UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ana Petrović

TESTIRANJE SOFTVERA U FUNKCIONALNIM PROGRAMSKIM JEZICIMA ELM I ELIXIR

master rad

Beograd, 2023.

Mentor:

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

Članovi komisije:

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

dr Ivan Čukić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Testiranje softvera u funkcionalnim programskim jezicima Elm i Elixir

Rezime: Apstrakt

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

Sadržaj

1 Uvod **1**

2 Funkcionalna paradigma **2**

2.1 Karakteristike funkcionalnih jezika 2

2.2 Testiranje u razvoju softvera 5

2.3 Testiranje funkcionalnih programa 10

3 Portal MSNR **13**

3.1 Funkcionalnosti i osnovni entiteti portala 13

3.2 Arhitektura portala 15

3.3 Testiranje portala 18

4 Testiranje serverskog dela aplikacije **19**

4.1 Testiranje jedinica koda u programskom jeziku Elixir 19

4.2 Integraciono testiranje 23

4.3 Testiranje komunikacije sa bazom podataka 23

5 Testiranje klijentskog dela aplikacije **42**

6 Testiranje celokupnog sistema — End to End **43**

6.1 Integracija klijentske i serverske strane 43

6.2 Testiranje opterećenja 43

7 Zaključak **44**

Bibliografija **45**

Glava 1

Uvod

Ovde ide uvod .. ..
uvod uvod
sta se nalazi u kom poglavlju...

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 [4]. 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*). To znači da u okviru jezika ne postoje restrikcije po pitanju

njihovog kreiranja i korišćenja. 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 transparetnosti 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. Definicija se može proširiti i na funkcije: funkcija poseduje transparentnost referenci ako pri pozivu 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 vrši nikakve izmene nad argumentima, kao ni nad promenljivima. 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. *Fold* prihvata tri argumenta: funkciju spajanja, početnu vrednost i listu. Iznova primenjuje funkciju spajanja na početnu vrednost i datu listu, sve dok se rezultat ne redukuje na jednu vrednost.

Prednost korišćenja ovih funkcija je u sažetom i čistom kodu. Takođe, pogodne su i za paralelizaciju. Primer koda 2.2 pokazuje upotrebu funkcija višeg reda u programskom jeziku Elm.

```
[1, 2, 3] |> List.map (\number -> number * 2) -- [2, 4, 6]
[1, 2, 3, 4, 5] |> List.filter (\number -> number <= 3) -- [1, 2, 3]
[1, 2, 3, 4, 5] |> List.foldl (\item total -> total + item) 0 -- 15
```

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. Funkcije se definišu rekurzivno — pozivaju same sebe i time postižu ponavljanje izvršavanja. U mnogim slučajevima, umesto rekurzije se koriste funkcije višeg reda.

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 [1]. 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 osigurava 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 [13]. Uloga piramide je da vizuelno predstavi logičku organizaciju standarda u testiranju. Sastoji se od tri sloja: bazu piramide predstavljaju jedinični testovi (eng. *unit test*). 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. 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 interfejsa. Vrh piramide čine sistemski testovi. Oni se ne fokusiraju na individualne komponente, već testiraju čitav sistem kao celinu i time utvrđuju da on radi očekivano i ispunjava 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 pona-

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

ša 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 jediničnih, integracionih i sistemskih testova kako bi se osigurala ukupna funcionalnost, pouzdanost i perfomanse sistema.

Slika 2.1: Model piramide testiranja

Testovi jedinica koda

Jedinica je mala logička celina koda: može biti funkcija, klasa, metod klase, modul i slično. Jedinični test 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 najjednostavniji za pisanje jer se bave malim delom aplikacije, te je kôd koji se testira najčešće vrlo jednostavan.

Cilj jediničnih testova 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.

Anatomija jediničnih testova

Kako bi kôd jediničnog testa bio čitljiv i jednostavan za razumevanje, obrazac četvorofaznog testa (eng. *four-phase test*) predlaže strukturu testa koja podrazumeva ne više od četiri faze [15]. Svaki test jedinice koda se može podeliti na četiri jasno odvojive celine:

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, 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.

```
import Test exposing(..)

sumTest : Test
sumTest =
    describe "sum" [
      test "should add two numbers correctly" <| \() ->
        let
        --Setup
           x = 2
           y = 3
           expected = 5
        in
        -- Exercise (sum)
```

```
        -- Verify (expect)
           expect (sum x y) |> toEqual expected
    ]

-- Teardown
teardown : Int -> ()
teardown _ =
    ()
```

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

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 testiranje 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 jediničnim testovima, 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 jediničnog 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 resursa. 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 perfomansi (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 obezbeđuje 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

Najvažnija stvar kod testiranja u funkcionalnoj paradigmi jeste pisanje čistih funkcija i njihovo testiranje u izolaciji, kako bi se obezbedila ispravnost i robustnost. Takođe je važno da se ne testira samo uspešan scenario, već i granični slučajevi, kao i slučajevi greške.

Testiranje čistih funkcija

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

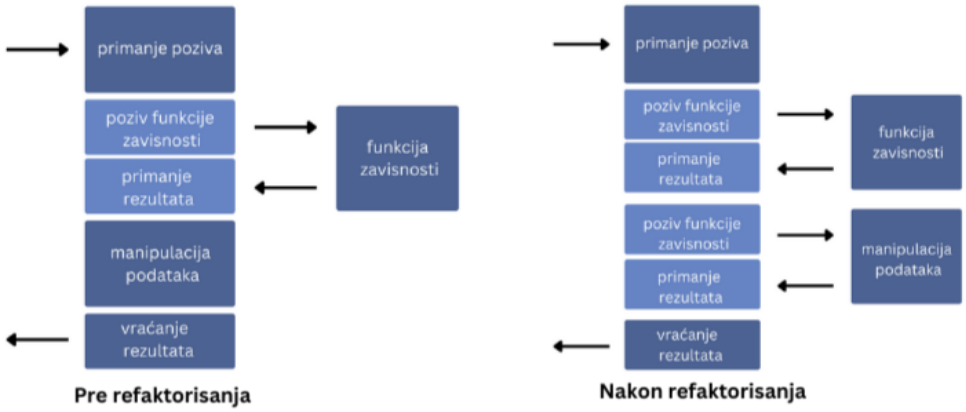
Kada je u pitanju čista funkcija, jedina priprema koja je potrebna jesu podaci koji će se proslediti kao parametri. Drugi korak jeste poziv funkcije, sa prosleđenim argumentima. Faza verfikacije podrazumeva samo provere nad rezultatom. Testovi su veoma jednostavni jer ne moraju da brinu o propratnim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

Aplikacije se u većini slučajeva neće sastojati od isključivo čistih funkcija, i u vezi sa tim postoje dve strategije [14]. 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.

[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.

Refaktorisanje ka čistim funkcijama

Ako je neki deo koda komplikovan za testiranje, najlakši način da se pojednostavi jeste refaktorisati ga u čistu funkciju, ukoliko je to moguće. Deo koda koji zavisi od nekog drugog dela iz spoljašnjosti će u većini slučajeva pozivati tu spoljašnju zavisnost i onda manipulisati rezultatom pre nego što vrati svoj rezultat. Što više takve manipulacije ima, to je taj deo koda bolji kandidat za premeštanje logike unutar čiste funkcije. Na slici 2.3 su date vizuelne reprezentacije kako ovaj proces izgleda pre i posle izmeštanja koda u čistu funkciju. Na početku, funkcija može biti komplikovana za testiranje. Svaki test, za svaki mogući način ponašanja bi nekako morao da garantuje da druga funkcija (ona od koje zavisi prva) vraća neki očekivani odgovor. Ideja je da se deo logike (na slici označen sa "manipulacija podataka ") izvuče van — u novu, čistu funkciju. Tako postaje zasebna komponenta, koja se može odvojeno lako testirati. Kada je taj deo logike dobro istestiran, može se smatrati sigurnim da se ponovo pozove u originalnoj funkciji. Zna se da će taj čisti kôd uvek vraćati isti rezultat, i može se značajno redukovati broj testova neophodnih za testiranje originalne funkcije. U primeru koda... TODO primer



Slika 2.2: Izmeštanje koda u čistu funkciju

U nekim slučajevima, nije lako odrediti šta se može izdvojiti u zasebnu funkciju. Tada postoji druga opcija za kreiranje kontrolisanog okruženja: napraviti zamenu za funkciju zavisnosti, i time izolovati kôd.

Izolovanje koda

Ubrizgavanjem zavisnosti i kreiranjem dublera moguće je eliminisati spoljašnje promenljive, i time kontrolisati situaciju u kojoj kod koji se testira mora da se nađe. Tako se omogućava očekivanje nekog konkretnog rezultata.

Zavisnost je bilo koji kôd na koji se originalni kôd oslanja. Korišćenjem DI (skraćenica za Dependency Injection) u testovima se kreiraju zamene zavisnosti koje se ponašaju na predvidljiv način, pa testovi mogu da se fokusiraju na logiku unutar koda koji se testira. U jediničnim testovima, najčešće se ubrizgava zavisnost tako što se prosledi kao parametar. Taj parametar može biti funkcija ili modul. Ubrizgavanje zavisnosti kroz API[3] obezbeđuje čist kôd i jednostavne i kontrolisane jedinične testove. Sa druge strane, integracioni testovi zahtevaju drugačije metode ubrizgavanja zavisnosti. TODO nastavak...

[3]skraćenica za aplikacioni veb interfejs (eng. *web applicatoin programming interface*)

Glava 3

Portal MSNR

Kôd aplikacije pod nazivom *Portal MSNR* koja će biti testirana javno je dostupan na *GitHub-u* [2]. Portal MSNR je veb aplikacija namenjena praćenju i upravljanju aktivnostima kursa *Metodologija stručnog i naučnog rada* [16]. 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.

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 rolu profesora, a pri odobravanju registracije studenta kreira se nalog sa rolom 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.

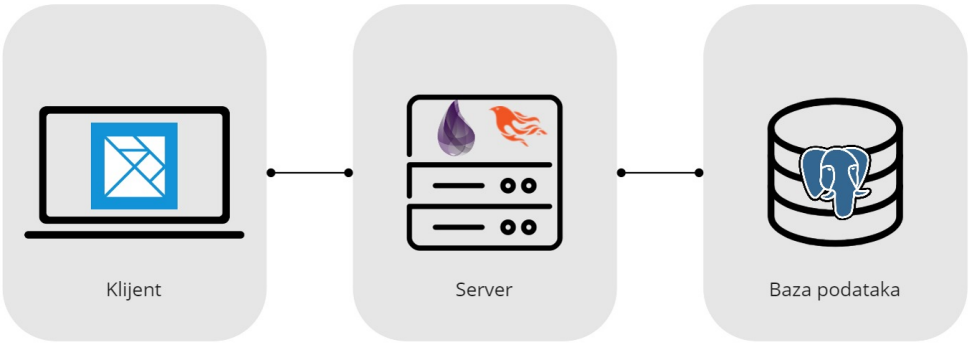
Spisak naziva entiteta i tabela u okviru baze podataka koje njima odgovaraju dati su u tabeli 3.1. Svaki od ovih entiteta, kao i relacije između njih, biće pojedinačno istestirani u narednom poglavlju.

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 podataka, i sistem za upravljanje bazom *PostgreSQL* [11]. Slika 3.2 prikazuje navedenu arhitekturu.

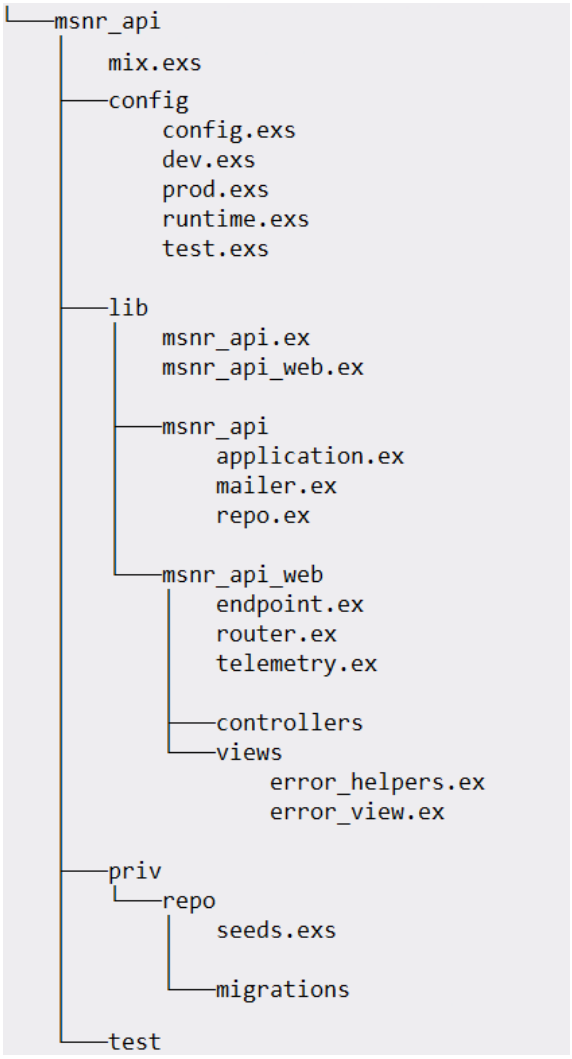


Slika 3.1: Arhitektura Portala MSNR [16]

Struktura serverske strane portala

U programskom jeziku Elixir, za razvoj veb aplikacija koristi se razvojni okvir *Phoenix* [12]. Zasnovan je na obrascu *model-pogled-upravljač* (eng. *Model-View-Controller pattern, MVC*). Serverski deo aplikacija MSNR portal 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* [10]. Pri kreiranju *Phoenix* projekta, dodeljeno mu je ime *msnr_api*. Na slici 3.2 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



Slika 3.2: Struktura *Phoenix* projekta *msnr_api* [16]

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.

Za sve obrade HTTP zahteva koristi se biblioteka *Plug* [9]. 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 dru-

gim, 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. Inicijalizacija ovog projekta podrazumeva kreiranje jednog praznog direktorijuma src i datoteke elm.json, a sam korisnik odlučuje o organizaciji datoteka unutar projekta. Na slici 3.2 prikazano je rešenje organizacije Elm datoteka u okviru aplikacije Portal MSNR.

```
src
    Activity.elm
    ActivityType.elm
    Api.elm
    Assignment.elm
    FileInfo.elm
    Group.elm
    LoginPage.elm
    Main.elm
    Page.elm
    ProfessorPage.elm
    RegistrationPage.elm
    Route.elm
    Session.elm
    SetPasswordPage.elm
    Student.elm
    StudentPage.elm
    Topic.elm
    UserType.elm
    Util.elm

    ProfessorPage

    StudentPage
```

Slika 3.3: Struktura *Elm* projekta *msnr_elm* [16]

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, 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 url promenama.

Aplikacija je kompajlirana tako da se kreira datoteka *app.js*, koja se uključuje dokument *index.html*. Pokreće se pozivanjem *init* funkcije 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

Testiranje ovakve aplikacije podrazumeva podelu na različite vrste testova. Za početak, jedinični testovi koji se odnose na individualne funkcije i upravljače koji barataju zahtevima u okviru serverskog dela aplikacije moraju biti napisani u programskom jeziku Elixir. Na serverskoj strani, potrebno je napisati i testove koji simuliraju zahteve API-ju i verifikuju odgovore od baze. Sa druge strane, jedinični testovi koji se fokusiraju na pojedinačne komponente i funkcije korisničkog interfejsa moraju biti napisani u programskom jeziku Elm. Nakon pojedinačnog testiranja klijentske i serverske aplikacije, slede integracioni testovi koji proveravaju kako korisnički interfejs funkcioniše zajedno sa API-jem. Na kraju je neophodno testirati celokupan sistem, od korisničkog interfejsa do baze podataka, pisanjem sistemskih testova. S obzirom na to da se radi o veb aplikaciji, mogu se sprovesti i testovi opterećenja koji će proveriti kako portal podnosi velike količine zahteva i korisnika.

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 [3].

4.1 Testiranje jedinica koda u programskom jeziku Elixir

S obzirom na to da je Elixir funkcionalni jezik, može se diskutovati o tome šta se smatra „jedinicom". Uobičajeno je da se jedinični testovi 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.

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 *test* direktorijuma. 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
```

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 funkcijima 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 slučajeve. U ovom primeru, modul pod nazivom *AccountsTest* sadrži testove koji se odnose na kontekst koji opisuje korisnike. Blok *describe* iz primera se sastoji od dva test slučaja, i odnosi se na funkciju *get_user*, koja vrši jednostavno dohvatanje korisnika iz baze prosleđivanjem identifikatora. 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 slučaj. Na primeru *describe* bloka funkcije *get_user*, prikazana su dva test slučaja, od kojih jedan proverava uspešno izvršavanje funkcije kada se prosledi validan identifikator, a drugi proverava da li se javlja greška kada se prosledi identifikator korisnika koji ne postoji.

Unutar jednog test slučaja 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 *get_user* vrati korisnika koji je jednak

postojećem korisniku iz baze, ovaj test uspešno prolazi.

```
defmodule MsnrApi.Queries.AccountsTest  do
...
describe "get_user/1" do
    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 a NoResultsError when a user doesn't
   exist" do
      assert_raise Ecto.NoResultsError, fn ->
        Accounts.get_user!(invalid_id) end
    end
 end
```

Listing 4.2: Opisivanje testova unutar jedne grupe, na primeru funkcije za dohvatanje korisnika

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 dohvaćeni korisnik nije onaj koji je očekivan, prikazan je na listingu 4.3.

```
1) test get_user/1 success: it returns a user when given a valid id (MsnrApi.Queries.
    AccountsTest)
     test/msnr_api/queries/accounts_test.exs:73
     ** (Ecto.NoResultsError) expected at least one result but got none in query:

     from u0 in MsnrApi.Accounts.User,
       where: u0.id == ^6

     code: assert returned_user = Accounts.get_user!(6)
     stacktrace:
       (ecto 3.7.1) lib/ecto/repo/queryable.ex:156: Ecto.Repo.Queryable.one!/3
       test/msnr_api/queries/accounts_test.exs:76: (test)

Finished in 0.5 seconds (0.0s async, 0.5s sync)

2 tests, 1 failures
```

Listing 4.3: Izlaz u slučaju testa koji ne prolazi

U narednoj listi dati su nazivi i opisi makroa koji se mogu koristiti u testovima pored makroa *assert*:

- *refute* — koristi se kada je potrebno utvrditi da je izraz uvek neistinit.
- *assert_raise* — koristi se kada je potrebno proveriti da li se javlja konkretan 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 standardanom 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.

U test slučaju greške iz primera 4.2, iskorišćen je i makro *assert_raise* — očekuje se da se pri dohvatanju korisnika koji ne postoji javi izuzetak *NoResultsError*. Upotreba makroa *refute* i *setup* prikazana je u primeru dela koda 4.23. 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. Pri testiranju funkcije *delete_user*, koja treba da ukloni korisnika iz baze, iskorišćen je makro *refute*. Kada se pokuša dohvatanje korisnika koji je prethodno izbrisan, trebalo bi da to nije moguće i da dođe do greške. Zbog toga je u ovom slučaju *refute* dobar izbor.

```
setup do
    Ecto.Adapters.SQL.Sandbox.checkout(MsnrApi.Repo)
end

describe "delete_user/1" do
   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
end
```

Listing 4.4: Upotreba makroa *setup* i *refute* na primeru funkcije *delete_user*

4.2 Integraciono testiranje

}

Nakon dobro istestiranih pojedinačnih funkcija i modula, potrebno je proveriti da li komponente funkcionišu ispravno kao celina. U ovoj sekciji biće prikazani testovi koji proveravaju da li različiti delovi sistema koji komuniciraju međusobno, kao i sa nekim eksternim sistemima, rade to na ispravan način.

Testiranje komunikacije sa bazom podataka

Ecto biblioteka zadužena je za sve interakcije sa relacionim bazama podataka u *Elixir* okruženju [6]. 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 ostvaraje 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. *Ecto.Changeset* modul opisuje kako se menjaju podaci. U ovom delu, prikazani su testovi koji proveravaju da li kôd koristi funkcionalnosti *Ecto* biblioteke 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 vezi sa bazom podataka 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 *Ecto.Schema* modula. 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 kasnije.

```
defmodule MsnrApi.Assignments.Assignment do
  use Ecto.Schema
  import Ecto.Changeset

  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 se odnosi na prethodni primer prikazan je u kodu 4.7. Ovo je primer jednostavnog jediničnog testa koji proverava da li definisana shema ima tačna polja i odgovarajuće tipove. Unutar testa se prvo prolaskom kroz sva polja Assignment strukture 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 *assert* naredbe, i prava i očekivana lista pretvorene su u *MapSet* strukturu, kako bi se redosledi polja poklapali sa obe strane. Slični testovi napisani su i za sve ostale entitete navedene u sekciji 3.1.

```
defmodule MsnrApi.Schema.AssignmentTest do
...
  describe "fields and types" do
    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
 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 *changeset* funkciju. 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 validaterequired proverava obavezna polja. Rezultat izvršavanja ovih funkcija je takođe *Ecto.Changeset* struktura, 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 *describe* bloka *"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 *mix.exs* datoteke, koje su prikazane u kodu 4.8. Ovime se govori aplikaciji da uključi sve datoteke u *test* direktorijumu 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 [
     ...
     elixirc_paths: elixirc_paths(Mix.env()),
]
```

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

Modul *SchemaCase* sadrži dve funkcije, od kojih jedna konstruiše realistične podatke ispravnih tipova, a druga konstruiše podatke koji su pogrešnog tipa u odnosu na polje tabele. Funkcije *valid_params* i *invalid_params* prikazane su u primeru koda 4.9. Ove funkcije kao povratnu vrednost imaju mapu koja sadrži niske naziva polja kao ključeve, i odgovarajuće generisane vrednosti koje njima odgovaraju.

Biblioteka *Faker* [8] ima ulogu u formiranju nasumičnih realističnih podataka. Na primer, pomoću modula ove biblioteke pod nazivom *Lorem*, mogu se dobiti nasumične reči koje će predstavljati polja tabela koja treba da imaju nisku kao svoju vrednost. *Faker* je potrebno uključiti u zavisnosti projekta, dodavanjem linije *{:faker, "~> 0.17", only: :test}* u *deps* delu konfiguracione datoteke *mix.exs*. Biblioteku nije potrebno koristiti u razvojnom i produkcionom okruženju, te se ovom linijom ograničava njena upotreba samo na testno okruženje.

```
def valid_params(fields_with_types) do

    valid_value_by_type = %{
      string: fn -> Faker.Lorem.word() end,
      naive_datetime: fn -> Faker.NaiveDateTime.backward(Enum.
   random(0..100)) end,
      id: fn -> Enum.random(0..100) end,
      ...
    }

    for {field, type} <- fields_with_types, into: %{} do
      case field do
        {Atom.to_string(field), valid_value_by_type[type].()}
      end
    end
  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,
      id: fn -> DateTime.utc_now() end,
      ...
    }

    for {field, type} <- fields_with_types, into: %{} do
      {Atom.to_string(field), invalid_value_by_type[type].()}
```

```
    end
  end
```

Listing 4.9: Definicije pomoćnih funkcija *valid_params* i *invalid_params*

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 slučaj koji proverava uspešnu putanju izvršavanja funkcije *changeset/2* prikazan je u primeru koda 4.10. 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 when given valid
   arguments" do
      valid_params = valid_params(@required_fields)
      changeset = Assignment.changeset(%Assignment{}, valid_
   params)

      assert %Changeset{valid?: true, changes: changes} =
   changeset

      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
  end
```

Listing 4.10: Test slučaj uspešne upotrebe funkcije *changeset/2*

Drugi test slučaj 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 *assert* naredbom. Ovaj test slučaj prikazan je u primeru koda 4.11.

```
test "error: returns an invalid changeset when given uncastable
   values" do
      invalid_params = invalid_params(@required_fields)

      assert %Changeset{valid?: false, errors: errors} =
   Assignment.changeset(%Assignment{}, invalid_params)

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

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

Listing 4.11: Test slučaj neuspešne upotrebe funkcije *changeset/2*, prosleđivanjem nekastabilnih 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 test slučaja dat je u kodu 4.12. 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 *refute* naredbe, utvrđuje se da se opciona polja ne nalaze u listi grešaka. To su polja koja nije neophodno navesti pri pozivanju ove funkcije, i oni se zato ne trebaju naći ni među greškama.

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

```
      for {field, _} <- @required_fields do
        assert errors[field], "The 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
   required when it shouldn't be."
      end
end
```

Listing 4.12: Test slučaj neuspešne upotrebe funkcije *changeset/2*, sa nedostajućim parametrima

Neke od shema će unutar svoje *changeset* funkcije 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 slučaj prikazan je u primeru koda 4.13. Neophodno je ostvariti direktan pristup bazi podataka, pa je prva linija unutar test slučaja 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 ubacivanje još jednog korisnika sa istom imejl adresom. To bi trebalo da izazove grešku, što se proverava prvom *assert* naredbom. Druga *assert* naredba 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 changeset 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, %Changeset{valid?: false, errors: errors}}
=
      MsnrApi.Repo.insert(changeset_with_reused_email)

    assert errors[:email], "The field :email is missing from
errors."
    {_, meta} = errors[:email]

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

Listing 4.13: Test slučaj neuspešne upotrebe funkcije *changeset/2*, pri narušavanju ograničenja jedinstvenosti

Što se tiče faze rušenja, *Ecto* obezbeđuje da svaki pojedinačni test ne mora da prati i vraća okruženje 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 4.14. U datoteci schema_case potrebno je dodati *setup* blok 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 *Ecto* delu, dodaju se linije koje obaveštavaju *Ecto* da će se koristiti *Sandbox*.

```
# 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 4.14: Podešavanje *Ecto.Sandbox*

Testiranje upita

U ovom delu biće prikazano kako su testirani konteksti entiteta. Modul koji će služiti kao primer se odnosi na korisnike — *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
  alias MsnrApi.Repo
  alias MsnrApi.Accounts.User

 def list_users do
    Repo.all(User)
  end

  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)
  end
```

Listing 4.15: Definicija modula *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*). 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* [7]. Ova biblioteka obezbeđuje podatake 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, pokrenuti komandu *mix deps.get* iz komandne linije.

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.ActivityTypeFactory
  use MsnrApi.AssignmentFactory
  use MsnrApi.DocumentFactory
  use MsnrApi.GroupFactory
  use MsnrApi.SemesterFactory
  use MsnrApi.StudentRegistrationFactory
  use MsnrApi.StudentFactory
  use MsnrApi.TopicFactory
end
```

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 *Faker* biblioteke.

```
defmodule MsnrApi.UserFactory do
  alias MsnrApi.Queries.AccountsTest
  alias MsnrApi.Accounts.User

  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 ba-

ratati 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 pravljenju 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}
      alias Ecto.Changeset

      import Ecto.Query
      import MsnrApi.Support.DataCase
    end
  end

  setup _ do
    Ecto.Adapters.SQL.Sandbox.mode(MsnrApi.Repo, :manual)
  end
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 testa 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, u ovom slučaju to je korisnik koji je ubačen u bazu, u ovim testovima treba voditi računa i o sporednim efektima. Sporedni 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 i dohvata konkretne podatke iz baze i poredi da li je vraćeni korisnik jednak onome 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.

Pošto su testovi skupova promena iz prethodnog dela pokrili sve slučajeve greške do kojih može doći, na ovom mestu je dovoljan samo jedan test neuspešne putanje. Sve što on treba da utvrdi je postojanje greške, i da li je povratna vrednost ispravnog oblika.

```
describe "create_user/1" do

    test "success: it inserts a user in the db and returns the
   user" do

      params = Factory.string_params_for(:user)

      assert {:ok, %User{} = 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,
          "Values did not match for field: #{field}\nexpected: #{
  inspect(expected)}\nactual: #{inspect(actual)}"
      end

      assert user_from_db.inserted_at == user_from_db.updated_at
    end

    test "error: returns an error tuple when user can't be
  created" do
      missing_params = %{}

      assert {:error, %Changeset{valid?: false}} = Accounts.
  create_user(missing_params)
    end
  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 testa ove funkcije prikazana su u primeru koda 4.20. Za uspešan scenario, na početku se ubacuje jedan korisnik u bazu pomoću fabrike, kako bi nakon toga mogao biti dohvaćen. U *assert* naredbi funkciji *get_user* prosleđuje se identifikator prethodno dodatog korisnika i nakon toga se još jednom *assert* naredbom utvrđuje da li je dohvaćeni korisnik identičan postojećem.

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

```
describe "get_user/1" do

    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 tuple when a user doesn't
   exist" do

      invalid_id = -1
      assert_raise Ecto.NoResultsError, fn ->
        Accounts.get_user!(invalid_id) end
    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, a zatim se dohvataju. Očekivana povratna vrednost je lista kreiranih korisnika. U testu slučaju greške, najpre se izbriše cela tabela *users*, a zatim proveri da li će funkcija vratiti praznu listu.

```
describe "list_users/0" do

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

      assert retrieved_users = Accounts.list_users()

      assert retrieved_users == existing_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
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. Slično kao kod kreiranja, i ovde se poziva najpre *User.changeset* funkcija, pa onda i *Repo.update*. Testovi su prikazani u primeru koda 4.22. Slučaj uspešne putanje počinje ubacivanjem novog korisnika u bazu podataka pozivanjem fabrike. Nakon toga, kreira se mapa parametara, a iz nje zatim dohvata jedan ključ/vrednost par. Uzima se samo podatak o imenu korisnika, za koji je malo verovatno da će se promeniti u budućnosti. Kada bi se proveravalo ažuriranje svakog dozvoljenog polja, povećala bi se šansa da test vremenom postane zastareo. Pored toga, 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.

Test slučaj greške podrazumeva dodavanje novog korisnika u tabelu, a zatim pokušaj ažuriranja tog korisnika prosleđivanjem nevalidnih parametara. Ime korisnika se umesto neke niske postavi kao tip podataka koji opisuje datum. Dodatnu sigurnost obezbeđuje poslednja linija testa koja utvrđuje da se ništa nije zapravo promenilo u bazi.

```
describe "update_user/2" do

    test "success: it updates database and returns the user" do
      existing_user = Factory.insert(:user)
      params = Factory.string_params_for(:user)
        |> Map.take(["first_name"])
      assert {:ok, returned_user} = Accounts.update_user(existing
```

```
  _user, params)
     user_from_db = Repo.get(User, returned_user.id)
     assert returned_user == user_from_db
     expected_user_data = existing_user
       |> Map.from_struct()
       |> Map.drop([:__meta__, :updated_at])
       |> Map.put(:first_name, params["first_name"])
     for {field, expected} <- expected_user_data do
       actual = Map.get(user_from_db, field)
       assert actual == expected,
         "Values did not match for field: #{field}\nexpected: #{
  inspect(expected)}\nactual: #{inspect(actual)}"
     end
   end

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

Listing 4.22: Testiranje funkcije *update_user/2*

Poslednja CRUD operacija se odnosi na brisanje određenog reda iz tabele. Funkcija u *Accounts* modulu koja ovo obezbeđuje je *delete_user/1*. Ona ima jednu liniju u kojoj poziva *Repo.delete*. Slučaj greške je skoro nemoguć, te ovde postoji samo jedan test slučaj koji predstavlja uspešnu putanju izvršavanja, prikazan u primeru koda 4.23. Kao i svi ostali, test počinje unosom novog korisnika. Zatim poziva funkciju brisanja i očekuje kao povratnu vrednost par koji sadrži atom *:ok* i obrisanog korisnika. Povratna vrednost nakon toga nije ni potrebna, jer taj korisnik više ne postoji. Poslednja linija pomoću makroa *refute* utvrđuje da korisnika više nema u bazi podataka.

```
describe "delete_user/1" do
    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
end
```

Listing 4.23: Testiranje funkcije *delete_user/1*

Glava 5

Testiranje klijentskog dela aplikacije

Glava 6

Testiranje celokupnog sistema — End to End

6.1 Integracija klijentske i serverske strane

6.2 Testiranje opterećenja

Glava 7

Zaključak

Bibliografija

[1] on-line at: https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf.

[2] Adresa sa implementacijom portala msnr. on-line at: https://github.com/NemanjaSubotic/master-rad/tree/master/portal.

[3] ExUnit. on-line at: https://hexdocs.pm/ex_unit/ExUnit.html.

[4] Functional programming paradigm. on-line at: https://www.geeksforgeeks.org/functional-programming-paradigm/.

[5] Rest architectural style. on-line at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[6] Zvanična dokumentacija biblioteke ecto. on-line at: https://hexdocs.pm/ecto/Ecto.html.

[7] Zvanična dokumentacija biblioteke exmachina. on-line at: https://hex.pm/packages/ex_machina.

[8] Zvanična dokumentacija biblioteke faker. on-line at: https://hexdocs.pm/faker/readme.html.

[9] Zvanična dokumentacija biblioteke plug. on-line at: https://hexdocs.pm/plug/readme.html.

[10] Zvanična stranica alata hex. on-line at: https://hex.pm/.

[11] Zvanična stranica baze podataka postgresql. on-line at: https://www.postgresql.org.

[12] Zvanična stranica razvojnog okruženja phoenix. on-line at: https://www.phoenixframework.org.

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

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

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

[16] Nemanja Subotić. Programski jezici elm i elixir u razvoju studentskog veb portala, 2022. on-line at: `http://elibrary.matf.bg.ac.rs/bitstream/handle/123456789/5482/MasterRadNemanjaSubotic.pdf?sequence=1`.