UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ana Petrović

TESTIRANJE FUNKCIONALNIH PROGRAMA NA PRIMERU APLIKACIJE KOJA KORISTI JEZIKE ELM I ELIXIR

master rad

Beograd, 2023.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

dr Ivan Čukić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Testiranje funkcionalnih programa na primeru aplikacije koja koristi jezike Elm i Elixir

Rezime: Cilj rada je prikaz ključnih pristupa prilikom testiranja funkcionalnog koda, sa ograničenjem na funkcionalne programske jezike *Elm* i *Elixir*. *Elm* je statički tipiziran, čist funkcionalni jezik koji već ima odličnu podršku za obradu grešaka, što čini testiranje veoma jednostavnim. U okviru rada, napisani su testovi u jeziku *Elm* za klijentsku stranu veb portala koji služi za upravljanje aktivnostima na kursu *Metodologija stručnog i naučnog rada*. Za serveski deo ove aplikacije, korišćen je funkcionalni programski jezik *Elixir*, zajedno sa svojim razvojnim okvirom za veb aplikacije *Phoenix*. Upotrebom ugrađenog razvojnog okvira za testiranje u *Elixir-u*, napisani su testovi koji pokrivaju serversku stranu portala. Implementacija samog portala nije deo ovog rada.

Ključne reči: funkcionalno programiranje, testiranje funkcionalnog koda, verifikacija softvera, programski jezik Elixir, programski jezik Elm

Sadržaj

Glava 1

Uvod

U savremenom razvoju softvera, funkcionalno programiranje sve više dobija na popularnosti među programskim paradigmama. Jezici *Elm* i *Elixir* su primeri programskih jezika koji se oslanjaju na principe funkcionalnog programiranja, i nude moćne alate koji obezbeđuju pouzdan i robustan softver.

Kako bi softver bio visoko kvalitetan i pouzdan, nezaobilazan deo razvoja predstavlja njegovo testiranje. Osim što ispituje da li se kôd ponaša na očekivan način, pisanje testova predstavlja i vid dokumentacije. Jedan od glavnih koncepta funkcionalne paradigme podrazumeva imutabilnost podataka i pisanje čistog koda, odnosno čistih funkcija — funkcija koja nemaju propratne efekte. Čiste funkcije olakšavaju proces testiranja, jer eliminišu potrebu za praćenjem promena stanja programa.

Programski jezik *Elixir* je funkcionalni jezik opšte namene koji se izvršava na virtuelnoj mašini *BEAM*, napravljenoj za programski jezik *Erlang*. Zbog toga je pogodan za skalabilne i distribuirane sisteme. U ovom radu korišćen je za testiranje serverskog dela prethodno implementirane veb aplikacije, napisanog pomoću razvojnog okvira *Phoenix*, koji je nastao i razvija se uporedo sa *Elixir-om*. *Elixir* kôd se izvršava konkurentno u obliku veoma lakih izolovanih procesa, koji obezbeđuju visoku otpornost na greške i time značajno olakšavaju testiranje. Pored toga, *Elixir* odlikuju i osobine kao što su poklapanje obrazaca i nepromenljivost podataka. *ExUnit*, razvojni okvir za testiranje *Elixir* programa, razvijan je zajedno sa ovim programskim jezikom od samog početka. Poseduje alate koji omogućavaju testiranje svih slojeva i aspekata jedne *Elixir* aplikacije.

Programski jezik *Elm* predstavlja čist funkcionalan programski jezik namenjen isključivo za klijentsko veb programiranje. Veoma je jednostavan za upotrebu, i jedna od njegovih najistaknutijih karakteristika je odsustvo grešaka za vreme izvršavanja.

Elm poseduje svoj kompilator i strog sistem tipova, koji obezbeđuju da se veliki broj grešaka pronađe već za vreme prevođenja. Testiranje u *Elm-u* najčešće podrazumeva testiranje zasnovano na svojstvima (eng. *property based testing*), kojim se proverava ispunjenost određenih uslova koje izlaz treba da ispuni za veliki broj različitih ulaza. Arhitektura *Elm* promoviše izolaciju koda, što olakšava testiranje pojedinačnih komponenti. Ugrađeni razvojni okvir za testiranje, *elm-test*, koristi se za pisanje testova jedinica koda i podstiče pisanje koda koji je testabilan.

Cilj ovog rada je da predstavi osnovne koncepte i pravila testiranja u jezicima *Elm* i *Elixir*, pisanjem testova za serverski i klijentski deo veb aplikacije *Portal MSNR*. Ovaj portal predstavlja mesto na kome studenti i profesori sa kursa *Metodologija stručnog i naučnog rada* mogu izvršavati različite aktivnosti u okviru kursa tokom semestra.

U poglavlju 2 dat je rezime osnovnih karakteristika funkcionalnih programskih jezika, a nakon toga opisan je značaj testiranja softvera, i osobine programa napisanih u jezicima *Elm* i *Elixir* koje doprinose testiranju njihovih programa. Poglavlje 3 predstavlja opis aplikacije koja je testirana — prikazane su funkcionalnosti, osnovni entiteti i arhitektura *Portala MSNR*. U poglavlju 4 dati su testovi za serverski deo aplikacije, napisani u programskom jeziku *Elixir*, a poglavlje 5 prikazuje testove napisane u programskom jeziku *Elm*, koji proveravaju ispravnost klijentskog dela aplikacije. Zaključak sa predlozima daljih unapređenja dat je u poglavlju 6.

Glava 2

Funkcionalna paradigma

Funkcionalno programiranje je specifičan pristup programiranju, tj. programska paradigma, koja se zasniva na pojmu matematičke funkcije. Programi se kreiraju pomoću izraza i funkcija, bez izmena stanja i podataka [2]. Iz tog razloga, jednostavniji su za razumevanje i otporniji na greške u odnosu na imperativne programe. Programski stil je deklarativnog tipa i umesto naredbi koriste se izrazi, tako da se izvršavanje programa svodi na evaluaciju izraza. Vrednost izraza je nezavisna od konteksta u kojem se izraz nalazi, što se naziva *transparentnost referenci* i predstavlja osnovnu osobinu *čistih* funkcionalnih jezika (eng. *pure functional programming language*). Transparentnost referenci kao osnovnu posledicu ima nepostojanje propratnih efekata. Sa druge strane, *nečisti* funkcionalni jezici (eng. *impure functional programming language*) dozvoljavaju propratne efekte, koji mogu izazvati suptilne greške i biti teži za razumevanje. Međutim, praktičniji su za specifične vrste zadataka, kao što je programiranje korisničkog interfejsa ili rad sa bazom podataka.

2.1 Karakteristike funkcionalnih jezika

U nastavku su objašnjene neke od najvažnijih osobina jezika funkcionalne paradigme.

Funkcije kao građani prvog reda

U funkcionalnim programima, funkcije se smatraju građanima prvog reda (eng. *first class citizen*). Građani prvog reda su entiteti u okviru programskog jezika koji mogu biti:

- deo nekog izraza
- dodeljeni nekoj promenljivoj
- prosleđeni kao argument funkcije
- povratne vrednosti funkcije

Mogućnost prosleđivanja funkcija kao argumenata drugih funkcija je ključna za funkcionalnu paradigmu.

Čista funkcija

Čista funkcija (eng. *pure function*) ima dve osnovne karakteristike:

- Transparentnost referenci
- Imutabilnost

Koncept transparentnosti referenci se odnosi na to da je vrednost izraza jedinstveno određena. Izraz se može zameniti svojom vrednošću na bilo kom mestu u programu, bez promene u ponašanju programa. Slično može važiti i za funkcije, ako se pri pozivu funkcije sa istim vrednostima argumenata uvek proizvodi isti rezultat. Ponašanje takve funkcije je određeno njenim ulaznim vrednostima.

Imutabilnost podrazumeva odsustvo propratnih efekata, tj. da čista funkcija ne generiše nikakve posledice u smislu izmena argumenata, promenljivih, ili stanja programa. To podrazumeva i prikazivanje izlaza na ekranu, ili trajno čuvanje podataka u bazi podataka ili datoteci. Jedini rezultat čiste funkcije jeste vrednost koju ona vrati. Kao posledica ovoga, funkcionalni programi su laki za debagovanje. Čiste funkcije takođe olakšavaju paralelizaciju i konkurentnost aplikacija. Na osnovu ovako napisanih programa, kompilator lako može da paralelizuje naredbe, sačeka da evaluira rezultate kada budu potrebni, i na kraju da zapamti rezultat, s obzirom na to da se on neće promeniti sve dok ulaz ostaje isti. Kôd 2.1 prikazuje primer jedne čiste funkcije u programskom jeziku *Elixir*.

```
defmodule Math do
  def fibonacci(0) do 0 end
  def fibonacci(1) do 1 end
  def fibonacci(n) do fibonacci(n-1) + fibonacci(n-2) end
end

IO.puts Math.fibonacci(9)
```

Listing 2.1: Primer čiste funkcije

Funkcije višeg reda

Funkcija višeg reda je funkcija koja kao argument uzima jednu ili više funkcija i/ili ima funkciju kao svoju povratnu vrednost. U funkcionalnom programiranju se intenzivno koriste ovakve funkcije, a po svojoj važnosti se posebno izdvajaju *map*, *filter* i *reduce (fold)*. Funkcija *map* kao argumente prima funkciju i listu, i zatim primeni datu funkciju na svaki element liste i kao povratnu vrednost proizvodi novu listu. Upotrebom funkcije *filter* mogu se eliminisati neželjeni elementi neke liste — na osnovu prosleđene funkcije predikata i date liste, *filter* vraća listu sa elementima koji ispunjavaju dati kriterijum. Funkcije *map* i *filter* se mogu implementirati i upotrebom funkcije *fold*. *Fold* prihvata tri argumenta: funkciju spajanja, početnu vrednost u kojoj će se akumulirati rezultat, i listu. Funkcija spajanja je binarna operacija koja se primenjuje redom na svaki element liste i trenutnu akumuliranu vrednost, sve dok se lista ne redukuje na jednu vrednost koja predstavlja krajnji rezultat.

Funkcije višeg reda dovode do sažetog i čistog koda. Takođe, pogodne su i za paralelizaciju. Primer koda 2.2 pokazuje upotrebu spomenutih funkcija u programskom jeziku *Elm*.

```
[1, 2, 3] |> List.map (\number -> number * 2)
[1, 2, 3, 4, 5] |> List.filter (\number -> number <= 3)
[1, 2, 3, 4, 5] |> List.foldl (\item total -> total + item) 0
```

Listing 2.2: Funkcije višeg reda

Odsustvo promenljivih i rekurzija

Čisti funkcionalni jezici nemaju stanje koje bi se menjalo tokom izvršavanja programa, pa zbog toga ne podržavaju koncept promenljivih. Sa druge strane, nečisti funkcionalni jezici podržavaju i karakteristike drugih programskih paradigmi, te je u okviru njih dozvoljena upotreba promenljivih. Iako su fleksibilniji po tom pitanju, nečisti funkcionalni jezici promovišu imutabilnost kao dobru praksu.

Kod funkcionalno napisanih programa može se primetiti odsustvo petlji. Kad god je moguće, funkcije se definišu kompozicijom drugih funkcija, a mogu se definisati i rekurzivno, i tako postići ponavljanje izvršavanja.

2.2 Testiranje u razvoju softvera

Testiranje koda je jedan od najvažnijih aspekata u procesu razvoja softvera. Cilj testiranja je pronalaženje grešaka, proverom da li su ispunjeni svi funkcionalni i nefunkcionalni zahtevi [35]. Softver koji ne radi onako kako je predviđeno može dovesti do različitih problema, kao što su gubitak novca i vremena, ili u najgorim slučajevima — povrede ili smrti. Testiranjem se proverava kvalitet softvera i smanjuje rizik od neželjenog ponašanja. Glavna uloga testiranja jeste verifikacija softvera — provera da li sistem zadovoljava specifikaciju, ali uključuje i validaciju — proveru da li sistem ispunjava sve potrebe korisnika.

Organizacija testova

Model piramide testiranja (prikazan na slici 2.2) je koncept koji pomaže u razmišljanju o tome kako testirati softver [29]. Uloga piramide je da vizuelno predstavi logičku organizaciju standarda u testiranju. Sastoji se od tri sloja: bazu piramide predstavljaju testovi jedinica koda (eng. *unit tests*). Njih bi trebalo da bude najviše — kako su najmanji, samim tim su i najbrži, a izvršavaju se u potpunoj izolaciji. Na sledećem nivou, u sredini piramide, nalaze se integracioni testovi (eng. *integration tests*). Integracija podrazumeva način na koji različite komponente sistema rade zajedno. Nisu potrebne interakcije sa korisničkim interfejsom, s obzirom na to da ovi testovi pozivaju kôd preko aplikativnog interfejsa. Vrh piramide čine sistemski testovi (eng. *system tests*). Oni se ne fokusiraju na individualne komponente, već testiraju čitav sistem kao celinu i time utvrđuju da on radi očekivano i ispunja-

va sve funkcionalne i nefunkcionalne zahteve. Takvi testovi su prilično skupi, pa je potrebno doneti odluku koliko, i koje od njih se isplati sprovesti.

Jedna vrsta sistemskih testova su takozvani E2E testovi[1], koji simuliraju korisničko iskustvo kako bi osigurali da sistem funkcioniše kako treba, od korisničkog interfejsa, pa sve do serverske strane i baze podataka. Testovi korisničkog interfejsa (eng. *User Interface tests, UI tests*) se staraju o tome da se korisnički interfejs ponaša na očekivan način. Obično su automatski i simuliraju interakcije pravih korisnika sa aplikacijom, kao što su pritiskanje dugmića, unos teksta i slično. S obzirom na to da oni proveraju da sistem, zajedno sa svojim korisničkim interfejsom, radi kako treba — mogu se u određenom kontekstu smatrati sistemskim testovima, ali su ipak specifičniji i mogu se sprovoditi i nezavisno od celokupnog sistema.

U opštem slučaju, testiranje projekta koji se sastoji od više slojeva podrazumeva kombinaciju testova jedinica koda, integracionih i sistemskih testova kako bi se osigurala ukupna funcionalnost, pouzdanost i perfomanse sistema.

Slika 2.1: Model piramide testiranja

Anatomija testa

Kako bi kôd testa bio čitljiv i jednostavan za razumevanje, obrazac četvorofaznog testa (eng. *four-phase test*) predlaže strukturu testa koja podrazumeva ne više od četiri faze [33]. Svaki test se može podeliti na četiri jasno odvojive celine:

[1]E2E je skraćenica za testove sa kraja na kraj (eng. *end-to-end*)

1. Priprema (eng. *setup*) — sređivanje podataka koji će se prosleđivati pred samu proveru (uglavnom nije neophodno u testiranju čisto funkcionalnih programa).
2. Delovanje (eng. *exercise*) — pozivanje koda koji se testira, ključni deo svakog testa.
3. Verifikacija (eng. *verify*) — testovi proveravaju ponašanje koda (često se spaja sa prethodnom fazom).
4. Rušenje (eng. *teardown*) — vraćanje podataka na prvobitno stanje, npr. ako se u prvoj fazi koriste neka deljena stanja, kao što je baza podataka. Ovaj korak se često izvršava implicitno.

Konkretan primer ovako organizovanog testa u programskom jeziku *Elm* dat je u primeru 2.3. Definisan je jednostavan test jedinice koda, koji proverava da li funkcija koja sabira dva broja daje ispravan rezultat. U fazi pripreme brojevima i njihovoj očekivanoj sumi se dodeljuju vrednosti. U fazi delovanja poziva se funkcija *sum*, a pozivanjem funkcije *expect* proverava se da li je rezultat jednak očekivanom u fazi verifikacije. Faza rušenja u ovom slučaju ne treba da uradi ništa. Tip `()` se naziva jedinični tip (eng. *unit type*). Predstavlja vrednost koja može biti samo jedna, ali nije važno koja je to vrednost. Navođenje _ sugeriše da postoji neka vrednost na tom mestu, ali se ona ne koristi i zbog toga se može ignorisati. Operatori prosleđivanja `|>` i `<|` (eng. *pipe*) podrazumevaju primenu funkcije, i koriste se kako bi se izbegla upotreba zagrada. Deo koda koji koristi uzastopno ove operatore često se naziva cevovod (eng. *pipeline*).

```
sumTest =
    describe "sum" [ test "should add two numbers" <| \() ->
        let
        --Setup
            x = 2
            y = 3
            expected = 5
        in
        -- Exercise (sum), Verify (expect)
            expect (sum x y) |> toEqual expected ]
    -- Teardown
    teardown _ = ()
```

Listing 2.3: Četiri faze testa koji proverava ispravnost funkcije sabiranja dva broja

Testovi jedinica koda

Jedinica je mala logička celina koda: može biti funkcija, klasa, metod klase, modul i slično. Test jedinice koda proverava samo da li se data jedinica ponaša prema svojoj specifikaciji. Ovi testovi se mogu pisati u potpunoj izolaciji, i ne zavise ni od jedne druge komponente, servisa, ni korisničkog interfejsa. Dakle, izdvajaju se najmanji testabilni delovi aplikacije i proverava se da li rade ono za šta su namenjeni. Ovi testovi su po pravilu najbrži i najjednostavniji za pisanje jer se bave malim delom aplikacije, te je kôd koji se testira najčešće vrlo jednostavan.

Cilj testova jedinica koda jeste da spreče greške koje mogu nastati izmenama koda, kao i da omoguće da se lako utvrdi lokacija dela koda koji izaziva grešku. Pri dizajniranju ovih testova, potrebno je proveriti da kôd radi tačno ono za šta je namenjen, a da se to uradi pisanjem najmanje moguće dodatne količine koda.

Može se diskutovati o tome šta se smatra „jedinicom", a posebno u kontekstu funkcionalnog programiranja. Uobičajeno je da se testovi jedinica koda fokusiraju na pojedinačnu funkciju i njenu logiku, kako bi se opseg testa održavao što užim, radi bržeg pronalaženja grešaka. Međutim, nekada ima smisla da se u opseg testa uključi više modula ili procesa, i time se proširi definicija jedinice koda i olakša održavanje samih testova.

Integracioni testovi

Jedan od ključnih koraka u razvoju softvera jeste pisanje integracionih testova. Oni utvrđuju da li različite komponente sistema rade zajedno na predviđen način. Pojedinačni moduli i komponente se kombinuju i testiraju kao jedna celina. Cilj integracionog testiranja jeste identifikacija i rešavanje problema koji mogu nastati nakon što se komponente softverskog sistema integrišu i krenu da međusobno komuniciraju. Svaka od njih pojedinačno možda radi kako treba, ali nakon što se to utvrdi testovima jedinica koda, potrebno je proveriti da li će njihova interakcija izazvati neželjeno ponašanje.

U zavisnosti od potreba konkretnog sistema, postoje različiti pristupi integracionom testiranju. Ako su komponente viših nivoa kritične za funkcionalnost sistema, ili od njih zavisi mnogo drugih komponenti, ima smisla prvo testirati njih, pa kasnije postepeno preći na komponente nižih nivoa. Ovakav pristup se naziva testiranje odozgo nadole (eng. *top-down integration testing*). U suprotnom, ako su komponente nižih slojeva arhitekture kritičnije za celokupan sistem, predlaže se testiranje odozdo

nagore (eng. *bottom-up integration testing*). Hibridno integraciono testiranje (eng. *hybrid integration testing*) podrazumeva kombinaciju prethodna dva — započinje sa testovima komponenti najvišeg sloja, zatim se prelazi na testiranje najnižeg sloja, sve dok se postepeno ne stigne do središnjih. Kada je sistem relativno jednostavan i ne postoji veliki broj komponenti, može se primeniti pristup po principu "velikog praska" (eng. *big-bang integration testing*), koji podrazumeva testiranje svih komponenti odjednom, kao jedne celine.

Ako se komponente nalaze u okviru istog sistema, gde postoji kontrola i neko očekivano ponašanje — integracioni testovi su prilično jednostavni. Međutim, kada su u pitanju spoljašnje komponente i testiranje interakcije sistema sa njima, pisanje integracionih testova postaje malo komplikovanije. Mnoge aplikacije koriste baze podataka, druge servise ili API-je, sa kojima se testovi moraju uskladiti. U testovima se mogu koristiti pravi podaci, ili se umesto njih ubaciti takozvani dubleri (eng. *test doubles*).

Integraciono testiranje je neophodno da bi se obezbedio kvalitetan i pouzdan softver, i zahvaljujući njemu rano se uočavaju različiti problemi do kojih može doći i time značajno redukuje vreme i cena celokupnog razvoja.

Sistemski testovi

Nakon završenog testiranja jedinica koda i integracionog testiranja, neophodno je sprovesti sistemske testove. Ova vrsta testiranja se vrši nad kompletno integrisanim sistemom, i podrazumeva proveru da li celokupni sistem ispunjava zahteve, odnosno da li je spreman za isporuku krajnjim korisnicima. Sistemski testovi se sprovode u okruženju koje je konfigurisano tako da bude što sličnije onom kakvo će biti u produkciji. Praksa je da ih pišu testeri koji nisu učestvovali u razvoju, kako bi se izbegla pristrasnost. Pored funkcionalnih i nefunkcionalnih specifikacija koje se tiču ponašanja sistema, testiraju i očekivanja korisnika. Mogu biti manuelni ili automatski.

Sistemsko testiranje se smatra testiranjem crne kutije (eng. *black-box testing*). Ponašanje sistema se evaluira iz ugla korisnika, što znači da ne zahteva nikakvo znanje o unutrašnjem dizajnu i strukturi koda. Ono što je neophodno jeste da očekivanja i zahtevi budu precizni i jasni, kao i da se razume upotreba aplikacije u realnom vremenu.

Postoji mnogo vrsta sistemskog testiranja, i potrebno je doneti odluku koje od njih će biti sprovedene, u zavisnosti od zahteva, tipa aplikacije i raspoloživih resur-

sa. Neke od vrsta sistemskog testiranja koje se odnose na nefunkcionalne osobine softvera su:

- Testiranje oporavka (eng. *recovery testing*) — nakon što se izazove pad sistema, proverava se da li se on vraća u prvobitno stanje na ispravan način.
- Testiranje performansi (eng. *perfomance testing*) — proverava se skalabilnost, pouzdanost, i vreme odgovora sistema.
- Testiranje sigurnosti (eng. *security testing*) — proverava se da li je sistem adekvatno zaštićen od upada ili gubitka podataka.
- Regresiono testiranje (eng. *regression testing*) — proverava se da li su se pojavile neke naknadne greške pri dodavanju novih funkcionalnosti.
- Testiranje kompatibilnosti (eng. *compatibility testing*) — proverava se da li sistem radi ispravno u različitim okruženjima, npr. kada se koristi na drugom hardveru ili operativnom sistemu.

Pisanjem sistemskih testova utvrđuje se kvalitet i pouzdanost softvera, umanjuje rizik od neispravnosti i povećava zadovoljstvo korisnika sistema. Temeljnim testiranjem sistema mogu se otkriti i ispraviti novi problemi pre samog puštanja u rad, koje nije bilo moguće primetiti u ranijim fazama testiranja.

2.3 Testiranje funkcionalnih programa — Elm i Elixir

Najjednostavniji kôd za testiranje jeste čista funkcija. Pri testiranju čiste funkcije, s obzirom na to da ne postoje propratni efekti, test može da se fokusira samo na dve stvari: ulazne podatke i sam izlaz funkcije.

Kada je u pitanju čista funkcija, jedina priprema koja je potrebna jesu podaci koji će se proslediti kao parametri. Drugi korak jeste poziv funkcije, sa prosleđenim argumentima. Faza verifikacije podrazumeva samo provere nad rezultatom. Testovi su veoma jednostavni jer ne moraju da brinu o propratnim efektima i njihovim neželjenim posledicama. Međutim, aplikacije se u većini slučajeva neće sastojati od isključivo čistih funkcija.

U ovom radu, testiranje funkcionalnih programa ograničeno je na testiranje programa napisanih na dva različita moderna funkcionalna programska jezika. Serverski

deo aplikacije koja se testira napisan je na programskom jeziku *Elixir*, u razvojnom okviru namenjenom razvoju veb aplikacija pod nazivom *Phoenix*. Drugi programski jezik, korišćen pri implementaciji klijentske strane testirane aplikacije, naziva se *Elm*. Iako oba pripadaju funkcionalnoj paradigmi, *Elixir* i *Elm* se značajno razlikuju po svojim osobinama i namenama. Stoga, testiranje programa napisanih u svakom od ovih jezika podrazumeva različite pristupe i pravila u pisanju testova.

Programski jezik Elixir i testiranje

Elixir je dinamički tipiziran, funkcionalan programski jezik opšte namene, a uspešno se koristi u razvoju veb aplikacija. Radi na virtuelnoj mašini programskog jezika *Erlang*, koji je poznat po podršci za rad sistema koji su otporni na greške [23, 25]. *Elixir* i *Erlang* su kompatibilni, pa zbog toga *Elixir* može direktno koristiti *Erlang* biblioteke i module.

Elixir nije čist funkcionalni jezik, jer dozvoljava propratne efekte. Pored toga, odlikuje ga imutabilnost podataka i izraženo poklapanje obrazaca, što dovodi do konciznog koda koji je lak za održavanje. Jedna od prednosti ovog jezika je njegova skalabilnost. *Elixir* kôd se izvršava na virtuelnoj mašini paralelno u vidu veoma lakih niti (eng. *lightweight thread*), koje se nazivaju procesi. Ovi procesi su potpuno izolovani, ne dele memoriju i komuniciraju preko asinhronih poruka. Na istoj mašini se konkurentno može izvršavati na hiljade ovakvih procesa, a da svaki od njih koristi sve resurse. Ovo omogućava *Elixir* sistemima da budu distribuirani, skalabilni i otporni na greške.

Jedan od najčešćih načina baratanja greškama u ovom programskom jeziku je upotreba izuzetaka, koji se koriste kada se neke specifične greške dogode u kodu. Izuzeci mogu biti različitih tipova, kao što su *ArithmeticError, RuntimeError, ArgumentError* i slično. Takođe, postoji mogućnost korisnički definisanih izuzetaka, kreiranjem modula u kome je neophodno iskoristiti ključnu reč *defexception*. Najčešće se definiše sa poljem *message* i niskom odgovarajuće poruke, kao što je prikazano u primeru koda 2.4. Ovako definisana greška se u kodu može podići u odgovarajućim situacijama.

```
iex > defmodule MyError do
iex >     defexception message : ''default message''
iex > end

iex > raise MyError
** (My Error) default message
```

Listing 2.4: Definicija izuzetka u programskom jeziku *Elixir*

Izuzetak u jednom procesu neće nikako uticati na izvršavanje drugih procesa, zahvaljući njihovoj međusobnoj izolovanosti. Ovo omogućava definisanje supervizorskih procesa, čija je uloga da kada se neki proces neočekivano prekine, umesto njega započnu novi. Zahvaljujući supervizorima, *Elixir* podstiče dopuštanje neuspešnih procesa pre nego oporavak od svake moguće greške upotrebom *try / rescue* koncepta.

Navedene osobine *Elixir* programa omogućavaju jednostavnije testiranje. Imutabilnost olakšava pisanje čistih funkcija, za koje se mogu pisati testovi jedinica koda u izolaciji i time osigurati ispravnost svake komponente. Iako podstiče pisanje funkcionalnog i čistog koda, *Elixir* nije čist funkcionalni jezik i dozvoljava funkcije sa propratnim efektima. Postoje dve strategije pomoću kojih se olakšava testiranje ovakvih funkcija [32]. Prva je izdvojiti logiku u čiste funkcije, a druga dizajnirati funkcije tako da koriste neku od metoda ubrizgavanja zavisnosti (eng. *dependency injection*) [2], što omogućava izolaciju koda.

U pisanju testova u *Elixir-u* značajno pomažu poklapanje obrazaca i upotreba čuvara (eng. *guards*). Čuvari su uslovi nad argumentima funkcije koji se navode prilikom njenog definisanja. Ova svojstva omogućavaju pokrivanje velikog broja različitih putanja izvršavanja u testovima. *Elixir* ima ugrađen razvojni okvir za testiranje koji je razvijan zajedno sa samim jezikom, koji pruža podršku za pokrivanje svih mogućih slojeva aplikacije pouzdanim i lako održivim testovima. *Elixir* razvojno okruženje za testiranje detaljno je predstavljeno u poglavlju 4, na testovima pisanim za različite komponente serverske strane aplikacije *Portal MSNR*.

[2] Ubrizgavanje zavisnosti je obrazac u kom objekat ili funkcija prihvata druge objekte ili funkcije od kojih zavisi. Jedan od oblika inverzije kontrole, za cilj ima da razdvoji konstrukciju i upotrebu objekata i time smanjuje spregnutost programa.

Programski jezik Elm i testiranje

Elm je statički tipiziran, čist funkcionalan programski jezik, namenjen programiranju korisničkog interfejsa, i njegov kôd se transpilira u *JavaScript*. Pored toga što je programski jezik, *Elm* je i platforma za razvoj aplikacija. Na zvaničnoj stranici može se pronaći vodič kroz *Elm*, pomoću kog se lako savladavaju osnovni koncepti jezika [24]. Sa sobom, ovaj jezik donosi alat i okruženje *Elm* (eng. *Elm Runtime*).

Jedna od najvažnijih osobina jeste nepromenljivost podataka (imutabilnost) i odsustvo promenljivih. Funkcije su Karijeve (eng. *Curried*), što znači da su funkcije jednog argumenta koje kao povratnu vrednost mogu imati drugu funkciju, čime olakšavaju kompoziciju funkcija. *Elm* sam zaključuje tip funkcije, ali podstiče i navođenje anotacije u liniji iznad definicije. Prilikom kompilacije, vrši se poklapanje anotacije sa stvarnim tipom, što dovodi do lakšeg uočavanja grešaka. Funkcije su čiste i podstiče se upotreba operatora prosleđivanja i kompozicija funkcija, što dovodi do čitljivijeg koda. Primer koji ovo prikazuje dat je u kodu 2.5. Nad listom niski potrebno je izvršiti nekoliko operacija, i to se može uraditi na dva prikazana načina. Drugi način, u kom se uz pomoć operatora `|>` leva strana prosleđuje kao poslednji argument funkcije sa desne strane, kôd postaje mnogo jednostavniji za razumevanje.

```
list = ["one", "two", "three", "four", "five", "six"]

-- without piping
withoutPipe =
    List.concatMap (\s -> if String.length s > 3 then [String.
   toUpper s] else []) list

-- with piping
withPipe =
        list
        |> List.map String.toUpper
        |> List.filter (\s -> String.length > 3)
        |> String.join " "
```

Listing 2.5: Primer kompozicije funkcija upotrebom operatora prosleđivanja

Za aplikacije napisane u ovom programskom jeziku važi da u praksi ne izbacuju neplanirane greške tokom izvršavanja (eng. *No Runtime Exceptions*). Sintaksa jezika je jednostavna, a jedan od najkorisnijih alata jeste njegov karakteristični kompila-

tor[3]. *Elm* kompilator je veoma udoban za upotrebu, jer ako dođe do greške, daje konkretna objašnjenja zbog čega je došlo do nje i predloge načina za njeno ispravljanje. Kompilator i ostali alati napisani su na programskom jeziku *Haskell*, koji je stoga značajno uticao na *Elm* [26]. *Haskell* podržava pisanje kratkog i čistog koda, i time osigurava veću pouzdanost. Pored kompilatora, korisnici mogu proveravati ispravnost napisanih funkcija upotrebom interpretatora *elm repl*, koji je takođe veoma prijatan za korišćenje. Jednostavno se uključe željeni moduli i funkcije i pozivaju sa konkretnim ulazima iz komandne linije.

Jedan od razloga za odsustvo grešaka tokom izvršavanja je svojstvo da *Elm* tretira greške kao podatke. Umesto da dođe do pada programa, neuspešni slučajevi se modeluju korisnički definisanim tipovima. U primeru koda 2.6 prikazan je jedan takav tip. Ako se tip *MaybeAge* iskoristi kao tip povratne vrednosti funkcije, takva funkcija će uvek vratiti neku vrednost. Za slučaj neispravnog ulaza, funkcija će vratiti *InvalidInput*, i time pokriti sve moguće putanje izvršavanja.

```
type MaybeAge
  = Age Int
    | InvalidInput

toAge : String -> MaybeAge
toAge input =
  ...
```

Listing 2.6: Primer definicije i upotrebe korisnički definisanog tipa

Za obradu grešaka se najčešće koriste sledeća dva tipa:

1. *type* ***Maybe*** *a = Just a | Nothing* — kada očekivani podatak možda postoji, a u slučaju da ne postoji, biće Nothing.

2. *type* ***Result*** *error value = Ok value | Err error* — koristi se kao povratna vrednost koja može pružiti dodatne informacije o grešci.

Maybe i *Result* se koriste umesto vrednosti `null`, koja u *Elm-u* ne postoji. Ako se neuspešni slučajevi definišu na ovaj način, sprečavaju se neprijatna iznenađenja. Tip *Result* može pomoći i kod oporavka nakon greške. Ako na osnovu poruke o grešci korisnik zaključi tačno koji problem je u pitanju, znaće kako i na kom mestu to da ispravi.

[3]Strogo govoreći, u pitanju je transpilator — prevodilac sa jednog programskog jezika na drugi. U ovom radu će se označavati kao kompilator, jer se često u literaturi tako naziva.

Zahvaljujući navedenim svojstvima programskog jezika *Elm*, testiranje u ovom jeziku postaje veoma jednostavno. Čist funkcionalni jezik kao što je *Elm*, sa svojim sistemom tipova već pokriva veliki spektar mogućih ishoda. Funkcije su napisane tako da budu male i deklarativne, bez propratnih efekata, i često se njihovo testiranje svodi na pisanje koda sa istom logikom dva puta. To je nepoželjno, jer testovi treba da budu otporni na refaktorisanje, a u ovom slučaju bi morale da se uvode izmene na dva mesta. U slučaju trivijalnih funkcija, jednostavno čitanje koda je često dovoljno. Postavlja se pitanje šta je uopšte vredno testiranja, naročito ako se ne koristi razvoj vođen testovima, već se testovi pišu nakon što je aplikacija koja ide u produkciju već napisana.

Iako je *Elm* dizajniran tako da tipovi pokriju najveći deo korektnosti programa, oni ipak ne mogu uhvatiti baš svaku grešku. Može da se desi da je neka funkcija komplikovana, i da tipovi ne mogu mnogo da pomognu u proveri njene ispravnosti. Kako bi se izbegla dupla implementacija, funkcije treba tretirati kao crne kutije. U *Elm-u*, to se svodi na testiranje zasnovano na svojstvima (eng. *property based testing*), gde se za veliki broj različitih ulaza koji se generišu nasumično testira očekivano ponašanje. Ovakvo testiranje se naziva i rasplinuto, ili faz testiranje (eng. *fuzz testing*). U specifičnim slučajevima gde je potrebno proveriti neki granični slučaj koriste se klasični testovi jedinica koda. U poglavlju 5 detaljno su objašnjeni koncepti i načini pisanja testova u programskom jeziku *Elm*, na primeru testiranja klijenta aplikacije *Portal MSNR*.

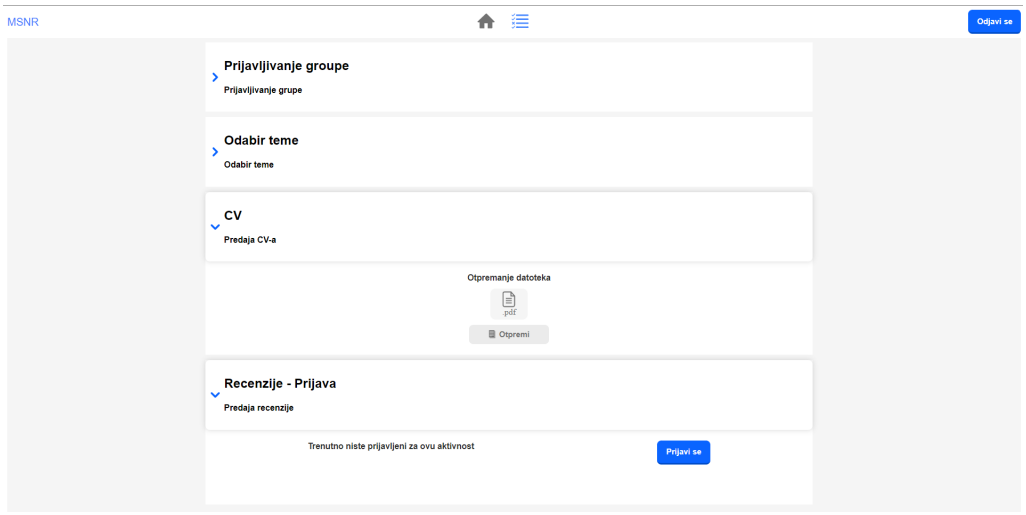
Glava 3

Portal MSNR

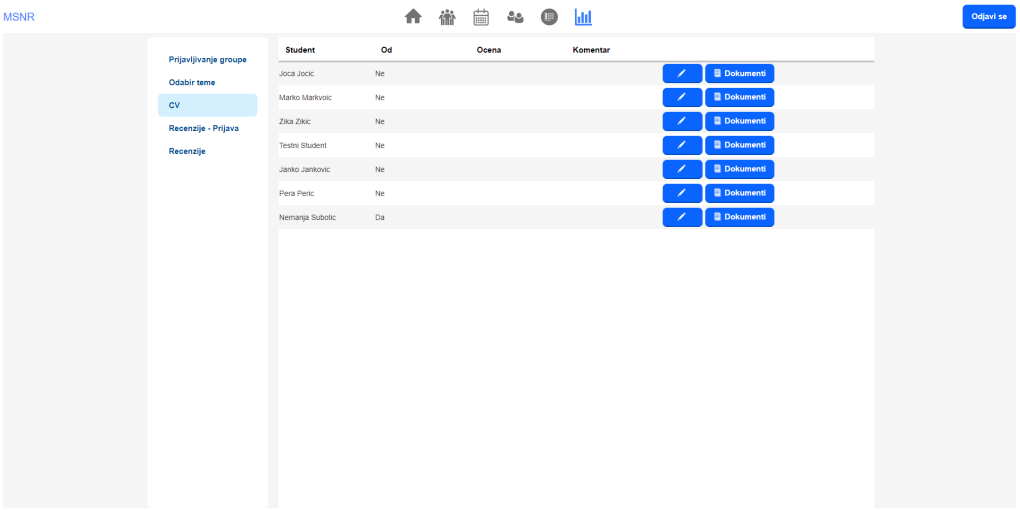
Kôd aplikacije pod nazivom *Portal MSNR* koja će biti testirana javno je dostupan na *GitHub-u* [1]. Implementacija portala nije deo ovog rada, a u ovom poglavlju biće objašnjene njegove glavne funkcionalnosti. *Portal MSNR* je veb aplikacija namenjena praćenju i upravljanju aktivnostima kursa *Metodologija stručnog i naučnog rada* [34]. Studenti na ovom kursu treba da steknu različite veštine koje se tiču pravilnog pisanja i recenziranja naučnih radova, pisanja CV-a, držanja prezentacija, i komunikacije u radu na timskim projektima.

3.1 Funkcionalnosti i osnovni entiteti portala

Različite aktivnosti na kursu *Metodologija stručnog i naučnog rada* implementirane su kao funkcionalnosti aplikacije. Korisnik portala može imati jednu od dve uloge: *student* ili *profesor*. Student na početku mora da podnese zahtev za registraciju, koju nakon toga odobrava profesor, i zatim student ima mogućnost da se prijavi na portal. Jedna od obaveza studenata na kursu jeste pisanje seminarskog rada — profesor vrši odabir tema za tekuću godinu, a studenti treba da prijave svoju grupu za izradu seminarskog rada. Student ima i opciju da se prijavi za recenziranje radova drugih studenata, ukoliko to želi. Drugi zadatak koji se očekuje od studenata jeste pisanje CV-a. U okviru portala, student može priložiti tri različite vrste dokumenata — prvu verziju seminarskog rada, recenzije, i svoju prvu verziju CV-a. Profesor, pored toga što vrši pregled zahteva za registraciju i odabir tema, ima mogućnost dodavanja svih aktivnosti tokom godine, i na kraju — njihovo ocenjivanje. Na slici 3.1 prikazan je izgled stranice iz ugla korisnika sa ulogom studenta, a na slici 3.2 prikazano je kako to izgleda kada je u pitanju profesor.



Slika 3.1: Izgled stranice kada je korisnik student [34]



Slika 3.2: Izgled stranice kada je korisnik profesor [34]

Entiteti

Osnovni entiteti aplikacije predstavljeni su tabelama u bazi podataka i relacijama između njih. Polazni entiteti su *zahtev za registraciju studenata*, *korisnik* i *semestar*. U tabeli korisnika inicijalno postoji jedan nalog koji ima ulogu profesora, a pri odobravanju registracije studenta kreira se nalog sa ulogom studenta, i studentu se šalje elektronska pošta sa vezom za postavljanje lozinke. Pored unosa u tabelu *users*, vrše se unosi u još dve tabele: *students*, koja sadrži referencu ka korisniku i *students_semesters*, koja predstavlja relaciju između studenta i semestra, a ima i

referencu ka tabeli *groups* — svaki student u toku jednog semestra može pripadati jednoj grupi. Nakon što profesor odabere teme za seminarske radove, vrši se unos u tabelu *topics*, koja ima referencu ka semestru u kom se mogu odabrati. Prethodno navedeni tipovi aktivnosti predstavljeni su tabelom *activity_types*, a tabela *activities* predstavlja relaciju između tipa aktivnosti i semestra. Tabela *assignments* odnosi se na dodeljene aktivnosti koje mogu biti grupne ili individualne, te može imati referencu ka studentu ili ka grupi. Većina dodeljenih aktivnosti podrazumeva predaju dokumenata, koji će se nalaziti na serveru, a informacije o predatim dokumentima čuvaju se u tabeli *documents*. Ova tabela sadrži referencu ka korisniku koji je priložio dokument, a tabela *assignments_documents* vezuje dokument i dodeljenu aktivnost.

Spisak naziva entiteta i tabela u okviru baze podataka koje njima odgovaraju dati su u tabeli 3.1. Svaki od ovih entiteta, kao i relacije između njih, biće pojedinačno istestirani u narednom poglavlju.

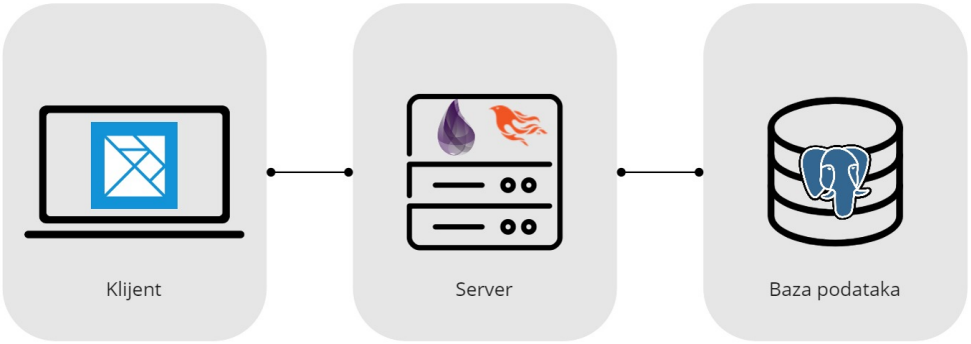
Tabela 3.1: Entiteti portala i odgovarajuće tabele u bazi

Entitet	**Tabele u bazi podataka**
Zahtev za registraciju studenata	*student_registrations*
Korisnik	*users*
Semestar	*semesters*
Student	*students* i *student_semester*
Grupa	*groups*
Tema seminarskog rada	*topics*
Aktivnost	*activities*
Tip aktivnosti	*activity_types*
Dodeljene aktivnosti	*assignments*
Dokument	*documents* i *assignments_documents*

3.2 Arhitektura portala

Portal MSNR je primer klijent/server aplikacije koja se sastoji od tri sloja. Klijentski sloj implementiran je u programskom jeziku *Elm*, kao jednostranična aplikacija (eng. *Single Page Application — SPA*) koja predstavlja korisnički interfejs. U sredini se nalazi aplikacioni veb interfejs koji je implementiran u programskom jeziku *Elixir* pomoću razvojnog okvira *Phoenix*, u stilu arhitekture *REST* (eng. *Representational State Transfer*) [5]. Treći sloj predstavlja relaciona baza podata-

ka, i sistem za upravljanje bazom podataka *PostgreSQL* [17]. Slika 3.3 prikazuje navedenu arhitekturu.

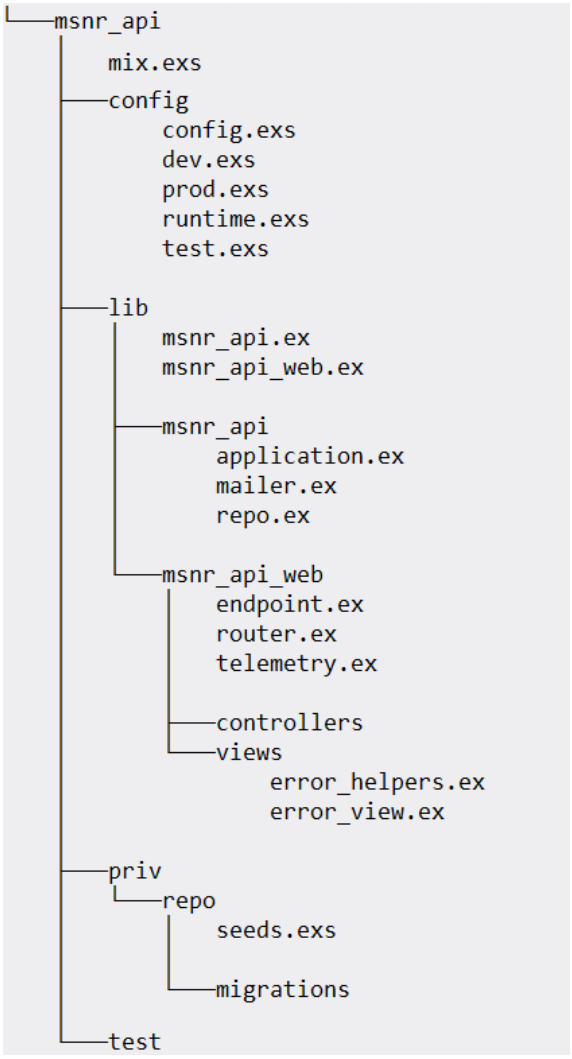


Slika 3.3: Arhitektura Portala MSNR [34]

Struktura serverske strane portala

U programskom jeziku *Elixir*, za razvoj veb aplikacija koristi se razvojni okvir *Phoenix* [27]. Zasnovan je na obrascu *model-pogled-upravljač* (eng. *Model-View-Controller pattern, MVC*). Serverski deo aplikacije *Portal MSNR* implementiran je kao *Phoenix* projekat. Preciznije, projekat je u osnovi *Mix* projekat, sa *Phoenix* proširenjima. *Mix* je osnovni alat ovog jezika koji se koristi za kreiranje, prevođenje i testiranje projekata. Pored ovog alata, potrebno je prethodno instalirati i menadžer paketa za ekosistem *Erlang* pod nazivom *Hex* [16]. Pri kreiranju *Phoenix* projekta, dodeljeno mu je ime *msnr_api*. Na slici 3.4 je prikazana struktura projekta nakon uspešnog pokretanja komande za kreiranje.

U direktorijumu *lib* nalaze se dva konteksta (eng. *context*), tj. dva modula, od kojih svaki grupiše funkcije sa zajedničkom svrhom. Prvi kontekst je *MsnrApi*, unutar koga je enkapsulirana sva domenska i poslovna logika, i definisani svi entiteti i funkcije za rad sa njima. Inicijalno su kreirana tri podmodula ovog konteksta: *MsnrApi.Application*, koji pokreće aplikaciju, *MsnrApi.Repo*, koji je zadužen za komunikaciju sa bazom, i *MsnrApi.Mailer*, koji služi za slanje elektronske pošte. Drugi kontekst ima naziv *MsnrApiWeb*, i on sadrži implementaciju za poglede i upravljače unutar arhitekture MVC. Njegovi podmoduli *MnsrApiWeb.Endpoint* i *MsnrApiWeb.Router* imaju ulogu u pripremi HTTP zahteva i njihovom prosleđivanju odgovarajućim upravljačima.



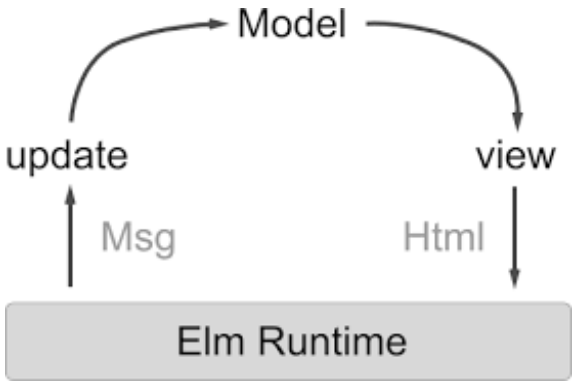
Slika 3.4: Struktura *Phoenix* projekta *msnr_api* [34]

Za sve obrade HTTP zahteva koristi se biblioteka *Plug* [11]. Utikač (eng. *plug*) je funkcija koja ima kao ulaznu i povratnu vrednost strukturu *Plug.Conn* koja sadrži sve informacije o primljenom HTTP zahtevu. *Phoenix* poziva utikače jedan za drugim, i svaki od njih transformiše ovu strukturu dok se obrada zahteva ne završi, i na kraju odgovor pošalje korisniku. Sa utikačima se povezuje veb server pod nazivom *Cowboy*, a u *mix.exs* je automatski ubačena zavisnost *plug_cowboy*.

Kada se kreira novi *mix* projekat, pored konfiguracione datoteke *mix.exs*, i direktorijuma *lib* koji sadrži osnovni kôd aplikacije, kreira se i direktorijum *test*. Unutar ovog direktorijuma će biti smešteni svi testovi vezani za serversku stranu aplikacije.

Struktura klijentske strane portala

Klijent aplikacija *Portala MSNR* predstavlja jedan *Elm* projekat. U svakom *Elm* programu uočava se obrazac projektovanja koji se naziva *Model-Pogled-Ažuriranje* (eng. *Model View Update — MVU*), ili *Arhitektura Elm* [31]. Model predstavlja stanje aplikacije, pogled se odnosi na transformaciju stanja u HTML, a ažuriranje na promene stanja. *Elm* program radi tako što se generiše HTML koji se prikazuje u pretraživaču, a nakon toga pretraživač šalje poruku programu ako se nešto dogodi. Na osnovu primljene poruke, funkcija *update* kreira novi model, koji se prosleđuje funkciji *view*, na osnovu koje se generiše HTML. Ovaj proces prikazan je na slici 3.5. U arhitekturi *Elm* celokupno stanje aplikacije se nalazi na jednom mestu (u modelu), a protok podataka je uvek u jednom smeru.



Slika 3.5: *Arhitektura Elm* [34]

Inicijalizacija ovog projekta podrazumeva kreiranje jednog praznog direktorijuma *src* i datoteke *elm.json*, a sam programer odlučuje o organizaciji datoteka unutar projekta. Na slici 3.6 prikazano je rešenje organizacije *Elm* datoteka u okviru aplikacije *Portal MSNR*.

U korenu projekta se nalazi osnovna datoteka *Main.elm*, koja sadrži funkciju *main* sa definicijom *Elm* aplikacije. Ostale datoteke sadrže definicije osnovnih stranica, entiteta, putanja i modula za komunikaciju sa serverom. U posebnim direktorijumima, izdvojene su datoteke za prikazivanje studentskih i profesorskih stranica.

Korisnički interfejs portala implementiran je pomoću četiri funkcije: *sandbox, element, document, application*. Ove funkcije se nalaze unutar modula *Browser*, koji je deo paketa čija je uloga kreiranje *Elm* programa u pretraživaču. Funkcija *sandbox* omogućava bazičnu interakciju sa korisnicima, bez komunikacije sa spoljnim svetom. Tu komunikaciju omogućava funkcija *element*, pomoću koncepta komande,

Slika 3.6: Struktura *Elm* projekta *msnr_elm* [34]

supskripcije, portova i oznaka (eng. *flags*). Funkcija *document* proširuje prethodnu funkciju tako što upravlja celim dokumentom i pruža kontrolu nad HTML elementima naslova i tela. Funkcija *application* kreira aplikaciju koja upravlja promenama nad url adresama.

Aplikacija je transpilirana tako da se kreira datoteka *app.js*, koja se uključuje u dokument *index.html*. Pokreće se pozivanjem funkcije *init* iz modula *Main* i tada se vrši prosleđivanje putanje ka veb interfejsu preko oznaka.

Elm aplikacija podeljena je na tri osnovna dela: stranice koje se koriste za prijavljivanje i registraciju korisnika, studentsku stranicu, i profesorske stranice. Pored njih, postoje i stranice koje nisu izdvojene u posebne module — početna stranica i stranica koja se prikazuje u slučaju pogrešne putanje.

3.3 Testiranje portala

Na početku testiranja bilo koje aplikacije je neophodno odrediti prioritete, najpre fokusiranjem na glavne funkcionalnosti koja ona treba da ispuni. Najvažnije je jasno formulisati koje su kritične komponente aplikacije, a koje je moguće ostaviti nepokrivene testovima u slučaju da za to ne bude vremena. Takođe, treba proceniti koliko bi koštalo ispraviti greške u različitim komponentama, i na osnovu toga doneti odluke o tome koji aspekti aplikacije moraju biti testirani [6, 30]. Naredni testovi predstavljaju ono što je najvažnije pokriti u slučaju ograničenog vremena i budžeta pri testiranju ovakve aplikacije.

Testiranje aplikacije *Portal MSNR* podeljeno je na tri glavne celine: pisanje testova za serversku stranu u programskom jeziku *Elixir*, pisanje testova za klijentsku stranu portala u programskom jeziku *Elm*, i manuelni testovi koji odgovaraju sistemskom testiranju celokupne aplikacije. *Elixir* testovi koje je potrebno napisati obuhvataju dva sloja serverske aplikacije. Prvi sloj implementira sve entitete i funkcije koje rade sa njima (model u arhitekturi MVC), a drugi sloj čine upravljači i pogledi. Na početku, neophodno je testirati komunikaciju sa bazom podataka. To podrazumeva testiranje ispravnosti polja i njihovih tipova u tabelama, a zatim pisanje integracionih testova koji proveravaju zahteve API-ju i verifikuju odgovore od baze. U ovom projektu, to su većinom jednostavne operacije dohvatanja, unosa i brisanja podataka iz baze. Za svaki od entiteta navednih na početku poglavlja, potrebno je napisati testove koji proveravaju navedene koncepte. Nakon toga, testirati upravljače i poglede kao jednu celinu, i u tim testovima prikazati kako proveriti očekivane odgovore na HTTP zahteve, koji su u ovom slučaju u JSON formatu. Na kraju, u delu sa *Elixir* testovima, potrebno je pokriti i nekoliko pomoćnih funkcija koje se koriste u aplikaciji, pisanjem testova jedinica koda.

Drugi deo rada, koji se odnosi na testiranje klijentske strane portala, obuhvata testove napisane u jeziku *Elm*. U ovom delu, prvo je potrebno proveriti ispravnost važnih funkcija koje se nazivaju dekoderi. Njihova uloga je da JSON podatke dobijene od servera prevedu u *Elm* vrednosti. Nakon toga, prateći arhitekturu *Elm*, neophodno je testirati funkcije ažuriranja i funkcije pogleda, za svaku od stranica. Testovi funkcija ažuriranja treba da provere da li se model (stanje) menja na ispravan način, a testovi funkcija pogleda pokrivaju iscrtavanje HTML sadržaja i interakciju sa korisničkim interfejsom.

Iako je fokus teze na automatskom testiranju, potrebno je sprovesti i određe-

ne manuelne testove. Oni se uglavnom odnose na provere korisničkog iskustva i podrazumevaju pokretanje aplikacije u pretraživaču i prolazak kroz moguće slučajeve upotrebe kliktanjem na dugmiće, unos teksta u odgovarajuća polja, proveru da li se pojavljuju očekivane poruke i da li su dovoljno informativne. Jednostavnim pregledom stranica mogu se videti slovne greške, ili pogrešno formatirani elementi korisničkog interfejsa. Kompatibilnost aplikacije sa različitim pretraživačima može se proveriti njenim pokretanjem na više njih. Najvažniji manuelni testovi koji se mogu sprovesti pri testiranju *Portala MSNR* su provere HTTP zahteva i odgovora sa klijentske strane. Zbog ograničenja okruženja za testiranje u programskom jeziku *Elm*, nije moguće napisati automatske testove koji proveravaju komunikaciju korisničkog interfejsa sa aplikativnim interfejsom na serveru. Umesto toga, potrebno je iskoristiti alat pretraživača *Inspect* i karticu *Network*, koja daje detaljan prikaz svih aktivnosti na mreži koje se dešavaju pri interakcijama sa veb stranicom. Ona beleži i prikazuje sve HTTP zahteve i odgovore koji se šalju pri akcijama kao što su klik na dugme ili slanje forme. Proučavanjem detalja unutar HTTP zahteva u *Network* kartici može se utvrditi da li se šalju odgovarajući zahtevi i dobijaju odgovarajući odgovori kada se komunicira sa serverom.

Glava 4

Testiranje serverskog dela aplikacije

U ovom poglavlju biće predstavljeni različiti koncepti testiranja u programskom jeziku *Elixir*, kroz pisanje testova za serverski deo aplikacije *Portal MSNR*. *ExUnit* je *Elixir-ov* ugrađeni razvojni okvir koji ima sve što je neophodno za iscrpno testiranje koda i biće osnova za sve testove kroz ovo poglavlje [13]. Testovi koji su predstavljeni u ovom delu mogu se naći na adresi `https://github.com/Anakin2012/master-rad/tree/main/testing-msnr-portal/portal/msnr_api/test`.

4.1 Uvod u testiranje u okruženju ExUnit

Pisanje testova u programskom jeziku *Elixir* je moguće bez potrebe za drugim bibliotekama, jer je *ExUnit* razvijan zajedno sa samim jezikom od početka. Svi testovi su implementirani kao *Elixir* skripte, pa je pri davanju imena testu neophodno koristiti ekstenziju datoteke *.exs*. Pre pokretanja testova potrebno je pokrenuti *ExUnit*, kao što je prikazano u primeru koda 4.1. Ova naredba se obično navodi unutar automatski generisane datoteke *test/test_helper.exs*.

```
# test/test_helper.exs

ExUnit.start()
```

Listing 4.1: Pokretanje *ExUnit*

Testovi se pokreću najpre pozicioniranjem u direktorijum projekta, a zatim navođenjem komande `mix test`. Ova komanda pokreće sve testove koji se nalaze unutar

direktorijuma *test*. Navođenjem parametra `--only` i imena testa ili modula može se pokrenuti specifičan test ili skup testova unutar jednog modula. Pozivanje naredbe `mix test` pokreće sve testove, i daje sledeći izlaz:

```
PS C:\Users\panap\testing-msnr-portal\portal\msnr_api> mix test


..............
Finished in 0.2 seconds (0.0s async, 0.2s sync)
16 tests, 0 failures

Randomized with seed 801308
```

ExUnit će testove izvršavati nasumičnim redosledom, koristeći ceo broj kao seme nasumičnosti (eng. *randomization seed*). Poželjno je da se testovi izvršavaju nasumičnim redosledom, jer se tako osigurava njihova izolovanost. Ako se desi da određeni test sporadično ne prolazi, to može biti jer neki od prethodnih testova menja stanje i ima posledice po druge testove. Ovako nešto može da se desi ako se testovi izvršavaju nekim određenim redosledom, i taj redosled se može dobiti pomoću ovog nasumičnog broja koji je dat u samom izlazu. Pokretanjem testova ponovo koristeći taj konkretni ceo broj kao seme nasumičnosti može se pronaći uzrok greške.

Svita testova (eng. *test suite*) je kolekcija testova slučajeva upotrebe, koji imaju isti posao, ali različite scenarije. Ona može služiti kao dokumentacija, sa opisima o očekivanom ponašanju koda, tako da treba voditi računa da bude dobro organizovana. *ExUnit* dolazi sa veoma korisnim funkcijama i makroima koji omogućavaju tu organizaciju u jednu čitljivu i održivu datoteku. Alat `describe` omogućava davanje opisa grupe testova, kao i dodeljivanja zajedničke pripreme podataka za celu grupu. Preporuka je za početak grupisati testove po funkciji, kao što je prikazano u primeru koda 4.2, ali odluka o načinu grupisanja je na pojedincu. Svrha je čitljivost i lakše razumevanje.

Testovi u *Elixir* projektima se organizuju u module i test primere. U ovom primeru, modul pod nazivom *MsnrApi.UnitTests.PasswordTest* sadrži testove jedinica koda koji se odnose na kontekst koji opisuje šifre korisnika (*MsnrApi.Accounts.Password*). Blok `describe` se sastoji od dva test primera i odnosi se na funkcije *hash* i *verify_with_hash*. Njihova uloga je da hešuju lozinku upotrebom nasumične tzv. *salt* niske i zatim verifikuju da je lozinka ispravno heširana, tj. zaštićena tako da postoji kao nasumična niska u bazi. Više o ovim funkcijama može se naći u dokumentaciji modula *Pbkdf2* [20]. Navođenjem ključne reči `test`, a za njom niske koja treba da opiše šta je to što test treba da uradi, definiše se jedna funkcija koja predstavlja test primer. U primeru koda 4.2 data su dva test primera, od kojih jedan proverava uspe-

šno izvršavanje funkcije kada se prosledi ispravna lozinka funkciji *verify_with_hash*, a drugi proverava da li se javlja greška kada se prosledi neodgovarajuća niska.

Unutar jednog test primera poziva se funkcija ili upravljač i proverava se očekivani rezultat. Makroom `assert` se testira da li je izraz istinit. U slučaju da nije, test ne prolazi i izbacuje grešku. Ako funkcija *verify_with_hash* vrati *true*, ovaj test uspešno prolazi. Na primeru testa koji proverava neuspešnu putanju izvršavanja, prikazan je makro `refute`, koji se koristi kada je potrebno proveriti da li je izraz neistinit (eng. *false*). U ovom slučaju očekuje se da funkcija vrati *false* kada joj se prosledi neispravna lozinka.

```
defmodule MsnrApi.UnitTests.PasswordTest do
...
  describe "verify password" do
    test "success: verifies the password by hashing" do
      password = "somepass123"

      assert hash = Password.hash(password)
      assert Password.verify_with_hash(password, hash) == true
    end

    test "error: returns false when given wrong password" do
      password = "somepass123"
      wrong = "wrong"

      assert hash = Password.hash(password)
      refute Password.verify_with_hash(wrong, hash)
    end
  end
```

Listing 4.2: Opisivanje testova unutar jedne grupe, na primeru funkcije za verifikaciju lozinke

U slučaju da leva i desna strana izraza navedenog nakon makroa `assert` nisu jednake, test ne prolazi, a *ExUnit* daje obaveštenje o tome koji od testova su neuspešni, kao i koje su prava i očekivana vrednost. Izlaz koji se dobije u slučaju da rezultat izvršavanja funkcije nije onaj koji je očekivan, prikazan je na listingu 4.3.

```
1) test verify password success: verifies the password by hashing (MsnrApi.UnitTests.
    PasswordTest)
   test/msnr_api/unit_tests/password_test.exs:6
   Assertion with == failed
   code:  assert Password.verify_with_hash(password, hash) == true
   left:  false
   right: true
   stacktrace:
     test/msnr_api/unit_tests/password_test.exs:10: (test)
.
Finished in 5.0 seconds (0.0s async, 5.0s sync)
2 tests, 1 failure
```

Listing 4.3: Izlaz u slučaju testa koji ne prolazi

Pored najčešće korišćenog makroa `assert`, *ExUnit* nudi još nekoliko njih koji se mogu koristiti u različitim situacijama. U narednoj listi dati su nazivi i opisi nekih takvih makroa:

- `assert_raise` — koristi se kada je potrebno utvrditi da je podignut odgovarajući izuzetak.
- `assert_receive` — koristi se kada je potrebno proveriti da li je proces primio konkretnu poruku.
- `capture_io` — koristi se kada je potrebno proveriti da li se na standardnom izlazu ispisuje očekivano.
- `capture_log` — koristi se kada je potrebno proveriti sadržaj log poruka, npr. pri pozivu *Logger.info*.
- `setup` i `setup_all` — koriste se kada je potrebno izvršiti pripremu testova, pokreću se pre svakog testa, ili pre jedne grupe.

Upotreba makroa `setup` i `assert_raise` prikazana je u primeru dela koda 4.4. Kôd unutar makroa `setup` će se pokretati pre svakog testa, a u ovom primeru priprema podrazumeva eksplicitno dohvatanje konekcije sa bazom podataka pre izvršavanja svakog od testova. Povezivanje sa bazom podataka biće detaljno objašnjeno u narednoj sekciji. Test primer iz ovog koda pokriva neuspešan slučaj funkcije *get_user* koja treba da dohvati korisnika iz baze. Pri pokušaju dohvatanja nepostojećeg korisnika, očekuje se izuzetak tipa *NoResultsError*.

```
setup do
    Ecto.Adapters.SQL.Sandbox.checkout(MsnrApi.Repo)
end
  ...
    invalid_id = -1
    assert_raise Ecto.NoResultsError, fn ->
       Accounts.get_user!(invalid_id) end
```

Listing 4.4: Upotreba makroa `setup` i `assert_raise` na primeru funkcije *get_user*

4.2 Testiranje komunikacije sa bazom podataka

Kod testiranja spoljašnjih entiteta i servisa koji su van kontrole pojedinca koji piše ili testira kôd, često se koristi proces koji se naziva mokovanje (eng. *mocking*) [36]. Mokovanje podrazumeva simuliranje tih spoljašnjih entiteta, bez njihove stvarne upotrebe. Glavna svrha ovog procesa jeste izolacija jedinice koja se testira, bez uticaja ponašanja eksternih entiteta. Primer jednog takvog entiteta je baza podataka. Umesto prave baze, mogu se koristiti kontrolisani objekti koji će simulirati njeno ponašanje. Testovi jedinica koda tako mogu da pokriju neke kompleksne manipulacije sa podacima iz baze, a da se pritom ne koristi pravi sadržaj baze koji u tom trenutku nije važan.

U kontekstu ovog projekta, baza podataka je sastavni deo aplikacije i nad njom postoji potpuna kontrola. Ona se može pokretati i zaustavljati po želji, i nema nepredviđenih rizika u njenom ponašanju. Takođe, operacije koje su testirane u ovom delu su jednostavni upiti i načini interakcije sa bazom. S obzirom na to da ne postoji dodatna logika koju je potrebno testirati u izolaciji, u ovoj situaciji nije primenljivo mokovanje. Svi testovi unutar kojih se komunicira sa bazom podataka će ostvarivati prave konekcije i dohvatati stvarne podatke iz baze. Ovakvi testovi blisko oslikavaju kako bi se aplikacija ponašala i u produkciji. Dodatno vreme izvršavanja koje zahteva pristupanje bazi je u ovom slučaju prihvatljivo.

Testiranje u okruženju Ecto

Biblioteka *Ecto* zadužena je za sve interakcije sa relacionim bazama podataka u okruženju *Elixir* [8]. Može se podrazumevati da ova biblioteka radi kako treba, kako je široko korišćena i sama detaljno istestirana. Međutim, ono što je potrebno utvrditi jeste da li se unutar koda aplikacije ispravno pozivaju funkcije biblioteke *Ecto*. Pored

komunikacije sa bazom, *Ecto* ima i ulogu u validaciji. Moduli ove biblioteke koje je značajno naglasiti su: *Ecto.Repo*, *Ecto.Schema* i *Ecto.Changeset*. *Ecto.Repo* opisuje gde se nalaze podaci, odnosno definiše omotač oko baze preko kog se ostvaruje komunikacija sa bazom. *Ecto.Schema* ima ulogu u definisanju mapiranja eksternih podataka u *Elixir* strukture. Koncept skupa promena (eng. *changeset*) odnosi se na proces validacije podataka, njihovog konvertovanja i provere dodatnih uslova pre nego što se upišu u bazu. Modul *Ecto.Changeset* opisuje kako se menjaju podaci. U ovom delu, prikazani su testovi koji proveravaju da li kôd koristi funkcionalnosti biblioteke *Ecto* na ispravan način.

Ecto i svi potrebni moduli se podrazumevano uključuju prilikom kreiranja *Phoenix* projekta. Pre samog pisanja testova, neophodno je podesiti sve parametre za komunikaciju sa bazom podataka *PostgreSQL* u testnom okruženju. U datoteci *config/test.exs* potrebno je uneti podatke kao što je prikazano u primeru koda 4.5. Pokretanjem naredbe `MIX_ENV=test mix ecto.create` iz komandne linije, lokalno će se kreirati *msnr_api_test* baza podataka.

```
config :msnr_api, MsnrApi.Repo,
username: "postgres",
password: "1234",
database:"msnr_api_test#{System.get_env("MIX_TEST_PARTITION")}",
hostname: "localhost",
pool: Ecto.Adapters.SQL.Sandbox,
pool_size: 10
```

Listing 4.5: Konfiguracija baze podataka u testnom okruženju

Svi testovi u ovom delu nalaze se u direktorijumima *'/test/msnr_api/schema'* i *'/test/msnr_api/queries'*, u okviru projekta *msnr_api*.

Na početku, napisani su jednostavni testovi koji proveravaju ispravnost definisanja struktura pomoću modula *Ecto.Schema*. Primer definisanja entiteta dodeljenih aktivnosti dat je u primeru koda 4.6. Pomoću makroa `schema` i `field` definišu se tabele, njihova polja i relacije sa drugim tabelama. Oni istovremeno definišu i *Elixir* strukturu — u ovom primeru, ta struktura se naziva *Assignment*. Pored ovog, i svi ostali entiteti su definisani kao konteksti u okviru konteksta *MsnrApi*, koji sadrži domensku logiku aplikacije. Funkcija *changeset/2* biće objašnjena u delu koji govori o testiranju skupa promena.

```
defmodule MsnrApi.Assignments.Assignment do

  schema "assignments" do
    field :comment, :string
    field :completed, :boolean, default: false
    field :grade, :integer
    field :student_id, :id
    field :group_id, :id
    field :activity_id, :id
    field :related_topic_id, :id
    timestamps()
  end

  def changeset(assignment, attrs) do
    assignment
    |> cast(attrs, [:comment, :grade])
    |> validate_required([:comment, :grade])
  end
...
```

Listing 4.6: Shema tabele *assignments*

Testiranje polja i tipova

Test koji proverava polja i tipove tabele *assignments* dat je u kodu 4.7. Ovo je primer jednostavnog testa koji proverava da li definisana shema ima tačna polja i odgovarajuće tipove. Unutar testa se prvo prolaskom kroz sva polja strukture *Assignment* izvuku polje i njegov tip, i zatim se navodi ključna reč `assert`, kojom se proverava da li su prava polja i tipovi jednaki očekivanim. Lista *@expected_fields_with_types* definisana je kao lista parova polja i odgovarajućih tipova, kao što su navedeni u primeru 4.6. Unutar naredbe `assert`, i prava i očekivana lista pretvorene su u strukturu *MapSet*, kako bi se redosledi polja poklapali sa obe strane. Slični testovi napisani su i za sve ostale entitete navedene u sekciji 3.1.

```
test "it has the correct fields and types" do
  actual_fields_with_types =
    for field <- Assignment.__schema__(:fields) do
      type = Assignment.__schema__(:type, field)
      {field, type}
    end

  assert MapSet.new(actual_fields_with_types) ==
         MapSet.new(@expected_fields_with_types)
end
```

Listing 4.7: Test za proveru polja i tipova tabele *assignments*

Testiranje skupa promena

Funkcija *changeset* iz primera koda 4.6 obuhvata različite transformacije podataka, kao i njihovu validaciju pre unosa u bazu podataka. Svrha ove funkcije je da svi podaci koji se unose ili ažuriraju u bazi budu ispravni i u skladu sa zahtevima aplikacije. Svaka od shema ima svoju definiciju polja i svoju funkciju *changeset*. Tokom razvoja aplikacije, shemama se mogu dodavati različite izmene, kao što su nova polja, ili izmene u samim validacijama unutar funkcije *changeset*. Funkcija *cast* je prva u nizu funkcija koje se pozivaju i ona ograničava polja koja se mogu menjati. U slučaju modula *Assignment*, to su polja *comment* i *grade*. Funkcija *validate_required* proverava obavezna polja. Rezultat izvršavanja ovih funkcija je takođe struktura *Ecto.Changeset*, koja sadrži informacije o promenama koje treba izvršiti, validnost izmena i greške validacije, ukoliko one postoje.

Testovi koji se odnose na ove funkcije implementirani su unutar bloka `describe` *"changeset/2"*, za svaki od entiteta aplikacije. Oni pokrivaju i uspešan scenario, i neke od slučajeva greški. Koji od ovih scenarija će se desiti, zavisi od ispravnosti prosleđenih parametara funkcije. Parametri koji će se prosleđivati u testovima formirani su unutar pomoćnih funkcija koje se nalaze u modulu *SchemaCase*. On se nalazi u direktorijumu *msnr_api/test/support*, zajedno sa ostalim datotekama koje sadrže zajednički kôd. Da bi ova datoteka bila prepoznata kada se pokreću testovi, potrebno je dodati dve linije unutar datoteke *mix.exs*, koje su prikazane u kodu 4.8. Ovime se govori aplikaciji da uključi sve datoteke u direktorijumu *test* prilikom kompilacije u testnom okruženju. Tako *SchemaCase* postaje dostupan isključivo prilikom testiranja.

```
defp elixirc_paths(:test), do: [''lib'', ''test'']
defp elixirc_paths(_), do :[''lib'']
...
def project do [
    app: :msnr_api,
    version: "0.1.0",
    elixir: "~> 1.12",
    elixirc_paths: elixirc_paths(Mix.env()),
    ...
]
```

Listing 4.8: Uključivanje datoteka iz direktorijuma *test* pri kompilaciji u testnom okruženju

Što se tiče faze rušenja, *Ecto* obezbeđuje da svaki pojedinačni test ne mora da prati okruženje i vraća ga na prvobitno stanje. Za to je zadužen *Ecto.Sandbox*, koji omogućava paralelno izvršavanje testova bez deljenog stanja u bazi podataka i automatski vrši poništavanje svih promena u bazi na kraju svakog testa. Konfiguracija je prikazana u primeru koda 4.9. U datoteci *schema_case* potrebno je dodati blok `setup`, koji će svi testovi koji koriste ovaj obrazac pokretati na početku izvršavanja. Manuelni režim podrazumeva da će svaki test moći da zahteva svoju *Sandbox* konekciju. Takođe, u konfiguracionoj datoteci *config/test.exs*, u delu *Ecto*, dodaju se linije koje obaveštavaju *Ecto* da će se koristiti *Sandbox*. Ovime se obezbeđuje da nijedan test u kome su vršene izmene u samoj bazi ne mora da ima eksplicitnu fazu rušenja — na kraju testa baza podataka se automatski vraća u prvobitno stanje.

```
# msnr_api/test/schema_case.ex
  setup do
    Ecto.Adapters.SQL.Sandbox.mode(MsnrApi.Repo, :manual)
  end
...
# msnr_api/config/test.exs
  config :msnr_api, MsnrApi.Repo,
  database: "msnr_api_test",
  pool: Ecto.Adapters.SQL.Sandbox,
```

Listing 4.9: Podešavanje *Ecto.Sandbox*

Generisanje lažnih podataka

Za potrebe testiranja svih entiteta, biće neophodno obezbediti lažne podatke odgovarajućih tipova koji će odgovarati poljima u tabelama. Modul *SchemaCase* je jedan od pomoćnih modula u kojem se nalaze funkcije koje će se pozivati u skoro svim testovima koji proveravaju sheme u okviru baze podataka. Sadrži dve funkcije, od kojih jedna konstruiše realistične podatke ispravnih tipova, a druga konstruiše podatke koji su pogrešnog tipa u odnosu na polje tabele.

Za generisanje nasumičnih realističnih podataka korišćena je biblioteka *Faker* [10]. *Faker* je potrebno uključiti u zavisnosti projekta, dodavanjem linije `{:faker, "~> 0.17", only: :test}`, u delu `deps` konfiguracione datoteke *mix.exs*. Biblioteku nije potrebno koristiti u razvojnom i produkcionom okruženju, te se ovom linijom ograničava njena upotreba samo na testno okruženje. U tabeli 4.1 prikazani su neki od modula biblioteke *Faker* i njihove funkcije koje su korišćene prilikom generisanja podataka za testiranje.

Tabela 4.1: Moduli biblioteke *Faker*

Modul	**Funkcije modula**	**Opis funkcija**
Faker.Lorem	*characters()* *paragraph()* *sentence()* *word()*	generisanje nasumičnih karaktera, paragrafa, rečenica ili pojedinačnih reči
Faker.Person	*first_name()* *last_name()* *title()*	generisanje nasumičnih podataka u vezi sa osobom — ime, prezime, ili zvanje
Faker.Internet	*email()* *url()* *domain_name()*	generisanje nasumičnih podataka sa Interneta — imejl adrese, url putanje, nazivi domena
Faker.Random	*random_between(int, int)* *random_uniform()*	generisanje nasumičnih celih i realnih brojeva
Faker.File	*file_extension()* *file_name()*	generisanje nasumičnih ekstenzija i imena datoteka
Faker.Date	*backward(days)* *forward(days)*	generisanje nasumičnih datuma određeni broj dana unazad ili unapred

Pomoćne funkcije *valid_params* i *invalid_params* iz modula *SchemaCase* prikazane su u primeru koda 4.10. Ove funkcije kao povratnu vrednost imaju mapu koja sadrži niske naziva polja kao ključeve, i odgovarajuće generisane vrednosti koje njima odgovaraju. Prva funkcija generiše realistične podatke ispravnog tipa, i ona se poziva u testovima koji proveravaju ispravnu putanju. Druga funkcija generiše podatke neodgovarajućeg tipa i ona u testovima služi za sprovođenje neuspešne putanje izvršavanja. Na primer, za polja koja bi trebalo da budu niske, ova funkcija generiše podatak tipa *DateTime*.

```
def valid_params(fields_with_types) do
  valid_value_by_type = %{
    string: fn -> Faker.Lorem.word() end,
    naive_datetime: fn -> Faker.Date.backward(Enum.random(0..100))
    end,
    ...
  }

def invalid_params(fields_with_types) do
  invalid_value_by_type = %{
    string: fn -> DateTime.utc_now() end,
    naive_datetime: fn -> Faker.Lorem.word() end,
    ...
  }
```

Listing 4.10: Definicije pomoćnih funkcija *valid_params* i *invalid_params*

Testiranje funkcija changeset

Funkcija *changeset/2* iz primera koda 4.6, koja kao argumente prihvata strukturu *Assignment* i listu atributa, ima ulogu da validira prisustvo dva polja u tabeli *assignments* — polja *comment* i *grade*. Test primer koji proverava uspešnu putanju izvršavanja funkcije *changeset/2* prikazan je u primeru koda 4.11. Funkciji se prosleđuju validni parametri, kreirani pomoću prethodno definisane funkcije *valid_params*. Nakon toga, proverava se da li je dobijeni skup promena validan, a onda se pojedinačno za svako neophodno polje proverava da li je ispravno.

```
test "success: returns a valid changeset" do
  assert %Changeset{valid?: true, changes: changes} =
     Assignment.changeset(%Assignment{}, valid_params)

  for {field, _} <- @required_fields do
    actual = Map.get(changes, field)
    expected = valid_params[Atom.to_string(field)]
    assert actual == expected,
"Values did not match for: #{field}\nexpected: #{inspect(
   expected)}\nactual: #{inspect(actual)}"
  end
...
```

Listing 4.11: Test primer uspešne upotrebe funkcije *changeset/2*

Drugi test primer koji je potrebno pokriti je slučaj kada dolazi do greške zbog prosleđenih parametara koji nisu ispravni. Funkciji *changeset* se proslede parametri formirani pomoću funkcije *invalid_params*, i očekuje se da će dobijeni skup promena biti nevalidan. Nakon te provere, proverava se lista grešaka, koja bi trebalo da sadrži svako od neophodnih polja. Pošto su prosleđeni parametri pogrešnog tipa, koji se ne može kastovati u odgovarajući ispravni tip, očekuje se da u okviru greške vrsta validacije bude `:cast`, pa se i to na kraju proverava još jednom naredbom `assert`. Ovaj slučaj prikazan je u primeru koda 4.12.

```
test "error: returns an invalid changeset" do
  assert %Changeset{valid?: false, errors: errors} =
     Assignment.changeset(%Assignment{}, invalid_params)

  for {field, _} <- @required_fields do
    assert errors[field], "#{field} is missing from errors."

    {_, meta} = errors[field]
    assert meta[:validation] == :cast,
      "The validation type #{meta[:validation]} is incorrect."
  end
end
```

Listing 4.12: Test primer neuspešne upotrebe funkcije *changeset/2*, prosleđivanjem neodgovarajućih parametara

Ako se funkciji *changeset* prosledi prazna mapa, tj. ako nedostaju polja koja inače moraju biti navedena, javlja se greška čiji je tip validacije `:required`. Primer ovog slučaja dat je u kodu 4.13. U ovom slučaju, nakon provere da li je skup promena neispravan, proverava se da li je svako od zahtevanih polja u listi grešaka, a nakon toga i da li je tip validacije `:required`. Na kraju, pomoću naredbe `refute`, utvrđuje se da se opciona polja ne nalaze u listi grešaka. To su polja koja nije neophodno navesti pri pozivanju ove funkcije, i ne bi trebalo da se nađu u listi grešaka.

```
test "error: returns an error changeset (required)" do
  assert %Changeset{valid?: false, errors: errors} =
     Assignment.changeset(%Assignment{}, %{})

  for {field, _} <- @required_fields do
     assert errors[field], "#{field} is missing from errors."
     {_, meta} = errors[field]
     assert meta[:validation] == :required,
        "The validation type #{meta[:validation]} is incorrect."
  end

  for field <- @optional_fields do
   refute errors[field],
    "The optional field #{field} is not required."
  end
end
```

Listing 4.13: Test primer neuspešne upotrebe funkcije *changeset/2*, sa nedostajućim parametrima

Neke od shema će unutar svoje funkcije *changeset* imati i dodatne validacije, kao što je na primer validacija jedinstvenih polja. Na primeru sheme *users*, nakon ostalih izmena i validacija, dodat je i sledeći poziv funkcije: `unique_constraint(:email)`. Ova funkcija obaveštava *Ecto* da u tabeli korisnika ne sme postojati dva korisnika sa istom imejl adresom, tj. polje `:email` mora biti jedinstveno za svakog korisnika. Ako se desi pokušaj registrovanja korisnika sa već iskorišćenom imejl adresom, dolazi do greške tipa `:unique`. Ovaj test primer prikazan je u primeru koda 4.14. Neophodno je ostvariti direktan pristup bazi podataka, pa je prva linija unutar test primera naredba kojom se kreira konekcija sa bazom. Zatim se u bazu ubacuje novi korisnik (pozivom `MsnrApi.Repo.insert()`), i time je završena priprema testa.

Nakon toga, pokušava se unos još jednog korisnika sa istom imejl adresom. To bi trebalo da izazove grešku, što se proverava prvom naredbom `assert`. Druga naredba `assert` proverava da li se greška odnosi na polje `:email`, a treća utvrđuje i tačnu vrstu greške, slično kao u prethodnim primerima. Za razliku od prethodnih testova, meta podaci u ovom slučaju nisu validacija, već ograničenje (eng. `constraint`).

```
test "error: returns an error when an email is reused" do
  Ecto.Adapters.SQL.Sandbox.checkout(MsnrApi.Repo)

  {:ok, existing_user} =
    %User{}
    |> User.changeset(valid_params(@required_fields))
    |> MsnrApi.Repo.insert()

  changeset_with_reused_email =
    %User{}
    |> User.changeset(valid_params(@required_fields)
    |> Map.put("email", existing_user.email))

  assert {:error, _}} =
    MsnrApi.Repo.insert(changeset_with_reused_email)

  assert errors[:email]
  {_, meta} = errors[:email]

  assert meta[:constraint] == :unique,
    "The constraint type #{meta[:constraint]} is incorrect."
end
```

Listing 4.14: Test primer neuspešne upotrebe funkcije *changeset/2*, pri narušavanju ograničenja jedinstvenosti

Testiranje upita

U ovom delu biće prikazano kako su testirani konteksti entiteta. Modul koji će služiti kao primer se odnosi na korisnike portala — *MsnrApi.Accounts*. Struktura jednog dela ove datoteke prikazana je u primeru koda 4.15, gde se može videti upotreba funkcija za interakciju sa bazom podataka kroz modul *Ecto.Repo*. Prikazane

su osnovne operacije dohvatanja redova iz tabele, dodavanja novog reda, ažuriranja reda, i brisanja reda iz zadate tabele. Unutar funkcija *create_user* i *update_user* poziva se i funkcija *User.changeset*, koja je već prethodno istestirana.

```
defmodule MsnrApi.Accounts do
 def list_users, do: Repo.all(User)

 def get_user!(id), do: Repo.get!(User, id)

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

 def update_user(%User{} = user, attrs) do
    user
    |> User.changeset(attrs)
    |> Repo.update()
 end

 def delete_user(%User{} = user), do Repo.delete(user)
```

Listing 4.15: Definicija konteksta *MsnrApi.Accounts*

Fabrike za pripremu podataka

Prilikom pisanja testova koji pristupaju tabelama baze podataka, za dohvatanje podataka u fazi pripreme, pogodno je iskoristiti obrazac fabrike (eng. *factory pattern*) [28]. Fabrike u testiranju jesu funkcije koje generišu podatke. Kako aplikacija raste, održavanje testova postaje zahtevnije, i u tome značajno pomaže imati jedan izvor za pripremu podataka. U slučaju testiranja interakcija sa bazom podataka, kreirana je zajednička datoteka *msnr_api/test/support/factory.ex*. Pored toga, kreiran je poseban direktorijum *msnr_api/test/support/factories* u kome će se nalaziti pojedinačne fabrike za svaki od entiteta. U osnovi ovih fabrika nalazi se biblioteka *ExMachina* [9]. Ova biblioteka obezbeđuje kreiranje kompleksnih struktura podataka za sheme, kao i mehanizam za ubacivanje podataka u bazu bez pisanja koda. Kao prvi korak, *ExMachina* je dodata kao zavisnost aplikacije: u

okviru datoteke *msnr_api/mix.exs* ubačena je linija `{:exmachina, "~> 2.7.0", only: :test}`. Da bi mogla da se koristi, pokreće se komanda `mix deps.get` iz komandne linije. U tabeli 4.2 prikazane su neke od funkcija unutar *ExMachina.Ecto* modula, koje se koriste pri ubacivanju podataka prilikom testiranja [12].

Tabela 4.2: Funkcije modula *ExMachina.Ecto*

Funkcija	**Opis funkcije**
insert(factory_name)	Kreira novu fabriku i ubacuje je u bazu podataka
insert_list(number_of_records, factory_name)	Kreira više fabrika i ubacuje ih u bazu podataka
params_for(factory_name)	Kreira novu fabriku i vraća njena polja
params_with_assoc(factory_name)	Kreira novu fabriku i vraća njena polja, i dodatno ubacuje sve relacije pripadanja drugim tabelama, kao i strane ključeve
string_params_for(factory_name)	Kreira novu fabriku i vraća njena polja, u vidu mape čiji su ključevi niske, a ne atomi

Datoteka *factory.ex* prikazana je u primeru koda 4.16. Prva linija uključuje biblioteku *ExMachina* i prosleđuje joj naziv repozitorijuma aplikacije, što znači da će ova fabrika moći da se koristi specifično u tom repozitorijumu. Ostatak datoteke su uključivanja pojedinačnih fabrika za svaki kontekst aplikacije. U okviru testova za te kontekste, importovaće se samo ova *factory.ex* datoteka.

```
defmodule MsnrApi.Support.Factory do
  use ExMachina.Ecto, repo: MsnrApi.Repo
  use MsnrApi.UserFactory
  use MsnrApi.ActivityFactory
  use MsnrApi.StudentFactory
  ...
```

Listing 4.16: Definicija modula *Factory*

Definicija funkcije fabrike data je u primeru koda 4.17, u kome je prikazana definicija fabrike za korisnika. Po konvenciji biblioteke, pri imenovanju ovih funkcija neophodno je navesti ime sheme, i zatim *"_factory"*. Povratna vrednost funkcije je struktura sheme sa popunjenim lažnim vrednostima, dobijenim iz biblioteke *Faker*.

```
defmacro __using__(_opts) do
  quote do
    def user_factory do
      %User {
        email: Faker.Internet.email(),
        first_name: Faker.Person.first_name(),
        last_name: Faker.Person.last_name(),
        ... }
    end
```

Listing 4.17: Definicija modula *UserFactory*

U direktorijumu za pomoćne datoteke *msnr_test/support* kreiran je još jedan modul — *DataCase*. Ovaj modul služiće za sve situacije u kojima je potrebno baratati podacima pri interakciji sa bazom. Modul je prikazan u primeru koda 4.18. U njemu se mogu nalaziti pomoćne funkcije koje će se koristiti u testovima, slično kao kod modula *SchemaCase* koji je korišćen pri testiranju samih shema. U okviru datoteke, obezbeđena je zajednička priprema *Sandbox* konekcija, i uključeni su aliasi za fabrike i za repozitorijum, koji će biti potrebni pri testiranju upita. Ako je neka funkcija fokusirana na kreiranje podataka, dobra je praksa smestiti je u fabriku. U suprotnom, pripadaće nekom ovakvom obrascu slučaja (eng. *case template*), kao što je *DataCase*.

```
defmodule MsnrApi.Support.DataCase do
  use ExUnit.CaseTemplate
  using do
    quote do
      alias MsnrApi.{Support.Factory, Repo}
      ...

  setup _ do
    Ecto.Adapters.SQL.Sandbox.mode(MsnrApi.Repo, :manual) end
```

Listing 4.18: Definicija modula *DataCase*

Testiranje osnovnih CRUD operacija

Datoteka *MsnrApi.Accounts* sadrži osnovne operacije kreiranja, čitanja, ažuriranja i brisanja (eng. *Create, Read/get, Update, Delete — CRUD*) iz tabele. Testiranje ovih jednostavnih funkcija biće prikazano na primeru datoteke *MsnrApi.Queries.AccountsTest*. U primeru koda 4.19 prikazani su testovi koji proveravaju ispravnost funkcije *create_user/1*. Prva linija unutar test primera uspešne putanje koristi funkciju fabrike. Funkcija *string_params_for* uzima atom *:user* i sama poziva funkciju *user_factory*. *ExMachina* obezbeđuje da povratna vrednost ove funkcije bude mapa sa ključevima koji su niske i predstavljaju parametre, koji se zatim prosleđuju funkciji *create_user* u fazi delovanja. S obzirom na to da je pozivom te funkcije izvršen upis u bazu, u fazi provere neophodno je izvršiti čitanje iz baze. Test direktno poziva *MsnrApi.Repo*, a ne koristi kôd iz same aplikacije. Nije poželjno da test zavisi od koda aplikacije, jer ako dođe do neke izmene koja može narušiti trenutnu funkcionalnost, mnogo testova ne bi više prolazilo, a bilo bi teško zaključiti zbog čega.

Pored povratne vrednosti funkcije, koja je u ovom slučaju korisnik koji je ubačen u bazu, u ovim testovima treba voditi računa i o propratnim efektima. Propratni efekat je to da su dodati novi podaci u bazu podataka. Pored toga što proverava povratnu vrednost funkcije (da li je vraćen novokreirani korisnik), test nakon toga dohvata i konkretne podatke iz baze i poredi da li je vraćeni korisnik jednak onome koji je direktno izvučen iz baze. Zatim, važno je proći kroz sve parametre i proveriti da li su oni sada prisutni u bazi podataka. Na samom kraju, vrši se još jedna provera kako bi test bio što temeljniji — porede se vremenske oznake kreiranja i ažuriranja, koje treba da budu identične nakon kreiranja.

S obzirom na to da su testovi skupova promena (funkcije *changeset*) iz prethodnog dela pokrili sve slučajeve greške do kojih može doći, na ovom mestu je dovoljan samo jedan test primer koji proverava neuspešnu putanju. Funkciji *create_user* se umesto potrebnih parametara prosleđuje prazna mapa kao ulaz. Sve što ovaj test treba da utvrdi jeste prisustvo greške, i da proveri da li je povratna vrednost ispravnog oblika.

```
test "success: it inserts user in the db" do

  params = Factory.string_params_for(:user)
  assert {:ok, returned_user} = Accounts.create_user(params)
  user_from_db = Repo.get(User, returned_user.id)
  assert returned_user == user_from_db

  for {field, expected} <- params do
    schema_field = String.to_existing_atom(field)
    actual = Map.get(user_from_db, schema_field)
    assert actual == expected
  end

  assert user_from_db.inserted_at == user_from_db.updated_at
end

test "error: returns an error when user can't be created" do

   missing_params = %{}
   assert {:error, _} = Accounts.create_user(missing_params)
end
```

Listing 4.19: Testiranje funkcije *create_user/1*

Naredna testirana operacija je čitanje podataka iz baze. U modulu *Accounts* tu operaciju izvršava funkcija *get_user/1*, tako što dohvata jedan red iz tabele na osnovu jedinstvenog identifikatora korisnika. Dva test primera koja proveravaju ovu funkciju prikazana su u primeru koda 4.20. Za uspešan scenario, na početku testa se ubacuje jedan korisnik u bazu pomoću fabrike, kako bi nakon toga mogao biti dohvaćen. Prilikom navođenja naredbe `assert`, funkciji *get_user* prosleđuje se identifikator prethodno dodatog korisnika. Nakon toga, još jednom naredbom `assert` utvrđuje se da li je dohvaćeni korisnik jednak onom koji je ubačen u bazu podataka u fazi pripreme testa.

Neuspešan scenario podrazumeva pokušaj dohvatanja korisnika sa nepostojećim identifikatorom -1, nakon čega se očekuje greška tipa *Ecto.NoResultsError*.

```
test "success: it returns a user when given a valid id" do
    existing_user = Factory.insert(:user)
    assert returned_user = Accounts.get_user!(existing_user.id)
    assert returned_user == existing_user
end

test "error: it returns an error when a user doesn't exist" do
    invalid_id = -1
    assert_raise Ecto.NoResultsError, fn ->
        Accounts.get_user!(invalid_id) end
end
```

Listing 4.20: Testiranje funkcije *get_user/1*

Funkcija *list_users/0* jednostavno poziva funkciju *all* iz *Repo* modula, i time dohvata sve redove zadate tabele. Testovi funkcije *list_users/0* dati su u primeru koda 4.21. Slično kao u prethodnom primeru, korisnici se ubace u bazu podataka, a zatim se dohvataju. Očekivana povratna vrednost je lista kreiranih korisnika. U drugom test primeru, najpre se izbrišu svi redovi tabele *users*, a zatim proveri da li će funkcija vratiti praznu listu.

```
test "success: returns a list of all users" do
    existing_users = [
        Factory.insert(:user),
        Factory.insert(:user),
        Factory.insert(:user)
    ]

    assert existing_users = Accounts.list_users()
end

test "success: returns an empty list when no users" do
  {:ok, _} = Ecto.Adapters.SQL.query(MsnrApi.Repo,
                                     "DELETE FROM users")

  assert [] == Accounts.list_users()
end
```

Listing 4.21: Testiranje funkcije *list_users/0*

Ažuriranje redova tabele vrši se pomoću funkcije *update_user/2*, koja kao ulazne parametre prima jednog korisnika i listu atributa koji se ažuriraju. Najpre se poziva funkcija *User.changeset*, pa zatim i *Repo.update*. Testovi su prikazani u primeru koda 4.22. Slučaj uspešne putanje počinje unosom novog korisnika u bazu pozivanjem fabrike. Nakon toga, kreira se mapa parametara, a iz nje zatim dohvata jedan par ključ/vrednost. Uzima se samo podatak o prezimenu korisnika. Kada bi se proveravalo ažuriranje svakog dozvoljenog polja, povećala bi se šansa da test vremenom postane zastareo. Provera dozvoljenih polja za izmenu je već odrađena u delu o testiranju skupa promena. Test treba da utvrdi da su sva polja osim jednog ostala nepromenjena. To se postiže formiranjem mape sa očekivanim ključevima i vrednostima, a zatim poređenjem sa vrednostima dohvaćenim iz baze. Izostavljaju se dva polja koje ne treba proveravati.

Slučaj greške podrazumeva dodavanje novog korisnika u tabelu, a zatim pokušaj ažuriranja tog korisnika prosleđivanjem nevalidnih parametara. Polje se postavi da ima tip koji opisuje datum, umesto očekivanog tipa niske. Dodatnu sigurnost obezbeđuje poslednja linija testa koja utvrđuje da se ništa nije zapravo promenilo u bazi.

```
test "success: it updates database and returns the user" do
  assert {:ok, user} = Accounts.update_user(existing_user,
                                    %{''last_name'', ''new''})
  assert user_from_db == Repo.get(User, user.id)
  expected_user_data = existing_user
            ...
          |> Map.put(:last_name, ''new'')

  for {field, expected} <- expected_user_data do
    assert Map.get(user_from_db, field) == expected end  ...

test "error: returns an error when user can't be updated" do
  existing = Factory.insert(:user)
  assert {:error, _} = Accounts.update_user(existing,
                %{"last_name" => DateTime.utc_now()})
  assert existing == Repo.get(User, existing.id)
end
```

Listing 4.22: Testiranje funkcije *update_user/2*

Poslednja CRUD operacija se odnosi na brisanje određenog reda iz tabele. Funkcija u modulu *Accounts* koja ovo obezbeđuje je *delete_user/1*. Ona ima jednu liniju u kojoj poziva *Repo.delete*. Slučaj greške je skoro nemoguć, a pritom zahteva pisanje velikog broja komplikovanih linija koda u fazi pripreme testa. U ovom slučaju se to ne isplati, pa se test primer neuspešne putanje izostavlja. Napisan je samo jedan test primer koji predstavlja uspešnu putanju izvršavanja, prikazan u primeru koda 4.23. Kao i svi ostali, test počinje unosom novog korisnika. Zatim poziva funkciju *delete_user*, nakon čega se očekuje kao povratna vrednost par koji sadrži atom `:ok` i obrisanog korisnika. Povratna vrednost nakon toga nije ni potrebna, jer taj korisnik više ne postoji. Poslednja linija upotrebom makroa `refute` utvrđuje da korisnika više nema u bazi podataka.

```
test "success: it deletes the user" do
  user = Factory.insert(:user)
  assert {:ok, _deleted_user} = Accounts.delete_user(user)
  refute Repo.get(User, user.id)
end
```

Listing 4.23: Testiranje funkcije *delete_user/1*

4.3 Testiranje upravljača i pogleda

Phoenix upravljači (eng. *controllers*) su interno utikači, što znači da prihvataju strukturu *Plug.Conn* kao ulaznu vrednost i vraćaju je kao povratnu. Upravljači predstavljaju posredničke module [14]. Funkcije unutar upravljača nazivaju se akcije. One se pokreću unutar modula *MnsrApiWeb.Router* kao odgovori na HTTP zahteve. Naziv akcije u upravljaču mora se poklapati se onim navedenim u putanji definisanoj u modulu *Router*. Njihova uloga je da prikupe sve neophodne podatke i izvrše odgovarajuće operacije pre nego što pozovu funkciju *render*, definisanu u sloju pogleda (eng. *view layer*). Ova funkcija prihvata *Plug.Conn* strukturu koja sadrži sve podatke o HTTP zahtevu, naziv šablona i podatke potrebne za generisanje mape, koja se prevodi u JSON objekat i vraća kao povratna vrednost. Struktura upravljača i pogleda na primeru entiteta koji predstavlja semestar prikazana je u primerima koda 4.24 i 4.25. Data je samo akcija za unos novog semestra. Implementacija je analogna i za sve ostale entitete u okviru *Portala MSNR*.

```
defmodule MsnrApiWeb.SemesterController do
  use MsnrApiWeb, :controller
  action_fallback MsnrApiWeb.FallbackController

   def create(conn, %{"semester" => semester_params}) do
      with {:ok, semester} <- Semesters.create(params) do
        conn
        |> put_status(:created)
        |> put_resp_header("location",
                Routes.semester_path(conn, :show, semester))
        |> render("show.json", semester: semester)
   end
```

Listing 4.24: Struktura upravljača *SemesterController*

```
defmodule MsnrApiWeb.SemesterView do
  use MsnrApiWeb, :view

  def render("show.json", %{semester: semester}) do
    %{data: render_one(semester, SemesterView, "semester.json")}
  end

  def render("semester.json", %{semester: semester}) do
    %{
      id: semester.id,
      year: semester.year,
      is_active: semester.is_active
    }
  end
```

Listing 4.25: Struktura pogleda *SemesterView*

Akcija *create* prihvata parametre za novi semestar i čuva ga u bazi, a zatim kreira JSON za taj novonastali semestar. Prvo vrši proveru da li semestar može biti kreiran, i ako može postavlja status na `:created` — 201. Nakon toga, postavlja zaglavlje `location` na lokaciju tog semestra, a zatim kreira `"show.json"` sa informacijama o novom semestru.

Specijalna naredba `with` eksplicitno proverava uspešno izvršavanje. Ako funkcija *Semesters.create(params)* iz nekog razloga ne uspe, vratiće grešku u obliku skupa promena — `{:error, changeset}`. Akcije upravljača ne znaju kako da obrade

ovakav tip greške. U tome im pomaže upravljač *Action Fallback*. Modul *FallbackController* aktivira se svaki put kada neka akcija upravljača ne vrati strukturu *Plug.Conn*. On definiše različite funkcije *call* koje prevode rezultate izvršavanja akcija u validne *Plug.Conn* odgovore. Implementacija ovog upravljača data je u primeru koda 4.26.

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

  def call(conn, {:error, %Ecto.Changeset{} = changeset}) do
    conn
    |> put_status(:unprocessable_entity)
    |> put_view(MsnrApiWeb.ChangesetView)
    |> render("error.json", changeset: changeset)
  end
...
```

Listing 4.26: Struktura upravljača *FallbackController*

U ovom primeru prikazana je funkcija *call* koja obrađuje grešku tipa *{:error, changeset}*. Ova funkcija se poziva kada se desi greška pri pozivu *Ecto* operacija `insert, update` ili `delete`. Postavlja status na 422 (*unprocessable entity*) i kreira `"error.json"` iz pogleda skupa promena *MsnrApiWeb.ChangesetView*, u kome prikazuje neuspešan skup promena.

Pored akcije *create*, upravljači definišu i druge, kojima se po konvenciji dodeljuju sledeća imena:

- *index* — generiše listu svih objekata datog tipa (u ovom slučaju semestar)
- *show* — kreira jedan JSON objekat preko identifikatora
- *update* — prihvata parametre za ažuriranje objekta i čuva ga u bazi
- *delete* — prihvata identifikator za dati objekat i briše ga iz baze

Svaka od ovih akcija kao prvi argument uzima strukturu *Plug.Conn*, koja sadrži informacije o korisničkom zahtevu i dolazi iz okruženja *Elixir Plug*. Drugi argument je mapa *params*, koja sadrži parametre koji se prosleđuju unutar HTTP zahteva.

Da bi funkcija *render* iz primera koda 4.25 ispravno radila, upravljač i pogled moraju da imaju isti naziv (u ovom slučaju *Semester*). Zadatak pogleda je da iz-

generiše podatke u određenom formatu. U slučaju ove aplikacije, korišćen je JSON veb interfejs, i svi pogledi generišu JSON sadržaj.

Priprema za testiranje

Pozivanjem komande `mix phx.gen.json` (koju pruža razvojno okruženje *Phoenix*), generišu se sve datoteke upravljača i pogleda. Parametri koji se navode uz ovu komandu su naziv konteksta, naziv strukture i naziv tabele u bazi, nakon čega slede definicije polja, odnosno kolona. Pored ovih datoteka, *Phoenix* automatski generiše i datoteke koje predstavljaju šablone za testove unutar *test\msnr_api_web* direktorijuma. Takođe, u *test\support* direktorijumu, nalaziće se i datoteka koja definiše modul *ConnCase*. Ovo je još jedan obrazac slučaja, koji se uključuje na početku svakog od testova upravljača, linijom `use MsnrApiWeb.ConnCase`. Modul *ConnCase* prikazan je u primeru koda 4.27.

```
defmodule MsnrApiWeb.ConnCase do
  use ExUnit.CaseTemplate
  alias MsnrApi.Support.DataCase

  using do
    quote do
      import Plug.Conn
      import Phoenix.ConnTest
      import MsnrApiWeb.ConnCase
      alias MsnrApiWeb.Router.Helpers, as: Routes
      @endpoint MsnrApiWeb.Endpoint
    end
  end

  setup tags do
    pid = Ecto.Adapters.SQL.Sandbox.start_owner!(MsnrApi.Repo,
   shared: not tags[:async])
    on_exit(fn -> Ecto.Adapters.SQL.Sandbox.stop_owner(pid) end)
    {:ok, conn: Phoenix.ConnTest.build_conn()}
  end
end
```

Listing 4.27: Modul *ConnCase*

Uključuju se neophodni moduli koji će se koristiti u testovima — *Plug.Conn* i *Phoenix.ConnTest*. Ova dva modula sadrže funkcije koje pomažu u testiranju krajnjih tačaka (eng. *endpoints*) i konekcija. Utikač *MsnrApiWeb.Endpoint* predstavlja početnu tačku prilikom obrade HTTP zahteva i sastoji se od niza utikača, kroz koje prolazi svaki zahtev. *Phoenix.ConnTest* obezbeđuje funkcije koje se koriste za različite operacije nad konekcijom koje je potrebno sprovesti pre nego što se ona isporuči krajnjoj tački. Dalje, u datoteci *ConnCase* navodi se utikač *MsnrApiWeb.Endpoint* uz atribut `@endpoint`, da se naznači da će to biti krajnja tačka koja se testira. Takođe, omogućava se upotreba putanja u testovima iz modula *MsnrApiWeb.Router*. Na kraju, definiše se blok pripreme koji će biti pozvan pre svakog testa. Tu se nalaze *SQL Sandbox* podešavanja, a na samom kraju se poziva funkcija *build_conn()* iz modula *Phoenix.ConnTest*. Ona vraća novu konekciju koja će biti dostupna u svakom od testova. Još korisnih funkcija za upravljanje konekcijama pri testiranju biće objašnjeno kroz opise konkretnih testova u nastavku.

Testovi akcija upravljača

Test koji proverava uspešnu putanju izvršavanja akcije *create* upravljača *SemesterController* prikazan je u primeru koda 4.28. Koristi funkciju *post* da kreira novi semestar. Ova funkcija dolazi iz modula *Phoenix.ConnTest*, i kao argumente prihvata strukturu *Plug.Conn*, putanju koju dohvata iz modula *MsnrApiWeb.Router* i preko koje poziva akciju *create*, i mapu polja neophodnih za kreiranje semestra. Zatim verifikuje da je vraćen JSON odgovor sa statusom 201 i da poseduje ključ `"data"` u sebi. Tu proveru obezbeđuje funkcija *json_response*. Izvršava se zahtev *get* nad putanjom `:show` i utvrđuje se da je semestar uspešno kreiran. Na kraju se naredbom `assert` upoređuju očekivana polja sa poljima iz JSON odgovora.

```
conn = post(conn,
       Routes.semester_path(conn, :create), semester: @attrs)
assert %{"id" => id} = json_response(conn, 201)["data"]
conn = get(conn, Routes.semester_path(conn, :show, id))
assert %{ "id" => ^id,
          "is_active" => true,
          "year" => 2023
        } == json_response(conn, 200)["data"]
```

Listing 4.28: Testiranje akcije *create* upravljača *SemesterController*

Drugi test unutar bloka `describe` proverava neuspešan scenario akcije za kreiranje. Prikazan je u primeru koda 4.29. Poziva se *post*, ali ovaj put prosleđivanjem neodgovarajućih parametara. Tada se aktivira upravljač *FallbackController* koji će generisati odgovarajući JSON odgovor. Očekuje se da taj odgovor ima status 422, i da mapa `"errors"` ne bude prazna.

```
test "renders errors when data is invalid", %{conn: conn} do
  conn = post(conn, Routes.semester_path(conn, :create),
                                semester: @invalid_attrs)
  assert json_response(conn, 422)["errors"] != %{}
end
```

Listing 4.29: Testiranje akcije *create* upravljača *SemesterController*

Naredne dve akcije koje se testiraju su akcije *index* i *show*. One imaju jednostavne uloge — da prikažu listu svih semestara (*index*) ili jednog konkretnog semestra na osnovu identifikatora (*show*). Testovi koji proveravaju ove funkcije pozivaju funkciju *get* iz modula *Phoenix.ConnTest* i verifikuju ispravnost JSON odgovora, slično kao u prethodnom primeru. Implementirani su unutar datoteke *msnr_api\test\msnr_api_web\controllers\semester_controller_test.exs*.

Testovi koji pokrivaju akcije *update* i *delete* prikazani su u primeru koda 4.30. U oba slučaja potrebno je prvo kreirati jedan semestar na kome bi mogle da se primene ove akcije upravljača. Kreiranje semestra obezbeđuje privatna funkcija koja koristi modul *MsnrApi.SemestersFixtures*, u kome se poziva funkcija iz konteksta *Semesters* koja kreira semestar u bazi, istestirana u prethodnoj sekciji.

Akcija ažuriranja se jednostavno proverava, pozivanjem funkcije *put*, i nakon toga verifikacijom ažuriranih podataka. Blok `describe` sadrži i test koji pokriva scenario u kom dolazi do greške i očekuje se kôd 422. Akcija brisanja semestra treba da izazove prazan odgovor (koji ovoga puta nije JSON) sa kodom 204 — *no_content*. Takođe, nakon što je semestar uspešno obrisan, očekuje se da on ne može biti dohvaćen i funkcija *get* izaziva grešku tipa 404 — *not_found*.

Testovi upravljača za neke od osnovnih entiteta nalaze se u direktorijumu *msnr_api\test\msnr_api_web\controllers*. Ostali upravljači testiraju se na sličan način, pa je za proveru njihove ispravnosti potrebno napisati testove analogne onima koji su opisani u ovoj sekciji.

```
describe "update semester" do
  setup [:create_semester]
  test "renders semester", %{conn: conn,
            semester: %Semester{id: id} = semester} do
    conn = put(conn, Routes.semester_path(conn,
                    :update, semester), semester: @attrs)
    assert %{"id" => ^id} = json_response(conn, 200)["data"]

    conn = get(conn, Routes.semester_path(conn, :show, id))
    assert %{
      "id" => ^id,
      "is_active" => false,
      "year" => 2024
    } = json_response(conn, 200)["data"]
  end

  test "renders errors when data is invalid", %{conn: conn,
                              semester: semester} do
    conn = put(conn, Routes.semester_path(conn, :update,
                             semester), semester: @invalid_attrs)
    assert json_response(conn, 422)["errors"] != %{}
  end
end

describe "delete semester" do
  setup [:create_semester]
  test "deletes chosen semester", %{conn: conn,
                           semester: semester} do
    conn = delete(conn, Routes.semester_path(conn,
                           :delete, semester))
    assert response(conn, 204)

    assert_error_sent 404, fn ->
      get(conn, Routes.semester_path(conn, :show, semester)) end
  end
end
```

Listing 4.30: Testiranje akcija *update* i *delete* upravljača *SemesterController*

Glava 5

Testiranje klijentskog dela aplikacije

Klijentski deo portala implementiran je u programskom jeziku *Elm*. *Elm* je statički tipiziran i čist funkcionalni jezik. Odsustvo propratnih efekata u funkcijama omogućava da dokazivanje njihove ispravnosti bude značajno jednostavnije u odnosu na nečiste funkcije iz prethodnog poglavlja. Testovi su predvidljivi i jednostavni za održavanje zahvaljujući imutabilnosti. Kako je već objašnjeno u delu 2.3, za *Elm* aplikacije važi da ne izbacuju neplanirane greške prilikom izvršavanja. U ovom poglavlju biće predstavljeni koncepti testiranja u ovom programskom jeziku na primeru korisničkog interfejsa aplikacije *Portal MSNR*. Testovi iz ovog poglavlja mogu se pronaći na adresi `https://github.com/Anakin2012/master-rad/tree/main/testing-msnr-portal/portal/msnr_elm/tests`.

5.1 Uvod u testiranje Elm aplikacija

Elm dolazi sa ugrađenim razvojnim okruženjem za testiranje pod nazivom *elm-test* [15]. Ovaj alat je distribuiran kao NPM paket (eng. *Node.js Package Manager*) [3]. Kako bi se instalirao *elm-test*, neophodno je iz terminala pozicionirati se u direktorijum aplikacije i pokrenuti narednu komandu:

```
PS C:\Users\panap\testing-msnr-portal\portal\msnr_elm> npm install elm-test -g
```

Nakon uspešne instalacije, potrebno je pokrenuti naredbu koja služi da pripremi projekat za pisanje testova:

```
PS C:\Users\panap\testing-msnr-portal\portal\msnr_elm> elm-test init
```

Naredba `elm-test init` kreira novi direktorijum pod nazivom *tests* i u njemu datoteku *Example.elm*, koja predstavlja šablon za pisanje prvog testa. Pored toga, instalira paket *elm-explorations/test*, koji služi za definisanje testova koji mogu da se pokreću iz komandne linije. Ažuriraće i datoteku *elm.json*, u kojoj će dodati ovaj paket u sekciji *test-dependencies*.

Struktura testa jedinice koda

Osnovni koncepti testova u jeziku *Elm* objašnjeni su na primeru testiranja jednostavne funkcije koja prevodi kalendarski mesec u nisku. Ta funkcija definisana je u modulu *Util* i prikazana je u primeru koda 5.1. Ona jednostavno prihvata vrednost tipa *Month* (iz modula *Time*), i na osnovu meseca vraća nisku od dve cifre koje predstavljaju taj mesec.

```
toTwoDigitMonth : Month -> String
toTwoDigitMonth month =
    case month of
        Jan ->
            "01"
        Feb ->
            "02"
        ...
```

Listing 5.1: Funkcija *toTwoDigitMonth*

Testovi za ovaj modul nalaziće se u datoteci *msnr_elm\tests\UtilTests.elm*, datoj u primeru koda 5.2. Na početku, pri definiciji modula, potrebno je izložiti (eng. *expose*) nazive grupa testova koji će se pokretati. Dalje, svaka od datoteka sa testovima importuje tri modula:

1. **Expect** — za specifikaciju očekivanog ponašanja koda
2. **Fuzz** — za pisanje faz testova
3. **Test** — za kreiranje i upravljanje testovima

```
module UtilTests exposing (toTwoDigitMonthTests)
import Expect exposing (Expectation)
import Fuzz exposing (Fuzzer, int, list, string)
import Test exposing (..)

toTwoDigitMonthTests =
    describe "ToDigitMonth"
        [test "output is 01 when input is Jan" <|
        \_ -> toTwoDigitMonth Jan
              |> Expect.equal "01",
         test "output is 02 when the input is Feb" <|
         \_  -> toTwoDigitMonth Feb
                |> Expect.equal "02",
        ...
```

Listing 5.2: Implementacija testova za funkciju *toTwoDigitMonth*

Unutar funkcije *toTwoDigitMonthTests* najpre se navođenjem bloka `describe` opisuje grupa svih testova koji će se odnositi na funkciju *toTwoDigitMonth*. Preporučuje se da se slični testovi uvek grupišu u zajedničke blokove `describe`, koji se mogu i ugnježdavati. Pri kreiranju testa poziva se funkcija *test* iz modula *Test*, koja ima dva argumenta: opis onoga što se testira i anonimnu funkciju koja sadrži sam test. Testovi ne koriste argumente date anonimnoj funkciji, i zato se taj argument ignoriše navođenjem _. Funkcija *Expect.equal* prihvata očekivanu vrednost i izraz sa kojim će uporediti tu vrednost, a vraća *True* ako su vrednosti jednake. Operator `|>` prosleđuje rezultat izraza sa leve strane kao poslednji argument funkcije sa desne strane. Prilikom pisanja testova preporučuje se upotreba ovog operatora, jer u suprotnom, kako se povećava broj operacija neophodan za pripremu i poziv funkcije koja se testira, testovi postaju sve nečitljiviji. Operatorom `<|` se izbegava upotreba zagrada.

Pored poređenja jednakosti, u modulu *Expect* postoje i druge funkcije koje mogu proveravati nejednakost (*Expect.notEqual*), ili na primer — da li je jedan izraz manji ili veći od drugog (*Expect.lessThan*, *Expect.greaterThan*). Više o ovom modulu i njegovim funkcijama može se pronaći na zvaničnoj stranici [18].

Naredbom `elm-test` pokreću se svi testovi, a da bi se pokrenuli samo testovi unutar ovog modula, potrebno je pozvati komandu `elm-test tests/UtilTests.elm` iz terminala. Kako broj testova bude rastao, preporučuje se pokretanje

samo određenog broja testova istovremeno. Funkcije *Test.skip* i *Test.only* su korisne za ovakvo pokretanje, a više o njima može se pronaći u dokumentaciji paketa *Test*. Nakon navođenja komande `elm-test`, trebalo bi da se dobije izlaz prikazan na slici 5.1. *Elm* će jasno ispisati ukupan broj testova koji se izvršava, i koliko njih prolazi, a koliko ne. Pored toga, ispisuje i jednu liniju koja se odnosi na faz testiranje, koja će biti objašnjena u delu o faz testovima. U slučaju da neki od testova ne prolaze, tj. ako se očekivana vrednost ne poklapa sa datim izrazom, `elm-test` će dati izlaz prikazan na slici 5.2.

```
Compiling > Starting tests

elm-test 0.19.1-revision12
--------------------------

Running 12 tests. To reproduce these results, run: elm-test --fuzz 100 --seed 392132510961291


TEST RUN PASSED

Duration: 206 ms
Passed:   12
Failed:   0
```

Slika 5.1: Izlaz nakon uspešnog izvršavanja testova

```
Compiling > Starting tests

elm-test 0.19.1-revision12
--------------------------

Running 12 tests. To reproduce these results, run: elm-test --fuzz 100 --seed 304772482267063

> UtilTests
> ToDigitMonth
> output is 02 when the input is Feb

    "02"
    |
    | Expect.equal
    |
    "03"


TEST RUN FAILED

Duration: 194 ms
Passed:   11
Failed:   1
```

Slika 5.2: Izlaz nakon neuspešnog izvršavanja testova

Što se tiče testiranja neispravnog ulaza, anotacija funkcije *toTwoDigitMonth* to ne dozvoljava. Funkcija očekuje kao ulaz vrednost tipa *Month*, koji dolazi iz modula *Time*, a kao izlaz vrednost tipa *String*. U ovakvim uslovima, u testu nije moguće

pozvati funkciju sa bilo kojim ulazom koji nije *Month*, jer će *Elm* kompilator protumačiti to kao grešku i dati odgovarajuće objašnjenje. Kako bi se proširili slučajevi ispravne upotrebe ovakve funkcije, ona se može izmeniti tako da povratna vrednost bude tipa *Maybe String*. Tada, pri prosleđivanju bilo čega što nije jedan od 12 mogućih ispravnih ulaza (po jedan za svaki mesec), funkcija bi vraćala *Nothing*.

U programskom jeziku *Elm* testovi iz primera 5.2 nazivaju se testovi jedinica koda i tako će biti nazivani u nastavku ovog poglavlja. Ovakvi testovi pišu se kada je potrebno proveriti specifičan scenario, kao što je neki granični slučaj. Testovi jedinica koda pozivaju kôd koji se testira samo jednom, sa odgovarajućim ulazom, i proveravaju da li je izlaz jednak očekivanom. Na primeru ove funkcije, postoji samo 12 mogućih ulaza, pa je zbog toga za testiranje ove funkcije napisano 12 testova jedinica koda.

Faz testiranje

Kada postoji jako veliki broj mogućih ulaza, bilo bi nemoguće ostvariti odgovarajuću pokrivenost samo testovima jedinica koda. Prednost testiranja u *Elm-u* je u tome što se mogu kombinovati različite vrste testova kako bi se postigla dobra pokrivenost koda. Pored testova jedinica koda, *Elm* nudi još jednu vrstu testova — faz testove. Faz testiranje je način testiranja u kome se isti test ponavlja iznova sa nasumično generisanim ulazima. Funkcije koje mogu imati veliki broj različitih ulaza predstavljaju dobre kandidate za faz testiranje. Unutar jednog bloka `describe`, mogu se kombinovati testovi jedinica koda i faz testovi.

Funkcija *intToMonth*, delimično prikazana u primeru koda 5.3 vrši jednostavno prevođenje celog broja koji predstavlja redni broj meseca u vrednosti tipa *Maybe Month*. U slučaju da je prosleđen broj između 1 i 12, njena povratna vrednost biće konkretan mesec, a za bilo koji drugi ulaz vratiće *Nothing*.

```
intToMonth : Int -> Maybe Month
intToMonth month =
    case month of
      1 -> Just Jan
        ...
      12 -> Just Dec
      _ -> Nothing
```

Listing 5.3: Implementacija funkcije *intToMonth*

Za konkretne ulaze koji će vratiti mesec, napisano je 12 testova jedinica koda. Što se tiče poslednjeg slučaja, kada se očekuje izlaz tipa *Nothing*, napisan je jedan test jedinice koda i dva faz testa. Prvi test pokrio je slučaj kada je ulaz 0, kako je to najverovatniji slučaj greške. Radi demonstracije upotrebe faz testiranja, u kodu 5.4, dat je jednostavan primer kako se faz test može sprovesti. Funkcija *fuzz* je slična funkciji *test*, ali prihvata i dodatni argument — fazer (eng. *fuzzer*). Uloga fazera je da generiše nasumične vrednosti datog tipa. Unutar modula *Fuzz* postoje fazeri za najčešće korišćene tipove podataka, kao što su *int, float, string, i list* [19]. Ako se koristi fazer za cele brojeve, on će u opštem slučaju generisati 100 vrednosti u intervalu [-50, 50]. U ovom primeru, iskorišćen je fazer *intRange*, kom se prosleđuje konkretan interval celih brojeva iz kojeg će se uzimati vrednosti. Napisana su dva faz testa, od kojih jedan proverava ulaze iz intervala [-50, -1], a drugi iz intervala [13, 50]. Iako ovi intervali nisu najverovatniji mogući unosi za ovu funkciju, ostavljeni su kako bi se prikazala mogućnost ovakvog testiranja. Još jedna razlika u odnosu na testove jedinica koda jeste što se anonimnoj funkciji prosleđuje pravi parametar (*month* u ovom primeru) koji se koristi u samom testu.

```
intToMonthTests =
 describe "intToMonth"
 [  ...
  fuzz (intRange -50 -1) "output is Nothing if input < 0" <|
     \month -> intToMonth month
               |> Expect.equal Nothing,
  fuzz (intRange 13 50) "output is Nothing if input > 12" <|
     \month -> intToMonth month
               |> Expect.equal Nothing
 ]
```

Listing 5.4: Implementacija faz testova za funkciju *intToMonth*

Pored fazera za osnovne tipove kao što su *int*, *string*, ili *bool*, modul *Fuzzer* omogućava i kreiranje specifičnih fazera za tipove koji su eksplicitno implementirani. Takvi fazeri se često generišu upotrebom kombinacije funkcija *Fuzz.map* i *Fuzz.oneOfValues*. Fazeri dolaze iz modula *Fuzzer*, ali funkcija *fuzz* potiče iz modula *Test* [21]. U tabeli 5.1 prikazane su tri često korišćene faz funkcije iz ovog modula i njihovi opisi.

Tabela 5.1: Funkcije modula *Test* za faz testiranje

Funkcija	**Opis funkcije**
fuzz2 : Fuzzer a -> Fuzzer b -> String -> (a -> b -> Expectation) -> Test	Slično kao *fuzz*, ali prihvata dva fazera i kreira dve nasumične vrednosti, za testiranje funkcija koje imaju dva argumenta.
fuzz3 : Fuzzer a -> Fuzzer b -> Fuzzer c -> String -> (a -> b -> c -> Expectation) -> Test	Slično kao *fuzz*, ali prihvata tri fazera i kreira tri nasumične vrednosti, za testiranje funkcija koje imaju tri argumenta.
fuzzWith : FuzzOptions a -> Fuzzer a -> String -> (a -> Expectation) -> Test	Kreira faz test sa datim opcijama, koje mogu biti broj faz testova (*runs*), ili statistička distrubucija testova (*distribution*).

Nakon pokretanja faz testova, na izlazu će se pojaviti seme nasumičnosti koje se može iskoristiti za rekreiranje konkretnih faz testova navođenjem komande `--seed` iz terminala, a može se i specifikovati konkretan broj faz testova koji će se izvršiti uz komandu `--fuzz`. Ako neki od testova izazove grešku, na izlazu će pisati koja od vrednosti je izazvala, a ako ih ima više, izabraće najjednostavniju od njih kako bi se što lakše pronašao uzrok. Navođenje većeg broja faz testova pokriva više ulaza, ali sa druge strane, time se povećava vreme izvršavanja.

Faz testiranje se smatra testiranjem zasnovanom na svojstvima (eng. *property based testing*). Njihova uloga je da utvrde da određeno svojstvo važi za sve ulaze i izlaze. U slučaju funkcije *intToMonth*, to svojstvo glasi: za svaki ulaz koji nije ceo broj između 1 i 12, izlaz uvek mora biti *Nothing*. Za razliku od testova jedinica koda koji proveravaju samo jedan konkretan scenario, faz testovi omogućavaju testiranje koda na mnogo višem nivou. Pri pisanju ovih testova, neophodno je dobro razmisliti šta je to što kôd treba da uradi i pronaći svojstvo koje mora biti zadovoljeno, a zatim u testu proveriti da li to svojstvo važi. Kad god je to moguće, uvek treba koristiti faz testove.

5.2 Testiranje rada sa JSON podacima

Kodiranje i dekodiranje JSON podataka izvršava se pomoću modula *Json.Encode* i *Json.Decode* iz paketa *elm/json*. Kada nakon HTTP zahteva server pošalje odgovor u JSON formatu, često je neophodno prevesti taj odgovor u odgovarajući slog. Funkcije koje vrše ovo prevođenje nazivaju se dekoderi. U primeru koda 5.5 data je funkcija dekodiranja iz modula *Session*, u kome su izdvojene funkcije za prijavljivanje i odjavljivanje korisnika. Funkcija *decodeSession* dekodira svako polje sloga *Session* i tako pruža validaciju podataka pre nego što oni dođu do aplikacije.

```
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))
```

Listing 5.5: Implementacija funkcije dekodiranja *decodeSession*

Unutar jednog dekodera mogu se pozivati i drugi dekoderi — u ovom primeru to su *decodeUser* i *decodeStudentInfo*, čija je uloga u dekodiranju slogova koji predstavljaju korisnika i studenta. Sistem tipova u *Elm-u* nije dovoljan u pronalaženju grešaka kada je u pitanju ispravnost dekodera. Pre pisanja samih faz testova, definisani su specifični fazeri, za svaki od slogova. Kako više datoteka sa testovima koristi iste fazere, svi oni su implementirani na jednom mestu — u pomoćnoj datoteci *tests\FuzzerHelper.elm*. Svaka datoteka sa testovima uključiće ovu datoteku zajedno sa konkretnim neophodnim fazerima. Za kreiranje nasumičnih kombinacija polja sloga *Session*, definisan je *sessionFuzzer*, prikazan u primeru koda 5.6. Funkcija *map5* će mapirati pet polja i za svako od njih kreirati odgovarajući fazer. Svaki od fazera kreiraće veliki broj različitih vrednosti. Na primer, *Fuzz.string* će generisati nasumične niske, ali sa većom verovatnoćom one za koje se često očekuju greške — prazne, jako dugačke ili kratke niske. Za slogove studenta i korisnika takođe su implementirani specifični fazeri.

```
sessionFuzzer : Fuzzer Session
sessionFuzzer =
    Fuzz.map5 Session
        Fuzz.string
        userInfoFuzzer
        ...
        (Fuzz.maybe studentInfoFuzzer)
```

Listing 5.6: Implementacija fazera za slog *Session*

Svaki od dekodera istestiran je pojedinačno, a faz testovi koji proveravaju ispravnost funkcije *decodeSession* prikazani su u primerima koda 5.7 i 5.8. Priprema testa podrazumeva enkodiranje svakog od polja, koje funkcioniše na sličan način kao i dekodiranje, samo u suprotnom smeru. Enkoderi se takođe mogu kreirati za specifične tipove podataka. Nakon što se objekat enkodira u JSON vrednost, poziva se funkcija *decodeValue* koja prihvata dekoder koji se testira kao prvi argument, i enkodirani objekat kao drugi. Ako je izvršavanje uspešno, očekuje se izlaz oblika `(Ok _)`, za čiju proveru je zadužena pomoćna funkcija *success*.

U drugom primeru, prikazano je kako se može izvući specifično polje iz rezultata, korišćenjem *Result.map*. Taj test proverava da li će polje *studentInfo* imati očekivanu vrednost *Nothing* ako se u ulazu enkodira `null` na njegovom mestu, s obzirom na to da mu je tip *Maybe*. Ovaj test odnosiće se samo na jedno konkretno polje, umesto na čitavu strukturu. To osigurava da neće biti potrebne manuelne ispravke ovog testa ako se u kodu dese neke izmene, kao na primer dodavanje novog polja. Zbog toga se preporučuje pisanje ovakvih testova koji se fokusiraju na jednu konkretnu stvar i tako sužavaju opseg testa. Takođe, ako se dese greške u nekim drugim poljima, ovaj test će i dalje prolaziti jer nije vezan za ta konkretna polja, i time će olakšati posao pronalaženja greške.

```
[fuzz sessionFuzzer "fuzz test for decoding session" <|
   \session ->
      [("access_token", Encode.string session.accessToken),
       ("user", Encode.float session.user), .... ]
       |> Encode.object
       |> Decode.decodeValue Session.decodeSession
       |> success
       |> Expect.equal True
```

Listing 5.7: Implementacija prvog faz testa za funkciju *decodeSession*

```
fuzz sessionFuzzer "student info field should be Nothing" <|
\session ->
    [ ... , ("student_info", Encode.null) ]
    |> Encode.object
    |> Decode.decodeValue Session.decodeSession
    |> Result.map (\s -> s.studentInfo)
    |> Expect.equal (Ok Nothing) ]
```

Listing 5.8: Implementacija drugog faz testa za funkciju *decodeSession*

U slučaju neispravnog ulaza, dekoderi će vratiti rezultat tipa `(Err, _)`. Kada se to desi, pomoćna funkcija *success* vratiće *False*. Primer testa koji proverava neispravan ulaz dat je u kodu 5.9, gde se testira dekoder sloga *Topic*. U testu se priprema ulaz koji je neispravan — polju *number* dodeli se niska umesto celog broja, zatim se prosledi funkciji i na kraju proveri da li je to dovelo do greške. *Expect.err* radi upravo to, a isto se može postići i upotrebom već spomenute funkcije *success*.

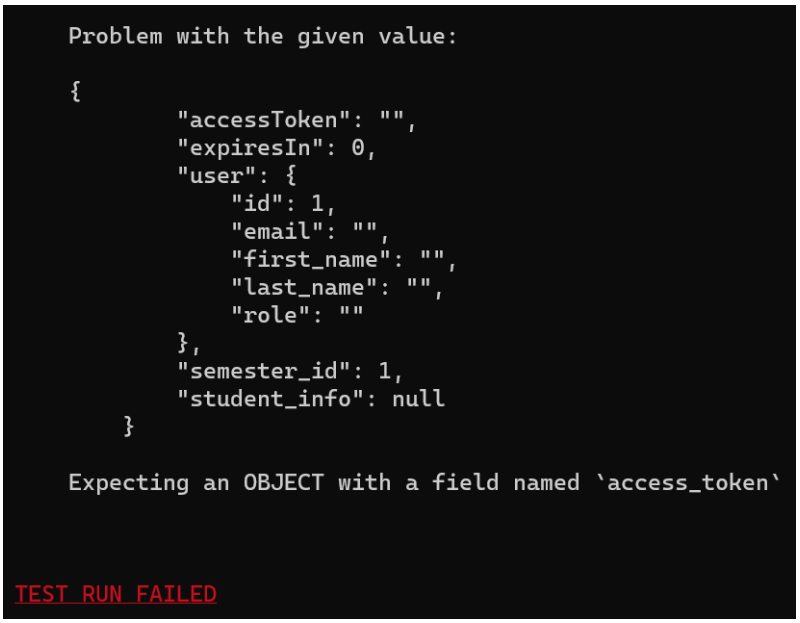
```
test "Given invalid input returns (Err _)" <|
  \_ -> let input = """
            { "id" : 1,
              "title" : "naslov",
              "number" : "1"}
            """
            decodedOutput = decodeString Topic.decoder input
        in
          Expect.err decodedOutput
```

Listing 5.9: Implementacija testa koji izaziva grešku

U modulu *Expect* postoji još jedna korisna funkcija, *Expect.fail*. Kada se desi da test sa *Expect.equal* ne prolazi, jedina informacija koja se dobije je da se očekivana i stvarna vrednost ne poklapaju. U testovima dekodera dodata je provera prikazana u kodu 5.10. Često se dešava da se pri pisanju enkodera unutar testa pogrešno unese polje JSON objekta, naročito kada su ovakvi podaci u pitanju — npr. unese se niska *accessToken* umesto *access_token*. Pozivanjem funkcije *Expect.pass*, test uvek prolazi. *Expect.fail* omogućava da se u slučaju neuspešnog testa ispiše poruka koja će objasniti o čemu se radi. U ovom slučaju, poruka je izvučena direktno iz rezultata, koji je tipa *Decode.Error*. Pokretanje ovog testa će dati izlaz prikazan na slici 5.3. Na osnovu ovakvog izlaza, tačno se može zaključiti na kom mestu nastaje problem, i u skladu sa tim ispraviti kôd ili sam test.

```
...
let result = encodeSession session
             |> Decode.decodeValue Session.decodeSession
in
  case result of
     Ok _ ->
          Expect.pass
     Err err ->
          Expect.fail (Decode.errorToString err)
```

Listing 5.10: Implementacija testa koji izbacuje poruku u slučaju neuspeha



Slika 5.3: Izlaz sa porukom o grešci dobijen upotrebom funkcije *Expect.fail*

5.3 Testiranje arhitekture Elm

U delu 3.2 objašnjen je *MVU* obrazac projektovanja koji svaki *Elm* program implementira. Glavne funkcije koje se nalaze u svakom od modula su funkcije ***update*** i ***view***. U ovoj sekciji, najpre će biti prikazano testiranje funkcija ažuriranja *update*, čija je uloga da na osnovu primljene poruke od pretraživača kreiraju novi model. Nakon toga, napisani su testovi za funkcije pogleda *view*, na osnovu kojih se generiše HTML.

Testiranje promena stanja aplikacije

Celokupno stanje *Elm* aplikacije predstavljeno je jednom vrednošću modela. Model aplikacije definisan je u datoteci *Main.elm*. *Model* se menja tako što funkcija *update* prihvati poruku *Msg* i kao rezultat izvršavanja vrati novi *Model*. Anotacija funkcije *update* prikazana je u kodu 5.11. Povratna vrednost je torka modela i komande. Komande u *Elm-u* predstavljaju vrednosti koje opisuju operacije koje okruženje *Elm* treba da izvrši, a koje se ne mogu predstaviti funkcijama. Funkcija *update* predstavlja centralno mesto izvršavanja komandi. *Msg* se odnosi na tip poruke koja se vraća aplikaciji.

```
update : Msg -> Model -> ( Model, Cmd Msg)
```

Listing 5.11: Anotacija funkcije *update*

Svaka promena stanja aplikacije može se testirati tako što se u testovima proslede odgovarajuća poruka i model kao argumenti funkcije *update*, i zatim ispita dobijeni model. Svaka stranica koja ima svoj modul, imaće definisan model, poruke i funkcije *update* i *view* koje se pozivaju iz glavnog modula aplikacije. Sve poruke objedinjuju se u modulu *Main* u jedan glavni tip poruke, a unutar funkcije *update* glavnog modula se na osnovu poruke poziva funkcija *update* odgovarajuće stranice. Trenutni model se ažurira novim modelom date stranice, a poruku komande je potrebno transformisati u odgovarajući glavni tip poruke.

U okviru modula *LoginPage*, definisana je funkcija *update* prikazana u kodu 5.12. Na osnovu vrste poruke koja se prosledi, model se menja na određeni način. Na primer, kada se prosledi tip poruke *Email*, desiće se jednostavna izmena stanja — ažuriraće se polje *email*.

```
update : Msg -> Model -> ( Model, Cmd Session.Msg )
update msg model =
    case msg of
        Email email ->
            ( { model | email = email }, Cmd.none )
        ...
        SubmittedForm ->
            ({model | error = Nothing, processing = True},
              getSession {...})
        ...
```

Listing 5.12: Funkcija *update* stranice za prijavljivanje korisnika *LoginPage*

Testovi za ovu funkciju prikazani su u primeru koda 5.13. Napisan je po jedan faz test za svaki od tipova poruka. Kreira se fazer za model i zatim se nad njim pozove funkcija ažuriranja. Pomoću *Tuple.first* dohvati se samo prvi član torke iz rezultata, koji predstavlja model. Na kraju se proveri da li polje ima odgovarajuću vrednost. Kada je u pitanju drugi deo povratne vrednosti (*Cmd msg*), *Elm* još uvek nema mogućnost za ispitivanje vrednosti *Cmd* kako bi se utvrdilo koju komandu predstavlja. Jedan od načina da se to postigne jeste da se testiraju podaci pre nego što se pretvore u komandu. Umesto direktnog testiranja komandi, testiraju se funkcije koje kreiraju te vrednosti. U ovom primeru to je funkcija *getSession*. Ideja je da se funkcija oblika *getCmd : Foo -> Cmd Msg* podeli na dve funkcije kako bi se olakšalo testiranje:

1. *getData : Foo -> FooData* — funkcija koja se testira
2. *sendFooData : FooData -> Cmd Msg* — jednostavna funkcija koja se ne testira

Jedan od alternativnih načina testiranja komandi je da se kreira specijalan tip koji će predstavljati sve moguće komande aplikacije i postaviti ga kao povratnu vrednost funkcije *update*. Pored toga, napisati funkciju koja prevodi taj tip u *Cmd Msg*. Na taj način će povratna vrednost biti nešto što se može do detalja ispitati. Iako korisna, ova tehnika se retko koristi, i najčešće se jednostavno preskoči testiranje samih komandi. Postoji još korisnih tehnika koje ovde neće biti spominjane jer zahtevaju značajne izmene originalnog koda [4, 7].

```
[fuzz2 string loginModelFuzzer "Email msg sets the email" <|
\email model ->
    model
    |> update (Email email)
    |> Tuple.first
    |> .email
    |> Expect.equal email,
...
```

Listing 5.13: Testovi za funkciju *update* modula *LoginPage*

Testiranje HTML sadržaja

Nakon što funkcija ažuriranja kreira model, on se prosleđuje funkciji *view*, koja taj model transformiše u HTML koji će se prikazati korisniku. Ona prihvata model

kao ulaz, a njena povratna vrednost ima tip HTML poruke. Anotacija svake funkcije *view* je ista i data je u kodu 5.14.

```
view : Model -> Html Msg
```

Listing 5.14: Anotacija funkcije *view*

Za kreiranje HTML čvorova i atributa koriste se funkcije iz modula *Html*. Za opisivanje izgleda stranice u ovom projektu korišćene su funkcije modula *Html.Styled*. Stilizovanje HTML elemenata bez upotrebe CSS datoteka omogućeno je pomoću paketa *NoRedInk/noredinkui*. U primeru koda 5.15 prikazana je funkcija pogleda za stranicu prijavljivanja korisnika. Funkcije koje pruža modul *Html*, koje služe za kreiranje čvorova i atributa, imaju nazive po HTML oznakama (*node*, *button* itd.) i tako omogućavaju bolju čitljivost koda. U ovom primeru, navode se jednostavni elementi kao što su zaglavlja, polja za unos imejl adrese i šifre, i jedno dugme koje aktivira događaj.

```
view model =
 Container.view
  [ Heading.h3 [ Heading.css [ marginBottom (px 20) ] ]
    [ Html.text "Prijava korisnika" ]
    ..., Button.button "Prijavi se"
     [  ... , Button.onClick SubmittedForm ]]]
```

Listing 5.15: Funkcija *view* modula *LoginPage*

Testiranje funkcija pogleda podrazumeva ispitivanje konkretnih HTML vrednosti. Pre pisanja testova za funkcije *view* neophodno je uključiti paket *elm-html-test* [22]. Ovaj paket omogućava kreiranje očekivanja o HTML vrednostima, i zbog toga je njegova glavna namena testiranje funkcija pogleda. Unutar paketa, postoje tri najvažnija podmodula namenjena za testiranje različitih svojstava koje funkcija *view* generiše:

1. ***Test.Html.Query*** — omogućava pisanje upita nad HTML strukturama
2. ***Test.Html.Event*** — omogućava simuliranje događaja nad HTML vrednostima
3. ***Test.Html.Selector*** — omogućava dohvatanje HTML elemenata

Pri testiranju pogleda neophodno je razmisliti šta konkretno testirati. Ne preporučuje se testiranje doslovnog izlaza, jer to izaziva visoku spregnutost između testa i

implementacije samog koda. Na primer, ako se dese sitne izmene u svrhu stilizovanja stranice, to bi dovelo do padanja testa koji doslovno proverava prisutne elemente. Ono što treba testirati jeste neka važna logika koju funkcija implementira. Ako neka funkcija *view* prikazuje odgovarajući tekst u zavisnosti od određenog polja modela, to je nešto što je vredno testiranja. Kako bi se predstavili osnovni koncepti modula *Test.Html.Query*, u primeru koda 5.16 prikazan je jednostavan test koji proverava prisustvo očekivanog teksta pri prikazivanju stranice za prijavljivanje korisnika. U svakom testu funkcija pogleda, nad rezultatom poziva funkcije sa datim modelom prvo se poziva funkcija *Query.fromHtml*. Njena uloga je da prihvati HTML i vrati vrednost tipa *Single*, koja predstavlja koreni čvor HTML-a. U ovom slučaju to je glavni kontejner u kome su sadržani svi elementi sa stranice. Nakon toga, upotrebljena je funkcija *Query.has* koja proverava da li prosleđeni element sadrži sve što mu je dato u listi selektora. U ovoj listi prosleđen je selektor `text` iz modula *Html.Selector*, koji treba da se poklopi sa svim elementima koji sadrže atribut `text` sa datom vrednošću. Ova funkcija vraća vrednost tipa *Expectation*, što znači da se može naći na kraju cevovoda u testu. Tekst u testu je fiksiran jer se podrazumeva da je jezik aplikacije srpski. U slučaju da se aplikacija proširi podrškom za neki drugi jezik, bilo bi potrebno parametrizovati testove. Slično kao sa tekstom, može se proveriti prisustvo dugmića, polja za unos teksta i drugih HTML elemenata. Funkcije kao što su *find*, *findAll, contains, count* iz modula *Html.Query*, i *tag, containing, attribute* iz modula *Html.Selector*, su neke od tih koje omogućavaju takve provere.

```
[fuzz loginModelFuzzer "Correctly renders text in DOM" <|
  \model ->
    model
    |> LoginPage.view
    |> Html.toUnstyled
    |> Query.fromHtml
    |> Query.has [ text "Prijava korisnika" ]
```

Listing 5.16: Test za funkciju *view* modula *LoginPage*

Testiranje interakcija sa korisnikom

Kada je utvrđeno da se na stranici prikazuje sve što je očekivano, može se preći na testiranje ispravnosti odgovarajućih događaja — na primer, da li se kôd ponaša očekivano kada se klikne na dugme. Na primeru prijave korisnika, klikom na dugme sa tekstom *"Prijavi se"*, funkciji *update* se prosleđuje poruka tipa *Submitted-*

Form, koja dalje poziva druge funkcije koje će izvršiti zahtev *post*. Pomoću modula *Html.Test.Event* simulira se događaj klika na dugme u testu. Deo testa prikazan je u kodu 5.17. Najpre se pronađe dugme pomoću upita i selektora koji proverava da li dugme sadrži tekst *"Prijavi se"*, a zatim se nad njim simulira događaj klika pozivom funkcije *Event.simulate*, za koji se očekuje da će vratiti poruku tipa *SubmittedForm*.

Drugi tip događaja simuliran u ovom modulu je događaj unosa teksta u polja namenjena za imejl adresu i šifru korisnika. Sintaksa je slična i koristi se funkcija *Event.input* koja prihvata nisku koja se unosi. Unosom teksta ažuriraju se odgovarajuća polja modela. Primer ovog testa dat je u kodu 5.18.

```
test "Click on login button returns SubmittedForm message" <|
  \_ ->
      LoginPage.view testModel
      |> Html.toUnstyled
      |> Query.fromHtml
      |> Query.find
          [ tag "button", containing [ text "Prijavi se" ]]
      |> Event.simulate Event.click
      |> Event.expect SubmittedForm
```

Listing 5.17: Simulacija događaja klika na dugme u testu funkcije *view* modula *LoginPage*

```
fuzz loginModelFuzzer "Input event returns Email msg" <|
  \model ->
      model
      |> LoginPage.view
      |> Html.toUnstyled
      |> Query.fromHtml
      |> Query.find [attribute <| Attributes.placeholder "Email"]
      |> Event.simulate (Event.input "someemail@gmail.com")
      |> Event.expect (Email "someemail@gmail.com")
```

Listing 5.18: Simulacija događaja unosa teksta u testu funkcije *view* modula *LoginPage*

Glava 6

Zaključak

U okviru rada, predstavljene su glavne karakteristike i pristupi testiranju funkcionalnih programa napisanih na programskim jezicima *Elm* i *Elixir*. Na primeru veb aplikacije koja predstavlja portal namenjen sprovođenju aktivnosti na kursu *Metolodogija stručnog i naučnog rada*, prikazani su testovi napisani u *Elixir-u* koji proveravaju ispravnost komponenti serverske strane, i testovi napisani u *Elm-u*, koji proveravaju ispravnost komponenti klijentske strane portala.

Elixir poseduje ugrađeni razvojni okvir za pisanje testova pod nazivom *ExUnit*, koji sadrži alate za testiranje svih komponenti aplikacije, i time omogućava pisanje stabilnih i pouzdanih testova bez potrebe za uključivanjem drugih biblioteka. *Elixir* testovi najpre su napisani za modul koji enkapsulira domensku i poslovnu logiku, u kojima je prikazano kako utvrditi da li su strukture podataka i polja tabela u bazi ispravno definisane, a potom i njihove izmene i validacije pre unosa u bazu, testiranjem koncepta koji se naziva skup promena. Drugi deo testova proverava upite, koji su uglavnom operacije dohvatanja, kreiranja, ažuriranja i brisanja podataka iz baze, gde su se iscprno koristile fabrike za kreiranje veštačkih podataka. Na kraju, testiran je i modul aplikacije koji predstavlja njen veb interfejs, čime je utvrđena ispravna upotreba upravljača i pogleda. Detaljno testiranje serverskog dela aplikacije obezbedilo je dodatnu dokumentaciju i olakšalo buduće refaktorisanje.

S obzirom na to da *Elm* poseduje kompilator koji proverom tipova garantuje ispravnost ulaznih podataka, i koristi tipove *Maybe* i *Result* za obradu neispravnih scenarija, broj mogućih grešaka u *Elm* aplikacijama je minimalan. Zbog toga, testiranje se često i preskače. U odnosu na opseg testova serverske aplikacije, opseg testova klijentske aplikacije napisane u programskom jeziku *Elm* je manji. Ono što je bilo važno pokriti testovima su najpre dekoderi i enkoderi JSON podataka. Testovi

su utvrdili mogućnost aplikacije da tačno serijalizuje podatke, što je jedan od najvažnijih zadataka aplikacija koje se oslanjaju na interakcije sa veb interfejsom. Druga grupa *Elm* testova koja je prikazana su testovi koji proveravaju ispravnost funkcija koje se odnose na interakcije sa korisnikom. Njima su pokrivene promene stanja koje se očekuju pri različitim akcijama, kao i da li se u korisničkom interfejsu prikazuje ispravan i očekivan sadržaj. Važno je napomenuti da su u ovom radu izostavljeni testovi koji proveravaju HTTP zahteve i interakcije klijenta sa API-jem, zbog ograničenog izbora razvojnih okvira za testiranje. Ove operacije su implementirane kao komande, koje u *Elm-u* predstavljaju akcije koje se ne mogu predstaviti funkcijama. U okviru okruženja *elm-test* i dalje ne postoji podrška za testiranje komandi, i kako bi se one ispitale potrebno je uvesti velike promene u originalnom kodu. Ovo ograničava opseg testova, ali u ovom slučaju ne predstavlja veliki nedostatak, s obzirom na to da *Elm* kompilator i njegovi tipovi pokrivaju veliki broj mogućih nepravilnosti. Takođe, jednostavna provera izvršavanja HTTP zahteva može se izvesti manuelno iz pretraživača upotrebom opcije *Inspect* i ispitivanjem sekcije *Network*.

Kvalitet koda aplikacije se može unaprediti dodavanjem obrada različitih vrsta neočekivanih scenarija, pogotovo kada je u pitanju *Elixir*. Što se tiče klijentske strane, ono što je potrebno uraditi je refaktorisati celu aplikaciju, specifično upotrebu komandi. U tom slučaju, za testiranje se može iskoristiti modul *ProgramTest*, koji simulira izvršavanje *Elm* programa. Testiranje *Elm* aplikacija ima mnogo veći značaj ako se sprovodi uporedo sa razvojem samog projekta, što u ovom radu nije bio slučaj. Na kraju, kako bi se utvrdilo da celokupan sistem funkcioniše ispravno, bilo bi korisno uključiti i dodatne sistemske testove, kao i testove opterećenja.

Ograničen broj napisanih testova nije otkrio značajne greške u implementaciji veb portala *MSNR*. Zbog toga što je u pitanju funkcionalan kôd koji je po prirodi robustan, i detaljno manuelno istestiran, a pritom ne sadrži kompleksne algoritme, i nije bilo mnogo prostora za greške. Može se diskutovati o tome koliko je prikazano iscrpno testiranje trenutno isplativo na ovako maloj aplikaciji. Ako bi se planirao dalji razvoj ove aplikacije, i kada bi ona dovoljno narasla, napisani testovi bi postali dragoceni, jer svako manuelno testiranje je neuporedivo skuplje od pokretanja automatskih testova. U slučaju budućih izmena funkcionalnosti koda i same aplikacije, održavanje ovih testova ne predstavlja velik posao, s obzirom na to da su izolovani po komponentama. Izmene u konkretnoj komponenti u kodu pratile bi jednostavne izmene u jednom testu.

Bibliografija

[1] Adresa sa implementacijom portala msnr. url: https://github.com/NemanjaSubotic/master-rad/tree/master/portal.

[2] Functional programming paradigm. url: https://www.geeksforgeeks.org/functional-programming-paradigm/.

[3] Node.js upravljač paketa. url: https://www.npmjs.com/package/elm-test.

[4] Paket elm-testable. url : https://package.elm-lang.org/packages/rogeriochaves/elm-testable/latest.

[5] REST architectural style. url: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[6] Software Testing Life Cycle. on-line at: https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/.

[7] Testing programs with Cmds. url : https://elm-program-test.netlify.app/cmds.html#introducing-the-example-program.

[8] Zvanična dokumentacija biblioteke Ecto. url: https://hexdocs.pm/ecto/Ecto.html.

[9] Zvanična dokumentacija biblioteke ExMachina. url: https://hexdocs.pm/ex_machina/readme.html.

[10] Zvanična dokumentacija biblioteke Faker. url: https://hexdocs.pm/faker/readme.html.

[11] Zvanična dokumentacija biblioteke Plug. url: https://hexdocs.pm/plug/readme.html.

[12] Zvanična dokumentacija modula ExMachina.Ecto. url: https://hexdocs.pm/ex_machina/ExMachina.Ecto.html.

[13] Zvanična dokumentacija okruženja ExUnit. url: https://hexdocs.pm/ex_unit/ExUnit.html.

[14] Zvanična dokumentacija Phoenix upravljača. url: https://hexdocs.pm/phoenix/controllers.html.

[15] Zvanična stranica alata elm-test. url: https://www.npmjs.com/package/elm-test.

[16] Zvanična stranica alata Hex. url: https://hex.pm/.

[17] Zvanična stranica baze podataka PostgreSql. url: https://www.postgresql.org.

[18] Zvanična stranica biblioteke Expect. url: https://package.elm-lang.org/packages/elm-explorations/test/latest/Expect.

[19] Zvanična stranica biblioteke Fuzz. url: https://package.elm-lang.org/packages/elm-explorations/test/latest/Fuzz.

[20] Zvanična stranica biblioteke Pbkdf2. url: https://hexdocs.pm/pbkdf2_elixir/Pbkdf2.html.

[21] Zvanična stranica biblioteke Test. url: https://package.elm-lang.org/packages/elm-explorations/test/latest/Test.

[22] Zvanična stranica paketa elm-html-test. url: https://package.elm-lang.org/packages/eeue56/elm-html-test/5.2.0/.

[23] Zvanična stranica programskog jezika Elixir. url: https://elixir-lang.org/.

[24] Zvanična stranica programskog jezika Elm. url: https://guide.elm-lang.org/.

[25] Zvanična stranica programskog jezika Erlang. url: https://www.erlang.org/.

[26] Zvanična stranica programskog jezika Haskell. url: `https://www.haskell.org/`.

[27] Zvanična stranica razvojnog okruženja Phoenix. url: `https://www.phoenixframework.org`.

[28] Ulisses Almeida. An Introduction to Test Factories and Fixtures for Elixir. 2023.

[29] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. A Mike Cohen signature book. Addison-Wesley, 2010.

[30] Elfriede Dustin. Effective Software Testing: 50 Specific Ways to Improve Your Testing.

[31] Richard Feldman. *Elm in Action*. Manning, 2020.

[32] Andrea Leopardi and Jeffrey Matthias. *Testing Elixir - Effective and Robust Testing for Elixir and its Ecosystem*. Pragmatic Bookshelf, 2021.

[33] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[34] Nemanja Subotić. Programski jezici Elm i Elixir u razvoju studentskog veb portala, 2022. url: `http://elibrary.matf.bg.ac.rs/bitstream/handle/123456789/5482/MasterRadNemanjaSubotic.pdf?sequence=1`.

[35] Klaus Olsen, Meile Posthuma, and Stephanie Ulrich. International Software Testing Certified Tester Foundation Level (CTFL) Syllabus. url: `https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf.`

[36] German Velasco. Mocking External Dependencies in Elixir. 2022.

Biografija autora

Ana Petrović rođena je 20.12.1996. godine u Beogradu. Osnovnu školu Miloš Crnjanski u Beogradu završila je 2011. godine. Nakon toga, pohađala je prirodno-matematički smer u Trinaestoj beogradskoj gimnaziji, koju je završila 2015. godine. Iste godine, upisala je Matematički fakultet Univerziteta u Beogradu, smer Informatika. Na istom je diplomirala u januaru 2020. godine. Nakon diplomiranja, upisala je master studije na istom fakultetu, na smeru Informatika, na kom je uspešno položila sve ispite sa prosečnom ocenom 9,80. Oblasti interesovanja uključuju funkcionalno programiranje, mašinsko učenje i veštačku inteligenciju.