Razvoj web aplikacije pokretan testiranjem

Turković, Michael

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:358853

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

zir.nsk.hr

Image not found or type unknown

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij

RAZVOJ WEB APLIKACIJE POKRETAN TESTIRANJEM

Diplomski rad

Michael Turković

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 21.09.2018.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Michael Turković
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D 890 R, 22.09.2017.
OIB studenta:	02073606192
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Krešimir Nenadić
Član Povjerenstva:	Izv. prof. dr. sc. Alfonzo Baumgartner
Naslov diplomskog rada:	Razvoj web aplikacije pokretan testiranjem
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu rad potrebno je analizirati i opisati mogućnosti razvoja pokretanog testiranjem. U praktičnom dijelu rada treba primijeniti navedeni pristup za izradu web aplikacije u okolini po izboru i prema prikladnom uzorku razvoja programske podrške. Nadalje, potrebno je provesti i analizirati testiranje na razini cjelina koda (unit testing), integracijskog i regresijskog testiranja, testiranja sustava, te nefunkcionalnih testova.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	21.09.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 26.09.2018.

Ime i prezime studenta:	Michael Turković
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D 890 R, 22.09.2017.
Ephorus podudaranje [%]:	7

Ovom izjavom izjavljujem da je rad pod nazivom**: Razvoj web aplikacije pokretan testiranjem**

izrađen pod vodstvom mentora Prof.dr.sc. Goran Martinović

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1

2. TESTIRANJE PROGRAMA 2

2.1. Ciljevi testiranja 3

2.2. Razlozi testiranja 5

2.3. Proces testiranja 7

2.3.1. Planiranje testiranja 8

2.3.2. Dizajn testova 8

2.3.3. Izvršavanje testova 9

2.3.4. Prijava i rješavanje problema 9

2.3.5. Prateće aktivnosti 9

2.4. Testiranje kao dio razvoja programa 10

2.4.1. Testiranje modelom vodopada 10

2.4.2. Testiranje V - modelom 11

2.4.3. Testiranje agilnim modelima 13

2.5. Metode testiranja 13

2.5.1. Ručno testiranje 14

2.5.2. Automatsko testiranje 14

2.5.3. Istraživačko testiranje 15

2.5.4. Regresijsko testiranje 16

2.6. Razine testiranja 16

2.6.1. Jedinično testiranje 16

2.6.2. Integracijsko testiranje 17

2.6.3. Sustavsko testiranje 19

2.7. Strategije testiranja 19

2.7.1. Bijela kutija 19

2.7.2. Crna kutija 20

2.8. Vrste testiranja 21

2.8.1. Funkcionalno testiranje 21

2.8.2. Nefunkcionalno testiranje 22

3. RAZVOJ POKRETAN TESTIRANJEM 23

3.1. Razvoj pokretan testiranjem kod web aplikacija 25

4. PRIMJENA TESTIRANJA NA PRIMJERU WEB APLIKACIJE 28

4.1. Zahtjevi sustava 28

4.2. Opis korištenih tehnologija i alata 28

4.3. Razvijeno programsko rješenje 31

4.1.1. Model strojnog učenja 38

4.3. Primjena i rezultati testova 41

4.3.1. Ručno testiranje 41

4.3.2. Testiranje poslužiteljske strane 43

4.3.3. Testiranje klijentske strane 51

4.3.4. Funkcionalno testiranje 56

5. ZAKLJUČAK 61

LITERATURA 62

SAŽETAK 64

ABSTRACT 65

ŽIVOTOPIS 66

PRILOZI 67

1. UVOD

Testiranje može predstavljati vrlo složen proces prilikom razvoja programske podrške, zbog rizika koji dovodi do velikih gubitaka ukoliko isporučeni program sadrži greške. Planiranje testiranja trebalo bi početi u ranoj fazi, prilikom definiranja zahtijeva. To znači da testni planovi i procedure moraju biti sustavno i kontinuirano razvijani i po potrebi redefinirani. Greškom se može smatrati i pogrešan potez koji korisnik sustava može napraviti, nakon čega sustav daje pogrešan rezultat, dok se neuspjeh definira kao odstupanje ponašanja programa u odnosu na zahtjeve ili specifikaciju proizvoda. Testiranjem programa može se otkriti neuspjeh, ali su greške ono što može i mora biti uklonjeno.

U drugom poglavlju obraditi će se ciljevi i razlozi testiranja, pobliže će se objasniti proces testiranja, te testiranja kao dijela razvoja programa. Objasniti će se metode, razine, strategije i vrste testiranja, a neke od njih će se primijeniti prilikom razvoja web aplikacije. Treće poglavlje opisuje razvoj pokretan testiranjem gdje će se navesti prednosti uz potrebne korake prilikom ovakvoga pristupa razvoja. U četvrtom poglavlju opisati će se razvijeno programsko rješenje i korištene tehnologije. Nakon upoznavanja s aplikacijom, prijeći će se na implementaciju i rezultate testove. Razvoj je kao i testiranje podijeljeno na poslužiteljsku i klijentsku stranu. Prilikom razvoja određenih funkcionalnosti poslužiteljske strane testiranje se obavljalo ručno, radi jednostavnosti i brže implementacije. Poslužiteljska strana pokrivena je integracijskim i jediničnim testovima, dok su na klijentskoj strani implementirani jedinični testovi. Implementacija testova prikazana je i na primjerima koda, rezultatima te izvješćima pokrivenosti koda. Funkcionalnim testiranjem se prikazao rad web aplikacije u stvarnom okruženju.

2. TESTIRANJE PROGRAMA

Testiranje programske podrške predstavlja aktivnost koja obuhvaća kompletan proces razvoja i održavanja i kao takav čini važan dio konstrukcije programa. To nije aktivnost koja se provodi nakon kompletiranja faze razvoja, već se izvodi zbog evaluacije kvalitete proizvoda i njegovog poboljšanja, putem identifikacije grešaka tijekom razvoja. Može se reći da je to proces vrednovanja sustava ili njegovih komponenti s namjerom da se utvrdi zadovoljava li navedene zahtjeva ili ne [1]. Kvaliteta bilo kojeg programa može se prepoznati samo putem testiranja. Kroz napredovanje tehnologije širom svijeta, povećao se broj tehnika i metoda vrednovanja kako bi testirali programsku podršku prije nego ode u produkciju i od nje do tržišta. Testiranje automatizacije utjecalo je na proces testiranja. Danas se većina testiranja programske podrške obavlja alatom za automatizaciju koje ne samo da smanjuje broj ljudi koji rade oko tog programa, već i pogreške koje se mogu ukloniti [2]. Greške nastale prilikom rada su normalna i svakodnevna stvar i nisu usko vezane za izradu. One se događaju u svim područjima rada i uzrokovane su kako ljudskim, tako i računalnim faktorima. Budući da su one realna pojava, u svim područjima rada uvode se sustavi kontrole kvalitete kojima se greške identificiraju i uklanjaju prije nego što se izazovu veći problemi u radu [3].

Program se implementira prema korisničkim zahtjevima kojima se rješava određeni realni problem ili se implementira korisna funkcionalnost potrebna krajnjim korisnicima. Tijekom implementacije, može se u većoj ili manjoj mjeri odgovarati originalnim zahtjevima prema kojima je izrađen. Svako ponašanje programa koje se ne slaže s originalnim zahtjevima predstavlja grešku koju je potrebno identificirati i otkloniti. U širem smislu, testiranje predstavlja sustav kontrole kvalitete kojim se ne provjerava samo program već i sve njegove pripadajuće komponente i karakteristike. Kvaliteta programske podrške prema [3] može se definirati na različite načine:

- Kompatibilnost s zahtjevima i potrebama korisnika – jedan od najbitnijih uvjeta da bi se program odredio kao kvalitetan je da olakšava rad krajnjim korisnicima
- Dobre karakteristike proizvoda kao što su brzina rada, malo zauzeće memorije, brzina pokretanja
- Lakoća održavanja i promjena u programu
- Kvalitetna dokumentacija, zahtjevi, dizajn
- Kompatibilnost s standardima – podrazumijeva usklađenost s općim standardima (ISO), usklađenost s zakonima i slično

- Rad u ekstremnim uvjetima s ogromnom količinom podataka, te slabim vezama i ograničenim resursima

Budući da programi i svi digitalni sustavi nisu kontinuirani, testiranje graničnih vrijednosti nije dodatno jamstvo za ispravnost. Sve moguće vrijednosti moraju biti testirane i provjerene, ali potpuno testiranje nije moguće. Broj mogućih testova za pojedine komponente gotovo je beskonačan, i svako testiranje koristi neku strategiju za odabir testova koji su izvedivi za raspoloživo vrijeme i resurse. Kao rezultat toga, sam taj posao je iterativni proces, tako da se prilikom rješavanja jednog problema, mogu se pronaći dublje greške ili čak pojaviti nove [4]. Ne uzimajući u obzir kako definiramo kvalitetu, svaka definicija predstavlja skup nefunkcionalnih zahtjeva kojim se opisuje šta se od određenog programa očekuje. Cilj testiranja u svrhu kontrole kvalitete je upravo provjera da li su ovi zahtjevi ispunjeni. Primjer provjere kvalitete programske podrške danas uključuje i provjeru smislenosti originalnih zahtjeva, jasnoće specifikacije i usklađenosti s originalnim zahtjevima, jednostavnosti korištenja i slično. Danas se u manjoj mjeri govori o testiranju kao posebnoj aktivnosti provjere funkcionalnosti te se teži općoj kontroli kvalitete svih komponenti sustava [3].

2.1. Ciljevi testiranja

Bez obzira na ograničenja, proces testiranja se može smatrati kao sastavni dio razvoja programske podrške. Široko je implementirano u svakoj fazi ciklusa razvoja. Više od 50% vremena razvoja potrošeno je u testiranju, a ono se izvodi se u sljedeće svrhe:

- Poboljšanje kvalitete:

  Kvaliteta znači usklađenost s navedenim zahtjevom za projektiranje. Točnije, minimalni zahtjev kvalitete, znači obavljanje prema potrebi u određenim okolnostima. Ispravljanje pogrešaka, točnije testiranje programa, izvodi se kako bi se otkrili nedostatci u dizajnu od strane programera. Nepravilnost ljudske prirode gotovo je nemoguće izjednačiti s umjereno složenim programom. Pronalaženje problema i njihovo fiksiranje je svrha uklanjanja pogrešaka u fazi programiranja [5].
- Verifikacija i validacija:

  Verifikacija je proces evaluacije programa kako bismo utvrdili da proizvod u određenoj fazi zadovoljava zadane uvjete, odgovara na pitanje: „*Jesmo li na pravom putu stvaranja konačnog proizvoda?*“. Navedenim procesom utvrđujemo da naš proizvod pokriva sve funkcionalnosti prema funkcionalnoj specifikaciji [6].

Validacija je proces evaluacije konačnog proizvoda kako bi se provjerilo zadovoljava li program određene definirane zahtjeve. Navedenim procesom utvrđujemo da sustav odgovara zadanim zahtjevima i odrađuje sve zadane funkcije. Odgovara na pitanje: „*Da li je proizvod koji stvaramo dobar?*“ [6].

Tablica 2.1. Razlike između verifikacije i validacije [6]

VERIFIKACIJA	VALIDACIJA
Da li dobro gradimo proizvod?	Gradimo li dobar proizvod?
Provjerava ispunjava li specifične zahtjeve u određenoj fazi.	Ocjenjuje konačni proizvod kako bi se provjerilo ispunjava li poslovne potrebe.
Provjerava je li proizvod izgrađen prema određenom zahtjevu i specifikaciji dizajna.	Određuje da li je program pogodan za uporabu.
Provjerava se bez izvođenja programa.	Provjerava se s izvođenjem programa.
Uključuje sve statičke tehnike ispitivanja.	Uključuje sve dinamičke tehnike ispitivanja.
Uključuje planove, specifikacije zahtjeva, specifikacije dizajna, test slučajeve itd.	Uključuje sve vrste tipova testiranja, regresijsko, funkcionalno, sistemsko testiranje itd.

- Procjenu pouzdanosti:

  Pouzdanost programske podrške ima važne odnose s različitim aspektima, podrazumijevajući strukturu i količinu testiranja na koju je podvrgnut. Na osnovu operativnog profila (procjena relativne učestalosti korištenja različitih ulaza u program), testiranje može poslužiti kao statička metoda uzorkovanja za dobivanje podataka o neuspjelim procjenama pouzdanosti. Testiranje nije zrelo, još uvijek ostaje umjetnost, jer još uvijek ne možemo napraviti znanost. Još uvijek se koriste iste tehnike testiranje koje su nastale prije 15 - 20 godina, od kojih su neke izrađene metode, a ne dobre inženjerske metode. Testiranje može biti skupo, ali može biti još skuplje, osobito na mjestima u kojima je u pitanju ljudski život. Nikad ne možemo biti sigurni da je komad programa točan. Nikad ne možemo biti sigurni da su specifikacije točne. Nijedan sustav potvrde ne može potvrditi svaki ispravan program. Nikad ne možemo biti sigurni da je sustav provjere također točan [7].

2.2. Razlozi testiranja

Tvrtke koje se bave razvojem programa suočavaju se s gubicima upravo zbog velikog broja grešaka u isporučenom proizvodu. Štete pojedinih grešaka mogu imati ogromne posljedice s obzirom na to da se računala i programi koriste u različitim situacijama. One mogu prouzrokovati ogromne gubitke. Kroz povijest, programske greške su u određenim sustavima uzrokovale katastrofe. U modernom društvu pouzdanost i kvaliteta imaju presudnu ulogu u procesu razvoja programske podrške. Zbog nesavršenosti ljudske prirode gotovo je nemoguće napisati složeniji program bez ijedne pogreške. Pronalazak određenih problema i njihovo otklanjanje je osnovna zadaća tijekom faze razvoja. Kao glavni zadatak, podrazumijeva se otkrivanje grešaka u programu s ciljem da se otkloni prije isporuke proizvoda klijentu. Također je potrebno identificirati i pratiti uočeni problem do mjesta nastanka greške [8].

Važno je postići optimalne rezultate ispitivanjem dok se provodi testiranje bez odstupanja od cilja. Kako bi odredili pravu strategiju testiranja, moramo se pridržavati nekih osnovnih načela testiranja. Prema [9] ovo su uobičajenih sedam načela testiranja koja se široko primjenjuju u programskoj industriji:

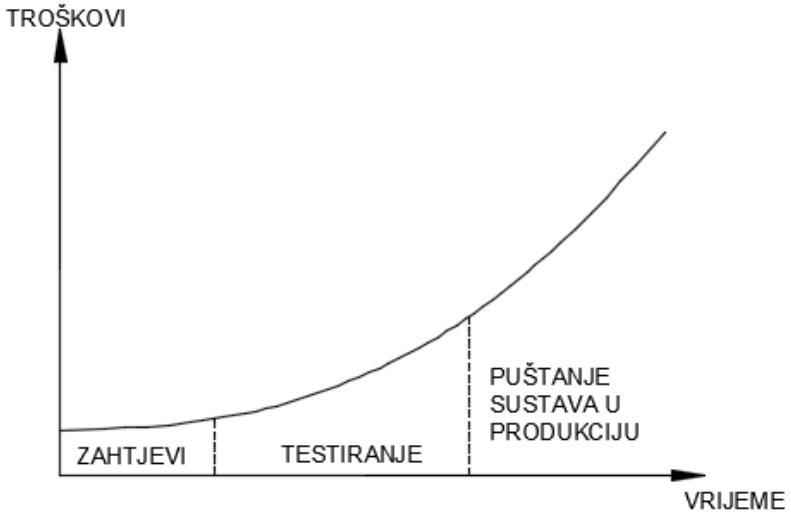
1. Iscrpno ispitivanje nije moguće - umjesto iscrpnog testiranja, potrebna je optimalna količina testiranja na temelju procjene rizika aplikacije.
2. Testiranje povezanih grešaka (engl. *Defect clustering*) - grupiranje grešaka koje navodi da mali broj modula sadrži većinu otkrivenih nedostataka. Ovo je primjena načela „*Pareto*“, na testiranje programa, koje kaže da se oko 80% problema se nalazi u 20% modula. Moguće je identificirati rizične module, ali ovaj pristup ima svoje vlastite probleme. Ako se ista ispitivanja ponavljaju iznova i iznova, na kraju isti testni slučajevi više neće pronaći nove greške.
3. Povezanost grešaka u modulima (engl. *Pesticide paradox*) - ako se provede isti niz ponovljenih testova, metoda će biti beskorisna za otkrivanje novih nedostataka. Kako bi se to prevladalo, testni slučajevi trebaju se redovito pregledavati i revidirati, dodajući nove i različite testne slučajeve kako bi pronašli više nedostataka. Test inženjeri ne mogu jednostavno ovisiti o postojećim ispitnim tehnikama. Moraju paziti da neprestano poboljšavaju postojeće metode kako bi testiranje bilo učinkovitije. No, čak i ako nakon svih ovih napora, nikada se ne može tvrditi da je proizvod bez grešaka.
4. Testiranjem dokazujemo postojanost grešaka - načelo testiranja navodi da, testiranje govori o prisutnosti nedostataka i ne govori o nedostatku nedostataka, tj. testiranje

programa smanjuje vjerojatnost neotkrivenih nedostataka koji ostaju u programu, ali čak i ako nema nedostataka, to nije dokaz ispravnosti.

5. Zabluda nepostojanja grešaka - moguće je da program koji je 99% bez grešaka još uvijek neupotrebljiv. To može biti slučaj ako je sustav temeljito ispitan zbog pogrešnog zahtjeva. Testiranje nije puko pronalaženje nedostataka, već i provjeru da li program rješava poslovne potrebe. Odsutnost pogreške je pogrešna, tj. pronalaženje i popravljanje nedostataka ne pomaže ako je izgradnja sustava neupotrebljiva i ne zadovoljava potrebe i zahtjeve korisnika.
6. Testiranje u ranoj fazi razvoja - testiranje bi trebalo početi što je prije moguće u životnom ciklusu razvoja programa. Tako da su svi nedostatci u zahtjevima ili fazi projektiranja snimljeni u ranoj fazi. Puno je jeftinije popraviti grešku u ranim fazama ispitivanja. Ali koliko ranije treba početi testirati? Preporuča se da se započne pronalaženje u trenutku definiranja zahtjeva.
7. Testiranje ovisi o kontekstu (pristup testiranja prema vrsti projekta) - što u osnovi znači da će način testiranja za različite programe biti različit. Svi razvijeni programi nisu identični. Može se koristiti drugačiji pristup, drugačije metodologije, tehnike i vrste testiranja, ovisno o vrsti aplikacije

Troškovi uklanjanja uočenih grešaka, umjesto pokušaja sprječavanja njihovog nastanka su veliki i prouzrokuju ogromni gubitak u poslovanju. Prije nego što dođe do završne faze implementacije komponenti programa, potrebno je preventivno djelovati na nastanak grešaka, kako bi se izbjegla kašnjenja i uvećali troškovi njihovog otklanjanja [8].

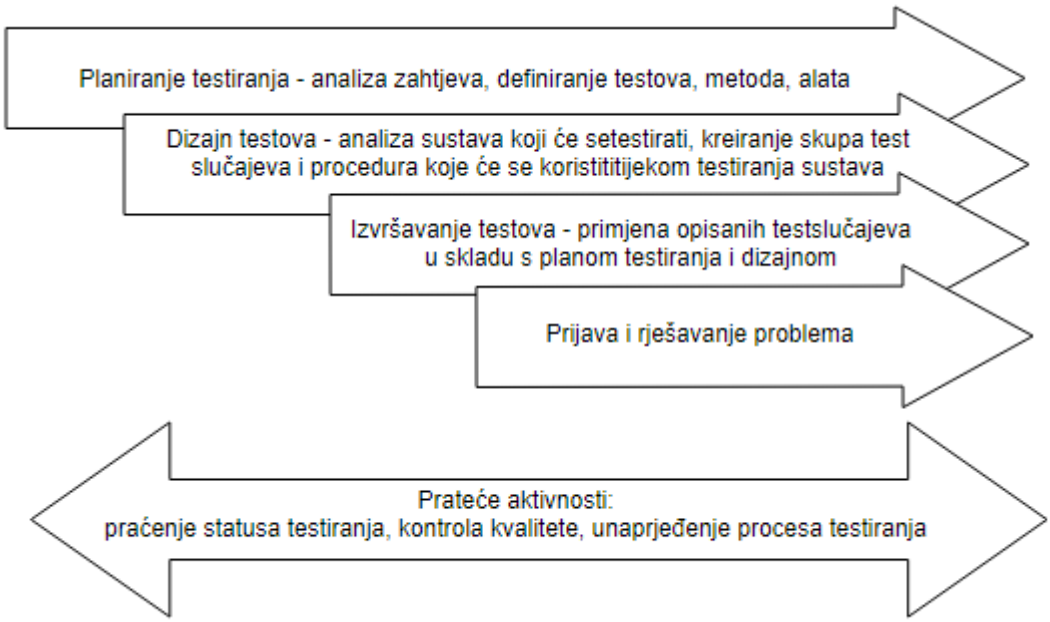
Boehmova krivulja prikazana na slici 2.1 govori da su greške uvijek skuplje za popravak u kasnijim fazama procesa. U osnovi navodi da se treba ići naprijed i popraviti što je moguće više grešaka tijekom zahtjeva i testiranja faza projekta, jer nakon implementacije, greške masivno rastu s povećanjem troškova.



Slika 2.1. Ovisnost troškova i faze u kojoj je greška prijavljena [8]

2.3. Proces testiranja

Kao što je već spomenuto, testiranje je proces koji obuhvaća velik broj određenih aktivnosti i usko je povezan s procesom razvoja programske podrške. U ranijim fazama projekta, koje predstavlja upoznavanjem s projektom i problemima koje je potrebno riješiti, pravi se plan što će se testirati i što je sve potrebno, a to predstavlja osnovu za dalji rad. Planiranje ne završava na početku projekta – ovisno o potrebama i zahtjevima na projektu, ono se nastavlja do kraja [3]. Slika 2.2 prikazuje aktivnosti u procesu testiranja tijekom izvođenja projekta.



Slika 2.2. Aktivnosti u procesu testiranja tijekom izvođenja projekta [3]

Nakon što se definiraju zahtjevi, prelazi se na proces dizajna testova. Dizajn testova određuje način na koji će se testirati program. Kao i kod planiranja, on se provodi sve do kraja projekta pošto svaka promjena ili prepravak sustava uključuje izmjenu testova. Provođenje testova i rješavanje problema vrši se u onom trenutku kada su na raspolaganju dijelovi sustava koje je moguće testirati. Paralelno s ovim aktivnostima izvršavaju se i prateće aktivnosti, kao što su sam procesa testiranja, evaluacija i kontrola kvalitete testiranja, unaprjeđenje procesa i slično [3].

2.3.1. Planiranje testiranja

Planiranje predstavlja osnovni korak u svakom procesu rada, to je priprema za cijeli proces testiranja i služi kako bi tim odlučio što je potrebno uraditi i na koji način. Tijekom planiranja definiraju se vrste testova koje će se provesti, metode testiranja, strategije testiranja, kao i kriterij završetka testiranja. Kao rezultat, dobije se skup dokumenata koji predstavljaju općeniti pogled na sustav, aktivnosti koje će biti provedene, te strategije i alate [3].

Pravljenje plana je aktivnost kojom se trenutno stanje plana i trenutne aktivnosti koje su identificirane dokumentom. Za razliku od pravljenja plana što je posebna aktivnost dokumentacije, planiranje se nastavlja izvoditi tijekom projekta daljnjim pogledom na probleme i pogodnosti koje nisu primijećene na početku, kako bi se poboljšao planirani proces. Taj proces se nastavlja i nakon pravljenja plana i prepliće se s ostalim aktivnostima upravljanja projekta [3].

2.3.2. Dizajn testova

Nakon definiranja potrebnih planova, pristupa se detaljnoj specifikaciji te načinima na kojima će se aktivnosti predviđene već definiranim planom izvršiti, definiraju se konkretne smjernice kako će se izvršavati testiranje. Tijekom ove faze analizira se sustav koji će biti testiran, identificiraju se problemi koji će biti testirani. Kao rezultat ove aktivnosti, definira se skup test slučajeva i procedura koje će biti korištene [3].

Proces dizajna testova predstavlja sve aktivnosti koje se moraju izvršiti kako bi se utvrdili načini pomoću kojih će se sustav testirati. Prema [3] aktivnosti koje su obuhvaćene su:

- Analiza korisničkih zahtjeva
- Validacija korisničkih zahtjeva i specifikacija
- Specifikacije test dizajna

2.3.3. Izvršavanje testova

Izvršavanje testova definiramo kao proces primjene određenih test slučajeva i procedura u skladu s planom i dizajnom. U slučaju da je taj proces neplanski i bez specifikacija, to predstavlja aktivnost u kojoj tim koji testira dolazi u kontakt s programskim sustavom s ciljem pronalaska greške. Ne uzimajući u obzir da li je ovakvo testiranje uspješno ili ne, uvijek postoji rizik da su propuštene neke važne funkcionalnosti [3].

2.3.4. Prijava i rješavanje problema

U „zrelijim“ procesima unaprijed je definirano što će biti testirano i na koji način. Planom je definirano kako će se primijeniti odstupanja u testovima i slično. Samo izvršavanje testova prate dvije dodatne aktivnosti prema [3] to su:

- Praćenje statusa problema - izvršava se paralelno s fazom izvršavanja testova i obuhvaća kontrolu prijavljenih problema, što uključuje i samo rješavanje problema
- Izvještavanje – završna aktivnost u procesu testiranja, kreiraju se izvještaji kojima je opisan proces testiranja i potvrđuje da je program spreman za korištenje u skladu s kriterijima kvalitete definiranim u planu

2.3.5. Prateće aktivnosti

Paralelno s navedenim aktivnostima izvršavaju se i dodatne aktivnosti kojima se kontrolira proces testiranja i analizira način na koji se neki dijelovi mogu poboljšati. Neke od pratećih aktivnosti prema [3] su:

- Praćenje i kontrola procesa testiranja - predstavlja skup aktivnosti kojima pratimo da li se sve aktivnosti izvršavaju po planu. Ovaj korak predstavlja vrlo važan dio, pošto nekada trajanje izvršavanja testova može prekoračiti rokove, pojavljivanje velikog broja nedostataka koje je potrebno ponovno testirati
- Kontrola kvalitete testiranja – predstavlja provjeru da li se kontrola kvalitete izvršava na zadovoljavajući način. Kod procesa testiranja nije dovoljno samo zadovoljiti određene zahtjeve i funkcionalnosti. Bitno se uvjeriti da je sustav testiran na dovoljno dobar način i da možemo biti sigurni da isporučujemo kvalitetan proizvod.
- Kontrola promjena – aktivnost u kojoj se prate promjene u sustavu i analiziraju promjene koje bi trebalo izvoditi u testovima. Kontrola inačice testova, usklađenost inačice testova s inačicama programskog koda.

- Poboljšanje procesa testiranja – predstavlja skup određenih mjera pomoću kojih se na osnovu rezultata kontrole provode određene promjene.

Prateće aktivnosti se definiraju tijekom plana i ovise od potreba projekta, pravila i organizacije [3].

2.4. Testiranje kao dio razvoja programa

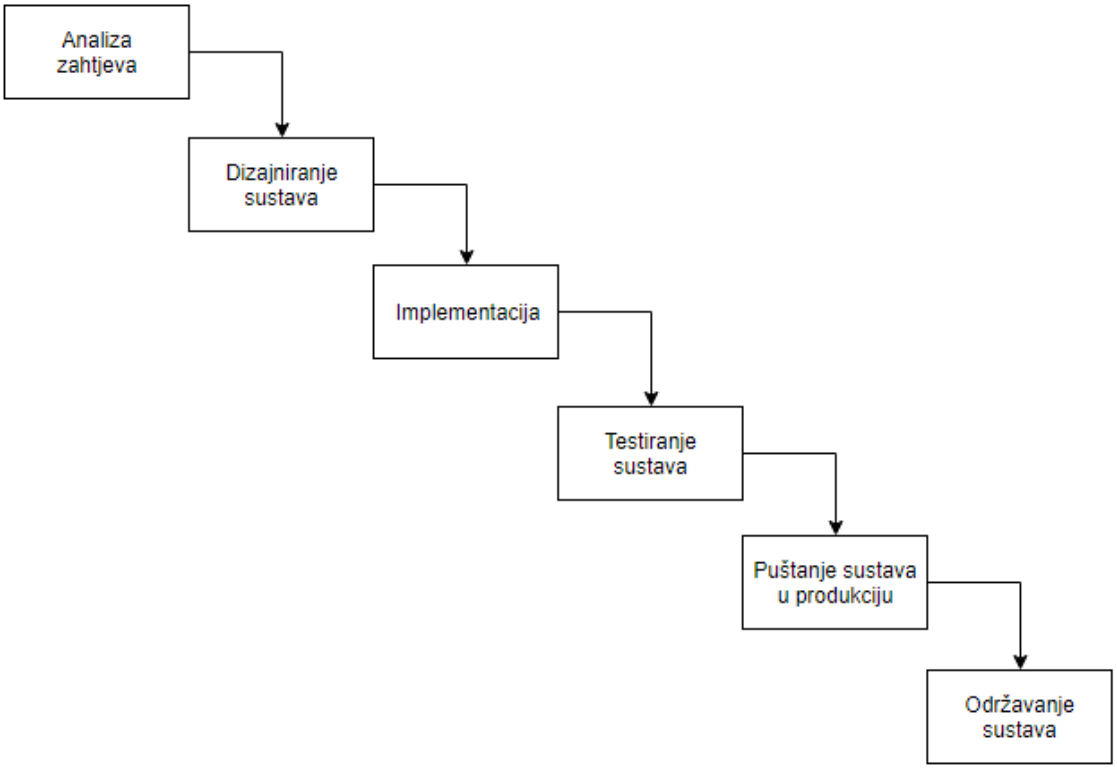
Do sada spomenuti proces testiranja ne upotrebljavaju se u svim slučajevima, nego u ovisnosti od uvjeta prilagođava se ostalim procesima. S obzirom da je proces razvoja programske podrške primarni proces, proces testiranja mu se mora prilagoditi, kako bi se aktivnosti iz procesa testiranja što uspješnije uklopile u aktivnosti razvoja. Zbog toga je potrebno znati na koji način se razvija program i u kojim fazama projekta se vrše aktivnosti testiranja. Sam razvoj programa uključuje veliki broj aktivnosti od analize zahtjeva koji trebaju biti implementirani, dizajna rješenja, programiranja, testiranja, pa sve do konačnog proizvoda koji se predaje korisnicima. Način na koji se organiziraju aktivnosti naziva se životni ciklus razvoja programa [3].

Postoji puno različitih modela razvoja programa kojima se olakšava organiziranje spomenutih aktivnosti, ali prema [3] te modele dijelimo na:

- Fazni model – aktivnosti razvoja dijele se u grupe, koje se izvršavaju serijski po fazama
- Iterativni model – projekt se dijeli na manji broj perioda u kojima se vrše aktivnosti u razvoju
- Agilne metode – slične iterativnima, ali na manje formalan način kombiniraju aktivnosti razvoja programa

2.4.1. Testiranje modelom vodopada

Model vodopada jedan je od prvih modela razvoja programa zasnovan na linearnom pristupu razvoja programske podrške. Osnova ideja je podjela na pojedine faze koje predstavljaju skupove sličnih aktivnosti. Određena faza traje sve dok se ne završe aktivnosti u okviru nje i ne može se prijeći u sljedeću fazu dok prethodna nije završena. Da bi se razvio program, prvo se moraju razumjeti zahtjevi. Rezultat ove faze je specifikacija zahtjeva. Kada su zahtjevi potvrđeni, moraju se definirati faze implementacije. Kada je sustav završen, potrebno je provesti testove kako bi se otklonile postojeće greške. Nakon što su zadovoljene sve faze sustav je spreman za predaju krajnjim korisnicima [3]. Slika 2.3 prikazuje korake prilikom razvoja programske podrške modelom vodopada.

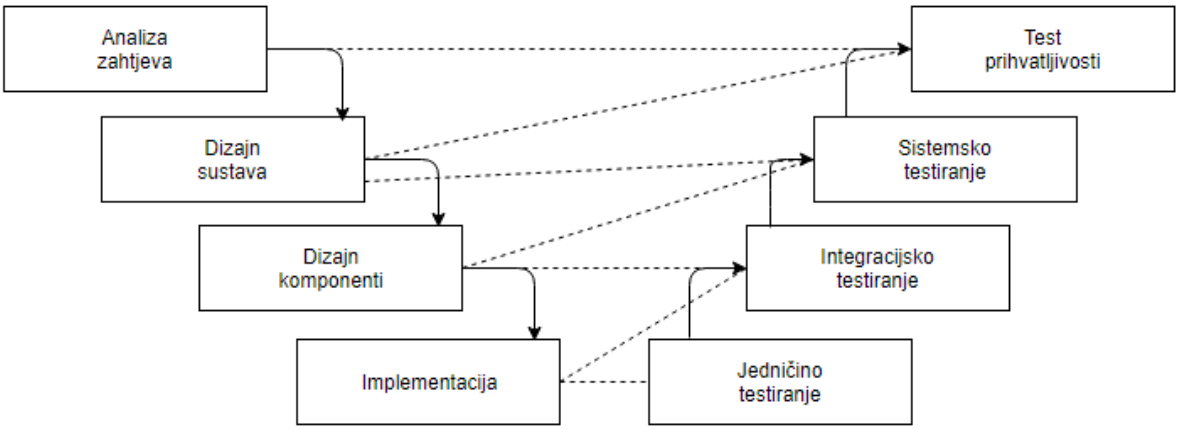


Slika 2.3. Razvoj programa modelom vodopada [6]

Projekti koji se implementiraju modelom vodopada, testiranje se izvodi tijekom faze verifikacije. U ovoj fazi testiranje obuhvaća analizu funkcionalnih zahtjeva, definiranje test planova, testiranje programa i prijavljivanje problema. Paralelno s ovom aktivnosti angažirani su i sami programeri koji rješavaju prijavljene probleme. Nakon ove faze potvrđuje se da je u sustavu nema problema. Prednost ovakvog pristupa je u tome što se definira faza testiranja programa, što olakšava planiranje testova. Mana ovakvoga pristupa leži u pretpostavci da faza verifikacije služi samo da bi se utvrdilo da sustav radi ispravno. Međutim, tijekom ove faze se mogu pronaći ozbiljni problemi koji mogu projekt vratiti na početak. Povratni tokovi u modelu vodopada često su uvjetovani samim kašnjenjem projekta, a u nekim slučajevima čak i neuspjehom projekta [3].

2.4.2. Testiranje V - modelom

V-model predstavlja modifikaciju standardnog modela vodopada. U V-modelu postoji odgovarajuća faza testiranja za svaku fazu razvoja programa, ispitivanje se vrši paralelno s fazom razvoja [9]. Slika 2.4 prikazuje dijagram aktivnosti prilikom razvoja programske podrške pomoću V – modela.



Slika 2.4. Razvoj programa V - modelom [3]

Kao i kod modela vodopada, V-model ima faze analize, implementiranja i testiranja gdje se u sljedeću fazu prelazi samo ako je izvršena prethodna, ali prema [3] uz dva značajna ograničenja:

- Poslije svake faze, prije prelaska u sljedeću fazu razvoja, pravi se plan provjere trenutne faze
- Uvjet za prelazak u sljedeću fazu je definiranje načina na koji će biti testirani rezultati prethodne faze

Osnovna ideja V-modela govori da iako nije moguće provjeriti određenu fazu, može se isplanirati proces provjere. Samim planiranjem provjere mogu se otkriti problemi u fazi razvoja. Na primjer, ako je definirana lista zahtjeva, i počne planiranje testiranja zahtjeva i za neke od zahtjeva se zaključi da su nejasni i ne mogu se testirati. U tome slučaju, tim se vraća u fazu analize zahtjeva, kako bi se razjasnio zahtjev. Na ovaj način se sprječava problem koji se javlja u modelu vodopada, gdje se tek nakon implementacije zahtjeva dolazi do zaključka da je zahtjev nejasan, te se sve vraća na početak. Veća je vjerojatnost da će rezultati pojedinih faza biti spremniji za sljedeću fazu [3].

U ovome modelu postoji nekoliko pravila kojima se određuju uvjeti za prelazak u sljedeću fazu prema [3] to su:

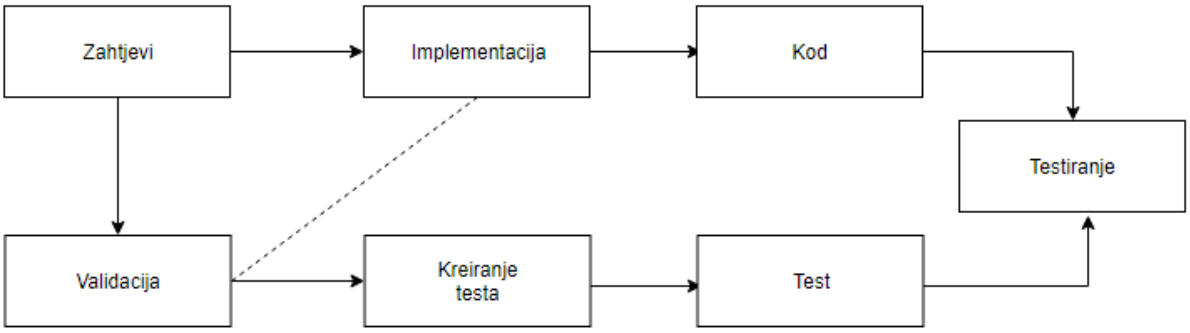
- Iz faze analize zahtjeva prelazi se u fazu dizajna samo ako su analizirani zahtjevi i ako su definirani načini kako će se zahtjevi testirati
- U fazu dizajna arhitekture može se prijeći ako je završena faza sustavskog dizajna i definiran način testiranja cjelokupnog sustav

- U fazu dizajna može se prijeći ako je dizajnirana arhitektura i definiran način testiranja komponenti tijekom implementacije
- U fazu implementacije može se prijeći ako su dizajnirani moduli koji će biti implementirati te ako je definiran način njihovog testiranja

U pogledu testiranja veliki je napredak u činjenici da su aktivnosti testiranja uključene od samoga početka i da je testiranje uvjet za nastavak projekta [3].

2.4.3. Testiranje agilnim modelima

Agilni modeli razvoja predstavljaju novi pristup razvoju programa, gdje se precizno definiraju potrebne faze i aktivnosti. Test aktivnosti se vrše kontinuirano, ne postoji trenutak početka testiranja pošto se analiza zahtjeva i testiranje vrše paralelno s razvojem. Na početku svake faze prije nego što se krene s implementacijom, proučavaju se zahtjevi. Tijekom faze implementacije zahtjeva, paralelno se vrši kreiranje testova [3]. Slika 2.5 prikazuje aktivnosti prilikom razvoja programske podrške agilnim modelom.



Slika 2.5. Testiranje agilnim modelima [3]

U agilnim metodama cijeli proces se prilagođava i po potrebi modificira prema promjenama koje će se dogoditi tijekom projekta.

2.5. Metode testiranja

Metode testiranja predstavljaju načine testiranja sustava. Definicija ovih metoda testiranja predstavlja strategiju testa. U planu je potrebno definirati metode koje će biti korištene kako bi se znalo koliko je vremena potrebno izdvojiti na testiranje, te koji je potreban profil ljudi [3].

2.5.1. Ručno testiranje

Ručno testiranje često predstavlja osnovnu metodu testiranja kojim se testiranje vrši bez upotrebe. To je postupak korištenja funkcija i značajki aplikacije kao krajnji korisnik kako bi se provjerilo funkcionira li program prema potrebi. Ručnim testom provode se testovi na programu slijedeći niz unaprijed definiranih testnih slučajeva [10].

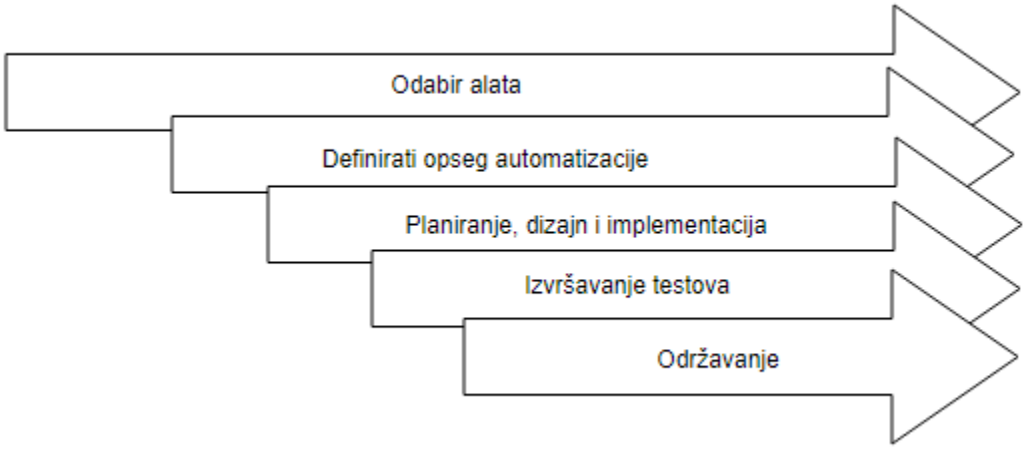
Prema [10] za ručno testiranje bitno je:

- Razumjeti zahtjeve – da bi se uspješno provodilo ručno ispitivanje, najprije se moraju razumjeti zahtjevi programa. Razumijući zahtjeve, znat će se što treba testirati i što klasificira grešku
- Pisanje testnih slučajeva – testni slučajevi vode testera kroz niz koraka za testiranje funkcija i različitih scenarija unutar aplikacije. Pisanje dobrih testnih slučajeva je važno jer čini izvršavanje testova brzim i osigurava dobru pokrivenost testovima.
- Izvođenje testova
- Izvješće o rezultatima ispitivanja – nakon testiranja, dobro je znati rezultate testova. Koliko testova je provedeno? Koliko testova nije uspjelo? Koliko testova je preskočeno?

Ručno testiranje će vjerojatno pronaći i riješiti stvarne probleme, ali u praksi zahtijeva jako puno napora. U poglavlju 4.3.1 korišteno je ručno testiranje tijekom implementacije poslužiteljske strane.

2.5.2. Automatsko testiranje

Automatsko testiranje predstavlja metodu automatiziranja skripti za izvršavanje testova. Automatski testovi prije kreiranja moraju biti detaljno opisani test slučajevima i procedurama [3]. Na slici 2.6 prikazan je proces automatskog testiranja.



Slika 2.6. Proces automatskog testiranja [9]

Prema [3], alate za testiranje može se podijeliti u dvije grupe:

- Alati koji sami generiraju skripte - u ovom okruženju prolazi se kroz scenarije testa, a alat snima izvršene akcije. Od svih snimljenih akcija kreira se skripta.
- Alati pomoću koji se pišu test skripte – samostalno se pišu skripte za testiranje koje su kopija koraka test procedura.

Bez obzira koju grupu alata koristili potrebno je imati pripremljeni test slučaj po kojem će se izvršavati test, ali prvi korak u tom smjeru je shvatiti da ni jedan testni plan ne može biti potpuno izveden s automatiziranim metodama. Izazov je utvrditi koje komponente ispitnog plana pripadaju ručnoj metodi ispitivanja i koje su komponente bolje poslužene pomoću automatiziranih testova. Automatizacija ne može sve, ljudi su trenutno pametniji od strojeva i mogu vidjeti i otkriti kvarove na način koji računala ne mogu [11].

2.5.3. Istraživačko testiranje

Istraživačko testiranje predstavlja metodu u kojoj se test procedure ne prate striktno. Tijekom testiranja otkrivaju se alternativni tokovi upotrebe sustava i odmah se provjeravaju, tako da su u ovoj metodi objedinjene aktivnosti identifikacije, dizajna i izvršavanja testova. Prednost ove metode je to što se otkrivaju mnogi problemi i scenariji koji unaprijed ne mogu biti planirani. Uz to manje vremena odlazi na pripremu, pošto se ne pišu zahtjevi, ni dokumentacija, tako da se vrijeme provodi efektivno na testiranju. Glavni nedostatak ovakvoga pristupa testiranju je to što se ne vodi evidencija o tome što je testirano, tako da se lagano mogu pronaći funkcionalnosti koje nisu testirane [3].

2.5.4. Regresijsko testiranje

Regresijsko testiranje predstavlja metodu testiranje koja se upotrebljava u iterativnim metodama razvoja programske podrške. Tijekom svake iteracije, implementira se određena funkcionalnost sustava i sam fokus testiranja temelji se na tim funkcionalnostima. U ovoj metodi postoji veliki rizik, zbog toga što neke funkcionalnosti testirane ranije mogu prestati s radom zato što se tijekom trenutne iteracije promjeni određeni dio koji utječe na ranije implementiranu funkcionalnost. Na ovaj način sigurno prolaze funkcionalnosti koje su implementirane u trenutnoj iteraciji, dok se za sve ostale mora izvršiti ponovna provjera [3].

Prema [3], dva načina za izvršavanja regresijskog testiranja su:

- Potpuno regresijsko testiranje - u svakoj iteraciji testiraju se sve funkcionalnosti implementirane u prethodnim iteracijama
- Regresijsko testiranje zavisnih funkcionalnosti – u svakoj iteraciji se testiraju samo funkcionalnosti izvedene u prethodnim iteracijama koje imaju dodirnih točaka sa funkcionalnosti u trenutnoj iteraciji.

2.6. Razine testiranja

2.6.1. Jedinično testiranje

Jedinično testiranje (engl. *Unit testing*) je razina testiranja gdje se ispituju pojedine jedinice ili komponente programa. Svrha je provjeriti da se svaka jedinica programa izvodi kako je dizajnirana, te je jedinica je najmanji testirani dio bilo kojeg programa. Obično ima jedan ili više ulaza i jedan izlaz. U proceduralnom programiranju, jedinica može predstavljati individualni program, funkciju itd. U objektno orijentiranom programiranju najmanja jedinica je metoda koja može pripadati osnovnoj, apstraktnoj ili izvedenoj klasi. Za testiranje jedinica se koriste driveri, čvorovi te lažni objekti [12].

Prema [12], prednosti jediničnog testiranja su:

- Jedinično testiranje povećava povjerenje u promjene i održavanje koda. Ako se dobiju dobri testovi i ako se pokreću svaki puta kada se promijeni bilo koji kod, moći će se odmah uočiti dobiveni nedostaci nakon promjena.
- Da bi se moglo provesti ispitivanje jedinica, kodovi moraju biti modularni. To znači da je kodove lakše ponoviti.

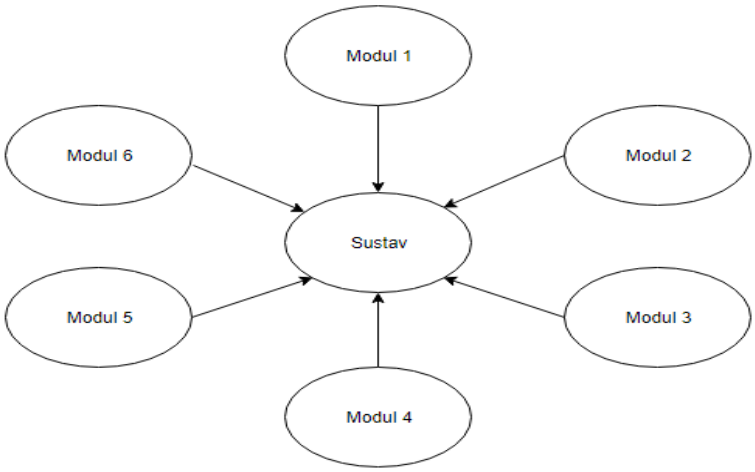
- Brži razvoj, zbog napora tijekom pronalaženja nedostataka.
- Trošak popravljanja kvarova otkrivenog tijekom ispitivanja uređaja je manji u usporedbi s onima kod nedostataka otkrivenih na višim razinama.
- Debugiranje je jednostavno. Kada test ne uspije, samo najnovije izmjene trebaju biti ispravljene.

2.6.2. Integracijsko testiranje

Testovi integracije određuju da li samostalno razvijene jedinice rade ispravno kada su međusobno povezane. Slični su jediničnim testovima, ali postoji jedna velika razlika, jedinični testovi su izolirani od drugih komponenti, dok testovi integracije nisu. Testiranje integracije uglavnom je korisno za situacije u kojem testiranje jedinica nije dovoljno. Ponekad se moraju imati testovi kako bi se utvrdilo da dva zasebna sustava, poput baze podataka i aplikacije ispravno rade. Sami po sebi testovi integracije su često sporiji od jediničnih testova, zbog dodatne složenosti. U nekim slučajevima moguće je da će se možda trebati konfigurirati neke postavke, kao što je postavljanje testne baze podataka. Testovi integracije se koriste u onim slučajevima kada treba ispitati dva zasebna sustava ili ako je dio koda previše složen za jedinično ispitivanje, a testovi integracije se obično pišu istim alatima kao i testovi jedinica [13].

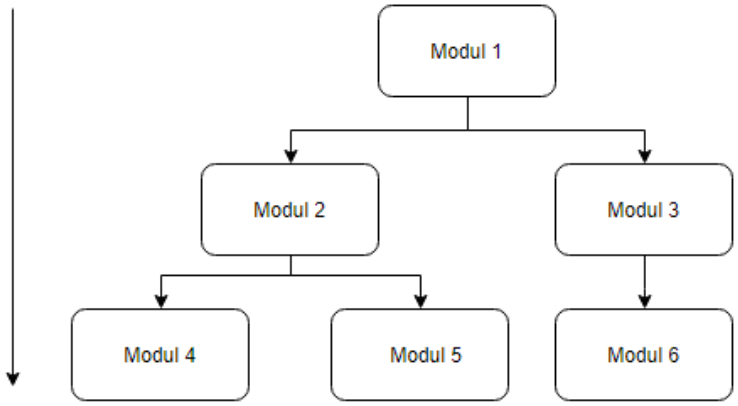
Prema [14], tri tehnike integracijskog testiranja su:

- „*Big Bang*“ integracijsko testiranje – sve komponente ili moduli su integrirani istodobno, kao što je prikazano na slici 2.7, nakon čega se sve ispituje u cjelini. Sve je završeno prije početka testiranja integracije što je prednost ove tehnike. Nedostatak je u tome što je proces općenito dugotrajan i teško je pratiti uzrok kvara.



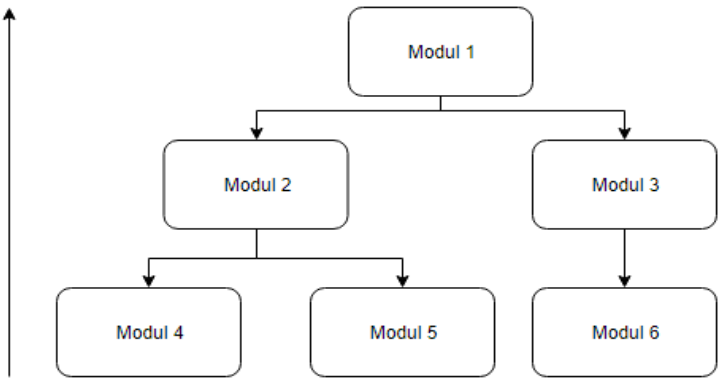
Slika 2.7. Primjer „*Big Bang*“ integracijskog testiranja [14]

- Testiranje integracije od vrha prema dolje – testiranje se odvija od vrha prema dolje kao što je prikazano na slici 2.8, nakon kontrole protoka, komponente ili sustavi se zamjenjuju. Biraju se komponente koje samo pozivaju druge komponente. Prednost je u tome što je ispitani proizvod vrlo dosljedan jer se ispitivanje integracije u osnovi izvodi u okruženju koje je gotovo slično stvarnosti. Dok je glavni nedostatak u tome što je osnovna funkcija testirana na kraju ciklusa.



Slika 2.8. Primjer integracijskog testiranja od vrha prema dolje [14]

- Testiranje integracije dolje prema vrhu – ispitivanje se odvija od dna kontrolnog toka prema vrhu, prikazano na slici 2.9, komponente ili sustavi se zamjenjuju. Biraju se komponente koje ne pozivaju druge komponente nego ih druge komponente koriste. Prednost je u tome što se razvoj i testiranje mogu se obaviti zajedno kako bi aplikacija bila učinkovita i prema specifikacijama klijenta. Dok glavni nedostatak je u tome što se tek na kraju ciklusa mogu se uočiti ključni nedostatci, potrebno je stvoriti testne upravljačke programe za module na svim razinama osim kontrole na vrhu.



Slika 2.9. Primjer integracijskog testiranja od vrha prema dolje [14]

2.6.3. Sustavsko testiranje

Sustavsko testiranje definira se kao testiranje kompletnog i potpuno integriranog programskog proizvoda. Ovo ispitivanje spada u kategoriju testiranja crne kutije, pri čemu poznavanje unutarnjeg dizajna koda nije preduvjet. Ispitivanje sustava se provodi u kontekstu specifikacija sustava ili specifikacija funkcionalnih zahtjeva. Posljednji test je provjeriti je li proizvod koje se isporučuje u skladu s specifikacijama navedenim u zahtjevnom dokumentu. Treba istražiti i funkcionalne i nefunkcionalne zahtjeve [15].

Postoje različite vrste testiranja sustava, a testni tim bih trebao odabrati one koji su im potrebni prije implementacije aplikacije. Prema [15], neke od vrsta testiranja sustava su:

- Testiranje upotrebljivosti – da bi se provjerilo da li aplikacija ili proizvod sadržava dobro korisničko iskustvo ili ne
- Regresijsko testiranje - da bi se potvrdilo da promjene u aplikaciji nisu nepovoljno utjecale na postojeće značajke
- Testiranje opterećenja – vrsta ne funkcionalnog testiranja koja pomaže razumjeti ponašanje aplikacije u određenom očekivanom opterećenju
- Testiranje migracija – testiranje aplikacije koji se koristi za migraciju podataka iz jedne aplikacije u drugu zamjensku aplikaciju
- Testiranje kompatibilnosti – provodi se za provjeru valjanosti, da se program jednako ponaša u različitim okolinama
- Test granične vrijednosti – namijenjen je uključivanju predstavnika graničnih vrijednosti
- „*Fuzz*“ testiranje – koristi se za unos nevažećih, nepostojećih ili slučajnih podataka

2.7. Strategije testiranja

2.7.1. Bijela kutija

Testiranje bijele kutije se odnosi na scenarij gdje testni tim duboko razumije unutrašnje funkcioniranje sustava ili komponenti sustava koji ispituje. Dobro razumijevanje sustava ili komponenti je moguće kada ih osoba koji ih testira razumije na razini programa ili koda.

Tako gotovo cijelo vrijeme, osoba koja testira treba razumjeti ili imati pristup izvornom kodu koji čini sustav, obično u obliku specifikacijskih dokumenata, tek tada će moći dizajnirati i izvršiti ispitne slučajeve koji pokrivaju sve moguće scenarije i uvjete [16].

Prema [16], ključna načela koja pomažu u izvršavanju testova bijele kutije su:

- Pokrivenost izvješća – osigurati testiranje svake linije koda.
- Pokrivenost grana – provjera svake grane
- Pokrivenost puta – provjera da li su svi mogući putevi testirani.

Testiranje se može obaviti na sustavskim, integracijskim i jediničnim razinama razvoja programa. Jedan od osnovnih ciljeva je provjera radnog toka za aplikaciju. To uključuje testiranje niza unaprijed definiranih ulaza od očekivanih ili željenih izlaza, ako određeni unos ne rezultira očekivanim izlazom, dogodila se greška [9].

Koraci prilikom testiranja bijelom kutijom prema [16] su:

- Utvrđivanje značajki i komponenti koje se žele testirati
- Obilježavanje svih puteva u grafu toka
- Odrediti sve moguće puteve iz grafa toka
- Pisanje testnih slučajeva kako bi se pokrio svaki pojedini put
- Izvršavanje i po potrebi ponavljanje test slučajeva

Testiranje pomoću ove strategije omogućuje testiranje sustava s gledišta razvojnog programera.

2.7.2. Crna kutija

Testiranje crne kutije je strategija testiranja u kojoj se provjerava funkcionalnost sustava, a da se ne gleda interna struktura koda, detalje implementacije i znanje o pojedinim unutarnjim putevima programa. Glavna usredotočenost je na ulaza i izlaze sustava [9]. U ovoj metodi ono može biti funkcionalno ili nefunkcionalno, iako je to češće funkcionalno. Testiranje svih mogućih kombinacija ulaznih vrijednosti aplikacije je iscrpno i većinu vremena nepraktično [17].

Prema [17], tehnike prilikom testiranja crnom kutijom su:

- Analiza granične vrijednosti – valjana na svim razinama testiranja, granična vrijednost analizira granice između ekvivalentnih odjeljaka

- Grafički učinak uzorka – analizira kombinacije unosa za utvrđivanje uzoraka i posljedica ili testne kombinacije ulaza koji rezultiraju različitim rezultatima
- Testiranje tablice odluka – koristi se u složenim scenarijima za ispitivanje kombinacija ulaza koji rezultiraju različitim rezultatima, tablice odluka mogu testnom timu ponuditi sustavni prikaz različitih mogućnosti
- Testiranje prijelaznog stanja – sustav je rijetko u jednom konačnom stanju, stoga ova analiza omogućuje testeru da vidi što se događa kada ulazi izazivaju promjene.
- Slučajno i stresno ispitivanje – izvođenje rijetkih scenarija, kao i tipičnih, pomaže povećati šanse za pokretanje kvara na sustavu.
- Pogreške nagađanja – iskusni testni tim može koristi intuiciju kako bi u testiranju mogli pogoditi gdje bi moglo doći do pogreške.

Opći koraci pri izvođenju testiranja crne kutije prema [9] su:

- Ispituju se zahtjevi i specifikacije sustava
- Odabiru se ispravni ulazi kako bi se provjerilo da li ih sustav pravilno obrađuje, također se odabiru i ne ispravni, radi potvrde o otkrivanju pogrešaka
- Određuju se očekivani izlazi za sve ulaze
- Konstruiraju se test slučajevi
- Uspoređuju se stvarni rezultati s očekivanim izlazima

Glavni nedostatak ovakve strategije je nepoznavanje unutarnje strukture, što znači da se testiranje obavlja na slijepo.

2.8. Vrste testiranja

2.8.1. Funkcionalno testiranje

Program se implementira prema zahtjevima korisnika kojima se definiraju funkcionalnosti. Funkcionalno testiranje predstavlja vrstu testiranja u kojoj se provjerava da li su funkcionalnosti predviđene korisničkim zahtjevima ispravno implementirane u sustav. Vrši se pregledom zahtjeva te se prolazi kroz različite scenarije korištenja sustava i utvrđuje se postojanje propusta [3].

Prema [3], postoje dva osnovna cilja funkcionalnog testiranja:

- Provjera da li sustav izvršava sve što je predviđeno zahtjevima
- Provjera da li sustav ne izvršava ništa drugo što nije predviđeno zahtjevima

U prvom slučaju strategija je jasna, potrebno je usporediti korisničke zahtjeve sa implementacijom u sustavu. Dok se drugi slučaj često zanemaruje, sustav ne bi trebao raditi ništa što nije predviđeno zahtjevima, pošto takve aktivnosti kasnije mogu predstavljati propust. Međutim ne postoji jasno definirana lista što sustav ne bi trebao raditi, pa je drugi slučaj u nekim sustavima i teško testirati. Pristup ovakvom testiranju nije lak pošto ne postoji nikakav izvor podataka na osnovu kojeg se mogu definirati određeni test slučajevi kojima bi se pokrile sve funkcionalnosti sustava, u praksi jedini način da se otkriju greške je pregled programskog koda [3]. Kroz proces funkcionalnog testiranja proći se u poglavlju 4.3.4.

2.8.2. Nefunkcionalno testiranje

To je vrsta testiranja za koja se upotrebljava za provjeru nefunkcionalnih aspekata (izvedba, upotrebljivost, pouzdanost, itd.). Namijenjeno je testiranju spremnosti sustava prema nefunkcionalnim parametrima koji se nikada ne rješavaju funkcionalnim testiranjem [9].

Prema [9], ciljevi nefunkcionalnog testiranja su:

- Trebalo bi povećati upotrebljivost, učinkovitost, održivost i prenosivost proizvoda.
- Pomaže smanjiti rizik proizvodnje i troškove povezane s nefunkcionalnim aspektima proizvoda.
- Optimiziranje načina instalacije, postavljanja, izvršavanja, upravljanja i nadzora proizvoda.

Prema [9], karakteristike nefunkcionalnog testiranja su:

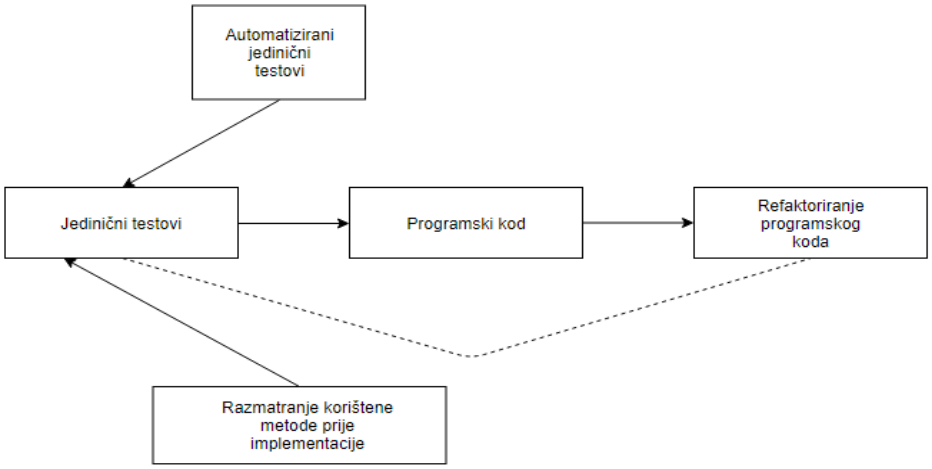
- Treba biti mjerljivo, tako da nema mjesta za subjektivnu karakterizaciju
- Važno je odrediti prioritet zahtjeva
- Provjera jesu li atributi kvalitete ispravno identificirani

Nefunkcionalno testiranje se sastoji od različitih parametara kao što su sigurnost, pouzdanost, izdržljivost, dostupnost, upotrebljivost, skalabilnost, fleksibilnost itd.

3. RAZVOJ POKRETAN TESTIRANJEM

Razvoj pokretan testiranjem (engl. *Test driven development*) je proces razvoja programa koji se oslanja na ponavljanje vrlo kratkog razvojnog ciklusa, te uključuje pisanje test slučajeva usporedno s pisanjem koda, što uvelike poboljšava razvoj, to je kombinirani proces detaljnog dizajna i razvoja, vođen nizom testova [18].

Ovakav pristup počinje s projektiranjem i razvojem testova za svaku malu funkcionalnost aplikacije, prvo se razvija test koji određuje i provjera što će kod učiniti. U uobičajenom postupku testiranja najprije se generira kod, a zatim testira. Testovi mogu biti neuspješni jer su razvijeni i prije samog razvoja aplikacije. Da bi test prošao razvojni tim mora razviti i preoblikovati kod. Refaktoriranje koda znači promjenu nekog djela koda, tj. unutarnje strukture koda bez utjecaja na njegovu funkcionalnost [9]. Slika 3.1 prikazuje aktivnosti prilikom razvoja pokretanog testiranjem.



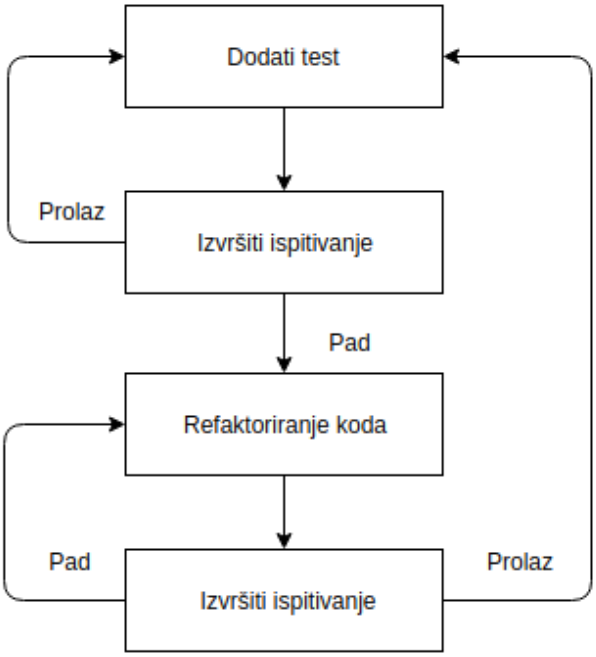
Slika 3.1. Primjer razvoja pokretanog testiranjem [9]

Ovaj jednostavni koncept omogućuje izbjegavanje pisanja dupliciranog koda, te ispravljanje koda pomoću testova, prije samog nastavka razvoja.

Prema [9], koraci prilikom razvoja pokretanog testiranjem su:

1. Dodavanje test slučaja
2. Pokrenuti sva ispitivanja i provjeriti da li je novi test slučaj uspio
3. Napisati programski kod
4. Pokrenuti testiranja i refaktorirati kod
5. Ponoviti sve korake

Slika 3.2 prikazuje potrebne korake prilikom razvoja pokretanog testiranjem.



Slika 3.2. Koraci prilikom razvoja pokretanog testiranjem [9]

Ciklus razvoja pokretanog testiranjem, prema [19], dijeli se na tri faze:

- Crvena faza - u ovoj fazi se pišu test slučajevi za postupke koji će biti provedeni testiranjem. U ovoj fazi stavljamo se u ulogu zahtjevnog korisnika koji želi koristiti napisani kod na što jednostavniji način. Donose se odluke o tome kako će se programski kod koristiti, temeljeno na potrebama u tome trenutku, a ne budućim potrebama.
- Zelena faza - najjednostavnija faza, piše se samo programski kod. U ovoj fazi zadatak je napisati jednostavno rješenje koje provodi testni slučaj, dovoljno da prođe crveni test. Dozvoljeno je kršiti najbolje primjere iz prakse i čak kopirati kod. Dupliciranje kodova biti će uklonjeno u refaktorizacijskoj fazi. Razlog tome je što želimo minimizirati greške, a jednostavan zadatak je manje sklop pogreškama.
- Faza refaktoriranja - u ovoj fazi dolazi do promjene koda, kako bi svi testovi došli u zelenu fazu, ali obavezno je ukloniti duplicirani kod. Stavljamo se u ulogu programera koji želi popraviti ili preoblikovati kod kako bi ga doveli u profesionalnu razinu. U crvenoj fazi prikazuju se vještine korisnicima, dok se u ovoj fazi pokazuju vještine programerima.

Prema [9], prednosti razvoja pokretanog testiranjem su:

- Rana obavijest o pogrešci
- Bolji dizajn, čišći kod - omogućuje pisanje manjeg koda s jedinstvenom odgovornošću, a ne monolitnim postupcima s višestrukim odgovornostima, što ga čini jednostavnijim za razumijevanje.
- Provjera u fazi refaktorizacije - rezultira bržim kodom s manje pogrešaka koje se mogu ažurirati uz minimalne rizike
- Dobar za timski rad - u nedostatku bilo kojeg člana tima, drugi članovi tima lako mogu preuzeti i raditi na programskom kodu.
- Dobar za razvojne programere - iako programeri moraju trošiti više vremena za pisanje test slučajeva, potrebno je mnogo manje vremena za ispravljanje pogrešaka i razvoj novih značajki.

U odnosu na tradicionalni pristup testiranju, prema [9], razvoj pokretan testiranjem ima svojih prednosti:

- Osigurava da sustav zadovolji definirane zahtjeve uz veće povjerenje u implementirani sustav.
- Veća je usredotočenost na proizvodni kod koji provjerava hoće li ispitivanje ispravno funkcionirati.
- Postiže se pokrivenost koda do 100%, svaka linija koda je testirana

Razvoj pokretan testiranjem je prije svega tehnika specifikacije, te nam osigurava da izvorni kod bude temeljito testiran na potvrdnoj razini. Poboljšava kvalitetu koda identificirajući rane pogreške [18].

3.1. Razvoj pokretan testiranjem kod web aplikacija

Razvijanje suvremenih web aplikacija veliki je izazov, prolazi se kroz brzi razvojni ciklus, zbog zahtjeva kupaca, evolucije zahtjeva, implementacije novih značajki te popravak pogrešaka kod već implementiranih sustava sve u roku od nekoliko dana. U tom kontekstu agilni pristupi za automatizirano testiranje smatraju se kao najbolji izbor za razvoj i kvalitetu web aplikacija [20].

Prema [20], pristup kod razvoja web aplikacija pokretanog testiranjem temelji se na:

- Analiza zahtjeva - zahtijeva izradu zahtjeva specifikacija za web aplikaciju koja se razvija. To uključuje podatke budućih korisnika, kupaca, analizu zahtjeva za razumijevanje jesu li potpuni, dosljedni, uzimajući zahtjeve kao slučajeve korištenja.
- „*Mockup*“ razvoj - zadaća je stvaranje skupa modela koji se koriste za prototipiranje korisničkog web sučelja. Koristi se da bi se smanjilo koliko god je moguće ručnih intervencija prilikom pokretanja automatskih testova.
- Test prihvatljivosti - nakon dostupnih modela moguće je simulirati njihov rad kroz različite test slučajeve. Vrlo je korisno za otkrivanje i što prije rješavanje mogućih problema.
- Razvoj web aplikacije - funkcionalnosti se provode po smjernicama test slučajeva sve dok svi testovi ne prođu uspješno.
- Poboljšanje održavanja test slučajeva - nakon razvijene web aplikacije, sljedeći korak je održavanje test slučajeva. Testne skripte generirane kroz faze implementacije često se ponavljaju, dobra praksa je uklanjane dupliciranih kodova. Nakon ovog koraka testne skripte lakše je razumjeti, mijenjati i održavati.
- Proširenje test slučajeva - poboljšanje učinkovitosti. Zamjena test skripti koje sadrže unaprijed definirane vrijednosti, radi regresijskog testiranja.
- Poboljšanje robusnosti - uključenje novih tvrdnji za postojeće testne slučajeve ili dodavanje novih test skripti za interakciju koji do sada nisu uzeti u obzir.

Web testiranje u jednostavnim terminima provjerava web aplikaciju i provjerava postojanje potencijalnih grešaka prije nego što aplikacija dođe u proizvodno okruženje, tj. do krajnjih korisnika. Tijekom ove faze provjeravaju se pitanja kao što su sigurnost web aplikacije, funkcionalnost, pristup korisnicima kao i sposobnost upravljanja prometom. Tako da testove možemo zamisliti kao zaštitne mjere aplikacije koju gradimo.

Neki od glavnih testova koji se provode prilikom testiranja web aplikacija su:

- Ispitivanje upotrebljivosti - kako bi aplikacija bila učinkovita, korisnička sučelja trebaju biti u skladu s standardima. Testiranje upotrebljivosti izmjenjuju se karakteristike interakcija korisnika i same aplikacije, te se utvrđuju slabosti. I kao takav test upotrebljivosti je ključan u dizajniranu aplikacije.
- Ispitivanje funkcionalnosti - test funkcionalnosti osigurava komunikaciju određenih funkcija međusobno, prema zahtjevima aplikacije. U web aplikaciji polazna točka za

funkcionalno testiranje može biti veza između promjena podataka u aplikaciji i bazi podataka. Neki od funkcionalnih testova uključuju: testiranje baze podataka, testiranje konfiguracije, ispitivanje kompatibilnosti, ispitivanje protoka podataka

- Ispitivanje sučelja - kako bi se osiguralo da su pojedine komponente ispravno povezane, tj. da je komunikacija između njih ispravna provodimo test sučelja. Potrebno je testirati kompatibilnost poslužitelja s programom, hardverom, mrežom i bazom podataka. Treba provjeriti provode li se sve interakcije između tih poslužitelja i da li se ispravno rukuje pogreškama. Ako se pojavi pogreška za bilo koji upit, aplikacija bi trebala prikazati poruke o pogrešci na odgovarajući način. Glavni zadatak je osigurati da su sučelja koja su izložena komponentama općenito i proširiva, trebaju imati mogućnost prilagođavanja promjenama, a pritom ostati kompatibilna.
- Ispitivanje performansi - vrsta testiranja kojim se osigurava da aplikacija dobro funkcionira pod neočekivanim opterećenjem. Cilj nije pronaći greške, već eliminirati ograničenja u pogledu vremena odziva, pouzdanosti, skalabilnosti i korištenja resursa.

Kao glavno pitanje koje možemo postaviti prije početka testiranja: *Što trebamo testirati u aplikaciji? Koliko testova moramo imati?* Odgovor se razlikuje od upotrebe, ali kao pravilo, možemo slijediti smjernice koje nam daje piramida testiranja.

Test piramida prikazana na slici 3.3 je pomoćni alat za rješavanje problema zbog prevelikog oslanjanja na dugotrajne UI testove. Glavno pravilo piramide kaže da su testovi na nižim razinama jeftiniji za pisanje i održavanje te brži za izvođenje. Testovi na gornjim razinama znatno su skuplji za pisanje i održavanje, a sporiji za pokretanje. Stoga prilikom testiranja trebali bismo imati puno jediničnih testova, malo manje integracijskih, te vrlo malo *UI* (engl. *User Interface*) testova.



Slika 3.3. Piramida testiranja [21]

4. PRIMJENA TESTIRANJA NA PRIMJERU WEB APLIKACIJE

Razvijeno programsko rješenje omogućuje korisnicima lakše skladištenje podataka prikupljenih s različitih senzora za praćenje klimatskih promjena, te na osnovu tih podataka predviđanje temperature za određeno vremensko razdoblje. Tehnologije u kojima je razvijeno programsko rješenje detaljnije će biti opisane u poglavlju 4.2.

4.1. Zahtjevi sustava

Prije samog razvoja definirani su određeni zahtjevi za funkcionalnosti koje bi programsko rješenje trebalo pružiti korisniku. Zahtjevi su prikazani u tablici 4.1.

Tablica 4.1. Zahtjevi koje aplikacija mora pružiti korisniku

Zahtjev	Opis
Prijava	Aplikacija mora pružiti prijavu korisniku
Registracija	Aplikacija mora pružiti registraciju korisnika
Odjava	Aplikacija mora pružiti odjavu korisnika
Unos podataka	Aplikacija mora pružiti registriranom korisniku spremanje podataka
Predviđanje temperature na osnovu unesenih podataka	Aplikacija mora pružiti registriranom korisniku predviđanje temperature na osnovu spremljenih podataka
Uklanjanje podataka	Aplikacija mora omogućiti registriranom korisniku uklanjanje određenih podataka
Predviđanje temperature na osnovu proizvoljnih podataka	Aplikacija mora omogućiti registriranom korisniku predviđanje temperature za bilo koji skup proizvoljnih podataka

4.2. Opis korištenih tehnologija i alata

Node.JS

Node.JS je više platformsko „*runtime*“ okruženje koje nam omogućuje izvršavanje JavaScript koda izvan web preglednika. Koristi Google-ov „*V8 JavaScript engine*“ za implementiranje i izvršavanje koda. Node.JS aplikacije temelje se na događajima i izvode se asinkrono. Kod koji je izgrađen na Node platformi ne slijedi tradicionalni model primanja, obrade, slanja, i čekanja. Umjesto toga, obrađuje dolazne zahtjeve u stalnom stablu događaja i šalje male zahtjeve jedan za drugim bez čekanja odgovora. Često se koristi za izradu web poslužiteljskih usluga, a idealan je za izradu visoko skalabilnih, podatkovno intenzivnih i „*real-time*“ aplikacija. Node.JS je izvrstan za prototipiranje i agilni razvoj, omogućuje implementaciju super brzih i

skalabilnih web usluga, JavaScript se koristi posvuda te pruža veliki ekosustav biblioteka otvorenog koda.

Express.JS

Express.JS je razvojni okvir za Node.JS koji nam pomaže u organiziranju web aplikacije u MVC (engl. *Model-View-Controller*) arhitekturi na strani poslužitelja. Napisan u JavaScriptu, djeluje samo kao tanki sloj osnovnih značajki web aplikacija. Nastoji staviti kontrolu u ruke programera i olakšati razvoj web aplikacija u Node.JS-u, što ga čini idealnim kandidatom za brzi razvoj. Pojednostavljuje razvoj i olakšava pisanje sigurnih, modularnih i brzih aplikacija. Pruža robustan skup značajki za web i mobilne aplikacije, te daje pristup i dolaznom i odlaznom web prometu. Uz Express.JS dolaze sve potrebne značajke za izradu HTTP poslužitelja.

MongoDB

MongoDB je vodeća NoSQL (ne-relacijska) baza podataka. Usmjerena je na dokumente i pohranjuje podatke u dokumente s dinamičnom shemom. To znači da se podatci pohranjuju neovisno o strukturi kao što je broj polja ili vrsta polja za pohranu vrijednosti. Dizajnirana je oko načela da podatci nisu uvijek isti. MongoDB dokumenti su slični JSON objektima. Vrlo lako se može promijeniti struktura zapisa (koju zovemo dokument) jednostavnim dodavanjem novih polja ili uklanjanje postojećih. Ta sposobnost MongoDB-a pomaže u predstavljanju hijerarhijskih odnosa, lako pohranjivanje polja i drugih složenijih struktura. Pruža visoku izvedbu, visoku dostupnost, jednostavnu skalabilnost i replikaciju izvan okvira, te omogućuje automatsko uklanjanje.

ReactJS

ReactJS je često opisan kao „View“ u MVC (engl. *Model-View-Controller*) strukturi. Točnije rečeno ReactJS je JavaScript biblioteka za izradu korisničkog sučelja. Središte svih ReactJS aplikacija su komponente, za koje vrijedi da su samostalni moduli koji donose izlaz. Može uključivati jednu ili više drugih komponenata u svojem izlazu. Općenito, za izradu aplikacije pišemo komponente koje odgovaraju različitim elementima sučelja. Nakon toga organiziranjem tih komponenti u komponente više razine definiramo strukturu aplikacije, što omogućuje razvojnim programerima stvaranje velikih web aplikacija koje mogu mijenjati podatke, bez ponovnog učitavanja. Glavna svrha ReactJS-a je da bude brz, skalabilan i jednostavan. Upotrebljava *JSX* što omogućuje upotrebu HTML sintakse za prikazivanje komponenti. Kao glavna značajka je virtualni model objekta dokumenta koji stvara memorijsku strukturu podataka,

na osnovu koje se izračunavaju napravljene promjene kako bi se ažuriralo sučelje. Tok podatak u React.JS aplikacijama je jednosmjerni. React.JS ima velik broj prednosti, a neke od njih su jednostavnost, povezivanje podataka, performanse te omogućuje jednostavno testiranje.

Uzorak REST

REST (engl. *Representational State Transfer*) je arhitekturalni stil koji definira skup ograničenja koja će se koristiti za izradu web usluga, što olakšava međusobnu komunikaciju sustava. U REST-u implementacija klijenta i implementacija poslužitelja mogu se obavljati samostalno, bez da znaju jedan o drugom. Što znači da se kod na strani klijenta može promijeniti u bilo kojem trenutku bez utjecaja na rad poslužitelja i obrnuto. Sve dok svaka strana zna format poruke koja se šalje. REST sve gleda kao resurse, a njima se pristupa preko URI-ja (engl. *Uniform Resource Identfier*). Koristi HTTP protokol sa svojim standardima kao što su GET, POST, PUT, DELETE, a dobiveni resursi su obično u XML ili JSON formatu [22].

Prema [23], svojstva REST-a su:

- performanse
- skalabilnost
- jednostavnost sučelja
- prenosivost
- pouzdanost
- mogućnost izmjene komponenti

Prema [23], ograničenja REST-a su:

- klijent-poslužitelj – klijenti ne brinu za spremanje podataka, a poslužitelj ne brine za korisničko sučelje
- bez stanja – poslužitelj ne pamti nikakva stanja klijenta
- mogućnost pričuvne memorije – dobra politika keširanja, poboljšavajući skalabilnost i performanse
- slojeviti sustav – klijent ne zna da li je spojen na neki među poslužitelj ili na krajnji poslužitelj
- jedinstveno sučelje – identifikacija resursa, manipulacija resursima, samo-opisujuće poruke, hipermedija kao osnovno stanje aplikacije
- kod na zahtjev – poslužitelj može proširiti ili prilagoditi funkcionalnost klijenta šaljući mu izvršni kod

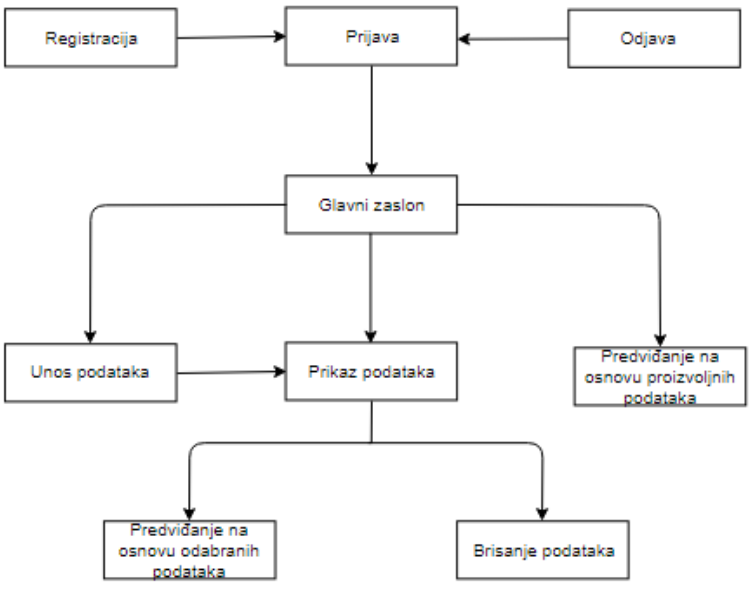
Jest

Jest je razvojni okvir za testiranje JavaScript aplikacija. To je projekt otvorenog koda kojeg je razvio i održava Facebook, a posebno je prikladan za testiranje ReactJS aplikacija. Vrlo je brz i jednostavan za korištenje, a kao glavna snaga je upotreba svih funkcija bez potrebne konfiguracije. Omogućuje automatsko pronalaženje testova i testove pokreće lažnom implementacijom DOM-a (engl. *Document object model*). Sadrži ugrađene izvještaje o pokrivenosti koda testova te pokreće ispitivanje u paralelnim procesima.

4.3. Razvijeno programsko rješenje

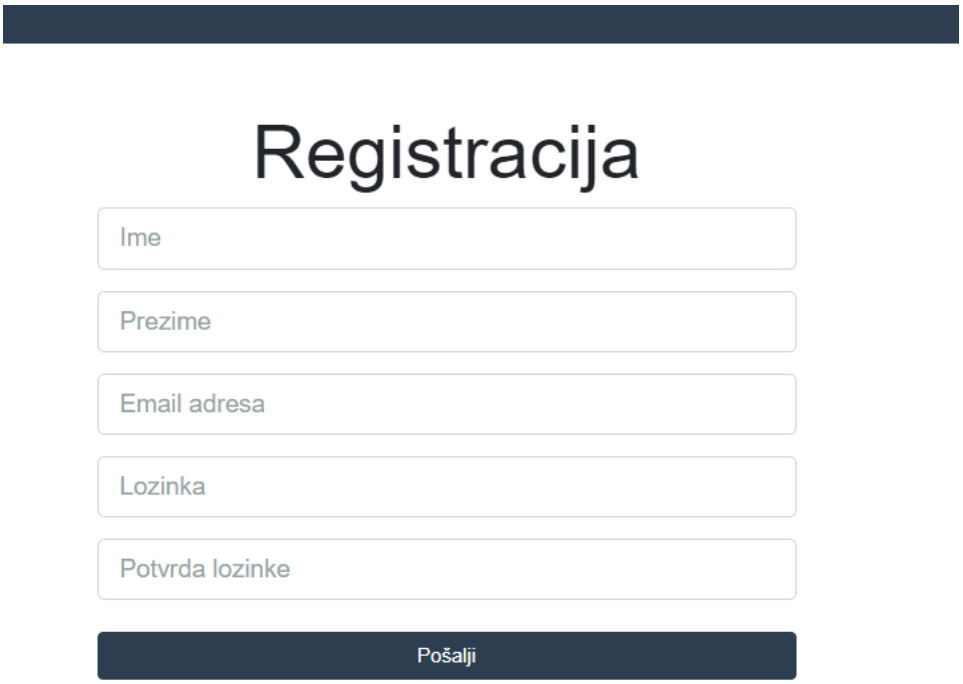
Aplikacija je razvijena u takozvanom „*MERN stack*“ okruženju koje obuhvaća nekoliko različitih tehnologija (*MongoDB, Express.JS, React i Node.JS*) objašnjenih u prethodnom poglavlju 4.2. Razvijeno programsko rješenje omogućuje korisnicima spremanje podataka prikupljenih s različitih meteoroloških senzora, te je na osnovu tih podataka omogućeno predviđanje temperature za određeno razdoblje. Za uslugu predviđanja temperature aplikacija komunicira s udaljenim *Azure Web API-em*. Koji na osnovu danih podataka predviđa temperaturu prema unaprijed treniranom modelu strojnog učenja. Također je omogućeno predviđanje temperature za bilo koji proizvoljni skup podataka. Tip podataka s kojim sustav radi su brojevi, zbog toga što *JavaScript* ima samo jedan tip brojeva, te oni mogu biti pisani sa ili bez decimalne točke.

Na slici 4.1 prikazan je blok dijagram sustava.

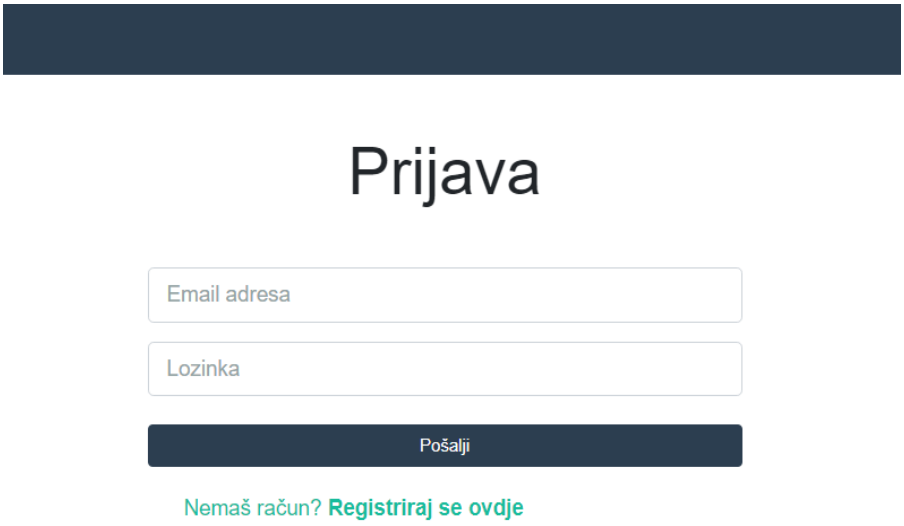


Slika 4.1. Blok dijagram sustava

Prilikom ulaska u aplikaciju korisniku će biti ponuđene opcije *Prijave* ili *Registracije*. Na taj način svaki korisnik ima uvid isključivo u svoje podatke. Prilikom prijave generira se *JSON web token* u kojem su sadržane informacije o korisniku, te se on pohranjuje u lokalnu memoriju web preglednika. Na slikama 4.2 i 4.3 prikazani su obrasci za *Registraciju* i *Prijavu* korisnika.



Slika 4.2. Izgled obrasca za registraciju korisnika



Slika 4.3. Izgled obrasca za prijavu korisnika

Programski kod 4.1 pokazuje implementaciju prijave korisnika na poslužiteljskoj strani. Gdje se uzima email adresa i lozinka poslani s klijentske strane, te se pomoću email adrese traži korisnik u bazi podatka. Ukoliko je korisnik pronađen lozinka se dekodira i poziva se funkcija koja generira *token*, u suprotnom se šalje poruka o netočnoj lozinki ili nepostojećem korisniku.

```
router.post('/auth', (req, res, next) => {
    const { errors } = validateLogin(req.body);

    if ( Object.keys(errors).length > 0 )
      return res.status(400).json(errors);

    const email = req.body.email;
    const password = req.body.password;

    User.findOne({ email })
        .then(user => {
            if (!user) {
                errors.email = 'Korisnik nije pronađen';
                return res.status(400).json(errors);
            }
            bcrypt.compare(req.body.password, user.password)
                .then(isMatch => {
                    if(isMatch) {
                        const token = user.generateAuthToken(user);
                        res.json(token);
                    } else {
                        errors.password = 'Netočna lozinka';
                        return res.status(400).json(errors);
                    }
                });
        });
});
```

Programski kod 4.1. Dio koda odgovoran za prijavu korisnika na poslužiteljskoj strani

U programskom kodu 4.2 prikazan je dio koda odgovoran za slanje podataka iz obrasca za prijavu korisnika s klijentske strane prema poslužiteljskoj strani. Nakon slanja čeka se odgovor, ukoliko je sve u redu prima se *token*. U slučaju primljene greške poruke se spremaju u privremeno skladište te prikazuju korisniku.

```
export const login = user => dispatch =>
  axios
    .post('http://localhost:5000/users/auth', user)
    .then(res => {
      const  token  = res.data;
      localStorage.setItem('token', token);
      setToken(token);
      const decoded = jwt_decode(token);
      dispatch(setCurrentUser(decoded));
    })
    .catch(err =>
      dispatch({
        type: 'GET_ERRORS',
        payload: err.response.data
      })
    );
```

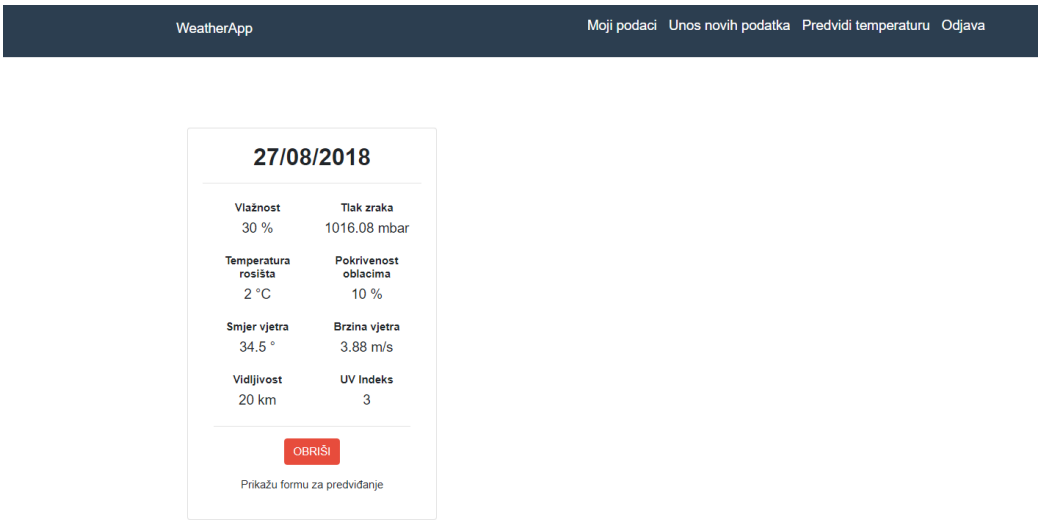
Programski kod 4.2. Dio koda odgovoran za prijavu korisnika na poslužiteljskoj strani

Programski kod 4.3 prikazuje dio koda koji je zaslužan za provjeru postojanja *tokena* u lokalnoj memoriji, ukoliko postoji, postavlja se u HTTP zaglavlje, te se dekodira kako bih se dobio korisnik.

```
if(localStorage.token) {
    setToken(localStorage.token);
    const decoded = jwt_decode(localStorage.token);
        store.dispatch({
        type: 'SET_CURRENT_USER',
        payload: decoded
    });
}
```

Programski kod 4.3. Dio koda odgovoran za provjeru postojanja tokena

Nakon prve prijave korisnik će svaki sljedeći put automatski biti preusmjeren na početni zaslon aplikacije prikazan na slici 4.4, sve dok se ne odabere opciju *Odjava*, gdje se *token* briše iz memorije web preglednika. Funkcija odgovorna za *Odjavu* korisnika i uklanjanje *tokena* prikazana je u programskom kodu 4.4.

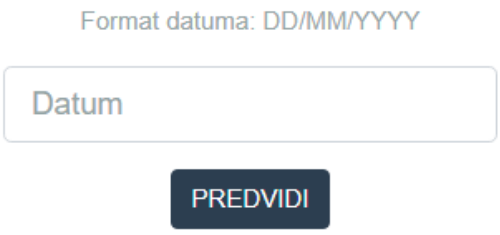


Slika 4.4. Početni zaslon aplikacije

```
export const logout = () => dispatch => {
  localStorage.removeItem('token');
  setToken(false);
  dispatch(setCurrentUser({}));
};
```

Programski kod 4.4. Dio koda odgovoran za odjavu korisnika i uklanjanje tokena

Nakon ulaska u aplikaciju korisniku se nudi mogućnost unosa podataka, prikaza podataka iz baze podataka, te predviđanje temperature na osnovu odabranih podataka. Na stranici prikaza podataka, korisnik ima mogućnost predviđanja temperature na temelju odabranih podataka, za određeni datum, te također uklanjanje tih podataka iz baze podataka. Odabirom predviđanja otvara mu se dodatni dijalog prikazan na slici 4.5 gdje korisnik unosi datum u određenom formatu, te ukoliko je format valjan podaci se šalju na *Azure Web API*, koji vraća predviđenu temperaturu na osnovu treniranog modela. Uneseni datum mora biti u odgovarajućem formatu. Dio koda koji pomoću regularnog izraza provjerava ispravnost datuma prikazan u programskom kodu 4.5.



Slika 4.5. Izgled obrasca za unos datuma

```
export const validateDate = date => {
    const re = /^(0?[1-9]|[12][0-9]|3[01])[\/\-](0?[1-9]|1[012])[\/\-]\d{4}$/;
    return re.test(date);
}
```

Programski kod 4.4. Dio koda za provjeru ispravnosti datuma

U formi za unos podatka koja je prikazana na slici 4.6 definirana su ograničenja i implementirana ja validacija za korisnički unos, kako bi uneseni podatci bili što stvarniji. Sva polja za unos su obaveza, nije dozvoljen unos stringova, te je za svaku varijablu posebno definiran raspon vrijednosti. Također ista validacija je upotrjebljena u formi za predviđanje temperature.



Slika 4.6. Izgled forme za unos podataka

U programskom kodu 4.5 prikazana je implementacija poslužiteljske strane za unos podataka. Ukoliko nema greška prilikom validacije podaci se spremaju u bazu podataka.

```
router.post('/', auth, (req, res) => {
    const { errors } = validateInputData(req.body);
    if ( Object.keys(errors).length > 0 )
      return res.status(400).json(errors);

    const newData = new Data({
        windBearing: req.body.windBearing,
        dewPoint: req.body.dewPoint,
        windSpeed: req.body.windSpeed,
        cloudCover: req.body.cloudCover,
        pressure: req.body.pressure,
        visibility: req.body.visibility,
        humidity: req.body.humidity,
        uvIndex: req.body.uvIndex,
        user: req.user.id
    });
    newData.save()
        .then(data => res.json(data));
});
```

Programski kod 4.5. Dio koda odgovoran za spremanje podataka na poslužiteljskoj strani

Programski kod 4.6 prikazuje funkciju koja se poziva prilikom spremanja podataka. Dohvaćaju se vrijednosti unesene u formi za unos podataka, te ukoliko je unos valja nakon spremanja podataka korisnika se prebacuje na početnu stranicu. U suprotnome, ispisuje se poruka o nastaloj pogrešci.

```
export const addData = (data, history) => dispatch => {
    dispatch(clearErrors());
    return axios
        .post('http://localhost:5000/data', data)
        .then(res => {
                dispatch({
                    type: 'ADD_DATA',
                    payload: res.data
                })
                history.push('/data');
            }
        )
        .catch(err =>
            dispatch({
                type: 'GET_ERRORS',
                payload: err.response.data
            })
        );
}
```

Programski kod 4.6. Dio koda odgovoran za slanje podataka s klijentske strane

4.1.1. Model strojnog učenja

Strojno učenje predstavlja način programiranja računala kako bi ona mogla djelovati kao ljudi te poboljšati vlastiti učinak tijekom vremena, kroz dane podatke i interakcije u stvarnom svijetu. Koristi se u rješavanju složenih problema, sustavima koji se dinamički mijenjaju i sustavima s ogromnim količinama podataka. Takav sustav treba imati sposobnost prilagođavanja okolini, tj. učenja iz okoline. Prema [24], postoje četiri oblika strojnog učenja, a to su nadzirano učenje, nenadzirano učenje, djelomično nadzirano učenje i učenje s povratnom vezom. Nadzirano učenje možemo još podijeliti na klasifikaciju, otkrivanje nepravilnosti i regresiju.

Za potrebe ovog programskog rješenja koristi će se regresija. Ona se upotrebljava u slučajevima kada je potrebno predvidjeti neku vrijednost, u ovom slučaju cilj je predvidjeti vrijednost temperature na osnovu danih podataka.

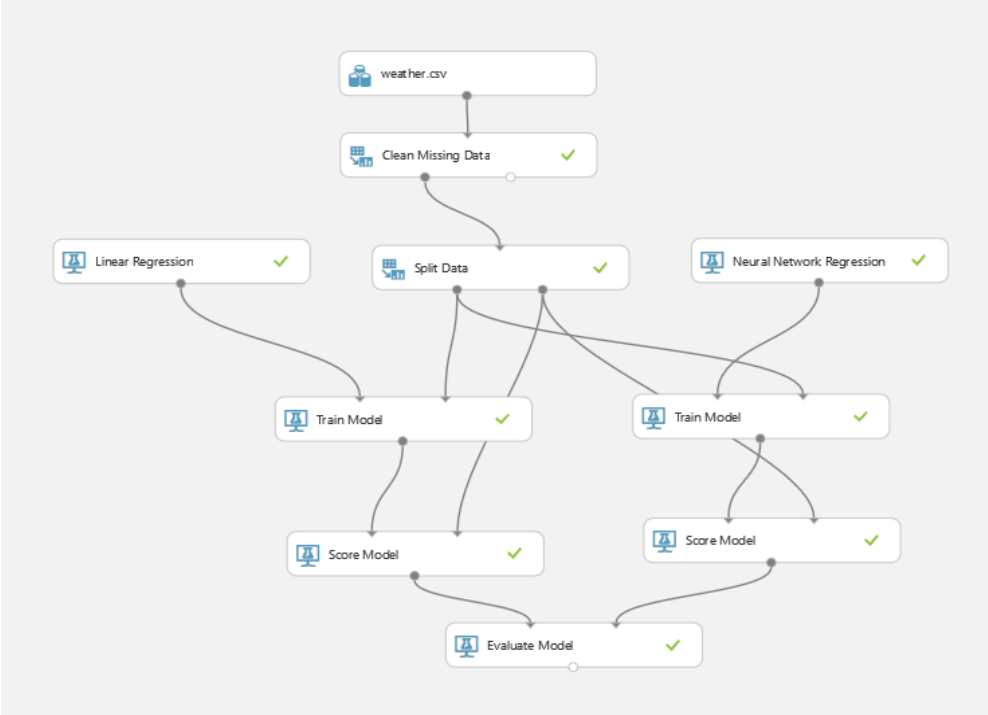
Jedna od najvažnijih stavki u procesu strojnog učenja su podatci. Prije same izrade modela izvršena je obrada nad podacima, podatci koji nisu potpuni su uklonjeni. Također su uklonjeni podatci koji su nepotrebni, a sve u svrhu dobivanja što veće preciznosti prilikom predviđanja. Za izradu ovog modela korišteni su podatci [25] koji su prikupljeni sa različitih senzora kroz određeni vremenski period.

Postoji više različitih postupaka regresije, ali za potrebe izrade ovog modela korištena su dva najčešća postupka za treniranje modela i postizanje što boljih rezultata na temelju ovakvih ili sličnih podataka, a to su:

- Neuronska mreža – tehnika strojnog učenja koja nastoji oponašati rad neurona u ljudskom mozgu za učenje. Isprva je nestabilna i nakon određenih iteracija nad podacima, prilagođava se, te se tako povećava točnost. Sastoji se o seta međusobno povezanih slojeva, od kojih svaki sloj ima proizvoljan broj čvorova. Opterećenja na rubovima se određuju pomoću ulaznih podataka trenirane neuronske mreže. Smjer grafa određuje se iz ulaznih podataka kroz skriveni sloj gdje su svi čvorovi povezani opterećenim rubovima na čvorove sljedećeg sloja. Zadatci predviđanja se lako rješavaju samo jednim ili više skrivenih slojeva. Regresija neuronskom mrežom je vrsta nadziranog strojnog učenja [26].
- Linearna regresija – osnovna i najčešće korištena vrsta regresijske analize. Pokušava modelirati odnos između dvije varijable upotrebom linearne jednadžbe na promatrane podatke. Jedna varijabla se smatra eksplanatornom varijablom, a druga zavisnom.

Često se parametri podešavaju uz pomoć metode najmanjih kvadrata, te takav model linearno ovisi o nepoznatim parametrima podataka [27].

Slika 4.7 prikazuje izrađeni model strojnog učenja unutar *Azure Machine Learing Studia*. Kreće se od učitavanja podataka, te uklanjanja redaka u slučaju da nedostaje pokoji podatak. Podatci se dijele u omjeru 0.75/0.25, gdje 75% podataka ide na treniranje modela, a preostalih 25% na model vrednovanja rezultata.



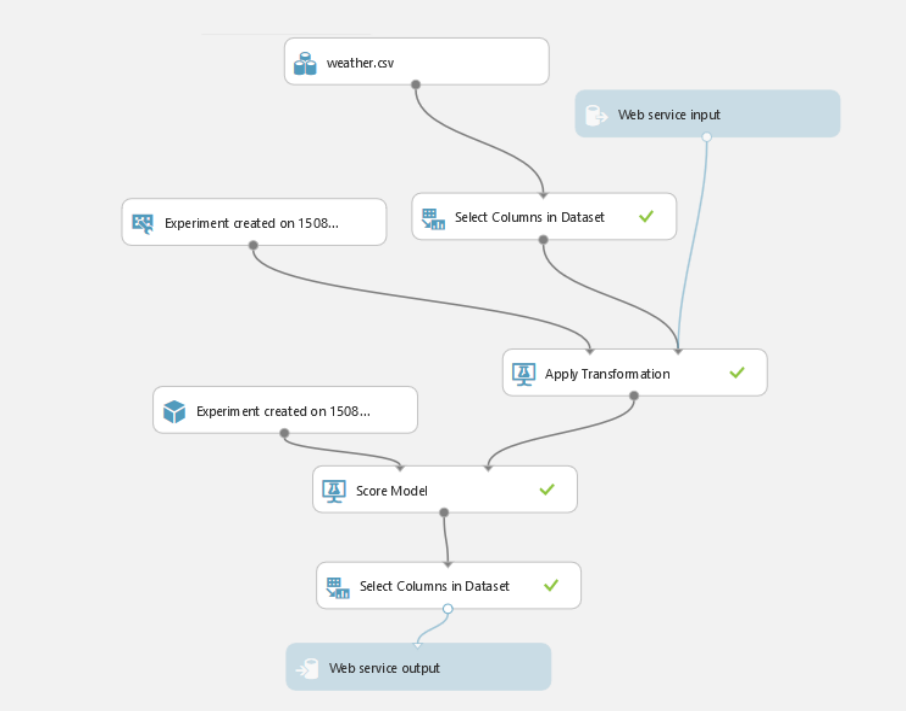
Slika 4.7. Model strojnog učenja

Nakon treniranja dva spomenuta regresijska modela nad odabranim podacima, uspoređeni su rezultati. „*Evaluate Model*“ uspoređuje dobivene rezultate te ih opisuje različitim parametrima. Kod usporedbe u obzir se uzima vrijednost pogreške, koja treba biti što manja, tj. što bliže 0, te koeficijent određivanja koji treba biti što bliži 1. Na slici 4.8 prikazani su rezultati dobiveni pomoću „*Evaluate Model-a*“.



Slika 4.8. Rezultati dobiveni pomoću „*Evaluate Model-a*“

Na osnovu dobivenih rezultata odabrana je linearna regresija. Pomoću „*Train Model-a*“ dobivenog linearnom regresijom, kreiran je web usluga prikazana na slici 4.9 s kojim će komunicirati web aplikacija za potrebe predviđanja. Nakon pokretanja web usluge *Azure Studio* generira krajnju točku na koju se šalje *POST* zahtjev uz slanje i *API* ključa za pristup.



Slika 4.9. Razvijena web usluga

4.3. Primjena i rezultati testova

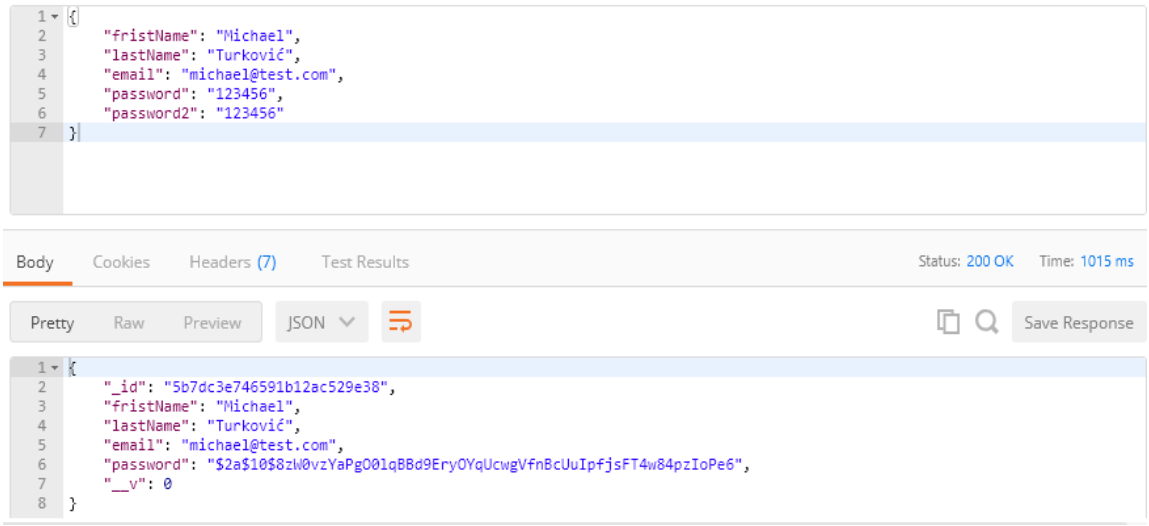
Kao što i „*pravilo*“ kaže: „*Što je kod aplikacije čišći, lakše ga se može testirati*“. Pridržavanjem tog pravila razvijana je aplikacija, te je podijeljena na dva dijela kao što će biti podijeljeni i testovi. Prvi dio ili „*Backend usluga*“ odgovorana je za pružanje usluga i komunikaciju s bazom podataka, dok „*Frontend*“ dio služi za prikazivanje sadržaja, te omogućuje korisniku upravljanje podacima.

Testiranje poslužiteljske i klijentske strane odrađeno je unutar radne okoline te u tim slučajevima nisu korišteni drugi alati. Za potrebe ručnog testiranja korišten je alat *Postman* koji će biti objašnjen u poglavlju 4.3.1, a za potrebe funkcionalnog testiranja korišten je alat *Cypress* opisan u poglavlju 4.3.4.

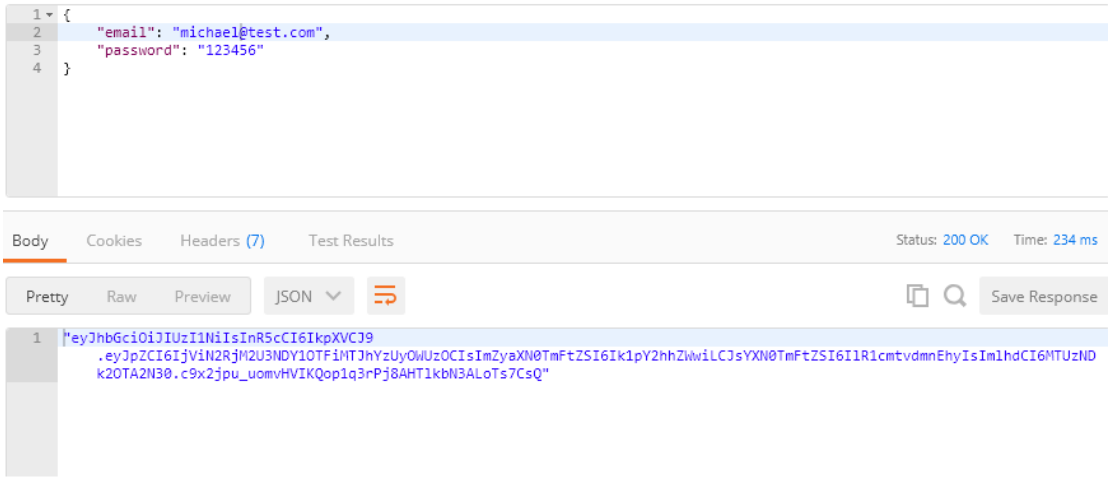
4.3.1. Ručno testiranje

Tijekom implementacije *API-ja,* za provjeru ispravnosti pojedinih dijelova upotrebljavan je alat *Postman*. *Postman* je alat koji omogućuje testiranje *RESTful API-ja* na jednostavan i brz način bez potrebe za pisanjem automatskih testova.

Na početku je kreiran *API* za *Prijavu* i *Registraciju* korisnika, te su ručni testovi pomoću alata *Postman* prikazani na slikama 4.10 i 4.11.

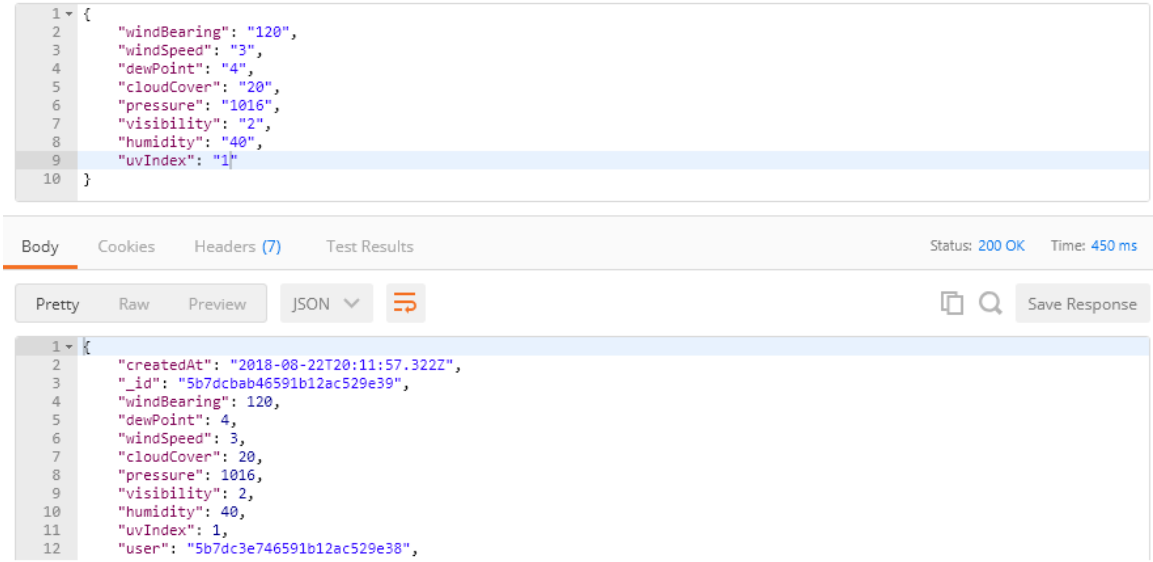


Slika 4.10. Provjera registracije korisnika

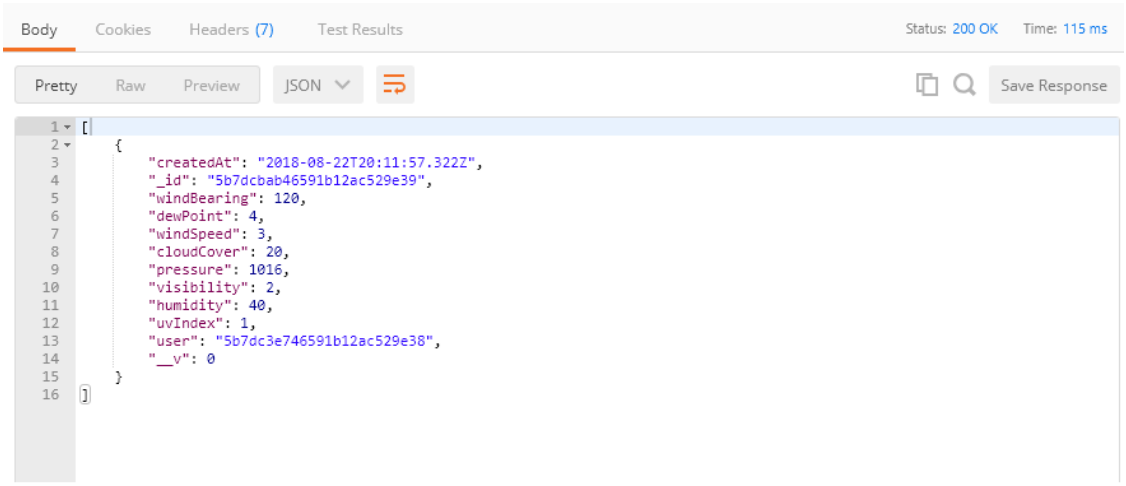


Slika 4.11. Provjera prijave korisnika

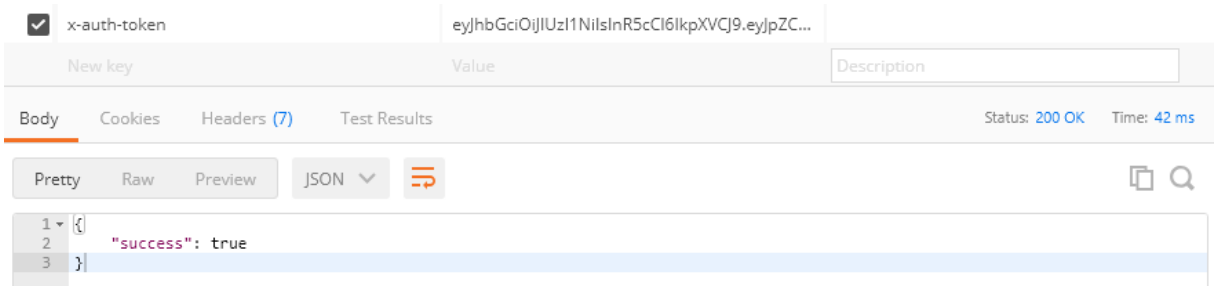
Kod registracije korisnika povratna informacija je registrirani korisnik, te kod prijave generirani *JSON web token*. U oba slučaja HTTP status odgovora je 200, što govori da su funkcionalnosti ispravno implementirane. Na slikama 4.12, 4.13 i 4.14 prikazana je provjera spremanja, dohvaćanja i brisanja podataka iz baze. Generirani *token* vraćen prilikom prijave korisnika biti će uključen u zaglavlje svakog zahtjeva rada s podacima.



Slika 4.12. Provjera spremanja podataka



Slika 4.13. Provjera dohvaćanja podataka



Slika 4.14. Provjera brisanja podataka

U svim slučajevima povratna informacije je HTTP statusa 200, tako da su sve funkcionalnosti ispravno implementirane. Alat *Postman* je na jednostavan i brz način omogućio provjeru ispravnosti i olakšao implementaciju poslužiteljske strane.

4.3.2. Testiranje poslužiteljske strane

Na poslužiteljskoj strani većina operacija uključuje stalnu komunikaciju s bazom podataka, te takvi testovi spadaju u vrstu integracijskog testiranja. Dok je jediničnim testovima provjeravana validacija korisničkog unosa, te operacije generiranja i postavljanja *tokena* u zaglavlje HTTP zahtjeva.

Programski kod 4.7 pokazuje implementaciju jediničnog testa za testiranje funkcionalnosti generiranja *tokena*.

```
describe('Auth middleware', () => {
    it('should populate req.user with payload of valid JWT', () => {
        const user = {
            id: mongoose.Types.ObjectId().toHexString(),
        }

        const token = new User(user).generateAuthToken(user);
        const req = {
            header: jest.fn().mockReturnValue(token)
        };
        const res = {};
        const next = jest.fn();

        auth(req, res, next);

        expect(req.user).toMatchObject(user);
    });
});
```

Programski kod 4.7. Dio koda za testiranje funkcije generiranja tokena

U programskom kodu 4.8 prikazan je dio koda odgovoran za testiranje validacije korisničkog unosa tijekom prijave. Prikazan su dva test slučaja, prvi opisuje ispravan korisnički unos bez poruke o pogrešci. U drugom test slučaju proslijeđena je pogrešna email adresa i očekuje se poruka o neispravnoj email adresi.

```
describe('Validate login input', () => {
    let user = {
        email: 'korisnik@test.com',
        password: '123456',
    }
    it('should not return errors', () => {
        const data = validateLogin(user);
        expect(data.errors).toMatchObject({})
    });
    it('should return error if email is not valid', () => {
        user.email = 'korisnik';
        const data = validateLogin(user);
        expect(data.errors.email).toBe('Email nije ispravan');
    });
});
```

Programski kod 4.8. Dio koda za testiranje validacije korisničkog unosa tijekom prijave

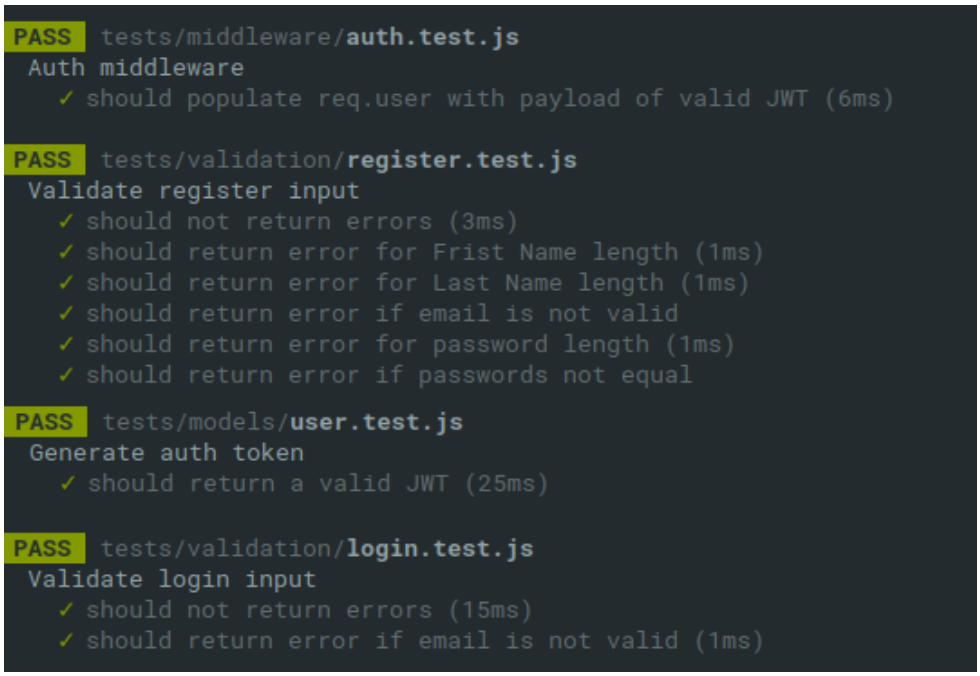
Programski kod 4.9 prikazuje test validacije korisničkog unosa prilikom registracije korisnika. U ovom slučaju prikazana su tri test slučaja. Prvi test slučaj kao i u prethodnom primjeru pokazuje ispravan korisnički unos bez poruke o pogrešci. U drugom test slučaju proslijeđeno je ime gdje se

javlja poruka o nedovoljnom broju znakova. Zadnji test slučaj opisuje scenarij u kojem se lozinke ne podudaraju.

```
describe('Validate register input', () => {
    let user = {
        fristName: 'Test',
        lastName: 'Korisnik',
        email: 'korisnik@test.com',
        password: '123456',
        password2: '123456'
    }
    it('should not return errors', () => {
        const data = validateRegister(user);
        expect(data.errors).toMatchObject({})
    });
    it('should return error for Frist Name length', () => {
        user.fristName = 'J';
        const data = validateRegister(user);
        expect(data.errors.fristName).toBe('Ime mora biti između 2 i 30
znakova');
    });
    it('should return error if passwords not equal', () => {
        user.password = '123456';
        user.password2 = '1234567';
        const data = validateRegister(user);
        expect(data.errors.password2).toBe('Lozinke se moraju podudarati');
    });
});
```

Programski kod 4.9. Dio koda za testiranje validacije korisničkog unosa tijekom registracije

Na slici 4.15 prikazani su rezultati jediničnih testova za validaciju korisničkog unosa prilikom prijave i registracije, te testovi kod generiranja *tokena* i dekodiranja istog. Testiranje JavaScript koda se izvršava pokretanjem naredbe u terminalu. Nakon završenog procesa testiranja, u terminalu se dobije ispis svih provedenih testova, testovi koji su uspješno implementirani označeni su zelenom bojom.



Slika 4.15. Rezultati jediničnih testova

Nakon što su svi jedinični testovi uspješno prošli, testirane su funkcionalnosti prijave i registracije korisnika, gdje je za testiranje korištena komunikacija s bazom podataka. U programskom kodu 4.10 prikazan je test slučaj za uspješno registriranje korisnika. Očekivani status HTTP odgovora je 200, a tijelo odgovora mora se podudarati s proslijeđenim podacima. Na kraju se vrši provjera pozivom upita na bazu podataka, gdje proslijeđena email adresa mora postojati, a lozinka mora biti kriptirana. Ovakav test pripada razini integracijskog testiranja.

```
describe('POST /users/register', () => {
    it('should create a new user', (done) => {
        request(server)
            .post('/users/register')
            .send(_.pick(user, ['fristName', 'lastName', 'email',
'password', 'password2']))
            .expect((res) => {
                expect(res.status).toBe(200);
                expect(res.body.fristName).toBe(user.fristName);
                expect(res.body.lastName).toBe(user.lastName);
                expect(res.body.email).toBe(user.email);
            })
            .end((err) => {
                if (err) return done(err);
                User.findOne({email: user.email})
                    .then((data) => {
                        expect(data).toBeTruthy();
                        expect(data.password).not.toBe(user.password);
                        done();
                    }).catch((e) => done(e));
            })
    });
});
```

Programski kod 4.10. Dio koda za testiranje registracije korisnika

U programskom kodu 4.11 prikazana je implementacije dva test slučaja prilikom prijave korisnika. Prvi test slučaj prikazuje uspješnu prijavu korisnika, dok je za drugi korištena pogrešna lozinka. Tijelo odgovora drugog test slučaja sadrži poruku o netočnoj lozinki, te status HTTP odgovora 400.

```
describe('POST /users/auth', () => {
    let user = {
        email: 'user@test.com',
        password: '123456',
    }
    it('should login user and return token', (done) => {
        request(server)
            .post('/users/auth')
            .send(_.pick(user, ['email', 'password']))
            .expect((res) => {
                expect(res.status).toBe(200);
                expect(res.body).toBeTruthy();
            })
            .end(done);
    });
    it('should not login user if password is incorrect', (done) => {
        user.email = 'user@test.com';
        user.password = '1234';

        request(server)
            .post('/users/auth')
            .send(_.pick(user, ['email', 'password']))
            .expect((res) => {
                expect(res.status).toBe(400);
                expect(res.body.password).toBe('Netočna lozinka');
            })
            .end(done);
    });
});
```

Programski kod 4.11. Dio koda za testiranje prijave korisnika

Slika 4.16 pokazuje uspješno završene testove za prijavu i registraciju korisnika. U samu implementacija funkcionalnosti uključeni su procesi validacije i generiranja *tokena*.

```
PASS tests/routes/user.test.js
POST /users/register
  ✓ should create a new user (512ms)
  ✓ should not create a new user if email already exist (24ms)
  ✓ should not create a new user if email is invalid (5ms)
  ✓ should not create a new user if passwords dont match (4ms)
POST /users/auth
  ✓ should login user and return token (184ms)
  ✓ should not login user if email is incorrect (5ms)
  ✓ should not login user if password is incorrect (175ms)
```

Slika 4.16. Rezultati integracijskih testova za prijavu i registraciju korisnika

Testiranje rada s podacima također pripada integracijskoj razini testiranja. Kod rada s podacima testiran je unos, uklanjanje i dohvaćanja podataka, te je testirana i funkcionalnost predviđanja temperature. U programskom kodu 4.12 prikazan je test slučaj za uspješno spremanje podataka, gdje se proslijeđene vrijednosti nakon spremanja očekuju kao povratna informacija odgovora.

```
describe('POST /data', () => {
    it('should save a new data', (done) => {
        request(server)
            .post('/data/')
            .set('x-auth-token', token)
            .send(_.pick(data, ['windBearing', 'dewPoint', 'windSpeed',
'cloudCover', 'pressure', 'visibility', 'humidity', 'uvIndex']))
            .expect((res) => {
                expect(res.status).toBe(200);
                expect(res.body).toHaveProperty('_id');
                expect(res.body).toHaveProperty('humidity');
            })
            .end(done);
    });
});
```

Programski kod 4.12. Dio koda za testiranje spremanja podataka

Programski kod 4.13 prikazuje testiranje funkcionalnosti brisanja određenog skupa podataka. Uzima se *id* skupa podataka koji se želi ukloniti iz baze podataka i on se šalje zajedno sa DELETE zahtjevom.

```
describe('DELETE /:id', () => {
    let id = '5b9024b1d342701cceff415f'
    it('should delete selected data', (done) => {
        request(server)
            .delete(`/data/${id}`)
            .set('x-auth-token', token)
            .expect((res) => {
                expect(res.status).toBe(200);
                expect(res.body.success).toBeTruthy();
            })
            .end(done);
    });
});
```

Programski kod 4.13. Dio koda za testiranje uklanjanja podatka

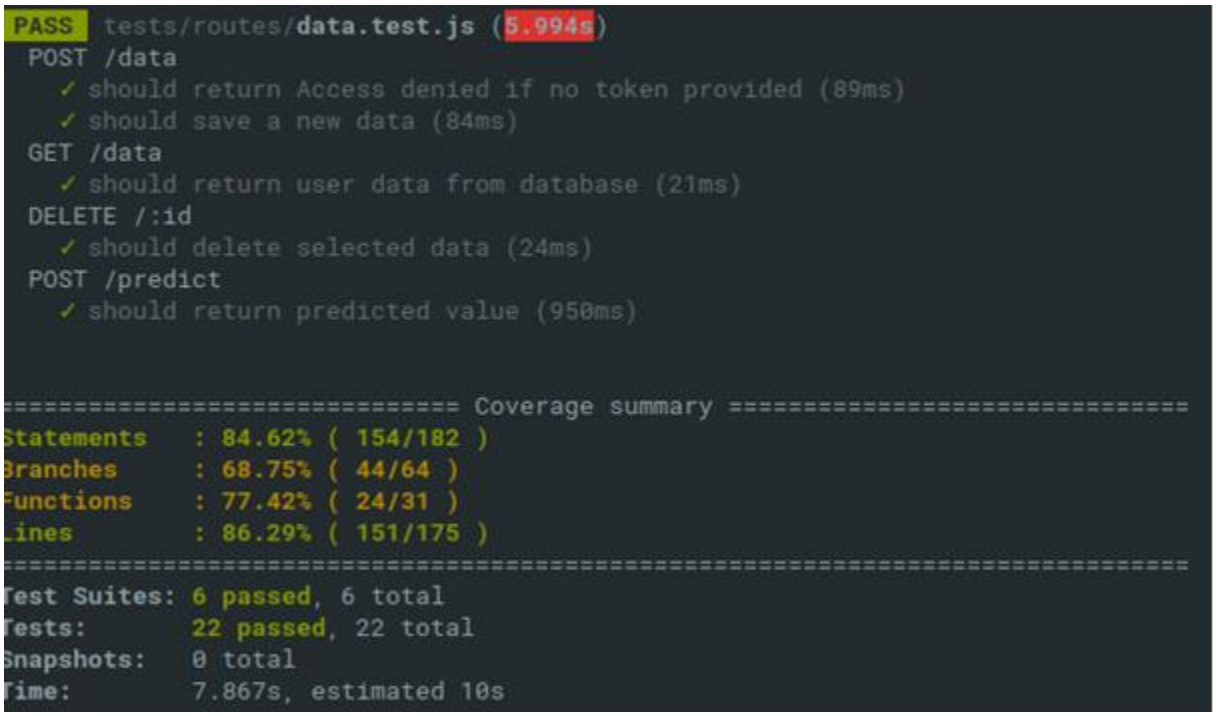
U programskom kodu 4.14 prikazano je testiranje dohvaćanja podataka. Nakon upita, ukoliko u bazi podataka postoje spremljene vrijednosti, duljina tijela odgovora mora biti različita od nula.

```
describe('GET /data', () => {
    it('should return user data from database', (done) => {
        request(server)
            .get('/data/')
            .set('x-auth-token', token)
            .expect((res) => {
                expect(res.status).toBe(200);
                expect(res.body.length).toBeGreaterThan(0);

            })
            .end(done);
    });
});
```

Programski kod 4.14. Dio koda za testiranje dohvaćanja podataka

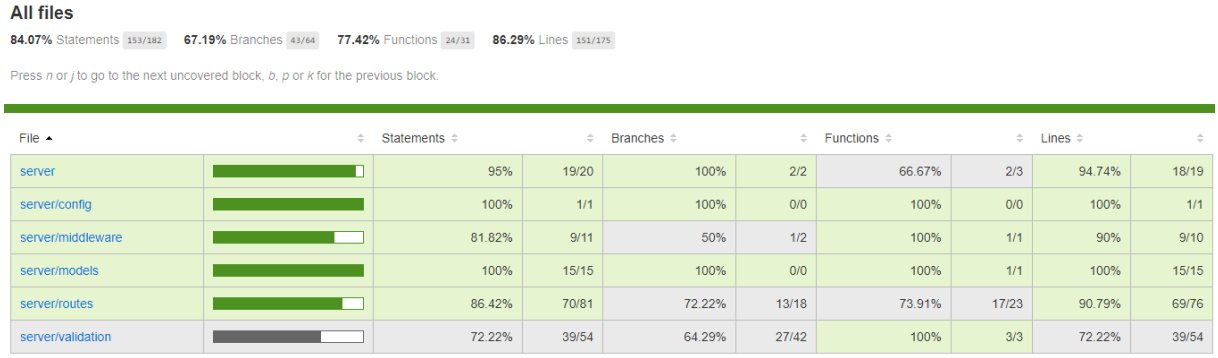
Na slici 4.17 prikazani su uspješno završenih testova za rad s podacima.



Slika 4.17. Rezultati integracijskih testova rada s podacima

Nakon svih provedenih testova generiran je izvještaj pokrivenosti koda testovima, koji je prikazan na slici 4.18. Izvještaj pokrivenosti automatski generira biblioteka za testiranje. Iz izvještaja se

može vidjeti da je ukupna pokrivenost koda 84.07%. Također se može vidjeti pokrivenost linija, funkcija i grana koda.



All files

84.07% Statements 153/182 67.19% Branches 43/64 77.42% Functions 24/31 86.29% Lines 151/175

Press n or j to go to the next uncovered block, b, p or k for the previous block.

File	Statements		Branches		Functions		Lines	
server	95%	19/20	100%	2/2	66.67%	2/3	94.74%	18/19
server/config	100%	1/1	100%	0/0	100%	0/0	100%	1/1
server/middleware	81.82%	9/11	50%	1/2	100%	1/1	90%	9/10
server/models	100%	15/15	100%	0/0	100%	1/1	100%	15/15
server/routes	86.42%	70/81	72.22%	13/18	73.91%	17/23	90.79%	69/76
server/validation	72.22%	39/54	64.29%	27/42	100%	3/3	72.22%	39/54

Slika 4.18. Izvještaj pokrivenosti koda testovima na poslužiteljskoj strani

4.3.3. Testiranje klijentske strane

Složenost aplikacije na klijentskoj strani je veća, te tako sadrži i više testova, uglavnom jediničnih. Prvi korak je ispitivanje ispravnosti komponenti, te provjera da li se prikazuju na korisničkom sučelju bez grešaka. Uz već spomenutu biblioteku za testiranje *Jest*, za testiranje komponenti korištena je i biblioteka *Enzyme*. *Enzyme* je biblioteka koja omogućuje lakše testiranje React aplikacija, simulirajući izlaz pojedinih komponenti. U programskom kodu 4.15 prikazan je dio testa glavne komponente aplikacije, u kojoj su povezane sve ostale pod komponente. Prvi test slučaj provjerava ispravnost prikaza komponente, dok je u drugom testirana funkcionalnost prebacivanja rute ukoliko korisnik nije prijavljen.

```
import { shallow } from 'enzyme';
import { Redirect } from 'react-router-dom';

describe('WeatherApp component testing', () => {
    const wrapper = shallow(<WeatherApp/>);

    it('WeatherApp renders without crashing', () => {
        expect(wrapper).toMatchSnapshot();
    });

    it('WeatherApp renders redirect route', () => {
        expect(wrapper.find('Route[exact=true][path="/"]').first()
.prop('render')()).toEqual(<Redirect to="/login"/>);
    });
});
```

Programski kod 4.15. Dio koda za testiranje glavne komponente

Programski kod 4.16 prikazuje tri test slučaja kod testiranja navigacijske komponente. U prvom test slučaju prikazano je testiranje ispravnosti komponente, da li se ispravno prikazuje unutar korisničkog sučelja. Drugi slučaj također prikazuje testiranje ispravnog prikaza komponente, ali u ovom slučaju kada je korisnik prijavljen. Zadnji slučaj prikazuje testiranje linka za odjavu korisnika.

```
describe('Header test', () => {
    const initialStore = {
        auth: {
            isAuthenticated: false,
            isRegisterSuccess: false,
            user: {}
        },
        logout: jest.fn(),
    };
    const mockStore = configureStore([thunk]);

    it('ConnectedHeader renders without crashing', () => {
      expect(shallow(<ConnectedHeader
      store={mockStore(initialStore)}/>)).toMatchSnapshot();
    });
    const mockProps = {
        auth: {
            isAuthenticated: true,
            isRegisterSuccess: false,
            user: 'test',
        }
    };

    const mockLogout = jest.fn();
    const wrapper = shallow(<Header {...mockProps} logout={mockLogout}/>);

    it('Header component renders without crashing with authenticated user',
      () => {
        expect(wrapper).toMatchSnapshot();
    });

    it('Header component calls logout', () => {
        const e = { preventDefault: () => jest.fn() };
        expect(wrapper.find('a.nav-link').length).toBe(1);
        wrapper.find('a.nav-link').simulate('click', e);
        expect(mockLogout).toBeCalled();
    });
});
```

Programski kod 4.16. Dio koda za testiranje komponente navigacije

Nakon uspješno testiranih komponenti na redu su bile akcije, one omogućuju interakciju između korisnika i komponenti.

U programskom kodu 4.17 prikazan je primjer testa akcije za uspješno spremanje podataka. Implementira se lažno skladište podataka, te se šalje određeni podatak. Nakon uspješne akcije podatak pohranjen u skladište mora odgovarati poslanom podatku.

```
describe('dataActions test', () => {
    const mockStore = configureMockStore([thunk]);
    beforeEach(() => moxios.install());
    afterEach(() => moxios.uninstall());

    it('creates successful ADD_DATA ', async (done) => {
        moxios.stubRequest('http://localhost:5000/data', {
            status: 201,
            response: 'some data'
        });
        const expectedActions = [
            {type: 'CLEAR_ERRORS'},
            {type: 'ADD_DATA', payload: 'some data'}
        ];
        const store = mockStore({});
        const mockHistory = {
            push: jest.fn()
        };
        await store.dispatch(actions.addData('some data', mockHistory))
            .then(() => {
                expect(store.getActions()).toEqual(expectedActions);
            });
        done();
    });
});
```

Programski kod 4.17. Dio koda za testiranje spremanja podataka

Programski kod 4.18 prikazuje test za slučaj ukoliko se prilikom dohvaćanja podataka pojavi greška. Šalje se lažni zahtjev s očekivanim status odgovora. Nakon slanja zahtjeva, vrijednost odgovora pohranjena u skladište mora odgovarati parametrima u pozvanoj akciji.

```
it('creates GET_ALL_DATA with errors', async (done) => {
        moxios.stubRequest('http://localhost:5000/data', {
            status: 404,
            response: {}
        });
        const expectedActions = [
            {type: 'DATA_LOADING'},
            {type: 'GET_ALL_DATA', payload: null}
        ];
        const store = mockStore({});
        await store.dispatch(actions.getAllData())
            .then(() => {
                expect(store.getActions()).toEqual(expectedActions);
            });
        done();
    });
});
```

Programski kod 4.18. Dio koda za testiranje dohvaćanja podataka ukoliko postoji greška

U programskom kodu 4.19 prikazan je dio testiranja reduktora. Reduktor služi za praćenje promjena u aplikaciji tijekom određene akcije i slanje podataka prema skladištu. U prvom slučaju vraća unaprijed definirano stanje ukoliko su proslijeđene neispravne vrijednosti. Drugi slučaj pokazuje slanje vrijednosti prema skladištu, ukoliko se prilikom predviđanja nije dogodila greška.

```
const initialState = {
    allData: [],
    data: {},
    loading: false,
    predictedData: null
};

describe('dataReducer test', () => {
    it('should return the initial state', () => {
        expect(dataReducer(undefined, {})).toEqual(initialState);
    });

    it('should handle GET_PREDICTED_DATA', () => {
        expect(dataReducer(initialState, {
            type: 'GET_PREDICTED_DATA',
            payload: 'predicted data',
        })).toEqual({
            ...initialState,
            predictedData: 'predicted data',
        });
    });
});
```

Programski kod 4.19. Dio koda za testiranje reduktora

Programski kod 4.20 pokazuje dio reduktora koji je odgovoran za uklanjanje podatka. Poziva se funkcija reduktora koja sadrži tip zahtjeva i *id* podatka. Nakon uspješnog uklanjanja vraća se stanje s preostalim podacima.

```
it('should handle DELETE_DATA', () => {
    const stateDeleteData = {
        ...initialState,
        allData: [{_id: 24, data: 'test'}, {_id: 231, data: 'test'}],
    };
    expect(dataReducer(stateDeleteData, {
        type: 'DELETE_DATA',
        payload: 24,
    })).toEqual({
        ...stateDeleteData,
        allData: [{_id: 231, data: 'test'}],
    });
});
```

Programski kod 4.20. Dio koda za testiranje reduktora prilikom uklanjanja podatka

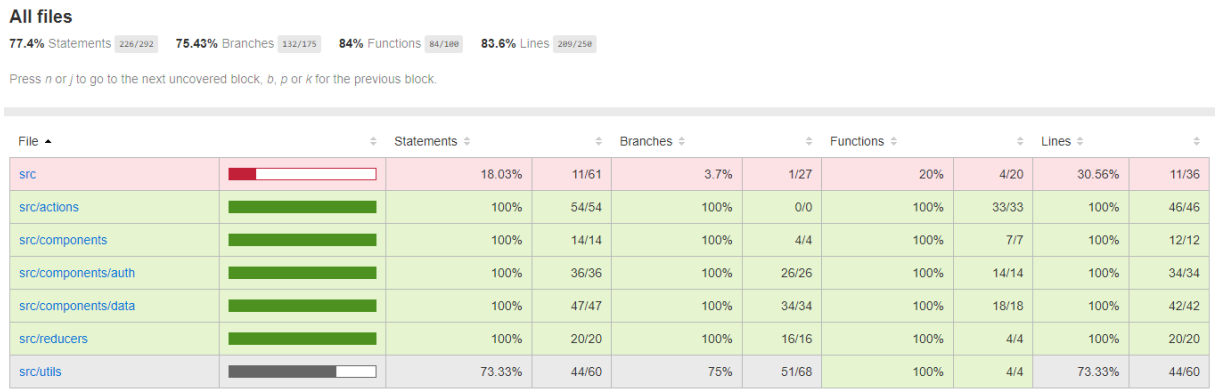
Na slici 4.19 prikazani su rezultati testova klijentske strane, ukupan broj testova i test slučajeva te potrebno vrijeme za izvršavanje

```
PASS  src/components/__tests__/data/DataCard.test.js
PASS  src/components/__tests__/auth/Register.test.js
PASS  src/components/__tests__/data/PostData.test.js
PASS  src/components/__tests__/auth/Login.test.js
PASS  src/components/__tests__/Header.test.js
PASS  src/WeatherApp.test.js
PASS  src/actions/__tests__/dataActions.test.js
PASS  src/actions/__tests__/authAction.test.js
PASS  src/App.test.js
PASS  src/components/__tests__/data/DataInfo.test.js
PASS  src/components/__tests__/data/MyData.test.js
PASS  src/reducers/__tests__/dataReducer.test.js
PASS  src/components/__tests__/ProtectedRoute.test.js
PASS  src/reducers/__tests__/errorReducer.js
PASS  src/reducers/__tests__/authReducer.test.js
PASS  src/components/__tests__/Loading.test.js

Test Suites: 16 passed, 16 total
Tests:       72 passed, 72 total
Snapshots:   22 passed, 22 total
Time:        4.137s
```

Slika 4.19. Rezultati testova klijentske strane

Kao i na poslužiteljskoj strani generiran je izvještaj pokrivenosti koda testovima, koji je prikazan na slici 4.20. U ovom slučaju pokrivenost koda je 77.4%, zbog toga što se neke funkcije ne moraju testirati, one su automatski generirane u samom postavljanju aplikacije.



All files

77.4% Statements 226/292 **75.43%** Branches 132/175 **84%** Functions 84/100 **83.6%** Lines 209/250

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
src	18.03%	11/61	3.7%	1/27	20%	4/20	30.56%	11/36
src/actions	100%	54/54	100%	0/0	100%	33/33	100%	46/46
src/components	100%	14/14	100%	4/4	100%	7/7	100%	12/12
src/components/auth	100%	36/36	100%	26/26	100%	14/14	100%	34/34
src/components/data	100%	47/47	100%	34/34	100%	18/18	100%	42/42
src/reducers	100%	20/20	100%	16/16	100%	4/4	100%	20/20
src/utils	73.33%	44/60	75%	51/68	100%	4/4	73.33%	44/60

Slika 4.20. Izvještaj pokrivenosti koda testovima na klijentskoj strani

4.3.4. Funkcionalno testiranje

Nakon što je aplikacija u potpunosti završena i testirana, ostalo je još funkcionalno testiranje, koje provjerava kompletnu ispravnost aplikacije automatskim testovima. Funkcionalno testiranje provedeno je uz pomoć alata *Cypress*. To je alat koji se koristi za automatsko testiranje web aplikacija. Sadrži niz dodatnih alata, pa je samo postavljanje testova vrlo lako. Uključuje se u web aplikaciju kao JavaScript skripta, te tako omogućuje kontroliranje web preglednika.

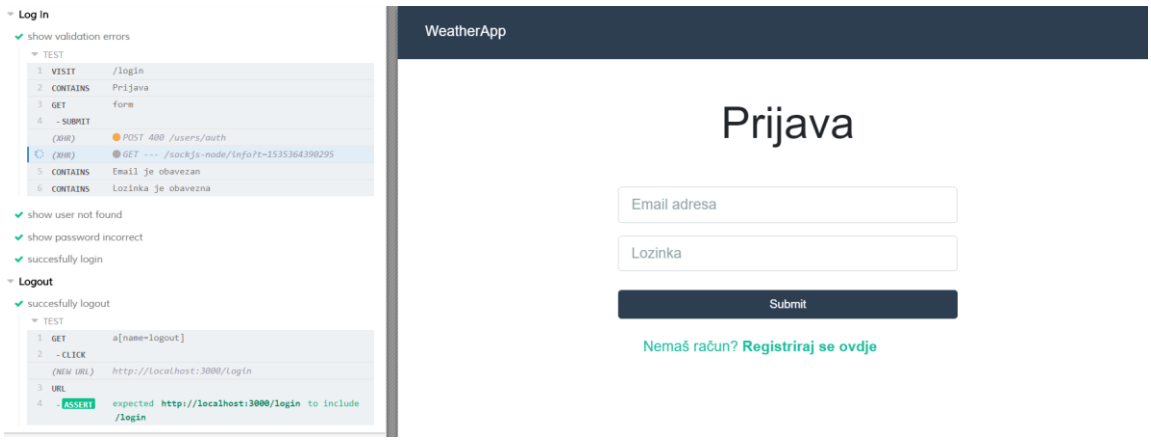
U programskom kodu 4.21 prikazana je implementacija funkcionalnog testa za uspješnu prijavu korisnika. Posjećuje se ruta za prijavu i unose podaci. Nakon uspješne prijave korisnika prebacuje se na stranicu s podacima.

```
describe('Log In', () => {
    it('succesfully login', () => {
        cy.visit('/login');
        cy.contains('Prijava');
        cy.get('input[name=email]').type('michael@test.com');
        cy.get('input[name=password]').type('123456');
        cy.get('form').submit();
        cy.url({ timeout: 3000 }).should('includes', '/data');
    });
});
```

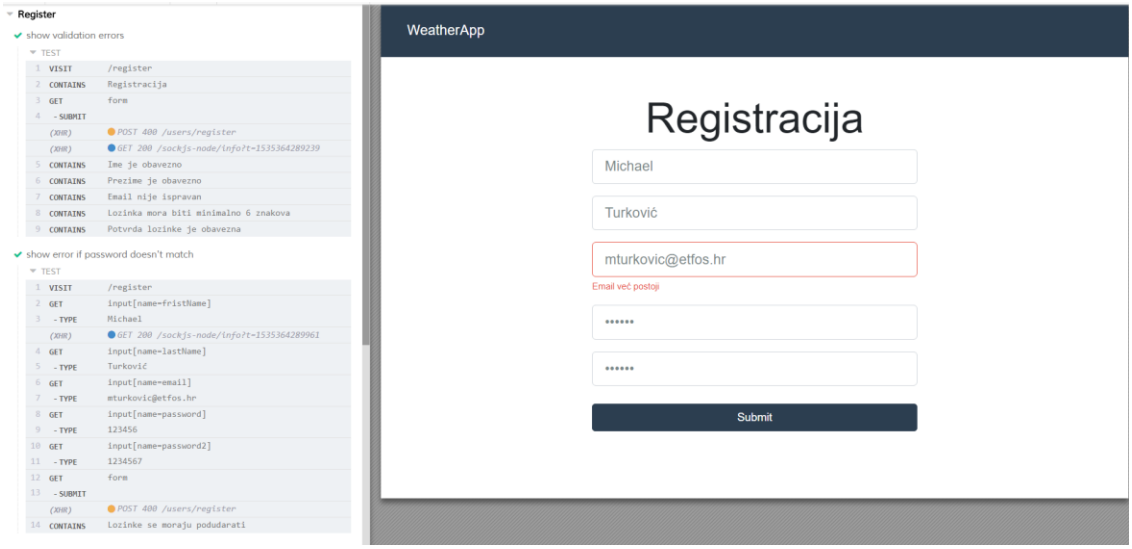
Programski kod 4.21. Dio koda za funkcionalnog testiranje uspješne prijave korisnika

Na slici 4.21 prikazan je kompletan proces funkcionalnog testa za prijavu i odjavu korisnika. S lijeve strane prikazan je izbornik s statusom testova, koji prikazuje uspješne i neuspješne test

slučajeve te vrijeme potrebno za izvršavanje. Odabirom pojedinog testa mogu se vidjeti detalji zapisa tijekom izvršavanja testa. Na desnoj strani se mogu pratiti naredbe koje se izvršavaju u stvarnom vremenu, te je moguće pregledati ili ispraviti svaku naredbu.



Slika 4.21. Funkcionalni test prijave i odjave korisnika



Slika 4.22. Funkcionalno testiranje registracije korisnika

U programskom kodu 4.22 prikazan je dio koda za uspješno spremanje podataka. Nakon spremanja podataka korisnika se prebacuje na stranicu s spremljenim podacima.

```
it('succesfully post data', () => {
        cy.visit('/data/new');
        cy.get('input[name=humidity]').type('45');
        cy.get('input[name=dewPoint]').type('4');
        cy.get('input[name=cloudCover]').type('20');
        cy.get('input[name=windBearing]').type('120');
        cy.get('input[name=uvIndex]').type('2');
        cy.get('input[name=visibility]').type('12');
        cy.get('input[name=pressure]').type('1016');
        cy.get('input[name=windSpeed]').type('5');
        cy.get('form').submit();
        cy.url({ timeout: 3000 }).should('includes', '/data');
        cy.get('.card').should(($card) => expect($card).to.have.length(1))
});
```

Programski kod 4.22. Dio koda za funkcionalnog testiranje spremanja podataka

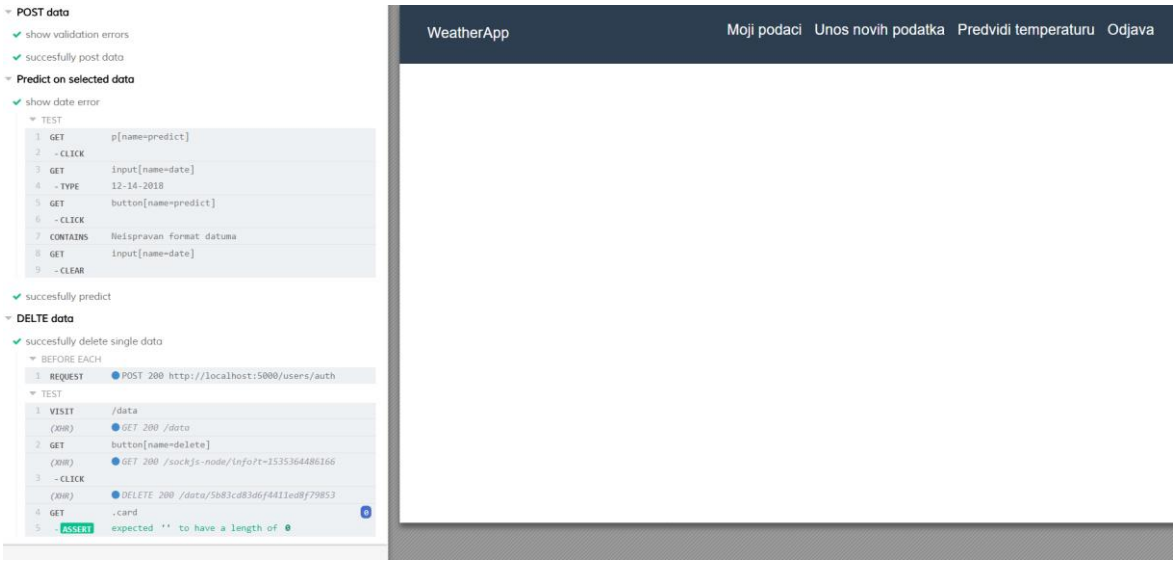
Programski kod 4.23 prikazuje dio koda funkcionalnog testa za uklanjanje odabranog skupa podataka.

```
describe('DELTE data', () => {
    beforeEach(() => {
        cy.login()
    });

    it('succesfully delete single data', () => {
        cy.visit('/data');
        cy.get('button[name=delete]').click();
        cy.get('.card').should(($card) => expect($card).to.have.length(0))
    });
});
```

Programski kod 4.23. Dio koda za funkcionalno testiranje uklanjanje podataka

Slika 4.23 prikazuje proces funkcionalnog testiranja prilikom rada s podacima.



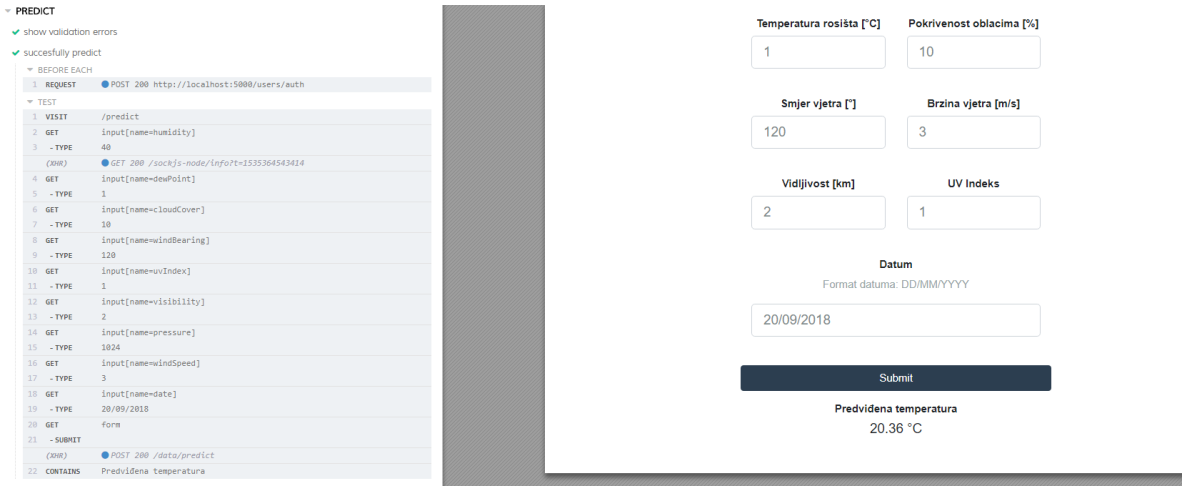
Slika 4.23. Funkcionalno testiranje rada s podacima

U programskom kodu 4.24 prikazan je funkcionalni test predviđanja temperature. Nakon popunjavanja obrasca i slanja upita, odgovor zahtjeva sadrži povratnu vrijednost temperature koja se prikazuje unutar polja s imenom „*Predviđena temperatura*“.

```
it('succesfully predict', () => {
        cy.visit('/predict');
        cy.get('input[name=humidity]').type('40');
        cy.get('input[name=dewPoint]').type('1');
        cy.get('input[name=cloudCover]').type('10');
        cy.get('input[name=windBearing]').type('120');
        cy.get('input[name=uvIndex]').type('1');
        cy.get('input[name=visibility]').type('2');
        cy.get('input[name=pressure]').type('1024');
        cy.get('input[name=windSpeed]').type('3');
        cy.get('input[name=date]').type('20/09/2018');
        cy.get('form').submit();
        cy.contains('Predviđena temperatura');
});
```

Programski kod 4.24. Dio koda za funkcionalno testiranje predviđanja temperature

Slika 4.24 prikazuje proces funkcionalnog testiranja predviđanja temperature na osnovu proizvoljno unesenih vrijednosti.



Slika 4.24. Funkcionalno testiranje funkcije predviđanja

5. ZAKLJUČAK

Cilj ovog rada bio je razviti web aplikaciju postupkom razvoja pokretanog testiranjem. Razvijeno programsko rješenje omogućuje korisnicima spremanje podataka prikupljenih s različitih meteoroloških senzora, te na osnovu tih podataka omogućeno je predviđanje temperature za određeno razdoblje. Tijekom razvoja pridržavalo se postupka ponavljanja vrlo kratkih razvojnih ciklusa, koji su pretvoreni u specifične test slučajeve, a zatim se programsko rješenje poboljšavalo provodeći nove dodatne testove. Refaktorizacijom koda poboljšala se kvaliteta i omogućila rana identifikacija pogrešaka. U svrhu osiguranja ispravnosti provođeni su jedinični, integracijski i funkcionalni testovi.

Kod testiranja poslužiteljske strane bila je potrebna stalna komunikacija s bazom podataka, te većina tih testova pripada integracijskoj razini testiranja. Prilikom izrade *REST API-ja* alat *Postman* je omogućio jednostavan i brz pristup testiranju određenih dijelova bez potrebe za pisanje automatskih testova. Na klijentskoj strani jediničnim testovima provjeravana je ispravnost prikaza komponentni na korisničkom sučelju, te funkcionalnosti akcija i reduktora. Nakon prolaska testova pomoću *Jest* biblioteke generirano je izvješće o pokrivenosti koda testovima. Pokrivenost koda testovima poslužiteljske strane je 84.07%, dok je kod klijentske strane 77.4%. Funkcionalni testovi su omogućili brz i jednostavan način testiranje aplikacije u stvarnom okruženju.

Sustav ima mogućnosti nadogradnje, a povezivanjem sustava s senzorima bilo bi omogućeno automatsko spremanje podataka, gdje bi korisniku bilo olakšano rukovanje aplikacijom. Moguće je još nadograditi i model strojnog učenja s stvarnim prikupljenim podacima korisnika, kako bi rezultati bili što realniji.

.

LITERATURA

[1] W.C. Hetzel, The Complete Guide to Software Testing, 2nd ed, Wellesley, Mass. Information Sciences, 1988.

[2] K. Sneha, G. M. Malle, Research on software testing techniques and software automation testing tools, IEEXplore Digital Library https://ieeexplore.ieee.org/document/8389562/, pristupljeno 25. lipnja 2018.

[3] J. Popović, Testiranje softvera u praksi: Pregled teorije testiranja sa primerima iz prakse, Računarski fakultet, Beograd, 2012

[4] G.J. Myers, The art of software testing, New York, Wiley, 1979

[5] C. Kaner, Testing Computer Software, Tab Books, Michigan University, 1993.

[6] What is Verification and Validation?, https://www.softwaretestinghelp.com/what-is-verification-and-validation/, pristupljeno 25. lipnja 2018.

[7] M.R.Lyu , Handbook of Software Reliability Engineering. McGraw-Hill, 1995.

[8] Laboratorijska vježba 4: Testiranje softvera, kolegij Računarstvo usluga i analiza podataka

[9] Software Testing Tutorial, https://www.guru99.com/software-testing.html, pristupljeno 10. kolovoza 2018.

[10] What is manual testing? https://blog.testlodge.com/what-is-manual-testing/, pristupljeno 21. kolovoza 2018.

[11] What is Automated Software Testing? https://www.inflectra.com/rapise/highlights/what-is-automated-software-testing.aspx pristupljeno 21.kolovoza 2018.

[12] Unit Testing, http://softwaretestingfundamentals.com/unit-testing/ pristupljeno 29. kolovoza 2018

[13] What are Unit Testing, Integration Testing and Functional Testing? https://codeutopia.net/blog/2015/04/11/what-are-unit-testing-integration-testing-and-functional-testing/, pristupljeno 25. kolovoza 2018.

[14] What is Integration testing?, http://tryqa.com/what-is-integration-testing/, pristupljeno 25. kolovoza 2018.

[15] Definition of System Testing, https://economictimes.indiatimes.com/definition/system-testing, pristupljeno 29. kolovoza 2018.

[16] White Box Testing, https://reqtest.com/testing-blog/white-box-testing-example/, pristupljeno 28. kolovoza 2018.

[17] Black Box Testing, https://test.io/black-box-testing/, pristupljeno 28. kolovoza 2018.

[18] R.S. Sangwan, P.A. La Plante, Test-Driven Development in Large Projects, , IEEXplore Digital Library, https://ieeexplore.ieee.org/document/1717338/, pristupljeno 6. rujna 2018.

[19] A. Koutifaris, Test Driven Development: what it is, and what is not, https://medium.freecodecamp.org/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2, pristupljeno 7.rujna 2018.

[20] D. Clerissi, M. Leotta, G. Reggio, F. Ricca, Test Driven Development of Web Applications: A Lightweight Approach, , IEEXplore Digital Library https://ieeexplore.ieee.org/document/7814511/, pristupljeno 7. rujna 2016.

[21] Test Pyramid, https://blog.primehammer.com/test-pyramid/, pristupljeno 15. kolovoza 2018.

[22] What is REST?, https://www.codecademy.com/articles/what-is-rest, pristupljeno 29. kolovoza 2018.

[23] Laboratorijska vježba 8: REST, kolegij Računarstvo usluga i analiza podataka

[24] Predavanje 5: Rudarenje podataka i strojno učenje Azure , kolegij Računarstvo usluga i analiza podataka

[25] Podatci za strojno učenje, https://www.kaggle.com/jeanmidev/smart-meters-in-london, pristupljeno 10. kolovoza. 2018.

[26] R. Gupta, Getting started with Neural Network for regression and Tensorflow, Medium https://medium.com/@rajatgupta310198/getting-started-with-neural-network-for-regression-and-tensorflow-58ad3bd75223, pristupljeno 25. kolovoza 2018.

[27] Linear Regression, http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm, pristupljeno 25. kolovoza 2018.

SAŽETAK

Svrha ovog rada je razviti web aplikaciju pokretanu testiranjem, te kroz testove prilikom razvoja osigurati kvalitetu i ispravnost. Ispravnost aplikacije osigurana je kroz testiranje pojedinih dijelova i funkcija. Tijekom razvoja, testiranje olakšava programeru pronalazak i brže ispravljanje grešaka. Za provedbu kvalitetnih testova potrebno je odabrati i određeni alat, a sve u ovisnosti o korištenoj tehnologiji. Za potrebe testiranja ove aplikacije korištene su tehnologije i alati koji omogućuju testiranje *JavaScript* koda. Testovi su podijeljeni na poslužiteljsku i klijentsku stranu. Na poslužiteljskoj strani testirane su funkcionalnosti komunikacije s bazom podataka te pružanja usluga klijentskoj strani. Na klijentskoj strani cilj je osigurati što kvalitetnije korisničko iskustvo. Nakon provedbe testova generirano je izvješće pokrivenosti koda testovima, koji na poslužiteljskoj strani iznosi 84.07%, dok je na klijentskoj 77.4%. Na temelju rezultata može se reći da se osigurala vrlo dobra kvaliteta aplikacije. Uz pomoć funkcionalnih testova utvrđeno je, da je uz određenu kvalitetu ispravnost aplikacije na visokoj razini.

Ključne riječi: automatsko testiranje, razvoj pokretan testiranjem, testiranje, web aplikacija

ABSTRACT

The purpose of this paper is to develop a web application by the process of test driven development and through tests ensure quality and correctness of application. Application validity is ensured through testing of individual parts and functions. During development, testing makes it easier for the developer to find and fix errors. For the implementation of quality tests, it is necessary to select a specific tool, all depending on the used technology. For testing purposes of this application, technologies and tools that allow testing *Javascript* code are used. Tests are divided into server and client side. On the server side, the functionality of communication with the database and functionality of providing services to clients are tested. On the client side, the goal was to provide a better user experience. After the tests were passed, a code coverage report was generated, which is 84.07% for the server side, and 77.4% for the client side. Based on the results, very good application quality is ensured. With the help of functional tests, it was determined that with a certain quality, the application is at a high level.

Keywords: automated testing, test driven development, testing, web application

ŽIVOTOPIS

Michael Turković rođen je 02.03.1994. u Albstadt-Ebingenu, Savezna Republika Njemačka. Osnovnu školu Ivane Brlić-Mažuranić u Orahovici upisuje 2000. godine, koju završava 2008. godine. Iste godine upisuje srednju tehničku školu Marka Marulića u Slatini, smjer elektrotehničar, koju završava 2012. godine. Nakon završene srednje škole upisao je preddiplomski studij elektrotehnike na Tehničkom fakultetu Sveučilišta u Rijeci. Nakon godinu dana prebacuje se na Elektrotehnički fakultet Sveučilišta Josipa Jurja Strossmayera u Osijeku, gdje nastavlja stručni studij na smjeru Informatika. Nakon završenog stručnog studija 2015. godine upisuje razlikovne obveze na Elektrotehničkom fakultetu Osijek, koje završava iste akademske godine 2015./2016. Po završetku razlikovnih obveza upisuje diplomski studij smjer podatkovne i informacijske znanosti na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku.

_____

Potpis

PRILOZI

Prilog 1. Dokument diplomskog rad

Prilog 2. Pdf diplomskog rada

Prilog 3. Programski kod