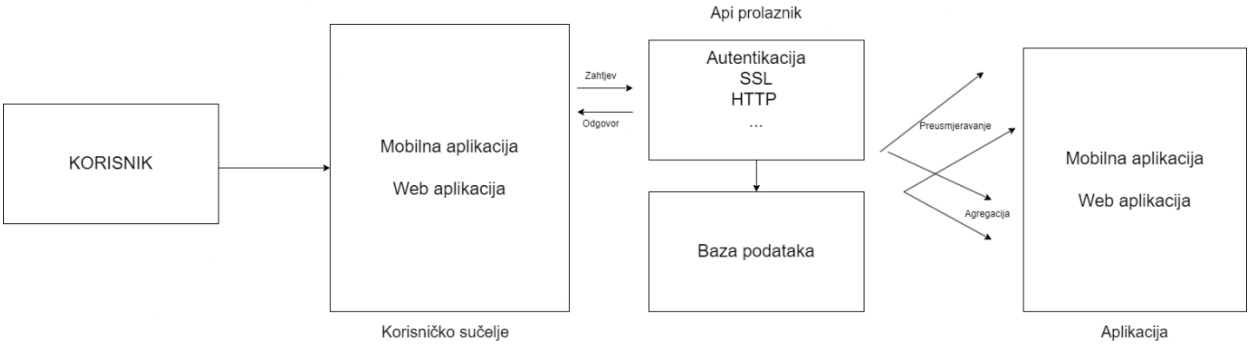
2.5 API pristupnik

Api pristupnik se ponaša kao obrnuti proxy. Obrnuti proxy je posrednički poslužitelj koji preusmjerava zahtjeve klijenata na jedan ili više poslužitelja [4]. Ima jedinstvenu ulaznu točku, te skriva kompleksnu implementaciju backenda. Agregira višestruke zahtjeve u jedan, smanjuje broj API poziva i pojednostavljuje komunikaciju između klijenta i mikroservisa (Slika 3).



Slika 2. Prikaz Api prolaznika s mikroservisima

Već smo spomenuli da je temelj mikroservisne arhitekture rastavljanje aplikacije ili softvera na male, neovisne servise fokusirane na specifične sposobnosti. Međutim, distribuirani pristup također uvodi i značajne izazove, kako iz tehničke, tako i iz organizacijske perspektive. API prolaznik je posebno dizajniran za rješavanje ovakvih izazova (Slika 2).



Slika 3Opis rada API prolaznika, precrtano s Geek for Geeks[11]

Ključne značajke API Gateway:

- **Usmjeravanje (routing) –** Slanje zahtjeva klijenta prema odgovarajućim mikroservisima.
- **Sigurnost -** Implementacija autentifikacije i autorizacije**.**
- **Kontrola prometa** - Balansiranje opterećenja, caching, ograničavanje brzine, itd.
- **Upravljanje -** Otkrivanje servisa, upravljanje greškama poput pokušaja ponovnog slanja zahtjeva i prekida kruga**.**
- **Opservabilnost -** Zapisivanje događaja, praćenje i praćenje**.**
- **Transformacija -** Prijevod protokola, oblikovanje odgovora, itd**.**

Koraci korištenja API Gatewaya:

1. **Usmjeravanje**: API Gateway određuje koji servis ili mikroservis treba obraditi dolazni zahtjev na osnovu URL putanje, HTTP metode, ili zaglavlja [4].
2. **Prijevod Protokola (Protocol Translation)**: API prolaznik može prevesti dolazne zahtjeve iz jednog protokola u drugi, na primjer, prihvatiti HTTP zahtjeve od klijenata i prevesti ih u gRPC ili WebSocket zahtjeve za backend servise.
3. **Agregacija Zahtjeva (Request Aggregation)**: API prolaznik može objediniti zahtjeve u jedan poziv kako bi poboljšao efikasnost i smanjio broj povratnih putovanja potrebnih za ispunjenje zahtjeva.
4. **Autentikacija i Autorizacija**: API prolaznik može rukovati autentifikacijom i autorizacijom za dolazne zahtjeve, provjeravajući identitet klijenta i da li klijent ima potrebne dozvole za pristup traženim resursima.
5. **Ograničavanje brzine i usporavanje**: API prolaznik može provoditi politike ograničenja brzine i kontrolu prometa kako bi spriječio zloupotrebu i osigurao pravilnu upotrebu resursa ograničavanjem broja zahtjeva koje klijent može napraviti unutar određenog perioda.
6. **Ravnoteža Opterećenja (Load Balancing)**: API prolaznik može distribuirati dolazne zahtjeve na više instanci servisa kako bi osigurao visoku dostupnost i skalabilnost.
7. **Spremanje u priručnu memoriju (Caching)**: API prolaznik može privremeno spremiti odgovore iz backend servisa i direktno ih poslužiti klijentima za naknadne identične zahtjeve, čime se postiže efikasnije izvođenje.
8. **Evidencija (Monitoring and Logging)**: API prolaznik može prikupljati metrike i logove za dolazne zahtjeve, pružajući uvid u upotrebu i učinak sistema.

API prolaznik igra ključnu ulogu u povezivanju decentralizirane mikroservise u cjelovite poslovne aplikacije. On ne samo da pomaže u prevladavanju tehničkih prepreka koje donosi distribucija, već stvara red za organizacijsko upravljanje [3].

Prednosti korištenja API prolaznika u mikroservisima:

- **Optimizacija iskustva krajnjeg korisnika**: Frontend aplikacije više ne treba izravno integrirati s brojnim backend mikroservisima. Gateway-i minimiziraju složenost tako što se pristupa kroz jednu ulaznu točku.
- **Implementacija skalabilnosti Distribuiranih sistema**: API Gatewayi implementiraju skalabilnosti konzistentno za sve servise, tako da organizacija ne mora svaki put iznova raditi temelje prije implementacije poslovne logike.
- **Poboljšana sigurnost**: API prolaznik može pomoći u poboljšanju sigurnosti aplikacija pružanjem centralizirane točke autentifikacije i autorizacije. Također, može pružiti zaštitu od DDoS napada i drugih vrsta zlonamjernih aktivnosti filtriranjem i praćenjem prometa prije nego što stigne do mikroservisa.
- **Ubrzanje inovacija:** Promiče brzinu pružajući funkcionalnosti kao ponovno upotrebljive resurse izvan uobičajenih okvira.
- **Centralizirano upravljanje politikama**: Kroz API prolaznik, organizacije mogu centralizirano upravljati politikama kao što su ograničenje broja zahtjeva (rate limiting), kontrola pristupa, te pravila za preusmjeravanje i preoblikovanje zahtjeva, čime se olakšava administracija i održavanje.
- **Poboljšana efikasnost**: API prolaznik omogućava optimizaciju prometa i smanjenje latencije koristeći priručnu memoriju, kompresiju odgovora, i agregaciju zahtjeva. Na ovaj način, smanjuje se opterećenje na backend mikroservisima i povećava ukupna efikasnost sustava.
- **Podrška za više protokola:** API prolaznik može podržavati različite protokole komunikacije (npr. HTTP, WebSocket), omogućavajući fleksibilnost u načinu na koji aplikacije komuniciraju s backend mikroservisima.
- **Monitoring i analiza:** API prolaznik može prikupljati i pružati detaljne metrike i logove vezane uz korištenje API-ja, što omogućava bolji uvid u performanse i ponašanje sustava te olakšava otkrivanje i rješavanje problema.

3. Osnove Go programskog jezika

U ovom poglavlju opisani su koncepti koji se odnose na objašnjenje rada i sintakse programskog jezika Go, te kako se pojedini koncepti ostvaruju uz pomoć ovog jezika. Programski jezik Go (ili Golang) je relativno nov jezik, koji je razvijen u Google-u 2007. godine. Iako posuđuje ideje iz postojećih jezika, posjeduje jedinstvena svojstva koja ga razlikuju programe od onih napisanih u srodnim jezicima poput C++ ili Jave [9]. Cilj stvaranja Go-a bio je adresirati probleme skalabilnosti i performansi. Pisanje programa u jeziku Go zahtijeva drugačiji način razmišljanja i pristup problemima. Na primjer, korištenje paradigmi i stilova programiranja iz Jave ili C++ u Go-u vjerojatno neće rezultirati zadovoljavajućim performansama ili kvalitetom koda. Na primjer, Java programi su dizajnirani za rad u JVM okruženju i koriste specifične biblioteke i sintaksu koje nisu optimalne u Go ekosistemu [9].
S druge strane, razmišljanje o problemu iz Go perspektive može proizvesti uspješan i efikasan program, ali vrlo različit od onoga što bi se napisalo u drugim jezicima. Go promiče jednostavnost, efikasnost i lakoću održavanja koda, što se vidi u njegovoj sintaksi i standardnim bibliotekama. Ključne karakteristike Go-a, kao što su go-rutine za konkurentnost, ugrađena podrška za sinkronizaciju putem kanala, i stroga pravila o upravljanju memorijom, omogućavaju razvoj robusnih i visokoučinkovitih aplikacija koje su prilagođene za moderne razvojne potrebe. Korištenje ovih koncepata zahtijeva promjenu pristupa i razumijevanja specifičnih prednosti i ograničenja koje Go pruža.

Prednosti Go-a:

- **Jednostavnost i čitljivost**: Go je dizajniran da bude jako jednostavan, čitljiv i razumljiv.
- **Performanse**: Kao kompajlirani jezik, Go se izvršava brže nego jezici kao što su Python i drugi interpretirani jezici.
- **Snažna konkurentnost**: Go ima ugrađenu podršku za konkurentnost kroz goroutine i kanale, omogućavajući jednostavno pisanje paralelnog koda.

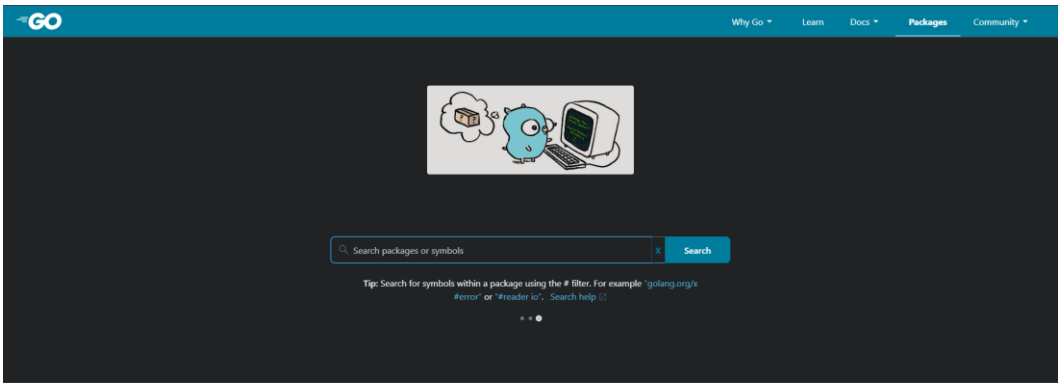
Nedostaci Go-a:

- **Nedostatak glomaznih funkcija**: Go je osmišljen kao jednostavan jezik i namjerno se drži tih principa, što znači da nema napredne funkcionalnosti koje su prisutne u nekim drugim jezicima, poput Jave.
- **Ograničene generičnost**i: Iako novije verzije Go-a imaju podršku za generičke tipove, oni su manje fleksibilni u usporedbi s drugim jezicima.

3.1 GO Paketi

Paketi u GO jeziku služe za organizaciju koda u logičke jedinice koje se mogu ponovo iskoristiti. Svaki Go program se sastoji od paketa, od kojih je **main** paket uvijek početna točka [8].

Go paketi ne služe samo kao biblioteke, već i kao praktični primjeri kako koristiti jezik. Mnogi paketi sadrže funkcionalne, samostalne primjere koji se mogu izravno pokrenuti s web stranice go.dev. Ako programer naiđe na problem ili ima pitanja o implementaciji, dokumentacija, kod i primjeri unutar tih paketa mogu pružiti odgovore, ideje i dublje razumijevanje. To omogućava programerima da brzo pronađu rješenja, nauče najbolje prakse i primjene konkretne koncepte u svojim projektima.



Slika 4. Prikaz go.dev-a

Proces stvaranja paketa uključuje sljedeće korake:

- Kreiranje direktorija za svoj paket.
- Dodavanje .go datoteke unutar direktorija.
- Prva izjava u svakoj datoteci treba deklarirati ime paketa. Ime paketa obično odgovara imenu direktorija.

```
package api
```

Izvoz Paketa (Export package)

U Go jeziku elementi (funkcije,tipovi i varijable) koje počinju velikim slovom su eksportirani i mogu se koristiti izvan paketa.

```
package gapi
  func NewStore(conn *sql.DB) *Store {
    return &Store{
        conn: conn,
    }
}
```

Uvoz paketa (Import Package)

Kako bi se koristio paket, potrebno ga je uvesti koristeći ključnu riječ import. Putanja (path) uvoza obično uključuje put do paketa unutar radnog prostora ili repozitorija.

```
  package api

import (
    "github.com/dbracic21-foi/simplebank/db"
)

// Koristi eksportovani `NewStore` iz paketa `db`
store := db.NewStore(dbConn)
```

Inicijalizacija paketa

Init funkcije se koriste za postavljanje paketa prilikom importanja (uvoza). One se automatski pozivaju prije main funkcije.

3.2 Formatiranje Koda

Formatiranje koda je često sporno, ali najmanje značajno pitanje. Svaki programer ima svoj stil formatiranja, pa se Go pobrinuo da umjesto ljudi to obavlja stroj. Program go-fmt formatira izvorni kod u standardiziranom stilu uvlake i vertikalnog poravnanja [10].

```
gofmt code.go
```

3.3 Kontrolne strukture

Kontrolne strukture u Go-u su slične onima u C-u, ali se razlikuju u ključnim aspektima. Switch-case je fleksibilnija opcija, ne postoje do i while petlje, a for petlja je generalizirana. Ključne riječi break i continue mogu koristiti opcionalne naljepnice za identificiranje točke na kojoj se operacija prekida ili nastavlja [9]. Postoje i nove kontrolne strukture koje uključuju select za

višekanalnu komunikaciju. Sintaksa se također razlikuje od mnogih drugih jezika - nema zagrada oko uvjeta (parenthesis), a tijelo mora biti unutar vitičastih zagrada (curly braces).

3.3.1 If

Jednostavni if u GO-u izgleda, kao i u C :

```
if x > 0 {
      return y
}
```

Obavezne zagrade potiču pisanje jednostavnih if naredbi na višestrukim linijama. To je dobra praksa pisanja posebno kada tijelo if statementa sadrži return ili break. Kako if prihvaća inicijalizaciju, često se može vidjeti postavljanje lokalne varijable.

```
If err := file.Chmod(0664); err != nil {
      Log.Print(err)
      return err
}
```

3.3.2 For

For loop u Go-u je sličan, ali ne isti kao onaj u C-u. For loop ujedinjuje for i while loop, te ne postoji do-while loop. Postoje tri različite forme:

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

3.3.3 Switch

Switch u GO-u je općenitiji nego u C-u. Izrazi ne trebaju biti cijeli brojevi ili konstante. Slučajevi su evaluirani od vrha prema dnu dok se ne pronađe traženi element.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

Iako nisu toliko općeniti kao u C-u, break naredbe se mogu koristiti da prekinu switch ranije. Ponekad, ako je potrebno izaći iz okolne petlje, u GO-u se to može ostvariti stavljajući oznaku (label) u petlju i prekidajući do te oznake. Sljedeći primjer nam pokazuje obje upotrebe :

```
Loop:
    for n := 0; n < len(src); n += size {
        switch {
        case src[n] < sizeOne:
            if validateOnly {
                break
            }
            size = 1
            update(src[n])

        case src[n] < sizeTwo:
            if n+1 >= len(src) {
                err = errShortInput
                break Loop
            }
            if validateOnly {
                break
            }
            size = 2
            update(src[n] + src[n+1]<<shift)
        }
    }
```

3.4 Funkcije

Funkcije su ključni segment programskog jezika Go, omogućujući organizaciju koda u odvojene blokove koji se mogu koristiti na više mjesta [8]. Definiraju se pomoću ključne riječi func i sastoje se od naziva funkcije, parametara (ako su potrebni) i tipa povratne vrijednosti (ako postoji). Osnovna funkcija u Go-u je main, koja se izvršava pri pokretanju programa. Kada funkcija prima parametre, oni moraju imati definiran tip. Primjer funkcije koja ispisuje sadržaj u konzolu na osnovu ulaznog parametra je:

```
func print(poruka string){
      fmt.println(poruka)

}
```

Ako funkcija treba vratiti neku vrijednost, koristi se ključna riječ return unutar funkcije, zajedno s varijablom ili vrijednošću koju funkcija vraća. Prilikom definiranja funkcije potrebno je odrediti tip povratne vrijednosti (ako postoji). Primjer funkcije koja vraća umnožak dvaju ulaznih parametara je:

```
func pomnozi_brojeve(a int, b int){
      return a * b

}
```

3.4.1 Višestruke povratne vrijednosti

Jedna od neobičnih značajki jezika Go je način na koji funkcije i metode rade s povratnim vrijednostima. Ova značajka omogućuje i poboljšava neke nezgrapne idiome u C programima, poput vraćanja pogrešaka putem -1 za EOF i izmjene argumenata proslijeđenih adresom [7]. U C-u se pogreška pri pisanju signalizira negativnim brojem, dok se kod pogreške pohranjuje na rizično mjesto. U Go-u, funkcija Write može vratiti broj upisanih bajtova i pogrešku, čime se olakšava rukovanje pogreškama i povećava sigurnost koda.

```
func (file *File) Write(b [] byte) (n int, err error)
```

3.5 Inicijalizacija, Konstante i Varijable

Inicijalizacija

Inicijalizacija u C-u i u Go-u na prvi pogled izgleda isto, ali inicijalizacija u Go-u je puno moćnija. Kompleksni objekti mogu se kreirati tijekom inicijalizacije, a redoslijed inicijalizacije objekata, čak i među različitim paketima, se ispravno rješava [9].

```
config, err := util.LoadConfig(".")
if err != nil {
    log.Fatal().Err(err).Msg("cannot load config")
}

connPool, err := pgxpool.New(ctx, config.DBSource)
if err != nil {
    log.Fatal().Err(err).Msg("cannot connect to db")
```

Ovdje možemo vidjeti da se koristi **inicijalizacija** za učitavanje konfiguracijskih varijabli iz datoteke, a također se inicijalizira veza s bazom podataka koristeći te varijable. Redoslijed ove inicijalizacije osigurava da su sve potrebne komponente dostupne prije nego što aplikacija krene s radom.

Konstante

Konstante u Go-u su uistinu konstante. Stvaraju se tijekom kompajliranja, čak iako su definirane lokalno unutar funkcija, i mogu biti samo brojevi, znakovi, stringovi ili booleani. Zbog ograničenja tijekom kompajliranja, izrazi koji ih definiraju moraju biti konstantni izrazi koje kompajler može evaluirati.

```
const DBDriver = "postgres"
const ServerPort = 8080
```

Ove konstante definiraju osnovne vrijednosti koje se neće mijenjati tijekom izvršavanja programa i olakšavaju održavanje jer su na jednom mjestu i jasno definiran

Varijable

Varijable mogu biti inicijalizirane kao i konstante ali inicijalizator može biti opći izraz koji se izračunava tijekom izvođenja programa [9]. U Go-u, varijable se definiraju nešto drugačije nego u C-u i C++-u. Varijabla se sastoji od tri glavna djela :

- Ključne riječi **var**
- Naziv varijable
- Tip varijable

```
var x string
```

Go podržava različite tipove varijabli ključne za izradu aplikacija, uključujući string, int, bool (boolean), nekoliko tipova cijelih brojeva (integer i unsigned integer), byte (koji predstavlja uint8 tip), rune (koji predstavlja int32 tip), float32, float64 (decimalni brojevi s pomičnim zarezom) i complex64, complex128 (kompleksni brojevi) [9]. Pored jednostavnih tipova, Go podržava i složene tipove podataka kao što su nizovi (arrays), dijelovi (slices) i mape (maps).

3.6 Gorutine i kanali

Gorutine su lagane funkcije koje omogućavaju istovremeno izvođenje više operacija unutar istog adresnog prostora [7]. Kreiranje gorutine postiže se prefiksom ključne riječi **„go“.** Gorutine rade u pozadini i završavaju kada i funkcija.

Kreiranje Gorutine

Kada se funkcija pokrene kao gorutine, Go runtime automatski upravlja njenim izvršenjem.

```
go funcName()
```

Kanali

Kanali omogućuju gorutinama međusobnu komunikaciju i sinkronizaciju. Deklariraju se pomoću **„make“** funkcije. Postoje dvije vrste:

1. Asinkroni
2. Sinkroni

Nebufferani kanali kombiniraju kombinaciju i sinkronizaciju, osiguravajući da dvije gorutine budu u poznatom stanju. Bufferani kanali mogu se koristiti kao semafori za ograničavanje propusnosti.

Paralelizacija

Paralelizacija omogućava raspodjelu skupa operacija na više CPU jezgri. Dijelovi operacije izvode se neovisno i signaliziraju završetak putem kanala.

3.7 Važne biblioteke u Go jeziku za razvoj mikroservisa

Kako bi se u potpunosti iskoristila snaga i fleksibilnost programskog jezika Go u razvoju mikroservisa, ključno je razumjeti i koristiti bogat skup biblioteka koje su dostupne unutar Go ekosustava. Ove biblioteke omogućuju jednostavniju i učinkovitiju izgradnju distribuiranih sustava, osiguravajući visoku razinu performansi, skalabilnosti i pouzdanosti.

Jedna od najvažnijih biblioteka u Go-u je **net/http**, koja omogućuje izravnu podršku za izgradnju web servera i rukovanje HTTP zahtjevima. Ova biblioteka, sa svojom jednostavnošću i fleksibilnošću, često je polazna točka u izgradnji mikroservisa temeljenih na RESTful arhitekturi. Osim toga, **gRPC** je još jedna ključna biblioteka koja omogućuje brzu i učinkovitu komunikaciju između mikroservisa koristeći binarni protokol, čime se postiže veća učinkovitost i manji prijenos podataka u odnosu na klasične tekstualne protokole.

Za složenije scenarije razvoja mikroservisa, biblioteke kao što su **go-kit** i **go-micro** pružaju alate za standardizaciju procesa, olakšavajući izgradnju modularnih, skalabilnih i lako održivih rješenja. Ove biblioteke dolaze s bogatim skupom funkcionalnosti kao što su upravljanje otkrivanjem servisa, balansiranje opterećenja, slanje poruka i još mnogo toga, čineći ih neophodnim za razvoj složenih distribuiranih sustava.

Rad s bazama podataka također je važan aspekt u razvoju mikroservisa, a **GORM** se ističe kao jedna od najpopularnijih ORM (Object-Relational Mapping) biblioteka u Go-u. GORM omogućuje lakše upravljanje podacima u relacijskim bazama putem jednostavne i intuitivne API podrške, što smanjuje potrebu za pisanjem složenih SQL upita.

Kombinacija ovih biblioteka s ugrađenim alatima jezika Go za konkurentno i paralelno izvršavanje čini ovaj jezik moćnim alatom za razvoj mikroservisa. Go biblioteke ne samo da ubrzavaju proces razvoja, već i omogućuju stvaranje sustava koji su visoko optimizirani za suvremene aplikacije, pogotovo one koje zahtijevaju brzu obradu velikih količina podataka i kontinuiranu dostupnost.

4. Izrada bankovnog servisa koristeći jezik Go

U ovom poglavlju detaljno su opisani različiti dijelovi aplikacije razvijene s pomoću programskog jezika Go. Dan je opsežan pregled strukture samog Go projekta, objašnjavajući kako su različiti dijelovi koda organizirani i međusobno povezani.
Funkcionalni zahtjevi aplikacije :

4.1 Opis domene

Aplikacija razvijena u ovom dijelu je backend web aplikacija izrađena u programskom jeziku Go, s ciljem izgradnje jednostavnog bankovnog servisa. Glavni cilj ove aplikacije je pružanje jednostavnog bankovnog servisa koji omogućava korisnicima osnovne bankovne transakcije poput otvaranje računa i prijenos sredstava.

Tablica 2. Funkcionalni zahtjevi aplikacije

Zahtjev	Opis
Registracija i prijava korisnika	Implementacija korisničke registracije i prijave s autentifikacijom koristeći JWT token.
Upravljanje računima	Kreiranje, ažuriranje i brisanje bankovnih računa korisnika.
Izvršavanje transakcija	Omogućavanje prijenosa sredstava između računa.
Autorizacija i autentifikacija	Sigurno upravljanje sesijama i pristupom korisnika putem API-ja.

Nefunkcionalni zahtjevi :

Tablica 3. Nefunkcionalni zahtjevi aplikacije

Zahtjev	Opis

Performanse	Podrška za skalabilnost, omogućujući aplikaciji da podrži velik broj korisnika
Sigurnost	Zaštita korisničkih podataka (hashiranje lozinki, enkripcija osjetljivih informacija)
Održavanje i ažuriranje	Podrška za jednostavne nadogradnje i održavanje sustava bez zastoja
Prijenosivost	Implementacija Docker kontejnera kako bi se aplikacija mogla lako prenositi između različitih okruženja (npr. lokalno, staging, produkcija).

4.2 Arhitektura rješenja

Detaljni pregled aplikacije se sastoji od :

1. **Dizajn baze podataka** - Aplikacija koristi PostgresSQL za upravljanje bazom podataka, također je korišten i golang-migrate za upravljanje migracijama baze podataka.
2. **Generiranje SQL koda** – Alat „Sqlc“ se koristi za generiranje sql koda direktno iz sql upita.
3. **RESTfull Api** – Za implementaciju api-ja za različite bankovne usluge koristi se gin okvir. Autentifikacija i autorizacija je postignuta preko JWT tokena.
4. **Upravljanje transakcijama** - Implementacija transakcija osigurava konzistentnost podataka, dok pravilno korištenje različitih izolacijskih nivoa pomaže u sprječavanju problema poput potpunog zastoja (deadlock).
5. **Implementacija gRPC-a** – Aplikacija također implementira Grpc API-je koristeći protobuf za definiranje servisa.
6. **Asinkrono procesiranje** – Za asinkrono procesiranje koristi se Redis i Asynq za upravljanje zadacima u pozadini.

7. **Deployment** – Za deployment je zadužen Docker sa svojim kontenjerima. Na jednoj Docker slici se pokreće baza podataka, sa cijelim backendom.

Aplikacija je organizirana na jasan i strukturiran način, kako bi se olakšao razvoj, održavanje i proširenje funkcionalnosti. Evo detaljnog pregleda strukture direktorija i datoteka koje čine ovu aplikaciju:

1. Direktoriji

- **api**: Ovaj direktorij sadrži kod koji implementira HTTP API-je. Ovdje se definiraju rute, handleri i logika za obrađivanje HTTP zahtjeva.
- **db**: Ovdje se nalaze datoteke vezane uz bazu podataka, uključujući migracije koje definiraju strukturu tablica i inicijalne podatke.
- **mail**: Kod za slanje emailova. Ovaj direktorij sadrži funkcije i konfiguracije za slanje različitih vrsta emailova korisnicima.
- **token**: Direktorij za generiranje i validaciju tokena za autentifikaciju i autorizaciju korisnika. Obuhvaća implementaciju JWT i PASETO tokena.
- **util**: Pomoćne funkcije i utilitarni kod koji se koristi na više mjesta unutar aplikacije.
- **val**: Direktorij za validaciju podataka. Ovdje se implementiraju funkcije za provjeru ispravnosti ulaznih podataka.
- **Worker**: Sadrži kod za asinkrono procesiranje zadataka, kao što su pozadinski poslovi koji se izvršavaju koristeći Redis i Asynq.

2. Datoteke u Korijenskom Direktoriju

- **.gitignore**: Datoteka koja specificira koje datoteke i direktorije Git treba ignorirati.
- **app.env**: Konfiguracijska datoteka koja sadrži varijable okruženja potrebne za pokretanje aplikacije.
- **docker-compose.yaml**: Datoteka koja definira Docker kontejnere potrebne za pokretanje aplikacije, uključujući bazu podataka, Redis i samu aplikaciju.
- **Dockerfile**: Skripta koja sadrži upute za izgradnju Docker slike aplikacije.
- **go.mod**: Datoteka koja definira module i zavisnosti Go projekta.
- **go.sum**: Datoteka koja sadrži kontrolne zbrojeve za zavisnosti navedene u go.mod datoteci, osiguravajući konzistentnost zavisnosti.
- **main.go**: Ulazna točka aplikacije. Ova datoteka sadrži glavni program koji pokreće aplikaciju.

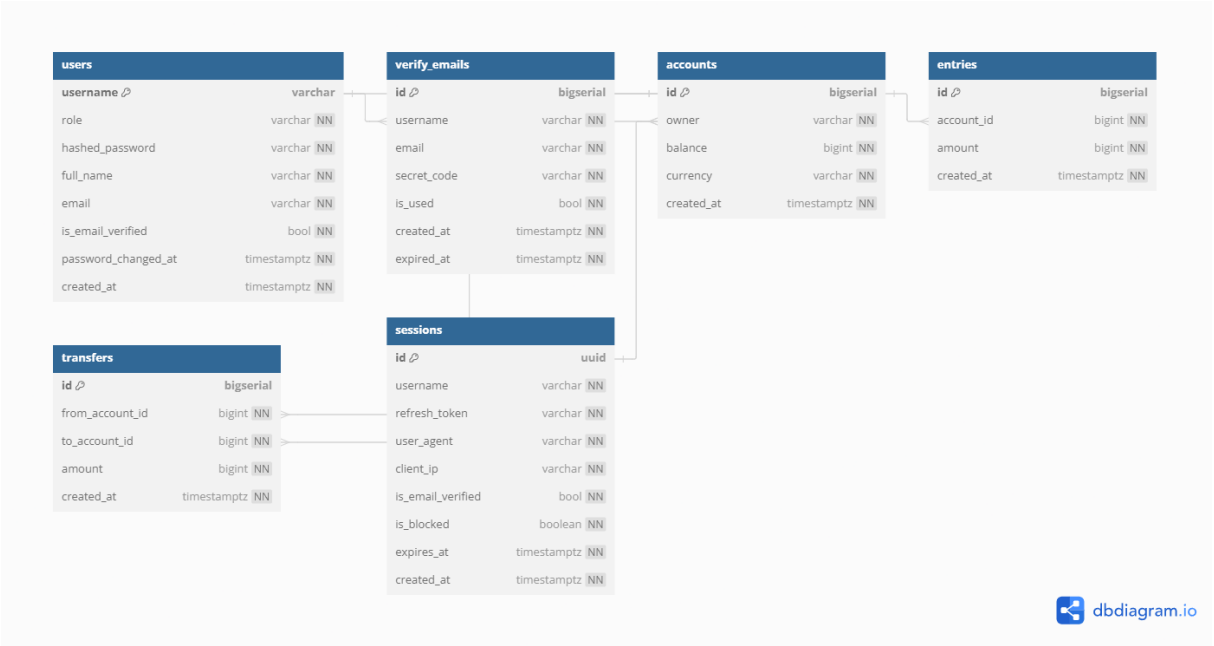
- **Makefile**: Skripta koja definira zadatke automatizacije, kao što su testiranje, izgradnja i pokretanje aplikacije.
- **sqlc.yaml**: Konfiguracijska datoteka za sqlc, alat koji generira Go kod iz SQL upita.
- **start.sh**: Skripta za pokretanje aplikacije.

4.3 Implementacija rješenja

Na samom početku kreiranja aplikacije, potrebno je osmisliti i dizajnirati model baze podataka, s potrebnim atributima i vezama. Za tako nešto koristit ćemo alat „dbdiagram.io“. Preko dizajna baze podataka ćemo generirati sql kod, koji će nam služiti za kreiranje upita.

Koraci u dizajniranju baze podataka :

Prvi korak je dizajn sheme baze podataka. Uz pomoć već navedenog alata dbdiagram.io kreiramo vizualni model baze podataka, alat nam omogućava definiranje tablica, atribute i odnose između tablica.



Slika 5 Dizajn sheme baze podataka

Shema baze podataka aplikacije sastoji se od šest glavnih tablica: users, verify_emails, accounts, entries, transfers, i sessions. Tablica users povezana je s tablicama verify_emails i sessions putem atributa username. Tablica accounts povezana je s tablicama entries i transfers preko atributa account_id. Veze između ovih tablica omogućavaju praćenje korisničkih računa, verifikaciju emailova, vođenje sesija, te evidentiranje transakcija i prijenosa sredstava između računa.

Sljedeći korak Generiranje SQL koda. Nakon što je shema baze podataka definirana, dbdiagram.io će generirati odgovarajući SQL kod. Ovaj kod koristimo za kreiranje tablica u PostgreSQL bazi.

Na kraju, Migracije Baze podataka su ključne za upravljanje promjenama u bazi podataka tijekom razvoja aplikacije. S pomoću alata golang-migrate kreiramo i upravljamo migracijama.

```
migrateup:
    migrate -path db/migration -database "${DB_URL}" -verbose up
migrateup1:
    migrate -path db/migration -database "${DB_URL}" -verbose up 1
migratedown:
    migrate -path db/migration -database "${DB_URL}" -verbose down
migratedown1:
    migrate -path db/migration -database "${DB_URL}" -verbose down 1

new_migration:
```

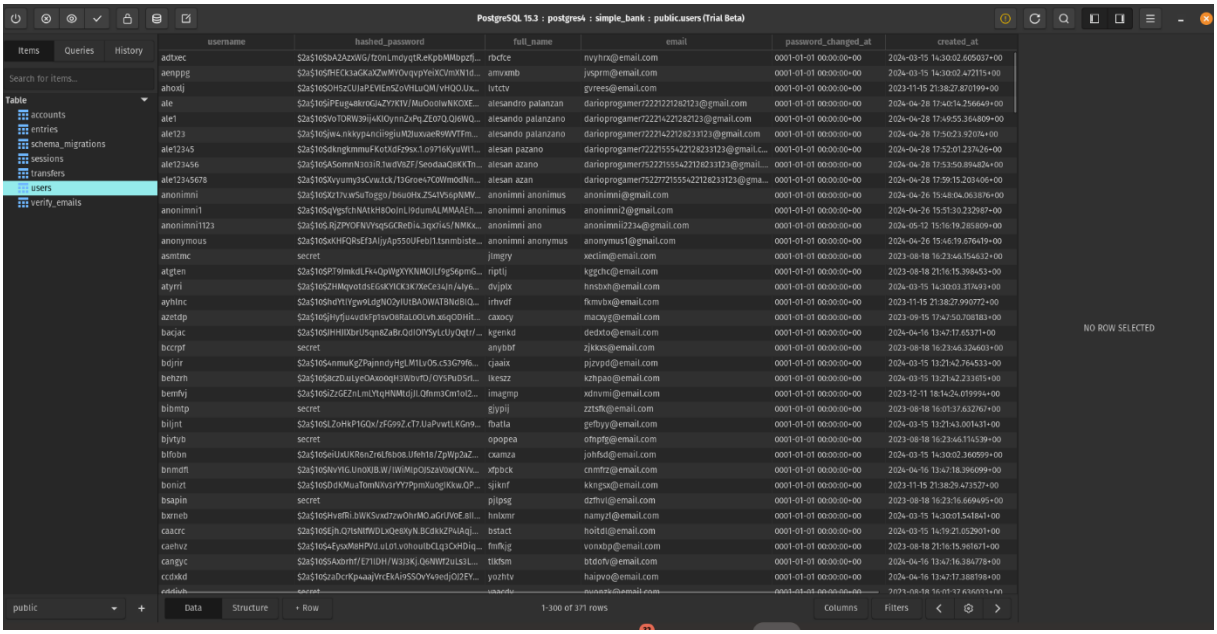
Slika 6. Komande migracija u Makefile-u

Za izvršenje komandi za migraciju baze podataka zadužen je makefile. Makefile je alat koji automatizira izvršavanje ponavljajućih zadataka u razvoju softvera. U ovoj aplikaciji, Makefile je konfiguracijska datoteka koja definira skup uputa koje se koriste za kompajliranje, testiranje i pokretanje aplikacije. Tako, na primjer, ako želimo pokrenuti migrateup, komanda bi bila :

```
make migrateup
```

4.3.1 Prikaz i Spajanje na bazu podataka

Prikaz podataka i tablica nalazi se u TablePlusu-u. TablePlus je moderni GUI alat za upravljanje bazama podataka, koji podržava različite sustave baza podataka uključujući MySQL, PostgreSQL, SQLite, Microsoft SQL Server, Redis, i mnoge druge. Baza podataka se pokreće s dockera, točno kako docker funkcionira u aplikaciji ćemo objasniti kasnije u radu. Baza je pokrenuta lokalno.



Slika 7. Prikaz podataka u table plus-u

Proces spajanja na bazu podataka sastoji se od nekoliko koraka, a glavna implementacija istog smještena je u datoteci main.go. Koraci za spajanje na bazu su:

1. Učitavanje konfiguracije – Prvo se učitavaju konfiguracije iz datoteke app.env. U app.env se nalaze potrebni podaci za spajanje na bazu.

```
.ENVIROMENT=development

DB_SOURCE=postgresql://root:secret@localhost:5433/simple_bank?sslmode=disab
le
DB_DRIVER=postgres
MIGRATION_URL=file://db/migration
HTTP_SERVER_ADRESS =0.0.0.0:8080
GRPC_SERVER_ADRESS =0.0.0.0:9090
ACCESS_TOKEN_DURATION=15m
TOKEN_SYMETRIC_KEY=neki_tajni_kljucaefaefuin4718nk
REFRESH_TOKEN_DURATION = 24h
REDIS_ADDRESS=0.0.0.0:6379
EMAIL_SENDER_NAME = Simple Bank
EMAIL_SENDER_ADDRESS = simplebank554@gmail.com
EMAIL_SENDER_PASSWORD = neka_tajna_sifra89q347feiun
```

Povezivanje s bazom podataka se ostvaruje korištenjem „pgxpool” paketa, koji omogućava rad s PostgreSQL bazama podataka. pgxpool je dio PostgreSQL Go drivera, poznatog kao pgx, koji omogućava upravljanje bazenom veza za PostgreSQL baze podataka. Primjer korištenja funkcije u pgxpool je:

```
pgxpool.New(ctx,config.DBSource)
```

Nakon što je uspostavljena veza, pokreću se migracije preko golang-migrate. Migracije osiguravaju da je struktura baze podataka ažurirana i odgovara potrebama aplikacije.

```
func main() {
	config, err := util.LoadConfig(".")
	if err != nil {
		log.Fatal().Err(err).Msg("Cannot load config:")

	}
	if config.Environment == "development" {
		log.Logger = log.Output(zerolog.ConsoleWriter{Out: os.Stderr})
	}
	ctx, stop := signal.NotifyContext(context.Background(),
interruptSignal...)
	defer stop()

	connpool, err := pgxpool.New(context.Background(), config.DBSource)
	if err != nil {

		log.Fatal().Msg("Cannot connect to db")
	}
```

Nakon uspješnog pokretanja migracija, kreira se „store" objekt, pomoću „db.NewStore(connpool)" funkcije.

4.3.2 Generiranje SQL upita

SQLC je alat za generiranje sigurnog koda iz sql upita. Dizajniran je da pojednostavi rad s bazama podataka, automatizirajući proces kreiranja koda koji komunicira sa bazom.

Proces generiranja SQL upita:

Pisanje SQL upita - prvo se piše sql upiti koje želimo koristiti u aplikaciji, na primjer :

```
-- name: CreateAccount :one
INSERT INTO accounts (
  owner,
  balance,
  currency
) VALUES (
  $1, $2, $3
) RETURNING *;

-- name: GetAccounts :one
SELECT * FROM accounts
WHERE id = $1 LIMIT 1;
```

Generiranje identičnog Go koda iz navedenog upita postiže se pokretanjem komande „sqlc generate". Alat uzima SQL upite iz specificiranih datoteka i generira odgovarajući SQL Go kod.

Taj cijeli proces rezultira kreiranjem .sql.go datoteka koje sadrže funkcije za izvršavanje SQL upita.

Generirani kod izgleda:

```
// Code generated by sqlc. DO NOT EDIT.
// versions:
//   sqlc v1.25.0
// source: account.sql

package db

import (
	"context"
)

const addAccountsBalance = `-- name: AddAccountsBalance :one
UPDATE accounts
  set balance = balance + $1
WHERE id = $2
RETURNING id, owner, balance, currency, created_at, country_code
`

type AddAccountsBalanceParams struct {
	Amount int64 `json:"amount"`
	ID     int64 `json:"id"`
}

func (q *Queries) AddAccountsBalance(ctx context.Context, arg AddAccountsBalanceParams) (Account, error) {
	row := q.db.QueryRow(ctx, addAccountsBalance, arg.Amount, arg.ID)
	var i Account
	err := row.Scan(
		&i.ID,
		&i.Owner,
		&i.Balance,
		&i.Currency,
		&i.CreatedAt,
		&i.CountryCode,
	)
	return i, err
}
```

Korištenje SQLC-a u razvoju aplikacija značajno pojednostavljuje i ubrzava rad s bazama podataka. Automatsko generiranje koda sa sigurnim tipovima smanjuje mogućnost grešaka i olakšava održavanje koda, čime se povećava pouzdanost i učinkovitost aplikacije. Za svaku tablicu generiraju se odgovarajuće .sql.go datoteke, što osigurava konzistentnost i preglednost koda.

4.3.3 RestFul Api

Aplikacija sadrži skup RESTful HTTP API-ja koji omogućuju različite bankovne operacije, uključujući otvaranje računa, izvršavanje transakcija i kreiranje korisnika. Ovi API-ji su ključni za omogućavanje korisnicima interakciju s bankovnim sustavom putem jednostavnih HTTP zahtjeve.

RESTful API-ji su implementirani korištenjem Gin okvira, popularnog web okvira za Go (Golang). Gin je poznat po svojoj brzini i jednostavnosti te omogućava strukturiranje i rukovanje HTTP zahtjevima na način koji je intuitivan i efikasan, osiguravajući da su sve operacije sigurne i pouzdane.

Prije samog objašnjena API-ja prvo je bitno objasniti alate koji se generiraju prije stvaranja API-ja. Glavna datoteka koja će nam pomoći za to je store.go, koja je glavno mjesto za sve CRUD operacije i transakcije. Također u store.go datoteci nalaze se metode za brisanje, ažuriranje, kreiranje i čitanje. Također, još jedna tehnika i alat koji su potrebni je mock, koji će pomoći u testiranju, ali i u stvaranju store.go datoteke.

```
// AddAccountsBalance indicates an expected call of AddAccountsBalance.
func (mr *MockStoreMockRecorder) AddAccountsBalance(arg0, arg1 interface{})
*gomock.Call {
	mr.mock.ctrl.T.Helper()
	return mr.mock.ctrl.RecordCallWithMethodType(mr.mock,
"AddAccountsBalance", reflect.TypeOf((*MockStore)(nil).AddAccountsBalance),
arg0, arg1)
}

// CreateAccount mocks base method.
func (m *MockStore) CreateAccount(arg0 context.Context, arg1
db.CreateAccountParams) (db.Account, error) {
	m.ctrl.T.Helper()
	ret := m.ctrl.Call(m, "CreateAccount", arg0, arg1)
	ret0, _ := ret[0].(db.Account)
	ret1, _ := ret[1].(error)
	return ret0, ret1
}
```

4.3.4 JWT

JWT – Json Web Token je standard za kreiranje sigurnih tokena koji se koriste za autentifikaciju i autorizaciju. U aplikaciji, JWT se koristi za generiranje i verifikaciju tokena za pristup API-ju.

JWT se sastoji od tri dijela :

1. Zaglavlje (Header)
2. Tijelo (Payload)
3. Potpis (Signature)

Za kreaciju JWT tokena u aplikaciji, prvo se mora definirati struktura „JWTMaker“ koja implementira „Create Token“ i „VerifyToken“ metode. Metoda CreateToken generira JWT koristeći jwt-go paket, potpisuje ga tajnim ključem i vraća token.

```
func (maker *JWTMaker) CreateToken(username string, role string, duration
time.Duration) (string, *Payload, error) {
	payload, err := NewPayload(username, role, duration)
	if err != nil {
		return "", payload, err
	}
	jwtToken := jwt.NewWithClaims(jwt.SigningMethodHS256, payload)
	token, err := jwtToken.SignedString([]byte(maker.secretKey))
	return token, payload, err

}
```

1. Kreacija JWT Tokena

Verifikacija JWT tokena parsira i verificira JWT token koristeći „jwt.ParseWithClaims“ funkciju. Funkcija provjerava potpis i validnost tokena. Ako token nije validan ili je istekao, smatra se da korisnik koji šalje upit više nije autoriziran za daljnje korištenje aplikacije.

```
func (maker *JWTMaker) VerifyToken(token string) (*Payload, error) {
	keyFunc := func(token *jwt.Token) (interface{}, error) {
		_, ok := token.Method.(*jwt.SigningMethodHMAC)
		if !ok {
			return nil, ErrInvalidToken
		}
		return []byte(maker.secretKey), nil

	}
```

Nakon objašnjena elemenata i dijelova aplikacije koje su nam bili potrebni za kreiranje api-ja. Objasnit ćemo kako kreiranje korisnika, kreiranja računa i provedba transakcije djeluju u aplikaciji.

4.3.5 Kreiranje Korisnika

Kreiranje korisnika – Api za kreiranje korisnika omogućava kreiranje/registraciju novih korisnika u sustavu. Registracija uključuje validaciju korisničkih podataka, računanje sažetka lozinki, slanje mailova za verifikaciju te pohranjivanje informacija o korisnicima u bazu podataka.
Prvi korak u procesu registracije korisnika je validacija ulaznih podataka. Koristeći Gin okvir, validacija se vrši pomoću oznaka (tagova).

```
type createUserRequest struct {
```

```
	Username string `json:"username" binding:"required,alphanum"`
	Password string `json:"password" binding:"required,min=6"`
	FullName string `json:"full_name" binding:"required"`
	Email    string `json:"email" binding:"required"`
}
```

Kako bi se osigurala sigurnost korisničkih lozinki, one se kriptiraju funkcijom sažetka prije pohrane u bazu podataka. To se postiže korištenjem paketa „bycript“.

```
func HashPassword(password string) (string, error) {

	hashPassword, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.DefaultCost)

	if err != nil {
		return "", fmt.Errorf("failed to hash password : %w", err)
	}
	return string(hashPassword), nil
}
```

2. Hashiranje lozinke

Funkcija HashPassword koristi bycript algoritam za generiranje sažetka lozinke, što osigurava da lozinka nikad ne bude pohranjena u običnom tekstu, već u kriptirnom obliku.

Nakon uspješne registracije, korisniku se šalje e-poruka za verifikaciju, kako bi potvrdio svoju adresu e-pošte. Za slanje e-poruke potrebno nam je asinkroni način rada, što postižemo koristeći radnike (Workers) i Redis. Asinkroni radnici omogućavaju izvršavanje zadataka u pozadini bez blokiranja glavnog toka aplikacije, a Redis je brza baza podataka otvorenog koda koja se koristi kao priručna memorija.

Nakon validiranja korisničkih podataka i nakon što je lozinka sažeta, informacije o korisniku se pohranjuju u bazu podataka. To se postiže preko funkcije za kreiranje korisnika koja koristi SQL upite za umetanje podataka u odgovarajuću tablicu.

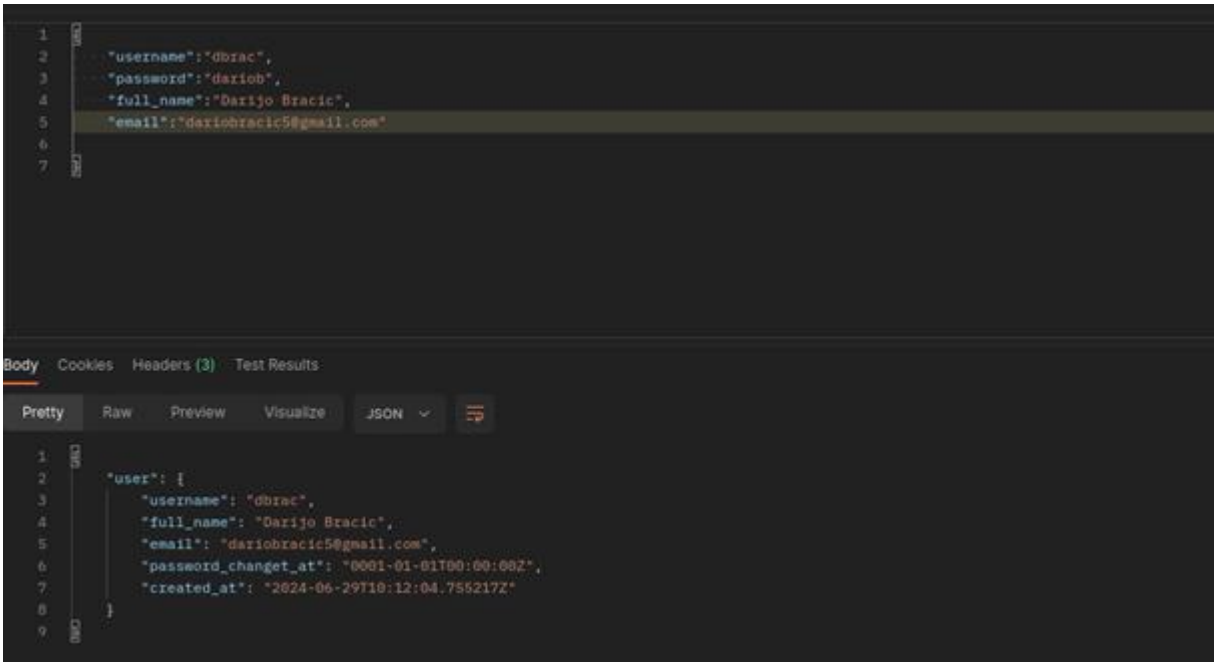
```
arg := db.CreateUserParams{
		Username:       req.Username,
		HashedPassword: hashedPassword,
		FullName:       req.FullName,
		Email:          req.Email,
	}

	user, err := server.store.CreateUser(ctx, arg)
```

Nakon uspješnog kreiranja korisnika, kao zadnji korak, generira se odgovor koji se šalje klijentu. Odgovor uključuje osnovne informacije o novom korisniku.

```
	rsp := newUserResponse(user)
	ctx.JSON(http.StatusOK, rsp)
```

Za provjeru rada api-ja koristit ćemo alat Postman. Postman je alat za testiranje API-ja koji koji omogućava da se lako šalju HTTP zahtjevi i pregledavaju odgovori. Zbog toga Postman će nam služiti za provjeru RestFul API-ja, te ćemo preko Postmana provjeravati svaki API.



Slika 8. Prikaz odgovora u postmanu

4.3.6 Kreiranje Računa

Kreiranje računa omogućava korisnicima kreiranje novih bankovnih računa. Proces je sličan prethodnom, prvi korak je sličan kao i u prethodnom API-ju, a to je validacija ulaznih podataka. Ulazni podaci uključuju valutu u kojoj se želi otvoriti račun. Koristimo Gin okvir za validaciju podataka.

```
type createAccountRequest struct {
	Currency string `json:"currency" binding:"required,oneof=USD EUR CAD HRK"`
}
```

Nakon što su ulazni podaci validirani, sljedeći korak je kreiranje novog računa u bazi podataka. Ovaj korak uključuje generiranje SQL upita za umetanje podataka u tablicu „accounts“ i izvršavanje tog upita.

```
	arg := db.CreateAccountParams{
		Owner:    authPayload.Username,
		Currency: req.Currency,
		Balance:  0,
	}
```

```
account, err := server.store.CreateAccount(ctx, arg)
```

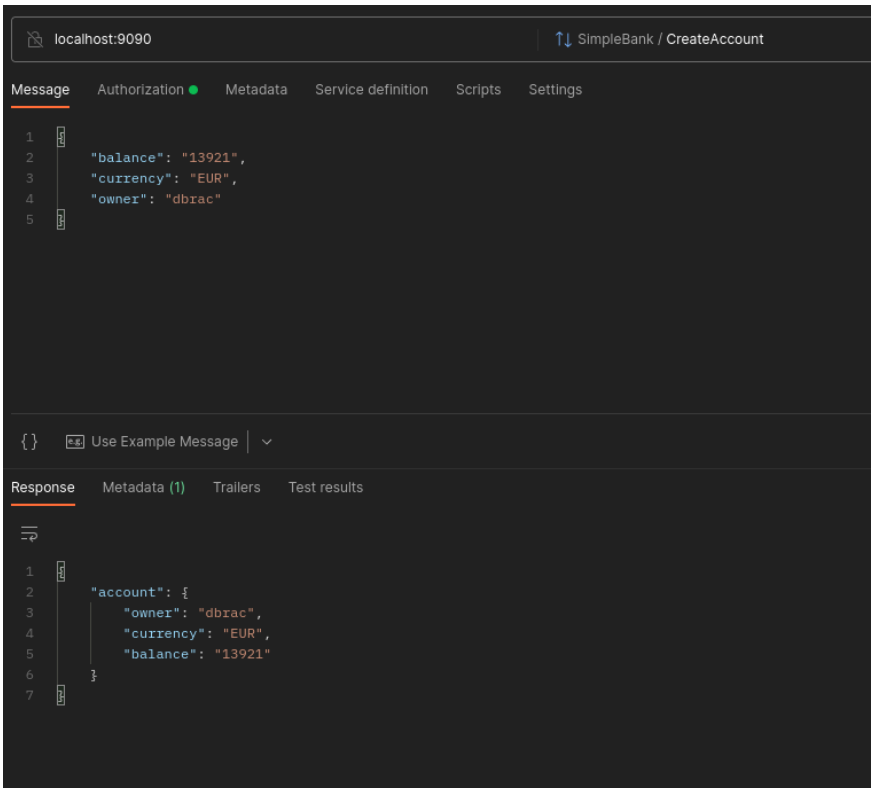
Nakon uspješnog kreiranja računa, vraća se odgovor korisniku koji uključuje osnovne informacije o novom računu.

```
rsp := newAccountReponse(account)
log.Info().Msg("account created successfully")

ctx.JSON(http.StatusOK, account)
```

3. Generiranje odgovora

Preko Postmana će se opet testirati valjanost API-ja



Slika 9. Prikaz odgovora u postmanu

4.3.7 Kreiranje transakcija u aplikaciji

API za transakcije u aplikaciji omogućuje prijenos sredstava između dva računa korisnika. Proces uključuje validaciju ulaznih podataka to jest broj računa korisnika koji šalje novce i broj računa korisnika koji ih prima. Ovaj API Provjerava stanje na računu, izvršava transakcije i evidentira transakcije u bazi podataka.

Prvi korak je validacija ulaznih podataka preko Gin okvira. Ulazni podaci za ovaj API su ID računa koji šalje i ID računa koji prima novce, iznos transakcije te valuta.

```
type transferRequest struct {
      FromAccountID int64  `json:"from_account_id" binding:"required,min=1"`
      ToAccountID   int64  `json:"to_account_id" binding:"required,min=1"`
      Amount        int64  `json:"amount" binding:"required,gt=0"`
      Currency          string  `json:"currency"  binding:"required,currency"
      `
}
```

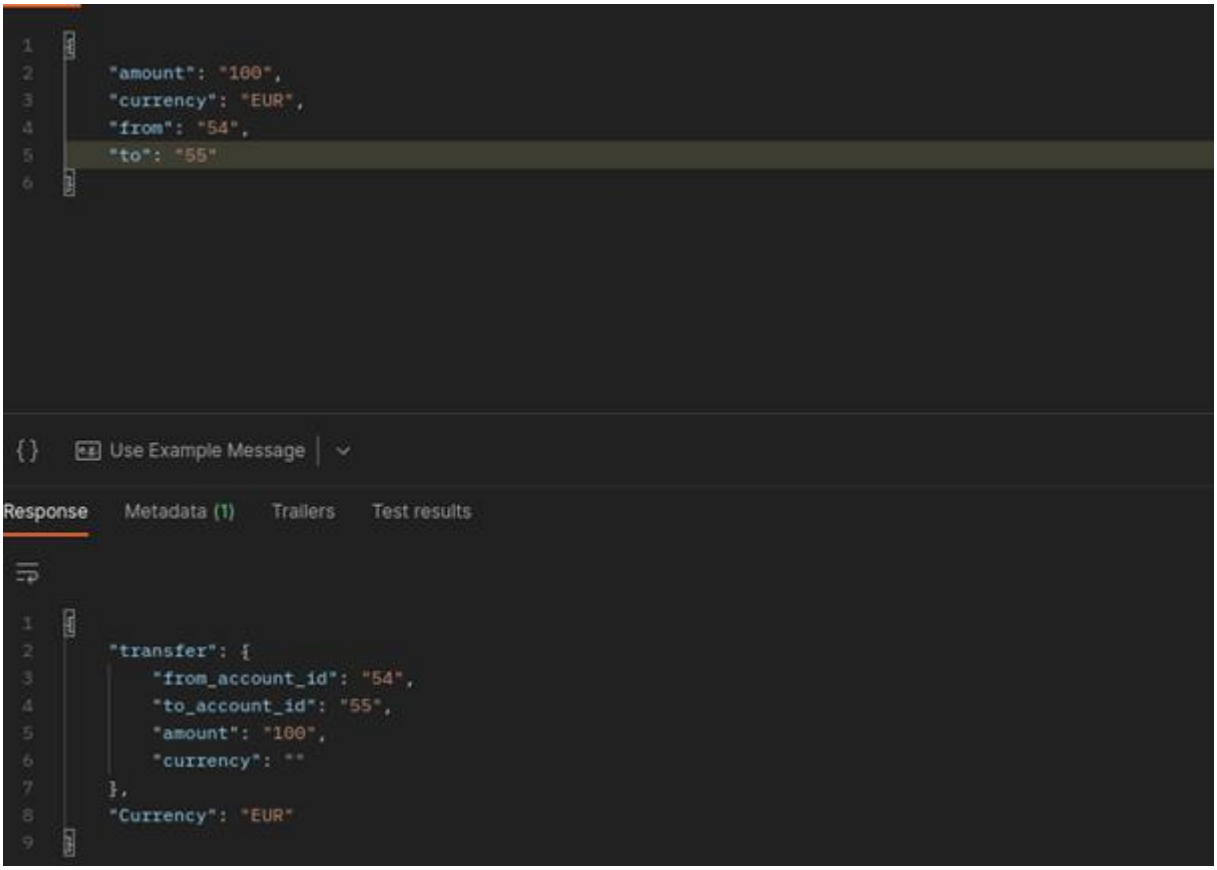
36.Validacija provedbe transakcije

Nakon što je validacija uspješno prošla, sljedeći korak je provjera stanja na računu i izvršenje transakcije. To se izvršava korištenjem funkcija za interakciju s bazom podataka koje provjeravaju stanje na računu i izvršavaju transakciju.

```
arg := db.TransferTxParams{
            FromAccountID: req.FromAccountID,
            ToAccountID:   req.ToAccountID,
            Amount:        req.Amount,
      }

      result, err := server.store.TransfersTx(ctx, arg)
```

Funkcija TransferTx izvršava transakciju i evidentira ju u bazi podataka. Ova funkcija koristi transakcije na razini baze podataka kako bi osigurala nedjeljivost (atomarnost) operacije, to jest ili se sve operacije izvrše ili niti jedna. U postmanu nakon testiranja dobivamo rezultat :



Slika 10. Izgled odgovora u Postmanu

4.3.8 Asinkrono programiranje

U aplikaciji asinkrono programiranje je implementirano korištenjem Redis-a i asynq biblioteke za upravljanje radnicima. Ovaj pristup omogućava izvršavanje dugotrajnih zadataka u pozadini, čime se povećava brzina aplikacije. Evo detaljnog pregleda svih koraka i komponenti uključenih u ovaj proces.

1. Redis

Redis je brza baza podataka koja se koristi za razne svrhe, uključujući caching. U ovom projektu, Redis se koristi za redove zadataka koje radnici obrađuju asinkrono.

- **Konfiguracija Redis Klijenta**:
  - Redis klijent je konfiguriran s pomoću asynq.RedisClientOpt. Ova konfiguracija se koristi za povezivanje s Redis serverom koji će držati redove zadataka.
  - Zadaci koje radnici obrađuju definiraju se pomoću struktura i funkcija koje kreiraju nove zadatke i serijaliziraju podatke potrebne za obradu.

- **Dodavanje Zadatka u Red**:
  - Kada aplikacija treba da izvrši dugotrajan zadatak, zadatak se dodaje u Redis red
  - Prilikom kreiranja novog korisnika, zadatak za slanje emaila dodaje se u red koristeći asynq klijent.

- **Konfiguracija Radnika**:
  - Konfiguracija radnika uključuje određivanje broja radnika, registraciju handlera za različite tipove zadataka i pokretanje servera radnika.
  - Radnici se konfiguriraju pomoću asynq.NewServer i asynq.NewServeMux. NewServer postavlja server za radnike, dok NewServeMux služi za registraciju handlera za različite tipove zadataka.

4.3.9 Slanje E-pošte

Već smo ranije spomenuli da, kada se korisnik registrira, dobiva e-poštu za verifikaciju. To je napravljeno asinkronim načinom. U protivnom bi slanje e-pošte zauzelo glavnu dretvu i korisnik se ne bi uspio registrirati. Za implementaciju ove funkcionalnosti prvo je bilo potrebno implementirati radnike i Redis.
Također, za realizaciju slanja emaila, potreban nam je SMTP server. Postavljanje SMTP servera uključuje definiranje postavki kao što su adresa servera, port, korisničko ime i lozinka. Funkcija za slanje Emaila koristi „gomail“ paket za sastavljanje i slanje email-a. Poruka se sastoji od zaglavlja, predmeta i tijela poruke.

Verifikacija E-pošte :

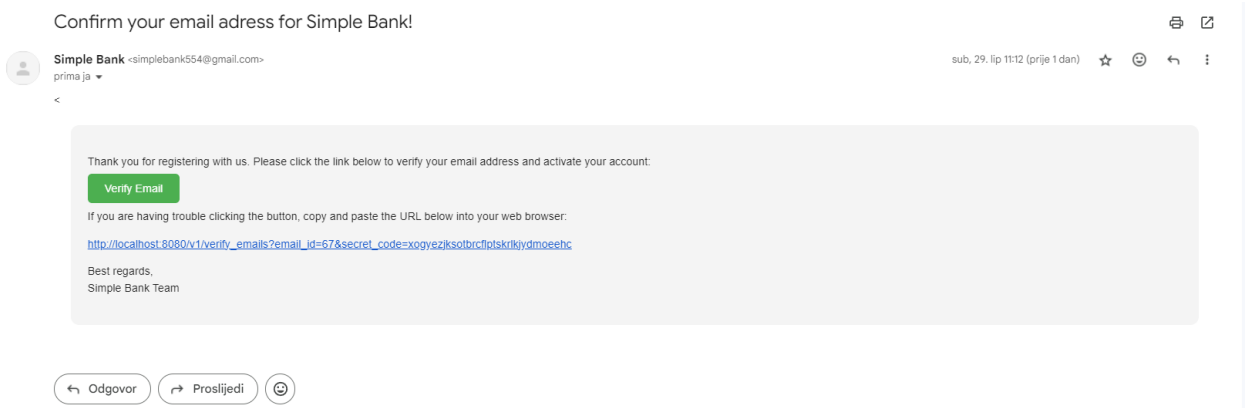
Za slanje verifikacijskih Emailova koristi se zadatak koji se definira unutar modula za radnike. Zadatak sadrži podatke potrebne za kreiranje i slanje verifikacijskog email-a.

```
verifyUrl                                                              :=
fmt.Sprintf("http://localhost:8080/v1/verify_emails?email_id=%d&secret_code
=%s", verifyEmail.ID, verifyEmail.SecretCode)
      subject := "Confirm your email adress for Simple Bank!"
      content := fmt.Sprintf(`<<html>
      <head>
        <style>
            body { font-family: Arial, sans-serif; line-height: 1.6; }
            .content { margin: 20px; padding: 20px; border-radius: 10px;
background-color: #f4f4f4; }
            .button { background-color: #4CAF50; color: white; padding: 10px
20px; text-decoration: none; border-radius: 5px; }
```

```
        </style>
      </head>
      <body>
        <div class="content">
            <p>Thank you for registering with us. Please click the link below
to verify your email address and activate your account:</p>
            <a href="%s" class="button">Verify Email</a>
            <p>If you are having trouble clicking the button, copy and paste
the URL below into your web browser:</p>
            <p>%s</p>
            <p>Best regards,<br>Simple Bank Team</p>
        </div>
      </body>
      </html>
      `, user.FullName, verifyUrl)
      to := []string{user.Email}
```

Proces slanja Email-a :

1. **Kreiranja zadataka za verifikaciju** – Kada novi korisnik kreira račun, aplikacija generira zadatak za slanje verifikacijske e-pošte. Zadatak se dodaje u Redis koristeći asynq biblioteku koja upravlja redovima zadataka.
2. **Izvršavanje zadataka** – Radnici preuzimaju zadatke iz Redis reda i izvršavaju ih. Kada radnik preuzme zadatak prvo dohvaća podatke o korisniku, zatim kreira sadržaj emaila.
3. **Slanje Email-a** – Nakon što je sadržaj email-a kreiran koriste se SMTP konfiguracija za slanje email-a. Email uključuje adresu pošiljatelja, predmet i tijelo poruke.



Slika 11. Izgled poruke za verifikaciju

4.4 Testiranje Api-ja pomoću GoMock-a

Potrebno je da svaki API ima svoju testnu datoteku kako bismo pokrili što više slučajeva i osigurali pravilno djelovanje našeg programa. Preko Postmana smo saznali kako se ponaša ako korisnik unese validne podatke. Sada trebamo testirati za svaki API slučajeve kada korisnik unese krive podatke. Naravno to možemo ispitati i ručno, preko Postmana, ali postoje i automatizirani načini koji osiguravaju veću pokrivenost. Zbog toga ćemo kreirati testne datoteke za svaki API. Za to će nam pomoći već prethodno spomenuti alat GoMock. Mock smo koristili kada smo spominjali store.go datoteku. Prvo što nam je bilo potrebno je sučelje koje je Mock automatski generirao i koje se nalazi na početku store.go datoteke :

```
type MockStore struct {
	ctrl     *gomock.Controller
	recorder *MockStoreMockRecorder
}
```

Nakon toga moramo stvoriti testnu funkciju. U Go-u, za stvaranje testne funkcije, potreban nam je paket „testing“, te svaka funkcija izgleda ovako :

```
func TestFunction(t *testing.T) {
	// Test body
}
```

42. Primjer testne funkcije

Sada je potrebno implementirati sučelje koje je mock generirao za nas. Tako će unutar svake testne funkcije nalaziti anonimna struktura :

```
func testGetAccountAPI(t *testing.T) {

	testcases := []struct {
		name		string
		setupAuth		func(t *testing.T, request *http.Request,
tokenMaker token.Maker)
		buildstubs	func(mockStore *mockdb.MockStore)
		checkResponses		func(t *testing.T, recorder
*httptest.ResponseRecorder)
	}
```

Anonimna struktura se sastoji od imena testnog slučaja. Tako npr, ako želimo da test uspije nazvat ćemo ga „Ok“ ili slično. SetupAuth nam služi za provjeru autentifikacije za zahtjev. Funkcija buildstub postavlja očekivanja za mock objekt, te je zadnja funkcija checkResponses koja provjerava odgovore na HTTP zahtjeve.

Izvršavanje testova :

Svaki testni slučaj se izvršava unutar petlje koja prolazi kroz našu anonimnu strukturu. Unutar petlje funkcija inicijalizira HTTP zahtjev, pokreće router i provjerava odgovor.

```
for i := range testcases {
            tc := testcases[i]

            t.Run(tc.name, func(t *testing.T) {

                  ctrl := gomock.NewController(t)

                  defer ctrl.Finish()

                  store := mockdb.NewMockStore(ctrl)
                  //BUILD STUBS
                  tc.buildstubs(store)
                  //start test server

                  server := newTestServer(t, store)
                  recorder := httptest.NewRecorder()

                  url := fmt.Sprintf("/accounts/%d", tc.accountID)
                  request, err := http.NewRequest(http.MethodGet, url, nil)
                  require.NoError(t, err)
                  tc.setupAuth(t, request, server.tokenMaker)
                  server.router.ServeHTTP(recorder, request)
                  tc.checkResponses(t, recorder)

            })
```

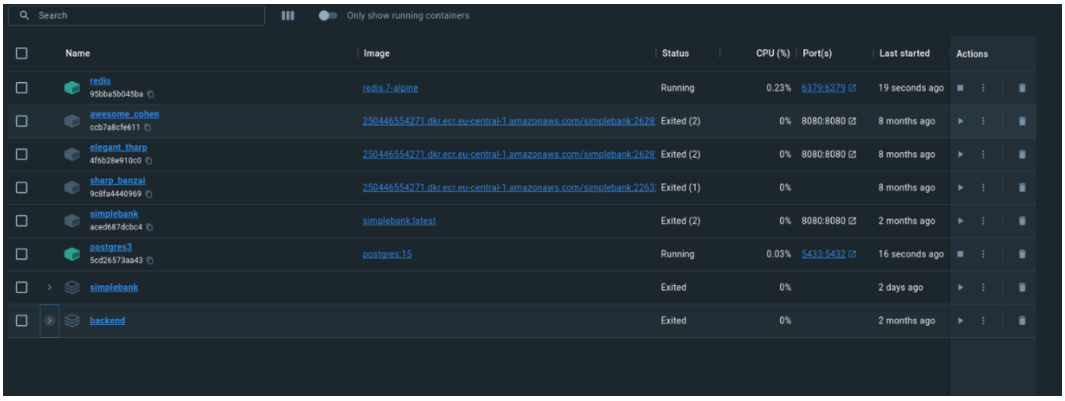
4.5 Isporuka rješenja

Docker omogućava pakiranje aplikacije sa svim ovisnostima u kontejnere. U aplikaciji imamo dva kontejnera. Jedan je za Redis, a drugi je za bazu podataka.

Ključne komponente Dockera:

1. **DockerFile** – Sadrži upute za kreiranje Docker slike aplikacije. Uključuje definiranje osnovne slike i instalaciju potrebnih ovisnosti. Za našu bazu podataka koristili smo PostgreSQL i Redis sliku.
2. **Docker-compose.yaml** – Docker compose se koristi za definiranje i jednostavno pokretanje više Docker servisa. Datoteka specificira konfiguraciju za različite servise kao što je baza podataka i Redis. Također u datoteci se definiraju mreže, volumeni i ovisnosti među servisima.

3. **Korištenja Dockera za migracije** – Docker se koristi i za upravljanje migracijama baze podataka. „Migrate“ alat pokreće se unutar Docker kontejnera kako bi se osigurala konzistentnost migracija.



Slika 12. Izgled docker desktopa, alata za upravljanje kontenjerima

Pokretanje Dockera

Naredba „docker-compose-up“ pokreće sve definirane servise u .yaml datoteci. Servisi uključuju aplikaciju, bazu podataka, te Redis. Docker Compose automatski kreira mreže i povezuje servise kako bi međusobno mogli komunicirati.

5. Zaključak

Kroz ovaj rad istražili, objasnili i implementirali smo mikroservisnu arhitekturu koristeći programski jezik Go. Cilj je bio demonstrirati kako suvremeni sustavi mogu izgraditi koristeći modularni pristup. Mikroservisna arhitektura omogućuje svakoj komponenti sustava da djeluje autonomno, što olakšava razvoj i održavanje.

Go je zbog svoje jednostavnosti, performanse i ugrađene podrške za istovremeno izvođenje idealan izbor za ovakav tip arhitekture. Njegova jednostavna sintaksa olakšava čitanje,pisanje čistog koda. Ugrađena podrška za gorutine i kanale omogućuje jednostavno pisanje, što je ključno za implementaciju mikroservisa.

U praktičnom dijelu rada izgrađen je bankovni servis koji uključuje funkcionalnosti, kao što su kreiranje korisnika, upravljanje računa i provedba transakcije s računa na račun. Koristili smo PostgreSQL kao bazu podataka, a za upravljanje migracijama koristili smo golang-migrate. Generiranje SQL koda postigli smo korištenjem SQLC.

Implementacija RESTFul API-ja korištenjem Gin okvira omogućila je jednostavno upravljanje HTTP zahtjevima i pružanje usluga korisnicima. Autentifikacija i autorizacija provedene su korištenjem JWT. Također, u radu je implementirano asinkrono programiranje, koristeći Redis i asynq biblioteku za upravljanje pozadinskim zadacima. Takav pristup omogućava obradu dugotrajnih operacija bez blokiranja glavne dretve.

6. Popis literature

[1]“What is Microservices“. (n.d.). Preuzeto 20.5. s https://microservices.io/

[2] “Microservice architecture style“(n.d). Preuzeto 25.5. s https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[3] Microservices(n.d.). Preuzeto 25.5. s https://aws.amazon.com/microservices/

[4] “The What, Why, and How of a Microservices Architecture“.(n.d.) Pristupano 25.5. s https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9

[5] “What are Microservices? Framework and Architecture Guide“(n.d.). Preuzeto 27.5. s https://www.talend.com/resources/what-is-a-microservice/

[6] “Microservices vs. monolithic architecture“ (n.d.). Preuzeto 5.6. s https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[7] “Microservices vs. Monolithic Architectures“(n.d.). Preuzeto 15.6. s https://www.baeldung.com/cs/microservices-vs-monolithic-architectures

[8] “Go“(n.d.). Preuzeto 15.6 s https://go.dev/,

[9] “Go Programming Language“(n.d.). Preuzeto 15.6. s https://www.geeksforgeeks.org/go-programming-language-introduction/

[10] “What is the Go or Golang programming language?“.(n.d.) Pristupano 20.6 s https://www.techtarget.com/searchitoperations/definition/Go-programming-language

[11] “What is API Gateway | System Design ?“(n.d.) Pristupano 15.6 s https://www.geeksforgeeks.org/what-is-api-gateway-system-design/

7. Popis Slika

8.Popis tablica