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 Ime Prezime, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Ime Prezime, 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

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. U slučaju funkcionalnih programskih jezika, osnovna apstrakcija je *funkcija*. 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. Osnovna osobina čistih funkcionalnih jezika (eng. *pure functional programming language*) jeste transparentnost referenci, što kao posledicu ima nepostojanje propratnih efekata. Sa druge strane, nečisti funkcionalni jezici (eng. *impure functional programming language*) su manje elegantni jer 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.

Funkcionalni jezici zasnovani su na *lambda računu* (eng. λ-*calculus*), čija je osnovna svrha da dá definiciju izračunljivosti. Pored toga što se smatra prvim funkcionalnim jezikom, lambda račun se naziva i najmanjim programskim jezikom na svetu. Sve što se može izračunati lambda računom smatra se izračunljivim. Ekspresivnost funkcionalnih jezika ekvivalentna je ekspresivnosti Tjuringove mašine [11].

2.1 Karakteristike funkcionalnih jezika

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

Odsustvo promenljivih

Čisti funkcionalni jezici nemaju stanje koje bi se menjalo tokom izvršavanja programa, pa zbog toga ne podržavaju koncept promenljivih. Da bi postigli mutabilnost, koriste koncepte kao što su lambda izrazi ili rekurzija. 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.

Transparentnost referenci

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. Kao posledica, redosled naredbi u funkcionalnim programskim jezicima nije važan.

Čista funkcija

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

- Transparentnost referenci
- Imutabilnost

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. Kod 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

Rekurzija

Za razliku od imperativnog programiranja, 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.

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.

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 paralelnom izvršavanju, s obzirom da funkcionalni programi nemaju stanje. Takođe, daju prilično sažet i čist kôd. 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

2.2 Testiranje uopšteno (smisliti naslov)

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.

Razvoj vođen testovima (eng. *Test-Driven Development , TDD*) je praksa u razvoju softvera koja nalaže da se prvo napišu testovi. Ti testovi na početku ne prolaze, s obzirom na to da kôd još uvek nije implementiran, a zatim se iznova pokreću tokom pisanja samog koda, sve dok ne prođu. Ova tehnika kontinualnog testiranja tokom razvoja se često preporučuje jer se lako i veoma rano uoče greške i time sprečavaju u kasnijim fazama razvoja. Ipak, u ovom radu neće biti primenjen TDD pristup. Kako je tema testiranje, fokus nije na pisanju koda aplikacije, već testova za već napisan projekat. Aplikacija koja će biti testirana služi samo kao primer na kome će biti prikazani značajni koncepti testiranja funkcionalnih programa. Kratak opis testiranog projekta biće preciznije objašnjen u poglavlju 3.

Organizacija testova

Model piramide testiranja (prikazan na slici 2.2) je koncept koji pomaže u razmišljanju o tome kako testirati softver [10]. 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 (ili testovi servisa). 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 bekenda. 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 testirati 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.

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.

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

Slika 2.1: Model piramide testiranja

Ovi testovi su najjednostavniji za pisanje jer se bave malim delom aplikacije, te je kôd koji se testira najčešće vrlo jednostavan.

Pri dizajniranju jediničnih testova, nephodno je razmišljati o sledećim ciljevima:

1. Dokazati da kôd radi ono za šta je namenjen
2. Sprečiti greške — izmena jednog dela koda može izazvati grešku u nekom drugom delu
3. Utvrditi lokaciju dela koda koji izaziva grešku
4. Napisati najmanju moguću količinu koda za testiranje

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 [13]. 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, često nije neophodna u testiranju čisto funkcionalnih programa.
2. Delovanje (eng. *exercise*) — pozivanje koda koji se testira, ključni deo svakog testa.

3. Verifikacija (eng. *verify*) — često deo prethodne faze, u ovom delu testovi proveravaju ponašanje koda.

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, te jednostavno vraća ().

```
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

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 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 su samo provere nad rezultatom, i samo nad rezultatom. Testovi su veoma jednostavni jer ne moraju da brinu o propratnim efektima i njihovim neželjenim posledicama, i uvek će dobijati isti odgovor od testiranog koda, ma koliko puta bili pokrenuti.

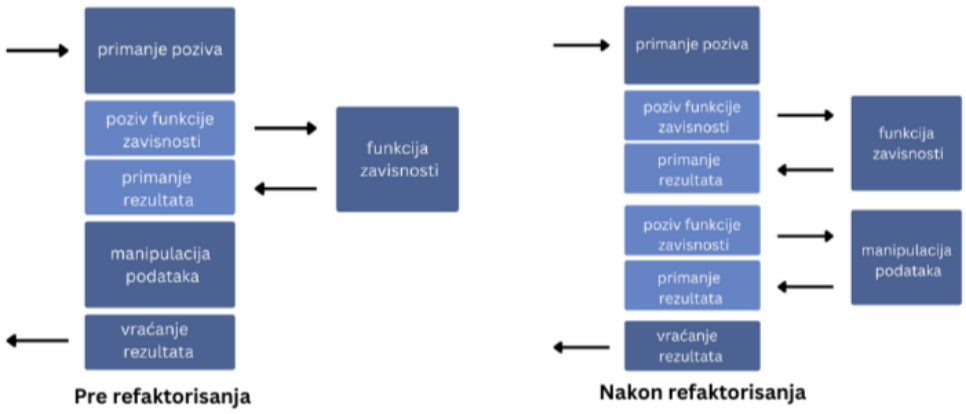
Aplikacije se u većini slučajeva neće sastojati od isključivo čistih funkcija, i u vezi sa tim postoje dve strategije [12]. Prva je izdvojiti logiku u čiste funkcije, a druga dizajnirati funkcije tako da koriste neku od metoda ubrizgavanja zavisnosti (eng. *dependency injection*) [2], što omogućava izolaciju koda.

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

[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.

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 te-

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

stove. Sa druge strane, integracioni testovi zahtevaju drugačije metode ubrizgavanja zavisnosti. TODO nastavak...

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* [14]. 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.

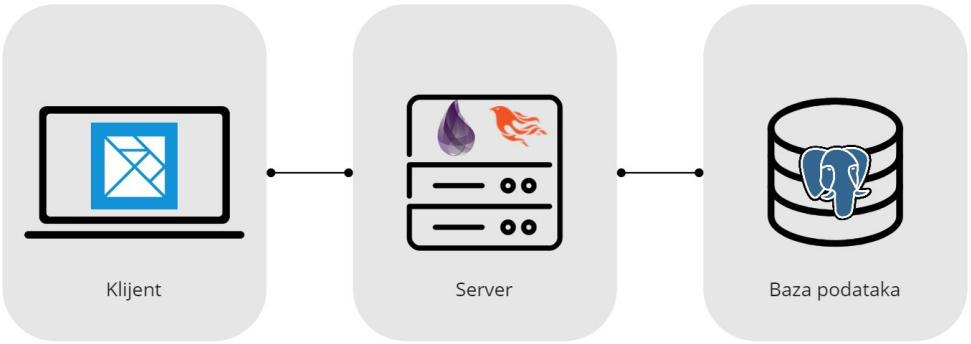
Spisak naziva entiteta i tabela u okviru baze podataka koje njima odgovaraju dati su u tabeli 3.1. Svaki od ovih entiteta, kao i relacije između njih, biće pojedinačno istestirani u narednom poglavlju.

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 *Elixir* programskom jeziku 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* [7]. Slika 3.2 prikazuje navedenu arhitekturu.



Slika 3.1: Arhitektura Portala MSNR [14]

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 Elm programskom jeziku. 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

Razvojni okvir *Phoenix* koristi se za razvoj veb aplikacija u programskom jeziku Elixir [9]. 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. 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.

Kôd napisan u Elixir-u zavisi od nekoliko Erlang biblioteka. Međutim, kada je testiranje koda u pitanju, alati i biblioteke koji se koriste su jedinstveni za svaki od ova dva jezika. Ovde će se govoriti o testiranju isključivo Elixir koda.

Zajedno sa ovim jezikom dolazi njegov razvojni okvir za testiranje, *ExUnit* [3]. U ovom poglavlju biće predstavljeni različiti koncepti testiranja u programskom jeziku Elixir, kroz pisanje testova za serverski deo aplikacije MSNR portal.

4.1 Testiranje jedinica koda u programskom jeziku Elixir

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. 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. Znati kada treba proširiti taj opseg može pozitivno uticati na jednostavnost i 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 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.

```
defmodule MyApp.ModuleTest do
   use ExUnit.Case
```

```
    describe ''thing_to_do/1'' do
     test ''it returns :ok, calls the function if the key is
    correct''
     test ''it does not call the function if the key is wrong''
    end
end
```

Listing 4.2: Opisivanje testova unutar jedne grupe

Testovi u Elixir projektima se organizuju u module i test slučajeve. U prethodnom primeru koda, modul pod nazivom *ModuleTest* se sastoji od dva test slučaja. 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.

Za svaku funkciju ili upravljač potrebno je napisati test slučajeve. 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. Korišćenje ovog makroa može se videti u primeru 4.3. Ako funkcija *hello* definisana unutar modula *Example* kao povratnu vrednost vraća atom *:world*, ovaj test prolazi.

```
defmodule HelloTest do
    use ExUnit.Case
    doctest Example

    test ''function returns expected atom'' do
      assert Example.hello() == :world
    end
end
```

Listing 4.3: Testiranje jednostavne funkcije

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. Ako se umesto *:world* u testu navede na primer atom *:word*, test ne prolazi i dobija se sledeći izlaz:

```
1) function returns expected atom (HelloTest)
     test/hello_test.exs:5
     Assertion with == failed
```

```
     code:  assert Example.hello() == :word
     left:  :world
     right: :word
     stacktrace:
       test/helllo_test.exs:6 (test)


.

Finished in 0.04 seconds
2 tests, 1 failures
```

U narednoj listi, navedeni su neki od makroa koji se koriste 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 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.

4.2 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. U ovom delu, prikazani su testovi koji proveravaju da li kôd koristi funkcionalnosti ove biblioteke na ispravan način. Dva značajna modula koja *Ecto* poseduje su *Ecto.Schema* i *Ecto.Changeset*. Prvi se odnosi na definisanje 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.

Svi testovi koji se odnose na *Ecto* biblioteku nalaze se u direktorijumu *'/test/msnr_api/schema'*, 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 4.4. Pomoću makroa *schema* i *field* definišu se tabele, njihova polja i relacije sa drugim tabelama. Oni istovremeno definišu i Elixir strukturu — u slučaju primera 4.4, ta struktura se naziva *Assignment*.

```
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 4.4: Shema tabele assignments

Testiranje polja i tipova

Test koji se odnosi na prethodni primer prikazan je u kodu 4.5. 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 4.4. 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
  use ExUnit.Case
  alias MsnrApi.Assignments.Assignment

  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 4.5: Test za proveru polja i tipova tabele assignments

Testiranje skupa promena

Funkcija *changeset* 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.

Testovi koji se odnose na ove funkcije implementirani su unutar describe bloka ”changeset/2”. Oni pokrivaju i uspešan scenario, i neke od slučajeva greški. Funkcija *changeset/2* iz primera koda 4.4, koja kao argumente prihvata strukturu *Assignment* i listu atributa, ima ulogu da validira prisustvo dva polja u tabeli *assignments* —

polja *comment* i *grade*. Testovi koji proveravaju ispravnost ove funkcije dati su u primeru koda 4.6.

```
describe "changeset/2" do
  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

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[:validaiton]} is incorrect.
```

```
    "
      end
    end

    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

end
```

Listing 4.6: Testovi za proveru funkcije changeset/2

4.3 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.

Test dubleri

Neki podnaslov

U aplikaciji MSNR portal — TODO konkretni primeri ...

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 stranica baze podataka postgresql. on-line at: https://www.postgresql.org.

[8] Zvanična stranica programskog jezika Elixir. on-line at: https://elixir-lang.org/.

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

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

[11] Liesbeth De Mol. Turing Machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2021 edition, 2021.

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

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

[14] 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`.