

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Nemanja Subotić

PROGRAMSKI JEZICI ELM I ELIXIR U
RAZVOJU STUDENTSKOG VEB PORTALA

master rad

Beograd, 2022.

Mentor:

dr Milena VUJOŠEVIĆ JANIČIĆ, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip MARIĆ, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Ivan ČUKIĆ, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: 16. septembar 2022.

Naslov master rada: Programski jezici Elm i Elixir u razvoju studentskog veb portala

Rezime: Funkcionalni programski jezici *Elm* i *Elixir* pripadaju novoj generaciji programskih jezika koji sve više dobijaju na popularnosti. *Elm* je statički tipiziran funkcionalni programski jezik namenjen isključivo za izradu korisničkog interfejsa veb aplikacija. Sa druge strane, *Elixir* je dinamički tipiziran funkcionalni programski jezik opšte namene i zajedno sa razvojnim okvirom *Phoenix* zauzima značajno mesto u razvoju veb aplikacija. Cilj rada je prikaz najbitnijih karakteristika i koncepta navedenih programskih jezika, kao i mogućnosti razvojnog okvira *Phoenix*, kroz razvoj studentskog veb portala koji prati aktivnosti kursa *Metodologija stručnog i naučnog rada*.

Ključne reči: programski jezici, funkcionalno programiranje, programski jezik *Elm*, programski jezik *Elixir*, razvojni okvir *Phoenix*

Sadržaj

1	Uvod	1
2	Programski jezik i okruženje Elm	3
2.1	Uputstvo za instalaciju	4
2.2	Osnovne odlike	4
2.3	Elm kao platforma	5
2.4	Uticaj drugih programskih jezika na Elm	6
2.5	Elm kao jezik	7
2.6	Arhitektura Elm	22
3	Programski jezik Elixir	27
3.1	Programski jezik i razvojna platforma Erlang	28
3.2	Uputstvo za instalaciju	30
3.3	Osnovne odlike	30
3.4	Osnove jezika Elixir	30
4	Portal MSNR	52
4.1	Arhitektura portala	53
4.2	Šema baze i opis osnovnih entiteta aplikacije	53
5	Implementacija serverskog dela portala	57
5.1	Razvojni okvir Phoenix	57
5.2	Registrovanje studenata	69
5.3	Autentifikacija i autorizacija korisnika	72
5.4	Ubacivanje i dodela aktivnosti	74
5.5	Izvršavanje i ocenjivanje aktivnosti	74
6	Implementacija klijentskog dela portala	77

SADRŽAJ

6.1	Elm aplikacija	78
6.2	Moduli stranica i putanja aplikacije	83
6.3	Stilizovanje aplikacije	86
6.4	Struktura profesorske stranice	87
6.5	Struktura studentske stranice	88
7	Zaključak	92
	Bibliografija	94

Glava 1

Uvod

Savremen razvoj programskih jezika i odgovarajućih razvojnih okvira za rad sa njima usmeren je ka ubrzanom procesu implementacije složenih programskih rešenja. Posebno napreduje razvoj podrške funkcionalnim konceptima, kao i razvoj novih funkcionalnih programskih jezika među kojima se nalaze i *Elm* i *Elixir*.

Programski jezik *Elm* odlikuje laka upotrebljivost, visoke performanse, robusnost i otpornost na greške. Kao jezik specifične namene za klijentsko (eng. *frontend*) veb programiranje uvodi poseban način izvršavanja aplikacije, poznat kao *arhitektura Elm*, koja je poslužila kao inspiracija mnogim *JavaScript* bibliotekama. Programski jezik *Elixir* nastaje kao moderan funkcionalni programski jezik sa osnovnom idejom da se izvršava na *Erlang* [33] virtuelnoj mašini. Kao takav je veoma zastupljen u razvoju distribuiranih sistema, ali je svoju upotrebu našao gotovo u svim oblastima računarstva. Za razvoj veb aplikacija najpopularniji razvojni okvir je *Phoenix*, koji nastaje i razvija se uporedo sa samim programskim jezikom.

Kurs *Metodologija stručnog i naučnog rada* (skraćeno MSNR) se održava na master studijama studijskog programa *Informatika* Matematičkog fakulteta Univerziteta u Beogradu. Osnovni cilj kursa jeste da studente uvede u metode stručnog i naučnog rada u oblasti računarstva i informatike. Tok kursa prati veliki broj studentskih aktivnosti koje kroz timski rad obrađuju teme pisanja, recenziranja i prezentovanja radova, kao i razvoj „mekih” veština. Dosadašnje izvršavanje aktivnosti zasnivalo se na komunikaciji putem mejlova i otpremanju datoteka na privremenim veb stranicama.

Cilj ovog rada je implementacija portala MSNR. Ovaj portal treba da bude centralno mesto organizacije kursa i izvršavanja aktivnosti. Nastavnik će na jednom mestu dodavati nove aktivnosti, dodeljivati ih studentima i ocenjivati njihovo izvr-

šavanje, dok će studenti sve aktivnosti izvršavati unutar portala i imati celokupan pregled aktivnosti, zajedno sa ocenama i komentarima nastavnika, na jednom mestu.

Detaljni opis programskog jezika *Elm* dat je u drugom poglavlju dok je u trećem poglavlju predstavljen programski jezik *Elixir*. Četvrto poglavlje sadrži opis funkcionalnosti portala MSNR i arhitekturu rešenja. U poglavlju pet prikazuje se implementacija serverskog dela upotrebom razvojnog okvira *Phoenix*. U poglavlju šest predstavljena je implementacija korisničkog interfejsa portala korišćenjem programskog jezika *Elm*. Sedmo poglavlje sadrži zaključak celog rada sa mogućim daljim unapređenjima.

Glava 2

Programski jezik i okruženje Elm

Evan Čapliki (*Evan Czaplicki*) je 2012. godine objavio svoju tezu „*Elm: Konkurentno FRP* ¹ za funkcionalne *GUI*-je ²” (eng. „*Elm: Concurrent FRP for Functional GUIs*”) [4] i, s ciljem da *GUI* programiranje učini prijatnijim, dizajnirao novi programski jezik — *Elm*. Na slici 2.1 prikazan je logo jezika. *Elm* je statički tipiziran, čisto funkcionalni programski jezik koji se kompilira, tačnije transpilira u *JavaScript* i namenjen je isključivo za kreiranje veb aplikacija. Takođe, *Elm* nije samo program-



Slika 2.1: Logo programskog jezika *Elm*

ski jezik već i platforma za razvoj aplikacija. Zahvaljujući funkcionalnoj prirodi i karakterističnom kompilatoru, *Elm* pruža programerima poseban osećaj sigurnosti i samouverenosti tokom refaktorisanja postojećih i dodavanja novih funkcionalnosti. Za *Elm* aplikacije važi da, u praksi, ne izbacuju neplanirane greške tokom izvršavanja (eng. *No Runtime Exceptions*).

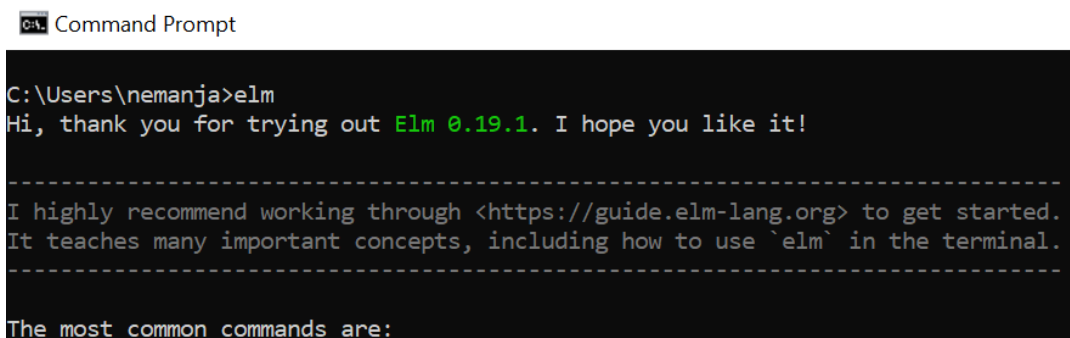
¹FRP je skraćenica za funkcionalno reaktivno programiranje (eng. *Functional Reactive Programming*)

²GUI je skraćenica za grafički korisnički interfejs (eng. *Graphical User Interface*)

2.1 Uputstvo za instalaciju

Pored želje da klijentsko (eng. *frontend*) programiranje učini prijatnijim, kreator jezika nastoji da ono bude i pristupačnije. Stoga, da bi se počelo sa korišćenjem programskog jezika *Elm*, instalacija nije potrebna, dovoljno je otići na zvaničnu veb stranicu i pokrenuti dostupan interaktivni kompilator [23], gde se može naći dosta primera, kao i vodič kroz *Elm*.

Za zahtevnije projekte neophodna je instalacija, koja je vrlo jednostavna. Potrebno je pratiti instrukcije sa zvanične stranice [10]. Takođe, moguća je instalacija pomoću alata **npm** [16]³. Na slici 2.2 prikazana je provera uspešne instalacije, koja se može izvršiti pokretanjem komande **elm** u komandnoj liniji, gde će se prikazati poruka dobrodošlice i spisak mogućih komandi o kojima će biti reči u sledećim poglavljima.



```
C:\Users\nemanja>elm
Hi, thank you for trying out Elm 0.19.1. I hope you like it!

-----
I highly recommend working through <https://guide.elm-lang.org> to get started.
It teaches many important concepts, including how to use `elm` in the terminal.
-----
The most common commands are:
```

Slika 2.2: Provera uspešne instalacije *Elm*-a

2.2 Osnovne odlike

Pored *No Runtime Exceptions*, jedna od glavnih odlika ovog jezika jeste kompilator, koji je izuzetno ugodan za rad. Mnogi programeri smatraju da *Elm* kompilator proizvodi najbolje poruke o greškama. Za razliku od drugih, *Elm* kompilator objašnjava zašto je došlo do greške i daje predloge za njihovo rešavanje, a takođe nema kaskadnih poruka. Kreator se vodio razmišljanjem da kompilator treba da bude asistent, ne samo alat.

³npm — *Node Package Manager* predstavlja alat za upravljanje paketima u *JavaScript* programskom jeziku

Elm koristi svoju verziju *virtuelnog DOM*⁴-a, koncepta koji se koristi u mnogim razvojnim okvirima klijentskog programiranja. Ideja je da se u memoriji čuva „virtuelna” reprezentacija korisničkog interfejsa na osnovu koje se ažurira „stvarni” *DOM*. Još jedna bitna karakteristika je nepromenljivost podataka, što znači da se jednom definisani podaci ne mogu više menjati. Direktna posledica nepromenljivosti podataka je veoma brzo iscrtavanje *HTML*-a, jer se poređenja u virtuelnom *DOM*-u mogu vršiti po referenci. Verzija *Elm 0.17* imala je najbrže iscrtavanje u poređenju sa tadašnjim aktuelnim verzijama popularnih okvira[6].

Elm se može integrisati i u postojeće *JavaScript* projekte za implementaciju pojedinačnih komponenti. Takođe, moguća je i komunikacija između ova dva programska jezika.

2.3 Elm kao platforma

Elm sa sobom donosi alat (tabela 2.1) i okruženje *Elm* (eng. *Elm Runtime*), koji su neophodni za razvoj i izvršavanje aplikacija. *Elm* kôd se nalazi u datotekama sa ekstenzijom *.elm* i prilikom kompilacije kreira se jedna izlazna *.js* datoteka. U izlaznoj datoteci se pored prevedenog koda iz ulaznih *.elm* datoteka nalaze i funkcije iz okruženja *Elm* potrebne za izvršavanje programa.

Alat	Kratak opis
repl	Pokretanje interaktivne sesije (eng. <i>Read-Eval-Print-Loop</i>)
init	Inicijalizacija projekta
reactor	Pokretanje lokalnog servera
make	Upotreba kompilatora
install	Preuzimanje paketa
diff	Prikazivanje razlika između različitih verzija istog paketa
bump	Određivanje broja naredne verzije paketa
publish	Objavljivanje paketa

Tabela 2.1: *Elm* alatke komandne linije

Kao zaseban jezik *Elm* ima i zaseban sistem za upravljanje paketima. Pokretanjem komande **elm init** kreira se prazan direktorijum *src* i datoteka *elm.json*, u kojoj se pored informacije o tipu projekta (aplikacija ili paket), *Elm* verzije i liste direktorijuma sa kodom, nalazi i spisak paketa koji se koriste u projektu. Dodavanje

⁴DOM — Obejktni model dokumenta (eng. *Document Object Model*)[17]

novog paketa se vrši pomoću komande **elm install** *naziv-paketa*. Svi paketi su javno dostupni (<https://package.elm-lang.org/>), nazivi paketa su oblika *autor/ime-paketa*.

Kompilacija se vrši naredbom **elm make** *<jedna-ili-više-elm-datoteka>*, ukoliko se ne navede izlazna datoteka pomoću argumenta **--output** generisaće se datoteka *index.html* sa prevedenim *JavaScript* kodom. Ostali argumenti kao i više informacija o drugim alatkama mogu se videti pomoću naredbe **elm naziv-alata --help**.

2.4 Uticaj drugih programskih jezika na Elm

Kao i većina statički tipiziranih funkcionalnih programskih jezika, *Elm* se zasniva na programskom jeziku *ML*, a budući da su u programskom jeziku *Haskell* napisani *Elm* kompilator i ostale alatke, *Haskell* je ostavio veliki uticaj i na sam jezik. Autor *Elm*-a smatra:

„Rekao bih da *Elm* pripada porodici *ML* jezika, sa sintaksom poput *Haskell*-a. Ako poredimo semantiku, *Elm* je dosta sličniji *OCaml*-u i *SML*-u.” [5]

ML (eng. *Meta Language*)[14] je statički tipiziran programski jezik opšte namene koji je nastao 1973. godine na Univerzitetu u Edinburgu. Vođa grupe koja je radila na dizajniranju programskog jezika *ML* bio je Robin Milner, dobitnik Turingove nagrade. Nastao je pod uticajem programskog jezika *LISP*, a razvijan je za implementiranje automatskog dokazivača teorema. Osnovna karakteristika jeste uvođenje automatskog zaključivanja tipova, a odlikuje ga i poklapanje obrazaca, Karijeve funkcije i posedovanje sakupljača otpadaka. *ML* nije čist funkcionalan jezik i nema ugrađenu podršku za lenjo izračunavanje. U porodicu *ML* jezika, između ostalih, spadaju i **Standard ML**, **OCaml** i **F#**.

Haskell[12] je čist funkcionalni programski jezik, naziv je dobio po matematičaru i logičaru Haskelu Bruks Kariju (eng. *Haskell Brooks Curry*). *Haskell* je strogo tipiziran, poseduje automatsko zaključivanje tipova i lenjo izračunavanje. Jezik je opšte namene, pruža podršku za paralelno i distribuirano programiranje. *Haskell* omogućava manje grešaka i veću pouzdanost kroz kraći i čistiji kôd, koji je lakši za održavanje.

2.5 Elm kao jezik

U ovom poglavlju predstavljene su osnove programskog jezika Elm — osnovni tipovi i strukture podataka, operatori, način definisanja i grupisanja funkcija, navođenje komentara, kontrola toka i obrada grešaka.

Komentari

Komentari u *Elm*-u se mogu navoditi na dva načina:

- Korišćenjem `--` za linijske komentare
- Navođenjem teksta između znakova `{- i -}` za komentare u više redova.

Osnovni tipovi podataka

```
> 'Z'
'Z' : Char
> "Zdravo!"
"Zdravo!" : String
> True
True : Bool
> 42
42 : number
> 42 / 10
4.2 : Float
> 42 // 10 --celobrojno deljenje
4 : Int
```

Primer koda 1: Osnovni tipovi podataka prikazani u interpretatoru

Osnovni tipovi podataka programskog jezika *Elm* su **Char**, **String**, **Bool**, **Int** i **Float**. U primeru koda 1 prikazani su osnovni tipovi korišćenjem interpretatora (`elm repl`). Budući da i *Elm* poseduje zaključivanje tipova, nakon izračunate vrednosti unetog izraza ispisuje se tip. U konkretnom primeru broj 42 se može posmatrati i kao tip **Int** i kao tip **Float**, pa interpretator vraća **number** kao tip, iako **number** nije konkretan tip podataka već poseban oblik tipske promenljive. Tipske promenljive objašnjene su dalje u posebnom poglavlju.

Tip **Char** služi za predstavljanje junikod (eng. *unicode*) karaktera. Karakteri se navode između dva apostrofa (`'a'`, `'0'`, `'\t'...`), a moguće je koristiti i junikod zapis `'\u{0000}'` - `'\u{10FFFF}'`.

Za razliku od programskog jezika *Haskell*, gde je niska (**String**) zapravo lista karaktera, u *Elm*-u je poseban tip i predstavlja sekvencu junikod karaktera. Sekvenca se navodi između jednostrukih ili trostrukih navodnika (primer koda 2).

```
> "\t Niska u jednom redu: escape navodnici \"Zdravo!\""
"\t Niska u jednom redu: escape navodnici \"Zdravo!\"" : String
>
> ""Niska u više redova
  sa "navodnicima"! ""
"Niska u više redova\n  sa \"navodnicima\"! " : String
```

Primer koda 2: Primeri niski

Tip **Bool** predstavlja logički tip i može imati vrednost **True** ili **False**.

Tip **Int** se koristi za prikazivanje celih brojeva. U trenutnoj verziji (*0.19.1*) opseg vrednosti je od -2^{53} do $2^{53} - 1$. Vrednosti se mogu navoditi i u heksadecimalnom obliku (**0x2A**, **-0x2b**).

Tip **Float** služi za predstavljanje brojeva u pokretnom zarezu po standardu *IEEE 754*. Vrednosti se mogu navoditi i pomoću eksponencijalnog zapisa, a decimalna tačka se mora nalaziti između dve cifre. Takođe, u skup vrednosti spadaju **NaN** i **Infinity** (primer koda 3).

```
> 1e3
1000 : Float
> 0/0
NaN : Float
> 1/0
Infinity : Float
```

Primer koda 3: Prikaz brojeva u pokretnom zarezu

Osnovni operatori

Kod aritmetičkih operacija, operatori **+**, **-**, ***** se mogu koristiti sa realnim i celim brojevima, dok imamo posebne operatore za deljenje (**/** i **//** koji su i ranije prikazani u primeru koda 1). *Elm* ne podržava implicitne konverzije tipova, pa prilikom sabiranja celog broja sa realnim, bez eksplicitne konverzije, kompilator prijavljuje grešku (primer koda 4). Postoji još i eksponencijalni operator **^**, a za celobrojno deljenje sa ostatkom koriste se funkcije **modBy** i **remainderBy**.

Od relacijskih operatora **==**, **/=**, **<**, **>**, **>=** i **<=**, jedino operator različitosti (**/=**) ima drugačiju sintaksu od uobičajene. Prioriteti između relacijskih operatora

```
> toFloat (9 // 3) + 3.2
6.2 : Float
> 9 // 3 + round 3.2
6 : Int
> 9 // 3 + 3.2 -- TYPE MISMATCH error
```

Primer koda 4: Upotreba eksplicitne konverzije tipova

nisu definisani i prilikom njihovog kombinovanja izrazi se moraju odvojiti zagradama. *Elm* pruža logičke operatore *i* `&&` i *ili* `||` kao i funkcije za negaciju `not` i ekskluzivno ili `xor`. Operator `&&` ima viši prioritet od operatora `||`, oba su levo asocijativna i lenjo izračunljiva. Pored navednih, *Elm* podržava i operator `++` koji se koristi za konkatenciju niski i listi. Primer koda 5 prikazuje upotrebe navedenih operatora.

```
> 2 > 3 == 3 > 4
-- INFIX PROBLEM - You cannot mix (>) and (==) without parentheses.
> (2 > 3) == (3 > 4)
True : Bool
> not (1 + 1 /= 2) && 2 + 2 <= 5 || 1^0 == 0^1
True : Bool
> 2^6 - 0x100 / 4 * (1 + 2)
-128 : Float
> "Spojena " ++ "niska!" == "Spojena niska!"
True : Bool
```

Primer koda 5: Primeri upotrebe osnovnih operatora

Funkcije

Sintaksa za definisanje funkcija je veoma jednostavna i prikazana je u primeru koda 6.

```
{-
  nazivFunkcije param1 param2 ... =
    izraz
-}
deljivSa x y =
  modBy x y == 0

dobarDan x = "Dobar dan, " ++ x ++ "!"
```

Primer koda 6: Primeri definisanja funkcija

Ime funkcije obavezno počinje malim slovom, nakon čega sledi niz slova (velikih i malih), simbola `_` i brojeva. Po konvenciji, sva slova se navode u neprekidnoj sekvenci, stoga je preporučena kamilja notacija (`camelCase`). Parametri se odvajaju razmakom, dok se zagrade ne navode ni prilikom definisanja, ni pozivanja funkcije. Ipak, primena funkcije je levo asocijativna, pa je česta upotreba zagrada za ograničavanje izraza. Telo funkcije predstavlja jedan jedini izraz koji se izvršava prilikom pozivanja, a izračunata vrednost predstavlja povratnu vrednost funkcije. Izraz se, po konvenciji, piše u novom redu, ali je moguće i u istom.

Funkcije u *Elm*-u mogu prihvatati funkcije kao parametre i vraćati funkcije kao povratne vrednosti, što ih čini funkcijama višeg reda. Nije moguće navoditi podrazumevane vrednosti parametara, kao ni preopterećivanje funkcija.

Konstante

U programskom jeziku *Elm* ne postoje promenljive, jednom definisani podaci se ne mogu promeniti, ali je moguće definisati konstante. Često se u literaturi definisanje konstanti naziva *imenovanjem vrednosti izraza* i ne dovodi se u vezu sa funkcijama, ali se konstante mogu posmatrati kao *konstantne funkcije*, koje se izvrše tokom kompilacije. Definišu se kao i funkcije, ali bez parametara (primer koda 7).

Anonimne funkcije

Anonimne funkcije se definišu slično kao i regularne, umesto imena navodi se simbol `\` koji predstavlja grčko slovo lambda - λ , dok se simbol `->` koristi umesto znaka pridruživanja (primer koda 7).

```
> broj3 = 3
3 : number
> (\x y -> x + y) broj3 4
7 : number
```

Primer koda 7: Primer anonimne funkcije

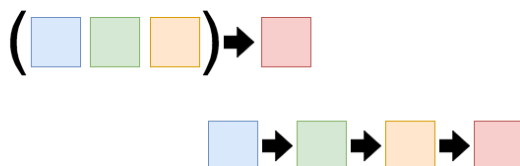
Tip funkcije

Prilikom definisanja funkcije u interpretatoru ili poziva funkcije bez parametara, kao vrednost izraza vraća se `<function>` i tip funkcije. U primeru koda 8 prikazano je nekoliko tipova funkcija u interpretatoru.

```
> not
<function> : Bool -> Bool
> deljivSa
<function> : Int -> Int -> Bool
> \x y z -> x + y + z
<function> : number -> number -> number -> number
> deljivSa 3 --parcijalna primena
<function> : Int -> Bool
```

Primer koda 8: Tipovi funkcija

U primeru funkcije `not` vidimo da je njen tip `Bool -> Bool`, što je na prvi pogled jasno i znači da se radi o funkciji jednog argumenta koja prihvata vrednost tipa `Bool` i vraća vrednost tipa `Bool`. U slučaju funkcije `deljivSa` i anonimne funkcije koja sabira tri broja, tip funkcije se može posmatrati tako da poslednji tip u nizu koji je razdvojen strelicama (`->`) predstavlja povratni tip funkcije, dok tipovi pre njega predstavljaju tipove argumenata funkcije. Tipovi argumenata su takođe odvojeni strelicama jer su sve funkcije u programskom jeziku *Elm* zapravo Karijeve (*Curried*) funkcije (slika 2.3), što znači da su funkcije jednog argumenta koje kao povratnu vrednost imaju funkciju. Strelica (`->`) je desno asocijativna, a zagrade se izostavljaju



Slika 2.3: Regularne i Karijeve funkcije

zbog jednostavnosti. Na primer, funkcija `deljivSa` ima tip `Int -> (Int -> Bool)`, što znači da prihvata vrednost tipa `Int` i vraća funkciju tipa `Int -> Bool`. Karijeve funkcije nam omogućavaju veću fleksibilnost i parcijalnu primenu funkcija, odnosno vezivanje argumenata za konkretne vrednosti (primer koda 8).

Anotacija tipa funkcije

Kao što je prethodno prikazano, *Elm* sam zaključuje tip funkcije, ali dozvoljava i korisniku da sam navede tip u liniji iznad definicije (primer koda 9).


```
> deljivSa: Int -> Int -> Bool
| deljivSa x y =
|   modBy x y == 0
|
<function> : Int -> Int -> Bool
```

Primer koda 9: Anotacija tipa funkcije

Korišćenje anotacije tipova nije obavezno, ali je vrlo preporučljivo iz više razloga. Prilikom kompilacije proverava se poklapanje anotacije sa stvarnim tipom funkcije, što dovodi do lakšeg uočavanja i otklanjanja grešaka. Pored toga, anotacije predstavljaju veoma dobar vid dokumentacije, a činjenica da kompilator uvek poredi navedeni i stvarni tip nam garantuje da je dokumentacija uvek važeća.

Funkcijski operatori

Operatore nad funkcijama možemo podeliti prema tipu, na operatore prosleđivanja i operatore kompozicije funkcija, i prema smeru u kom se primenjuju, unapred ili unazad.

Operatori prosleđivanja ili **pipe** operatori su zapravo operatori primene funkcije i omogućavaju pisanje čitljivijeg koda sa manje zagrada.

- `<|` - pipe operator unazad radi isto što i primena funkcije, s tim što nas oslobađa pisanja zagrada. Preciznije, `f <| x` je drugi zapis za `f (x)`
- `|>` - pipe operator inspirisan je Unix pipe-om, odatle i naziv, i služi za prosleđivanje argumenta funkciji. Preciznije, `x |> f` je drugi zapis za `f (x)`

Operator kompozicije unazad `<<` predstavlja operator matematičke kompozicije funkcija \circ . Tako da se definicija kompozicije dve funkcije, $(g \circ f)(x) = g(f(x))$, u programskom jeziku *Elm* može posmatrati kao: `(g << f) x == (\x_-> g (f x_)) x`. Operator kompozicije `>>` je simetričan operatoru `<<`, stoga važi: `(f << g) x == (g >> f) x`.

Moduli

Moduli se koriste za grupisanje funkcija u logičke jedinice i kreiranje imenskih prostora (eng. *namespace*). Osnovni tipovi, kao i funkcije i operatori nad njima definisani su u modulu **Basics**, koji je podrazumevano uvezen i nalazi se unutar

paketa `elm/core`. Svaki modul predstavlja jednu `.elm` datoteku, koja se mora zvati isto kao i modul, dok ime modula mora počinjati velikim slovom. Za definisanje modula koristi se ključna reč `module` nakon koje sledi ime modula, ključna reč `exposing` i lista funkcija kojima se može pristupiti van modula.

```
module Krug exposing (povrsina, obim)
-- module Krug exposing (..) - otkrivanje svega iz modula
pi = 3.14

povrsina r =
    naKvadrat r * pi

obim r =
    2 * r * pi

naKvadrat x =
    x * x
```

Primer koda 10: Primer modula

Da bi se modul iz primera koda 10 koristio u interpretatoru (`elm repl`), prvo je potrebno inicijalizovati *Elm* projekat (`elm init`) i u direktorijumu `src` napraviti datoteku `Krug.elm` sa prikazanim sadržajem. Zatim, u interpretatoru naredbom `import` treba uvesti modul. Načini korišćenja funkcija iz modula *Krug* prikazani su u primeru koda 11.

```
import Krug                                -- Krug.obim Krug.povrsina
import Krug as K                          -- K.obim K.povrsina

import Krug exposing (obim)                 -- obim, Krug.povrsina
import Krug exposing (..)                   -- obim, povrsina
import Krug as K exposing (povrsina)        -- K.obim, povrsina
```

Primer koda 11: Primer korišćenja modula

Osnovne strukture podataka

Osnovne strukture podataka programskog jezika *Elm* čine liste, torke i slogovi.

Liste

Lista predstavlja kolekciju u obliku jednostruko povezane liste. Elementi se navode unutar uglastih zagrada—`[]` i moraju biti istog tipa. Funkcije za rad sa listama

nalaze se unutar modula `List`, među kojima su i tri veoma važne funkcije višeg reda u funkcionalnoj paradigmi — `map`, `filter` i `fold` (`reduce` ili `accumulate` u drugim programskim jezicima).

Funkcija `map` prihvata funkciju i listu i primenjuje datu funkciju na svaki element liste čime se kreira nova lista. Funkcija `filter` prihvata predikat (funkciju koja vraća logičku vrednost) i listu, a vraća novu listu koja sadži samo elemente koji zadovoljavaju dati predikat. Funkcije `fold` koriste se za „savijanje” liste u jednu vrednost koja se često naziva akumulator. U zavisnosti od smera „savijanja” — da li se ide sa početka ili kraja liste koriste se leva (`foldl`), odnosno desna (`foldr`) funkcija. Funkcije (`fold`) prihvataju binarnu funkciju, početnu vrednost akumulatora i listu i prilikom prolaska kroz elemente liste primenjuje se data funkcija nad trenutnim elementom i akumulatorom čime se kreira nova vrednost akumulatora. Tipovi navedenih funkcija prikazani su u primeru koda 12, a njihova upotreba u primeru koda 13.

```
> List.map
<function> : (a -> b) -> List a -> List b
> List.filter
<function> : (a -> Bool) -> List a -> List a
> List.foldl
<function> : (a -> b -> b) -> b -> List a -> b
```

Primer koda 12: Tipovi funkcija `map`, `filter` i `foldl`

Pored operatora za nadovezivanje `++`, postoji i operator `::` koji dodaje element na početak liste, iz istorijskih razloga ovaj operator naziva se *kons* (eng. *cons*).

```
> "aaa" :: ["bbb", "ccc"]
["aaa", "bbb", "ccc"] : List String
> List.map (List.member 2) [[1,2,3], [2,2], [42]]
[True, True, False] : List Bool
> [1,2]++[3,4,5] |> List.filter (\x -> modBy 2 x == 0) |> List.length
2 : Int
> List.foldl (\acc x -> acc + x) 0 <| List.range 1 5
15 : Int
```

Primer koda 13: Primeri lista različitih tipova i funkcija za rad sa njima

Torke

Za razliku od listi koje mogu imati promenljivi broj elemenata istog tipa, torke predstavljaju kolekcije fiksne dužine čiji elementi ne moraju biti istog tipa. Mogu

sadržati samo dva ili tri elementa, navode se unutar običnih zagrada— (). U torke se ne mogu ubacivati elementi niti se iz torke mogu uklanjati elementi. Funkcije nad torkama koje imaju dva elementa nalaze se u modulu `Tuple`, dok se za rad sa torkama od tri elementa koristi poklapanje obrazaca o kojem će biti više reči kasnije.

```
> (1,"2",'3')
(1,"2",'3') : ( number, String, Char )
> (1,2) == Tuple.pair 1 2
True : Bool
> Tuple.second ("nebitan", "drugi")
"drugi" : String
> Tuple.mapFirst String.length ("mapiran", 1)
(7,1) : ( Int, number )
```

Primer koda 14: Primeri torki i upotreba funkcija iz modula `Tuple`

Slogovi

Slog (eng. *Record*) predstavlja strukturu podataka koja može sadržati više vrednosti različitih tipova, pri čemu je svakoj vrednosti dodeljen naziv. Liče na *JavaScript* objekte, čak je i sintaksa veoma slična, umesto dvotačke slog koristi znak jednakosti za dodelu naziva. Prilikom definisanja sloga, *Elm* kreira funkcije za pristup njegovim svojstvima. Nije moguće dodavanje, ni uklanjanje svojstava, ali je dozvoljena promena njihovih vrednosti. Zbog imutabilnosti, ne vrši se promena nad postojećim slogom već se pravi novi.

```
> pera = {ime = "Pera", prezime = "Perić", godine = 23}
{ godine = 23, ime = "Pera", prezime = "Perić" }
  : { godine : number, ime : String, prezime : String }
> {pera | prezime = "Petrović", godine = 24}
{ godine = 24, ime = "Pera", prezime = "Petrović" }
  : { godine : number, ime : String, prezime : String }
> pera.ime
"Pera" : String
> .godine pera
23 : number
> .prezime
<function> : { b | prezime : a } -> a
```

Primer koda 15: Primeri pristupa i promene svojstava sloga

Pored navedenih struktura podataka, *Elm* pruža podršku za rad sa nizovima (`Array`), skupovima (`Set`) i rečnicima (`Dict`).

Tipske promenljive

U primeru koda 15 vidimo da funkcija za pristup prezimenu ima tip: `{ b | prezime : a } -> a`. Ovo znači da funkcija kao argument prima slog, koji može biti bilo kog tipa, ali mora imati svojstvo `prezime`, koje takođe može biti bilo kog tipa, i čiji tip je ujedno povratni tip funkcije. Promenljive `a` i `b` nazivaju se *tipske promenljive*, a prisustvo dve tipske promenljive nam govori da one mogu, ali ne moraju, predstavljati različite tipove. U konkretnom primeru `a` i `b` su uvek različitog tipa, dok u slučaju funkcije `Tuple.pair : a -> b -> (a, b)` mogu biti istog tipa. Prilikom anotacije tipova mogu se koristiti i duža imena tipskih promenljivih, a pravila imenovanja su ista kao i za funkcije.

Tipske promenljive u programskom jeziku *Elm* ukazuju na prisustvo **parametarskog polimorfizma**, jedine vrste polimorfizma u ovom jeziku.

Uslovne tipske promenljive

Za razliku od programskog jezika *Haskell*, *Elm* nema toliko složen sistem tipova i umesto tipskih klasa (eng. *typeclasses*)[7] poseduje jednostavniji koncept — *uslovne tipske promenljive*.

Uslovne tipske promenljive omogućavaju da se na određni način ograniči skup tipova koji se može koristiti u izrazima. Najčešći primer je `number`, koji dozvoljava isključivo `Int` ili `Float` tipve.

U trenutnoj verziji (*0.19.1*) postoje četiri uslovne tipske promenljive:

1. `number` — dozvoljava tipove `Int` i `Float`
2. `appendable` — dozvoljava tipove `String` i `List a`
3. `comparable` — dozvoljava tipove `Int`, `Float`, `Char`, `String`, liste i torke koje sadrže `comparable` vrednosti
4. `compappend` — dozvoljava tipove `String` i `List comparable`

Operatori i tipske promenljive

```
> (+) 1 2
3 : number
> (*)
<function> : number -> number -> number
> (++)
<function> : appendable -> appendable -> appendable
> (==)
<function> : a -> a -> Bool
> (>=)
<function> : comparable -> comparable -> Bool
```

Primer koda 16: Upotreba operatora u prefiksnoj notaciji

Operatori predstavljaju funkcije koje se mogu pozivati u infiksnoj notaciji. Takođe, mogu se pozivati i u prefiksnoj ukoliko ih navedemo unutar zagrada: `(+)`, `(++)`, `(>=)`... , dok pozivanjem bez argumenata možemo videti i kog su tipa (primer koda 16).

U ranijim verzijama jezika bilo je moguće definisati korisničke operatore, ali je ta opcija izbačena u verziji *0.19.0*, a od verzije *0.18.0* nije moguće pozivanje binarnih funkcija u infiksnoj notaciji.

Alijasi tipova

Prilikom definisanja funkcija koje rade nad istim strukturama podataka višestruko navodimo iste anotacije tipova, pritom anotacije podataka mogu biti predugačke, samim tim i teško čitljive. Alijasi tipova nam omogućavaju ponovnu upotrebu anotacija i bolju čitljivost. Definišu se pomoću ključnih reči `type alias`, nakon kojih sledi ime koje mora počinjati velikim slovom.

```
>type alias MatricaInt3 = List (List (List Int))
>
>type alias Osoba = {ime : String, prezime : String, godine : Int }
> Osoba
<function> : String -> String -> Int -> Osoba
> Osoba "Pera" "Perić" 23
{ godine = 23, ime = "Pera", prezime = "Perić" } : Osoba
```

Primer koda 17: Primeri definisanja alijasa tipova

Prilikom kreiranja alijasa za slog kreira se i konstruktor za slog, što se može videti u primeru koda 17. Redosled argumenata funkcije za konstrukciju identičan je redosledu u alijasu.

Korisnički definisani tipovi

Pored korišćenja alijasa za postojeće tipove podataka, *Elm* pruža mogućnost kreiranja novih tipova. Korisnički definisani tipovi se često nazivaju algebarskim tipovima (eng. *algebraic datatype*), kao i *unijskim* tipovima, jer mogu predstavljati uniju više varijanti definisanog tipa. Definišu se ključnom reči `type`, a varijante se odvajaju simbolom `|`.

Slično kao kod alijasa tipova za slogove i ovde se kreiraju konstruktori za definisani tip. Bitna razlika u odnosu na alijase tipova je ta što algebarski tipovi mogu biti rekurzivni, tj. tip koji se definiše se može navoditi unutar svoje definicije. U primeru koda 18 se mogu videti primeri definisanja algebarskih tipova i tipovi njihovih konstruktora. Takođe je prikazana i definicija binarnog stabla kao primer rekurzivne strukture.

```
> type VectorF4 = Vector4F Float Float Float Float
> Vector4F
<function> : Float -> Float -> Float -> Float -> VectorF4
>
> type StatusPrijave
  = NaCekanju
  | Greska String
  | Uspesno {id : Int, token : String}
> NaCekanju
NaCekanju : StatusPrijave
> Greska
<function> : String -> StatusPrijave
> Uspesno
<function> : { id : Int, token : String } -> StatusPrijave
> type BinarnoStablo a
  = PraznoStablo
  | CvorStabla a (BinarnoStablo a) (BinarnoStablo a)
```

Primer koda 18: Primeri korisnički definisanih tipova

Kontrola toka

Elm ne izvršava naredbe, već evaluira izraze, tako da umesto naredbi grananja imamo izraze `if` i `case`, a funkcije višeg reda ili rekurziju umesto petlji.

Izraz if

Izraz `if` se može posmatrati kao ternarni operator u programskim jezicima *JavaScript*, *C++* i mnogim drugim.

```
{-      uslov    ?  izraz1    :  izraz2    - ternarni operator
      if uslov then izraz1 else izraz2    - if izraz
-}
if x >= 0 then "pozitivan" else "negativan"
if x == 1 then x * 2 else if x == 2 then x / 2 else x
```

Primer koda 19: Sintaksa izraz `if` i primer upotrebe

Ključna reč `else` je sastavni deo izraza `if` tako da `else` „grana” uvek postoji. Mogu se navoditi i ugnježdeni izrazi `if`, pa `else if` grana predstavlja korišćenje izraza `if` nakon ključne reči `else`.

Izraz case

Za razliku od izraza `if`, izraz `case` omogućava grananje na osnovu šireg raspona vrednosti, umesto samo sa *tačno* ili *netačno*. Takođe, pruža mogućnost navođenja više grana. Primer upotrebe izraza `case` prikazan je u primeru koda 20. Vrednost navedena nakon ključne reči `case` se redom upoređuje sa navedenim opcijama i prvim poklapanjem se određuje vrednost izraza.

```
case mesto of
  1 -> "zlato"
  2 -> "srebro"
  3 -> "bronza"
  _ -> "zahvalnica"
```

Primer koda 20: Primer upotrebe izraza `case`

Prilikom korišćenja izraza `case` moraju se pokriti sve mogućnosti, ukoliko to nije slučaj kompilator prijavljuje grešku. Za podrazumevani slučaj može se koristiti simbol `_`. Izraz `case` zauzima značajno mesto u programskom jeziku *Elm*, jer se koristi u **poklapanju obrazaca**.

Izraz let

Budući da ne postoje blokovi naredbi, izrazi `let` nam omogućavaju da ograničimo oblast važenja — dosega (eng. *scope*) funkcija i konstanti u okviru jedne funkcije. Doprinosе boljoj čitljivosti koda, a moguće je koristiti anotacije tipova unutar njih.


```
let
  nula: Int
  nula = 0

  pozitivan: Int -> Bool
  pozitivan =
    \x -> x > nula
in pozitivan 10
```

Primer koda 21: Primer upotrebe izraza `let`

Rekurzija

Rekurzivno definisana funkcija poziva samu sebe, čime se postiže ponavljanje izvršavanja koje se u imperativnom programiranju ostvaruje korišćenjem petlji (ili takođe rekurzijom). U mnogim slučajevima, umesto rekurzije mogu se koristiti funkcije višeg reda i treba ih upotrebiti kad god je to moguće.

```
factorial n =
  if n <= 1 then 1
  else n * factorial (n - 1)

factorialFold n =
  List.foldl (*) 1 (List.range 1 n)
```

Primer koda 22: Upotreba rekurzije i funkcije `foldl` za iteraciju kroz listu

Poklapanje obrazaca

Poklapanje obrazaca (eng. *pattern matching*) može se posmatrati kao pokušavanje usklađivanja (poklapanja) ulaznog podatka sa unapred definisanim obrascem. Ukoliko dođe do usklađivanja, poklopljenim vrednostima se može pristupiti putem identifikatora definisanim u obrascu. Pored spomenutih `case` izraza, poklapanje obrazaca može se koristiti u vidu razlaganja slogova ili torki prilikom definisanja funkcija ili korišćenja `let` izraza.

Unutar `case` izraza sekvencijalno se vrši poklapanje obrazaca. Kada dođe do poklapanja izračunava se izraz dodeljen datom obrascu, ne nastavlja se sa poklapanjem. Kompilator prepoznaje ukoliko može doći do nepoklapanja nijednog obrasca i prijavljuje grešku. Takođe, greška se prijavljuje i ukoliko se navede redundantan obrazac, tj. obrazac čiji se skup vrednosti preklapa sa skupom vrednosti prethodno definisanog obrasca. Simbol `_` služi za poklapanje vrednosti koje se ne koriste, a ključna

```
-- liste
case lista of
  [] -> "prazna lista"
  [_] -> "jedan element"
  [a,b] -> "dva elementa: " ++ a ++ " i " ++ b
  a :: _ -> "više od dva elemenata, prvi je: " ++ a

-- unijski tipovi
case prijava of
  NaCekanju -> "Molimo za strpljive"
  Greska poruka -> "Došlo je do grške: " ++ poruka
  Uspesno {id} -> "Uspešna prijava, id: " ++ String.fromInt id

-- torke
case tacka3D of
  (0, 0, 0) -> "centar"
  (0, _, _) -> "na x-osi"
  _ -> "van x-ose"

-- razlaganje
let
  (x,_,_) = tacka3D
in "x koordinata je " ++ String.fromFloat x

--nije moguće poklapanje ugnježđenih slogova
prikaziPodatke ({ime, adresa} as osoba) =
  ime ++ " " ++ osoba.prezime ++ " " ++ adresa.ulica
```

Primer koda 23: Primeri poklapanja obrazaca

reč `as` se može koristiti ukoliko je potrebno pristupiti celom ulaznom podatku. U primeru koda 23 mogu se videti primeri poklapanja obrazaca.

Obrada grešaka pomoću `Maybe` i `Result`

Unutar programskog jezika *Elm* ne postoje `try catch` blokovi, kao ni `undefined`, `null`, `nil` i ostale slične vrednosti prisutne u drugim programskim jezicima. Umešto njih koriste se `Maybe` i `Result` koji potiču iz programskog jezika *Haskell*, gde imamo `Maybe` i `Either`.

Maybe

U slučaju da je potrebno napisati funkciju koja vraća prvi element liste, ukoliko on postoji, rezultat bi bio **baš** prvi element. Dok u slučaju prazne liste, funkcija ne bi vratila **ništa**. Upravo tako radi funkcija `List.head : List a -> Maybe a`, ukoliko se pozove **možda** vrati prvi element.

`Maybe` se definiše kao `type Maybe a = Just a | Nothing` i može se koristiti za opcione argumente, obradu grešaka i u slogovima sa opcionim svojstvima. Zapravo svuda gde očekivani podatak može, ali ne mora, postojati.

Result

Za razliku od `Maybe`, koji bi u slučaju greške vratio `Nothing`, `Result` nam daje mogućnost pružanja dodatnih informacija o grešci. Definiše se na sledeći način:

```
type Result error value
  = Ok value
  | Err error
```

2.6 Arhitektura Elm

Arhitektura Elm je veoma jednostavan obrazac projektovanja veb aplikacija koji se pojavljuje ubrzo nakon nastanka samog jezika. Nastaje prirodno, tako što se već u prvim *Elm* programima uočavaju tri osnovne celine:

1. Model - stanje aplikacije
2. Pogled (eng. *View*) - transformacija stanja u *HTML*
3. Ažuriranje (eng. *Update*) - promena stanja

U levom delu slike 2.4 prikazano je kako radi svaki *Elm* program: generiše se *HTML* koji se prikazuje u pretraživaču, nakon čega pretraživač šalje poruku programu ukoliko se nešto dogodilo. U desnom delu slike vidimo šta se dešava unutar programa, na osnovu primljene poruke funkcija `update` kreira novi `model`, koji se prosleđuje funkciji `view`, na osnovu koje se generiše *HTML*.

Ovaj obrazac projektovanja ce često naziva i *MVU* (eng. *Model-View-Update*). Za razliku od obrazaca *MVC* [18] (eng. *Model-View-Controller*) i *MVVM* [18] (eng.

Slika 2.4: Arhitektura *Elm*

Model-View-ViewModel), koji stanje aplikacije dele na više manjih modela, u arhitekturi *Elm* celokupno stanje aplikacije se nalazi na jednom mestu, tj. modelu, a protok podataka kroz aplikaciju je uvek u jednom smeru.

Arhitektura *Elm* uticala je na nastajanje velikog broja biblioteka za upravljanje stanjem veb aplikacije, među kojima su najpoznatije *Redux*[2] i *Vuex*[3]. Takođe, podrška za rad sa obrascem *MVU* počinje da se pojavljuje i u drugim tehnologijama, jedna od njih je i *.NET 5* [15].

Generisanje HTML sadržaja

Za razliku od drugih razvojnih okvira koji uglavnom koriste *HTML* šablone (eng. *templates*), *Elm* za opisivanje izgleda stranice koristi funkcije. Stoga imamo funkcije za kreiranje *HTML* čvorova i atributa.

```
-- <button id="btn-ok" class="btn-default"> Ok <\button>
node "button" [id "btn-ok", class "btn-default"] [text "Ok"]
-- korišćenje pomoćne funkcije button umesto node "button"
  button [id "btn-ok", class "btn-default"] [text "Ok"]
```

Primer koda 24: Primeri kreiranja *HTML* čvorova

Funkcija `node` iz modula `Html` predstavlja generičku funkciju za kreiranje *HTML* čvorova, koja kao argumente prima *HTML* oznaku, listu atributa i listu čvorova (deca čvora). U primeru koda 24 predstavljena je analogija kreiranja *HTML* čvora funkcijom `node` i *HTML* sintakse. Modul `Html` pruža veliki broj pomoćnih funkcija, koje imaju nazive po *HTML* oznakama (npr. `button` — primer koda 24) i omogućavaju bolju čitljivost. Stoga se funkcija `node` koristi prilikom upotrebe korisnički definisanih elemenata ili ukoliko ne postoji pomoćna funkcija za željeni element.

Funkcija `text` služi za postavljanje teksta u DOM, dok su `id` i `class` funkcije za kreiranje atributa iz modula `Html.Attributes`.

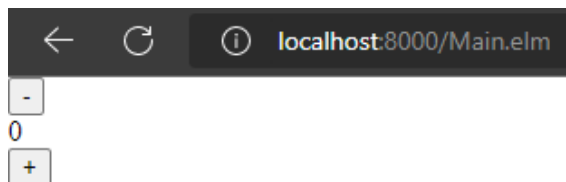
Elm program

Prilikom inicijalizacije *Elm* projekta se, pored paketa `elm/core` sa osnovnim funkcionalnostima i strukturama podataka i paketa `elm/html` za generisanje *HTML* stranica, nalazi paket `elm/browser` za kreiranje *Elm* programa u pretraživaču. Unutar ovog paketa, u modulu `Browser` se nalaze funkcije koje nam omogućavaju kreiranje različitih tipova programa, među kojima je i funkcija `sandbox` namenjena za učenje osnova arhitekture *Elm* i omogućava bazičnu interakciju sa korisnicima, bez komunikacije sa spoljnim svetom. Primer jednostavnog *Elm* programa upotrebom funkcije `sandbox` prikazan je u primeru koda 25.

Da bi okruženje *Elm* znalo koja je polazna tačka programa potrebno je da se u glavnom modulu definiše i izloži konstanta `main`, dok naziv modula nije bitan. Pored funkcija iz modula `Browser`, u slučaju statičkih stranica, konstanta `main` se može definisati funkcijama iz modula `Html`, na primer:

```
main = text "Zdravo svete!"
```

Da bi se pokrenuo program potrebno je pozicionirati se u komandnoj liniji na inicijalizovani *Elm* projekat i pokrenuti `elm reactor`. Nakon toga, potrebno je otvoriti pretraživač na adresi `http://localhost:8000` i unutar direktorijuma `src` kliknuti na `Main.elm` čime se vrši kompilacija i pokretanje programa. Takođe, primer se može naći i na zvaničnom *Elm* vodiču[19]. Na slici 2.5 prikazan je izgled programa u pregledaču.



Slika 2.5: Izgled programa u pregledaču

U primeru jednostavnog brojača iz primera koda 25 vidimo da funkcija `sandbox` kao argument prima slog sa inicijalnom vrednošću modela i funkcijama za ažuriranje i prikazivanje modela. Program počinje pozivanjem funkcije `view` sa parametrom `init`. Unutar funkcije `view` se pomoću funkcije za kreiranje atributa iz modula

```
module Main exposing (main)

import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
    Browser.sandbox { init = init, update = update, view = view }

--MODEL
type alias Model = Int

init : Model
init =
    0

-- UPDATE
type Msg
    = Increment
    | Decrement

update : Msg -> Model -> Model
update msg model =
    case msg of
        Increment ->
            model + 1

        Decrement ->
            model - 1

-- VIEW
view : Model -> Html Msg
view model =
    div []
        [ button [ onClick Decrement ] [ text "-" ]
        , div [] [ text (String.fromInt model) ]
        , button [ onClick Increment ] [ text "+" ]
        ]
```

Primer koda 25: Primer *Elm* programa

`Html.Events` mogu definisati načini slanja poruka, a kao rezultat izvršavanja nastaje virtuelni *DOM*, na osnovu kog okruženje *Elm* izmenjuje stvarni *DOM*. Ukoliko dođe

do odgovarajuće akcije korisnika, u ovom slučaju klikom na dugme, okruženje *Elm* generiše poruku i prosleđuje je zajedno sa modelom funkciji `update` koja kreira novi model nad kojim se poziva ponovo funkcija `view`. Okruženje *Elm* poredi prethodni virtuelni *DOM* sa novim i vrši minimalan broj izmena.

Glava 3

Programski jezik Elixir

Osnovne odlike jezika *Elixir* sažete su u okviru njegove zvanične strane [32]:

Elixir je dinamički tipiziran, funkcionalni programski jezik dizajniran za izgradnju skalabilnih aplikacija, lakih za održavanje.

Elixir koristi virtuelnu mašinu programskog jezika *Erlang*, poznatu po podršci za rad sistema sa malim kašnjenjem, distribuiranih sistema i sistemima koji su otporni na greške, ali se uspešno koristi i u razvoju veba, u sistemima sa ugrađenim računarom (eng. *embedded software*), učitavanju podataka (eng. *data ingestion*) i u domenima za obradu multimedije.

Prva verzija programskog jezika *Elixir* objavljena je 2011. godine. Autor jezika, Žozé Valim (José Valim), prethodno je radio kao programer u programskom jeziku *Ruby* i uvideo je probleme rada razvojnog okvira *Ruby on Rails* na višejezgarnim sistemima. Rešenje problema video je u novom funkcionalnom programskom jeziku koji će koristiti prednosti virtuelne mašine jezika *Erlang*. Stoga, najveći uticaj na razvoj programskog jezika *Elixir* imali su *Erlang*, sa kojim je semantički vrlo sličan, i *Ruby* u smislu sintakse.

Ruby je dinamički tipiziran programski jezik opšte namene koji pruža mogućnost rada u više programskih paradigmi. Nastao je sredinom devedesetih godina prošlog veka, a razvio ga je japanski naučnik i programer Jukihiro Macumoto (*Yukihiro Matsumoto*). *Ruby* je dizajniran kao programski jezik fokusiran na programera, tako da svojom lakoćom korišćenja, jednostavnošću i fleksibilnošću učini programiranje prijatnijim. Sam autor kaže da pokušava da napravi *Ruby* što prirodnijim.

Budući da je *Elixir* nastao na ideji upotrebe virtuelne mašine jezika *Erlang*, više o programskom jeziku *Erlang* biće rečeno u sledećem poglavlju. Takođe, po-

red dva navedena programska jezika, uticaj na razvoj *Elixir*-a imali su *Clojure* [9], *Haskell*[12] i *Python*[13].

3.1 Programski jezik i razvojna platforma Erlang

Erlang nije samo programski jezik, već i razvojna platforma za izgradnju skalabilnih i pouzdanih sistema koji gotovo neprestano pružaju usluge. Švedska telekomunikaciona kompanija Erikson (eng. *Ericsson*) osmislila je ovu platformu sredinom osamdesetih godina prošlog veka. Da bi *Erlang* mogao da upravlja telekomunikacionim sistemima kompanije, morao je da bude pouzdan, skalabilan, da ima brz odziv i bude konstantno dostupan, jer telefonska mreža mora da funkcioniše bez obzira na broj istovremenih poziva i neočekivane greške.

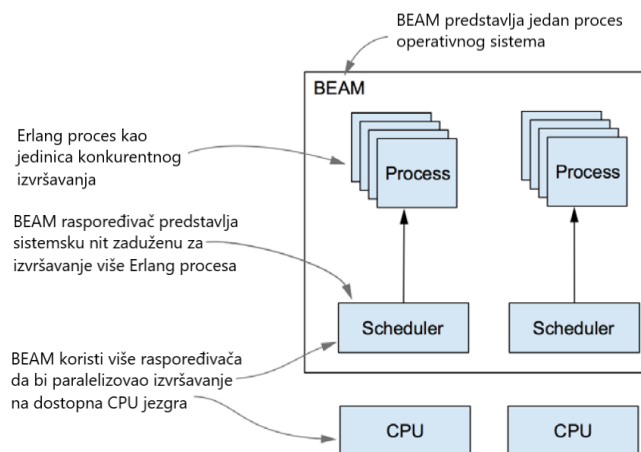
Iako je prvobitno izgrađen za telekomunikacione sisteme, *Erlang* ni na koji način nije specijalizovan za ovaj domen, ne sadrži podršku za programiranje telefona i drugih telekomunikacionih uređaja. *Erlang* predstavlja razvojnu platformu koja pruža posebnu podršku tehničkim zahtevima kao što su: paralelnost, skalabilnost, tolerancija na greške, distribuiranost i velika dostupnost. U vreme njegovog nastajanja, programi su se uglavnom koristili bez komunikacije sa nekim veb servisom (eng. *desktop-based sotfware*), pa je upotreba *Erlang* jezika bila ograničena na telekomunikacione sisteme. Međutim, zahtevi modernih sistema i aplikacija poklapaju se sa tehničkim zahtevima za koje *Erlang* pruža podršku, tako da je u poslednje vreme privukao veliku pažnju. *Erlang* pokreće razne velike sisteme, među kojima su aplikacije *WhatsApp*[29] i *WeChat*[28], distribuirana baza podataka *Riak*[31] i *Heroku Cloud*[8].

Kao razvojna platforma, *Erlang* se sastoji od jezika, virtuelne mašine, razvojnog okvira (eng. *framework*) i alata.

Virtuelna mašina Erlang — BEAM

Jezik *Erlang* predstavlja primaran način pisanja koda koji se izvršava na sopstvenoj virtuelnoj mašini koja se naziva *BEAM*. *BEAM* je skraćenica za *Bogdan's Erlang Abstract Machine*, po Bogumilu Bogdanu Hausmanu koji je kreirao originalnu verziju, ali se može posmatrati i kao *Björn's Erlang Abstract Machine*, po Bjornu Gustavsonu koji održava trenutnu verziju virtuelne mašine.

Kôd napisan u programskom jeziku *Erlang* se prevodi u binarni kôd (eng. *byte-code*) koji se unutar *BEAM*-a izvršava paralelno u vidu veoma lakih *Erlang procesa*. *BEAM*, umesto da se oslanja na procese i niti operativnog sistema, samostalno raspoređuje konkurentne *Erlang procese* na dostupna procesorska jezgra, što je prikazano na slici 3.1. Ovi laki procesi su međusobno potpuno izolovani, ne dele memoriju i



Slika 3.1: Način izvršavanja Erlang procesa unutar BEAM-a [11]

komuniciraju preko asinhronih poruka, što pruža mogućnost *Erlang* sistemima da budu skalabilni, distribuirani i otporni na greške.

Razvojni okvir OTP

OTP je skraćenica za *Open Telecom Platform*, što je donekle pogrešan naziv za ovaj razvojni okvir. On nije vezan za telekomunikacione sisteme, već predstavlja razvojni okvir opšte namene koji apstrahuje tipične zadatke *Erlang* sistema:

- Obrasce konkurentnosti i distribuiranosti
- Detekciju i oporavak od grešaka u konkurentnim sistemima
- Pravljenje biblioteka
- Postavljanje (eng. *deployment*) sistema
- Uživo ažuriranje softvera

OTP predstavlja sastavni deo *Erlang*-a, stoga i zvanična distribucija nosi naziv *Erlang/OTP*.

Odnos programskih jezika Erlang i Elixir

Elixir predstavlja alternativni način pisanja programa koji se izvršava na virtualnoj mašini BEAM i samim tim preuzima sve osobine platforme Erlang. Za razliku od Erlang-a pruža dodatne koncepte koji omogućavaju značajno smanjenje ponavljajućeg i šablonskog (eng. *boilerplate*) koda. *Elixir* i *Erlang* su kompatibilni, pa *Elixir* može direktno da koristi *Erlang* biblioteke i module. Važi i obrnuto, a sve što je moguće implementirati u programskom jeziku *Erlang*, moguće je u programskom jeziku *Elixir* bez razlike u performansama. Takođe, važno je napomenuti da oba jezika odlikuje nepromenljivost podataka (imutabilnost).

3.2 Uputstvo za instalaciju

Proces instalacije je vrlo jednostavan, potrebno je pratiti uputstva sa zvanične *Elixir* stranice [32] za odgovarajući operativni sistem. Podržani su gotovo svi operativni sistemi. Provera uspešne instalacije može se izvršiti pokretanjem komande `elixir --version` u komandnoj liniji ili pokretanjem interaktivnog *Elixir*-a komandom `iex`.

3.3 Osnovne odlike

Za razliku od programskog jezika *Elm*, *Elixir* nije čist funkcionalni programski jezik, jer dozvoljava bočne efekte. Pored prethodno spomenutih karakteristika preuzetih od *Erlang*-a, dinamičke tipiziranosti i sintakse slične *Ruby*-ju, *Elixir* odlikuju izraženo poklapanje obrazaca, polimorfizam, makroi i metaprogramiranje, kao i podrška za lenjo izračunavanje.

3.4 Osnove jezika Elixir

U ovoj glavi predstavljeni su osnovni operatori i tipovi podataka, poklapanje obrazaca, definisanje i grupisanje funkcija u module.

Osnovni tipovi podataka

Osnovne tipove podataka predstavljaju brojevi (celobrojni i realni), logičke vrednosti, atomi i niske. Takođe, u osnovne tipove podataka jezika *Elixir* ubrajaju se

liste, torke i mape, ali će one biti objašnjene u poglavlju o strukturama podataka. Pregled osnovnih tipova dat je u primeru koda 26, gde se može videti i način pisanja komentara. *Elixir* podržava samo linijske komentare koji se označavaju simbolom #.

```
iex> 1           # integer
iex> 1.0         # float
iex> true        # boolean
iex> :atom       # atom
iex> "elixir"    # string
```

Primer koda 26: Pregled osnovnih tipova podataka u *Elixir*-u

Celi brojevi

Celi brojevi su još jedna razlika programskog jezika *Elixir* u odnosu na *Elm* u smislu da nemaju ograničen opseg vrednosti. Takođe, pored heksadecimalnog zapisa, *Elixir* pruža mogućnost navođenja celobrojnih vrednosti u binarnom i oktalnom obliku. Simbol `_` može se koristiti za odvajanje cifara i kod celih i kod brojeva u pokretnom zarezu (primer koda 27).

Brojevi u pokretnom zarezu

Brojevi u pokretnom zarezu su dvostruke preciznosti (64 bita) i delimično prate standard *IEEE-754*, jer ne postoji podrška za vrednosti `NaN` i `Infinity`. Vrednosti brojeva se mogu navoditi i u eksponencijalnom zapisu.

```
iex> 10_000
10000
iex> 0o77
63
iex> 0b111_111
63
iex> 0xFF
255
iex> 1_000_000.000_123
1000000.000123
iex> 1.23e-3
0.00123
```

Primer koda 27: Primeri celih i realnih brojeva u *Elixir*-u

Atomi

Atom je konstanta čiju vrednost predstavlja njeno ime. Definiše se dvotačkom (:), nakon koje slede alfanumerički karakteri, simboli @ ili _, a može se završavati znakom pitanja ili uzvika. Takođe, moguće je koristiti i razmake, tada se sadržaj posle dvotačke mora navesti unutar navodnika. Postoji i notacija atoma bez početne dvotačke, ali u tom slučaju moraju počinjati velikim slovom, takvi atomi se nazivaju *alijasi* i tokom kompilacije se transformišu u atome oblika `:’Elixir.<naziv aliasa>’`. Upotreba atoma prikazana je u primeru koda 28.

Atom se sastoji od teksta i vrednosti. U toku izvršavanja tekst se čuva u *tabeli atoma*, dok vrednost predstavlja referencu na tekst u tabeli atoma, što omogućava brže poređenje atoma kao i veću uštedu memorije. Obzirom da predstavljaju efikasan način za imenovanje konstanti, atomi su svuda prisutni.

```
iex> :neka_vrednost
:neka_vrednost
iex> "atom sa razmacima"
:"atom sa razmacima"
iex> AliasAtom # kompilacijom se transformiše u :’Elixir.AliasAtom’
AliasAtom
iex> AliasAtom == :’Elixir.AliasAtom’
true
```

Primer koda 28: Primeri atoma u *Elixir*-u

Logičke vrednosti kao atomi *Elixir* zapravo nema poseban tip za logičke vrednosti, već koristi atome `:true` i `:false`, i pruža mogućnost njihovog navođenja bez početne dvotačke, tj. samo sa `true` i `false`. *Elixir* pruža funkcije za proveru tipa podataka, među kojima su `is_atom` i `is_boolean` i čija upotreba je predstavljena u primeru koda 29.

```
iex> is_atom(true)
true
iex> is_boolean(:true)
true
iex> false == :false
true
```

Primer koda 29: Predstavljanje logičkih vrednosti preko atoma

Pored logičkih vrednosti, jos jedna specifična vrednost predstavljena atomom jeste `:nil`, slična vrednosti `null` u drugim programskim jezicima. Ona se takođe može navoditi bez početne dvotačke (sa `nil`).

Niske

Niske (`string`) se navode slično kao u *Elm*-u, unutar jednostrukih ili trostrukih navodnika, s tim da se u slučaju trostrukih navodnika oni moraju navesti u posebnim redovima. U *Elixir*-u, niske koriste *UTF-8* kodiranje i pružaju mogućnost ugrađivanja izraza — interpolaciju niski, što se može videti u primeru koda 30.

```
iex> "Čao!"
"Čao!"
iex(1)> ""
...(1)> Niska u više
...(1)> redova.
...(1)> ""
"Niska u vise\nredova.\n"
iex> "1+1=#{1+1}"
"1+1=2"
```

Primer koda 30: Primeri niski u *Elixir*-u

Niske bitova i sekvence bajtova Kao i kod logičkih vrednosti, *Elixir* nema poseban tip za niske, već za njihovo predstavljanje koristi sekvencu bajtova (eng. *binary*) i *UTF-8* kodiranje. Zapravo fundamentalni tip za njihovo predstavljanje jeste niska bitova — **bitstring**, kojoj odgovara neprekidni niz bitova u memoriji. Niske bitova mogu biti proizvoljne dužine, a sekvenca bajtova — *binary* predstavlja posebnu vrstu niske bitova čiji je broj bitova deljiv sa osam.

Niske bitova se navode kao sekvenca brojeva unutar znakova `<< i >>`. Podrazumevano se koristi osam bitova za čuvanje svakog broja u bitstringu, ali je moguće navesti broj bitova pomoću modifikatora `::n` da bi se označila veličina od `n` bitova, ili u opširnijoj notaciji `::size(n)`. U primeru koda 31 vidimo da ukoliko sadržaj niske bitova čine samo ASCII karakteri, interaktivni *Elixir* ispisuje nisku. *Elixir* nema tip **Char** za rad sa karakterima, ali daje mogućnost određivanja koda pomoću simbola `?`, nakon kog se navodi karakter.

```
iex> is_binary(<<1,2>>)
true
iex> is_bitstring(<<1::1>>) # 1 bit
true
iex> is_binary(<<1::1,2>>) # 9 bitova
false
iex> <<1::1,0::1,1::1>>
<<5::size(3)>>
iex> is_binary("neki string")
true
iex> byte_size("Ćao") # Ć zauzima 2 bajta
4
iex> ?a # određivanje koda karakter a
97
iex> <<97,98,99>>
"abc"
```

Primer koda 31: Predstavljane niski kao niz bajtova

Drugi ugrađeni tipovi podataka

Pored navedenih, *Elixir* poseduje i druge ugrađene tipove podataka:

- Referenca — jedinstven podatak u jednoj BEAM instanci i jedinstvenost je zagarantovana samo tokom životnog veka te instance
- Identifikator procesa (*pid*) — koristi se za identifikaciju *Erlang* procesa
- Identifikator porta — koristi se za identifikaciju porta, mehanizma koji se koristi za komunikaciju sa spoljnim svetom (datotekama, programima...)

Osnovni operatori

Za razliku od *Elm*-a, *Elixir* vrši implicitnu konverziju celih brojeva u realne prilikom primene aritmetičkih operatora +, - i * ukoliko operandi nisu istog tipa, dok se u slučaju operatora / ona uvek vrši, tako da je rezultat deljenja uvek realan broj. Za celobrojno deljenje i izračunavanje ostatka pri celobrojnem deljenju koriste se funkcije `div` i `rem`, koje se mogu koristiti samo nad celim brojevima.

Relacijski operatori u *Elixir*-u slični su kao i u drugim jezicima. S tim što pored operatora `==`, `!=`, `<=`, `>=`, `< i >`, postoje i operatori `===` i `!==` koji se od operatora `==` i `!=` razlikuju samo prilikom poređenja brojeva, tako što ne vrše konverziju tipova, tj. vrše poređenje po tipu i vrednosti. Upotreba operatora poređenja prikazana je u

primeru koda 32. Takođe, u primeru koda 32 vidimo da *Elixir* dozvoljava poređenje

```
iex> 1 == 1.0
true
iex> 1 === 1.0
false
iex> 1 == "1"
false
iex> 1 < false and "12" > '123'
true
```

Primer koda 32: Rad sa operatorima poređenja

vrednosti različitih tipova i to po sledećem uređenju : `number < atom < reference < function < port < pid < tuple < map < list < bitstring`

Logički operatori u *Elixir*-u su specifični po tome što rezultat njihove primene ne mora biti logička vrednost, već može biti vrednost proizvoljnog tipa. Takođe, *Elixir* ima dvostruke operatore za logičko *i*, *ili* i *ne*, koji se mogu podeliti u dve grupe:

1. `and`, `or` i `not` — prvi operand mora biti logička vrednost
2. `&&`, `||` i `!` — oba operanda mogu biti proizvoljnog tipa

Operatori čiji operandi mogu biti proizvoljnog tipa zasnivaju se na konceptu istinitosti, gde se vrednosti `nil` i `false` smatraju za lažne, a sve ostale za istinite. Operatori su lenjo izračunljivi, tako da u slučaju primene operatora konjukcije rezultat predstavlja drugi operand samo ako je prvi istinit ili se može smatrati istinitim u slučaju operatora `&&`. Dok prilikom primene operatora disjunkcije rezultat je prvi operand ukoliko je tačan ili se može smatrati tačnim (operator `||`), a u suprotnom vraća se vrednost drugog operanda. Rad sa logičkim operatorima prikazan je u primeru koda 33.

Funkcije i moduli

Elixir, kao i *Elm*, poseduje imenovane i anonimne funkcije. Imenovane funkcije se navode isključivo unutar modula, a sintaksa je veoma jednostavna i prikazana je u primeru koda 34.

Za razliku od *Elm*-a, gde funkcija predstavlja izvršavanje jednog izraza, u *Elixir*-u je moguće izvršiti više njih, a poslednji izraz predstavlja povratnu vrednost. U slučaju jednog izraza unutar funkcije može se koristiti i skraćena sintaksa —


```
iex> true and true
true
iex> false or 1
1
iex> not 1
** (ArgumentError) argument error
iex> !1
false
iex> nil && "bilo sta"
nil
iex> :atom || false
:atom
```

Primer koda 33: Primeri upotrebe logičkih operatora

```
defmodule Kalkulator do
  def pomnozi(x, y), do: x * y

  #privatna funkcija
  defp saberi(x, y) do
    x + y
  end

  def saberi_i_ispisi(x, y) do
    IO.puts("#{a} + #{b} = #{a+b}") #primer bočnog efekta
    saberi(x, y)
  end
end
```

Primer koda 34: Primeri definisanja funkcija

funkcija `pomnozi` u primeru koda 34, dok u funkciji `saberi_i_ispisi` vidimo da je dozvoljeno pisanje funkcija koje imaju bočne efekte. Ukoliko je predviđeno da se neke funkcije koriste samo unutar modula, one se definišu kao privatne ključnom rečju `defp`.

Naziv modula mora počinjati velikim slovom i po konvenciji se pišu u kamiljoj notaciji, dok je imenovanje funkcija identično imenovanju promenljivih, s tim što funkcija čiji naziv se završava simbolom `?` po konvenciji vraća vrednosti `true` ili `false`, a simbolom `!` se označava funkcija koja može izazvati grešku prilikom izvođenja. Pozivanje imenovanih funkcija moguće je sa i bez navođenja zagrada, dok su argumenti razdvojeni zarezima. Na primer, funkciju `pomnozi` možemo pozvati sa: `Kalkulator.pomnozi(2, 3)` i `Kalkulator.pomnozi 2, 3`.

Elixir, nasuprot *Elm*-u, nema ugrađene Karijeve funkcije, dozvoljava navode-

nje podrazumevanih vrednosti, kao i preopterećivanje funkcija, što je i prikazano u primeru koda 36. Funkciju potpuno određuje modul u kome se nalazi, naziv i arnost (broj argumenata), tako da potpun naziv operatora nadovezivanja listi jeste *Kernel.++/2*.

```
defmodule Pravougaonik do
  def površina(a), do: a * a      # Pravougaonik.površina/1
  def površina(a, b), do: a * b  # Pravougaonik.površina/2
end

defmodule Brojac do
  # podrazumevana vrednost za n je 1
  def uvecaj(x, n \\ 1), do: x + n
end
```

Primer koda 35: Upotreba preopterećivanja funkcija i podrazumevanih vrednosti

Pored različitog broja argumenata, preopterećivanje funkcije može se vršiti korišćenjem čuvara (eng. guards), tj. postavljanjem uslova nad argumentima prilikom definisanja funkcija. Primer preopterećivanja funkcije upotrebom čuvara dat je u primeru koda

```
defmodule Math do
  def sgn(x) when x < 0 do
    -1
  end

  def sgn(0), do: 0

  def sgn(x) when x > 0 do
    1
  end
end
```

Primer koda 36: Preopterećivanja funkcija upotrebom čuvara

Anonimne funkcije

Sintaksa navođenja anonimnih funkcija prikazana je u primeru koda 37. Za razliku od imenovanih funkcija, argumenti se ne navode unutar zagrada po konvenciji, mada je moguće koristiti zagrade. Takođe, prilikom pozivanja anonimnih funkcija obavezno je korišćenje zagrada i navođenje tačke (.) pre njih.

```
iex> saberi = fn x, y ->
...>   x + y
...> end
saber1.(1, 2)
3
```

Primer koda 37: Primer definisanja i pozivanja anonimne funkcije

Anonimne (lambda) funkcije u *Elixir*-u se razlikuju od imenovanih i ubrajaju se u osnovne tipove podataka. Imenovane funkcije nije moguće proslediti kao parametar ili vezati za promenljivu, već se za to mogu koristiti isključivo anonimne funkcije. Ovaj nedostatak imenovanih funkcija se može prevazići upotrebom operatora `&` (eng. *capture*), koji imenovanu funkciju pretvara u anonimnu. Operator `&` se može koristiti i za kraću sintaksu anonimnih funkcija, što je i prikazano u primeru koda 38.

```
iex> saberi1 = &Kernel.+/2
iex> saberi1.(3, 4)
7
iex> saberi2 = &(&1 + &2) # &n je n-ti argument
iex> saberi2.(5, 6)
11
```

Primer koda 38: Primer upotrebe `&` operatora

Zatvorenja (eng. *closures*) predstavljaju još jednu specifičnost lambda funkcija u *Elixir*-u. Prilikom definisanja, lambda funkcija ima pristup svim vrednostima unutar opsega u kom se definiše. Ukoliko koristi vrednosti promenljivih van svog opsega, kreiraju se reference na trenutne vrednosti promenljivih i ponovno vezivanje promenljivih ne utiče na izvršavanje funkcije. U primeru koda 39 dat je primer zatvorenja.

```
iex> x = 3
3
iex> saberi_sa_3 = &(&1 + x)
iex> saberi_sa_3.(3)
6
iex> x = 4
4
iex(35)> saberi_sa_3.(3)
6
```

Primer koda 39: Primer zatvorenja u *Elixir*-u

Operator prosleđivanja - *pipe* operator (`|>`)

Operator `|>` u *Elixir*-u prosleđuje vrednost sa leve strane kao prvi argument funkciji sa desne strane, što je različito ponašanje koje isti operator ima u *Elm*-u, gde se vrednost sa leve strane prosleđuje kao poslednji argument funkciji sa desne strane. Takođe, *Elixir* ne podržava slične operatore koji postoje u *Elm*-u. Različito ponašanje operatora `|>` u oba jezika predstavljeno je u primeru koda 40.

```
iex> podeli = &(&1 / &2)           > podeli x y = x / y
iex> 1 |> podeli.(2)               > 1 |> podeli 2
0.5                                2 : Float
```

Primer koda 40: Upotreba **pipe** operatora u *Elixir*-u (levo) i *Elm*-u (desno)

Atributi

Atributi u modulu se mogu koristiti za definisanje konstanti. Takođe, *Elixir* pruža attribute za dokumentovanje koda: `@moduledoc` i `@doc`, čija je upotreba prikazana u primeru koda 41. Atributi imaju višestruku upotrebu i mnogi bitni koncepti se za-

```
defmodule Krug do
  @moduledoc "Osnovne funkcije za rad sa krugovima"
  #definisane konstante
  @pi 3.141593

  @doc "Izračunava obim kruga"
  def obim(r), do: 2*r*@pi

  @doc "Izračunava površinu kruga"
  def površina(r), do: r*r*@pi
end
```

Primer koda 41: Definisanje konstante i dokumentovanje koda pomoću atributa

snivaju na njima, među kojima je i specifikacija tipova (eng. *typespecs*) koja se može koristiti za definisanje korisničkih tipova, kao i za anotaciju tipova sličnu anotaciji u *Elm*-u. Više o specifikaciji tipova i samim atributima može se videti na zvaničnoj stranici [32].

Korišćenje funkcija iz drugog modula

Funkcije modula mogu se pozivati kao *NazivModula.naziv_funkcije*, što često nije praktično jer nazivi modula mogu biti veoma dugački. Stoga *Elixir* nudi dve

direktive za kraći i čitljiviji kôd prilikom upotrebe funkcija iz drugih modula — `import` i `alias`, koje su prikazane u primeru koda 42.

Upotrebom direktive `import` sve funkcije navedenog modula uključuju se u trenutni modul i mogu se pozivati bez navođenja imena modula, a moguće je ograničiti uključivanje samo određenih funkcija. Sa druge strane, `alias` omogućava korišćenje određenog imena (alijasa) za bilo koji modul i prilikom pozivanja funkcija obavezno je njegovo navođenje.

```
alias Api.Accounts.User, as: AccUser
alias Api.Accounts.User #isto kao alias Api.Accounts.User, as: User

# uključivanje samo authenticate/2 funkcija
import Api.Accounts, :only [authenticate: 2]
```

Primer koda 42: Upotreba direktiva `import` i `alias`

Osnovne strukture podataka

Osnovne strukture podataka u programskom jeziku *Elixir* predstavljaju liste, torke i mape. Pored njih, bitno mesto zauzimaju i liste ključnih reči. Podaci u *Elixir*-u su nepromenljivi (imutabilni) i prilikom izvršavanja operacija nad njima ne menja se trenutna vrednost, već se uvek kreira nova.

Liste

Kao i u *Elm*-u, lista kao tip predstavlja jednostruko povezanu listu, čiji se elementi navode unutar uglastih zagrada, ali za razliku od *Elm*-a ne moraju biti istog tipa. U modulu `Kernel` se nalaze operatori za proveru da li element pripada listi (`in`), za nadovezivanje listi (`++`), kao i operator oduzimanja (`--`) koji predstavlja razliku elemenata levog i desnog operanda. Tu su i funkcije za određivanje glave, repa i dužine liste. Umesto operatora *cons*, *Elixir* pruža sintaksu u obliku `[glava|rep]` koja se može koristiti u kreiranju nove liste i poklapanju obrazaca. Za rad sa listama može se koristiti veliki broj funkcija iz modula `List`, kao i funkcije iz modula `Enum` o kom će biti više reči kasnije. Primeri upotrebe nekih od navedenih funkcija prikazani su u primeru koda 43.

Liste karaktera Lista karaktera (eng. *charlists*) je lista celih brojeva, gde svaki element predstavlja jedan karakter. Veoma je slična niskama, za navođenje se

```
iex> [1,2,3, "123", true] ++ [:atom, ["aaa", "bbb"]]
[1, 2, 3, "123", true, :atom, ["aaa", "bbb"]]
iex> [1, 2, 1, 3, 5] -- [1, 5, 6]
[2, 1, 3]
iex> "abc" in [1, 2, "abc", false]
true
iex> List.insert_at([0,1,2], 1, 3)
[0, 3, 1, 2]
iex> tl([1,2,3,4])
[2, 3, 4]
```

Primer koda 43: Rad sa listama u *Elixir*-u

koriste apostrofi umesto navodnika, ali glavna razlika je u internoj reprezentaciji, kao i funkcijama koje se nad njima izvršavaju. Mogu se navoditi u jednostrukoj i trostrukoj notaciji, a moguća je i interpolacija. U primeru koda 44 dati su primeri liste karaktera.

```
iex> '''
Primer u
više redova
'''
'Primer u\nviše redova\n'
iex> '2+2=#{2+2}'
'2+2=4'
iex> [97,98,99]
'abc'
iex> '123' ++ [0]
[49, 50, 51, 0]
```

Primer koda 44: Primeri lista karaktera u *Elixir*-u

Torke

I u *Elixir*-u torke mogu sadržati elemente različitih tipova, ali za razliku od *Elm*-a, navode se unutar vitičastih zagrada, broj elemenata nije ograničen i elementi se mogu uklanjati i dodavati. Iako dozvoljavaju promenljiv broj elemenata, torke su zamišljene kao kolekcija podataka fiksne dužine. U modulu `Kernel` se nalaze funkcije za pristupanje i ažuriranje elemenata, kao i funkcija za određivanje broja elemenata torke, dok se ostale funkcije za rad sa torkama nalaze u modulu `Tuple`. Prikaz rada navedenih funkcija dat je u primeru koda 45.

```
iex> tuple_size({1, "aaa", false})
3
iex> elem({:prvi, :drugi, :treci}, 0)
:prvi
iex> put_elem({1, 2}, 1, 4)
{1, 4}
iex> Tuple.insert_at({1, 2}, 1, 4)
{1, 4, 2}
iex> Tuple.to_list({1, 2, 3})
[1, 2, 3]
```

Primer koda 45: Rad sa torkama u *Elixir*-u

Liste ključnih reči

Liste ključnih (eng. *keyword list*) reči predstavljaju posebnu vrstu listi gde je svaki element dvočlana torka, čiji je prvi član (tj. ključna reč) obavezno atom, a drugi može biti bilo kog tipa. Takođe, jedna ključna reč može se navesti više puta. *Elixir* pruža i jednostavniju sintaksu koja izuzima pisanje vitičastih zagrada. Za rad sa listama ključnih reči mogu se koristiti sve funkcije i operatori kao i sa običnim listama i, dodatno, funkcije iz modula `Keyword` i operator `[]` za pristup po određenoj ključnoj reči. Prikaz nekih od funkcija i operatora dat je u primeru koda 46.

```
iex> [{:prvi, 1}, {:drugi, 2}, {:treci, 3}]
[prvi: 1, drugi: 2, treci: 3]
iex> Keyword.get([prvi: 1, drugi: 2, treci: 3], :drugi)
2
iex> lista = [prvi: 1, drugi: 2, treci: 3]
[prvi: 1, drugi: 2, treci: 3]
iex> lista[:prvi]
1
iex> [a: 1, b: 2] ++ [c: "3"]
[a: 1, b: 2, c: "3"]
```

Primer koda 46: Primeri rada sa listama ključnih reči u *Elixir*-u

Mape

Za razliku od liste ključnih reči, mape predstavljaju kolekciju elemenata u obliku *ključ-vrednost* gde oba člana mogu biti proizvoljnog tipa. Liste ključnih reči čuvaju uređenje, što je osobina koju nemaju mape, ali u slučaju većeg broja elemenata mape pružaju bolju efikasnost. Takođe, ključevi unutar mape moraju biti jedinstveni.

Elementi mape navode se koristeći `%{}` sintaksu, a ključ i vrednost se odvajaju znakom `=>`. U slučaju da su svi ključevi atomi, može se koristiti jednostavnija sintaksa kao kod liste ključnih reči. Oba slučaja prikazana su u primeru koda 47.

```
iex> %{1 => false, "bb" => [1,2], {1,2} => 3}
%{1 => false, {1, 2} => 3, "bb" => [1, 2]}
iex> %{a: 123, b: 'bbb', c: "c"}
%{a: 123, b: 'bbb', c: "c"}
```

Primer koda 47: Primeri mapa u *Elixir*-u

Pristup elementima mape moguć je pomoću operatora `[]`, kao i funkcija iz modula `Map`, a ukoliko su svi ključevi mape atomi, dostupna je sintaksa oblika *mapa.ključ*. Pored funkcija iz modula `Map`, nad mapama se mogu koristiti i funkcije iz modula `Enum`. Mape predstavljaju dinamičku strukturu u kojoj se mogu dodavati i uklanjati elementi, a moguće je i ažuriranje elemenata. Pored funkcija iz modula, dostupna je i sintaksa ažuriranja slična sintaksi ažuriranja slogova u *Elm*-u i prikazana je u primeru koda 48.

```
iex> mapa = %{a: 1, b: 2, c: 3}
%{a: 1, b: 2, c: 3}
iex> mapa.b
2
iex> mapa[:c]
3
iex> %{mapa | a: 4}
%{a: 4, b: 2, c: 3}
iex> %{ %{ "a" => true, :b => 0 } | :b => false }
%{:b => false, "a" => true}
```

Primer koda 48: Pristup i ažuriranje elemenata mape

Modul Enum

Elixir pruža koncept nabrojivih podataka (eng. *enumerables*), odnosno tipove podataka kroz koje je moguće iterirati. Veliki broj funkcija uobičajenih za ovaj tip podataka (pronalazak i transformisanje elemenata, grupisanje, sortiranje, filtriranje...) nalazi se u modulu `Enum`. Liste i mape spadaju u nabrojive tipove, a pored njih, *Elixir* omogućava i rad sa *opsezima* (eng. *ranges*). Upotreba nekih od funkcija iz modula `Enum` nad listama, opsezima i mapama se može videti u primeru koda

49. Funkcije iz modula `Enum` mogu raditi sa bilo kojim tipom koji implementira `Enumerable` protokol. Više o protokolima biće rečeno u poglavlju 3.10.

```
iex> Enum.map([1, 2, 3], fn x -> x * 3 end)
[3, 6, 9]
iex> Enum.reduce(%{1 => 2, 3 => 4}, 0, fn {k, v}, s -> s + k + v end)
10
iex> Enum.filter(1..10, fn x -> rem(x, 3) == 0 end)
[3, 6, 9]
```

Primer koda 49: Upotreba funkcija iz modula `Enum` nad listama, mapama i opsezima

Skraćenice Filtriranje i mapiranje nad nabrojivim podacima predstavljaju čestu pojavu u *Elixir* programima, te su uvedene skraćenice (eng. *comprehensions*) kao sintaksička olakšica (eng. *syntactic sugar*), koja grupiše ovakve operacije u specijalnu formu koja se naziva `for`. Upotreba skraćenica prikazana je u primeru koda 50, a više o njima može se pronaći na zvaničnoj [32] stranici.

```
for n <- 1..5, do: n * n
[1, 4, 9, 16, 25]
iex> for i <- [:a, :b, :c], j <- [1, 2], do: {i, j}
[a: 1, a: 2, b: 1, b: 2, c: 1, c: 2]
iex> nije_paran? = &(rem(&1, 2) != 0)
iex> for n <- 1..10, nije_paran?.(n), do: 2 * n
[2, 6, 10, 14, 18]
```

Primer koda 50: Primeri upotrebe skraćenica

Tokovi

Tokovi (eng. *Streams*) predstavljaju posebnu vrstu nabrojivih tipova koji se mogu koristiti za kreiranje složenih lenjih operacija nad nabrojivim podacima. Tokovi su definisani u modulu `Stream` i funkcije unutar ovog modula izgledaju vrlo slično funkcijama u modulu `Enum`, s tim da je povratna vrednost ovih funkcija uvek tok, što omogućava njihovu kompoziciju. Primer lenjeg izračunavanja upotrebom tokova dat je u primeru koda 51.

Strukture

Strukture predstavljaju koncept razvijen na osnovu mapa, s tim što imaju podrazumevane vrednosti, a ključevi su isključivo atomi. Sintaksa definisanja struktura

```
# kompozicija operacija, ne vrši se izračunavanje
iex> stream = 1..1_000_000 |>
...> Stream.filter(&(rem(&1, 3) == 0)) |>
...> Stream.map(&(&1 * 2))
# lenjo izračunavanje - samo za prvih 5
iex> stream |> Enum.take(5)
[6, 12, 18, 24, 30]
```

Primer koda 51: Kompozicija tokova i lenjo izračunavanje

data je u primeru koda 52. Mogu se navoditi samo unutar modula i to najviše jedna.

```
defmodule Osoba do
  defstruct ime: "Pera", godine: 25
end

defmodule Osoba1 do
  defstruct [:ime, godine: 25]
end
```

Primer koda 52: Primeri definisanja struktura

Ukoliko se postavljaju podrazumevane vrednosti svih ključeva, moguće je izostaviti navođenje uglastih zagrada. U suprotnom obavezno je njihovo korišćenje, pri čemu se nenavedenim ključevima podrazumevana vrednost postavlja na `nil`. U slučaju da se nekim ključevima ne dodeljuje podrazumevana vrednost, takvi ključevi se obavezno navode na početku liste. Sintaksa kreiranja modula je slična sintaksi kreiranja mapa — `%NazivModula{}`. U primeru koda 53 može se videti kreiranje struktura iz primeru koda 52.

```
iex> %Osoba{}
%Osoba{godine: 25, ime: "Pera"}
iex> %Osoba1{}
%Osoba{godine: 25, ime: nil}
#moguće je specifikacija ključeva tokom kreiranja
iex> %Osoba1{godine: 30}
%Osoba1{godine: 30, ime: nil}
```

Primer koda 53: Primeri kreiranja struktura

Nad strukturama se mogu koristiti funkcije iz modula `Map`, ali ne i funkcije iz modula `Enum`, kao ni operator `[]`. Za pristup ključevima koristi se isključivo sintaksa *struktura.ključ*, a ažuriranje vrednosti može se izvršiti korišćenjem simbola `|` kao kod

mapa. Svaka struktura ima specijalni ključ `__struct__` koja sadrži naziv strukture, odnosno modula.

Poklapanje obrazaca

U prethodnom poglavlju nije naveden jedan od najvažnijih operatora u programskom jeziku *Elixir* — operator uparivanja ili poklapanja (`=`), koji ima veoma bitnu ulogu u poklapanju obrazaca. Za razliku od *Elm*-a gde se poklapanje obrazaca koristi isključivo unutar izraza `case` i `let`, u *Elixir*-u je ono dosta prisutnije i koristi se prilikom definisanja promenljivih i funkcija, kao i u kontroli toka.

Operator uparivanja pokušava da izraz sa leve strane upari sa izrazom sa desne strane, ukoliko ne dođe do uspešnog uparivanja prijavljuje grešku — `MatchError`. U većini drugih programskih jezika, operator `=` koristi se za dodeljivanje vrednosti promenljivoj, što se u *Elixir*-u postiže kroz poklapanje obrazaca. Ukoliko dođe do uspešnog poklapanja, promenljive u levom izrazu *vezuju se* za odgovarajuće vrednosti u desnom. Imenovanje promenljivih slično je imenovanju atoma, s tim što umesto dvotačke promenljiva mora počinjati malim slovom, po konvenciji se koristi *snake_case* notacija. Moguće je koristiti operator uparivanja bez promenljivih, kao i izvršiti ponovno vezivanje promenljive za drugu vrednost, što je prikazano u primeru koda 54. Takođe, ukoliko ne želimo ponovno vezivanje možemo koristiti *pin* operator `^`.

```
iex> x = 1
1
iex> 1 = 1
1
iex> x = x + 1
2
iex> 1 = x
** (MatchError) no match of right hand side value: 2
iex> ^x = 1
** (MatchError) no match of right hand side value: 1
```

Primer koda 54: Vezivanje promenljive za vrednost

Operator uparivanja se često upotrebljava za pristupanje elementima torki, listi i mapa, gde se može izvršiti vezivanje više promenljivih odjednom. Kao i u *Elm*-u, simbol `_` se može koristiti za poklapanje vrednosti koje se neće dalje koristiti. Primeri poklapanja obrazaca listi i torki dati su u primeru koda 55.

```
iex> {a, b} = {1, 2}
{1, 2}
iex> [c, c] = [3, 3]
[3, 3]
iex> {1, 2, 3, d} = {a, b, c, {4}}
{1, 2, 3, {4}}
iex> [ _ | rep ] = 'abcd'
iex> rep
'bcd'
```

Primer koda 55: Poklapanje obrazaca listi i torki

U slučaju mapa moguće je parcijalno poklapanje, tj. leva strana ne mora da sadrži sve ključeve desne, što omogućava pristupanje samo potrebnim vrednostima iz mape. Parcijalno poklapanje je prikazano u primeru koda 56.

```
iex> %{ime: ime, godine: godine} = %{ime: "Pera", godine: 25}
%{godine: 25, ime: "Pera"}
iex> "#{ime} ima #{godine} godina."
"Pera ima 25 godina."
iex> %{ime: ime} = %{ime: "Pera", godine: 25}
%{godine: 25, ime: "Pera"}
iex(55)> %{ime: ime, prezime: prezime} = %{ime: "Pera", godine: 25}
** (MatchError) no match of right hand side value: %{godine: 25, ...}
```

Primer koda 56: Poklapanje obrazaca mapa

Prilikom definisanja funkcija može se koristiti poklapanje obrazaca za razlaganje argumenata, ali i za preopterećivanje funkcija iste arnosti korišćenjem različitih šablona. Primer preopterećivanja funkcije upotrebom poklapanja obrazaca dat je u primeru koda 57.

```
defmodule Oblik do
  def površina({:pravougaonik, a, b}), do: a * b
  def površina({:kvadrat, a}), do: a * a
  def površina({:krug, r}), do: r * r * 3.141593
  # podrazumvani slučaj
  def površina(oblik), do: {:error, {:nepoznat_oblik, oblik}}
end
```

Primer koda 57: Preopterećivanje funkcija upotrebom poklapanja obrazaca

Makroi

Makroi predstavljaju jednu od najvažnijih novouvedenih karakteristika *Elixir*-a u odnosu na *Erlang*. Makroi omogućavaju metaprogramiranje, tj. pisanje koda koji generiše kôd, što dovodi do značajnog smanjenja količine šablonskog (eng. *boilerplate*) koda, a samim tim i do vrlo čitljivog i elegantnog koda. Upotrebom makroa se tokom kompilacije vrše transformacije nad kodom, čime se ne utiče na performanse izvršavanja. Makroi i metaprogramiranje nisu predmet ovog rada, ali je neophodno razumeti kako makroi funkcionišu, jer su mnoge funkcionalnosti *Elixir*-a implementirane pomoću njih i njihova upotreba je gotovo neizbežna.

Makroi se definišu unutar modula. Pre upotrebe, budući da se prevode tokom kompilacije, moduli u kojima su definisani moraju biti dostupni, što se obezbeđuje direktivom `require`.

Veoma bitan makro koji se često povezuje sa direktivama jeste makro `use`, koji omogućava ubacivanje spoljne funkcionalnosti (koda) u trenutni modul. U primeru koda 58 prikazan je primer upotrebe makroa `use` za kreiranje modula sa jediničnim testovima pomoću biblioteke *ExUnit*, koja je instalirana zajedno sa *Elixir*-om.

```
defmodule ModulSaTestom do
  use ExUnit.Case

  test "uvek prolazi" do
    assert true
  end
end
```

Primer koda 58: Upotreba makroa `use`

Kontrola toka

Elixir nema izraze ili naredbe za kontrolu toka i prethodno je prikazano kako se korišćenjem čuvara i poklapanja obrazaca može vršiti kontrola toka izvršavanja. Ovakva rešenja nisu uvek adekvatna i jednostavna za upotrebu, jer podrazumevaju kreiranje zasebnih funkcija i prosleđivanje neophodnih argumenata što dovodi do povećanja šablonskog (eng. *boilerplate*) koda. Stoga *Elixir* uvodi makroe koji ovo rade tokom kompilacije umesto nas, i to su `if`, `unless`, `cond` i `case`.

Upotreba makroa `if` i `unless`

Makro `if` se koristi za standardno *if-else* grananje. Makro `unless` suprotan je makrou `if` i može se posmatrati kao `if not`. Sintaksa upotrebe oba makroa data je u primeru koda 59.

```
if uslovni_izraz do
  blok_izraza_1
else
  blok_izraza_2
end

unless uslovni_izraz do
  blok_izraza_2
else
  blok_izraza_1
end
```

Primer koda 59: Sintaksa makroa `if` i `unless`

Kao kod operatora `&&` i `||` uslovni izraz se smatra tačnim ukoliko njegova vrednost nije `nil` ili `false`. Za razliku od Elm-a, gde je navođenje `else` grane obavezno, u *Elixir*-u to nije slučaj. Vrednost izraza odgovara vrednosti poslednjeg izraza u bloku, a ukoliko se `else` grana ne navede i uslov nije istinit (odnosno nije neistinit u slučaju makroa `unless`), vrednost izraza je `nil`. Postoje i skraćene, jednolinijske verzije sintaksi i prikazane su u primeru koda 60.

```
if uslovni_izraz, do: izraz_1, else: izraz_2

unless uslovni_izraz, do: izraz_2, else: izraz_1
```

Primer koda 60: Skraćena sintaksa makroa `if` i `unless`

Upotreba makroa `cond` i `case`

Makroi `cond` i `case` se mogu posmatrati kao grananje oblika `if...else if ... else`, pri čemu se u slučaju makroa `cond` proverava istinitost izraza, a kod makroa `case` koristi poklapanje obrazaca. Sintaksa navedenih makroa data je u primeru koda 61.

```
cond do
  izraz_1 -> blok_1
  izraz_2 -> blok_2
  ...
end

case izraz do
  obrazac_1 -> blok_1
  obrazac_2 -> blok_2
  ...
end
```

Primer koda 61: Sintaksa makroa `cond` i `case`

Bitan je redosled navođenja, jer će se izvršiti blok prvog istinitog izraza (`cond`) ili poklopljenog obrasca (`case`). Povratna vrednost predstavlja vrednost izvršenog bloka, a u slučaju da se ne izvrši nijedan blok izbacuje se greška.

Iteracije

Slično *Elm*-u, *Elixir* umesto petlji koristi funkcije višeg reda i rekurziju, s tim što *Elixir* pruža mogućnost lenjog izračunavanja korišćenjem tokova.

Upravljanje greškama

Elixir za razliku od *Elm*-a, nema tipove kao `Maybe` i `Result`, ali je praksa da se prilikom pisanja funkcija, koje mogu dovesti do greške, rezultat izvršavanja vraća kao torka koja podseća na `Result` ili `Maybe` u slučaju da nije potrebna poruka o grešci. U primeru koda 62 može se videti primer takve obrade grešaka. Ipak, u nekim

```
iex> case File.read "primer.txt" do
...>   {:ok, sadrzaj}    -> IO.puts "Ok: #{sadrzaj}"
...>   {:error, razlog} -> IO.puts "Error: #{razlog}"
...> end
```

Primer koda 62: Primer pravilne obrade grešaka

vrlo retkim slučajevima kada neke biblioteke ili delovi koda nisu implementirani u duhu funkcionalnog programiranja, moguća je obrada grešaka slična većini drugih programskih jezika — korišćenjem `try`, `catch` i `rescue` mehanizma o kom se može videti više u zvaničnoj dokumentaciji [32].

Polimorfizam preko protokola

Protokoli predstavljaju mehanizam za postizanje polimorfizma ukoliko je potrebno razlikovati ponašanje u odnosu na tip podatka. Ovo je takođe moguće uraditi korišćenjem poklapanja obrazaca i čuvara, ali protokoli omogućavaju implementiranje različitog ponašanja u različitim delovima koda.

Protokol je zapravo modul koji sadrži samo deklaracije funkcija ali ne i implementaciju, sličan je interfejsima ili apstraktnim klasama u drugim programskim jezicima. Definiše se pomoću makroa `defprotocol`, a primer protokola za izračunavanje veličine podatka dat je u primeru koda 63. Kada postoji definisan protokol moguće je dodati neograničen broj implementacija u različitim modulima. Za implementiranje protokola koristi se makro `defimpl` pri čemu se navodi protokol koji

```
defprotocol Size do
  def size(data)
end
```

Primer koda 63: Primer definisanja protokola

```
defimpl Size, for: Map do
  def size(map), do: map_size(map)
end
```

```
defmodule User do
  defstruct [:email, :name]

  defimpl Size do
    # dva polja
    def size(%User{}), do: 2
  end
end
```

Primer koda 64: Primeri implementacije protokola

se implementira, tip za koji se implementira i implementacija funkcija protokola. Prilikom implementacije protokola nad strukturama moguće je izostaviti navođenje tipa. Primeri nekoliko implementacija protokola iz primera koda 63 može se videti u primeru koda 64.

Glava 4

Portal MSNR

Portal MSNR je zamišljen kao veb aplikacija koja će ispratiti mnogobrojne aktivnosti tokom kursa *Metodologija stručnog i naučnog rada* i olakšati njihovo sprovođenje. Jedan od osnovnih ciljeva kursa jeste da uvede studente u pisanje i recenziranje naučnih radova, kako se navode reference kao i koji su etički kodeksi naučnoistraživačkog rada. Takođe, velika pažnja se posvećuje timskom radu, komunikaciji, sticanju „mekih” veština, držanju prezentacija i pisanju CV-a.

Funkcionalnosti portala definisani su kroz aktivnosti tokom kursa:

- Prijava grupa za izradu seminarskog rada
- Odabir tema za tekuću godinu
- Prijava studenata koji žele da rade recenziranje
- Predavanje CV-a
- Predavanje prve verzije seminarskog rada
- Predavanje recenzija
- Prijava studenata za ocenjivanje prezentacija
- Predavanja odgovora na recenzije i finalne verzije
- Ocenjivanje prezentacija

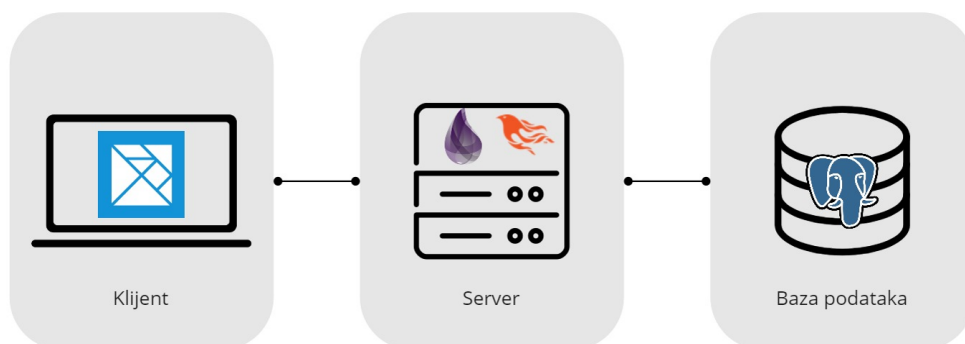
Na samom početku kursa, očekuje se da studenti podnesu zahtev za registraciju na portalu. Nakon što profesor odobri njihov zahtev, studenti mogu da se prijave na portal gde im se izlistavaju sve trenutne aktivnosti na kursu. Za svaku aktivnost

prikazuju se osnovni podaci — naziv i opis aktivnosti, rok do kada može da se izvrši, broj poena, kao i sadržaj aktivnosti. Profesor pregleda zahteve za registracije, dodaje aktivnosti i teme za tekuću godinu, i ocenjuje izvršene studentske aktivnosti. Širi opis funkcionalnosti portala dat je u sekciji gde su predstavljeni entiteti portala i njihov odnos. Takođe, prijava i ocenjivanje studentskih prezentacija neće biti obrađene u sklopu rada, ali je predviđeno njihovo naknadno dodavanje.

Kôd implementacije portala MSNR javno je dostupan na *GitHub*-u [1].

4.1 Arhitektura portala

Arhitektura portala MSNR prikazana je na slici 4.1 i predstavlja tipičan primer troslojne arhitekture. Na klijentskoj strani se nalazi korisnički interfejs koji je implementiran u *Elm*-u kao jednostranična aplikacija (eng. *Single Page Application* — *SPA*). Središnji sloj predstavlja aplikacioni veb interfejs - (eng. *Web Application Programming Interface* — *API*) koji je pomoću razvojnog okvira *Phoenix* [34] implementiran u stilu arhitekture REST (eng. *Representational State Transfer*) [21]. Treći sloj čini relaciona baza podataka, a odabrani sistem za upravljanje bazom je *PostgreSQL* [30] .



Slika 4.1: Arhitektura portala MSNR

4.2 Šema baze i opis osnovnih entiteta aplikacije

Na slici 4.2 data je šema baze gde su prikazani osnovni entiteti i relacije između njih, u nastavku je dat njihov opis.



Slika 4.2: Dijagram tabela portala MSNR

Zahtev za registraciju studenata Zahtev za registraciju, pored entiteta semester i korisnik, predstavlja polazni entitet. Predstavljen je tabelom *student_registrations* i sadrži informacije potrebne za registrovanje studenata — ime, prezime, broj indeksa i email adresu, kao i status zahteva koji označava da li je odobren, odbijen ili nerazrešen.

Korisnik Tabela *users* predviđena je za upravljanje korisničkim nalogima. Mogu postojati dva tipa, odnosno uloge, korisnika — student i profesor. Pored osnovnih informacija o korisniku — ime, prezime, rola i email, sadrži šifrovanu lozinku (*hashed_password*), osvežavajući token (*refresh_token*) čija će upotreba biti objšnjena

kasnije, i jedinstveni univerzalni identifikator koji se koristi prilikom postavljanja lozinke pre prvog prijavljivanja ili u slučaju da se zaboravi. Inicijalno će postojati profesorski nalog u tabeli, a prilikom prihvatanja zahteva za registraciju kreira se studentski nalog i studentu se šalje mail sa vezom za postavljanje lozinke.

Semestar Sve aktivnosti studenata vezane su za semestar, koji se predstavlja godinom u kojoj se održava i oznakom da li je aktivan, gde samo jedan semestar može biti trenutno aktivan.

Student Kada profesor prihvati zahtev za registraciju, pored unosa u tabelu korisnici, vrše se još dva unosa — jedan u tabelu *students*, koja sadrži samo referencu ka korisniku i broj indeksa studenta, i drugi u tabelu koja predstavlja relaciju studenta i semestra — *students_semestars*. Ova tabela, pored referenci ka studentu i semestru, ima i referencu ka grupi tako da svaki student u toku jednog semestra može biti član samo jedne grupe.

Grupa Prilikom prijave grupe, kreira se novi unos u tabeli grupe, dok se u tabeli *students_semestars* unosi referenca ka kreiranoj grupi za sve studente iz grupe. Tabela grupe sadrži i referencu ka temi koja se unosi prilikom odabira teme.

Tema seminarskog rada Profesor unosi naslove tema seminarskih radova koji se mogu odabrati u toku trenutnog semestra. Teme su predstavljene tabelom *topics* i sadrže naslov, redni broj i referencu ka semestru u kom se mogu odabrati.

Aktivnost i tip aktivnosti Navedene aktivnosti na početku poglavlja su zapravo tipovi aktivnosti koji su predstavljeni tabelom *activity_types*, a aktivnosti (tabela *activities*) predstavljaju relaciju između tipa aktivnosti i semestra. Tip aktivnosti sadrži naziv i opis aktivnosti, jedinstven kôd, oznake da li ima prijavu i da li je grupna aktivnost, i sam sadržaj aktivnosti. Pored reference ka semestru i tipu, aktivnost sadrži informacije o periodu u kom se može izvršiti, broju poena koji nosi i oznaku da li predstavlja samo prijavu za referisani tip ili konkretno izvršavanje.

Dodeljene aktivnosti Dodeljena aktivnost je centralni entitet portala i predstavljena je tabelom *assignments*. U zavisnosti da li je aktivnost grupna ili ne, dodeljena aktivnost će imati referencu ka studentu ili grupi. Neke aktivnosti, kao u

slučaju recenzije, mogu se odnositi na određenu temu zbog čega je dodata kolona *related_topic_id*. Pored navedenih referenci, tabela sadrži i broj poena koji je osvojen, komentar koji je ostavio profesor, kao i oznaku da li je dodeljena aktivnost urađena.

Dokument Veliki broj aktivnosti se zasniva na predaji dokumenata. Svi predati dokumenti nalaziće se na serveru, a u tabeli *documents* čuvaće se informacije o nazivu, lokaciji dokumenta na serveru i referenca ka korisniku koji je priložio dokument. Tabela *assignments_documents* označava koja dokumenta su povezana sa kojom dodeljenom aktivnosti, a pored referenci ka ovim tabelama, sadrži i oznaku da li je taj dokument prikazan — što označava da je taj dokument priložio profesor.

Glava 5

Implementacija serverskog dela portala

Programski jezik *Elixir*, kao jezik opšte namene, nema ugrađenu podršku za razvoj veb aplikacija i za tu svrhu razvojni okvir *Phoenix* predstavlja najpopularniji izbor. U prvom delu ove glave predstavljen je deo razvojnog okvira *Phoenix* na primeru portala MSNR, nakon čega je opisana implementacija osnovnih delova samog portala.

5.1 Razvojni okvir Phoenix

Razvojni okvir *Phoenix* napisan je na programskom jeziku *Elixir* i razvija se uporedo sa samim jezikom. Jedan od autora razvojnog okvira je i Žozé Valim (autor jeziku *Elixir*). Analogno uticaju programskog jezika *Ruby* na *Elixir*, razvojni okvir *Phoenix* je u velikoj meri inspirisan razvojnim okvirom *Ruby on Rails*, ali i drugim popularnim razvojnim okvirima.

Instalacija razvojnog okvira

Instaliranjem *Elixir*-a ujedno je instaliran i osnovni alat za rad sa *Elixir* projektima — *Mix*, koji se koristi za kreiranje, kompajliranje i testiranje projekata, instaliranje i objavljivanje paketa, kao i za upravljanje zavisnim paketima projekta. Upravo se pomoću njega instalira razvojni okvir *Phoenix* i alat za generisanje *Phoenix* projekata, ali prethodno je potrebno instalirati *Hex* [27] — menadžer paketa

za ekosistem *Erlang*. Instalacija potrebnih alati prikazana je u primeru koda 65. Pored navedenih alati potrebno je instalirati i *PostgreSQL*.

```
mix local.hex  
mix archive.install hex phx_new
```

Primer koda 65: Instalacija alati *hex* i *phx_new*

Kreiranje i struktura projekta

Phoenix je okvir za razvoj svih tipova veb aplikacija, omogućava iscrtavanje *HTML*-a na serverskoj strani i ima odličnu podršku za komunikaciju u realnom vremenu. Posедуje i svojstven koncept *LiveView* koji pruža bogato korisničko iskustvo u realnom vremenu sa iscrtavanjem na serverskoj strani. Za potrebe implementacije portala MSNR kreira se jednostavan veb API i prethodno navedene mogućnosti nisu potrebne. Takođe, portal ne podržava lokalizaciju, stoga nema potrebe za uključivanje modula *gettext*.

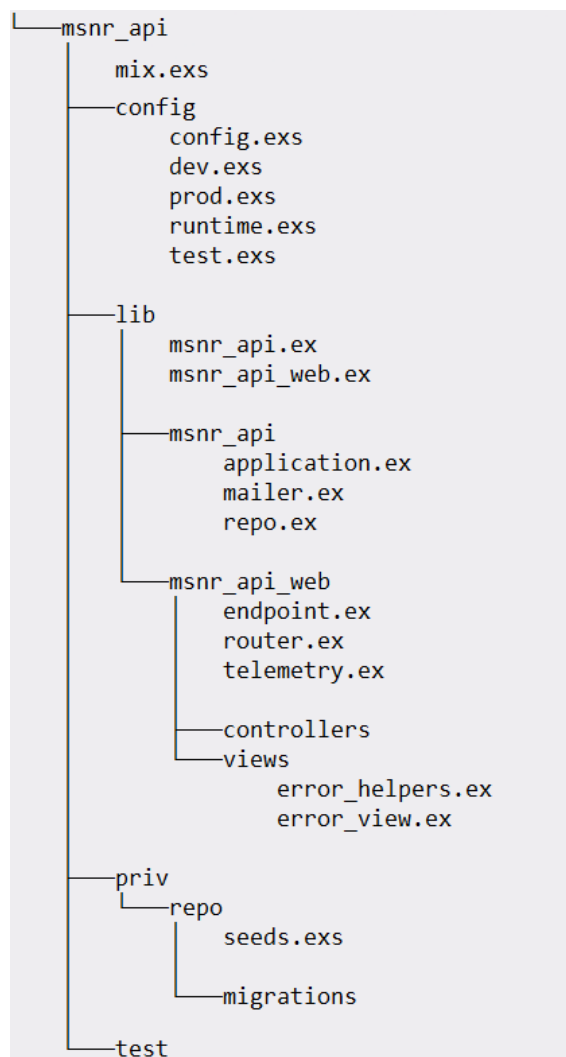
Komanda korišćena za kreiranje projekta:

```
mix phx.new msnr_api --no-assets --no-html --no-gettext
```

Parametrom *--no-assets* isključuje se generisanje direktorijuma *assets* koji je predviđen za *JavaScript* i *CSS* datoteke, dok se postavljanjem *--no-html* isključuje generisanje *HTML* šablona za serversko iscrtavanje. Nakon uspešno izvršene komande, struktura projekta prikazana je na slici 5.1.

Nastali projekat je u osnovi *Mix* projekat sa *Phoenix* proširenjima. Svaki *Mix* projekat ima konfiguracionu datoteku *mix.exs*, direktorijum *lib* koji sadži osnovni kôd i direktorijum *test* sa testovima. Datoteka *mix.exs* sadrži osnovne informacije o projektu — kako se kompajlira, pokreće i listu zavisnih paketa koji se koriste u projektu. Nakon kompajliranja projekta, generisaće se datoteka *mix.lock* koja čuva precizne verzije zavisnih paketa čime se garantuje da se u produkciji koriste iste verzije kao i tokom razvoja.

Razvojni okvir *Phoenix* dodaje i konfiguracije vezane za okruženje u kom se aplikacija izvršava. Podržana okruženja su razvojno (*dev.exs*), testno (*test.exs*) i produkciono (*prod.exs*) i konfiguracije su smeštene u direktorijumu *config*, zajedno sa glavnom konfiguracijom (*config.exs*) koja se odnosi na sva okruženja. Mogu se dodati i druga ukoliko je potrebno, a pored navedenih u direktorijumu *config* nalazi

Slika 5.1: Struktura *Phoenix* projekta

se i *runtime.exs* konfiguracija zadužena za učitavanje šifri (eng. *secrets*) i drugih konfiguracionih vrednosti iz varijabli okruženja.

U razvojnom okviru *Phoenix* projekti se organizuju po kontekstima (eng. *context*). Kontekst predstavlja modul koji grupiše funkcije sa zajedničkom svrhom. Direktorijum *lib* sadrži dva konteksta, odnosno modula: *MsnrApi* koji enkapsulira svu domensku i poslovnu logiku i modul *MsnrApiWeb* koji definiše veb interfejs. Podmoduli se definišu u direktorijumima *msnr_api* i *msnr_api_web*. Kontekst *MsnrApi* se odnosi na *Model* u smislu arhitekture *MVC* — *Model-View-Controller* i u njemu će se definisati svi entiteti i funkcije za rad sa njima. Inicijalno su kreirani:

- *MsnrApi.Application* — modul zadužen za pokretanje aplikacije,

- *MsnrApi.Repo* — modul zadužen za komunikaciju sa bazom,
- *MsnrApi.Mailer* — modul zadužen za slanje elektronske pošte.

Kontekst *MsnrApiWeb* sadrži implementaciju za poglede i upravljače unutar arhitekture *MVC*. Direktorijum *controllers* je inicijalno prazan, dok su u direktorijumu *views* kreirani pomoćni moduli za prikazivanje grešaka. Moduli *MsnrApiWeb.Endpoint* i *MsnrApiWeb.Router* imaju ulogu da pripreme *HTTP* zahtev i proslede ga odgovarajućem upravljaču.

Direktorijum *priv* sadrži sve resurse koji su potrebni u produkciji ali nisu direktan deo izvornog koda. U slučaju portala MSNR sadržaće skripte vezane za menjanje i popunjavanje baze podataka. Testiranje, kao ni telemetrija, nije tema ovog rada.

Obrada HTTP zahteva

U osnovi obrade *HTTP* zahteva nalazi se biblioteka *Plug* [26], koja se koristi gotovo kroz sve faze obrade zahteva. Osnovne komponente razvojnog okvira *Phoenix*, kao što su *Endpoint*, ruter i upravljači, su interno utikači (eng. *plug*). Utikač predstavlja funkciju čija je ulazna i povratna vrednost *Plug.Conn* struktura koja sadrži sve informacije o primljenom *HTTP* zahtevu. Veb server daje početne podatke za obradu zahteva, nakon čega *Phoenix* poziva utikače jedan za drugim, gde svaki utikač transformiše strukturu *Plug.Conn* dok se obrada ne završi i odgovor pošalje korisniku. Za upotrebu utikača neophodan je veb server i njegovo povezivanje sa utikačem, stoga je automatski ubačena zavisnost `plug_cowboy` u *mix.exs* — *Cowboy* je najpopularniji veb server u ekosistemu *Erlang*.

```
defmodule MsnrApiWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :msnr_api

  plug Plug.Static, ...
  plug Plug.RequestId
  plug Plug.Telemetry, ...
  plug Plug.Parsers, ...
  plug Plug.MethodOverride
  plug Plug.Head
  plug Plug.Session, ...
  plug MsnrApiWeb.Router
end
```

Primer koda 66: Utikači modula *Endpoint*

Modul *MsnrApiWeb.Endpoint* predstavlja početnu tačku prilikom obrade *HTTP* zahteva i sastoji se od niza utikača. Kostur modula prikazan je u primeru koda

66. Navedeni utikači imaju različite uloge — za prikazivanje statičkih datoteka, za logovanje, za parsiranje i druge. Svaki zahtev prolazi kroz sve navedene utikače. Postavljaju se pomoću makroa `plug`, a njihovo navođenje se može posmatrati kao korišćenje operatora prosleđivanja, što je prikazano u primeru koda 67.

```
connection
|> Plug.Static.call()
|> Plug.RequestId.call()
|> Plug.Telemetry.call()
|> Plug.Parsers.call()
|> Plug.MethodOverride.call()
|> Plug.Head.call()
|> Plug.Session.call()
|> MsnrApiWeb.Router.call()
```

Primer koda 67: Prikaz pozivanja utikača pomoću operatora `|>`

Ruter

Poslednji utikač modula *Endpoint* je *Router* koji se takođe sastoji od utikača, ali se oni grupišu u tokove (eng. *pipeline*). Unutar modula *Router*, na osnovu putanje iz zahteva definišu se opsezi (eng. *scope*), u kojima se navodi tok utikača koji će se koristiti i upravljači koji će se pozivati za određene rute. Deo modula *Router* koji sadrži definicije toka i opsega dat je u primeru koda 68.

```
defmodule MsnrApiWeb.Router do
  use MsnrApiWeb, :router

  pipeline :api do
    plug :accepts, ["json"]
    plug MsnrApiWeb.Plugs.TokenAuthentication
  end

  scope "/api", MsnrApiWeb do
    pipe_through :api

    get "/auth/refresh", AuthController, :refresh
    post "/auth/login", AuthController, :login
    get "/auth/logout", AuthController, :logout

    resources "/activity-types", ActivityTypeController, except: [:new, :edit]
    ...
  end
end
```

Primer koda 68: Deo modula *MsnrApiWeb.Router*

U slučaju portala MSNR definisan je jedan tok i jedan opseg koji nadgleda sve zahteve koji počinju sa `/api`. Svaki od tih zahteva prolazi kroz prethodno definisa-

ni tok, nakon čega se na osnovu metoda *HTTP* zahteva i nastavka putanje poziva određena akcija, tj. funkcija iz navedenog upravljača. Na primer, za zahtev sa metodom *GET* i putanjom */api/auth/refresh* poziva se funkcija *refresh* iz upravljača *AuthController*. Za upravljač tipova aktivnosti koristi se funkcija *resource* koja predstavlja skraćenu sintaksu za navođenje ruta. Jednim pozivom se kreira mapiranje metoda za akcije (*index*, *new*, *create*, *show*, *edit*, *update*, i *delete*) koje su komplementarne REST akcijama. Poslednji argument označava da će se izostaviti akcije *new* i *edit* koje se koriste prilikom upotrebe *HTML* šablona. Takođe je moguće navesti i samo akcije koje će se koristiti pomoći atoma *:only* umesto *:except*. U primeru koda 69 prikazan je deo rezultata izvršene komande `mix phx.routes` koja izlistava sve definisane rute aplikacije i njihovo mapiranje na upravljače.

```
$ mix phx.routes
      auth_path GET    /api/auth/refresh    AuthController :refresh
      auth_path POST   /api/auth/login      AuthController :login
      auth_path GET    /api/auth/logout     AuthController :logout
activity_type_path GET    /api/activity-types  ActivityTypeController :index
activity_type_path GET    /api/activity-types/:id ActivityTypeController :show
activity_type_path POST   /api/activity-types  ActivityTypeController :create
activity_type_path PATCH  /api/activity-types/:id ActivityTypeController :update
activity_type_path PUT    /api/activity-types/:id ActivityTypeController :update
activity_type_path DELETE /api/activity-types/:id ActivityTypeController :delete
...
```

Primer koda 69: Izlistavanje ruta komandom `mix phx.routes`

Upravljači i pogledi

Budući da je upravljač takođe utikač, njegova osnovna uloga je da sakupi i pripremi podatke za sledeći korak obrade korisničkog zahteva. To može uključiti pozivanje baze podataka, eksternog veb servisa ili obradu neke statičke datoteke. Na primeru upravljača *ActivityTypeController* u primeru koda 70 može se videti generalna struktura upravljača i implementacija akcije *:index*.

Poslednji poziv u akciji upravljača je funkcija **render** definisana u modulu *ActivityTypeView* koja za argumente ima *Plug.Conn* strukturu, naziv šablona i podatke potrebne za iscertavanje. U slučaju *JSON* veb interfejsa kreira se mapa koja se prevodi u *JSON* objekat, čime se završava obrada zahteva unutar razvojnog okvira *Phoenix*.

```
defmodule MsnrApiWeb.ActivityTypeController do
  use MsnrApiWeb, :controller

  alias MsnrApi.ActivityTypes

  def index(conn, _params) do
    activity_types = ActivityTypes.list_activity_types()
    render(conn, "index.json", activity_types: activity_types)
  end

  def create(conn, %{"activity_type" => activity_type_params}) do
    ...
  end

  def show(conn, %{"id" => id}) do
    ...
  end

  def update(conn, %{"id" => id, "activity_type" => activity_type_params}) do
    ...
  end

  def delete(conn, %{"id" => id}) do
    ...
  end
end
```

Primer koda 70: Primer strukture upravljača

Komunikacija sa bazom podataka

Za komunikaciju sa bazom podataka u *Elixir* projektima pretežno se koristi biblioteka *Ecto* [24]. Ona nije sastavni deo razvojnog okvira *Phoenix*, ali se podrazumevano uključuje i prilikom kreiranja *Phoenix* projekta dodaju se potrebni *Ecto* moduli i konfiguracija. Ukoliko nije potrebno korišćenje ove biblioteke prilikom kreiranja projekta navodi se parametar `--no-ecto`.

Biblioteka *Ecto* pruža jednostavan način za izvršavanje upita, transakcija, opisanje kako se strukture mapiraju u određene tabele i navođenje odnosa modula. Ima specifičan koncept pod nazivom *skup promena* (eng. *changeset*) koji enkapsulira ceo proces prihvatanja eksternih podataka, validiranja, konvertovanja i proveravanja dodatnih uslova pre nego što se upišu u bazu. Osnovni moduli biblioteke su:

- *Ecto.Repo* — za definisanje omotača oko baze podataka preko kog se ostvaruje komunikacija sa bazom, opisuje gde se nalaze podaci,
- *Ecto.Schema* — za definisanje mapiranja eksternih podataka u *Elixir* strukture, opisuje šta su podaci,

- *Ecto.Query* — za definisanje upita u *Elixir* sintaksi, opisuje kako se čitaju podaci,
- *Ecto.Changeset* — za praćenje i validiranje izmena, opisuje kako se menjaju podaci.

Za integraciju sa bibliotekom *Ecto*, *Phoenix* uključuje paket `phoenix_ecto`, adapter za bazu podataka *PostgreSQL* — `postgrex` i paket `ecto_sql` koji pored prethodno navedenih *Ecto* osobina ima i podršku za migracije, što omogućava menjanje i praćenje strukture baze tokom vremena. U generisanom modulu *MsnrApi.Repo* definiše

```
config :msnr_api,  
  ecto_repos: [MsnrApi.Repo]
```

Primer koda 71: Navođenje *Ecto* repozitorijuma u konfiguraciji

se repozitorijum i adapter koji se koristi. Repozitorijum koji se koristi neophodno je navesti u glavnoj konfiguraciji *config.exs*, što je prikazano u primeru koda 71. *Ecto* podržava rad sa više repozitorijuma istovremeno, zato se navodi lista naziva repozitorijuma. Parametri za definisanje komunikacije sa bazom podataka postavljaju se posebno za svako okruženje. U primeru koda 72 prikazana je konfiguracija za razvojno okruženje (*config/dev.exs*) i pokretanjem komande `mix ecto.create` lokalno se kreira `msnr_api_dev` baza podataka.

```
config :msnr_api, MsnrApi.Repo,  
  username: "postgres",  
  password: "postgres",  
  database: "msnr_api_dev",  
  hostname: "localhost",  
  show_sensitive_data_on_connection_error: true,  
  pool_size: 10
```

Primer koda 72: Konfiguracija baze podataka u razvojnom okruženju

Definisanje entiteta

Budući da kontekst *MsnrApi* sadži domensku logiku aplikacije, u njemu su definisani svi entiteti aplikacije, takođe kao konteksti. Na primeru tipa aktivnosti, pod direktorijumom *msnr_api* nalazi se direktorijum *activity_types* sa definicijom strukture *ActivityType* kojom se predstavlja entitet i datoteka *activity_types.ex* u kojoj je definisan modul *ActivityTypes* sa funkcijama za rad sa entitetom.

Ecto omogućava definisanje struktura čija se pojedinačna polja povezuju sa kolonama u bazi podataka. U primeru koda 73 prikazano je definisanje entiteta tipa aktivnosti korišćenjem modula *Ecto.Schema*. Makroi `schema` i `field` definišu isto-

```
use Ecto.Schema
import Ecto.Changeset

schema "activity_types" do
  field :content, :map
  field :description, :string
  field :has_signup, :boolean, default: false
  field :is_group, :boolean, default: false
  field :name, :string
  field :code, :string

  has_many :activities, MsnrApi.Activities.Activity

  timestamps()
end
...
```

Primer koda 73: Upotreba *Ecto.Schema* na primeru tipa aktivnosti

vremeno tabelu u bazi i Elixir strukturu. Svako polje definisano makroom `field` odgovara polju u strukturi *ActivityType* i polju, odnosno koloni, tabele *activity_types*. Makro `field` ima tri argumenta — naziv polja, tip i opcije. U prikazanom primeru korišćena je opcija za postavljanje podrazumevane vrednosti kolone. Funkcija `timestamps` dodaje još dva polja, `created_at` i `inserted_at` za praćenje promena entiteta u bazi.

Makroom `has_many` opisuje se asocijacija entiteta — jedan tip aktivnosti se može navesti u više aktivnosti. Sa druge strane, svaka aktivnost ima svoj tip, tako da se u modulu *MsnrApi.Activities.Activity* asocijacija sa odgovarajućim tipom aktivnosti definiše pomoću `belongs_to`. Pored navedenih, za druge tipove asocijacija mogu se koristiti makroi `has_one` i `many_many`.

Pored prikazanih polja, *Ecto* podrazumevano dodaje i `:id` polje koje predstavlja primarni ključ tabele. U slučaju kada je potrebno drugačije definisati primarni ključ koristi se atribut `@primary_key`, što se može videti na primeru definisanja studenta.

Migracije

Kreiranjem šeme entiteta, definiše se način na koji *Ecto* komunicira sa bazom, dok se kreiranje odgovarajućih tabela u bazi vrši migracijama. Migracije su predstavljane modulima u kojima se definišu promene koje će se izvršiti nad bazom i pružaju

mogućnost da se izvršene promene ponište. Kreiraju se pomoću *Mix* komande `mix ecto.gen.migration <naziv migracije>` i nalaze se u direktorijumu `priv/repo/migrations`, a naziv datoteke je oblika `<vreme kreiranja>_<naziv migracije>.exs`. U primeru koda 74 prikazana je migracija kojom se kreira tabela `activity_types` i dodaju jedinstveni indeksi za naziv i kôd tipa aktivnosti.

```
defmodule MsnrApi.Repo.Migrations.CreateActivityTypes do
  use Ecto.Migration

  def change do
    create table(:activity_types) do
      add :name, :string, null: false
      add :code, :string, null: false
      add :description, :string, null: false
      add :has_signup, :boolean, default: false, null: false
      add :is_group, :boolean, default: false, null: false
      add :content, :map

      timestamps()
    end

    create unique_index(:activity_types, [:name])
    create unique_index(:activity_types, [:code])
  end
end
```

Primer koda 74: Migracija za kreiranje tabele `activity_types`

Migracije se mogu definisati na dva načina — prvi, predstavljen u primeru koda, koristi funkciju `change` i drugi sa dve funkcije `up` i `down`, kojima se definišu akcije koje se pozivaju prilikom izvršavanja i poništavanja migracije. U slučaju funkcije `change` *Ecto* sam zaključuje potrebne korake prilikom poništavanja migracije. Prilikom implementacije portala MSNR funkcije `up` i `down` korišćene su za definisanje tipova uloga korskika i statusa registracije.

Kreiranje, preuzimanje i brisanje podataka iz baze

Ecto pruža brojne funkcije za interakciju sa bazom podataka kroz modul *Ecto.Repo*. U tabeli 5.1 prikazane su najčešće korišćene funkcije i njihov kratak opis.

Funkcije `get` i `one` imaju verzije sa znakom `!` na kraju naziva (npr. `get!`) koje u slučaju da red nije pronađen izbacuju izuzetak. U slučaju funkcija za modifikaciju podataka, takođe postoje pandan funkcije sa `!` u nazivu i prilikom greške izbacuju izuzetak. One prilikom uspešnog izvršavanja vraćaju samo red, bez torke.

Funkcija	Opis	Povratna vrednost
all	Vraća svaki red iz tabele za zadati upit	Lista redova
get	Vraća jedan red za zadati primarni ključ	Red ili <code>null</code> ako nije pronađen
one	Vraća jedan red za zadati upit	Red ili <code>null</code> ako nije pronađen, ukoliko pronađe više redova izbacuje izuzetak
delete	Briše red iz tabele na osnovu primarnog ključa iz strukture	<code>{:ok, red}</code>
insert	Dodaje red u tabeli na osnovu date strukture	<code>{:ok, red}</code> ili <code>{:error, changeset}</code>
update	Ažurira red u tabeli na osnovu date strukture sa primarnim ključem	<code>{:ok, red}</code> ili <code>{:error, changeset}</code>

Tabela 5.1: Funkcije modula *Ecto.Repo* za rad sa bazom podataka

U primeru koda 75 prikazan je modul *MsnrApi.ActivityTypes* i način upotrebe nekih od funkcija iz tabele. Za izlistavanje tipova aktivnosti, funkciji `all` se prosleđuje samo naziv tabele, tako da funkcija vraća sve redove.

Definisanje upita

Modul *Ecto.Query* pruža sintaksu za pisanje upita koja je veoma slična samoj *SQL* sintaksi. S tim da sintaksa upita započinje makroom `from` nakon čega se ostali delovi upita definišu pomoću liste ključnih reči. Praksa je da se ključna reč `:select` navodi na kraju, kao kod *LINQ* [25] sintakse iz razvojnog okvira *.NET* koja je bila inspiracija za *Ecto*, a može se i izostaviti. Za interpolaciju parametara koristi se operator `^`. Primer jednostavnog upita dat je u primeru koda 76 gde je prikazana funkcija za prikazivanje studentskih registracija iz modula *StudentRegistrations*.

Modul *Ecto.Query* pruža mogućnost definisanja dinamičnih upita, podupita i dosta drugih karakteristika o kojima se može videti više na zvaničnoj stranici.

Izmena entiteta

U primeru koda 75 tokom kreiranja i izmene tipa aktivnosti poziva se funkcija *ActivityType.changeset* kojom se kreira skup promena. Prilikom kreiranja podataka se može koristiti i struktura definisana sa *Ecto.Schema*, dok je menjanje


```
defmodule MsnrApi.ActivityTypes do
  alias MsnrApi.Repo

  alias MsnrApi.ActivityTypes.ActivityType

  def list_activity_types do
    Repo.all(ActivityType)
  end

  def get_activity_type!(id), do: Repo.get!(ActivityType, id)

  def create_activity_type(attrs \\ %{}) do
    %ActivityType{}
    |> ActivityType.changeset(attrs)
    |> Repo.insert()
  end

  def update_activity_type(%ActivityType{} = activity_type, attrs) do
    activity_type
    |> ActivityType.changeset(attrs)
    |> Repo.update()
  end

  def delete_activity_type(%ActivityType{} = activity_type) do
    Repo.delete(activity_type)
  end
end
```

Primer koda 75: Definicija modula *MsnrApi.ActivityTypes*

```
import Ecto.Query

def list_student_registrations(semester_id) do
  Repo.all(
    from sr in StudentRegistration, where: sr.semester_id == ^semester_id
  )
end
```

Primer koda 76: Primer upita definisanog pomoću modula *Ecto.Query*

podataka moguće samo korišćenjem strukture *Ecto.Changeset*. Kreiranje strukture *Ecto.Changeset* funkcijom *changeset* prikazano je u primeru koda 77.

```
def changeset(activity_type, attrs) do
  activity_type
  |> cast(attrs, [:code, :name, :description, :has_signup, :is_group, :content])
  |> validate_required([:code, :name, :description, :content])
  |> unique_constraint(:name)
  |> unique_constraint(:code)
end
```

Primer koda 77: Definicija funkcije *changeset*

Funkcija *cast* je pretežno prva u nizu funkcija koje se pozivaju i ograničava polja

koja se mogu promeniti. U prikazanom primeru, ne mogu se izmeniti polja `:id`, `:created_at` i `:inserted_at`. Polje `:inserted_at` će izmeniti sam *Ecto* prilikom izvršavanja funkcije `update`.

Zatim se funkcijama `validate_required` i `unique_constraint` proveravaju obavezna polja i jedinstvenost naziva i koda. Rezultat izvršavanja niza funkcija nad strukturom *Ecto.Changeset* je takođe struktura *Ecto.Changeset*. Rezultujuća struktura sadrži mnoge informacije među kojima su promene koje treba izvršiti, validnost izmena i greške validacije ukoliko ih ima. Modul *Ecto.Changeset* pruža brojne funkcije za validiranje, a moguće je definisati i prilagođene funkcije.

Generisanje datoteka

Prilikom definisanja entiteta, dodaje se njegov kontekst sa *Ecto.Schema* strukturom, kao i migracija. Ukoliko je potrebno dodati i upravljač, to su još dve datoteke — upravljač i pogled. Da bi se olakšao ovaj proces, *Phoenix* pruža komande za automatsko generisanje navedenih datoteka zajedno sa šablonima za testove. Za potrebe portala MSNR najčešće korišćena komanda jeste `mix phx.gen.json` koja generiše sve prethodno navedene datoteke. Prethodno prikazani listinzi vezani za *ActivityType* entitet rezultat su komande prikazane u primeru koda 78. Parametri su redom: naziv konteksta, naziv strukture i naziv tabele u bazi, nakon čega slede definicije polja, odnosno kolona. Razvojni okvir *Phoenix* pruža veliki broj koman-

```
mix phx.gen.json ActivityTypes ActivityType activity_types code:string:unique
↳ name:string:unique description:string has_signup:boolean is_group:boolean
↳ content:map
```

Primer koda 78: Upotreba `mix phx.gen.json` komande na primeru *ActivityType* entiteta

di za generisanje, kao što su `mix phx.gen.context`, `mix phx.gen.schema` i mnoge druge, sa mnogim dodatnim opcijama, o čemu se može pročitati više na zvaničnoj dokumentaciji.

5.2 Registrovanje studenata

Registrovanje studenata može se podeliti u tri celine. Prva se odnosi na podnošenje, druga na prihvatanje zahteva i kreiranje korisničkog naloga, a treća na završno podešavanje naloga — postavljanje lozinke.

Prve dve celine se izvršavaju unutar upravljača *StudentRegistrationController* čije su osnovne akcije `create` za podnošenje zahteva, `index` za izlistavanje svih zahteva u toku trenutnog semestra i `update` za obradu zahteva. Akcija za izlistavanje zahteva se razlikuje od drugih po tome što uključuje i informaciju o semestru, tako da se za akciju `index` definiše putanja pod upravljačem *SemesterController*. Ovo je moguće zato što razvojni okvir *Phoenix* omogućava definisanje potputanja, kao i korišćenje jednog upravljača u više putanja. Putanje koje se koriste prilikom registrovanja studenata prikazane su u primeru koda 79. U poslednjem koraku registrovanja studenata koristi se upravljač *PasswordController*.

```
resources "/passwords", PasswordController, only: [:update]
resources "/registrations", StudentRegistrationController, except: [:new, :edit, :index]

resources "/semesters", SemesterController, except: [:new, :edit] do
  resources "/registrations", StudentRegistrationController, only: [:index]
...

```

Primer koda 79: Definisanje putanje za registrovanje studenata

Unutar portala MSNR putanje su organizovane hijerarhijski, budući da se sve aktivnosti vrše u toku semestra, putanja `/semesters` se nalazi na vrhu hijerarhije. Delom koda prikazanim u prethodnom primeru koda registruju se putanje `api/semesters` za rad sa semestrima i `api/semesters/<semester_id>/registrations` za izlistavanje registracija. Vrednost `<semester_id>` iz putanje prosleđuje se kao parametar akciji upravljača što je prikazano u primeru koda 80.

```
def index(conn, %{"semester_id" => semester_id}) do
  student_registrations = StudentRegistrations.list_student_registrations(semester_id)
  render(conn, "index.json", student_registrations: student_registrations)
end

```

Primer koda 80: Definicija akcije `index` u upravljaču *StudentRegistrationController*

Podnošenje zahteva se izvršava slanjem *POST HTTP* zahteva za čiju obradu je dovoljan kôd generisan komandom `mix phx.gen.json`, s tim da je prilikom kreiranja potrebno povezati zahtev sa aktivnim semestrom. Sa druge strane, obrada zahteva od strane profesora unutar akcije `update` je dosta kompleksnija. Unutar transakcije je potrebno ažurirati zahtev, kreirati nalog za studenta, ukoliko je prihvaćen, i poslati mail. Funkcija za prihvatanje zahteva unutar modula *StudentsRegistrations* prikazana je u primeru koda 81. Struktura *Ecto.Multi* se koristi za grupisanje više operacija nad repozitorijumom, pri čemu se svakoj operaciji daje jedinstveno ime koja će identifikovati njen rezultat u slučaju uspeha ili neuspeha. Svaka od operacija mora vraćati rezultat oblika `{:ok, vrednost}` ili `{:error, greška}`.

```
def update_student_registration(  
  %StudentRegistration{} = registration,  
  %{"status" => "accepted"} = attrs  
) do  
  create_user = fn _repo, %{student_registration: reg} ->  
    Accounts.create_student_account(reg)  
  end  
  
  create_student = fn _repo, %{user: user} ->  
    Students.create_student(user, %{index_number: registration.index_number})  
  end  
  
  send_email = fn _repo, %{user: user} ->  
    Emails.accept(user) |> Mailer.deliver()  
  end  
  
  Multi.new()  
  |> Multi.update(  
    :student_registration,  
    StudentRegistration.changeset(registration, attrs))  
  |> Multi.run(:user, create_user)  
  |> Multi.run(:student, create_student)  
  |> Multi.run(:email, send_email)  
  |> MsnrApi.Repo.transaction()  
end
```

Primer koda 81: Prihvatanje zahteva za registraciju studenta

Za slanje mejlova koristi se biblioteka *Swoosh*, koja je, kao i *Ecto*, uključena prilikom kreiranja projekta. Modul *MsnrApi.Mailer*, definisan slično kao i *MsnrApi.Repo*, sadrži funkcije za slanje mailova. Unutra modula *MsnrApi.Emails* definišu se funkcije kojima se kreira sastav mejlova pomoću modula *Swoosh.Email*.

Da bi student mogao da se prijavi na portal, potrebno je da postavi lozinku, čime se zaokružuje proces registracije. U primeru koda 82 prikazan je upravljač *PasswordController* sa akcijom `update` koja je zadužena za postavljanje lozinke. Parametri `email` i `password` dolaze iz tela zahteva, a `id` iz putanje i očekuje se da odgovara vrednosti kolone `password_url_path` iz tabele *users*.

Nakon uspešne verifikacije korisnika, unutar funkcije `set_password` poziva se `User.password_changeset` kojom se proverava dužina lozinke i vrši kriptovanje lozinke. Ukoliko i provera lozinke prođe uspešno, šalje se odgovor sa statusom 200 bez sadržaja.

Unutar svakog upravljača makroom `action_fallback` se registruje utikač, odnosno upravljač, *MsnrApiWeb.FallbackController* koji se poziva u slučaju da dođe do greške prilikom izvršavanja akcije. U slučaju da korisnik nije verifikovan, funkci-

```
defmodule MsnrApiWeb.PasswordController do
  use MsnrApiWeb, :controller

  alias MsnrApi.Accounts

  action_fallback MsnrApiWeb.FallbackController

  def update(conn, %{"id" => uuid, "email" => email, "password" => password}) do
    with {:ok, user} <- Accounts.verify_user(%{email: email, uuid: uuid}),
         {:ok, _} <- Accounts.set_password(user, password) do
      send_resp(conn, :no_content, "")
    end
  end
end
```

Primer koda 82: Definicija upravljača *MsnrApiWeb.PasswordController*

Ja `verify_user` vraća `{:error, :unauthorized}` i pošto u `with` izrazu taj slučaj nije obrađen dolazi do greške. Greška se ne prosleđuje odmah korisniku već se pomoću upravljača *FallbackController* prevodi u 401 grešku i korisniku vraća validan odgovor.

5.3 Autentifikacija i autorizacija korisnika

Autentifikacija i autorizacija korisnika portala MSNR zasnovana je na tokenima. Nakon uspešnog prijavljivanja korisniku se prosleđuje token koji se prilikom svakog *HTTP* zahteva šalje serveru kroz zaglavlje *Authorization*. Token nosi informacije o korisniku i ima ograničeno vreme trajanja, nakon čega više nije validan. Pomoću utikača se prilikom obrade zahteva validira token i proveravaju prava pristupa određenim resursima.

Unutar implementacije portala MSNR koriste se dva tokena:

- *access_token* — Sadrži informacije o korisniku i koristi se za pristupanje portalu. Ima kraće vreme trajanja (do pola sata) i korisniku se prosleđuje kroz telo *HTTP* odgovora.
- *refresh_token* — Koristi se za obnavljanje *access_token*, ima duže vreme trajanja (nekoliko dana) i prosleđuje se u *HttpOnly* i *Secure* kolačiću.

Prijavljivanje korisnika izvršava se unutar akcije `login` upravljača *AuthController* i način obrade zahteva je sličan akciji `refresh` za obnavljanje tokena. U prvoj se korisnik verifikuje na osnovu imejla i lozinke, a u drugoj na osnovu sadržaja kolačića,

nakon čega se kreiraju novi tokeni. Razvojni okvir *Phoenix* ima ugrađenu podršku za rad sa tokenima i preko modula *Phoenix.Token* pruža funkcije za potpisivanje i verifikaciju tokena koje se koriste unutar modula *MsnrApiWeb.Authentication*. Akcijom `logout` se uklanja token za obnavljanje iz kolačića i kada token za pristup istekne potrebno je ponovno prijavljivanje.

Utikač za autentifikaciju korisnika

Uključivanje utikača *MsnrApiWeb.Plugs.TokenAuthentication* prikazano je zajedno sa registrovanjem akcija upravljača *AuthController* u primeru koda 68 u poglavlju 5.3 prilikom opisa obrade *HTTP* zahteva. Njegova uloga jeste da, ukoli-

```
defmodule MsnrApiWeb.Plugs.TokenAuthentication do
  import Plug.Conn

  def init(opts), do: opts

  def call(conn, _opts) do
    with ["Bearer " <> token] <- get_req_header(conn, "authorization"),
        {:ok, payload} <- MsnrApiWeb.Authentication.verify_access_token(token) do
      assign(conn, :user_info, payload)
    else
      _ ->
        assign(conn, :user_info, nil)
    end
  end
end
```

Primer koda 83: Definicija utikača *MsnrApiWeb.Plugs.TokenAuthentication*

ko se token verifikuje, informacije o korisniku iz tokena dodele trenutnoj konekciji tako da se one mogu koristiti dalje u upravljačima. Definicija utikača *MsnrApiWeb.Plugs.TokenAuthentication* prikazana je u primeru koda 83.

Utikač za autorizaciju studenata

Utikači se mogu definisati na dva načina. Prvi, pomoću modula, prikazan je na primeru utikača za autentifikaciju, i drugi, pomoću funkcije, kojim su definisani utikači za autorizaciju. Autorizacija pomoću utikača se zasniva na proveru uloge korisnika i uključuju se različito za svaki upravljač. Unutar modula *MsnrApiWeb.Plugs.Authorization* definisane su funkcije `professor_only` i `allow_students` koje imaju iste argumente kao i funkcija `call` (primer koda 83) u slučaju modula.

Prilikom provere studentske uloge proverava se, ukoliko postoji, i *student_id* parametar iz putanje. Upotreba utikača se može ograničiti na određene akcije unutar upravljača, što se može videti na primeru upravljača *AssignmentController* (primer koda 84).

```
defmodule MsnrApiWeb.AssignmentController do
  use MsnrApiWeb, :controller
  import MsnrApiWeb.Plugs.Authorization

  action_fallback MsnrApiWeb.FallbackController

  plug :allow_students when action in [:index]
  plug :professor_only when action in [:create, :update, :delete, :show]
  ...
```

Primer koda 84: Primer upotrebe utikača za autorizaciju

5.4 Ubacivanje i dodela aktivnosti

Osnovni tipovi aktivnosti (prijava grupe, predaja CV-a, rada i recenzije) biće inicijalno ubačene u bazu prilikom izvršavanja skripte *seed.exs*. Sadržaj tipova aktivnosti je tipa `:map`, što odgovara *JSON* objektu u bazi. Fleksibilna struktura sadržaja je uvedena sa idejom da se kasnije mogu dodavati i drugačiji tipovi aktivnosti (npr. forme). Upravljanje tipovima aktivnosti vrši se preko *ActivityTypeController* upravljača.

Rad sa aktivnostima vrši se unutar upravljača *ActivityController* čije su akcije `index` i `create` pod putanjom *semesters*. Kreiranjem aktivnosti za trenutni semestar, ona se istovremeno dodeljuje studentima ili grupama u zavisnosti od tipa aktivnosti. Takođe, ukoliko postoji prijava za aktivnost dodela se vrši samo studentima koji su se prijavili. Neki tipovi aktivnosti, poput recenzija, zahtevaju dodatnu obradu, ali osnovni princip dodele je isti za sve i izvršava se unutar funkcije `create_activity` unutar *Activities* (primer koda 85).

Funkcija `create_assignments_for_activity` na osnovu aktivnosti kreira odgovarajuću listu dodeljenih aktivnosti.

5.5 Izvršavanje i ocenjivanje aktivnosti

Upravljač *AssignmentController* studentima pruža pristup samo akciji `index`, tj. preko ovog upravljača studentima je moguće samo prikazivanje njihovih dodeljenih

```
def create_activity(str_semester_id, attrs) do
  {semester_id, _} = Integer.parse(str_semester_id)
  activity_changeset =
    %Activity{semester_id: semester_id}
    |> Activity.changeset(attrs)

  multi_struct = Multi.new()
  |> Multi.insert(:activity, activity_changeset)
  |> Multi.insert_all(:assignments, Assignment, &create_assignments_for_activity/1)

  case MsnrApi.Repo.transaction(multi_struct) do
    {:ok, %{activity: activity}} -> {:ok, activity}
    {:error, :activity, changeset, _changes} -> {:error, changeset}
    _ -> {:error, :bad_request}
  end
end
```

Primer koda 85: Kreiranje i dodela aktivnosti

aktivnosti. Za izvršavanje aktivnosti kreirani su drugi upravljači.

Jedne od osnovnih aktivnosti su prijavljivanje grupe i odabir teme. Prijavljivanje grupe nije grupna aktivnost, već se dodeljuje svakom studentu. Dodeljena aktivnost prijavljivanja grupe se izvršava pozivom akcije `create` upravljača *GroupController*. Prilikom prijave, unutar transakcije, se kreira nova grupa i dodeljuje prijavljenim studentima, takođe se svim studentima obeležava da je aktivnost ispunjena. Odabir teme se takođe izvršava unutar upravljača *GroupController*, pozivanjem akcije `update` grupi se dodeljuje izabrana tema. Upravljač *TopicController* se koristi prilikom dodavanja i izlistavanja tema.

Prijavljivanje studenata za aktivnosti koje zahtevaju prijavu, vrši se unutar posebnog upravljača — *SignupController*. On ima samo akciju `update` preko koje se ažurira kolona *completed* u tabeli *assignments*. Ukoliko je vrednost kolone *true* smatra se da je student izvršio prijavu.

Najveći broj aktivnosti odnosi se na prilaganje dokumenata. Rad sa dokumentima izvršava se unutar upravljača *DocumentController*, čije se akcije izlistavanja i kreiranja nalaze pod putanjom *assignments*. Kao i u prethodnim slučajevima, prilikom prilaganja dokumenta, izvršava se dodeljena aktivnost. Priloženi dokumenti se čuvaju na serveru u direktorijumu podešenom u konfiguraciji. Tip i broj dokumenata koji se prilaže određen je sadržajem tipa aktivnosti. *JSON* objekat za prilaganje dokumenata prikazan je na primeru recenzije u primeru koda 86. Unutar polja `files` nalazi se niz objekata koji opisuju dokumenta koja je potrebno predati. Dokumenta se predaju kroz *multipart* sadržaj *HTTP* zahteva i telo zahteva čine dva


```
{
  "files" : [
    {"name" : "Recenzija", "extension" : ".pdf"},
    {"name" : "Recenzija", "extension" : ".tex"}
  ]
}
```

Primer koda 86: Primer upotrebe utikača za autorizaciju

niza — `documentsIds` koji sadži niz identifikatora dokumenata koji nastaju konkatencijom polja `name` i `extension`, i niza `documents` koji sadrži sama dokumenta. Prilikom obrade zahteva proverava se prosledena struktura sa odgovarajućim sadržajem tipa aktivnosti. Takođe, polje `name` učestvuje u organizaciji dokumenata na serveru, a priloženi dokumenti se imenuju na osnovu tipa aktivnosti i studenta ili grupe kojoj je aktivnost dodeljena. Izmena priloženih dokumenata se vrši preko akcije `update` upravljača *DocumentController*, Nije moguće promeniti više dokumenata istovremeno i čuva se samo poslednja verzija promenjenog dokumenta.

Profesor ocenjuje aktivnosti pozivanjem akcije `update` upravljača *AssignmentController*, unoseći odgovarajući komentar i broj poena. Takođe, preko upravljača *DocumentController* profesor može priložiti dokument, ukoliko je to potrebno

Glava 6

Implementacija klijentskog dela portala

Za razliku od *Phoenix* projekata, inicijalizacijom *Elm* projekta kreira se samo prazan direktorijum *src* i datoteka *elm.json*, tako da se kreiranje i organizacija datoteka u potpunosti prepušta korisniku. Rešenje organizacije *Elm* datoteka u slučaju portala MSNR prikazano je na slici 6.1.



Slika 6.1: Organizacija *Elm* datoteka unutar portala MSNR

Aplikacija je podeljena na različite stranice koje se prikazuju u zavisnosti od

trenutne *url* putanje. U korenu projekta nalazi se osnovna datoteka *Main.elm* u kojoj se nalazi funkcija `main` sa definicijom *Elm* aplikacije. Zajedno sa njom su i datoteke koje sadrže definicije osnovnih stranica, entiteta, putanja i modula za komunikaciju sa serverom, a u posebnim direktorijumima odvojene su datoteke za prikazivanje studentskih i profesorskih stranica. U nastavku rada, zajedno sa načinom rada *Elm* aplikacije, biće prikazana uloga navedenih modula.

6.1 Elm aplikacija

U uvodnom delu rada, na primeru brojača, prikazan je najjednostavniji primer *Elm* programa kreiran funkcijom `sandbox`. Pored nje, u modulu `Browser` se nalaze funkcije `element`, `document` i `application` kojima je implementiran korisnički interfejs portala MSNR.

Prethodno navedene četiri funkcije poredane su po kompleksnosti i svaka naredna pruža dodatne mogućnosti. Funkcijom `element` se uvodi mogućnost komunikacije sa „spoljašnjim svetom” pomoću koncepta komande, supskripcije, portova i oznaka (*flags*). Pruža kontrolu nad jednim *HTML* elementom i ova funkcija je veoma bitna za *Elm* adaptaciju, jer omogućava laku integraciju u postojeći *JavaScript* projekat. Funkcija `document` proširuje funkciju `element` time što upravlja celim dokumentom i pruža kontrolu nad *HTML* elementima `<title>` i `<body>`. Na kraju, funkcija `application` kreira aplikaciju koja upravlja *url* promenama, u primeru koda 87 prikazana je njena anotacija.

```
application :  
{ init : flags -> Url -> Key -> ( model, Cmd msg )  
  , view : model -> Document msg  
  , update : msg -> model -> ( model, Cmd msg )  
  , subscriptions : model -> Sub msg  
  , onUrlRequest : UrlRequest -> msg  
  , onUrlChange : Url -> msg  
}  
-> Program flags model msg
```

Primer koda 87: Tipovi funkcija

Oznakama (*flags*) je moguće prosleđivanje podataka *Elm* programu iz *JavaScript* koda. Komande (`Cmd`) služe za izvršavanje akcija izvan okruženja *Elm* (npr. *HTTP* zahtevi), a supskripcijama (`Sub`) je moguće pretplatiti se na određeni događaj. Polja `onUrlRequest` i `onUrlChange`, kao i parametar `Url` funkcije `init`, koriste se za rad

sa putanjama prilikom implementacije jednostranične aplikacije. Navedeni koncepti i njihova upotreba unutar portala MSNR biće detaljnije objašnjeni dalje u radu.

Inicijalizacija aplikacije

Tip funkcije `init` prikazan je u primeru koda 87 i kao argumente prihvata oznake, trenutnu putanju i ključ navigacije, a povratna vrednost je torka modela i komande. Ključ navigacije generiše se prilikom inicijalizacije, obavezan je parametar prilikom menjanja putanja unutar aplikacije i kao takav se mora čuvati u modelu. Pored ključa navigacije, u modelu se čuvaju informacije o trenutnom korisniku, putanji i stranici koja se prikazuje. Takođe, model sadrži podatke o aktivnostima čiji sadržaj zavisi od trenutnog korisnika, adresu veb interfejsa, dužinu trajanja tokena za pristup i oznaku da li je u toku inicijalno učitavanje aplikacije. Model aplikacije može se videti u primeru koda 88, a definisan je u datoteci *Main.elm*.

```
type alias Model =  
{ currentUser : UserType  
  , currentRoute : Route  
  , currentPage : Page  
  , accessTokenExpiresIn : Float  
  , key : Nav.Key  
  , mainContent : ContentModel  
  , initialLoading : Bool  
  , apiUrl : String  
}
```

Primer koda 88: Definicija modela

Aplikacija je kompajlirana tako da se kreira datoteka *app.js* koji se uključuje u dokument *index.html*, čija je osnovna struktura prikazana u primeru koda 89. Pokreće se pozivanjem `init` funkcije iz modula `Main` i tom prilikom se vrši prosleđivanje putanje ka veb interfejsu, tj. serverskom delu portala, preko oznaka. Prilikom inicijalizacije, trenutna stranica i putanja se određuju na osnovu *url* vrednosti i čuva se vrednost prosleđena oznakama. Takođe se proverava da li je trenutni korisnik autentifikovan slanjem *HTTP* zahteva za obnovu tokena. Korisnik je podrazumevano neautentifikovan dok se ne dobije odgovor servera.

Komande

Slanje *HTTP* zahteva, generisanje slučajnih brojeva, određivanje trenutnog vremena i slične operacije mogu imati različite rezultate ukoliko se izvrše više puta.

```
<html>
  <head>
    <script src="/app.js"></script>
  </head>
  <body>
    <div id="app"></div>
    <script>
      var app = Elm.Main.init({
        node: document.getElementById("app"),
        flags: { apiBaseUrl : "http://localhost:4000/api" }
      });
    </script>
  </body>
</html>
```

Primer koda 89: Pokretanje *Elm* aplikacije

Kao takve, ne mogu se predstaviti *Elm* funkcijama već se za to koriste komande.

Komanda predstavlja vrednost koja opisuje operaciju koju okruženje *Elm* treba da izvrši. Rezultat izvršene operacije se preko poruke prosleđuje funkciji `update`. Tako da se svako izvršavanje komandi posmatra kao ulazni podatak aplikacije čime se garantuje da sve funkcije budu čiste, iako se izvršavaju operacije koje to nisu.

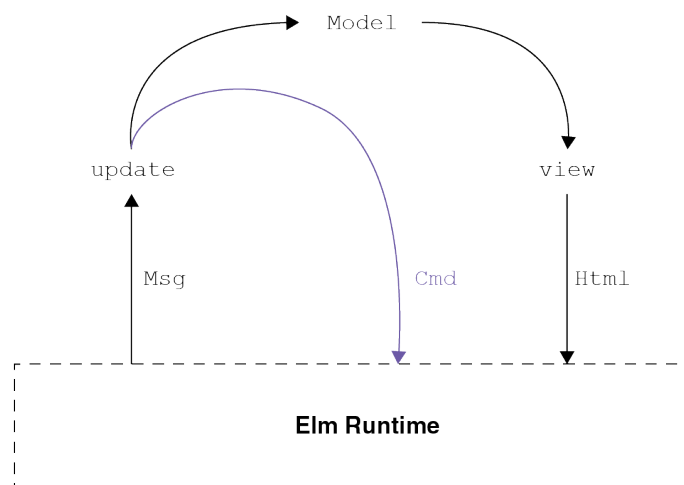
Definicija komande — `type Cmd msg` nalazi se unutar modula *Platform.Cmd* u osnovnom paketu *elm/core*, gde `msg` predstavlja tip poruke koja se vraća aplikaciji. Funkcija `Cmd.none` se koristi kada nema komandi za izvršavanje, a ukoliko je potrebno izvršiti više operacija koristi se `Cmd.batch`.

Inicijalna komanda se izvršava samo prilikom pokretanja aplikacije, centralno mesto izvršavanja komandi predstavlja funkcija `update`, čiji je povratni tip, kao i kod funkcije `init`, proširen sa `Cmd msg`. Ilustracija funkcije `update` sa komandama unutar arhitekture *Elm* prikazan je na slici 6.2.

Komunikacija sa serverom

Kreiranje komandi za slanje *HTTP* zahteva izvršava se pomoću paketa *elm/http*. Osnovna funkcija unutar paketa je `request`, čija je anotacija prikazana u primeru koda 90.

Polja `method`, `headers`, `url`, `body` i `timeout` označavaju standardna svojstva *HTTP* zahteva. Poljem `expect` se označava koji tip podataka se očekuje kao odgovor i kojom porukom će se ažurirati aplikacija. Polje `tracker` služi za praćenje progressa zahteva, ali kao ni `timeout`, nije korišćeno u aplikaciji. Pored funkcije `request` i funkcija `riskyRequest` koja ima istu anotaciju, s tim što omogućava postavljanje

Slika 6.2: Tok komandi unutar arhitekture *Elm*

```

request :
{ method : String
, headers : List Header
, url : String
, body : Body
, expect : Expect msg
, timeout : Maybe Float
, tracker : Maybe String
}
-> Cmd msg

```

Primer koda 90: Anotacija funkcije `Http.request`

kolačića od strane servera (uključuje opciju `withCredentials` [22]).

Za komuniciranje sa veb interfejsom portala MSNR kreiran je modul *Api* koji otkriva funkcije `get`, `post`, `put` i `delete` koje se oslanjaju na `request`, kao i funkcije koje se oslanjaju na `riskyRequest` — `getWithCredentials` i `postWithCredentials`. Takođe, unutar modula su navedene sve putanje veb interfejsa koje se koriste u aplikaciji. Anotacija funkcije `get` prikazana je u primeru koda 91. Prilikom slanja

```

get :
{ baseUrl : String
, endpoint : String
, token : String
, expect : Http.Expect msg
}
-> Cmd msg

```

Primer koda 91: Anotacija funkcije `Api.get`

zahteva MSNR veb interfejsu potrebno je proslediti token za autorizaciju, osnovnu

adresu i krajnju tačku (nastavak putanje) kojom se određuje resurs kom se pristupa. U slučaju `post` i `put`, dodaje se i telo zahteva pored prikazanih parametara.

Funkcije `getWithCredentials` i `postWithCredentials` koriste se za obnavljanje tokena, prijavljivanje i odjavljivanje korisnika i njihova upotreba izdvojena je u poseban modul *Session*. Deo modula *Session* sa definicijom poruka i funkcije za obnavljanje tokena prikazan je u primeru koda 92. Ukoliko u kolačiću postoji

```
type alias Session =
  { accessToken : String
  , expiresIn   : Float
  , userInfo    : UserInfo
  , semesterId  : Int
  , studentInfo : Maybe StudentInfo
  }

type Msg
  = GotSessionResult (Result Http.Error Session)
  | GotTokenResult   (Result Http.Error Session)
  | DeleteSessionResult (Result Http.Error ())

silentTokenRefresh : String -> Cmd Msg
silentTokenRefresh apiBaseUrl =
  Api.getWithCredentials
    { apiBaseUrl = apiBaseUrl
    , endpoint    = Api.endpoints.refreshToken
    , expect      = Http.expectJson GotTokenResult decodeSession
    }
```

Primer koda 92: Definicija funkcije `silentTokenRefresh`

validan token za obnavljanje, server će vratiti *JSON* objekat koji sadži token za pristupanje i informacije o trenutnom korisniku. Očekivani rezultat odgovara slogu `Session`. Funkcija `Http.expectJson` prima dva argumenta — poruka kojom se rezultat komande prosleđuje aplikaciji i dekodier kojim se *JSON* objekat mapira u slog `Session`.

Rad sa *JSON* podacima (kodiranje i dekodiranje) izvršava se pomoću modula *Json.Encode* i *Json.Decode* iz paketa *elm/json*. Funkcija `decodeSession` iz prethodnog primera sa upotrebom funkcija za dekodiranje prikazana je u primeru koda 93. Unutar dekodera mogu se pozivati i drugi dekodieri što se može videti na primeru `decodeUser` i `decodeStudentInfo`. Ovakva obrada *JSON* podataka pruža validaciju podataka pre nego što oni dođu do aplikacije. Ukoliko podaci nemaju očekivanu strukturu definisanu u dekodieru dobija se greška `BadBody`.

```
decodeSession : Decoder Session
decodeSession =
    Decode.map5 Session
        (Decode.field "access_token" Decode.string)
        (Decode.field "expires_in" Decode.float)
        (Decode.field "user" decodeUser)
        (Decode.field "semester_id" Decode.int)
        (Decode.maybe (Decode.field "student_info" decodeStudentInfo))
```

Primer koda 93: Definicija funkcije `silentTokenRefresh`

Supskripcije

Nakon inicijalizacije, obnavljanje tokena se izvršava svaki put kada istekne period važenja tokena za pristup. Budući da se tada komande mogu izvršavati samo unutar funkcije `update`, potrebno je nekako proslediti poruku kojom će se komanda obnavljanja izazvati. To upravo omogućavaju supskripcije.

Supskripcije predstavljaju način da se određeni događaji izvan *Elm* programa prevedu u poruku koja će biti prosleđena funkciji `update`. Kao i komande, definisane su unutar paketa *elm/core* i mogu se koristiti funkcije `Sub.none` i `Sub.batch`. U primeru koda 94 prikazana je supskripcija za obnavljanje tokena. Funkcijom `Time.every` se periodično, na određeni broj milisekundi uzima trenutno vreme i prosleđuje se aplikaciji preko poruke.

```
subscriptions : Model -> Sub Msg
subscriptions { currentUser, accessTokenExpiresIn } =
    let
        refreshTick =
            case currentUser of
                Guest ->
                    Sub.none

                _ ->
                    Time.every (1000 * (accessTokenExpiresIn - 5)) RefreshTick
    in
    Sub.batch [ refreshTick ]
```

Primer koda 94: Supskripcija za obnavljanje tokena

6.2 Moduli stranica i putanja aplikacije

Definicija svih putanja aplikacije nalazi se u modulu *Route* i prikazane su u primeru koda 95. Profesorskih putanja ima više i definisane su posebnim tipom, takođe unutar istog modula. Prilikom inicijalizacije aplikacije, upotrebom funkcije


```
type Route
  = Home
  | Student
  | Login
  | Registration
  | Professor ProfessorSubRoute
  | SetPassword String
  | NotFound
```

Primer koda 95: Putanje aplikacije

`Route.fromUrl` vrši se parsiranje trenutne *url* putanje iz pregledača i mapira se na jednu od definisanih putanja. Sve nevalidne putanje predstavljaju se putanjom `NotFound`. U slučaju putanje `SetPassword` očekuje se univerzalni identifikator za postavljanje lozinke kao deo putanje.

Modul *Route* sadrži funkciju `guard` za sprečavanje neautorizovanog pristupa određenim stranicama i funkciju `redirectTo` za kreiranje komande promene putanje. Funkcija `redirectTo` poziva funkciju `pushUrl` iz modula *Browser.Navigation* koja koristi navigacioni ključ. Nakon izvršavanja komande, nova putanja se prosleđuje funkciji `update` kroz poruku navedenu kao `onUrlChange` prilikom pokretanja aplikacije. Poruka navedena u drugom polju — `onUrlRequest` emituje se prilikom klika na vezu unutar aplikacije.

Osnovna podela aplikacije jeste na stranice koje se koriste za prijavljivanje i registraciju korisnika (moduli *LoginPage*, *RegistrationPage* i *SetPasswordPage*), studentsku stranicu (modul *StudentPage*) i profesorske stranice kojima se koordiniše kroz modul *ProfessorPage*. Pored njih, tu su i stranice koje nisu izdvojene u posebne module — početna i stranica koja se prikazuje u slučaju pogrešne putanje. Definicija tipova stranica prikazanih u primeru koda 96 nalazi se u modulu *Page*, a definicija profesorskih podstranica se nalazi u modulu *ProfessorPage*. Stanje, od-

```
type Page
  = HomePage
  | LoginPage LP.Model
  | RegistrationPage RP.Model
  | SetPasswordPage SPP.Model
  | ProfessorPage
  | StudentPage
  | NotFoundPage
```

Primer koda 96: Stranice aplikacije

nosno model, studentskih i profesorskih stranica čuva se u glavnom modelu pod poljem `mainContent`. Stranice koje se koriste za prijavljivanje i registrovanje kori-

snika uključuju svoje modele u tip i unutar glavnog modela se nalaze pod poljem `currentPage`. Unutar modula *Page* nalazi se funkcija `forRoute` kojim se na osnovu date putanje kreira odgovarajuća stranica.

Svaka stranica koja ima svoj modul ima i definisan svoj model, poruke, kao i funkcije `update` i `view` koje se pozivaju iz glavnog modula aplikacije. Poruke svih stranica, zajedno sa ostalim porukama aplikacije, objedinjuju se unutar modula *Main* u glavni tip poruke (primer koda 97). Unutar funkcije `update` glavnog modula

```
type Msg
= ClickedLink Browser.UrlRequest
| ChangedUrl Url
| GotProfessorMsg ProfessorPage.Msg
| GotStudentMsg StudentPage.Msg
| GotLoginMsg Login.Msg
| GotRegistrationMsg Registration.Msg
| GotPasswordMsg SetPassword.Msg
| GotInitSessionMsg Session.Msg
| GotSessionMsg Session.Msg
| RefreshTick Time.Posix
| Logout
```

Primer koda 97: Objedinjene poruke aplikacije

se na osnovu poruke poziva funkcija `update` odgovarajuće stranice. Trenutni model se ažurira novim modelom date stranice, a poruku rezultujuće komande je potrebno transformisati u odgovarajući glavni tip poruke. Primer ažuriranja studentske stranice dat je u primeru koda 98. Analogno ažuriranju, unutar glavne funkcije `view` se na osnovu trenutne stranice poziva funkcija `view` odgovarajućeg modula.

```
update msg model =
  case ( msg, model.currentPage, model.mainContent ) of
    ...
    ( GotStudentMsg studentMsg, _, StudentModel model_ ) ->
      let
        ( studentModel, cmd ) =
          StudentPage.update studentMsg model_
      in
        ( { model | mainContent = StudentModel studentModel }
        , cmd |> Cmd.map GotStudentMsg
        )
    ...
```

Primer koda 98: Ažuriranje studentske stranice

6.3 Stilizovanje aplikacije

Elm kao jezik specijalne namene sadrži sve što je potrebno za razvoj korisničkog interfejsa veb aplikacija. Zgodno je, ipak, iskoristiti gotove, stilizovane komponente (npr. modale, tabove i slično) kako ne bismo implementirali sve od početka.

Za potrebe implementacije portala MSNR korišćen je paket *NoRedInk/noredink-ui*, koji se oslanja na dosta drugih paketa među kojima je bitno izdvojiti paket *tesk9/accessible-html-with-css*. Ovaj paket predstavlja proširenje paketa *rtfeldman/elm-css* koji se takođe koristi u projektu i omogućava stilizovanje *HTML* elemenata unutar programskog jezika *Elm*, bez upotrebe *css* datoteka. Za iscrtavanje *HTML* elemenata koriste se odgovarajuće funkcije iz modula *Html.Styled* umesto iz modula *Html*.

```
import Accessibility.Styled as Html exposing (Html)
import Browser exposing (Document)
import Css exposing (..)
import Html.Styled.Attributes exposing (css, href)
...
view_ : Model -> Html Msg
view_ model =
    ...
    Html.div
        [ css [ height (vh 100), displayFlex, flexDirection column ] ]
        [ {-sadrzaj-}
        ]

view : Model -> Document Msg
view model =
    { title = "MSNR"
    , body = [ view_ model |> Html.toUnstyled ]
    }
```

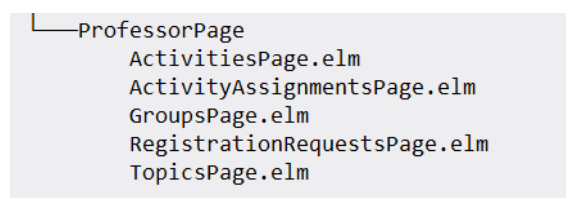
Primer koda 99: Primer stilizovanja elementa i upotreba funkcije `toUnstyled`

Modul *Accessibility.Styled* iz paketa *tesk9/accessible-html-with-css* pored stilizovanja pruža i podršku za pristupačnost [20] (eng. *accessibility* — skraćeno *a11y*). Funkcija `toUnstyled` omogućava korišćenje *Accessibility.Styled.Html* unutar *Elm* aplikacije. Primer njene upotrebe unutar modula *Main* prikazan je u primeru koda 99.

6.4 Struktura profesorske stranice

Profesorske stranice nalaze se pod direktorijumom *ProfessorPage* (slika 6.3). Svaka stranica predstavljena je zasebnim modulom i to su:

- *RegistrationRequestsPage* — za obradu studentskih zahteva za registraciju,
- *ActivitiesPage* — za dodavanje novih aktivnosti,
- *GroupsPage* — za prikazivanje svih grupa,
- *TopicsPage* — za dodavanje tema studentskih radova,
- *ActivityAssignmentPage* — za pregled i ocenjivanje studentskih aktivnosti.



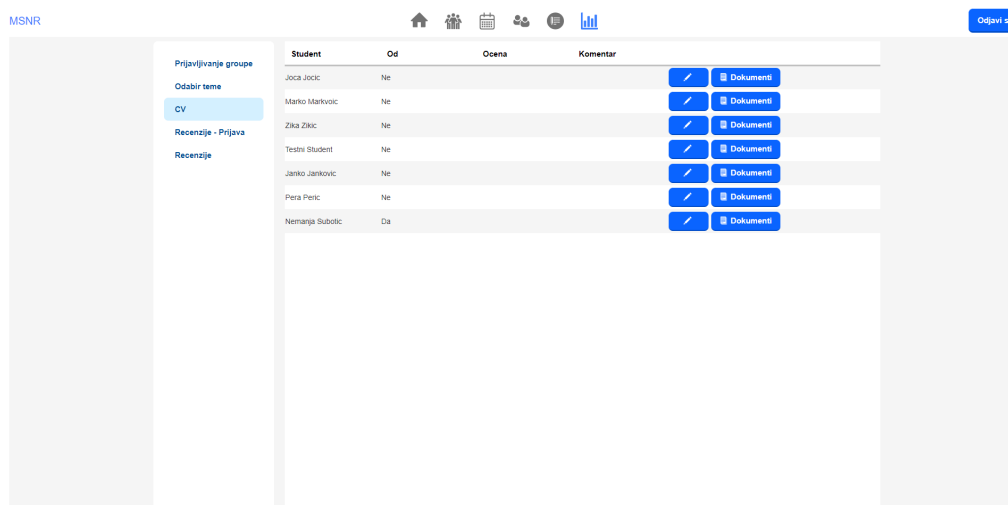
Slika 6.3: Sadržaj direktorijuma *ProfessorPage*

Prikazivanje odgovarajućih stranica u odnosu na trenutnu putanju obrađuje se unutar modula *ProfessorPage*, slično obradi svih stranica unutar modula *Main*. Svaka stranica ima svoj model, kao i funkcije `view` i `update`. Model definisan u modulu *ProfessorPage* sadrži sve modele stranica kao i zajedničke podatke o studentima, grupama i aktivnostima. U primeru koda 100 prikazan je alijas sveobuhvatnog modela, gde se pored modela stranica mogu videti i ostali podaci potrebni za njihovo prikazivanje. Modeli stranica čuvaju podatke koji se koriste isključivo unutar odgovarajuće stranice. Prilikom prikazivanja profesorskih stranica u modulu *ProfessorPage* poziva se funkcija `view` odgovarajuće stranice, kojoj se prosleđuje adekvatan model, zajedno sa drugim potrebnim podacima.

Kretanje kroz stranice vrši se pomoću navigacione trake čiji prikaz se može videti na slici 6.4 zajedno sa izgledom stranice za ocenjivanje aktivnosti.

```
type alias Model =
{ zone : Zone
, accessToken : Token
, apiBaseUrl : String
, currentSemesterId : Int
, requestesModel : Requests.Model
, activitiesModel : Activities.Model
, activityAssignmentsModel : ActivityAssignments.Model
, topicsModel : TopicsPage.Model
, activities : Dict Int Activity
, activityTypes : Dict Int ActivityType
, groups : Dict Int Group
, students : Dict Int Student
, assignments : List ShallowAssignment
, loadingActivities : Bool
, loadingActivityTypes : Bool
, loadingGroups : Bool
, loadingStudents : Bool
, loadingAssignment : Bool
, hasLoadingError : Bool
}
```

Primer koda 100: Glavni model za prikazivanje profesorskih stranica



Slika 6.4: Izgled stranice za ocenjivanje aktivnosti

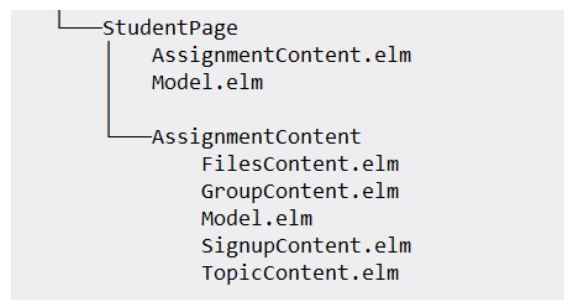
6.5 Struktura studentske stranice

Za razliku od profesorskog dela aplikacije koji je podeljen na više stranica, studentski deo se sastoji iz samo jedne stranice u kojoj su prikazane sve dodeljene aktivnosti studenta.

Svaka dodeljena aktivnost ima naziv i opis aktivnosti, rok za izvršenje i broj poena koji nosi. Način prikazivanja pojedinačnih aktivnosti razlikuje se po jedin-

stvenom kodu i sadržaju tipa aktivnosti, kao i po tome da li je u pitanju prijava za aktivnost. Tako da se prikazivanje same aktivnosti može podeliti u dva dela, jedan se odnosi na osnovne podatke, a drugi na sam sadržaj aktivnosti potreban za prijavljivanje grupe, tema, prilaganja dokumenata ili prijavljivanje za određenu aktivnost.

Prikazivanje studentske stranice obrađuje se unutar modula *StudentPage*, a pomoćni moduli koji se tom prilikom koriste nalaze se pod direktorijumom *StudentPage* čija je struktura prikazana na slici 6.5.



Slika 6.5: Sadržaj direktorijuma *StudentPage*

Modeli studentske stranice i sadržaja aktivnosti izdvojeni su u posebne module — *StudentPage.Model* i *AssignmentContent.Model* da bi se izbegla ciklična zavisnost, jer se osnovni model koristi unutar modula *AssignmentContent*. U primeru koda 101 predstavljen je osnovni model studentske stranice u kom se nalaze svi podaci potrebni za njen prikaz. Za razliku od modela profesorskih stranica, modeli sadržaja aktivnosti čuvaju se u nizu (*assignmentsModels*). Svakom elementu niza modela odgovara dodeljena aktivnost (niz *assignments*) na istom indeksu.

Osnovni modul za prikazivanje studentske stranice, pored modula *StudentPage*, jeste modul *AssignmentContent* koji ima sličnu ulogu kao modul *ProfessorPage* kod profesorskih stranica. U direktorijumu *AssignmentContent* nalaze se moduli za obradu sadržaja — *GroupContent*, *FilesContent*, *SignupContent* i *TopicContent*, i kao kod profesorskih stranica svaki od njih ima svoj model i funkcije *update* i *view*.

Prilikom pozivanja funkcije *update* u modulu *StudentPage* unutar poruke (*Msg*), pored informacije o samoj poruci, nalazi se i indeks na osnovu kog se menja odgovarajući model u nizu. Definisanje poruke sa indeksom i njena obrada prikazana je u primeru koda 102. Funkcija *AssignmentContent* iz modula *AssignmentContent* na osnovu tipa poruke poziva funkciju ažuriranja iz odgovarajućeg modela za obradu sadržaja.

```
type alias Model =
{ accessToken : Token
, apiUrl : String
, email : String
, firstName : String
, lastName : String
, studentId : Int
, semesterId : Int
, groupId : Maybe Int
, loadingGroup : Bool
, group : Maybe Group
, indexNumber : String
, zone : Zone
, currentTimeSec : Int
, loading : Bool
, assignments : Array Assignment
, assignmentsModels : Array ACM.Model
, assignmentsState : AssignmentsState
, loadingStudents : Bool
, students : List Student
, loadingTopics : Bool
, topics : List Topic
}
```

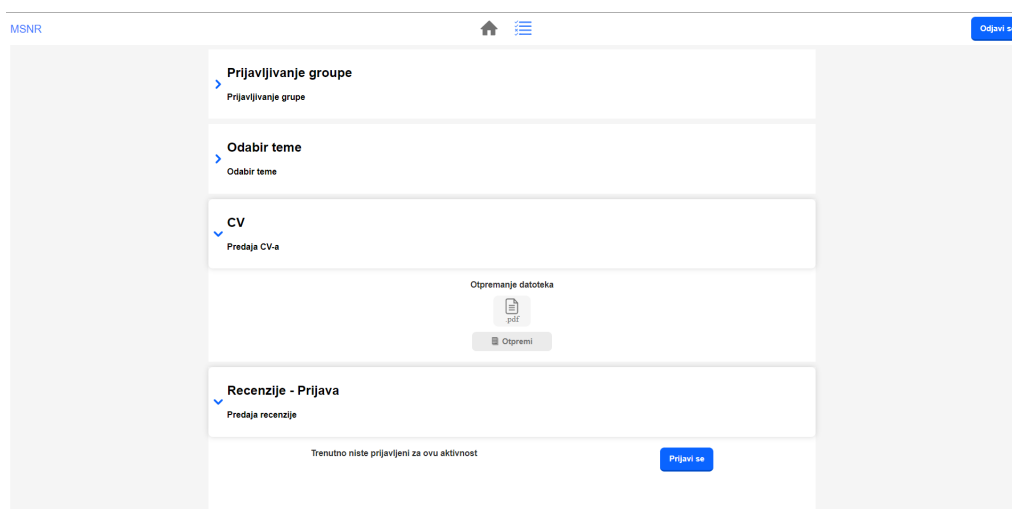
Primer koda 101: Osnovni model za prikazivanje studentske stranice

```
type Msg
= ...
| GotAssignmentContentMsg Int AssignmentContent.Msg

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
...
GotAssignmentContentMsg index assignmentMsg ->
let
  ( model_, cmd ) =
    AssignmentContent.update assignmentMsg index model
in
( model_, Cmd.map (GotAssignmentContentMsg index) cmd )
```

Primer koda 102: Ažuriranje sadržaja aktivnosti

Za prikazivanje dodeljenih aktivnosti koristi se komponenta harmonika (eng. *accordion*) iz paketa *NoRedInk/noredink-ui*, tako da se inicijalno prikazuju samo zaglavlja aktivnosti sa osnovnim informacijama. Klikom na zaglavlje prikazuje se i sam sadržaj aktivnosti. Izgled studentske stranice prikazan je na slici 6.6. Prikazivanje zaglavlja aktivnosti definiše se unutar modula *StudentPage*, a za prikazivanje sadržaja, analogno ažuriranju, poziva se funkcija *view* iz modula *AssignmentContent*.



Slika 6.6: Izgled studentske stranice

Glava 7

Zaključak

U ovom radu predstavljen je razvoj studentskog portala upotrebom funkcionalnih programskih jezika *Elm* i *Elixir*. Sam portal predstavlja prototip na kom su prikazane osnovne osobine i koncepti ovih programskih jezika. U okviru rada data su objašnjenja osnovnih funkcionalnosti i komponenti portala, kao i objašnjenje implementacije najvažnijih delova.

Razvojni okvir *Phoenix* daje jasne smernice kako struktura projekta treba da izgleda. Biblioteke unutar ekosistema *Elixir* nastale su na osnovu dobrih praksi postojećih biblioteka drugih programskih jezika što njihovu adaptaciju čini lakom. Stoga, rad sa razvojnim okvirom *Phoenix* deluje poznato, a posebnu udobnost u radu, u odnosu na druga razvojna okruženja, pružaju izraženo poklapanje obrazaca i njegova funkcionalna priroda. Iako se za osnovnu prednost *Phoenix*-a uzima izvršavanje na virtualnoj mašini *BEAM*, autor definitivno preporučuje njegovu upotrebu za razvoj svih tipova veb aplikacija.

Sa druge strane, rad sa programskim jezikom *Elm*, sa njemu svojstvenom arhitekturom *Elm* kao centralnim delom, predstavlja potpuno novo iskustvo. Nakon početnog prilagođavanja na sam jezik i način izvršavanja programa, rad sa programskim jezikom *Elm* postaje vrlo ugodan i daje poseban osećaj sigurnosti tokom rada u odnosu na programski jezik *JavaScript* (i razvojnim okvirima zasnovanim na njemu). Kompilator proverom tipova garantuje ispravnost ulaznih podataka i forsira proveru svih slučajeva upotrebom tipova *Maybe* i *Result*, što značajno smanjuje broj grešaka u aplikaciji. Prilikom razvoja portala, korisnički interfejs je prošao kroz nekoliko iteracija i značajnih izmena tokom kojih se sam autor uverio u lakoću refaktorisanja koju *Elm* zajedno sa kompilatorom pruža programerima. U toku rada nije predstavljena integracija sa *JavaScript*-om, koja je moguća, ali podrazumeva

uvođenje dosta šablonskog koda. Stoga, *Elm* ne predstavlja pravi izbor za razvoj aplikacija koje se zasnivaju na velikom broju biblioteka napisanih u programskom jeziku *JavaScript*.

Pre upotrebe portala trebalo bi ubaciti podršku za telemetriju na serverskoj strani u cilju nadgledanja rada aplikacije. Sadržaj tipa aktivnosti trenutno podržava samo rad sa dokumentima, dodavanjem podrške za kreiranje prilagođenih formi zaokružile bi se sve aktivnosti unutar portala i pružila mogućnost sprovođenja anketa i sličnih aktivnosti. Kao značajnije unapređenje portala trebalo bi razmotriti upotrebu rešenja za čuvanje dokumenata na oblaku. Na klijentskoj strani potrebno je prilagoditi upotrebu korisničkog interfejsa za manje uređaje kao što su mobilni telefoni i tableti.

Bibliografija

- [1] *Adresa sa implementacijom portala MSNR*. URL: <https://github.com/NemanjaSubotic/master-rad/tree/master/portal>.
- [2] *Biblioteka Redux*. URL: <https://redux.js.org/introduction/prior-art>.
- [3] *Biblioteka Vuex*. URL: <https://vuex.vuejs.org>.
- [4] Evan Czaplicki. *Elm: Concurrent FRP for Functional GUIs*. 2012. URL: <https://elm-lang.org/assets/papers/concurrent-frp.pdf>.
- [5] Evan Czaplicki. *Github comment*. 2015. URL: <https://github.com/elm/elm-lang.org/issues/408#issuecomment-151656681>.
- [6] *Elm - Veoma brz HTML*. URL: <https://elm-lang.org/news/blazing-fast-html-round-two>.
- [7] Learn You a Haskell - Types **and** Typeclasses. *Miran Lipovača*. URL: <http://learnyouahaskell.com/types-and-typeclasses>.
- [8] *Heroku Cloud Platform*. URL: <https://www.heroku.com>.
- [9] Daniel Higginbotham. *Clojure for the Brave and True*. No Starch Press, 2015. ISBN: 978-1-59327-591-4.
- [10] *Install Elm*. URL: <https://guide.elm-lang.org/install/elm.html>.
- [11] Saša Jurić. *Elixir in Action*. 2 **edition**. Manning Publications, 2019. ISBN: 1617295027, 978-1617295027.
- [12] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. ISBN: 1-59327-283-9, 1-59327-283-9, 978-1-59327-283-8.
- [13] Bill Lubanovic. *Introducing Python*. 2 **edition**. O'Reilly Media, 2019. ISBN: 9781492051329, 1492051322, 9781492051343, 1492051349.
- [14] *ML programski jezik*. URL: <https://courses.cs.washington.edu/courses/cse341/04wi/lectures/02-ml-intro.html>.

- [15] *MVU obrazac u .NET 5*. URL: <https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui>.
- [16] *Npm Elm*. URL: <https://www.npmjs.com/package/elm>.
- [17] *Obejktni model dokumenta*. URL: https://en.wikipedia.org/wiki/Document_Object_Model.
- [18] Addy Osmani. *Learning JavaScript Design Patterns*. 1 edition. O'Reilly Media, 2012. ISBN: 1449331815,9781449331818.
- [19] *Primer Elm programa*. URL: <https://elm-lang.org/examples/buttons>.
- [20] *Pristupačnost u veb programiranju*. URL: <https://developer.mozilla.org/en-US/docs/Web/Accessibility>.
- [21] *REST architectural style*. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.
- [22] *Svojstvo withCredentials u HTTP zahtevu*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/withCredentials>.
- [23] *Try Elm*. URL: <https://elm-lang.org/try>.
- [24] *Zvanična dokumentacija biblioteke Ecto*. URL: <https://hexdocs.pm/ecto/Ecto.html>.
- [25] *Zvanična dokumentacija biblioteke LINQ*. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/basic-linq-query-operations>.
- [26] *Zvanična dokumentacija biblioteke Plug*. URL: <https://hexdocs.pm/plug/readme.html>.
- [27] *Zvanična stranica alata Hex*. URL: <https://hex.pm/>.
- [28] *Zvanična stranica aplikacije Wechat*. URL: <https://www.wechat.com>.
- [29] *Zvanična stranica aplikacije Whatsapp*. URL: <https://www.whatsapp.com>.
- [30] *Zvanična stranica baze podataka PostgreSQL*. URL: <https://www.postgresql.org>.
- [31] *Zvanična stranica baze podataka Riak*. URL: <https://riak.com>.
- [32] *Zvanična stranica programskog jezika Elixir*. URL: <https://elixir-lang.org/>.
- [33] *Zvanična stranica programskog jezika Erlang*. URL: <https://www.erlang.org>.

BIBLIOGRAFIJA

- [34] *Zvanična stranica razvojnog okruženja Phoenix*. URL: <https://www.phoenixframework.org>.

Biografija autora

Nemanja Subotić rođen je 04.09.1993. godine u Užicu, odrastao je u Novoj Varoši, gde je završio prirodno-matematički smer gimnazije „Pivo Karamatijević”. Smer *Informatika* na Matematičkom fakultetu Univerziteta u Beogradu upisuje 2012. godine u septembru i završava u julu 2015. sa prosečnom ocenom 9,38. Iste godine upisuje i master studije.

Od marta 2016. godine radi kao softverski inženjer na različitim projektima u nekoliko domaćih i stranih firmi. Veći deo karijere radio je na razvoju veb aplikacija odakle je i potekla ideja za temu rada.