Web-aplikacije u programskom jeziku Go

Halavanja, Matija

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Matija Halavanja

WEB-APLIKACIJE U PROGRAMSKOM JEZIKU GO

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, Ožujak, 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Web-aplikacije su aplikacije dostupne kroz web-preglednik. Često, unutar preglednika vidimo samo mali dio cijele aplikacije, odnosno vidimo samo dio klijentske aplikacije. Klijentske aplikacije moraju komunicirati sa serverom kako bi aplikacija omogućila individualizaciju za svakog pojedinog korisnika. Jedan programski jezik za razvoj servera i drugih servisa koji nam omogućuju takvo iskustvo je Go, programski jezik razvijen u Googleu.

Cilj ovog diplomskog rada je napraviti pregled programskog jezika Go te opisati mogući način izgradnje web-aplikacije bazirane na Go-u. Osim pisanog, teorijskog dijela rada, u sklopu diplomskog rada razvijena je i aplikacija koja se temelji na konceptima iz drugog i trećeg poglavlja.

U prvom poglavlju ćemo napraviti detaljan pregled programskog jezika Go i njegove okoline. U drugom poglavlju prolazimo kroz koncepte, tehnologije i konkretne primjere za razvoj jedne općenite web-aplikacije u Go. Treće i zadnje poglavlje se fokusira na specifične, dodatne koncepte koji su korišteni u razvoju aplikacije u sklopu praktičnog dijela diplomskog rada.

Poglavlje 1

Kratki pregled programskog jezika Go

1.1 Općenito o programskom jeziku

Go je statički tipizirani, kompilirani programski jezik s ugrađenim oslobađanjem resursa (*eng. garbage collection*), sintaktički sličan C-u. Robert Griesemer, Rob Pike i Ken Thompson su programeri i računalni inženjeri iz Googlea koji su ga dizajnirali 2007. godine. Trenutna aktualna verzija je 1.19, a početkom 2023. godine se očekuje nova verzija 1.20. Osim spomenutog automatskog oslobađanja resursa, Go omogućava sigurno upravljanje memorijom, strukturno tipiziranje i jednostavan model paralelnosti. Go je dizajniran kako bi se povećala produktivnost kod razvoja aplikacija tako da je jezik vrlo koncizan.

Dizajneri Go-a su htjeli usvojiti određene pozitivne karakteristike:

- statičko tipiziranje i brzina izvođenja kao u C-u;
- čitljivost i jednostavnost korištenja kao u Pythonu ili Javascriptu;
- mrežna i višeprocesorska komunikacija visokih performansi.

Pošto je Go inspiriran i sintaksno sličan C-u, mogli bismo pomisliti da je Go sličan i programskom jeziku C++, no dizajneri Go-a su otvoreno kritizirali C++ i htjeli su izbjeći određene mane C++ poput prevelike kompleksnosti jezika i sporog vremena kompiliranja.

Osim spomenutih poboljšanja i zahtjeva na jezik, Go dolazi s još nekim ugrađenim konceptima poput:

- tipovi podataka za višedretveno programiranje: lagani procesi (*goroutines*), kanali (*channels*) i naredba *select*;
- ugrađivanje tipova umjesto nasljeđivanja;
- sistem sučelja umjesto virtualnog nasljeđivanja.

Nakon kompiliranja izvorne datoteke pisane u Go-u dobivamo jednu izvršnu datoteku, statički povezanu sa svim potrebnim vanjskim paketima.

1.2 Sintaksa

Za razliku od mnogih drugih programskih jezika, u Go-u kod deklaracije varijabli prvo pišemo ime varijable pa nakon toga tip varijable, kao što ćemo vidjeti kroz većinu primjera u ovom poglavlju. Novost koju Go uvodi naspram C-a je zaključivanje tipova pomoću := operatora kod inicijalizacije tako, na primjer, možemo pisati kod kao u isječku 1.1 umjesto analognog isječka 1.2 pisanog u Javi. U donjem primjeru, Go automatski zaključi da je varijabla `x` tipa `int`, a varijabla `y` tipa `string`.

```
x := 0
y := "Neki tekst"
```

Primjer 1.1: Deklaracija i inicijalizacija varijabli u Go-u

```
int x = 0;
String y = "Neki tekst";
```

Primjer 1.2: Deklaracija i inicijalizacija varijabli u Javi

Točka-zarez (;) se koristi kao i u C-u na završetku nekog izraza, no u Go-u, točka-zarez se implicitno pretpostavlja na kraju retka tako da se ne mora eksplicitno napisati.

Osim uobičajenog string tipa u kojem možemo izbjeći (*escape*) znakove, postoji i doslovni string koji se piše unutar povratnih kvačica (*backticks*), u oznaci “. Unutar povratnih kvačica, svi znakovi se interpretiraju doslovno. To može biti korisno ako želimo tekst koji nije samo u jednome redu ili ako želimo koristiti posebne znakove poput duplih navodnih znakove, koji se koriste za uobičajene stringove.

Za razliku od većine programskih jezika, Go nema ključnu riječ `while`. Umjesto `while`, Go koristi `for` petlju. U sljedećem primjeru su prikazana četiri načina korištenja `for` petlje. Prvi je uobičajen način gdje imamo inicijalizaciju varijable, krajnji uvjet i ažuriranje varijable. Drugi način korištenja je identičan `while` petlji u mnogim drugim programskim jezicima. Treći način je da koristimo `for` kao beskonačnu petlju. U donjem primjeru koristimo `break` da je prekinemo, a usput, `break` i `continue` rade na uobičajen način. Četvrti način korištenja `for` izraza je u `for-range` sintaksi koju koristimo za iteriranje po određenim tipovima podataka.

```
package main

import "fmt"

func main() {
  sum := 0
```

```
  for i := 1; i < 5; i++ {
    sum += i
  }
  fmt.Println(sum) // 10

  sum = 0
  i := 1
  for i < 5 {
    sum += i
    i++
  }
  fmt.Println(sum) // 10

  sum = 0
  for {
    sum++
    if sum == 10 {
      break
    }
  }
  fmt.Println(sum) // 10

  letters := []string{"a", "b", "c"}
  for i, s := range letters {
    fmt.Println(i, s)
  }
}
```

Primjer 1.3: Različiti načini korištenja izraza for u Go-u

Sljedeća vrlo korisna stvar koju Go uvodi je mogućnost funkcija da koriste više od jedne varijable, istih ili različitih tipova podataka, kao svoje povratne vrijednosti, što vidimo u sljedećem primjeru.

```
func LoadConfig(path string) (Config, error) {
  ...
}
```

Primjer 1.4: Funkcija s dvije povratne vrijednosti

Funkcija `LoadConfig` ima jedan ulazni parametar `path`, tipa `string` te dvije povratne vrijednosti od kojih je prva tipa `Config`, a druga tipa `error`. Ako funkcija općenito ima više od jedne povratne vrijednosti, tipove tih povratnih vrijednosti kod definicije funkcije moramo navesti unutar zagrada, baš kao u gornjem primjeru. Ovim konceptom se izbjegava potreba za nespretnim uvođenjem dodatnog spremnika (npr. `List` ili `Tuple`) samo kako bismo iz jedne funkcije vratili više povratnih vrijednosti te ih odmah raspakirali u funkciji koja prima te vrijednosti. U gornjem primjeru također vidimo kako se u Go-u, za razliku od većine drugih programskih jezika, tip varijable piše nakon imena varijable.

Također, u statički tipiziranim jezicima poput Jave, ako želimo imati sigurnost tipova podataka kada želimo vratiti više varijabli različitih tipova podataka kao vrijednosti neke funkcije, morali bismo uvesti novu klasu koja sadrži sve te tipove podatka (ne želimo vratiti `List <Object>` ili nešto slično čime narušavamo sigurnost tipova podataka).

Umjesto `try - catch` sintakse za upravljanje greškama, Go se oslanja na upravo spomenuti mehanizam vraćanja više povratnih vrijednosti. Funkcije često vraćaju neku varijablu i grešku koja se može dogoditi tijekom izvođenja te funkcije. Ako pak funkcija ne vraća grešku, može vratiti `nil` (ekvivalent `null` ili `None` u nekim drugim programskim jezicima) što bi značilo da se greška nije dogodila. Tada na mjestu gdje se takva funkcija poziva, jednostavno možemo s izrazom `if` provjeriti je li vraćena vrijednost te funkcije `nil` ili nije, te ovisno o tome znamo je li se desila greška ili ne.

```
1  config, err := util.LoadConfig(".")
2  if err != nil {
3    log.Fatal("cannot load configuration: ", err)
4  }
```

Primjer 1.5: Upravljanje greškama u Go-u

`defer` je ključna riječ u Go-u koja odgađa izvršavanje izraza. Preciznije, kada unutar funkcije napišemo `defer`, dani izraz će se izvršiti na kraju izvođenja funkcije, prije nego što se tok izvođenja programa vrati funkciji koja ju je pozvala, a poslije zadnjeg izraza unutar trenutne funkcije. `defer` je posebno koristan da grupiramo kod za oslobađanje resursa ili hvatanje grešaka s kodom koji zauzima resurs ili bi mogao baciti grešku. Sljedeći primjer ilustrira korištenje `defer` naredbe kod otvaranja i zatvaranja datoteke.

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6  )
7
8  func main() {
9      f := createFile("/tmp/defer.txt")
10     defer closeFile(f)
11     writeFile(f)
12 }
13
14 func createFile(p string) *os.File {
15     f, err := os.Create(p)
16     if err != nil {
17         panic(err)
18     }
19     return f
20 }
```

```
21
22 func writeFile(f *os.File) {
23     fmt.Fprintln(f, "data")
24 }
25
26 func closeFile(f *os.File) {
27     err := f.Close()
28
29     if err != nil {
30         fmt.Fprintf(os.Stderr, "error: %v\n", err)
31         os.Exit(1)
32     }
33 }
```

Primjer 1.6: Korištenje defer naredbe

U gornjem primjeru kreiramo datoteku **/tmp/defer.txt**, pišemo u nju i zatvaramo je. U nekim drugim programskim jezicima, morali bi koristiti koncepte poput `try-catch-finally` pomoću kojih bi osigurali da (neovisno o tome je li se dogodila greška ili nije) zatvorimo datoteku. Pomoću `defer` imamo elegantan način da grupiramo kod otvaranja i zatvaranja datoteke i ne brinemo o greškama. Iz gornjeg primjera još vrijedi spomenuti ugrađenu funkciju `panic` s kojom možemo prisilno zaustaviti izvođenje programa ako se dogodila greška od koje se program nikako ne može oporaviti.

Na kraju ovog potpoglavlja ćemo prikazati *quick sort* algoritam kako bi se čitatelji dodatno naviknuli na sintaksu Go-a.

```
1  package main
2
3  import "fmt"
4
5  func partition(arr []int, low, high int) int {
6    pivot := arr[high]
7    i := low - 1
8    for j := low; j < high; j++ {
9      if arr[j] <= pivot {
10       i++
11       arr[i], arr[j] = arr[j], arr[i]
12     }
13   }
14   arr[i+1], arr[high] = arr[high], arr[i+1]
15   return i + 1
16 }
17
18 func quicksort(arr []int, low, high int) {
19   if low < high {
20     pi := partition(arr, low, high)
21     quicksort(arr, low, pi-1)
```

```
22      quicksort(arr, pi+1, high)
23    }
24  }
25
26  func main() {
27    arr := []int{38, 27, 43, 3, 9, 82, 10}
28    quicksort(arr, 0, len(arr)-1)
29    fmt.Println(arr)
30  }
```

Primjer 1.7: Quick sort

1.3 Tipovi podataka

Kao što smo već spomenuli, Go je statički tipiziran programski jezik. Go ima podršku za numeričke tipove različite duljine, s ili bez predznaka, decimalne brojeve i kompleksne brojeve. Također ima ugrađene tipove *bool* i *string* koji redom predstavljaju Booleov tip i tekstualni tip podatka. Tip *string* je nepromjenjiv u Go-u, što znači da nakon inicijalizacije više nije moguće modificirati znakove od kojih je sastavljen, odnosno ako to napravimo, konstruirat će se nova vrijednost u memoriji (slično je, na primjer, u Javi). Za svaki tip T i nenegativni cijeli broj n, postoji tip polja duljine n čiji su elementi tipa T, a koja se označavaju s *[n]T*. Osim polja fiksne duljine, postoje i polja dinamičke duljine (*slice*) koji se označava s *[]T*. Za par tipova podataka K i V, postoji tip hash tablice *map[K]V* koji pridružuje ključu tipa K vrijednost tipa V. Kao i u C-u, za svaki tip podataka T, postoji tip podatka pokazivač na T, a označavamo ga s *T. Operatori referenciranja i dereferenciranja se također koriste kao u C-u. Aritmetika s pokazivačima je dostupna jedino koristeći paket *unsafe* i njegovu strukturu *Pointer*. Spomenuta aritmetika nije dozvoljena s "običnim" pokazivačima.

Jedna prava novost koju Go uvodi naspram popularnih, modernih programskih jezika je tip *chan T* koji predstavlja kanal (*channel*) koji se koristi za slanje vrijednosti tipa T između paralelnih Go procesa. Više ćemo govoriti o tome u sljedećem poglavlju.

Go ima rezerviranu riječ *type* s kojom se mogu definirati novi tipovi podataka. Sistem tipova podataka u Go-u je nominalni, što znači da se tipovi podataka razlikuju po njihovom imenu. To znači da možemo definirati vlastite tipove podataka koji su svojim sadržajem jednaki nekom postojećem tipu podataka, no oni će se smatrati različitima u svakom pogledu. Konverzija iz jednog u drugi mora biti eksplicitna, te se na vlastitom tipu mogu definirati neke metode koje se neće moći koristiti na naizgled istom tipu podatka koji se drugačije zove. Primjer toga je dan sljedećim isječkom koda koji također prikazuje sintaksu definiranja metode na nekoj klasi. U donjem primjeru, `ID` je klasa na kojoj se definira nova metoda `ToClusterName` koja nema niti jedan parametar, a povratni tip podatka

joj je `string`. `id` u ovom slučaju predstavlja referencu na sami objekt klase `ID` na kojoj se metoda definira (drugim riječima, služi nam kao ključna riječ `this` u Javi ili `self` u Pythonu).

```
type ID string
func (id ID) ToClusterName() string {
  return "mongodb-" + string(id) + "-cluster"
}
```

Primjer 1.8: Nominalni sistem tipova u Go-u

Tip varijable možemo promijeniti koristeći koncept konverzije tipova (*type conversion*) koji je upravo prikazan u gornjem primjeru sa `string(id)`. Kada ne bismo koristili `string(id)`, već samo `id` u prethodnom primjeru, došlo bi do greške jer Go ne može automatski pretvoriti podatak tipa *ID* u podatak tipa *string*. Druga stvar koju smo spomenuli s nominalnim sustavom tipova podataka je također ilustrirana ovim primjerom. Kada bismo pokušali pozvati metodu *ToClusterName* na nekoj varijabli tipa *string*, dobili bi grešku jer iako sadrže iste podatke, metoda nije definirana na tipu *string*. Spomenuti koncept konverzije tipova se odvija za vrijeme kompiliranja što znači da će nam kompajler vratiti grešku pri kompiliranju ako konverzija tipova nije uspješna. Oblik konverzije za vrijeme izvođenja programa ćemo spomenuti u sljedećem potpoglavlju.

U sami jezik ugrađen je i funkcijski tip podataka odnosno funkcije koje se označuju s ključnom riječi *func*. Konkretan tip funkcije je određen brojem i tipovima njenih parametara i povratnih vrijednosti. Funkcije se mogu koristiti u definicijama drugih funkcija i možemo ih pohraniti u varijable. Sljedeći primjer ilustrira spomenute koncepte.

```
package main

import "fmt"

type Operator func(int, int) int

func doOperation(a, b int, op Operator) int {
  return op(a, b)
}

func main() {
  plus := func(a, b int) int {
    return a + b
  }
  fmt.Print(doOperation(3, 4, plus)) // 7
}
```

Primjer 1.9: Nominalni sistem tipova u Go-u

Osim spomenutih tipova podataka, postoje i tip sučelja *interface* o kojem ćemo više govoriti u sljedećem potpoglavlju, a kao i sučelja u mnogim drugim programskim jezicima, ovisi o funkcijama koje su definirane na konkretnom sučelju.

1.4 Sistem sučelja i objektno orijentirano programiranje

Programski jezik Go je dizajniran tako da nema jedno od osnovnih svojstava objektno orijentiranog programiranja, nasljeđivanje. Umjesto toga, koriste se dva koncepta koja ga zamjenjuju. Prvi je ugrađivanje (*embedding*) što je jedna vrsta kompozicije objekata. Koncept je najjasniji ako ga pokažemo na primjeru.

```
type Shape struct {
  Sides int
}

func (shape Shape) GetNumOfSides() int {
  return shape.Sides
}

type Triangle struct {
  Shape
  Color string
}

type Square struct {
  Shape
  Name string
}

func main() {
  s1 := Triangle{
    Shape: Shape{
      Sides: 3,
    },
    Color: "red",
  }
  fmt.Println(s1.GetNumOfSides()) // 3

  s2 := Square{
    Shape: Shape{
      Sides: 4,
    },
    Name: "Richard",
  }
  fmt.Println(s2.GetNumOfSides()) // 4
```

```
35 }
```

Primjer 1.10: Ugrađivanje struktura

Klasa *Shape* je ugrađena u klase *Triangle* i *Square* što znači da instance tih klasi imaju pristup atributima i metodama definiranim na klasi *Shape*. Kod definiranja struktura *Triangle* i *Square* vidimo kako ime tipa podatka kojeg ugrađujemo samo trebamo napisati uz ostale članske varijable, bez dodavanja imena varijable. Problemom višestrukog nasljeđivanja ćemo se još pozabaviti na kraju ovog potpoglavlja.

Kada definiramo članove struktura u Go-u, ona mogu imati dodatnu meta informaciju pored sebe u obliku stringa koja se naziva *tag*. Iako tag može biti bilo kakav string, postoji konvencija koja se prati u raznim popularnim Go paketima. Konvencija je za tag koristiti string oblika *key:"value"* gdje onda treba imati na umu da takav string mora biti unutar povratnih kvačica. Tagovima možemo pristupiti koristeći paket (više o paketima kasnije) *reflect*. Ako ne koristimo paket *reflect* unutar vlastitog koda ili unutar vanjskog koda koji pozivamo, tagovi se u potpunosti ignoriraju. Slijedi primjer strukture koja koristi tagove, a cijelu klasu *Config* ćemo prikazati i pri kraju sljedećeg poglavlja.

```
1 type Config struct {
2   DBDriver            string        `mapstructure:"DB_DRIVER"`
3   DBSource            string        `mapstructure:"DB_SOURCE"`
4   ServerAddress       string        `mapstructure:"SERVER_ADDRESS"`
5   Client              string        `mapstructure:"CLIENT_ADDRESS"`
6 }
```

Primjer 1.11: Struktura s tagovima

Osim struktura ili klasa vrlo važan koncept u objektno orijentiranom programiranju predstavljaju i sučelja (*interfaces*). Sučelja sadržavaju funkcije koje su definirane svojim imenom te brojem i tipovima podataka koje primaju i vraćaju. Svaki objekt tipa T za kojeg su definirane sve funkcije koje definira neko sučelje I je i tip tog sučelja. To znači da se taj objekt može pridružiti varijabli tipa sučelja I.

Kada neku varijablu x tipa podatka T pridružimo sučelju I, njegov temeljni (*underlying*) tip podatka T ostaje isti te pohranjene informacije nisu izgubljene. Njima možemo pristupiti ponovno ako varijabli x utvrdimo (*type assertion*) originalni tip podatka. Za razliku od konverzije tipova, utvrđivanje tipova se odvija za vrijeme izvođenja aplikacije. Također za razliku od konverzije tipova, utvrđivanje tipa se može raditi samo na sučeljima, što je zapravo i logično pošto varijablama koje nisu tipa sučelja znamo temeljni tip podatka. Za utvrđivanje tipa podatka postoje sintaktički dva različita načina, ali oba se temelje na istom spomenutom konceptu. Prvi način je dan sljedećim primjerom.

```
1 var greeting interface{} = "Hello world"
2 greetingStr, ok := greeting.(string)
```

Primjer 1.12: Potvrđivanje tipova

Varijabla *ok* je tipa *boolean* koja nam govori je li potvrda tipa bila uspješna ili ne. U gornjem primjeru *greetingStr* varijabla je tipa *string* i sada kada imamo pristup temeljnom tipu podatka, možemo napraviti dodatne stvari poput koristiti razne metode koje su definirane za tip podatka *string*. Drugi način za pretvaranje tipa podatka je koristiti *type switch* izraz kojeg koristimo na poznat način ilustriran sljedećim primjerom koji se nadovezuje na gornji.

```
func printType(g Greeting) {
  switch g.(type) {
  case HiGreeting:
    fmt.Println("Hi type")
  case HelloGreeting:
    fmt.Println("Hello type")
  default:
    fmt.Println("No type")
  }
}
```

Primjer 1.13: Korištenje *type switch* izraza kod potvrđivanja tipova

U programskim jezicima s uobičajenim nasljeđivanjem klasa tipično možemo koristiti polimorfizam za vrijeme izvođenja (*runtime polymorphism*). Sljedeći primjer prikazuje kako u Go-u koristimo sučelja kako bi omogućili upravo to.

```
package main

import "fmt"

type Person interface {
  Greet()
}

type Friend string

func (friend Friend) Greet() {
  fmt.Println("Hi", friend)
}

type Professor string

func (professor Professor) Greet() {
  fmt.Println("Hello professor", professor)
}

func main() {
  var person Person

  person = Friend("Mark")
```

```
  person.Greet() // Hi Mark

  person = Professor("Marquina")
  person.Greet() // Hello professor Marquina
}
```

Klasa *Friend* i klasa *Professor* implementiraju sve metode sučelja *Person*, što je zapravo samo metoda *Greet*. Zbog toga, varijabla *person* koja je deklarirana kao varijabla tipa sučelja *Person* može za vrijeme izvođenja programa poprimiti vrijednosti tipa tih klasa. Koja metoda se poziva na toj varijabli ovisi o tome kojeg je tipa varijabla *person* za vrijeme izvođenja programa. Sada također vidimo da su sučelja u Go-u, za razliku od ostatka jezika, strukturalno tipizirana, a ne nominalno.

Kako nemamo klasično nasljeđivanje, nemamo, naravno, niti uobičajeno višestruko nasljeđivanje. To ne bi trebalo stvarati problem, jer se na isti način može ugrađivati više od jednog tipa podatka. Ilustraciju toga možemo vidjeti na sljedećem primjeru 1.14 koji nam također odgovara na pitanje kako Go rješava dijamantni problem (*Diamond problem*). Problem nastaje kod jezika koji imaju višestruko nasljeđivanje, ali i ugrađivanje struktura dovodi do istog problema. Neka imamo klase *B* i *C* koje obje nasljeđuju ili ugrađuju sučelje *A* i klasu *D* koja ugrađuje klase *B* i *C*. Nadalje, ako instanciramo objekt *d* tipa *D* i pozovemo funkciju definiranu u sučelju *A*, dolazi do dvoznačnosti. Naime, Go ne može znati treba li pozvati funkciju definiranu u klasi *B* ili klasi *C*. Sve je to prikazano na donjem primjeru gdje i vidimo kako Go rješava dijamantni problem.

```
package main

import "fmt"

type A interface {
  F()
}

type B struct {
  A
}

func (b B) F() {
  fmt.Println("F() of B")
}

type C struct {
  A
}

func (c C) F() {
  fmt.Println("F() of C")
```

```
23 }
24
25 type D struct {
26   B
27   C
28 }
29
30 func main() {
31   d := D{}
32
33   // d.F() // Error! "ambiguous selector d.F".
34
35   d.B.F() // "F() of B"
36   d.C.F() // "F() of C"
37 }
```

Primjer 1.14: Dijamantni problem i višestruko ugrađivanje

Dakle, ako bismo pokušali napisati `d.F()`, dobili bismo grešku. Rješenje je u tome da specificiramo točno koja nam implementacija treba tako da napišemo `d.B.F()` ili `d.C.F()`.

Za kraj potpoglavlja, spomenimo jedno vrlo važno sučelje u Go-u, a to je prazno sučelje odnosno `interface{}`. Ono zbog svoje važnosti ima i alias `any` koji se može koristiti potpuno isto, umjesto `interface{}`. Ako smo pažljivo čitali ovo potpoglavlje, shvaćamo da je svaki tip podatka ujedno i tipa `interface{}`, što je ujedno i razlog zašto je alias upravo riječ `any`. To znači da ako funkcija prima `interface{}`, može primiti varijablu bilo kojeg tipa podatka u Go-u, odnosno varijablu bilo kojeg tipa podatka možemo konvertirati u tip `interface{}`. Sličan koncept u Javi predstavlja klasa `Object` koja je na vrhu hijerarhije nasljeđivanja, odnosno svaka klasa u Javi zapravo nasljeđuje od `Object` klase i može se napraviti konverzija iz neke klase u klasu `Object`.

1.5 Paralelno programiranje

Jedan od primarnih ciljeva programskog jezika Go i razloga njegovog nastajanja je da olakša paralelno i asinkrono programiranje, odnosno programiranje s više dretvi. Go to postiže koristeći tzv. *gorutine*(*goroutines*) koje su lagane dretve (*light-weight threads*) kontrolirane od strane Go okoline za vrijeme izvođenja (*runtime environment*). Gorutinu dobijemo koristeći rezerviranu riječ *go* koja se stavi ispred poziva funkcije. To za posljedicu ima da se evaluacija funkcije dogodi u trenutnoj gorutini (možemo zamišljati u trenutnoj dretvi), a samo izvođenje te funkcije se događa u drugoj gorutini. Kao i dretve, gorutine se izvode u istom adresnom prostoru pa se moraju sinkronizirati kada pristupaju zajedničkoj memoriji.

```
1 package main
```

```
2
3  import (
4    "fmt"
5    "sync"
6  )
7
8  var wg sync.WaitGroup
9
10 func f() {
11   defer wg.Done()
12   for i := 0; i < 1000; i++ {
13     fmt.Println(i)
14   }
15 }
16
17 func main() {
18   wg.Add(2)
19   go f()
20   go f()
21   wg.Wait()
22 }
```

Primjer 1.15: Korištenje gorutina

U prethodnom primjeru ispisat će se po dva puta svi brojevi između 0 i 1000, no nemamo garanciju kojim će redom biti ispisani. Također, moramo koristiti *sync.WaitGroup* i njene funkcije `Done`, `Add` i `Wait` jer gorutina u kojoj se izvodi funkcija *main* neće implicitno čekati da ostale gorutine završe s izvođenjem.

Gorutine mogu slati i primati podatke koristeći već spomenute kanale. Kao zadana opcija, slanje i primanje poruka blokira izvršavanje gorutine dok druga strana nije spremna. Ovo pak omogućava sinkronizaciju gorutina bez eksplicitnog korištenja lokota ili nekih kondicionalnih varijabli. Kanali također mogu imati i spremnik zadane veličine. Tada slanja poruka blokiraju, tek kada je spremnik pun, a primanja poruka blokiraju kada je spremnik prazan. Kanal se može i zatvoriti, što obično radi gorutina koja šalje podatke kada je sigurna da više neće slati podatke. To se radi pomoću ključne riječi *close*, no za razliku od primjerice datoteka, zatvaranje kanala nije obavezno. S kanalima možemo koristiti i koncept *for - range*, što označava primanje vrijednosti iz kanala dok se kanal ne zatvori.

```
1  package main
2
3  import "fmt"
4
5  func f(c chan int) {
6    for i := 0; i < cap(c); i++ {
7      c <- i
```

```
  }
  close(c)
}

func main() {
  c := make(chan int, 10)
  go f(c)
  for i := range c {
    fmt.Println(i)
  }
}
```

Primjer 1.16: Korištenje kanala

U gornjem, pomalo dosadnom primjeru, ispisujemo brojeve od 0 do 9 redom, no primjer prikazuje sve prethodno spomenute koncepte u primjeni. Također koristimo funkciju *cap* koja vraća kapacitet kanala.

Kako bi se olakšala manipulacija s više istovremenih gorutina, postoji izraz *select*. Koristeći *select*, gorutina čeka dok se ne ispuni jedan od više uvjeta i onda se izvršava ta grana izvođenja. Ako je više uvjeta zadovoljeno u isto vrijeme, bira se jedan nasumično. Izraz *select* može imati i zadani slučaj koji se izvršava ako nijedan drugi uvjet nije ispunjen. Sljedeći isječak koda prikazuje kako se koristi *select*.

```
package main

import "fmt"

func f(x int, c chan int) {
  for i := 5*(x-1); i < 5*x; i++ {
    c <- i
  }
}

func main() {
  c1 := make(chan int)
  c2 := make(chan int)
  go f(1, c1)
  go f(2, c2)
  for {
    select {
      case x:= <-c1:
      fmt.Println(x)
    case x:= <-c2:
      fmt.Println(x)
      if x == 9 {
      return
      }
```

```
25      default:
26        fmt.Println(".")
27      }
28    }
29  }
```

Primjer 1.17: Korištenje *select* izraza

U gornjem primjeru ispisujemo cijele brojeve između 0 i 9 (ne nužno sve) te točku, ako nijedna gorutina nije poslala vrijednost u svoj kanal.

1.6 Okolina i pomoćni alati

Go možemo jednostavno instalirati na vlastito računalo sa službene stranice [6]. Nakon same instalacije, moramo promijeniti i PATH okolinsku varijablu kako bismo mogli koristiti program `go` u terminalu. To je program koji koristimo za razvoj aplikacija pisanih u Go programskom jeziku. Dakle, postoji program go i programski jezik Go. Kada se odnosi na program, obično se piše s malim slovom jer se tako i zove izvršna datoteka, odnosno program koji pokrećemo.

Program go dolazi s mnogo naredbi korisnih za razvoj aplikacija, a ovdje ćemo spomenuti neke najvažnije. Za početak, Go kao i mnogi drugi programski jezici se oslanja na neke okolinske varijable za svoj rad. Te okolinske varijable možemo prikazati s `go env`, a pojedinu varijablu možemo promijeniti s `go env -w <ime_variable>=<vrijednost>`.

Ako želimo instalirati neki Go program i koristiti ga na vlastitom računalu, koristimo `go install <ime_paketa>@<verzija>`, gdje verzija može biti i rezervirana riječ *latest* koja označava da želimo najnoviju verziju paketa. Dakle, `go install` preuzme programski kod, kompilira ga i instalira na računalo, a izvršna datoteka se sprema u direktorij koji sami možemo odrediti tako da damo vrijednost još jednoj okolinskoj varijabli pod nazivom GOBIN. Ako ne specificiramo vrijednost te varijable, Go izvršne datoteke će se spremati u direktorij **$GOPATH/bin**. U taj isti direktorij će se spremati i izvršne datoteke svih naših lokalnih programa. To znači da ako se $GOBIN (ili $GOPATH/bin ako GOBIN nije specificirana) nalazi unutar PATH varijable, Go programe možemo jednostavno pozivati kroz terminal samo s imenom izvršne datoteke, bez da specificiramo punu putanju do datoteke.

Kada koristimo neki vanjski paket u vlastitoj aplikaciji, moramo taj paket imati lokalno na računalu. To postižemo naredbom `go get <ime_paketa>`. `go get` služi za dodavanje, ažuriranje i brisanje paketa koji su referencirani u *go.mod* datoteci projekta pa iz tog razloga, naredba se ne može izvršiti jedino unutar direktorija koji sadrži *go.mod* datoteku ili ako neki od njegovih naddirektorija sadrži *go.mod* datoteku (drugim riječima, ako smo unutar direktorija koji je dio nekog Go projekta). Go također dolazi s naredbom za formatiranje izvornih datoteka, `go fmt`. Formatiranje u ovom kontekstu znači dodavanje ili micanje bjelina u datoteci s izvornim kodom s ciljem da kod bude čitljiviji.

Formatiranje naravno ne utječe na semantiku koda. Posebno korisna verzija ove naredbe je `go fmt ./...` koja će formatirati sve Go datoteke u trenutnom direktoriju i svim poddirektorijima. Naredba će također ispisati imena datoteka koje su promijenjene.

`go run <ime_datoteke>` gdje je *ime_datoteke* ime neke Go izvorne datoteke koja sadrži funkciju *main*, kompilira i pokrene dobivenu izvršnu datoteku.

`go mod <naredba>` gdje *naredba* opet može biti neka ključna riječ. Iako ima više mogućih naredbi (i onda još dodatni argumenti za svaku pojedinu naredbu) spomenut ćemo samo jednu. `go mod init` inicijalizira i kreira novu datoteku *go.mod* u trenutnom direktoriju, što zapravo stvara novi modul s trenutnim direktorijem kao korijenskim. Iz tog razloga, datoteka *go.mod* ne smije već postojati u danom direktoriju.

Zadnja naredba koju ćemo spomenuti je `go test` koja kompilira sve pakete i pokreće datoteke čije je ime formata **_test.go* te ispisuje rezultate testova. Ova naredba kao i ostale, imaju još mnogo drugih opcija prilikom pokretanja. Detaljniji pregled svih opcija je dostupan unutar dokumentacije samog go programa [4].

1.7 Upravljanje paketima i modulima

Go koristi pakete (*packages*) i module (*modules*) kao organizacijske jedinice koda. Paket je kolekcija izvornih datoteka u istom direktoriju koji se kompiliraju zajedno. Modul je kolekcija povezanih go paketa koji se objavljuju (*release*) zajedno. Datoteka nazvana `go.mod` sadrži reference na sve vanjske pakete koje koristimo u svojoj aplikaciji. Osim toga, u istoj datoteci se deklarira *module path* koji označava prefiks za uvoz (*import*) svih paketa unutar tog modula. Modul sadrži sve pakete unutar direktorija koji sadrži `go.mod` datoteku, ali i sve pakete koji se nalaze u poddirektorijima tog direktorija sve do poddirektorija koji također sadrži `go.mod` datoteku. Ipak, tipično Go repozitoriji sadrže samo jedan modul koji se nalazi u korijenskom direktoriju.

Za neke objekte (varijable, funkcije, strukture) želimo da budu vidljivi samo unutar paketa u kojem su definirani, a za neke želimo da budu vidljivi i izvan tog paketa. U Go-u se to postiže tako što ime objekta može počinjati malim slovom, i tada objekt neće biti vidljiv izvan paketa, ili velikim slovom kada će biti vidljiv izvan paketa u kojem je definiran. Na početku svake izvorne datoteke pisane u Go-u definiramo u kojem se paketu datoteka nalazi. Ime paketa najčešće odgovara direktoriju u kojem se datoteka nalazi. Iako postoje načini da se to izbjegne, lakše je držati se ovog pravila. Važno je spomenuti da moduli nisu postojali od početka programskog jezika Go, već su uvedeni u verziji Go 1.11 stoga postoje alati, okolinske varijable i načini razvoja aplikacije koji se ne bi trebali koristiti od kada postoje moduli koji su postali preferirani način verzioniranja Go paketa. Fokus ovoga dijela diplomskog rada je pregled i približavanje modernog Go-a čitateljima te najbolje prakse u razvoju Go aplikacija pa iz tog razloga se nećemo fokusirati na stariji način korištenja Go paketa, prije modula.

Kako bismo koristili druge pakete, bilo lokalne iz trenutnog Go projekta ili neke vanjske, koristimo ključnu riječ `import` nakon koje slijedi *import path* što je tekst koji jedinstveno identificira neki paket. U isječku koda 1.18 možemo vidjeti tipičan primjer početka neke izvorne datoteke pisane u Go-u.

```
1  package api
2
3  import (
4    "database/sql"
5    "github.com/mhalavanja/go-rest-api/db/sqlc"
6    "github.com/mhalavanja/go-rest-api/token"
7    "log"
8    "net/http"
9
10   "github.com/gin-gonic/gin"
11 )
```

Primjer 1.18: Početak izvorne datoteke pisane u Go-u

U gornjem primjeru, u prvoj liniji se nalazi ime paketa (`api`) kojem pripada datoteka. Nadalje, nabrajamo pakete koje uvozimo u našu datoteku.

U praktičnom dijelu ovog diplomskog rada, *module path* varijabla je postavljena na `github.com/mhalavanja/go-rest-api`. Tako postavljena *module path* varijabla je uobičajena za Go i napisana je u specifičnom formatu *github.com*\<*korisničko ime*>\<*projekt*>. Takvo imenovanje vlastitih paketa nije neophodno za rad aplikacije, ali je vrlo praktično iz više razloga.

Prvo, paketi standardne biblioteke dobivaju kratke nazive paketa poput gornja tri primjera `database/sql`, `log` i `net/http` tako da moramo izabrati vrijednost *module path* varijable tako da bude različita od postojećih i budućih paketa iz standardne biblioteke.

Drugo, iako možda ne planiramo objavljivati vlastiti kod na nekoj platformi za verzioniranje i dijeljenje izvornog koda poput GitHub-a, možda ćemo ubuduće htjeti, a dodatna prednost je što ovako imenovani paketi sigurno neće postojati u budućnosti u standardnoj biblioteci ili u većem Go ekosustavu. Vrijedi još i spomenuti da nije nužno uopće imati GitHub korisnički račun kako bismo imenovali pakete na ovaj način.

Treće, ako želimo da naši paketi budu dostupni drugima za korištenje unutar njihovih izvornih datoteka, pomoćni alati koje Go interno koristi moraju znati gdje se nalazi naš paket. Koristeći GitHub (ili neki drugi sustav poput GitLaba) i pristup otvorenog koda (*open source*), automatski imamo tu mogućnost. Svaka izvorna datoteka pisana u Go-u, za koju želimo da se može izvršiti kao program, mora biti dio posebnog paketa pod imenom `main`. Unutar te datoteke također mora postojati i funkcija pod istim imenom koja onda služi kao polazna točka za izvršavanje programa. Ako pak pišemo samo paket s određenim funkcionalnostima koji se ne treba moći izvršiti kao zaseban program, onda nam ne trebaju paket i funkcija `main`.

Poglavlje 2

Razvoj web aplikacija

U ovom poglavlju ćemo opisati tehnologije i koncepte koje bismo mogli koristiti kod razvoja web aplikacije pisane u programskom jeziku Go. Općenito, web aplikacija je aplikacija koja se izvršava na web serveru, a pristupa joj se pomoću web preglednika. Postoji više paradigmi razvoja web aplikacija, a jedna od najstarijih i još uvijek najpopularnijih je model servera i klijenta. Uz server se i često veže riječ *backend* (jer nije izravno vidljiv korisniku), a uz klijent *frontend* (jer je vidljiv korisniku i njega koristi izravno).

Samo poglavlje se sastoji od četiri potpoglavlja. Prvo ćemo se upoznati s vrlo fleksibilnim razvojnim okvirom *Gin* koji koristimo za *backend* infrastrukturu naše aplikacije. Zatim govorimo o modelu i poslovnoj logici te paketu *sqlc* koji koristimo kao srednju opciju između klasičnih ORM sustava i izravne manipulacije upitima i bazom unutar samog *backend* koda. Treće potpoglavlje govori o autentifikaciji korisnika koje se ne temelji na dugotrajnoj sesiji (*session*) već na kratkotrajnim tokenima. U zadnjem potpoglavlju ove cjeline govorimo o tome kako aplikaciju možemo koristiti u različitim okolinama te kako upravljamo konfiguracijskim varijablama koje aplikacija koristi.

Ovakav način razvoja web aplikacije posebno je pogodan za REST API-je. Mnogo primjera iz ovog poglavlja je upravo i uzeto iz praktičnog dijela ovog diplomskog rada. U zadnjem poglavlju ćemo opisati i pokazati primjer potpune aplikacije koja koristi REST API napravljen po principima iz ovog poglavlja.

2.1 Backend infrastruktura i razvojni okvir Gin

Gin je razvojni okvir za razvoj web-aplikacija pisan u Go-u. Ključne značajke Gina su usmjerivač (eng. *router*) s nula alokacija, brzina, podrška za međuprograme *middleware*, bez rušenja servera na pojedinim zahtjevima, validacija JSON-a, grupiranje putanji (*URI*), upravljanje greškama, ugrađeno prikazivanje (*rendering*) te proširivost. Gin jednostavno

možemo dodati u naš projekt pomoću naredbe `go get -u github.com/gin-gonic/gin` u terminalu. Prije nego što prijeđemo na primjere, objasnimo nekoliko pojmova.

Međuprogram je softver (u slučaju Gina, to je funkcija u Go-u) koji rukuje sa zahtjevima koji dolaze na server. Međuprogrami obično imaju jednu zadaću koju moraju napraviti na svakom zahtjevu i ovisno o tome proslijediti zahtjev sljedećem međuprogramu u nizu. Te zadaće mogu biti prije nego što zahtjev dođe do samog servera (odnosno do konkretne funkcije za obradu zahtjeva na danoj putanji) ili nakon što server obradi zahtjev, a prije nego li se vrati odgovor klijentu. Ako međuprogram odluči da zahtjev ne treba poslati dalje sljedećem međuprogramu (ili na server), može odgovoriti klijentu prikladnim odgovorom. Međuprogrami su zaduženi za zadaće poput logiranja, provjere ili dodavanja HTTP zaglavlja, autentifikacije, autorizacije i obrade grešaka. Iako smo sada objasnili međuprograme, primjer konkretnog međuprograma je na kraju ovog potpoglavlja (primjer 2.4), prije njega ćemo proći kroz neke standardnije primjere.

Router, u kontekstu razvojnih okvira za web aplikacije, označava softver (u slučaju Gina, to je klasa iz paketa Gin) u kojoj povezujemo HTTP metodu, web putanju i funkciju koja obrađuje poslane zahtjeve. Routeri također mogu imati i određene međuprograme koji obrađuju zahtjeve. Ti međuprogrami mogu djelovati na svim putanjama na serveru, ali Gin nam omogućava i da grupiramo određene putanje i samo na njih dodamo neki međuprogram. Česta primjena opisanog je kada imamo stranice za logiranje i registriranje korisnika koje trebaju biti dostupne svima, a sve ostale stranice možemo zaštititi međuprogramom koji provjerava da je korisnik autentificiran. Routere možemo i dodatno konfigurirati po potrebi, a među korisnijim mogućnostima je definiranje liste posrednika (*proxies*) kojima vjerujemo i čiji promet želimo prihvatiti. Posrednici mogu biti serveri ili neki drugi servisi poput *load balancera* koji stoje između klijenta i naše gin aplikacije. Nakon malo teorije, pogledajmo jednostavni primjer iz službene dokumentacija paketa [5] koji, ako je sve prethodno namješteno, možemo samo kopirati i pokrenuti s `go run <ime_datoteke>.go`.

```
1  package main
2
3  import (
4    "net/http"
5    "github.com/gin-gonic/gin"
6  )
7
8  func main() {
9    r := gin.Default()
10   r.GET("/ping", func(c *gin.Context) {
11     c.JSON(http.StatusOK, gin.H{
12       "message": "pong",
13     })
14   })
```

```
  r.Run() // listen and serve on 0.0.0.0:8080
}
```

Primjer 2.1: Jednostavni Gin server

Funkcija `gin.Default` vraća pokazivač na klasu `gin.Engine` koja je zapravo router kakav smo prethodno opisali. Prvi parametar funkcije `r.GET` je putanja, a drugi je funkcija koja obrađuje zahtjeve koji dođu na server za danu putanju. `*gin.Context` je pokazivač na klasu koja je najvažniji dio razvojnog okvira Gin. Neke od stvari koje kontekst omogućava su prosljeđivanje varijabli između međuprograma, upravljanje tijekom zahtjeva, validiranje JSON-a u zahtjevima i prikazivanje JSON odgovora.

U gornjem primjeru koristimo `gin.Context` za prikazivanje i slanje JSON odgovora klijentskoj aplikaciji. Preciznije, koristimo funkciju `JSON` definiranu na strukturi `gin.Context`. Ona ima prima dva argumenta od kojih je prvi HTTP status, a drugi varijabla tipa `map[string]interface{}`. Struktura `gin.H` predstavlja skraćeni zapis za `map[string]interface{}` koji olakšava pisanje JSON-a u Go-u. `r.Run()` pokreće server. Slijedi primjer koji je skraćen i malo izmijenjen kako ne bismo ulazili u (trenutno) nebitne detalje i imali još više koda u primjeru. Također, ovo je samo fragment koda u kojem nisu svi objekti i funkcije koji se spominju definirani, a inspiriran je datotekom *server.go* iz praktičnog dijela diplomskog rada.

```
package api

import "github.com/gin-gonic/gin"

router := gin.Default()
router.SetTrustedProxies([]string{"http://localhost:5173"})

router.POST("/register", createUser)
router.POST("/tokens/authenticate", authUser)

authGroup := router.Group("/").Use(authMiddleware(*tokenMaker))

authGroup.GET("/user", getUser)
authGroup.DELETE("/user", deleteUser)
authGroup.PUT("/user", updateUser)
```

Primjer 2.2: Jednostavni Gin server

U gornjem primjeru smo postavili posrednički server kojem vjerujemo s metodom *SetTrustedProxies* kojoj smo proslijedili web adresu frontend aplikacije o kojoj ćemo više reći u sljedećem poglavlju. Putanje unutar naše gin aplikacije koje bi trebale biti dostupne svima neovisno o tome jesu li registrirani korisnici ili ne, nisu dodatno zaštićene i dostupne su svima s interneta. Većina putanja (ovdje nisu prikazane sve) je ipak dodatno zaštićeno i radimo provjeru koristeći *authMiddleware* kojem proslijedimo pokazivač na strukturu

tipa *JWTMaker* koje će vršiti autentifikaciju. *JWTMaker* je struktura koja je zadužena za kreiranje i validiranje tokena za autentifikaciju korisnika i o njoj ćemo govoriti u trećem potpoglavlju. Te putanje i HTTP glagole grupiramo koristeći *authGroup* varijablu, dok smo prijašnje nezaštićene putanje direktno definirali na varijabli *router*. Kao i u prethodnom primjeru, za svaki par HTTP glagola i putanje definiramo funkciju koja će obraditi zahtjev i vratiti prikladan odgovor. Slijedi primjer jedne takve funkcije.

```
func getUser(ctx *gin.Context) {
  userId := ctx.MustGet("authorization_payload").(*token.Payload).UserId
  user, err := (*sqlc.Queries).GetUser(ctx, userId)

  if err != nil {
    log.Println("ERROR: getUser userId=", userId, " err=", err.Error())
    if err == sql.ErrNoRows {
      ctx.JSON(http.StatusNotFound, "User does not exist")
      return
    }
    ctx.JSON(http.StatusInternalServerError, "Something went wron on our
     end. Please try again later.")
    return
  }
  ctx.JSON(http.StatusOK, user)
}
```

Primjer 2.3: Funkcija za dohvaćanje podataka o korisniku

Funkcija *getUser* se koristi u prethodnom primjeru gdje se mapira na HTTP glagol GET i putanju "/user". Funkcija prima pokazivač na *gin.Context* što je struktura koja nam omogućava neke od najvažnijih stvari unutar gin paketa poput prenošenja varijabli između međuprograma, upravlja tijekom zahtjeva kroz međuprograme, validira JSON zahtjeva te prikazuje (*renders*) JSON odgovor. *ctx.MustGet* vraća vrijednost za dani ključ iz mape koja je definirana unutar strukture *gin.Context*. Tu vrijednost smo prethodno spremili u mapu kada smo autentificirali korisnički zahtjev koristeći međuprogram za autentifikaciju. Dobiveni *userId* i kontekst prosljeđujemo funkciji iz paketa *sqlc* o kojem ćemo diskutirati u sljedećem potpoglavlju. Ona nam vraća varijablu koja predstavlja korisnika aplikacije i varijablu *err* s kojom provjeravamo je li se dogodila greška prilikom izvođenja upita na bazi. Dalje, ovisno o tome je li se dogodila greška ili ne, vraćamo prikladnu poruku ili objekt predstavljen varijablom *user*. Opet koristimo varijablu *ctx* koja prikazuje podatke u JSON formatu i šalje ih natrag korisniku, odnosno klijentskoj aplikaciji.

Sada slijedi primjer međuprograma za autentifikaciju korisnika koji je skoro identičan datoteci *middleware.go* iz praktičnog dijela diplomskog rada. U ovom primjeru spominjemo JWT token iako ga još nismo potpuno definirali, a o njemu ćemo puno više govoriti u trećem potpoglavlju.

```
package api
```

```
2
3  import (
4    "errors"
5    "fmt"
6    "net/http"
7    "strings"
8    "github.com/mhalavanja/go-rest-api/token"
9    "github.com/gin-gonic/gin"
10 )
11
12 const authPayload = "authorization_payload"
13
14 func authMiddleware(tokenMaker token.JWTMaker) gin.HandlerFunc {
15   return func(ctx *gin.Context) {
16     authorizationHeader := ctx.GetHeader("authorization")
17
18     if len(authorizationHeader) == 0 {
19       err := errors.New("authorization header is not provided")
20       ctx.AbortWithStatusJSON(http.StatusUnauthorized, err))
21       return
22     }
23
24     fields := strings.Fields(authorizationHeader)
25     if len(fields) < 2 {
26       err := errors.New("invalid authorization header format")
27       ctx.AbortWithStatusJSON(http.StatusUnauthorized, err))
28       return
29     }
30
31     authorizationType := strings.ToLower(fields[0])
32     if authorizationType != "bearer" {
33       err := fmt.Errorf("unsupported authorization type %s",
     authorizationType)
34       ctx.AbortWithStatusJSON(http.StatusUnauthorized, err))
35       return
36     }
37
38     accessToken := fields[1]
39     payload, err := tokenMaker.VerifyToken(accessToken)
40     if err != nil {
41       ctx.AbortWithStatusJSON(http.StatusUnauthorized, err))
42       return
43     }
44
45     ctx.Set(authPayload, payload)
46     ctx.Next()
47   }
```

```
48 }
```

Primjer 2.4: Međuprogram za autentifikaciju

Za autentifikaciju koristimo standardno zaglavlje *authorization* HTTP zahtjeva. Konkretno, koristimo *bearer* autentifikaciju, nekad zvanu i token autentifikaciju jer se vlasnik ili nosač (*bearer*) tokena treba autentificirati. Token se dobije kada se korisnik ulogira u aplikaciju što ćemo vidjeti u sljedećem primjeru. Nakon toga, prilikom svakog zahtjeva na server, šalje se i token (koji je zapravo samo *string*) unutar HTTP zaglavlja *authorization*.

Ako je veličina zaglavlja jednaka 0, odnosno ako zaglavlja nema, prekidamo lanac međuprograma i šaljemo klijentskoj aplikaciji prikladnu poruku i prikladni HTTP status. Nadalje, podijelimo *string* po bjelinama. Ako tako dobijemo manje od dva *stringa*, prekidamo lanac međuprograma i šaljemo klijentskoj aplikaciji prikladnu poruku. Ako prva riječ nije *bearer* opet vraćamo klijentu poruku i prekidamo lanac međuprograma jer se ne koristi pravilan način autentifikacije. Druga riječ mora biti token čiju valjanost onda provjerimo. Ako je token valjan, dodamo teret (običan *string*) JWT tokena u mapu unutar strukture *gin.Context* i pozivamo metodu *Next* koja je odgovorna za nastavljanje lanca međuprograma. Ako token nije valjan, lanac međuprograma završava i vraćamo prikladni odgovor klijentu.

Sada slijedi primjer razdvojen u dva dijela radi preglednosti. Prvo ćemo prikazati potrebne strukture, a zatim i funkciju koja se koristi kod prijave korisnika u aplikaciju. To se dakle događa dok korisnik još nema JWT token, ali se registrirao, odnosno napravio je korisnički račun. Znači njegovo korisničko ime i hash lozinke su prethodno spremljeni u bazu podataka. Kada se korisnik ulogira u aplikaciju dobije JWT token i za vrijeme trajanja tokena, ne može se ponovno ulogirati u aplikaciju.

```
1  package api
2
3  import (
4    "database/sql"
5    "net/http"
6    "time"
7
8    "github.com/gin-gonic/gin"
9    "github.com/mhalavanja/go-rest-api/consts"
10   "golang.org/x/crypto/bcrypt"
11 )
12
13 type authUserRequest struct {
14   Username string `json:"username" binding:"required"`
15   Password string `json:"password" binding:"required"`
16 }
17
18 type authUserResponse struct {
19   AccessToken          string    `json:"access_token"`
```

```
  AccessTokenExpiresAt time.Time `json:"access_token_expires_at"`
  UserID               int64     `json:"user_id"`
}
```

Primjer 2.5: Strukture korištene kod prijave korisnika

U gornjem primjeru vidimo pakete koje ćemo koristiti i dvije strukture. Struktura *authUserRequest* predstavlja podatke koje korisnik mora unijeti da bi se ulogirao, a *authUserResponse* predstavlja podatke koje server vraća klijentskoj aplikaciji.

```
func (server *Server) authUser(ctx *gin.Context) {
  var req authUserRequest
  if err := ctx.ShouldBindJSON(&req); err != nil {
    ctx.JSON(http.StatusBadRequest, "You have to provide username and password")
    return
  }

  user, err := server.store.GetUserByUsername(ctx, req.Username)
  if err != nil {
    if err == sql.ErrNoRows {
      ctx.JSON(http.StatusUnauthorized, "Wrong username or password")
      return
    }
    ctx.JSON(http.StatusInternalServerError, "Something went wrong on our end, please try again")
    return
  }

  err = bcrypt.CompareHashAndPassword([]byte(user.HashedPassword), []byte(req.Password))
  if err != nil {
    ctx.JSON(http.StatusUnauthorized, "Wrong username or password")
    return
  }

  accessToken, accessPayload, err := server.tokenMaker.CreateToken(
    user.ID,
    server.config.AccessTokenDuration,
  )
  if err != nil {
    ctx.JSON(http.StatusInternalServerError, "Something went wrong on our end, please try again")
    return
  }

  rsp := authUserResponse{
    AccessToken:          accessToken,
```

```
35      AccessTokenExpiresAt: accessPayload.ExpiredAt,
36      UserID:               user.ID,
37    }
38
39    ctx.JSON(http.StatusOK, rsp)
40  }
```

Primjer 2.6: Funkcija za prijavu korisnika

U gornjem primjeru prvo provjerimo JSON koji nam pošalje klijentska aplikacija u zahtjevu. Ako unutar JSON-a nemamo polja *username* i *password* vraćamo korisniku poruku i HTTP status 404. Dalje pokušamo dohvatiti korisničko ime i hash lozinke iz baze, koristeći korisničko ime koje smo dobili od korisnika. Ako ne postoji takav zapis unutar baze podataka, vraćamo HTTP status neovlašteno (*unauthorized*), ali iz sigurnosnih razloga ne želimo vratiti poruku u kojoj piše da korisničko ime nije ispravno, iako znamo da nije. Namjerno nismo precizni kako bi se otežali bilo kakvi pokušaju *brute force* napada na našu aplikaciju. Ako je upit na bazu bio uspješan, znači da korisnik s danim korisničkim imenom stvarno postoji. Na danu korisničku lozinku primijenimo hash funkciju i usporedimo s hashem lozinke koji smo dobili iz baze. Ako se ne podudaraju, to znači da je korisnik unio krivu lozinku. Opet, iako znamo da je problem u lozinki, klijentskoj aplikaciji ćemo samo vratiti HTTP status neovlašteno i poruku o krivom korisničkom imenu ili lozinki. Napokon, ako su korisničko ime i lozinka korektni, korisniku vraćamo JWT token i efektivno je prijavljen u aplikaciju. Jedino se još može dogoditi neka greška tijekom kreiranja tokena, a u tom slučaju obavještavamo klijentsku aplikaciju o tome da nešto nije u redu s aplikacijom.

2.2 Model, poslovna logika i sqlc paket

Model (u ovom kontekstu) se odnosi na objekte, često iz stvarnog svijeta, koje prikazujemo kao klase unutar naše aplikacije. Njihovo stanje mijenjamo i pohranjujemo u bazu podataka kako ga ne bismo izgubili gašenjem servera na kojem se izvodi program. Kreiranje, čitanje, ažuriranje i brisanje (*CRUD* - Create, Read, Update, Delete) tih objekata upravljano je poslovnom logikom naše aplikacije. Poslovna logika predstavlja specifične zahtjeve o ponašanju objekata koji se onda implementiraju u aplikaciji. Pošto koristimo bazu podataka, moramo izabrati između SQL i NoSQL baze podataka. U ovom diplomskom radu, odlučili smo se za SQL bazu podataka PostgreSQL, pošto je model (a samim time i tablice korištene u bazi) dobro definiran i ne trebaju nam mogućnosti horizontalnog skaliranja koje pružaju NoSQL baze podataka. Poslovna logika se može implementirati na *backendu* koristeći programski jezik koji i koristimo za *backend* ili unutar same baze koristeći SQL funkcije i procedure.

U današnje vrijeme za implementaciju poslovne logike i samih tablica unutar SQL baza podataka, često se koriste ORM (*Object-Raltional Mapping*) alati. Oni apstrahiraju bazu podataka i enkapsuliraju SQL tako da se svi upiti na bazu mogu pisati u nativnom programskom jeziku u kojem je pisan i ostatak *backend* aplikacije. ORM-ovi (ORM alati, paketi) uvelike ubrzavaju vrijeme razvoja aplikacije i pridonose održivosti koda. Iz tih razloga, vrlo su popularni u velikim firmama koje razvijaju velike (*enterprise*) projekte. Osim već spomenutih, postoje i drugi razlozi zašto bi se koristili ORM-ovi poput pripremljenih izraza (*prepared statements*), vrlo laganog korištenja transakcija, automatskog upravljanja bazom podataka te internacionalizacije i lokalizacije podataka. Zadnji vrlo važan razlog za korištenje ORM-ova je taj što programeri često ne vole, ne žele ili jednostavno ne znaju pisati dobar SQL kod.

Sad nakon što smo objasnili što su i naveli dobre strane ORM-ova, možemo spomenuti i neke lošije strane i neke alternative. Kod kompleksnih upita, generirani SQL ne mora biti optimalan i može biti usko grlo ako je brzina izvođenja aplikacije vrlo bitna. ORM-ovi zahtijevaju dodatno učenje i razumijevanje te ako nismo zadovoljni brzinom izvođenja upita koju dobivamo koristeći ORM, moramo ili pisati ručno SQL ili učiti o samim unutarnjim komponentama ORM-a.

Alternativa pisanju SQL-a unutar *backend* programskog jezika, izravna manipulacija sa svakim upitom te mapiranje na objekte je prilično loša i stoga koristimo treći pristup koristeći paket *https://github.com/kyleconroy/sqlc* (dokumentacija dostupna na [2]) u Go-u, koji ćemo uskoro opisati, u kombinaciji sa SQL funkcijama i procedurama. SQL funkcije i procedure koristimo kada je riječ o kompliciranijim upitima, a za jednostavnije upite koristimo samo sqlc. Kad kažemo kompliciraniji upiti, mislimo na upite koji zahtijevaju neku posebnu poslovnu logiku za koju nam trebaju dodatne varijable i moraju se izvršavati unutar transakcije. Iako i sami sqlc paket ima odličnu potporu za izvođenje transakcija unutar Go koda, mora slati više od jednog upita na bazu (kao i ostali ORM-ovi). Tome doskačemo tako da koristimo transakcije izravno na bazi unutar SQL funkcija i procedura.

sqlc paket služi za generiranje sigurnog u smislu tipa podataka (*type-safe*), idiomatskog Go koda iz SQL-a. To znači da umjesto ORM pristupa gdje koristimo programski jezik u kojem pišemo *backend* za generiranje SQL-a i upravljanje entitetima u bazi, radimo suprotno. Koristimo SQL i sqlc kako bismo generirali objekte našeg modela, funkcije za mapiranje dobivenih podataka iz upita na objekte te funkcije koje pozivamo u našem Go kodu kako bismo izvršili dani SQL upit i dobili podatke nazad. sqlc koristimo kao paket koji uvozimo u naše datoteke, ali kao i program u terminalu jer s naredbom `sqlc generate` generiramo traženi go kod.

Za početak, sqlc moramo instalirati na računalo. Jedan mogući način je naredbom `go install github.com/kyleconroy/sqlc/cmd/sqlc@latest`. sqlc ima podršku za nekoliko baza podataka, a mi ćemo se nadalje fokusirati isključivo na PostgreSQL. sqlc koristi *sqlc.yaml* ili *sqlc.json* datoteke za konfiguriranje koje moraju biti unutar direk-

torija u kojem pokrećemo naredbu `sqlc make`. Do kraja ovog potpoglavlja imat ćemo niz primjera koji su vezani jedan uz drugi, odnosno jedan veliki primjer koji se može naći u službenoj dokumentaciji paketa, preciznije *https://docs.sqlc.dev/en/latest/tutorials/getting-started-postgresql.html*.

```
version: "2"
sql:
  - engine: "postgresql"
    queries: "query.sql"
    schema: "schema.sql"
    gen:
      go:
        package: "tutorial"
        out: "tutorial"
```

Primjer 2.7: Jednostavna datoteka sqlc.yaml

Gornji primjer prikazuje jednostavnu datoteku *sqlc.yaml* u kojoj su definirane najvažnije stvari potrebne za generiranje Go koda. `version` predstavlja verziju same konfiguracije koju koristimo. Za vrijeme pisanja, verzija 2 je najnovija. `engine` predstavlja bazu podataka koju koristimo. `queries` je putanja i ime SQL datoteke u kojoj su upiti, a `schema` je putanja i ime datoteke u kojoj se nalazi shema naše baze. Za ove dvije varijable možemo specificirati i cijeli direktorij, a ne samo jednu datoteku. Direktorij bismo naveli ako nam treba više datoteka za upite odnosno za shemu. Za upite postoji jednostavan razlog zašto bismo htjeli imati više datoteka, a to je da odvojimo kod u neke logične cjeline. Za shemu možda nije odmah jasno zašto bismo koristili više datoteka, ali postoji dobar razlog. U direktorij možemo smjestiti migracijske datoteke koje se koriste kod rekreiranja baze podataka i vraćanja baze podataka na točno određeno stanje u prošlosti. Tako nešto bi bilo potrebno ako shvatimo da imamo neki kritični *bug* unutar aplikacije koji smo unijeli najnovijom verzijom aplikacije te stoga trebamo vratiti bazu i aplikaciju na prijašnju verziju, odnosno u prijašnje stanje. `package` je ime paketa koji će se koristiti za generirani Go kod. `out` je putanja i ime direktorija koji će se koristiti za generirani Go kod. Još se mnogo stvari može regulirati pomoću konfiguracijske datoteke *sqlc.yaml*, a za više o samoj konfiguraciji pogledajte službenu dokumentaciju *https://docs.sqlc.dev/en/latest/reference/config.html*.

Nakon što postavimo konfiguracijsku datoteku moramo napisati shemu baze. U to nećemo previše ulaziti u ovom poglavlju jer je vrlo specifično za svaku aplikaciju, a sqlc koristi shemu kako bi mogao generirati Go strukture i kako bi znao tipove podataka njihovih članskih varijabli.

Sljedeći korak je zapravo glavni dio cijelog paketa. U datoteci ili direktoriju specificiranom varijablom *schema* moramo napisati upite koji će se izvršavati i koji će služiti za

generiranje Go koda. Prije nego što damo primjer SQL upita, prikazat ćemo jednostavnu shemu baze podataka na koju će se odnositi upiti, a ona se sastoji od samo jedne tablice.

```
CREATE TABLE authors (
  id   BIGSERIAL PRIMARY KEY,
  name text      NOT NULL,
  bio  text
);
```

Primjer 2.8: Sadržaj datoteke shema.sql

```
-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1 LIMIT 1;

-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;

-- name: CreateAuthor :one
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
RETURNING *;

-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1;
```

Primjer 2.9: SQL upiti uz sqlc paket

sqlc zahtijeva da uz svaki upit navedemo ime funkcije čiji će Go kod sqlc generirati i čijim ćemo pozivom izvršiti upit. Nakon što specificiramo ime, sqlc treba malu pomoć. Trebamo specificirati je li upit vraća jedan podatak (*:one*), više podataka (*:many*) ili smo izvršava upit i ništa ne vraća (*:exec*). Nakon toga, možemo pisati upit u kojem koristimo *$n*, gdje je *n* neki prirodni broj, za oznaku *n*-tog parametra funkcije koja poziva dani upit. Na ovo ćemo se još vratiti i razjasniti stvari u sljedećem odlomku. Kada napišemo upite, koristimo naredbu `sqlc generate` kako bismo generirali Go kod iz danih datoteka. Generirani kod ne bismo trebali mijenjati jer ćemo izgubiti vlastite promjene sljedeći put kada pokrenemo naredbu `sqlc generate`. Ako se nađemo u potrebi da mijenjamo generirane datoteke, to je znak da vrlo vjerojatno postoji bolji i lakši način da ostvarimo željeni cilj.

Među generiranim datotekama bit će datoteka *db.go*, a u njoj pri kraju datoteke će biti definiran novi tip podatka: struktura imena *Queries*. Ta struktura sadrži pokazivače na sve generirane funkcije koje ćemo pozivati iz vlastitog koda. Dakle, uz prethodno definirane

datoteke *sqlc.yaml* i *queries.sql*, nakon generiranja Go koda, možemo pozvati funkciju `(*tutorial.Queries).GetAuthor` unutar neke druge datoteke, kao i ostale funkcije generirane iz naših SQL upita.

Osim datoteke *db.go*, za svaku SQL datoteku u koju smo pisali upite ćemo dobiti istoimenu Go datoteku s generiranim funkcijama i strukturama za upite definirane u danoj datoteci. Slijedi primjer dijela datoteke koji bi bio generiran iz datoteke *query.sql* definirane u prethodnom primjeru. Generirana datoteka bi se stvarno zvala *query.sql.go*, a ne samo *query.go*. Primjer prikazuje samo početak datoteke i generirane objekte potrebne samo za upit *CreateAuthor*. Za svaki drugi upit u datoteci *query.sql* bi se generirali slični objekti.

```
1  package tutorial
2
3  import (
4    "context"
5    "database/sql"
6  )
7
8  const createAuthor = `-- name: CreateAuthor :one
9  INSERT INTO authors (
10   name, bio
11 ) VALUES (
12   $1, $2
13 )
14 RETURNING id, name, bio
15 `
16
17 type CreateAuthorParams struct {
18   Name string
19   Bio  sql.NullString
20 }
21
22 func (q *Queries) CreateAuthor(ctx context.Context, arg
       CreateAuthorParams) (Author, error) {
23   row := q.db.QueryRowContext(ctx, createAuthor, arg.Name, arg.Bio)
24   var i Author
25   err := row.Scan(&i.ID, &i.Name, &i.Bio)
26   return i, err
27 }
```

Primjer 2.10: Dio sadržaja datoteke query.sql.go

Nećemo ulaziti u kod same funkcije *CreateAuthor*, ali moramo nešto reći o prvom ulaznom argumentu čije značenje nije sasvim očito. Varijabla *ctx* tipa *Context* iz paketa *context* predstavlja kontekst koji je trajanjem vezan uz jedan zahtjev (*request scoped*) te se prenosi između različitih procesa i API-ja. Koncept ćemo još vidjeti i u sljedećem potpoglavlju

gdje se kontekst u paketu *gin* koristi za istu zadaću, ali pruža i dodatne mogućnosti koje ćemo spomenuti. Kontekst ćemo dakle ili napraviti novi ili (što je vjerojatnije) dobiti iz funkcije koja će pozivati funkciju *CreateAuthor*.

Osim generiranih funkcija koje koristimo u kodu generiranih iz upita, također se generira datoteka *models.go* u kojoj se nalaze strukture koje predstavljaju tablice iz naše baze podataka. Te strukture se koriste u generiranim funkcijama, ali ih mi možemo koristiti i neovisno u vlastitom kodu. Navodimo primjer generirane datoteke *models.go* iz za koju je korištena tablica iz prethodnog primjera.

```
1  package tutorial
2
3  import (
4    "database/sql"
5  )
6
7  type Author struct {
8    ID   int64
9    Name string
10   Bio  sql.NullString
11 }
```

Primjer 2.11: Sadržaj datoteke models.go

2.3 Json Web Token i autentifikacija korisnika

JSON Web Token (*JWT*) je otvoreni standard koji definira kompaktan i samostalan način za siguran prijenos informacija između dva sudionika korištenjem JSON objekata ([7]). Podaci poslani koristeći JWT se mogu provjeriti i može im se vjerovati jer su digitalno potpisani. JWT-ovi se mogu potpisati korištenjem tajnog niza znakova (s HMAC algoritmom) ili para javnih/privatnih ključeva pomoću RSA ili ECDSA algoritama. Zbog spomenutih svojstava, JWT se najčešće koriste za autentifikaciju korisnika i siguran prijenos informacija. JWT se sastoji od tri dijela: zaglavlje (*header*), teret (*payload*) i potpis (*signature*). Ti dijelovi se spajaju u jedan *string*, a unutar njega se odvajaju točkom. Zaglavlje sadrži informacije o algoritmu korištenom u kriptografskom potpisu i tip tokena koji je JWT. Zaglavlje je pisano JSON formatom, ali se prije daljnjeg korištenja kodira Base64Url kodiranjem. Kada koristimo JWT tokene, moramo imati na umu da se oni zapravo vrlo lako mogu dekodirati i stoga ih uvijek trebamo koristiti preko enkriptirane mreže (koristeći TLS) te ne bismo trebali stavljati nikakve povjerljive podatke poput korisničke lozinke.

```
1  {
2    "alg": "HS256",
3    "typ": "JWT"
```

```
}
```

Primjer 2.12: JWT zaglavlje

Teret sadrži tvrdnje (*claims*) koje se šalju, a to su najčešće informacije o korisniku i njegovim ovlastima. Kao i zaglavlje, pisano je u JSON formatu, ali se kodira s Base64Url kodiranjem prije daljnjeg korištenja.

```
{
  "id": 123,
  "user_id": "abc123",
  "issued_at": 10000 ,
  "expired_at": 20000
}
```

Primjer 2.13: JWT teret

Treći dio je potpis i upravo on osigurava da zaglavlje i teret JWT tokena nisu promijenjeni na neželjen način. Da bismo napravili potpis moramo uzeti kodirano zaglavlje, kodirani teret, tajni niz znakova i algoritam specificiran u zaglavlju te napraviti potpis. Na primjer, da bismo napravili potpis koristeći HMAC SHA256 algoritam, morali bismo (u nekom programskom jeziku) napraviti sljedeće:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Primjer 2.14: Generiranje JWT potpisa koristeći HMAC SHA256 algoritam

Dalje ćemo prikazati kako se koristi JWT u Go-u pomoću paketa *github.com/golang-jwt/jwt*. Sljedećih nekoliko primjera je zapravo jedan veliki primjer, ali ovako ćemo se lakše fokusirati na pojedine dijelove i nećemo imati jednu cijelu, zastrašujuću stranicu punu koda. Prvo su definirane strukture i varijable korištene u donjim primjerima gdje opisujemo funkcije za kreiranje i validiranje JWT tokena.

```
package jwtToken

import (
  "time"
  "errors"
  "log"
  "github.com/golang-jwt/jwt"
  "github.com/google/uuid"
)
type Payload struct {
  ID        uuid.UUID `json:"id"`
```

```
  UserId    int64     `json:"userId"`
  IssuedAt  time.Time `json:"issued_at"`
  ExpiredAt time.Time `json:"expired_at"`
}

type JWTMaker struct {
  secretKey string
}

var (
  ErrInvalidToken = errors.New("token is invalid")
  ErrExpiredToken = errors.New("token has expired")
)
```

Primjer 2.15: Strukture korištene za kreiranje JWT tokena

U gornjem primjeru koristimo univerzalno jedinstvene identifikatore *universally unique identifier, skraćeno UUID* kako bi svaki teret JWT tokena mogli jedinstveno identificirati. Tako nešto bi bilo posebno korisno kada bismo logirali neke podatke ili trebali debugirati kod. Vidimo da struktura `JWTMaker` ima samo jednu člansku varijablu, no ipak smo je definirali upravo tako. Želimo da nam kod bude logički povezan tako da su sljedeće funkcije definirane kao metode na toj strukturi. Još jedan dobar razlog za ovako organiziran kod je to što ako bi nam u buduće trebala neka dodatna informacija, ne moramo mijenjati potpise funkcija i njihove pozive, već možemo dodati novu člansku varijablu u strukturu i jednostavno je koristiti u funkcijama. Na kraju smo još definirali dvije varijable koje predstavljaju moguće greške tijekom validiranja tokena.

```
func (maker *JWTMaker) CreateToken(userId int64, duration time.Duration)
    (string, *Payload, error) {
  tokenId, err := uuid.NewRandom()
  if err != nil {
    log.Println(err)
    return nil, err
  }

  payload := &Payload{
    ID:        tokenId,
    UserId:    userId,
    IssuedAt:  time.Now(),
    ExpiredAt: time.Now().Add(duration),
  }

  jwtToken := jwt.NewWithClaims(jwt.SigningMethodHS256, payload)
  token, err := jwtToken.SignedString([]byte(maker.secretKey))
  return token, payload, err
```

```
}
```

Primjer 2.16: Funkcija za kreiranje JWT tokena

Gornju funkciju `CreateToken` koristimo za stvaranje novog JWT tokena tako što napravimo novu instancu tereta JWT tokena, potpišemo token i vratimo potrebne informacije pozivatelju.

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

Primjer 2.17: Funkcija koja vraća tajni niz znakova i provjerava algoritam

Funkcija `keyFunc` vraća tajni niz znakova koji smo definirali prethodno i koristili kod kodiranja zaglavlja i tereta tokena. Primijetimo da iz sigurnosnih razloga moramo eksplicitno provjeriti algoritam korišten kod potpisa tokena, to nije automatski napravljeno za nas. Iako izgleda benigno, propuštanje ovog koraka može dovesti do vrlo jednostavnih, sigurnosnih napada u kojima zlonamjerni akter može promijeniti algoritam koji smo koristili pri enkripciji.

```
func (maker *JWTMaker) VerifyToken(token string) (*Payload, error) {
  jwtToken, err := jwt.ParseWithClaims(token, &Payload{}, keyFunc)
  if err != nil {
    vErr, ok := err.(*jwt.ValidationError)
    if ok && errors.Is(vErr.Inner, ErrExpiredToken) {
      return nil, ErrExpiredToken
    }
    return nil, ErrInvalidToken
  }

  payload, ok := jwtToken.Claims.(*Payload)
  if !ok {
      return nil, ErrInvalidToken
  }

  return payload, nil
}
```

Primjer 2.18: Funkcija za verificiranje JWT tokena

Zadnja metoda služi za parsiranje i verificiranje tokena te vraćanje tokena ili prikladne greške ako token nije ispravan. Primijetimo da se ovdje koristi prethodno definirana funkcija `keyFunc`.

Ovako definiranu strukturu `JWTMaker` možemo dakle koristiti za kreiranje novih tokena kada se korisnici logiraju u aplikaciju te za verifikaciju tokena pri svakom sljedećem korisnikovom zahtjevu prema zaštićenim resursima sa servera. To je naravno puno sigurnije i brže nego da korisnik u svakom zahtjevu šalje korisničko ime i lozinku.

2.4 Konfiguracija i okolina aplikacije

Iako je ova tema kraća od prethodnih, volio bih reći nešto o njoj i pritom spomenuti Go paket koji sam koristio. Kada se proizvodi aplikacija na industrijskoj razini, ona se mora izvoditi u različitim okolinama. Okolinu čine razne okolinske varijable, slično okolinskim varijablama koje imamo i na kompjuteru te različite postavke koje na neki način utječu na rad aplikacije. Okoline mogu biti razne, a neke od njih su lokalno računalo na kojem pojedinac razvija aplikaciju, razvojno okruženje kojem mnogo programera ima pristup, testno okruženje u kojem QA inženjeri testiraju aplikaciju te produkcijsko okruženje. Također, takve aplikacije su često komplicirane i zahtijevaju pristup određenim resursima poput baza podataka, servisa, servera i API-ja unutar i izvan naše firme. Da bismo imali pristup takvim resursima često nam treba njihova IP adresa ili web adresa, username, lozinka i slični podaci.

Takve podatke bismo mogli direktno upisati unutar aplikacije kao nepromjenjive *stringove*, ali imamo problem s takvim pristupom. Naime, aplikacija se unutar svake okoline treba spojiti na različitu instancu baze (lokalna, testna, produkcijska), servisa ili API-ja, a na neke servere se možda uopće ne spaja za vrijeme razvoja već isključivo u produkciji. Iz tih razloga, takvi podaci za pristup resursima i još općenitije, svi podaci ili postavke koje su vezane uz okolinu u kojoj se aplikacija izvodi, se pohranjuju izvan samog izvornog koda aplikacije. Takvi podaci su zapisani u konfiguracijskim datotekama koje često imaju nastavak *.env* što dolazi od engleske riječi za okolinu (*environment*).

```
1 DB_DRIVER=postgres
2 DB_SOURCE=postgresql://root:secret@localhost:5432/diplomski?sslmode=
    disable
3 SERVER_ADDRESS=0.0.0.0:8080
4 TOKEN_SYMMETRIC_KEY=12345678901234567890123456789012
5 CLIENT_ADDRESS=http://localhost:5173
6 GIN_MODE=debug
```

Primjer 2.19: Sadržaj datoteke app.env

Gornji primjer je skoro identičan datoteci *app.env* koja se koristi u praktičnom dijelu ovog diplomskog rada. Kada bismo stvarno htjeli postaviti ovu aplikaciju u produkciju, morali bismo promijeniti neke od varijabli, a to se najlakše postiže tako da imamo više konfiguracijskih datoteka, jednu za svaku okolinu. Aplikaciju onda pokrećemo tako da joj

na neki način proslijedimo putanju do konkretne konfiguracijske datoteke, ovisno o okolini u kojoj se izvodi.

Sada kad imamo konfiguracijsku datoteku za svaku okolinu, moramo učitati te vrijednosti u našu aplikaciju, ne želimo stalno čitati iz datoteke.

```
1  package util
2
3  import "github.com/spf13/viper"
4
5  type Config struct {
6    DBDriver            string        `mapstructure:"DB_DRIVER"`
7    DBSource            string        `mapstructure:"DB_SOURCE"`
8    ServerAddress       string        `mapstructure:"SERVER_ADDRESS"`
9    TokenSymmetricKey   string        `mapstructure:"TOKEN_SYMMETRIC_KEY"`
10   Client              string        `mapstructure:"CLIENT_ADDRESS"`
11 }
12
13 func LoadConfig(path string) (config Config, err error) {
14   viper.AddConfigPath(path)
15   viper.SetConfigName("app")
16   viper.SetConfigType("env")
17
18   viper.AutomaticEnv()
19
20   err = viper.ReadInConfig()
21   if err != nil {
22     return
23   }
24
25   err = viper.Unmarshal(&config)
26   return
27 }
```

Primjer 2.20: Učitavanje podataka iz konfiguracijske datoteke

U gornjem primjeru je sav sadržaj datoteka iz praktičnog dijela diplomskog rada koja učitava sve konfiguracijske varijable iz prethodnog primjera. Za to koristimo paket *https://github.com/spf13/viper* o kojem možemo saznati sve potrebne informacije iz dostupne dokumentacije [3]. Bez da ulazimo u detalje, definiramo strukturu koja predstavlja konfiguracijske varijable i napišemo imena konfiguracijskih varijabli koja trebaju odgovarati onima unutar konfiguracijske datoteke. Nadalje, proslijedimo putanju, ime i tip konfiguracijske datoteke, parsiramo datoteku i učitamo vrijednosti u instancu *Config* strukture. Nakon toga, podaci su dostupni unutar aplikacije.

Poglavlje 3

Primjer web aplikacije

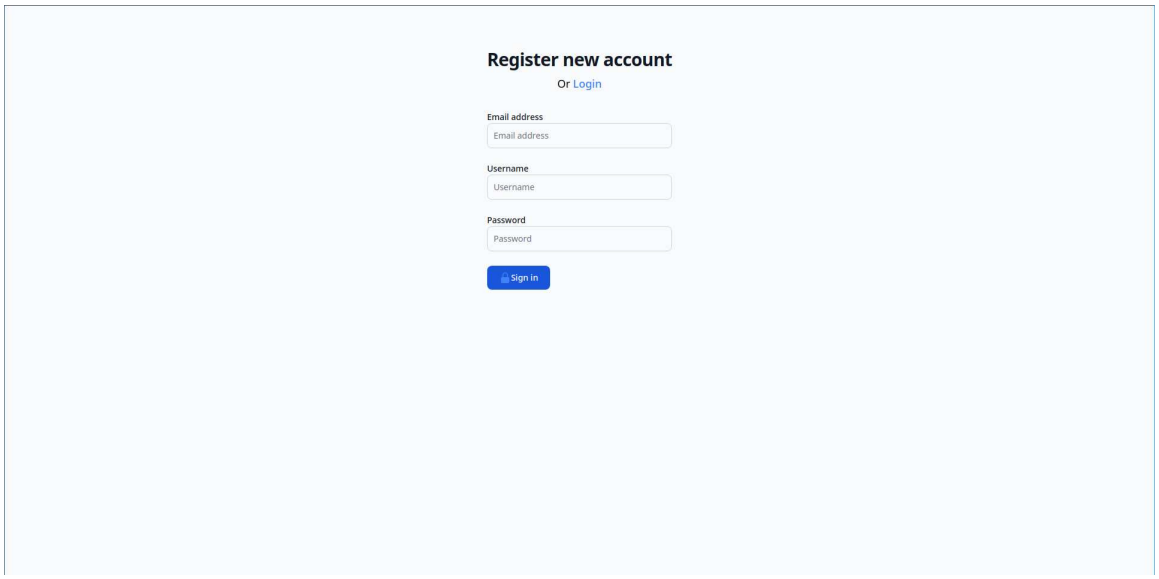
U posljednjem poglavlju ovog diplomskog rada govorit ćemo o još nekim konceptima koji se koriste u web aplikaciji implementiranoj u sklopu praktičnog dijela rada. Mnogi primjeri i isječci koda koje smo vidjeli u prošlom poglavlju su uzeti (ponekad uz minimalnu izmjenu radi čitljivosti) upravo iz praktičnog dijela diplomskog rada, tako da smo zapravo već dosta upoznati s dobrim dijelom aplikacije. Dijelovi aplikacije koje nismo spominjali do sad su ili specifični za ovu konkretnu aplikaciju ili nisu striktno vezani uz Go na kojem je fokus ovog diplomskog rada. U ovom poglavlju ćemo se fokusirati na još dva koncepta koja su korištena u aplikaciji, a to su sesija (*session*) i konkretna implementacija sesije te *WebSocket* server koji omogućuje slanje poruka unutar aplikacije. Više o tome u posljednja dva potpoglavlja, no za početak recimo nešto o samoj aplikaciji i tehnologijama koje su korištene za razvoj. Sav izvorni kod aplikacije je otvoren i može se naći na priloženom CD-u te na GitHubu. Kod serverske aplikacije nalazi se na [9], a kod klijentske aplikacije na [8].

3.1 Razvijena web aplikacija

Razvijena aplikacija se sastoji zapravo od dvije aplikacije, *backend* dijela na kojem je fokus ovog rada i *frontend* dijela o kojem ćemo samo reći par rečenica. Prvo, za *backend* važnu ulogu ima i *PostgreSQL* baza podataka. Dijelovi poslovne logike su implementirani unutar same baze koristeći *PostgreSQL* funkcije i procedure. *Frontend* aplikacija je pisana koristeći *TypeScript*, *Svelte* i *SvelteKit*. *Svelte* je razvojni okvir za razvoj korisničkih sučelja (eng. *user interface*, *UI*), na primjer poput *Reacta* ili *Vuea*, no za razliku od njih se ne oslanja na virtualni *Document Object Model* (*DOM*) već se kod pisan u *Svelteu* kompilira u nativni *JavaScript*. *SvelteKit* je razvojni okvir za razvoj modernih web aplikacija koji pruža različite načine razvoja cjelovitih, *full stack* aplikacija. Za samo stiliziranje elemenata je korišten *Tailwind CSS* što je CSS razvojni okvir koji se sastoji od velikog broja

pripremljenih CSS klasa.

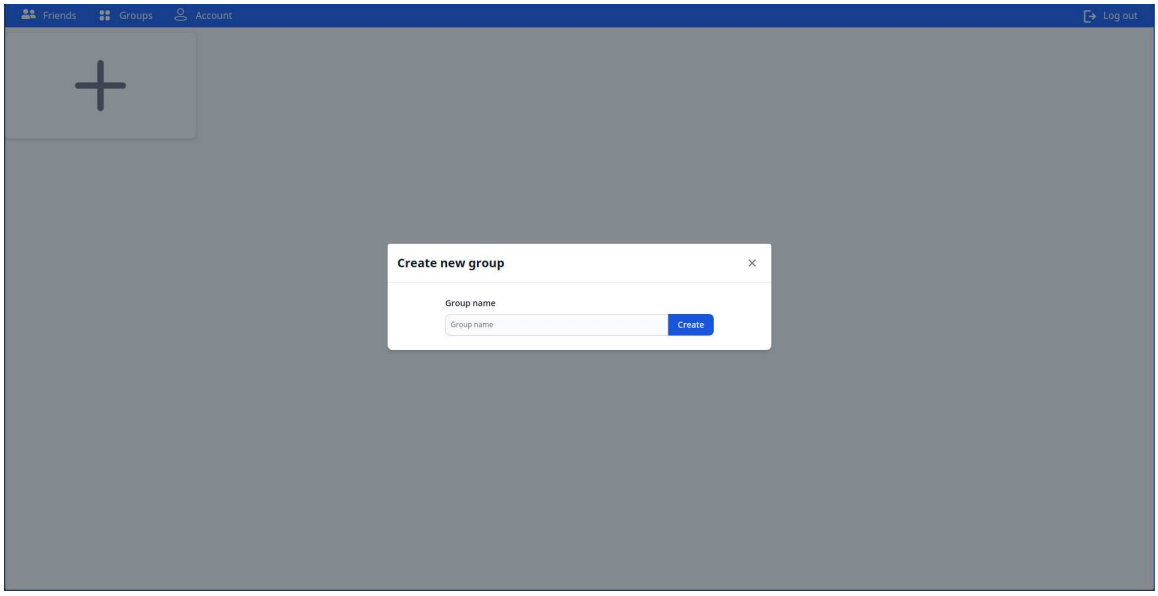
Aplikacija koja je razvijena u sklopu diplomskog rada je jednostavna *chat* aplikacija. Aplikacija se sastoji od korisnika i grupa, a glavni cilj aplikacije je omogućiti umrežavanje korisnika, komunikaciju unutar grupa te administraciju grupa. Ako želimo koristiti aplikaciju, prvo moramo napraviti korisnički račun koji se sastoji od korisničkog imena, emaila i lozinke. Na sljedećoj slici 3.1 vidimo kako izgleda stranica za registraciju.



Slika 3.1: Stranica za registraciju

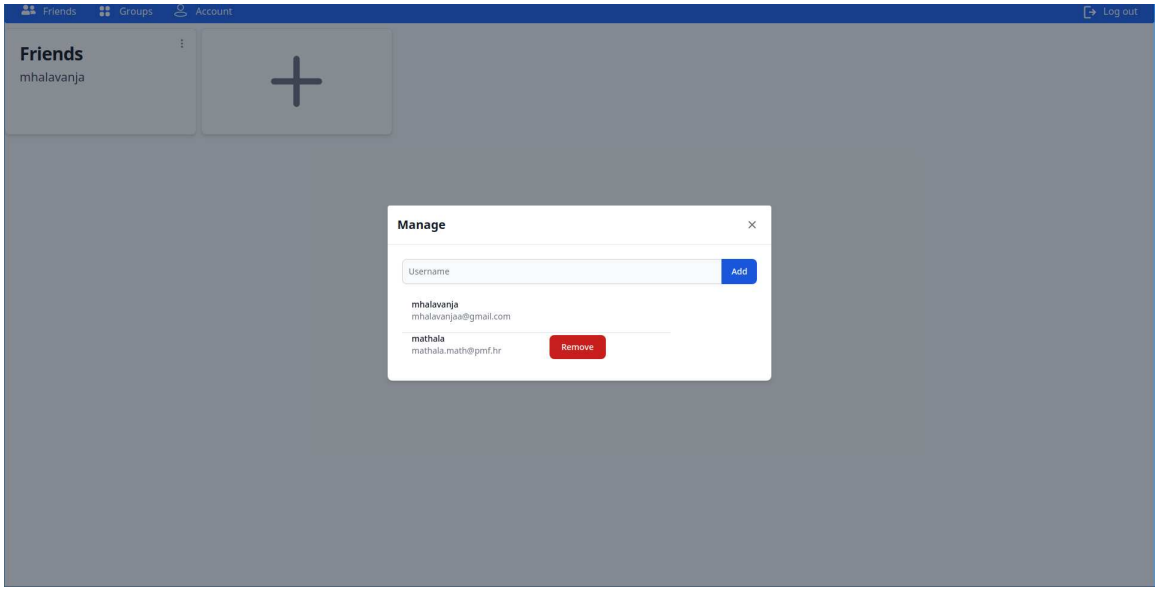
Nakon registracije smo odmah prijavljeni u aplikaciju, ne moramo ponovno unositi korisničko ime i lozinku. Ako pak već imamo korisnički račun napravljen, prijavit ćemo se u aplikaciju na stranici vrlo sličnoj za registraciju, bez polja za unos emaila. Podsjetimo se, nakon unosa korisničkog imena i lozinke, koristimo *JWT* tokene kako bi mogli autentificirati korisnika. Korisnik dobije *JWT* token nakon uspješne autentifikacije. Nakon prijave u aplikaciju, smješteni smo na stranicu za upravljanje grupama. Ovdje možemo vidjeti listu grupa čiji smo članovi. Svaka grupa predstavljena je jednom karticom. Na svakoj kartici piše ime grupe i ime vlasnika grupe. Na kraju liste grupa imamo jednu dodatnu karticu s oznakom plus, a klikom na tu karticu možemo napraviti novu grupu.

Ako smo kreirali grupu, onda smo vlasnik grupe. Na svakoj kartici imamo gumb za upravljanje grupom koje se zasniva na tome jesmo li ili nismo vlasnik te grupe. Ako smo vlasnik grupe možemo obrisati cijelu grupu i dodavati ili uklanjati članove grupe. Brisanjem grupe svi korisnici prestaju biti članovi grupe i svi podaci o grupi se brišu iz baze podataka. Ako pak nismo vlasnik grupe, možemo napustiti grupu i vidjeti koji su



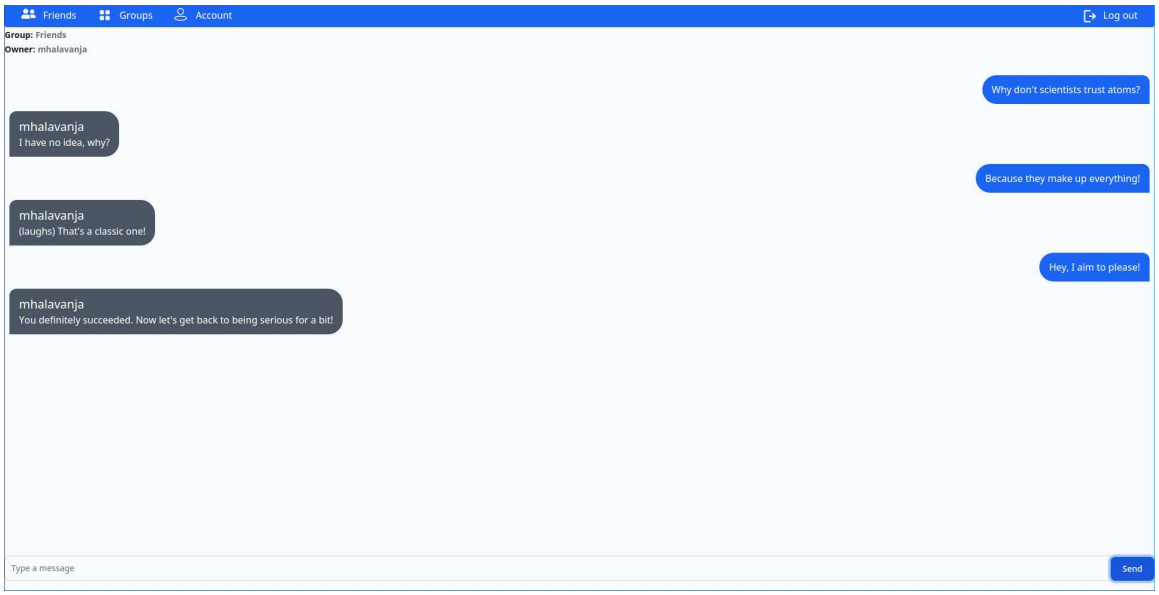
Slika 3.2: Modal za kreiranje nove grupe

članovi grupe. Na slici 3.3 vidimo kako izgleda modal za upravljanje grupom čiji smo vlasnik.



Slika 3.3: Modal za upravljanje grupom

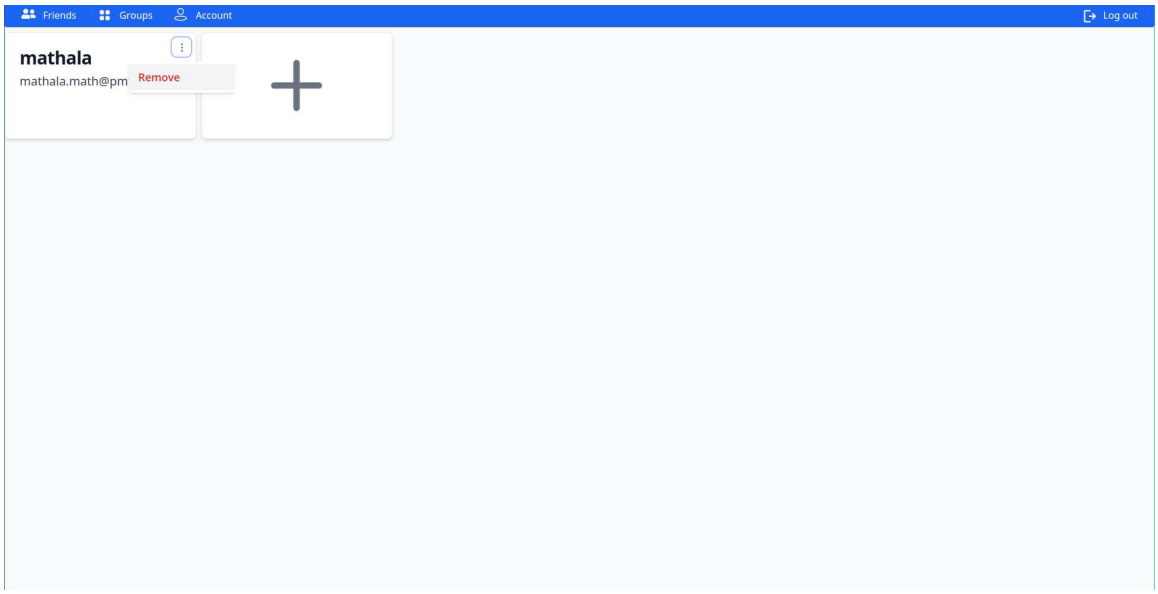
Ako nismo vlasnik grupe, modal izgleda slično, samo nemamo spomenute mogućnosti dodavanja i uklanjanja korisnika iz grupe. Kada kliknemo na karticu grupe, na vrhu ekrana vidimo kratke informacije o imenu grupe i vlasniku, a na dnu ekrana imamo HTML element za upisivanje poruke koju možemo poslati pritiskom na gumb. Sve poruke se pojavljuju od gore prema dolje. Ako smo mi sami poslali poruku, poruka je plave boje i nije nam prikazano korisničko ime. Ako je neki drugi korisnik poslao poruku, poruka je sive boje, piše korisničko ime pošiljatelja i poruka je malo drugačije zaobljena. Poruke se nigdje ne spremaju tako da ih vide isključivo korisnici grupe koji su trenutno aktivni i unutar grupe u trenutku slanja (odnosno primanja) poruke. Za slanje poruka koristimo *WebSocket* protokol o kojem ćemo više reći u zadnjem potpoglavlju. Slanje poruka je prikazano na slici 3.4.



Slika 3.4: Slanje poruka u grupi

Kao što smo već spomenuli, ako smo vlasnik grupe možemo dodavati korisnike u grupu. Ipak, ne možemo dodati bilo kojeg korisnika aplikacije već samo korisnike koji su nam prijatelji. Listom prijatelja možemo upravljati ako na gornjoj traci, na vrhu web aplikacije, kliknemo na *Friends*. Slično kao i kod grupa, klikom na karticu s oznakom plus možemo dodati novog prijatelja pomoću njegovog korisničkog imena. To je moguće jer su korisnička imena jedinstvena i dodatno, korisničko ime mora biti jedna riječ (bez bjelina). Dodavanjem korisnika na vlastiti popis prijatelja, ne dodajemo sebe automatski na njegov popis prijatelja, već nas on mora dodati. Kartice prijatelja opet imaju gumb u gornjem desnom kutu pomoću kojeg možemo maknuti korisnika s popisa prijatelja što je upravo i

prikazano na slici 3.5.

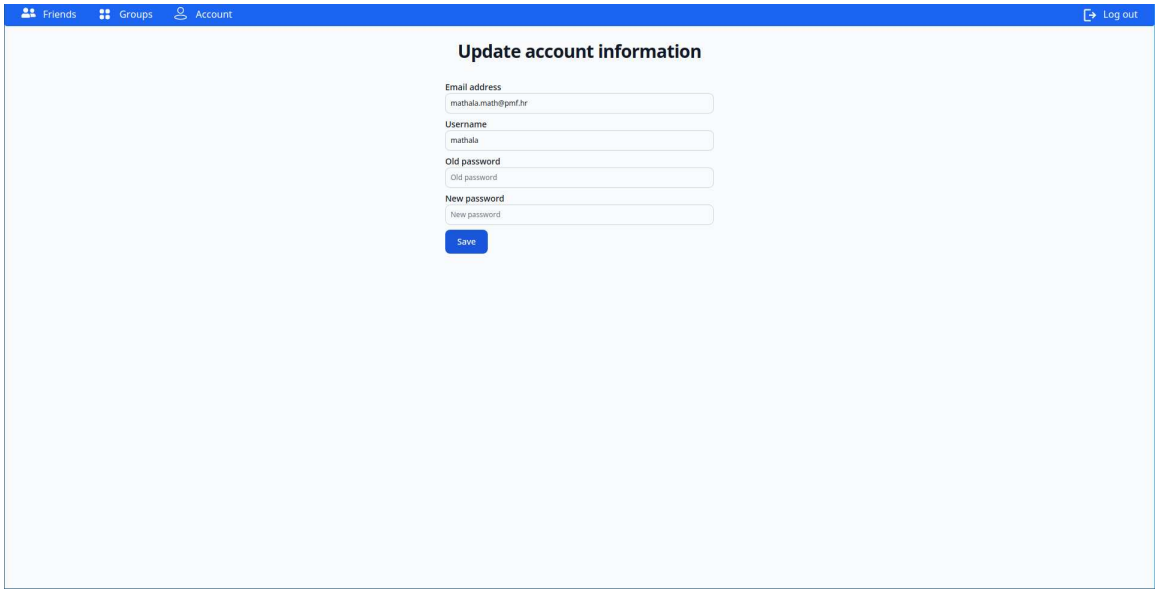


Slika 3.5: Micanje korisnika s liste prijatelja

Ako na gornjoj traci kliknemo na *Account*, otvara se forma za upravljanje korisničkim računom. Tu možemo promijeniti email, korisničko ime i lozinku. Lozinka, iz sigurnosnih razloga, nije prikazana, za razliku od korisničkog imena i emaila. Kod svake promjene podataka, moramo upisati i trenutnu lozinku kako bismo spriječili krađu korisničkih računa. Polje za unos nove lozinke može ostati prazno kada mijenjamo podatke što označava da lozinku ne mijenjamo. Izgled forme za mijenjanje korisničkih podataka vidimo na slici 3.6.

Zadnja opcija koju još vidimo na gornjoj traci je gumb za odjavu korisnika koji obriše sve kolačiće na stranici i time gubimo pristup aplikaciji dok se ponovno ne prijavimo. Svaki korisnički klik u klijentskoj aplikaciji koji mijenja stanje entiteta u bazi, šalje zahtjev serverskoj aplikaciji. Na serveru, pristup svakom zaštićenom resursu autentificiramo i prosljeđujemo pojedinoj funkciji koja obrađuje dani zahtjev. Zahvaljujući Go paketu *gin*, sve nam je to omogućeno relativno jednostavno, kako smo već opisali u prethodnom poglavlju. Unutar ove razvijene aplikacije, upravo tako i je implementirano sve na serverskoj strani aplikacije.

Kada bi se dalje nastavilo raditi na aplikaciji, mogli bismo dodati razne zanimljive aspekte društvenih mreža i tako obogatiti mogućnosti aplikacije. Prva stvar koja bi se mogla poboljšati je spremanje poruka koje korisnici šalju. Tako nešto bi postigli spremajući poruke na korisnička računala ili ostale uređaje. Iako korisno, to ne bi bilo povezano uz



Slika 3.6: Mijenjanje korisničkih podataka

Go, već bi se proširile mogućnosti klijentskog dijela aplikacije. Mogli bismo dodati slanje poruka izravno između dva korisnika, a ne samo unutar grupa, prikaz kada je prijatelj *online*, a kada nije, omogućiti slanje proizvoljnih datoteka, dodavanje statusa vidljivih na korisničkom profilu prijatelja i mnogo drugih mogućnosti.

Pošto se aplikacija zasniva na REST API-ju, sada ćemo još navesti sve rute API-ja i kratko objasniti za što se koristi pojedina ruta. Prvo ćemo navesti sve nezaštićene rute, odnosno rute kojima može pristupiti neautentificirani korisnik. *WebSocket* i *refresh* token ćemo sada samo spomenuti, a objašnjenja tih pojmova su dana upravo u sljedeća dva potpoglavlja. Rute navodimo u obliku: HTTP metoda ”putanja” - kratki opis.

- POST ”/register” - registracija korisnika
- POST ”/tokens/authenticate” - autentifikacija korisnika
- POST ”/tokens/renewAccess” - obnavljanje JWT pristupnog tokena
- GET ”/ws/groups/:id” - uspostavljanje *WebSocket* konekcije

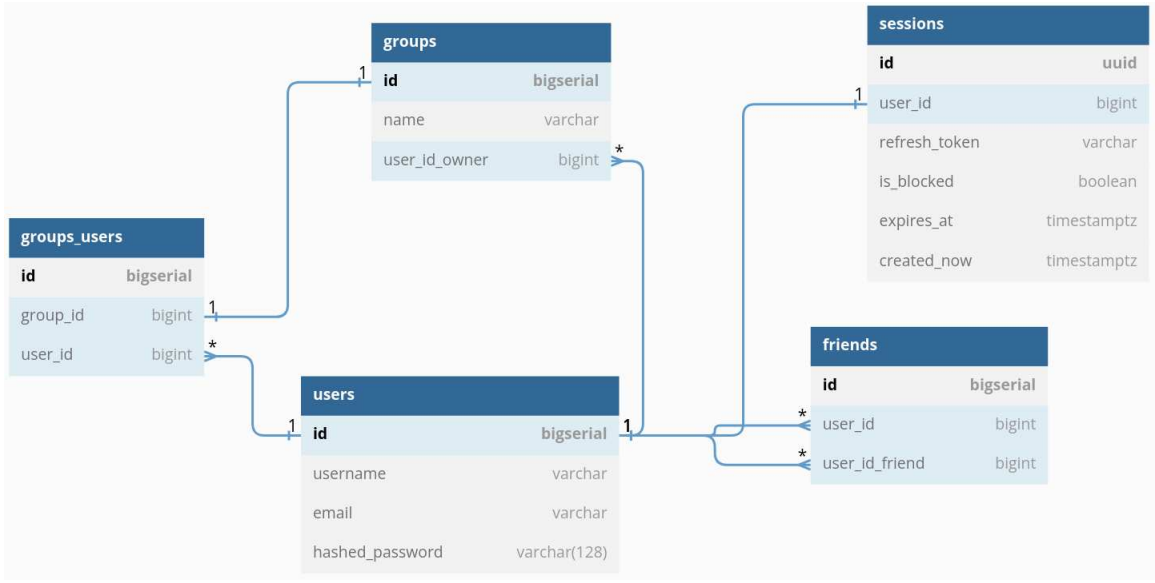
Sada ćemo u istom formatu navesti zaštićene rute odnosno rute kojima se može pristupiti isključivo ako smo autentificirani korisnik aplikacije. Kao što smo već spominjali u prethodnom poglavlju, korisnici su autentificirani uz pomoć *JWT* tokena.

- DELETE ”/tokens/refreshToken” - brisanje *refresh* tokena

- GET ”/user” - dohvaćanje podataka o prijavljenom korisniku
- PUT ”/user” - promjena korisničkih podataka
- DELETE ”/user” - brisanje korisničkog računa
- GET ”/friends” - dohvaćanje liste prijatelja
- POST ”/friends” - dodavanje novog prijatelja
- DELETE ”/friends/:id” - brisanje prijatelja s liste
- GET ”/groups” - dohvaćanje grupa kojima pripada prijavljeni korisnik
- GET ”/groups/:id” - pristup pojedinoj grupi
- GET ”/groups/:id/users” - dohvaćanje članova određene grupe
- POST ”/groups/:id/users” - dodavanja prijatelja u grupu
- DELETE ”/groups/:id/users” - brisanje korisnika iz grupe
- POST ”/groups” - kreiranje nove grupe
- DELETE ”/groups/:id/leave” - napuštanje grupe
- DELETE ”/groups/:id” - brisanje grupe

Za kraj potpoglavlja o razvijenoj aplikaciji, navodimo sliku 3.7 na kojoj su prikazani svi entiteti korišteni za razvoj aplikacije s pripadnim kardinalitetima. Dijagram je napravljen koristeći *DB Diagram*, jednostavan alat za vizualizaciju ER (*Entity-Relation*) dijagrama [10].

Vidimo da je centralna tablica upravo tablica *users* koja je povezana sa svim ostalim tablicama. O tablici *session* ćemo govoriti više u sljedećem potpoglavlju, a vidimo da je ona isključivo vezana uz korisnika odnosno tablicu *users*. Prijatelje spremamo u tablicu *friends* i to tako da ako korisnik A doda za prijatelja korisnika B, tada će u tablicu biti upisan novi redak gdje će *user_id* biti *ID* korisnika A, a *user_id_friend* bit će *ID* korisnika B. Na sličan način spremamo podatke i o tome koji korisnik pripada kojoj grupi. Kao što vidimo model nije kompliciran, a kada bi htjeli dodatno proširiti mogućnosti aplikacije, morali bismo dodati nove stupce i nove tablice. Nakon što imamo shemu baze postavljenu, kako bi pojednostavili upravljanje bazom, mijenjanje stanja entiteta baze i samo pisanje upita i njihovo pozivanje u aplikaciji, koristimo Go paket *sqlc* koji smo spominjali u prethodnom poglavlju.



Slika 3.7: Shema baze podataka s kardinalitetima

3.2 *Refresh* token i sesija

Kao što smo već rekli, JWT tokeni nam služe za autentifikaciju korisnika u aplikaciji. Korisnik dobije JWT token nakon što upiše svoje korisničko ime i lozinku, no JWT token je u pravilu kratkog trajanja, na primjer 5 minuta. Kada bi korisnik svakih 5 minuta korištenja aplikacije morao upisivati novu lozinku, to bi dovelo do vrlo lošeg korisničkog iskustva, i to želimo izbjeći. Za to nam treba neki oblik sesije (*session*) što je samo mehanizam kojim možemo na relativno siguran način prepoznati pojedinog, prijavljenog korisnika aplikacije. Potreba za takvim mehanizmom proizlazi iz činjenice da je HTTP protokol koji ne pohranjuje stanje između dva zahtjeva (*stateless protocol*).

Za implementaciju sesije u ovoj aplikaciji se koristi *refresh* token. *Refresh* token je također JWT token s vlastitim teretom. JWT token kakav smo spominjali u prošlom poglavlju služi za davanje pristupa (*access*) korisniku našoj aplikaciji pa da ne bi došlo do zabune, nazivat ćemo ga nadalje pristupni token. Postoje dvije bitne razlike između ta dva tokena. Prva, *refresh* token ima puno duži vijek trajanja. Druga, *refresh* token spremamo u bazu podataka, a pristupni token ne spremamo.

Vrijeme trajanja *refresh* tokena određuje koliko se često korisnik mora prijaviti u našu aplikaciju. Sami *refresh* token ne koristimo prilikom svakog zahtjeva već samo kada nam pristupni token istekne. Tada moramo zatražiti od servera da nam generira novi pristupni token koristeći naš *refresh* token. Kao što smo već rekli, *refresh* token spremamo u bazu podataka. Tablica u koju ga spremamo može izgledati slično kao u sljedećem primjeru.

```
CREATE TABLE "sessions" (
  "id" uuid PRIMARY KEY,
  "user_id" bigint UNIQUE NOT NULL,
  "refresh_token" varchar NOT NULL,
  "is_blocked" boolean NOT NULL DEFAULT false,
  "expires_at" timestamptz NOT NULL,
  "created_at" timestamptz NOT NULL DEFAULT (now()),
  CONSTRAINT fk_user FOREIGN KEY(user_id) REFERENCES users(id)
);
```

Primjer 3.1: Tablica za spremanje *refresh* tokena

Tablicu zovemo *sessions* jer zapravo to predstavlja, apstraktni koncept sesije implementiramo koristeći *refresh* token. Uz svaku sesiju vežemo korisnika, odnosno korisnički ID i ta veza je 1:1 u smislu veza između dva entiteta. Cijeli kodirani (standardno Base64Url kodiranje kod JWT tokena) *refresh* token spremamo u bazu podataka tako da možemo usporediti s tokenom koji dobijemo od klijentske aplikacije. Radi jednostavnosti, spremamo u bazu podatke kada smo napravili token i kada mu ističe valjanost, iako su te vrijednosti već kodirane u sami token. Za ta dva podatka koristimo *timestamptz*, tip podatka koji olakšava rad s različitim vremenskim zonama.

is_blocked stupac zahtjeva dodatno objašnjenje. JWT tokene generiramo na serveru i šaljemo korisniku. Ako je korisnik zapravo frontend aplikacija koju također razvijamo, moramo razmisliti gdje ćemo spremati te tokene. Najsigurnije mjesto su kolačići (*cookies*) s atributom *HTTP only*. *HTTP only* atribut znači da API *Document.cookie* u JavaScriptu ne može pristupiti danom kolačiću. Time znatno povećavamo sigurnost naše aplikacije i eliminiramo mogućnosti nekih sigurnosnih napada. Ipak, kada bi neki zlonamjerni akter uspio doći do našeg *refresh* tokena, efektivno bi bio prijavljen u aplikaciju s našim korisničkim računom. Postoje različiti načini kojima se takvi napadi mogu spriječiti, ali kad se tako nešto dogodi, efektivan način za eliminiranje problema je da invalidiramo *refresh* token. Da bi ovaj pristup radio, moramo svaki put kada izdajemo novi pristupni token na temelju *refresh* tokena, provjeriti je li *refresh* token invalidiran (blokiran). Ako je, ne izdajemo novi JWT token. Tada se pravi korisnik korisničkog računa može prijaviti u aplikaciju s korisničkim imenom i lozinkom i pritom će dobiti novi *refresh* token.

Slijedi isječak iz funkcije za autentifikaciju korisnika koja se poziva kada se korisnik želi prijaviti u našu aplikaciju. Pretpostavka je da smo već ranije unutar funkcije dohvatili potrebne podatke o korisniku i imamo ih u varijabli `user`, varijabla `ctx` predstavlja pokazivač na instancu klase `gin.Context`.

```
session, err := server.store.GetSessionByUserId(ctx, user.ID)
  if err != nil || time.Now().After(session.ExpiresAt) || session.
    IsBlocked {
     refreshToken, refreshPayload, err := server.tokenMaker.CreateToken(
    user.ID, server.config.RefreshTokenDuration)
     if err != nil {
```

```
5       log.Println("ERROR: ", err.Error())
6       ctx.JSON(http.StatusInternalServerError, consts.
    InternalErrorMessage)
7       return
8     }
9
10    session, err = server.store.CreateSession(ctx, sqlc.
    CreateSessionParams{
11      ID:           refreshPayload.ID,
12      UserID:       user.ID,
13      RefreshToken: refreshToken,
14      UserAgent:    ctx.Request.UserAgent(),
15      ClientIp:     ctx.ClientIP(),
16      IsBlocked:    false,
17      ExpiresAt:    refreshPayload.ExpiredAt,
18    })
19    if err != nil {
20      log.Println("ERROR: ", err.Error())
21      ctx.JSON(http.StatusInternalServerError, consts.
    InternalErrorMessage)
22      return
23    }
24  }
```

Primjer 3.2: Provjera postojećeg *refresh* tokena

U gornjem primjeru prvo pokušamo dohvatiti *refresh* token iz baze pomoću korisničkog ID-a, što možemo napraviti zahvaljujući tome što je korisnički ID jedinstveni stupac u tablici *session*. Ako prilikom dohvaćanja *refresh* tokena (odnosno sesije) dobijemo grešku (što uključuje i ne postojanje tokena) ili ako je token istekao ili blokiran, napravimo novi token i spremimo ga u bazu.

Izdavanje novog pristupnog tokena za korisnika koji je prijavljen u aplikaciji i ima *refresh* token je implementirano na posebnoj putanji na serveru (*endpoint*). Kada klijentska aplikacija zatraži neki zaštićeni resurs na serveru, mora poslati svoj pristupni token. Ako je taj token istekao, mora prvo zatražiti novi pristupni token i tek onda zatražiti zaštićeni resurs. *Refresh* token iz klijentske aplikacije možemo poslati na server pomoću zaglavlja i onda na serveru napraviti potrebne provjere valjanosti tokena, slično kao u prethodnom poglavlju s pristupnim tokenom. Nakon što validiramo *refresh* token dobiven od klijentske aplikacije, možemo iz njegovog tereta dobiti ID tokena i s tim ID-em dohvatiti *refresh* token iz baze podataka i napraviti dodatne validacije da potvrdimo kako se podudaraju. Ako je sve u redu, generiramo novi pristupni token i vraćamo ga klijentskoj aplikaciji. Kod je vrlo sličan validiranju pristupnog tokena iz prošlog poglavlja pa ga nećemo sada ponavljati. Samo trebamo imati na umu da usporedimo podatke iz tokena dobivenog od klijentske aplikacije (kojem ne vjerujemo) i podataka koje dobijemo iz baze podataka.

3.3 Slanje poruka i *WebSocket* protokol

U aplikaciji razvijenoj u sklopu ovog diplomskog rada postoje korisnici i grupe kojima se korisnici mogu pridružiti. Svi korisnici grupe mogu slati poruke tako da ih ostali, aktivni korisnici grupe mogu vidjeti. Poruke se ne spremaju nigdje, tako da ako korisnik nije u web pregledniku navigiran unutar same grupe za vrijeme slanja poruke, neće je moći vidjeti. Poruke bismo mogli spremati u bazu podataka i na svakom ulasku u grupu prikazati sve poruke, no to vjerojatno nije najbolji način za spremanje poruka i nije usko vezan uz Go pa nije niti implementiran. Slanje poruka je implementirano dijelom u Go-u, a dijelom na klijentskoj aplikaciji. O implementaciji na klijentskoj aplikaciji nećemo govoriti, samo ćemo se fokusirati na važniji i veći dio koji je u Go-u.

Pošto se radi o porukama koje se šalju u stvarnom vremenu, za njihovo slanje koristimo *WebSocket* protokol. *WebSocket* je komunikacijski protokol koji pruža potpunu dvostranu (*full-duplex*) komunikaciju preko jedne TCP konekcije. U ovom primjeru, to znači da klijentska aplikacija može slati podatke serveru i server klijentskoj aplikaciji sve dok jedna strana ne prekine konekciju. Za razliku od HTTP protokola, kod slanja poruka preko *WebSocket* protokola, server ne autentificira korisnika. Autentifikacija se događa kada unutar preglednika se pozicioniramo na *endpoint* određene grupe i dalje za slanje poruka se koristi *WebSocket* protokol. Unutar *gin* routera moramo dodati novi *endpoint*, znači nešto poput sljedeće linije: `router.GET("/ws/groups/:id", hub.ServeWs)`, gdje je `hub.ServeWs` samo funkcija koja je odgovorna za obradu zahtjeva koji dolaze na taj endpoint. *WebSocket* implementacija unutar JavaScripta (koja se koristi u ovoj aplikaciji) ne dopušta slanje dodatnih zaglavlja pa je ovaj endpoint zapravo dostupan svima koji bi se probali spojiti, što je naravno, loše. Za industrijski standard, morali bismo koristiti neki oblik autentifikacije korisnika i kod ovog endpointa, što je trivijalna promjena u Go-u, a ne tako trivijalna promjena u JavaScriptu pa se nećemo fokusirati na to.

Sada ćemo navesti nekoliko primjera funkcija i struktura koje koristimo za omogućavanje slanja poruka unutar grupa. Sve potrebne strukture i funkcije za implementaciju *WebSocket* protokola, naravno, ne pišemo sami, već koristimo Go paket *github.com/gorilla/websocket*, o kojem možemo saznati sve informacija iz službene dokumentacije [1].

```
1  func NewUpgrader(config *util.Config) *websocket.Upgrader {
2    return &websocket.Upgrader{
3      CheckOrigin: func(r *http.Request) bool {
4        return config.Client == r.Header.Get("Origin")
5      },
6    }
7  }
8
9  func NewHub(config *util.Config) *hub {
10   return &hub{
11     broadcast:  make(chan message),
```

```
12      register:   make(chan subscription),
13      unregister: make(chan subscription),
14      groups:     make(map[int64]map[*connection]bool),
15      upgrader:   NewUpgrader(config),
16    }
17
18  }
```

Primjer 3.3: Funkcija za kreiranje *WebSocket upgradera*

Prva funkcija u gornjem primjeru vraća pokazivač na instancu strukture *Upgrader* iz paketa *websocket*. Ona je zadužena za nadogradnju HTTP protokola u *WebSocket* protokol. Dodatno jedino još definiramo funkciju za člansku varijablu *CheckOrigin* koja uspoređuje je li zaglavlje *Origin* jednako klijentskoj aplikaciji specificiranoj unutar konfiguracije. Druga funkcija je zadužena za kreiranje strukture *hub* i vraća pokazivač na stvorenu strukturu. Slijedi primjer struktura koje koristimo kod *WebSocket* dijela servera, čija je središnja komponenta upravo spomenuta struktura *hub*.

```
1  type connection struct {
2    ws   *websocket.Conn
3    send chan []byte
4  }
5
6  type subscription struct {
7    hub     *hub
8    conn    *connection
9    groupId int64
10 }
11
12 type message struct {
13   data    []byte
14   groupId int64
15 }
16
17 type hub struct {
18   groups     map[int64]map[*connection]bool
19   broadcast  chan message
20   register   chan subscription
21   unregister chan subscription
22   upgrader   *websocket.Upgrader
23 }
```

Primjer 3.4: Strukture korištene za *WebSocket* dio servera

Struktura *connection* predstavlja jednu primitivnu konekciju na naš *WebSocket* server. Sastoji se od dvije članske varijable. `ws` predstavlja *WebSocket* konekciju, a kanal `send` predstavlja kanal na koji dolaze poruke korisnika koji je otvorio danu *WebSocket* konekciju. Struktura *subscription* predstavlja nadogradnju na strukturu *connection*. Ona po-

vezuje *WebSocket* server, jednu konekciju i grupu kojoj pripada korisnik koji je otvorio konekciju. *message* sadrži poruku koju korisnik šalje ili prima preko *WebSocket* protokola, no osim samih bajtova, također pohranjuje podatak o kojoj se grupi radi. Zadnja struktura *hub* povezuje sve i djeluje kao router za poruke koje dolaze. Prva varijabla `groups` u strukturi predstavlja sve otvorene konekcije, podijeljene po grupama u koje pripadaju. `map[*connection]bool` dio je zapravo nezgrapan i pokazuje jedan od nedostataka Go-a. Naime, ovdje nam uopće ne treba struktura mapa već skup (*set*), ali struktura skup nije ugrađena u Go pa je ovo najbrži način da koristimo "skup". `broadcast` je kanal koji koristimo kako bismo dobivenu poruku poslali svim konekcijama u grupi kojoj pripada poruka. `register` je kanal koji koristimo da registriramo novo zaprimljenu konekciju i dodamo je u varijablu `groups`. `unregister` služi za micanje konekcija iz mape `groups`. `upgrader` je pokazivač iz prethodnog primjera.

Slijedi primjer funkcije *Run* na strukturi *hub* koju pokrećemo u vlastitoj gorutini, a pokrećemo je i kad HTTP server. Ona je zapravo beskonačna petlja koja prima poruke na spomenute kanale *hub* strukture i ovisno o kanalu odredi dalje što treba s porukom.

```
func (hub *hub) Run() {
  for {
    select {
    case sub := <-hub.register:
      connections := hub.groups[sub.groupId]
      if connections == nil {
        connections = make(map[*connection]bool)
        hub.groups[sub.groupId] = connections
      }
      hub.groups[sub.groupId][sub.conn] = true

    case sub := <-hub.unregister:
      connections := hub.groups[sub.groupId]
      if connections != nil {
        if _, ok := connections[sub.conn]; ok {
          delete(connections, sub.conn)
          close(sub.conn.send)
          if len(connections) == 0 {
            delete(hub.groups, sub.groupId)
          }
        }
      }

    case msg := <-hub.broadcast:
      connections := hub.groups[msg.groupId]
      for conn := range connections {
        select {
        case conn.send <- msg.data:
        default:
```

```
30            close(conn.send)
31            delete(connections, conn)
32            if len(connections) == 0 {
33              delete(hub.groups, msg.groupId)
34            }
35          }
36        }
37      }
38    }
39  }
```

Primjer 3.5: Metoda *Run* strukture *hub*

Ako poruka dođe na kanal `register`, dodamo novu konekciju u pripadnu grupu u varijabli `groups`, a ako nema niti jedne konekcije za danu grupu, napravimo i novu mapu (odnosno po funkciji, set). Ako poruka dođe na kanal `unregister`, brišemo konekciju iz pripadne grupe te ako više nema konekcija za danu grupu, brišemo i mapu koja predstavlja konekcije za tu grupu. Ako poruka dođe na kanal `broadcast`, imamo dvije mogućnosti. Ako poruka ima ikakve podatke (*msg.data*) proslijedimo svim konekcijama u pripadnoj grupi. Ako poruka nema podataka, onda predstavlja poseban signal da zatvorimo sve konekcije za danu grupu.

Slijede dvije metode definirane na strukturi *subscription*. One služe za prosljeđivanje poruka s pojedine konekcije prema *hubu* i s *huba* prema pojedinoj konekciji. Obje metode se sastoje od beskonačne petlje i pozivaju se u vlastitim gorutinama što ćemo vidjeti u primjeru nakon ovoga.

```
1  func (sub subscription) readPump() {
2    conn := sub.conn
3    defer func() {
4      sub.hub.unregister <- sub
5      conn.ws.Close()
6    }()
7    for {
8      _, msg, err := conn.ws.ReadMessage()
9      if err != nil {
10       break
11     }
12     m := message{msg, sub.groupId}
13     sub.hub.broadcast <- m
14   }
15 }
16
17 func (sub *subscription) writePump() {
18   conn := sub.conn
19   for msg := range conn.send {
20     conn.ws.WriteMessage(websocket.TextMessage, msg)
```

```
21    }
22    conn.ws.Close()
23  }
```

Primjer 3.6: Metode za prosljeđivanje poruka

Prva metoda *readPump* blokira dretvu dok ne dobije poruku pomoću *ReadMessage* funkcije. Ako funkcija vrati grešku, prekidamo petlju i oslobodimo resurse. Ako nema greške, poruku koristimo kako bi je poslali na `broadcast` kanal strukture *hub* zajedno s pripadnim ID-em grupe. Kada *hub* zaprimi poruku, pošalje je svim konekcijama u grupi na njihov pripadni `send` kanal. Druga metoda u gornjem primjeru, *writePump*, dalje obrađuje poruku poslanu s *huba*. Sve dok kanal `send` nije zatvoren, dobivene poruke prosljeđujemo na *socket*. Kada je kanal zatvoren, zatvaramo i konekciju. Napokon, slijedi i posljednja funkcija *ServeWs* koju smo već spomenuli na početku potpoglavlja.

```
1  func (hub *hub) ServeWs(ctx *gin.Context) {
2    groupId, _ := strconv.Atoi(ctx.Param("groupId"))
3    ws, err := hub.upgrader.Upgrade(ctx.Writer, ctx.Request, nil)
4    if err != nil {
5      return
6    }
7    c := &connection{
8      send: make(chan []byte, 256),
9      ws:   ws,
10   }
11   sub := subscription{
12     hub:     hub,
13     conn:    c,
14     groupId: int64(groupId),
15   }
16   hub.register <- sub
17   go sub.writePump()
18   go sub.readPump()
19 }
```

Primjer 3.7: Metoda *ServeWs* strukture *hub*

ServeWs metoda je zadužena za obradu HTTP zahtjeva koji dolaze na HTTP server. Iz URI-a dobivamo parametar *groupId*. Zatim nadogradimo HTTP protokol u *WebSocket*. Stvorimo potrebne instance struktura *connection* i *subscription*. Pošaljemo poruku na `register` kanal čime efektivno registriramo konekciju tako da nam *hub* može prosljeđivati poruke kao i mi njemu. Na kraju još pokrećemo funkcije *writePump* i *readPump* na danoj instanci *subscription* strukture u vlastitim gorutinama. To je sve što je potrebno za jedan jednostavan *WebSocket* server.

Bibliografija

[1] Više autora (otvoreni kod), *Dokumentacija paketa koji implementira WebSocket protokol u Go-u*, `https://pkg.go.dev/github.com/gorilla/websocket`, posjećeno u veljači 2023.

[2] _____, *Dokumentacija paketa sqlc*, `https://docs.sqlc.dev/en/stable/`, posjećeno u veljači 2023.

[3] _____, *Dokumentacija paketa za konfiguraciju Go aplikacije viper*, `https://github.com/spf13/viper`, posjećeno u veljači 2023.

[4] _____, *Dokumentacija za program go*, `https://pkg.go.dev/cmd/go`, posjećeno u veljači 2023.

[5] _____, *Dokumentacija za razvojni okvir gin*, `https://gin-gonic.com/docs/`, posjećeno u veljači 2023.

[6] _____, *Službena stranica za instalaciju Go-a*, `https://go.dev/dl/`, posjećeno u veljači 2023.

[7] Više autora (otvoreni standard), *Uvodna dokumentacija o Json Web Token*, `https://jwt.io/introduction`, posjećeno u veljači 2023.

[8] Matija Halavanja, *Klijentska aplikacija razvijena u sklopu diplomskog rada*, `https://github.com/mhalavanja/client`, posjećeno u veljači 2023.

[9] _____, *Serverska aplikacija razvijena u sklopu diplomskog rada*, `https://github.com/mhalavanja/go-rest-api`, posjećeno u veljači 2023.

[10] Holistics, *DB Diagram, alat za vizualiziranje entiteta i veza u bazi*, `https://dbdiagram.io/home`, posjećeno u veljači 2023.

Sažetak

U ovom diplomskom radu predstavili smo programski jezik Go, naveli razne koncepte i konkretne Go pakete koji ih implementiraju i olakšavaju nam razvoj web-aplikacija. Prvo smo se upoznali sa sintaksom i tipovima podataka u Go-u. Nakon toga, vidjeli smo kako se Go razlikuje od drugih popularnih jezika u njegovom pristupu objektno orjentiranom programiranju i paralelnom programiranju. Zatim smo prošli kroz postavljanje i upravljanje razvojnom okolinom i paketima. Dalje, istaknuli smo razvojni okvir *gin* kao jedan od mogućih paketa za razvoj serverske infrastrukture. Fokusirali smo se na autentifikaciju korisnika i razvoj sesije uz pomoć *JWT* tokena. Pokazali smo kako možemo koristiti SQL i paket *sqlc* za upravljanje modelom i poslovnom logikom, za razliku od uobičajenih rješenja baziranih na konceptu *ORM*. Spomenuli smo i važnost konfiguracijskih varijabli kod različitih okolina u kojima aplikacija može raditi. U zadnjem poglavlju smo još prikazali razvijenu aplikaciju te pokazali jednostavan *WebSocket* server uz pomoć kojeg smo implementirali mogućnost slanja poruka između korisnika.

Summary

In this thesis, we presented the Go programming language, listed various concepts and specific Go packages that implement them and facilitate the development of web applications. First, we introduced the syntax and data types in Go. After that, we saw how Go differs from other popular languages in its approach to object-oriented programming and parallel programming. We then went through setting up and managing the development environment and packages. Next, we highlighted the *gin* development framework as one of the possible packages for server infrastructure development. We focused on user authentication and session development with the help of *JWT* tokens. We have shown how we can use SQL and the *sqlc* package to manage the model and business logic, unlike the usual solutions based on the *ORM* concept. We also mentioned the importance of configuration variables in different environments in which the application can work. In the last chapter, we also presented the developed application and a simple *WebSocket* server with the help of which we implemented the feature of sending messages between users.

Životopis

Rođen sam 21.10.1997. godine u Sisku gdje sam završio Osnovnu školu Viktorovac. Nakon završene osnovne škole 2012. godine, upisao sam XV. gimnaziju u Zagrebu. Na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu 2016. godine, upisao sam *Preddiplomski sveučilišni studij Matematika*. Diplomu sveučilišnog prvostupnika matematike stekao sam 2020. godine završivši preddiplomski studij. Iste godine upisujem diplomski studij *Računarstvo i matematika*, također na PMF-MO. Tijekom većeg dijela fakultetskog obrazovanja bavio sam se programiranjem, posebno razvojem web aplikacija, što mi je i dalje najveći interes.