Ručno i automatizirano testiranje web aplikacije i usporedba alata Cypress i Selenium

Nađ, Andreja

Master's thesis / Diplomski rad

2023

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:200:396970

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-10**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni diplomski studij

RUČNO I AUTOMATIZIRANO TESTIRANJE WEB APLIKACIJE I USPOREDBA ALATA CYPRESS I SELENIUM

Diplomski rad

Andreja Nađ

Osijek, 2023.

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

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 18.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Andreja Nađ
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1232R, 08.10.2021.
OIB studenta:	63306360546
Mentor:	prof. dr. sc. Goran Martinović
Sumentor:	,
Sumentor iz tvrtke:	Valentina Valentić, mag.ing.comp.
Predsjednik Povjerenstva:	doc. dr. sc. Ivan Vidović
Član Povjerenstva 1:	prof. dr. sc. Goran Martinović
Član Povjerenstva 2:	izv. prof. dr. sc. Tomislav Matić
Naslov diplomskog rada:	Ručno i automatizirano testiranje web aplikacije i usporedba alata Cypress i Selenium
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu diplomskog rada treba opisati i analizirati probleme, izazove i mogućnosti ručnog i automatiziranog testiranja programske podrške. Nadalje, potrebno je opisati mogućnosti alata Cypress i Selenium za automatizirano testiranje web aplikacija, predložiti projekt složene web aplikacije za provedbu ručnog i automatiziranog testiranja na korisničkoj i poslužiteljskoj strani, te opisati značenje testiranja u ciklusu razvoja web aplikacije tvrtke. U praktičnom dijelu rada, treba predložiti planove, scenarije i slučajeve testiranja za provedbu ručnog i automatiziranog testiranja na funkcionalnoj i nefunkcionalnoj razini, provesti testiranje, predložiti moguća unaprjeđenja programskog rješenja, te napraviti analizu rezultata testiranja i utjecaja na
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 od strane mentora:	18.09.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 29.09.2023.

Ime i prezime studenta:	Andreja Nađ
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1232R, 08.10.2021.
Turnitin podudaranje [%]:	5

Ovom izjavom izjavljujem da je rad pod nazivom**: Ručno i automatizirano testiranje web aplikacije i usporedba alata Cypress i Selenium**

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 I PRIKAZ STANJA U PODRUČJU *2*

2.1. Testiranje i važnost testiranja **2**

2.2. Testna načela **5**

2.3. Tipovi testiranja **6**

2.3.1. Ručno testiranje 6

2.3.2. Automatizirano testiranje 7

2.3.3. Usporedba ručnog i automatiziranog testiranja 8

2.4. Koraci testiranja **8**

2.5. Strategija testiranja **9**

2.6. Razine testiranja **11**

2.7. Vrste testiranja **12**

2.8. Tehnike testiranja **13**

2.8.1. Crna kutija 13

2.8.2. Bijela kutija 14

2.8.3. Testiranje na temelju iskustva 15

2.8.4. Testiranje od kraja do kraja 15

3. ALATI ZA AUTOMATIZIRANO TESTIRANJE *17*

3.1. Pregled postojećih alata **17**

3.2. Cypress i pridajuće biblioteke **18**

3.3. Selenium i pridajuće biblioteke **19**

3.4. Usporedba testnih alata Cypress i Selenium **21**

4. DEFINIRANJE TESTNE OKOLINE I POSTAVKE TESTIRANJA *23*

4.1. Testna okolina **23**

4.2. Web aplikacija za prodaju auta **23**

4.2.1. Izvješće prodaje po prodajnim savjetnicima 23

4.2.2. Izvješće prodaje po markama 26

4.2.3. Učinkovitost prodaje po prodajnom savjetniku 27

4.3. Značenje testiranja u razvoju web aplikacije **28**

4.4. Testni plan **28**

4.5. Testni podaci za automatizirano testiranje **29**

5. RUČNO I AUTOMATIZIRANO TESTIRANJE WEB APLIKACIJE ***30***

5.1. Ručno testiranje **30**

5.1.1. Prvi scenarij – ručno testiranje 30

5.1.2. Drugi scenarij – ručno testiranje 30

5.1.3. Treći scenarij – ručno testiranje 31

5.1.4. Četvrti scenarij – ručno testiranje 31

5.1.5. Postupci provođenja testova 31

5.2. Automatizirano testiranje web aplikacije u alatu Cypress **32**

5.2.1. Prvi scenarij – automatizirano (Cypress) 32

5.2.2. Drugi scenarij – automatizirano (Cypress) 39

5.2.3. Treći scenarij – automatizirano (Cypress) 40

5.2.4. Četvrti scenarij – automatizirano (Cypress) 45

5.3. Automatizirano testiranje web aplikacije u alatu Selenium **46**

5.3.1. Prvi scenarij – automatizirano (Selenium) 47

5.3.2. Drugi scenarij – automatizirano (Selenium) 50

5.3.3. Treći scenarij – automatizirano (Selenium) 52

5.3.4. Četvri scenarij – automatizirano (Selenium) 57

5.4. Nefunkcionalno testiranje **58**

5.5. Izvješća automatiziranog testiranja **59**

6. USPOREDBA, ANALIZA I PRIMJENA REZULTATA TESTIRANJA ***60***

6.1. Ručni i automatizirani testovi **60**

6.2. Selenium i Cypress automatizirani testovi **62**

6.3. Primjena rezultata testiranja **62**

6.4. Osvrt na testiranje **66**

7. ZAKLJUČAK ***69***

LITERATURA ***70***

SAŽETAK ***73***

ABSTRACT ***74***

ŽIVOTOPIS ***75***

PRILOZI ***76***

1. UVOD

Testiranje ima ključnu ulogu u osiguravanju visoke kvalitete i pouzdanosti aplikacije, neovisno radi li se o web stranici, mobilnoj aplikaciji ili računalnom programu. Testiranje predstavlja postupak koji pomaže odrediti i ispraviti potencijalne probleme prije nego što aplikacija dođe u ruke korisnika. Pod testiranje spada i proces prijavljivanja grešaka i ponovno testiranje nakon otklanjanja istih. Testiranje se prema tipovima dijeli na ručno i automatizirano testiranje. Kod ručnog testiranja provode se testni scenariji i slučajevi bez alata za automatizaciju. Kod automatiziranog testiranja potrebno je pravilno odabrati alat i napraviti testove.

Cilj ovog diplomskog rada je prikazati važnost i proces testiranja tijekom razvoja web aplikacije tvrtke. Zadatak je predstaviti strategije, tehnike, razine i vrste prema kojima se provodi testiranje i prikazati mogućnosti testiranja prilikom obavljanja ručnog i automatiziranog testiranja. Testiranje se provodi prema isplaniranom testnom planu koji sadrži scenarije, tehnike, strategije, vrste i razine koje je nužno koristiti pri testiranju web aplikacije tvrtke. U ovom radu se obrađuju oba tipa testiranja i njihove razlike prilikom provođenja istih scenarija. Automatizirano testiranje se provodi u alatima Cypress i Selenium zbog usporedbe i analiziranja razlika između njih.

Drugo poglavlje se bavi uvodom u testiranje i njegovom važnošću, objašnjavaju se načela, tipovi, koraci, strategije, razine, vrste i tehnike testiranja. Treće poglavlje se bavi pregledom postojećih automatiziranih alata u industriji te se opisuju dva odabrana alata, Cypress i Selenium. Opisuju se korištene biblioteke i glavne funkcionalnosti te se prikazuje usporedba navedenih alata. Četvrto poglavlje opisuje testnu okolinu i sve postavke potrebne za testiranje kao što su testni plan i testni podaci. Opisuje se web aplikacija koja se testira i značenje testiranja unutar web aplikacije. U petom poglavlju se izvodi ručno i automatizirano testiranje web aplikacije. Testiranja su provedena prema scenarijima navedenim u testnom planu te su njihovi rezultati prikazani u izvješćima. U šestom poglavlju se uspoređuju ručni i automatizirani testovi, kao i testovi obavljeni alatima Cypress i Selenium, a uz to se prikazuje primjena rezultata testiranja unutar tvrtke.

2. TESTIRANJE I PRIKAZ STANJA U PODRUČJU

U ovom poglavlju će se opisivati testiranje, njegova važnost, način, strategije, razine, vrste i tehnike testiranja. Unutar poglavlja se spominju riječi kao što su greška i nedostatak, te one zahtijevaju kratko objašnjenje. Greška (*eng. error ili bug*) je ljudska radnja koja je uzrokovala nevaljani ili neželjeni rezultat. Nedostatak, ili *defekt*, je mana u komponenti ili sustavu koja može izazvati neuspjeh u izvođenju zahtijevanih funkcija. Najvažnija mjera kvalitete proizvoda leži u opažanju korisnika. Drugim riječima, ako kupac izrazi nezadovoljstvo i proglasi proizvod neispravnim, to je jasan znak da postoji problem koji zahtijeva rješavanje. Nedostatak predstavlja sva odstupanja od zahtjeva ili neočekivana ponašanja i može uzrokovati kvar (*eng. failure*). Kvar je devijacija od očekivanog ponašanja ili rezultata. Kvar nastaje ukoliko bilo koji nedostatak pri izvođenju sustava uzrokuje neočekivan rad sustava. Svi nedostaci ne rezultiraju kvarom jer je neke nedostatke teško pronaći te ostanu skriveni u kôdu [1].

2.1. Testiranje i važnost testiranja

Testiranje programa je verifikacijski proces za procjenu kvalitete i poboljšanje programa [2]. Dvije metode kojima se odvija procjena kvalitete i poboljšanje programske podrške su statička i dinamička analiza.

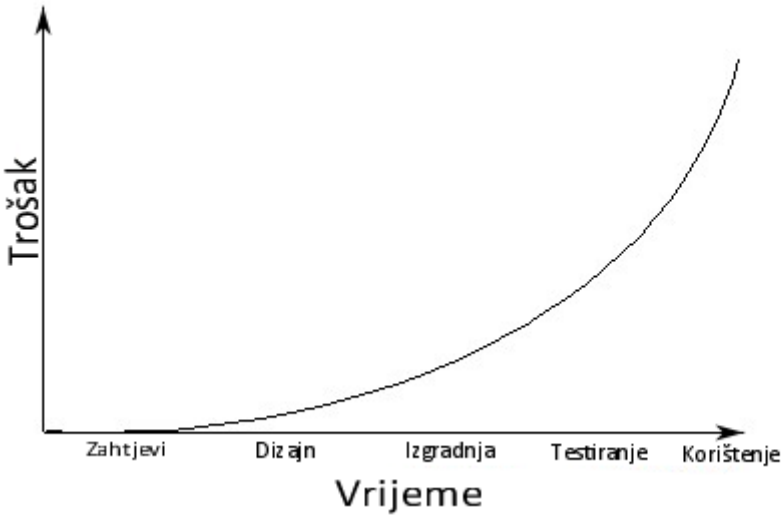
Tijekom statičke analize se ne izvršava programski proizvod već se pregledava izvorni kôd, dizajn, dokumentacija i sl. kako bi se pronašli nedostaci, potencijalni problemi i nesuglasice. Glavni cilj je pronaći greške i nedostatke u što ranijoj razvojnoj fazi kako bi se smanjili troškovi i rizici tijekom kasnijih faza. Neke od tehnika koje se koriste tijekom statičke analize uključuju verifikaciju i validaciju dokumenata kao i dizajna. Verifikacija je proces u kojem se provjerava točnost tj. je li proizvod dobro napravljen, dok je validacija proces u kojem se provjerava ispunjava li proizvod svoju namjenu tj. odgovara li zahtjevima korisnika. U testiranju to znači da se verifikacijom provjeravaju implementacije i njihove odgovarajuće specifikacije koje moraju ispuniti, a validacijom se gleda proizvod tj. program u cijelosti te ga se uspoređuje sa početnim zahtjevima koje je zadao korisnik. Dodatno kroz tehniku pregleda kôda moguće je uočiti probleme pri nepravilnoj sintaksi, logičke greške, loše prakse i nepoštivanje standarda.

Dinamička analiza se provodi tijekom izvršavanja programskog proizvoda. U ovoj fazi ima se programski proizvod, te je moguće provođenje testova i provjera ispravnosti. Testiraju se karakteristike programskog proizvoda kao što su: funkcionalnost, izvođenje, pouzdanost i sigurnost. Glavni cilj je provjeriti radi li program očekivano, ima li grešaka ili nekih neočekivanih

ponašanja na klijentskoj ali i na poslužiteljskoj strani aplikacije. Osim izvođenja testova, bitna je i analiza prikupljenih rezultata kako bi se putem njih dalje unaprjeđivao program. Primjeri dinamičke analize će se dalje obraditi kod vrsta testiranja.

Obje analize imaju komplementarne snage i slabosti. Rezultati statičke analize mogu bit slabiji nego očekivani, ali zajamčeno će se generalizirati za buduća izvršenja Dinamička analiza učinkovita je i precizna jer ne zahtijeva skupe analize, ali zahtijeva testne slučajeve te daje vrlo detaljne rezultate [3]. Testni slučaj je skup preduvjeta, ulaza, radnji i očekivanih rezultata razvijenih na temelju testnih uvjeta i zadanim zahtjevima. Statička i dinamička analiza najčešće se koriste zajedno jer se izvođenjem obje analize dobiva se više informacija nego izvođenjem samo jedne analize. Obje analize mogu se koristiti na istom programu te tako zapravo omogućuju visoku sigurnost u programu.

Važnost testiranja najviše je prikazana u pronalasku nedostataka jer iako se ne može osigurati da su svi nedostaci pronađeni, pronalaskom bilo kojih nedostataka i grešaka povećava se kvaliteta proizvoda i zadovoljstvo kupca tj. korisnika. Izostavljanjem testiranja u procesu razvoja programa riskiraju se dodatni troškovi, gubitak zadovoljstva i ugleda. Prema [1], slika 2.1 prikazuje dijagram troškova grešaka tijekom razvoja. Prema dijagramu vidljivo je da kasno uvođenje testiranja uvodi i veliki trošak, ali i kasni pronalasci nedostataka isto tako dovode do velikih troškova. Što je nedostatak kasnije otkriven, to su troškovi njegovog ispravljanja veći.



Slika 2.1. Dijagram troškova grešaka tijekom razvoja

Prema istraživanjima 85% grešaka i nedostataka programa se stvaraju u najranijim fazama razvoja, čak i prije nego je kôd izvršen [4]. Stoga je važno što prije uvesti testiranje u faze razvoja. Pri određivanju vremena za razvoj aplikacije prvobitno se određuje vrijeme potrebno za samo pisanje

kôda, te se ponekad za testiranje ne ostavi dovoljno vremena. Ako se testiranje ne odvija istodobno s razvojem kôda tj. ako je testiranje zadnja faza razvoja, dolazi do mogućnosti manjka vremena za ispravak svih pronađenih grešaka i problema u aplikaciji. Dodatno, kada programeri znaju da imaju tim koji će odraditi testiranje za njih, ponekad se ne provodi nikakvo testiranje prije nego što dođe do faze testiranja. Jeftinije je uvesti testiranje pri početku razvoja inače će testiranje zahtijevati dodatne resurse i vrijeme za koje se možda nije planiralo u početku, a ako se želi imati kvalitetan proizvod, testiranje je neizbježno. Osim za pronalazak nedostataka, testiranje je važno kako bi se ustanovila pouzdanost i upotrebljivost kôda.

Načela dobrog testiranja su [4]:

- poslovni rizik može se smanjiti pronalaskom nedostataka
- pozitivno i negativno testiranje doprinosi smanjenju rizika
- statičko i dinamičko testiranje pridonosi smanjenju rizika
- automatizirani alati za testiranje mogu značajno doprinijeti smanjenju rizika
- najveći rizici zahtijevaju prvi prioritet testiranja
- najčešće poslovne aktivnosti su drugi prioritet testiranja
- statičke analize obrazaca pojavljivanja kvarova i drugih karakteristika kvara su vrlo učinkoviti način za predviđanje završetka testiranja
- testirati sustav na način na koji će ga kupci koristiti
- pretpostaviti da su nedostaci rezultat procesa, a ne osobnosti
- ispitivanje nedostataka je ulaganje, ali i trošak.

Rizik je nešto što se još nije dogodilo i možda se ni neće dogoditi, to je potencijalan problem [1]. Kada taj potencijalan problem postane stvaran osjeti se negativan utjecaj koji može uzrokovati velike gubitke. Stoga se kod testiranja procjenjuju mogući rizici i njihovi utjecaji kako bi se znalo na što se treba prvo fokusirati te što zahtijeva kakvo testiranje. Važno je znati kako testirati, što ima prednost tj. veće prioritete pri testiranju te kada prestati testirati. Iako testiranje pomaže, potrebno je biti svjestan da i ono ima svoje troškove, ali ti troškovi nisu ništa naspram troškova koji bi nastali kada bi se izbacilo testiranje, napravio loš program i onda ponovno ubacilo testiranje.

Prema [5], testiranje je kompromis između proračuna, vremena i kvalitete. Smatra se da će programske greške gotovo uvijek postojati u bilo kojem programskom modulu umjerene veličine jer je riječ o složenim sustavima, te bilo kakve mane kao nedostaci dizajna ili dokumentacije mogu pogoršati broj grešaka u programu [5]. Tipično više od 50% vremena za razvoj je utrošeno na testiranje. Problem kod testiranja je isti kao i kod razvoja, a to je složenost programa zbog kojeg

je teško testirati svaku liniju kôda na sve moguće načine (iscrpno testiranje). Dva najčešća uzroka nedostataka prema [6] su propusti, promjene i pogreške kod dizajna i zahtjeva.

Prema [1] i [7], testiranje nije jedna aktivnost već proces u kojem se provodi planiranje, priprema i evaluacija programskog proizvoda i povezanih proizvoda kako bi se utvrdilo da zadovoljavaju specificirane zahtjeve, da se demonstrira ispravnost i pronađu nedostaci. Prema tome ne testira se samo kôd već zahtjevi i dizajni te se pregledavaju svi vezani dokumenti i rezultati. Zadnji dio rečenice predstavlja objektive testiranja, a oni su:

- provjera ispunjavaju li se svi zahtjevi
- radi li ispravno (kako je kupac namijenio)
- pronalazak nedostataka.

Testiranje kao izvođenje testova i izvršavanje programa je samo jedan dio koji spada pod testiranje. Iako je objektiv testera pronalazak nedostataka, oni ih ne popravljaju sami već ih prijavljuju te se prijavljeni nedostaci uklanjaju u fazi otklanjanja pogrešaka (*eng. debugging*) od strane programera.

2.2. Testna načela

Prema [1] i [8], sedam načela testiranja su:

- testiranje prikazuje postojanje nedostataka
- iscrpno testiranje je nemoguće
- ranije testiranje smanjuje utrošeno vrijeme i novac
- nedostaci se grupiraju
- paradoks pesticida
- testiranje ovisi o kontekstu
- zabluda o odsutnosti grešaka.

Testiranje prikazuje postojanje nedostataka, a ne njihovu odsutnost. Važno je znati da ne pronalaženjem nedostataka tijekom testiranja nismo sigurni da ih nema, nego ih način testiranja ne pronalazi. Stoga je potrebno promijeniti pristup testiranja. Kao i ranije spomenuto, potpuno ili iscrpno testiranje je nemoguće. Nemoguće je testirati sve, svaki preduvjet, svaki ulaz, svako ponašanje, te je sami takav pothvat gubitak vremena i novca a ne utječe previše na kvalitetu proizvoda. Potrebno je odabrati prioritete testiranja tj. ključne funkcionalnosti koje imaju najveći utjecaj i fokusirati se na njih. Nije bitno testirati svaku liniju kôda već imati što veće pokriće sa testovima kroz planiranje i procjenu. Znajući da se čak 85% nedostataka se dogodi u početnoj fazi

razvoja, bolje ih je odmah ukloniti. Paretov princip kaže da otprilike 80% posljedica dolazi od 20% uzroka. Kada na složenom kôdu primijenimo Paretov princip, dobivamo da 80% nedostataka dolazi od 20% modula ili komponenata. Ideja je da mali broj komponenata ili modula sadrži najviše nedostataka. Ako se pronađe nedostatak na jednom od tih modula ili komponenata, velika je vjerojatnost da postoji još nekoliko nedostataka na istom mjestu. Naziv paradoks pesticida dolazi od primjene pesticida na usjevima kako bi se uklonili insekti. No dugotrajnom upotrebom istog pesticida, insekti se naviknu te postani otporni pa se daljnjom uporabom pesticida više ne uklanja problem. Ideja je da ako se ponavljaju isti testovi, kad-tad oni više neće naći nove nedostatke. Potrebno je aktualno izmijeniti i nadograđivati scenarije kako bi se izbjegao paradoks pesticida. Testiranje ovisno o kontekstu je vrlo jednostavan princip koji je razumljiv po samom imenu. Jedna strategija, jedan test, ne može odgovarati za svaki scenarij. Vrste testiranja koje vršimo će ovisiti o kontekstu. To znači da je testiranje ovisno o onome što se testira, bilo to mobilna ili web aplikacija. Testiranje aplikacija kojima je važna sigurnost će se testirati drugačije nego obična web stranica.

Zabluda o odsutnosti grešaka je ideja da je zabluda očekivati da sam pronalazak i popravljanje velikog broja nedostataka osigurava uspješnost sustava. Jedan od najpoznatijih primjera za ovaj slučaj je Google+ koji je imao jaku konkurenciju (Facebook), i osim što je izgubio povjerenje zbog povrede podataka, imao je nespretno korisničko sučelje koje je otežavalo upravljanje korisnicima. Njegova konkurencija imala je bolje korisničko sučelje te tako zadobila korisnike. Prema tome, iako program možda nema nedostataka i radi, važno je znati i testirati odgovara li taj program namjeni, jesu li ispunjeni zahtjevi i je li program odrađen na kvalitetan način. Pri testiranju važno je paziti za koga je namijenjen proizvod i testirati u te svrhe. Korisničko iskustvo pri korištenju stranice važno je koliko i sama funkcionalnost stranice.

2.3. Tipovi testiranja

Podjela testiranja ovisi o načinu na koji se testira. Prema tome postoje dva tipa testiranja: ručno i automatizirano.

2.3.1. Ručno testiranje

Ručno testiranje predstavlja testiranje u kojem tester ručno izvodi testne slučajeve bez alata za automatizaciju. U ručnom testiranju vrlo je važno iskustvo samog testera. Ručno testiranje nije u potpunosti zamjenjivo s automatiziranim testiranjem, ali ono nije uvijek dosljedno zbog ljudske greške te je time manje pouzdano i zahtjeva puno vremena i napornog rada [9]. Testeri pišu i provode testne slučajeve uz unošenje ulaznih podatka, provjeravaju dobivene i očekivane rezultate

te ako dođe do pogreške, zapisuju ih u izvješća. Većinom tijekom ručnog testiranja, testeri pregledavaju jesu li osnovne značajke pravilno prikazane te provode testove na način koji bi krajnji korisnici koristili program. Ti scenariji su provedeni pazeći na zadane zahtjeve, ali se uzima u obzir ono što bi korisnik koji ne zna ništa o sustavu pokušao napraviti, i time se dodatno testira kvaliteta. Prednost ručnog testiranja je fleksibilnost i mogućnost prilagodbe promjenama kao i nekim nepredvidljivim situacijama tijekom testiranja. Još neke prednosti prema [10] su: jeftinije jer ne koristi skupe alate tako da je kratkoročni trošak nizak, mogućnost ljudske procjene tj. korištenje programa na način koji bi koristio i korisnik, brzi prikaz rezultata. Neki od nedostataka prema [10] su: neki testovi su teški za ručno izvođenje te je time moguće uvesti greške, testiranje može biti dosadno i ponavljajuće što znači da testeri gube angažiranost te se ponovno mogu događati greške, ako dođe do promjena potrebno je ponoviti sve testove ispočetka što zahtijeva više vremena. Glavni nedostatak je da može biti vremenski i resursno zahtjevno, pogotovo za velike i složene programske sustave. Ne samo što je vremenski zahtjevno, već je moguće da se uvede greške jer su ljudi sami skloni greškama [11].

2.3.2. Automatizirano testiranje

Većina ručnih testova se mogu automatizirati. Prema [12], automatizirano testiranje smanjuje napore ručnog testiranja tako da se usredotočuje na automatiziranje testova koje ručno testiranje teško može postići. Automatizirano testiranje je pouzdanije od ručnog testiranja. To je proces u kojem se testiranje izvodi pomoću alata za testiranje, a detaljnije će biti obrađeno u kasnijim poglavljima. Automatizacija je dobar način da se smanji potrebno vrijeme i trošak za testiranje programa. Osim smanjenja troška i vremena, još jedna prednost prema [4] je to da postoji mogućnost ponovnog korištenja testnih slučajeva i obrazaca otkrivanja nedostataka nakon nadogradnje sustava ili čak kod novih, drugih sustava koje imaju slične funkcionalnosti. Ovaj način testiranja pogodan je za veliki broj unosa jer ima mogućnost odvajanja unosnih podataka u odvojenu datoteku te pošto se ona lako može izmijeniti, testovi su prilagodljivi i brzo se može pokrenuti testiranje. Prema [10], automatizacija predstavlja mogućnost efektivnog i brzog izvođenja testova uz nizak dugoročni trošak. Nedostaci su: skupi alati, iako je proces brži od ručnog testiranja svejedno zahtjeva puno vremena za pravljenje, ispravljanje i izvođenje testova, alati imaju ograničenja pogotovo u vizualnom testiranju. Automatizirano testiranje pruža sigurnost od neželjenih nedostataka nastalih pri ispravljanju poznatog nedostatka. Kod automatiziranog testiranja lako je ponoviti sve testove i dobiti brzu povratnu informaciju kako bi se utvrdili ispravci i potvrdila valjanost prethodno prošlih testova, tj. da popravljanje jednog nedostatka nije

uzrokovalo negativno na postojeće funkcije. Ovakva provjera i testiranje se naziva regresijsko testiranje.

2.3.3. Usporedba ručnog i automatiziranog testiranja

Usporedbom prednosti i nedostataka oba tipa testiranja može se reći da je ručno testiranje bolje u vizualnom testiranju, malim programima i otkrivanju ljudskih grešaka, dok je automatizirano testiranje bolje za velike projekte i za mali broj testera, jer programeri mogu brzo dobiti rezultate ako je sve pravilno povezano. Iako automatizirani testovi mogu preskočiti neke greške jer program ne može „razmišljati" kao čovjek, oni su ipak brži i pouzdaniji. Dodatno, pošto se izvode u alatima, testovi su programabilni te tako mogu dohvaćati skrivene informacije. Glavna razlika je u načinu testiranja. U automatiziranom testiranju potrebno je prolaziti kroz elemente na aplikaciji i dohvaćati ih putem njihovih ID-eva, klasa ili sličnoga, dok kod ručnog testiranja je potrebno samo da tester razumije gdje je što i kada što treba odraditi. U velikoj većini, ID-evi i klase su navedeni u tehničkim specifikacijama, no ako nisu, onda se putem korištenja alata inspekcija stranice (za Internet preglednike) može pronaći ključna riječ ili neki drugi odgovarajući način za pronalazak elemenata.

2.4. Koraci testiranja

Prije objašnjavanja koraka testiranja potrebno je definirati neke ključne definicije osim već prethodno spomenutog testnog slučaja. Testna osnova definirana je kao sva potrebna dokumentacija za pisanje testnih slučajeva i za analizu testova. Testni postupci se definiraju kao skup detaljnih uputa za postavljanje i izvođenje korak po korak jednog ili više zadanih testnih slučajeva. Testni paketi (*eng. test suite*) su definirani kao skup testnih skripti koje će se izvršiti u određenom testnom izvođenju, odvojene po komponentama ili sustavu koji se testira. Završni uvjeti se često koriste kao početni uvjeti sljedećeg testa. Testna skripta skup je uputa za izvođenje testa. Testni plan je dokument na kojem je zacrtan trošak i raspored aktivnosti te na koji način će se aktivnosti obavljati i tko će ih izvoditi. Zapisana je i testna okolina, korištene tehnike i strategije kao i njihovi razlozi. Dodatno ima i kontingencijske planove za rizike. Tijekom razvijanja testnog plana gledaju se rizici i ovisnosti te se prema njima on stvara.

Koraci testiranja su:

- Plan testiranja ili testni plan je korak koji se događa tijekom čitavog procesa testiranja, izvode ga sami testeri ili testni menadžeri.

- Praćenje i kontrola testova konstantna je usporedba napretka testiranja naspram testnog plana. Ukoliko testiranje kasni potrebno je poduzeti odgovarajuće mjere kako bi se postigli ciljevi zadani u testnom planu. Izrađuje se izvješće o napretku testova.
- Analiza testova je korak u kojem se definiraju i određuju prioriteti testova. Ovaj korak odgovara na pitanje „Što testirati?" te analizira testnu osnovu.
- Test dizajn odgovara na pitanje „Kako testirati?". U ovom koraku se dizajniraju i prioritiziraju testni slučajevi, testni podaci te se dizajnira ispitno okruženje.
- Implementiranje testova odgovara na pitanje „Imamo li sve potrebno za izvršavanje testova?". U ovom koraku se razvijaju i prioritiziraju testni postupci. U koraku se prave testni paketi te se organiziraju prema rasporedu izvršavanja testova. Pravi se i testna okolina.
- Izvršavanje testova predstavlja izvršavanje testnih paketa, usporedbe stvarnih i očekivanih rezultata te prijavljivanje nedostataka od strane testera prema programeru. Događa se potvrdno i regresijsko testiranje.
- Završavanje testova je korak koji se izvodi na prekretnicama projekta. Sakupljaju se podaci testnih aktivnosti, provjeravaju se izvješća nedostataka, pravi se sažetak izvješća o testiranju te se predaje *testware*. *Testware* predstavlja artefakte proizvedene tijekom procesa testiranja potrebne za planiranje, dizajn i izvođenje testova kao što su dokumentacija, skripte, unosi, očekivani rezultati, procedure postavljanja i čišćenja, datoteke, baze podataka, okruženje i bilo koji dodatni program ili uslužni programi koji se koriste pri testiranju. Prema [13] i [14], *testware* je potreban za održavanje te ako izvorni proizvođači nisu i održavatelji onda se on predaje dalje.

Pri izradi testova važno je razmišljati o potencijalnim izvorima nedostataka i kvarova, a oni se mogu nalaziti u: specifikacijama, dizajnu i implementaciji sustava i programa; načinu korištenja sustava; okolišnim uvjetima; potencijalnim posljedicama prethodnih grešaka, namjernim oštećenjima, nedostacima i kvarovima.

2.5. Strategija testiranja

Prema [13] i [15] , strategije testiranja dokumenti su visoke razine koji sadrže opis razine testiranja koje treba izvesti i testiranja unutar razina za organizaciju ili program, a mogu se primjenjivati za jedan ili više projekata. Strategije su dugoročni planovi djelovanja za cjelokupni testni pristup. Strategija testiranja predstavlja okvir za regulaciju procesa testiranja. Strategije testiranja dijele se prema [1], [16], [17] i [18] na:

- analitičke strategije
- strategije na temelju modela
- metodičke strategije
- strategije sukladne procesu ili standardu
- usmjerene (savjetodavne) strategije
- strategije nesklone regresiji
- reaktivne ili dinamične strategije.

Analitička strategija je strategija u kojoj se prvobitno analiziraju faktori projekta kao što su zahtjevi. Analiziraju se i pronalaze rizici i specifični zahtjevi koji se moraju ostvariti. Primjer ovakve strategije je testiranje na temelju rizika gdje su testovi dizajnirani i prioritizirani prema razini rizika. Strategija na temelju modela se temelji, kao što i ime kaže, na modelu kojeg stvaraju testeri. Modeli predstavljaju neku situaciju uzimajući u obzir ulaze, izlaze, procese i moguća ponašanja. Modeli imitiraju stanje i uvjete u kojem projekt radi i omogućuju virtualni pristup testiranju. Primjerice kod testiranja izvođenja projekta, model bi predstavljao dolazni i odlazni promet na mreži. Drugim riječima, testovi su dizajnirani prema modelima potrebnih aspekta projekta što mogu predstavljati funkcionalnost, unutarnju strukturu i sl. Metodička strategija predstavlja strategiju gdje testeri slijede unaprijed definirane standarde kvalitete poput ISO25000 standarda, popise za provjeru ili skupove uvjeta testiranja. Predstavlja sistematsko korištenje unaprijed definiranih skupova testova ili uvjeta testova kao što su skup uobičajenih ili vjerojatnih vrsta grešaka ili lista važnih karakteristika kvalitete. Strategije sukladne procesu ili standardu sadržavaju analizu, dizajn i implementaciju testova prema vanjskim zakonima i standardima tj. testeri slijede procese ili smjernice koje su postavljene od strane industrije Usmjerene (savjetodavne) strategije vođene su savjetima, usmjeravanjem ili instrukcijama stakeholdera – stručnjaka unutar poslovne domene ili tehnoloških stručnjaka koju mogu biti van testnog tima. Strategije nesklone regresiji usmjerene su željom da se izbjegne regresija postojećih sposobnosti tj. usmjerene su na smanjenje rizika regresije za funkcionalne ili nefunkcionalne dijelove proizvoda.

Sve prethodne strategije su proaktivne tj. potrebno ih je isplanirati prije testiranja, no reaktivne se strategije mogu kretati kada se program objavi. Testovi su dizajnirani i implementirani tek nakon što je sustav gotov. Time je testiranje zapravo utemeljeno na nedostacima pronađenima u pravom, gotovom sustavu. Ne postoji skup propisa, preduvjeta ili analiza za ovu vrstu testiranja. Glavni primjer ovakve strategije je istraživačko testiranje. Iz strategije dobivamo pristup testu tj.

implementaciju strategije na projekt. Iz pristupa možemo definirati razine, vrste i tehnike testiranja.

Odabir strategije ima puno faktora koji se trebaju uzeti u obzir, ali neki od najvažnijih prema [1] su: rizici, vještine testera, objektivi testiranja, regulacije, proizvod i poslovanje. Testiranje mora zadovoljiti potrebe dionika, a ponekad i regulatora. Odabir strategije će biti lagan ako se pogledaju koji od ovih faktora predstavljaju najveću ulogu za ostvarivanje cilja testiranja. Npr. kod već izrađene aplikacije koja se nadograđuje, najveći rizik predstavlja regresija pa prema tome strategije nesklone regresiji su dobar odabir. No, kod novih aplikacija, analitička strategija temeljena na riziku bi bila bolja opcija. Odabir pravilne strategije će ovisiti i o troškovima testiranja, troškovima mogućih nedostataka te minimiziranju rizika.

2.6. Razine testiranja

Razine testiranja prema [1] su:

- Testiranje komponente (*eng. component testing or unit testing*) traži nedostatke i provjerava ispravnost funkcionalnosti u dijelovima programa koje je moguće odvojeno testirati. Testiraju se smislene cjeline bez utjecaja drugih cjelina na njih (bez integracije drugih komponenata). Ovakvo testiranje sadrži i testiranje ponašanja resursa, izvođenja ili robusnosti.
- Testiranje integracije (*eng. integration testing*) testira sučelja između komponenti, interakcije s drugim sustavima kao što su datotečni i operacijski sustav te sučelja između sustava. Prema tome imamo podjelu na testiranje integracije komponenti i testiranje integracije sustava.
- Testiranje sustava (*eng. system testing*) se obazire na ponašanje cijelog sustava ili proizvoda. Testiranje sustava inače je zadnje testiranje koje se provodi te je potrebno osvrnuti se na funkcionalne ali i nefunkcionalne zahtjeve sustava. Testiranje se provodi na kontroliranom testnom okruženju koje mora što više odgovarati konačnom cilju ili proizvodnom okruženju. Time se povećava mogućnost pronalaska nedostataka na konačnom proizvodnom okruženju pa se time i rizik istoga smanjuje.
- Testiranje prihvatljivosti (*eng. acceptance testing*) dolazi nakon provedbe testiranja sustava. Ovo je razina testiranja na kojoj se dobiva samouvjerenost u valjanost i upotrebljivost proizvoda. Testira se u okruženju koje je što više približno stvarnom okruženju, kao da je aplikacija puštena na tržište. Testiranje prihvatljivosti se većinom

fokusira na validaciju gdje se provjerava je li sustav zadovoljio svoju svrhu, je li spremno za puštanje na tržište, i postoje li neki poslovni rizici.

Testiranje komponente testira pojedinu komponentu dok integracijsko testiranje testira integrirane komponente. Sustavno testiranje testira cijeli sustav, a testiranje prihvatljivosti testira konačni sustav. Ovakav način testiranja omogućuje testerima testiranje u ranijim fazama razvoja programa te smanjuje mogućnosti pronalaska nedostataka u završnoj fazi razvoja.

2.7. Vrste testiranja

Vrste testiranja su skupine aktivnosti testiranja namijenjene testiranju specifičnih karakteristika sustava programa ili dijela sustava prema određenim objektivima testiranja. Četiri glavne podjele na koje se može podijeliti testiranje su:

- funkcionalno testiranje
- nefunkcionalno testiranje
- testiranje temeljeno na strukturi
- testiranje vezano uz promjene.

Funkcionalno testiranje predstavlja testiranje u kojem se provjeravaju funkcionalnosti koje mora sustav provoditi. Odgovara na pitanje „Što sustav mora raditi?". Testira se ponašanje sustava kao što su održivost, interoperabilnost, sigurnost, točnost i usklađenost. Tehnike testiranja za funkcionalno testiranje su najčešće temeljene na zahtjevima. Jedno od važnijih i početnih testiranja koje spada pod funkcionalno testiranje je testiranje dima (*eng. smoke testing*) koje se koristi za određivanje je li nova inačica aplikacije za testiranje spremna za sljedeću fazu testiranja tj. je li aplikacija uopće spremna za testiranje. Ukoliko ovo testiranje podbaci, inačica aplikacije se vraća nazad na razvijanje.

Nefunkcionalno testiranje odgovara na pitanje „Koliko dobro se sustav ponaša?". Procjenjuju se karakteristike sustava i programa kao što su: upotrebljivost, učinkovitost, izvođenje i sigurnost. Neka od važnijih testiranja koja spadaju pod ovu skupinu su testiranje izvođenja, opterećenja, stresa i sigurnosti. Testiranje izvođenja je testiranje izvođenja programa pod svim povoljnim i nepovoljnim uvjetima što uključuje vremenske parametre (vrijeme učitavanja, izvođenja, izvršenja itd.), srednje vrijeme između kvarova, ukupna pouzdanost programa stopa uspješnosti i učestalost neuspjeha [19]. Sigurnosno testiranje je vrsta testiranja programa namijenjena pronalasku svih mogućih rupa i slabosti sustava kao i rizika, prijetnja i ranjivosti. Ideja je da se pronađu sve slabosti kako bi se mogle popraviti i spriječiti zlonamjerni napadi uljeza koji bi inače rezultirali gubitcima

prihoda, informacija te ugleda. Najvažnije za naglasiti je da se ovim testiranjem obuhvaćaju svi aspekti koji nisu obuhvaćeni unutar funkcijskog testiranja kako bi se ostvario pouzdan proizvod. Pouzdan proizvod je onaj koji ispunjava sva očekivanja potrošača i efikasan je u bilo kojim okolnostima. Cilj je ispunjavanje klijentskih očekivanja.

Testiranje temeljeno na strukturi izvodi testove na temelju unutarnje strukture ili implementacije sustava. Pod unutarnju strukturu se smatra kôd, arhitektura, tijek rada i/ili tokovi podataka unutar sustava. Prema [1], ovo testiranje se koristi kao mjera za temeljitost testiranja kroz pokrivanje skupa strukturnih elemenata ili stavki pokrivenosti.

Testiranje vezano uz promjene provjerava tj. potvrđuje da su promjene ispravile nedostatke ili točno implementirale funkcionalnosti. Neke od vrste testova koji spadaju pod ovo su: potvrdno i regresijsko testiranje. Potvrdno testiranje je ono koje potvrđuje ispravak nedostataka, dok regresijsko testiranje otkriva neželjene nuspojave nastale utjecajem promjene jednog kôda na drugi. Regresijsko testiranje najbolje je za automatizaciju jer se može izvršiti više puta, ali je sporija izvedba.

2.8. Tehnike testiranja

Tehnike testiranja su metode koje se primjenjuju na sustav ili komponentu kako bi procijenili zadovoljava li zadane zahtjeve. Iako ima više podjela tehnika testiranja, najčešće koje se spominju prema [1], [11], [19], [20], [21], [22] i [23] su:

- crna kutija (*eng. Black box*)
- bijela kutija (*eng. White box*)
- na temelju iskustva.

2.8.1. Crna kutija

Testne osnove su specifikacije programskih zahtjeva, dokumentacija poslovnih, funkcionalnih i kvalitetnih zahtjeva. Crna kutija predstavlja testiranje ponašanja tj. funkcionalno testiranje. Tehnika je dobila naziv po tome da se gleda samo ono što se nalazi van kutije tj. gleda se samo funkcionalnosti a ne kako one rade. Crna kutija kao da prekriva cijeli sustav te se samo gleda ulaz i izlaz. Drugim riječima, testiranje tehnikom crne kutije je testiranje izlaza prema zahtjevima bez obzira na unutarnju strukturu ili kôd programa. Najčešće vrste tehnika testiranja su:

- podjela ekvivalencije
- analiza granične vrijednosti
- testiranje tablice odlučivanja

- testiranje prijelaza stanja
- slučaj upotrebe (*eng. use case*).

Podjela ekvivalencije je tehnika testiranja gdje se ulazni podaci podjele na particije pravilnih i nepravilnih vrijednosti. Sve particije dijele iste rezultate, što znači da ako je jedna particija točna onda njoj jednaka particija mora imati isti rezultat. Vrijednosti koje se uzimaju za testiranje moraju se uzeti iz svake particije. Analiza granične vrijednosti je tehnika u kojoj se testiraju koje granične vrijednosti trebaju biti prihvaćene a koje ne. Koristi se za testiranje zahtjeva koji imaju raspone brojeva te kada neke granice nisu jasne. Testiranje tablice odlučivanja je tehnika koja se koristi za testiranje ponašanja sustava za različite ulazne kombinacije. Svaki stupac odgovara pravilu odlučivanja koje definira jedinstvene kombinacije uvjeta koje rezultiraju u izvođenju akcije vezane za to pravilo. Testiranje prijelaza stanja je tehnika koja prikazuje moguće promjene stanja programa te kako sustav ulazi, izlazi i prelazi između stanja. Prijelazi su pokrenuti događajima, a promjene stanja mogu uzrokovati program da odradi neku akciju. Ovo testiranje pomaže u analizi ponašanja programa za različite ulazne uvjete. Slučaj upotrebe je slikovni prikaz (dijagram i tablica) načina na koji korisnik komunicira sa sustavom. Testovi su dizajnirani da provedu sva definirana ponašanja. Dobar je način za vizualizaciju arhitekture sustava, procjenu mogućih rizika i ovisnosti o sustavu, utvrđivanju zahtjeva te prikazu raznih načina komunikacije korisnika sa sustavom.

2.8.2. Bijela kutija

Testna osnova je kôd, arhitektura programa i detaljan dizajn programa. Bijela kutija predstavlja testiranje strukture, arhitekture tj. nefunkcionalno testiranje. Suprotno od crne kutije, bijela kutija ili bolje rečeno prozirna kutija je metoda gdje se gleda program u cijelosti. Testovi se prave gledajući unutarnju strukturu programa.

Najčešće vrste tehnika testiranja:

- pokrivenost izjava (*eng. Statement coverage*)
- pokrivenost odluka.

Pokrivenost izjava predstavlja proračun koliko je linija kôda pokriveno unutar testova. Poželjno je ostvariti pokrivenost od 100%, kao i kod pokrivenosti odluka. Pokrivenost odluka jača je mjera od prekrivenosti izjava jer provjerava sva moguća grananja jednom iz svake točke odluke.

2.8.3. Testiranje na temelju iskustva

Testna osnova ove metode je znanje i iskustvo testera ili onih u okolini testera. Testni slučajevi su izvedeni iz vještina, znanja i intuicije testera. Iako je pokriće testova teško izračunati ili odrediti, ova metoda testiranja odgovara kod pronalaska nedostataka koje sistematske metode možda ne bi pronašle.

Neki od pristupa testiranju koji su na temelju iskustva:

- pogađanje grešaka
- istraživačko testiranje
- testiranje na temelju kontrolne liste.

Pogađanje greške je nagađanje nedostataka i pogrešaka temeljeno na prethodnim iskustvima testera. Iskustva mogu biti kako je program radio u prethodnim inačicama, najčešće greške programera i kvarove koji su se dogodili u drugim aplikacijama. Neki testeri naprave listu mogućih grešaka, nedostataka i kvarova pa prema tome dizajniraju i provode testove kako bi našli uzrok. Istraživačko testiranje je nasumično ili nestrukturirano testiranje putem kojeg se mogu otkriti greške koje se možda ne bi vidjele tijekom strukturirane faze testiranja. U istraživačkom testiranju tester koristi svoje iskustvo i znanje kako bi upravljao sustavom i izvodio testove. Ovo je neformalni način testiranja gdje su testovi dizajnirani, zabilježeni i procijenjeni tijekom izvođenja testova. Najkorisnije je kada postoje mane u specifikacijama (premale ili neodgovarajuće) ili kada ima jako malo vremena. Testiranje na temelju kontrolne liste je testiranje u kojem testeri dizajniraju, implementiraju i izvršavaju testove kako bi pokrili sve uvjete testova u kontrolnoj listi. Korisno je kada se nemaju detaljni testni slučajevi pa prateći kontrolnu listu nastaju smjernice i neki stupanj dosljednosti.

2.8.4. Testiranje od kraja do kraja

Prema [24], testiranje od kraja do kraja (*eng. End-to-End testing, E2E*) je tehnika testiranja koja se sve više i više koristi u testiranju jer je moderni program postao zamršen s desecima sustava koji istovremeno međusobno komuniciraju. E2E testiranje provjerava funkcionalnost i izvedbu cijelog sustava, od početka do kraja. Prave se korisnički scenariji ili priče iz kojih se stvaraju testovi. Prema [25], scenariji mogu biti jednostavni ili kompliciran, a prave se prema scenarijima iz stvarnog svijeta što omogućuje testerima uvid u funkcioniranje aplikacije iz perspektive krajnjeg korisnika te time i razumijevanje o kvaliteti programa prije izdavanja. E2E testiranje uključuje testiranje cijelog sustava iz perspektive korisnika što pomaže osigurati da komponente rade u skladu s očekivanjima tj. zahtjevima. Obično se E2E testiranje provodi nakon integracijskog

testiranja i zadnja je faza prije puštanja programa na tržište. Svrha ovakvog testiranja je provjera valjanosti sustava i njegovih komponenti za integraciju i integritet podataka. Prema [26], cilj je provjera i potvrda funkcionalnosti, upotrebljivosti, pouzdanosti i sigurnosti programa kao i identifikacija problema integracije koji se mogu pojaviti između različitih sustava i komponenti. Tijekom ovog testiranja moguće je identificirati nedostatke sustava koje bi osjetili krajnji korisnici tijekom interakcije sa sustavom.

3. ALATI ZA AUTOMATIZIRANO TESTIRANJE

U ovom poglavlju se kratko opisuju poznati popularni alati koji se koriste za automatizaciju testova. Izdvajaju se dva alata, Cypress i Selenium i objašnjavaju se njihove karakteristike, biblioteke i funkcionalnosti kao i neke glavne metode. Na kraju se uspoređuju oba alata te se prikazuju razlike.

3.1. Pregled postojećih alata

Iako postoji veliki broj alata koji se mogu koristiti u svrhe automatizacije testova, najčešće spominjani prema [27], [28] i [29] su: Katalon, Slenium, Appium i TestComplete, te jedan od novijih popularnih alata, Cypress. Postoje još neki alati koji se koriste kao što su Ranorex, Watir i sl. Katalon se sadrži od Katalon Studio i Recorder koji podržavaju web, mobilno, računalno i API automatizirano testiranje. Podržava Apache Groovy i Java programske jezike te ima opcije koje su besplatne i koje se plaćaju. Ne zahtijeva pisanje testnog kôda već ima opciju snimanja događaja i spremanja istih kao testove. Podržava više pretraživača (Chrome, Firefox, Safari, Edge i Internet Explorer). Selenium nije samo jedan alat već paket koji se sadrži od: Selenium IDE, Selenium RC, Selenium Webdrive i Selenium Grid. Selenium je prijenosni automatizirani testni paket otvorenog kôda namijenjen za testiranje web aplikacija. Podržava razne preglednike i aplikacije. Podržava Java, C#, Python, JavaScript, Ruby, PHP i Perl programske jezike te je besplatan. Appium, kao i Selenium, je alat otvorenog kôda, ali za razliku od Seleniuma, Appium se fokusira na testiranje mobilnih aplikacija. Koristi JSON žičani protokol te omogućuje pisanje testova za korisničko sučelje za izvorne, web-bazirane i hibridne mobilne aplikacije na Androidu i iOS-u. Podržava testiranje na simulatorima i emulatorima. TestComplete je alat koji je dostupan samo za Windows operacijski sustav, ali po novome ima i opcije za web, Android i iOS aplikacije. Testovi se mogu snimati, skriptirati ili ručno unositi s ključnim riječima. Snimljeni testovi se mogu izmijeniti u slučaju novog testa ili poboljšanja postojećeg. TestComplete je komercijalan alat. Cypress je alat otvorenog kôda za testiranje korisničkog sučelja web aplikacija koji se temelji na JavaScriptu i napravljen je za moderni web i potpuno je besplatan. Ima i opcija Cypress Cloud koja ima besplatne i komercijalne opcije. Cypress Cloud koristi se za snimanje testova u svrhe automatizirane kontinuirane integracije (omogućuje promjenu kôda od više suradnika). U svrhe izrade ovog diplomskog rada uzet će se alati Selenium i Cypress za automatizirano testiranje. Oba alata podržavaju web testiranje te dok je Selenium jedan od najpopularnijih i bitnih alata unutar zajednice testera, Cypress polako dolazi na snagu sa svojim novim idejama.

3.2. Cypress i pridajuće biblioteke

Cypress se sastoji od velikog broja biblioteka zapakiranih zajedno zbog čega je instalacija lagana. Testovi se vode unutar preglednika što omogućuje testeru direktnu komunikaciju s aplikacijom koja se testira. Sve promjene koje su spremljene se automatski prikažu u Cypress aplikaciji što olakšava pisanje i izmjenu testova. Cypress ima opciju snimanja izvedbe testova za lakši pregled rezultata. Cypress dolazi s ugrađenim metodama kao što su:

- *visit* – posjećuje Internet stranicu preko predanog URL-a s mogućim argumentom *options*
- *wait* – čeka uneseni broj milisekundi ili čeka odgovor resursa
- *within* – dohvaća elemente unutar zadanog elementa
- *url* – dohvaća trenutni URL stranice s mogućim argumentom *options*
- *should* – uspoređuje danu vrijednost, metodu, povratnu funkciju ili lančanu metodu s vrijednosti koja je definirana
- *session* – sprema i vraća vrijednosti kolačića (eng. cookies), lokalne pohrane (eng. localStorage) i pohrane sesije (eng. sessionStorage). Argumenti su naziv sesije i funkcija koja se izvodi. Prilikom poziva provjerava se postoji li sesija s predanim imenom te ako ne postoji, onda se stvara nova te se tek onda izvodi predana funkcija i spremaju se vrijednosti. Ako sesija već postoji, funkcija se ne izvodi već se odmah spremaju vrijednosti.
- *request* – podnosi zahtjev koristeći određenu metodu (zadana je metoda GET), rezultat je objekt s podacima statusa, tijela odgovora, zaglavlja i trajanja
- *env* – dohvaća i modificira varijable okoline
- *contains* – dohvaća element objektnog modela dokumenta (eng. Document Object Model, DOM) koji sadrži predani sadržaj
- *get* – dohvaća DOM element
- *click* – simulira akciju pritiska na DOM element.

Većina metoda Cypressa imaju dodatni argument *options* koji se koristi za definiranje posebnog ponašanja metode na kojem je argument. Cypress ima i mogućnost pisanja novih metodi na dva načina, a jedan od njih je zapisivanje metoda unutar Cypress-ove datoteke *commands* uz ključnu riječ *add*, dok je drugi način zapisivanje u drugoj datoteci uz ključnu riječ *export*. Slika 3.1 prikazuje ta dva različita načina pisanja novih funkcija.

```
Cypress.Commands.add("waitSpinner", () => {
    cy.get('clr-progress-spinner').should('not.be.visible');
});
```

```
export function clickOnButton(path) {
    cy.get(path).click();
    cy.waitSpinner();
}
```

Slika 3.1. Prikaz deklariranja novih funkcija u Cypressu

Osim metoda, Cypress ima ugrađenu biblioteku Mocha.js što omogućuje pozive kao što su: *describe, context, it, before, beforeEach, after, afterEach .only i .skip.* Funkcije *describe* i *context* koriste se za odvajanje testnih paketa i testnih slučajeva jednih od drugih s definicijom ili nazivom koji se može napisati. Koriste se za grupiranje i ugnježđivanje testova kao i za lakše čitanje i organiziranje. Funkcija *it* koristi se za pojedinačne testne slučajeve. Funkcije *.only* i *.skip* koriste se iza funkcija *it* za testove. Ove funkcije pomažu pri testiranju kako bi se određeni testovi preskočili uz *skip* ili kako bi se izveo samo jedan test uz naredbu *only*. Funkcije *before* i *beforeEach* koriste se za postavljanje preduvjeta ili funkcija koje se trebaju izvesti unutar svakog testa. Kako se ne bi više puta pisale iste linije kôda, koristi se funkcija *beforeEach* koja se pokreće prije svakog testa te funkcija *before* koja se pokreće jednom pri pokretanju testnog bloka. Funkcije *after* i *afterEach* su sinonimne prethodnima, samo se pokreću poslije testova.

Cypress zadani događaji kao što su *cy.click* su simulirani. To znači da se pokreću iz JavaScripta. Takvi događaji su nepouzdani te se mogu ponašati malo drugačije od izvornih događaja. Biblioteka *CypressRealEvents* popravlja taj problem. Isto tako se koristi kako bi se omogućile funkcije poput prelaska mišem preko elementa *(eng. hover).* Biblioteka koja se također koristi je biblioteka *moments.js* koja pomaže pri obradi datuma i vremena u JavaScriptu. Omogućuje lakše raščlanjivanje, potvrđivanje, manipuliranje i prikazivanje datuma i vremena. Cypress podržava Mochu te time i njene funkcionalnosti. Mocha reporteri, kao što je Junit Reporter, omogućuju zapis rezultata automatiziranih testova u XML datoteku s vremenom izvođenja i rezultatom prolaska ili pada testa.

3.3. Selenium i pridajuće biblioteke

Selenium je popularan alat za automatizaciju testova koji automatizira web-preglednike. Koristi se više od desetljeća i dokazao je svoju pouzdanost. Selenium Webdriver prevodi testne naredbe u URL koji se primaju na strani preglednika putem HTTP zahtjeva. Selenium zahtijeva preuzimanje upravljačkih programa specifičnih za preglednik koji se želi testirati i postavljanje testnog okruženja. Selenium instancira WebDriver koji provodi testove na aplikaciji.

TestNG je okvir za automatizaciju otvorenog kôda za Javu. TestNG pruža napredne značajke kao što su anotacije, testiranje temeljeno na podacima, sekvencirano testiranje i paralelno testiranje u

svrhu organizacije i učinkovitijih i djelotvornijih izvršavanja testova. Anotacije su ključne riječi koje se mogu zadati testovima kao što su *@BeforeTest* ili *@BeforeSuite*. Označene metode će se pokrenuti prije izvođenja svih testova u paketu u slučaju *@BeforeSuite*, dok metoda označena sa *@BeforeTest* će se pokrenuti prije pokretanja bilo koje testne metode koja pripada klasama unutar testne oznake. Postoje korisničke anotacije kao što su: *@AfterSuit, @AfterTest, @AfterMethod, @BeforeMethod* i dr. Testiranje temeljeno na podacima omogućuje pokretanje istog testnog slučaja s više skupova testnih podataka. Sadrži bolje značajke izvješćivanja i zapisivanja od drugih okvira. Korištenjem TestNG okvira u Seleniumu se može poboljšati učinkovitost i djelotvornost automatizacije testiranja. TestNG ima unaprijed definirane slušatelje kao dio svoje biblioteke te se oni prema zadanim postavkama dodaju svakom izvođenju testa i generiraju XML izvješća.

Neke od važnih metoda korištenih u radu za Selenium su:

- *get* – otvara dani URL u zadanom pretraživaču
- *findElement* – pronalazi i vraća prvi pronađeni element prema zadanoj ključnoj riječi koje mogu biti: *By.tagName* (HTML oznake), *By.id* (identifikacijske oznake elementa), *By.xpath* (XML put do elementa) i *By.className* (klasa elementa)
- *findElements* - pronalazi i vraća listu pronađenih elemenata prema zadanoj ključnoj riječi
- *getText* – dohvaća unutarnji tekst web elementa (tekst unutar HTML oznaka)
- *getAttribute* – dohvaća vrijednosti navedenog atributa elementa. Npr. ako element ima atribut „shape“ naredba *element.getAttribute(„shape“)* vraća vrijednost pod atributom „shape“
- *contains* – provjerava sadrži li niz predani niz znakova. Npr. sadrži li tekst neke riječi ili ima li atribut određenu ključnu riječ
- *click* – simulira pritisak na element
- *isDisplayed* – provjerava je li element prisutan i prikazan
- *isEnabled* – provjerava je li element omogućen
- *getWait().until(ExpectedConditions. ...)* – izvodi čekanje dok se uvjet pod ExpectedConditions ne ispuni

  Npr. *getWait().until(ExpectedConditions.visibilityOfElementLocated(By.id(elementID))* predstavlja čekanje dok se element ne vidi zbog uvjeta *visibilityOfElementLocated*. Osim vidljivosti postoje i uvjeti kao što su: element dostupan, element ima određen atribut, element nije prazan, stanje elementa je, element se može pritisnuti i sl.

Zadnjih se par metoda sve pozivaju na već pronađeni element što znači da se ulančaju na rezultat metode *findElement*.

Selenium ima mogućnosti utvrđivanja ili usporedbi tvrdnji putem *Assert* ili *SoftAssert* klase. Metode klase Assert se još zovu „tvrde tvrdnje“ jer se koriste kada se želi zaustaviti s izvođenjem testne skripte (ili metode) čim se dobije neodgovarajući rezultat, tj. kada se uvjet tvrdnje ne podudara s očekivanim rezultatom. Metode klase *SoftAssert* ne zaustavljaju izvođenje testova već se na kraju izvođenja prikažu sve pronađene greške. Neke od metoda u ovim klasama su *assertTrue* i *assertEquals.* Metoda *assertTrue* utvrđuje točnost dane izjave. Koristi se za npr. provjeru je li element vidljiv. Metoda *assertEquals* utvđuje podudarnost dva predana parametra. Koristi se za npr. provjeru je li elementov naziv ispravan.

3.4. Usporedba testnih alata Cypress i Selenium

Cypress podržava JavaScript i Typescript, dok Selenium podržava više popularnih programski jezika. Selenium i Cypress podržavaju Chrome, Edge i Firefox, ali Selenium dodatno podržava IE, Safari i Operu, dok Cypress podržava Electron preglednik. Cypress je jednostavniji za postavljenje jer nisu potrebna dodatna preuzimanja ili ovisnosti. Selenium je star te time i ima široku zajednicu, iako Cypress-ova zajednica brzo raste. Selenium omogućuje pokretanje testova na više različitih preglednika i podržava testiranje u više prozora (eng. tabs). Selenium kroz Appium može testirati mobilne aplikacije, dok Cypress ne može. Cypress nije dizajniran da u potpunosti zamijeni Selenium. Prema [30], oba alata se znatno razlikuju u arhitekturi i performansama. Cypress testovi se provode direktno u pregledniku te time ne gube vrijeme izvođenja na pokretanje upravljačkih programa za preglednike kao Selenium. Selenium nema za cilj biti okvir za sve u jednom paketu već njegova mogućnost integracije ostalih okvira mu omogućuje lako proširivanje i nadograđivanje u svrhu stvaranja savršenog cjevovoda automatizacije testiranja. Glavna razlika je dodatno vidljiva u njihovom načinu izvođenja testova. Selenium se izvodi sinkrono, radi sve liniju po liniju, te uz naredbe za čekanje npr. vidljivosti pojedinog elementa s metodom get.wait().until(...) može osigurati da je stranica u ispravnom koraku za testiranje. Kod Cypressa naredbe su asinkrone i stavljaju se u red za kasnije izvršavanje. Problem kod ovoga je da se naredbe ponekad pokreću prije nego li su se svi elementi stranice učitali do kraja. Cypress nema mogućnost čekanja za element osim ako se ne navede točno vrijeme čekanja, ali se time produžuje vrijeme izvođenja testova. Također, Cypress ne omogućuje spremanje elemenata unutar varijabli za daljnje izvođenje zbog svoje asinkrone prirode. Npr. ako želimo dohvatiti gumb u varijablu *Button* pa na taj *Button* pozvati *click*, ne će raditi. Poziv za dobivanje metode, *cy.get* je asinkron te se ne će

postaviti u varijablu prije nego što se na njega pozove naredba *cy.click*. Ideja je ulančati naredbe jer Cypress naredbe daju (ne vraćaju) objekte lančanicima koji stavljaju radnje u red čekanja izvođenja.

4. DEFINIRANJE TESTNE OKOLINE I POSTAVKE TESTIRANJA

U ovom poglavlju postavlja se testna okolina, objašnjava se web aplikacija tvrtke i testni plan. Web aplikacija se sastoji od više stranica te je svaka opisana sa svojim ključnim funkcionalnostima koje je potrebno testirati. Definira se testni plan koji sadrži upute za daljnje testiranje u petom poglavlju.

4.1. Testna okolina

Operacijski sustav na kojem se provodi testiranje je Windows 10. Selenium i Cypress testovi se pišu i provode u IntelliJ programu. Selenium testovi se pišu u Java programskom jeziku uz TestNG programski okvir, dok se Cypress testovi pišu u TypeScriptu. Testovi se pretežito provode u Google Chrome pregledniku, ali se koriste i Edge i Firefox (Selenium) ili Electron (Cypress) za nefunkcionalno testiranje.

4.2. Web aplikacija za prodaju auta

Web aplikacija tvrtke komplicirana je aplikacija koja se sastoji od više funkcionalnosti vezanih za prodaju auta. Dio aplikacije koji će se testirati u ovom diplomskom radu se sastoji od tri stranice:

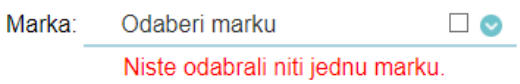
- izvješće prodaje po prodajnim savjetnicima
- izvješće prodaje po markama
- učinkovitost prodaje po prodajnom savjetniku.

Sve tri stranice sastoje se od filtara koji omogućuju korisniku pretraživanje. Rezultati pretraživanja prikazuju se kao tablice ili grafički prikazi (*eng. widget*). Slike prikazane u diplomskom radu ne reprezentiraju stvarni izgled aplikacije na kojoj radi tvrtka kako bi se izbjeglo kršenje autorskih prava. Web aplikacija je prevedena na hrvatski za lakše razumijevanje te je pojednostavljen i izmijenjen raspored korisničkog sučelja, dok je većina funkcionalnosti je zadržana.

4.2.1. Izvješće prodaje po prodajnim savjetnicima

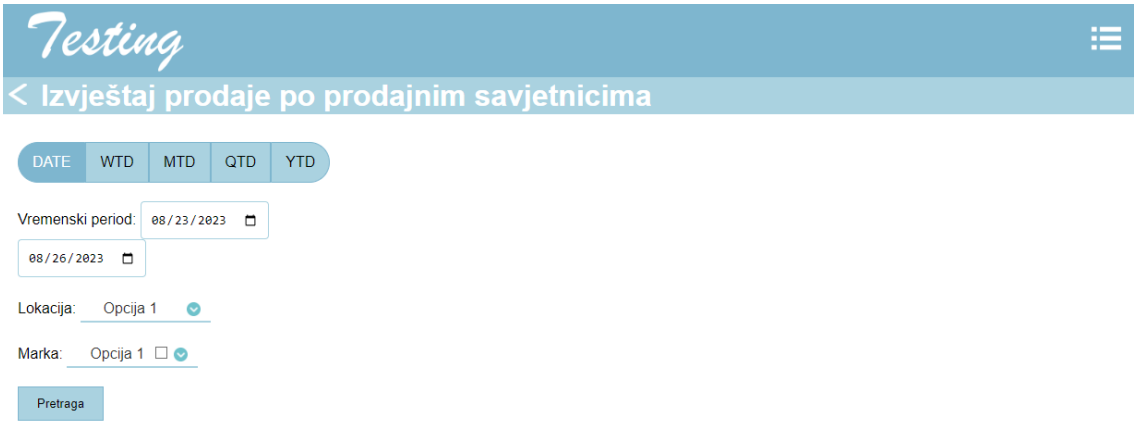
Otvaranjem izvješća filtri su postavljeni na zadane vrijednosti koje ovise o odabranom gumbu perioda, a tablica i widgeti nisu prikazani. Odabirom filtra i klikom na gumb „Pretraga“ moguće je prikazati tablice i widgete. Za ostvarivanje pretrage korisnik mora unijeti valjani period i marku (ili više njih). U slučaju nepoštivanja toga, korisniku se prikazuju verifikacijske poruke ispod filtra

koji nije ispunjen. Verifikacijske poruke se prikazuju kada korisnik klikne na gumb „Pretraga“ te slika 4.1 prikazuje primjer.



Slika 4.1. Primjer verifikacijske poruke

Slika 4.2 prikazuje početno stanje web aplikacije prije vršene pretrage.



Slika 4.2. Prikaz početnog stanja stranice „Izvješće prodaje po prodajnim savjetnicima“

Filtri vidljivi na slici 4.2 su:

- birač datuma – od do
- period (DATE, WTD, MTD, QTD, YTD)
- ime prodajne lokacije
- popis marki – opcija više odabira.

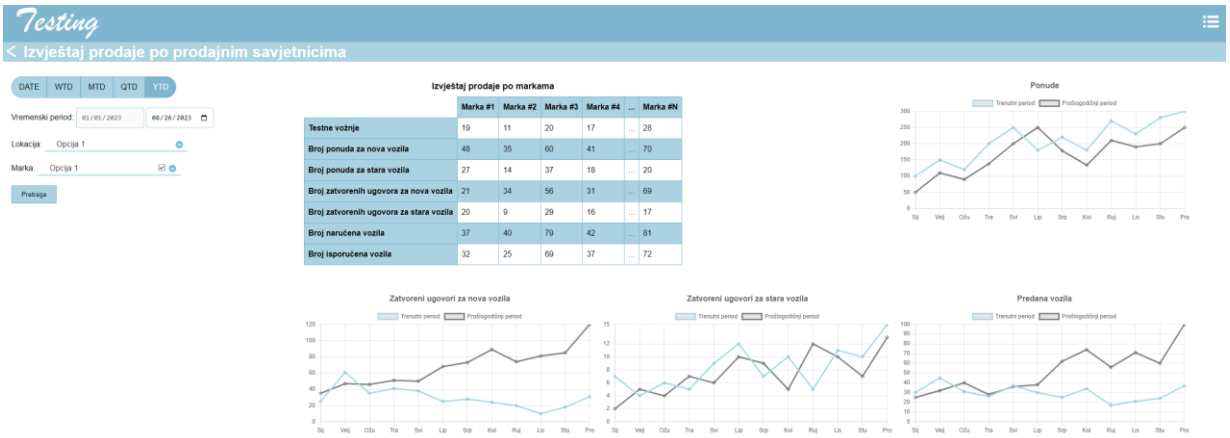
Filtri perioda utječu na birače datuma. Kada je „DATE“ uključen, korisnik bira početni i završni datum perioda, dok kod ostala četiri gumba korisnik može birati samo završni datum, a početni je definiran ovisno o gumbu. „WTD“ predstavlja početni tjedan odabranog datuma, „MTD“ je početni mjesec odabranog datuma, „QTD“ je najbliži kvartil prije datuma te „YTD“ predstavlja početnu godinu odabranog datuma. Prelaskom miša preko periodnih filtara treba se prikazati puni naziv filtra. Primjeri za korisnički postavljen datum i datum koji je prikazan za početno vrijeme perioda vidljivi su u tablici 4.1.

Tablica 4.1. Vrijednosti perioda u ovisnosti o odabranom gumbu

	Odabran datum završetka	Prikazan datum početka
WTD	26.08.2023.	21.08.2023.
MTD	26.08.2023.	01.08.2023.
QTD	26.08.2023.	01.07.2023.
YTD	26.08.2023.	01.01.2023.

„DATE“ opcija zahitjeva period od minimalno tri dana kako bi se omogućilo pretraživanje. „DATE“ i „WTD“ opcije pri pretraživanju prikazuju samo tablicu, dok „MTD“, „QTD“ i „YTD“ prikazuju tablicu i widgete. Slika 4.3 prikazuje primjer rezultata pretrage „DATE“ i „WTD“ filtra, dok slika 4.4 prikazuje primjer pretrage po „MTD“, „QTD“ i „YTD“ filtrima. Oba primjera imaju uključen višestruki odabir marki. Na slici 4.3 vidljivo je da pri odabiru filtra koji nisu „DATE“, lijevi birač datuma koji se postavlja prema odabranom filtru je onemogućen te korisnik ne može mijenjati datum.

Slika 4.3. Prikaz pretrage po „DATE“ periodu



Slika 4.4. Prikaz pretrage po „YTD“ periodu

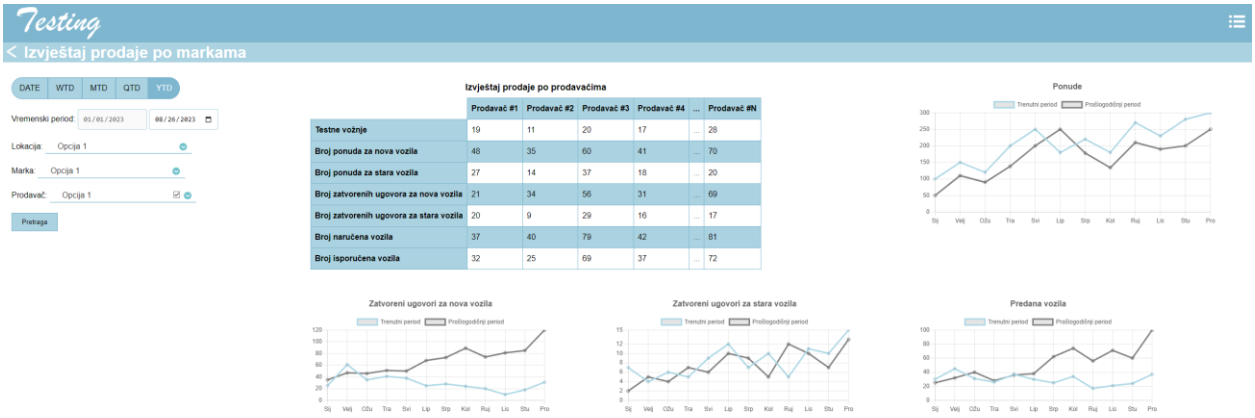
Widgeti prikazani na slici 4.4 su:

- ponude
- zatvoreni ugovori za nova vozila
- zatvoreni ugovori za stara vozila
- predana vozila.

Grafički prikaz ponuda predstavlja prikaz broja predanih ponuda za odabrane marke u odabranom periodu. Grafički prikazi zatvorenih ugovora za stara i nova vozila te predana vozila isto tako ovise o odabranim markama i odabranom periodu pri pretrazi. Tablica prikazana na slikama 4.3 i 4.4 predstavlja prikaz informacija po odabranim markama. Tablica može imati samo jedan stupac „Marka #1“ ili više njih ovisno o broju odabranih marki. Tablica sadrži informacije o: broju testnih vožnja, broju ponuda za nova i stara vozila, broju zatvorenih ugovora za nova i stara vozila, broj naručenih i isporučenih vozila. Klikom na naziv marke unutar tablice, korisnik je preusmjeren na stranicu „Izvješće prodaje po markama“ te je odabrana marka postavljena kao filtar na toj stranici. Osim kliknute marke, prethodno odabrani period i lokacija također se postave kao zadane vrijednosti na preusmjerenoj stranici.

4.2.2. Izvješće prodaje po markama

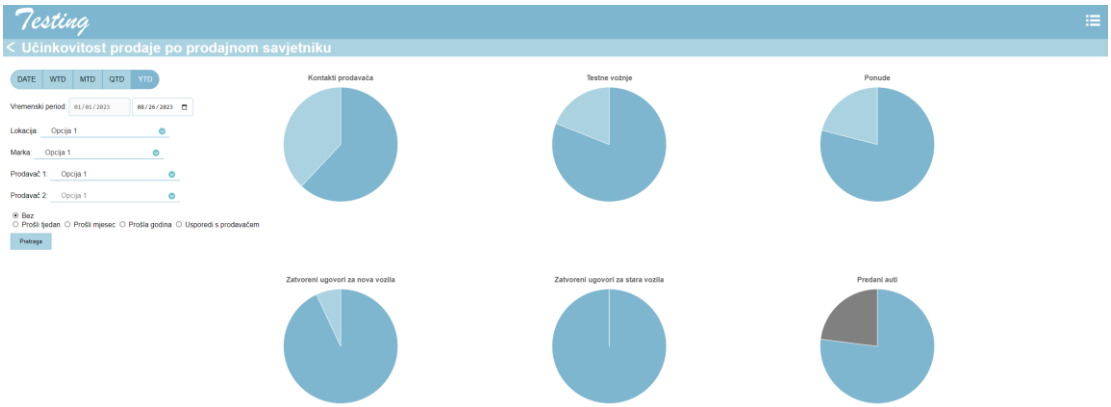
Stranica „Izvješće prodaje po markama“, slika 4.5, pretežito je slična prethodno objašnjenoj stranici. Razlike su u filtru marki gdje se na ovoj stranci može pretraživati samo po jednoj marki, te ima novi filtar „Prodavači“ koji ima opciju višeg odabira prodajnih savjetnika. Prema tome, ova stranica se fokusira na izvješće prodaje po markama za pojedinog prodavača ili više odabranih. Time se može utvrditi koliko je koji prodavač u kojem periodu uspio zatvoriti ugovora, predati vozila i ponude. Kako bi se pretraga mogla izvršiti potrebno je odabrati period i prodajnog savjetnika. Ako se ne odabere prodajni savjetnik i klikne se na gumb „Pretraga“, dobiva se verifikacijska poruka: „Niste odabrali niti jednog prodajnog savjetnika.“ Dodatna razlika je u tablici koja umjesto „Marka #1“ prikazuje „Prodavač #1“ te klikom na bilo koje ime prodajnog savjetnika korisnik je preusmjeren na stranicu „Učinkovitost prodaje po prodajnom savjetniku“ na kojoj su prikazani filtri sa stranice „Izvješće prodaje po markama“ te ime prodajnog savjetnika na kojeg smo kliknuli.



Slika 4.5. Prikaz stranice „Izvješće prodaje po markama“ prema „YTD“ pretrazi

4.2.3. Učinkovitost prodaje po prodajnom savjetniku

Filtri na ovoj stranici nemaju opciju višeg odabira nego sve ili samo jedan odabir za lokaciju i marku. Postoji dodatni filtar za odabir drugog prodajnog savjetnika, a oba filtra za prodajne savjetnike mogu odabrati samo jednu osobu. Kako bi se obavila pretraga potrebno je odabrati barem jednog prodajnog savjetnika inače se prikaže prije spomenuta verifikacijska poruka. Postoje dodatne opcije za filtriranje: bez, prošli tjedan, prošli mjesec, prošla godina i usporedi s prodavačem. Filtar za drugog prodajnog savjetnika je uvijek onemogućen osim u opciji „Usporedi s prodavačem“. Odabirom filtara kao u prošle dvije stranice uz odabrani dodatni filtar „Bez“ dobiva se rezultat kao na slici 4.6.



Slika 4.6. Prikaz stranice učinkovitost prodaje po prodajnom savjetniku prema "YTD" pretrazi

Ako se klikne na filtre: prošli tjedan, prošli mjesec, prošla godina i usporedi s prodavačem prikazuju se drugačiji grafovi. Kod prve tri opcije filtri su kao na slici 4.7, dok kod usporedbe s prodavačem grafovi ne uspoređuju prošlogodišnji period sa odabranim već se uspoređuju odabrana dva prodajna savjetnika. Odabirom filtra prošli tjedan, prošli mjesec ili prošla godina automatski se postavi na odgovarajući periodski filtar „WTD“, „MTD“ ili „YTD“ te se primjenjuje prethodno objašnjeni period ovisno o odabiru.

Slika 4.7. Prikaz stranice učinkovitost prodaje po prodajnom savjetniku prema dodatnom filtru "Prošla godina"

4.3. Značenje testiranja u razvoju web aplikacije

Razvijanje i izmjena svake komponente i značajke zahtijeva testiranje istih kako bi se potvrdila njihova ispravnost te poštivanje zahtjeva koji su opisani u dokumentaciji. Provođenjem ovih testova omogućava se dobivanje povratne informacije kako bi se znalo trenutno stanje aplikacije i eventualne greške ili nedostaci na koje se treba obazirati i ispraviti. Testiranjem se potvrđuje da je svaka inačica aplikacije spremna za puštanje na tržište. Svakim novim izdanjem potrebno je provjeriti kroz regresijske testove da se nešto nije pokvarilo prilikom dodavanja, primjerice, nove značajke. Rezultati testiranja se uvijek prijavljuju i dostupni su programerima na Internet uslugama kao što je JIRA, stranica koja omogućuje lakšu organizaciju i vođenje projekata, kako bi se olakšalo prenošenje informacija i spremanje rezultata. Na toj istoj stranici kreiraju se izvješća o nedostacima, ako takvih ima.

4.4. Testni plan

Testne strategije koje se koriste su strategije nesklone regresiji i analitičke strategije. Glavna strategija je strategija nesklona regresiji jer je aplikacija testirana u ovom radu samo dio velike aplikacije. Sa svakim novim dodatkom ili promjenom kôda bitno je da promjene ne utječu na ispravnost testirane aplikacije, a putem regresijskog testiranja možemo omogućiti brzu provjeru istoga. Analitičke strategije primjenjuju se na način da se provjerava potencijalan rizik koji imaju određeni zahtjevi na rad aplikacije te se prema tome najviše prate ti visoko rizični testovi. Aplikacija će se testirati putem: ispitivanja dima, testiranja od kraja do kraja, regresijskim testiranjem, testiranjem funkcionalnosti (pod E2E) i testiranjem integracije komponenti. Korištene tehnike su tehnike crne kutije jer ne vidimo cijeli kôd aplikacije, i tehnike na temelju iskustva.

Testni scenariji prema kojima će se provoditi testiranje dizajnirani su prema mogućem korisničkom načinu korištenja web aplikacije tj. korisničke priče. Ti scenariji glase:

1. korisnik otvori stranicu
2. korisnik pretražuje s nepravilnim filtrima
3. korisnik pretražuje s pravilnim filtrima
4. korisnik pritisne na poveznicu u tablici.

Tijekom prvog scenarija za automatizirano testiranje će se provoditi *smoke testing* i provjera zadanih postavki pri otvaranju stranice za sve tri stranice. Pregledava se opća valjanost stranice tj. da se vide svi elementi koji se trebaju vidjeti. Drugi scenarij i treći scenariji predstavljaju

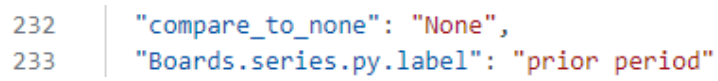
funkcionalno testiranje. Drugi scenarij provjerava funkcionalnosti verifikacijskih poruka, dok treći scenarij pokriva ostale funkcionalnosti. Četvrti scenarij provodi testiranje integracije komponenti na način da provjerava je li komunikacija između poveznica valjana te postavljaju li se pravilne vrijednosti za odabrane vrijednosti filtara. Ručno testiranje je istraživačko te prati opise scenarija.

4.5. Testni podaci za automatizirano testiranje

Svi podaci koji će se koristiti u testiranju su odvojeni u posebne datoteke. Slika 4.8 pokazuje primjere spremanja naziva ID-eva elemenata stranice koji će se testirati. Slika 4.9 prikazuje primjere veza prema kojima se testira trenutačna stranica. Na istoj slici vidljiv je primjer testnih podataka za provjeru slike loga koji se prikazuje na stranici. Slika 4.10 prikazuje primjer spremanja tekstualnih vrijednosti potrebnih za usporedbe vidljivog teksta i prijevoda. Izbjeljeni dijelovi na slikama su osjetljive informacije tvrtke.

Slika 4.8. Isječak spremanja testnih podataka - ID

Slika 4.9. Isječak spremanja testnih podataka - veze i logo



Slika 4.10. Isječak spremanja testnih podataka - tekstualne vrijednosti

5. RUČNO I AUTOMATIZIRANO TESTIRANJE WEB APLIKACIJE

U ovom poglavlju obrađuje se izvedba ručnog i automatiziranog testiranja koji poštuju upute zadane u testnom planu kao i zahtjeve i specifikacije klijenta. Opisuju se načini testiranja za svaki scenarij i kako se koji automatizirani test ostvario unutar svog programskog jezika i biblioteka.

5.1. Ručno testiranje

5.1.1. Prvi scenarij – ručno testiranje

Tijekom ručnog testiranja prvog scenarija prvi korak je otvaranje prve stranice i pregled filtara, jesu li svi na mjestu ili ne. Potrebno je provjeriti funkcioniraju li tj. kada se klikne na filtar te odabere ili unese vrijednost, da se ta vrijednost primjeni. Potrebno je provjeriti da prilikom učitavanja stranice prije pretrage nisu vidljivi widgeti i/ili tablice. Potrebno je provjeriti svaki filtar i više pravilnih odabira filtara pri pretraživanju kako bi se utvrdilo da je aplikacija dovoljno funkcionalna za daljnje testiranje.

5.1.2. Drugi scenarij – ručno testiranje

Tijekom drugog scenarija potrebno je provjeriti sve kombinacije filtara koji se mogu definirati kao „nepravilno“ uneseni. Svaka stranica ima definirane filtre bez kojih je pretraga nemoguća, a ti filtri su navedeni ispod.

- za stranicu „Izvješće prodaje po prodajnim savjetnicima“ obvezan filtar je odabir marke ili više njih
- za stranicu „Izvješće prodaje po markama“ obvezan filtar je odabir prodajnog savjetnika ili više njih
- za stranicu „Učinkovitost prodaje po prodajnom savjetniku“ obvezan filtar je odabir jednog prodajnog savjetnika te u slučaju usporedbe dva prodajna savjetnika, obvezan filtar je odabir i drugog prodajnog savjetnika.

Potrebno je proći kroz sve kombinacije i provjeriti pojavljuju li se verifikacijske poruke pravilno i je li njihov tekst ispravan. Tekst verifikacijskih poruka za filtre je slijedeći:

- za odabir marke: „Niste odabrali niti jednu marku“
- za odabir prodajnog savjetnika: „Niste odabrali niti jednog prodajnog savjetnika“
- za odabir prvog prodajnog savjetnika: „Niste odabrali prodajnog savjetnika 1“, te za odabir drugog prodajnog savjetnika: „Niste odabrali prodajnog savjetnika 2“.

Dodatno je potrebno utvrditi da pritiskom gumba „Pretraga“ se prikažu samo verifikacijske poruke, a ne tablice i/ili widgeti.

5.1.3. Treći scenarij – ručno testiranje

Nasuprot drugog scenarija, treći scenarij zahtijeva pravilan unos prethodno navedenih obveznih filtara i pretrage. Potrebno je provjeriti prikazuju li se tablice i/ili widgeti povodom pretrage. S obzirom na to da neki od filtara imaju više opcija, ovo je scenarij koji zahtijeva najviše vremena u ručnom testiranju. Potrebno je provjeriti sve ili što više funkcionalnosti i kombinacija filtara. Periodni filtri („DATE“, „WTD“, „MTD“, „QTD“ i „YTD“) i filtar za odabir datuma (funkcionalnost kalendara) su među važnijim funkcionalnostima koje se trebaju provjeriti prema prethodno navedenim pravilima. Pri testiranju datuma potrebno je provjeriti granične vrijednosti datuma. Npr. za „WTD“ potrebno je provjeriti ponedjeljak i nedjelju kao početni i zadnji dani u tjednu. Ako je krajnji datum postavljen na ponedjeljak, onda početni datum ne može biti isti ponedjeljak već se mora uzeti prethodni tjedan. Ovakva vrsta testiranja spada pod crnu kutiju, analizu graničnih vrijednosti. Za stranicu „Učinkovitost prodaje po prodajnom savjetniku“ potrebno je provjeriti ako se klikne na filtre: prošli tjedan, prošli mjesec i prošla godina da su se pravilno pomjerili vremenski filtri i datumi. Također, pri odabiru usporedbe s prodavačem, drugi prodajni savjetnik se može odabrati, a u suprotnom ne. Pri odabiru filtra bez ili usporedi s prodavačem, vremenski filtri se ne mijenjaju.

5.1.4. Četvrti scenarij – ručno testiranje

Četvrti scenarij predstavlja scenarij koji započinje s pritiskom na poveznicu koja se nalazi u tablici prve spomenute stranice. Poveznica bi trebala otvoriti drugu stranicu te postaviti filtre na odabrane filtre s prve stranice i pritisnute poveznice. Pritiskom na poveznicu u tablici druge stranice otvara se zadnja stranica te se postavljaja filtar s druge stranice i pritisnute poveznice. Potrebno je provjeriti da su se stranice pravilno promjenile prilikom pritiska na poveznicu i da su se promjenili filtri na odgovarajuće. Prilikom otvaranja nove stranice preko poveznice widgeti i/ili tablica se ne prikazuju dok se ne obavi pretraga sa popunjenim obveznim filtrima.

5.1.5. Postupci provođenja testova

Ručni testovi kao i automatizirani testovi se provode korak po korak, no razlika u izvođenju ručnih testova je to što su oni zapisani u normalnom opisnom tekstu. To su pojednostavljene upute kojih se treba držati te na kraju njih treba se prikazati rezultat. Tablica 5.1 prikazuje primjer zapisanog ručnog testa iz trećeg scenarija unutar JIRA stranice.

Tablica 5.1. Primjer zapisa ručnog testa

Broj testa	Akcija	Podaci	Očekivani Rezultat
1	Otvorite stranicu „Izvješće prodaje po prodajnim savjetnicima“		• Izvješće je učitano i naziv izvješća je vidljiv i ispravno preveden. • Filtri i gumb za izračunavanje vidljivi su i na njih se može kliknuti. • Prilikom prelaskom miša iznad gumba WTD, MTD, QTD i YTD, prikazuje se puno ime. • Widgeti i tablica nisu vidljivi.
2	Odaberite filtre i pritisnite gumb „Pretraga“		• Widgeti i tablica su vidljivi i prikazuju podatke. • Popis widgeta: Ponude, Zatvoreni ugovori za NV, Zatvoreni ugovori za UV i Predana vozila. • Tablica „Izvješća prodaje po markama“ je vidljiva. Prikazuje se okomita traka za pomicanje ako broj stupaca proširuje širinu zaslona. • Širina tablice uvijek ostaje iste veličine i responzivna je na svim uređajima (poklapa se s krajem 1. i 4. widgeta). • Naziv tablice i vrijednosti pohranjeni su u svakom retku vodoravno. Samo nazivi glavnih stupava i redova su napisani podebljano.
3	Prelazite mišem preko tablice		• Redovi u tablici označeni su (svijetlozeleno) kada se pređe mišem preko njih.

5.2. Automatizirano testiranje web aplikacije u alatu Cypress

5.2.1. Prvi scenarij – automatizirano (Cypress)

Primjer prvog scenarija će se obraditi na stranici „Izvješće prodaje po prodajnim savjetnicima“. Pri automatizaciji potrebno je postaviti testne podatke. Svi testni podaci su podijeljeni ili po funkcionalnosti ili po stranici te prema tome postoji lakše i razumno dohvaćanje podataka. Kao što je prethodno pokazano, slika 5.1 prikazuje dohvaćanje podataka, a slika 5.2 prikazuje primjer podataka za prve dvije varijable sa slike 5.1. Podaci se zapisuju unutar varijabli jer Cypress testovi se ne stvaraju dinamički.

```
9   const defaultFilters :string [] = getKeyElementsFromJson(Board1.paths.filters);
10  const allPeriods :string [] = getKeyElementsFromJson(Board1.paths.period);
11  const allWidgets :string [] = getKeyElementsFromJson(Board1.paths.widgets);
12  const allDates :string [] = getKeyElementsFromJson(Board1.paths.date);
13  const allButtons :string [] = getKeyElementsFromJson(Board1.paths.buttons);
14  const allMultiselect :string [] = getKeyElementsFromJson(Board1.paths.multiselect);
```

Slika 5.1. Postavljanje testnih podataka

```
114     "period": {
115       "datePeriod": "DATE",
116       "wtdPeriod": "WTD",
117       "mtdPeriod": "MTD",
118       "qtdPeriod": "QTD",
119       "ytdPeriod": "YTD"
120     },
121     "filters": {
122       "site": "ui.site.label"
123     },
```

Slika 5.2. Prikaz testnih podataka korištenih u testu

Prije svih testova potrebno je kreirati sesiju za danu poveznicu. Sesija je vremenski omeđena dvosmjerna veza koja omogućuje razmjenu informacija. Slika 5.3 prikazuje metodu *before* koja se pokreće jednom pri pokretanju testnog bloka. Sve navedene funkcije su napravljene i dodane u Cypress u *commands.ts* datoteku.

```
16  before(() => {
17      cy.sessionLink(Cypress.env('ApplicationBaseUrl'), link.links.Board1);
18      cy.verifyDashboardName(Board1.translation.boardName);
19      cy.waitSpinner();
20  });
```

Slika 5.3. Funkcija before

Funkcija *sessionLink* vidljiva na slikama 5.3 i 5.4 čisti sve prethodne sesije ako takve ima te kreira novu za zadanu poveznicu koju onda posjećuje.

```
29  Cypress.Commands.add('sessionLink', (baseUrl, link) => {
30      Cypress.session.clearAllSavedSessions();
31      cy.session('session', ()=> {
32          cy.login(baseUrl);
33      });
34      cy.visit(baseUrl + link);
35      cy.initializeSessionData(baseUrl + link);
36      cy.visit(baseUrl + link);
37  });
```

Slika 5.4. Funkcija sessionLink

Na slici 5.5 se vidi funkcija *verifyDashboardName* koja služi za provjeru naziva *h2* HTML elementa. U ovom slučaju on treba glasiti „Izvješće prodaje po prodajnim savjetnicima“. Unutar funkcije zove se funkcija *compareTranslation*, vidljiva na slici 5.6. . U toj funkciji provjerava se potrebno korištenje prijevoda. Aplikacija trenutačno podržava dva jezika: engleski i njemački. Nakon provjere uzima se točan tekst *h2* HTML elementa i uspoređuje se sa stvarnim tekstom.

```
126 Cypress.Commands.add('verifyDashboardName', (translation) => {
127     cy.get('h2').should('be.visible').within(($title) => {
128         cy.compareTranslation(translation, $title.text());
129     });
130 });
```

Slika 5.5. Funkcija verifyDashboardName

```
110 Cypress.Commands.add('compareTranslation', (expected, actual) => {
111     console.log('Translaion: ' + useTranslation);
112     if (useTranslation)  {
113         expect(actual).to.be.equal(expected);
114     }
115     else {
116         let translated = translation[expected];
117         console.log("Translated: " + translated);
118         expect(actual).to.be.equal(translated);
119     }
120 });
```

Slika 5.6. Funkcija compareTranslation

Aplikacija ima svoj ugrađeni element koji se vrti, prema tome i naziv *spinner*, kada aplikacija nije spremna tj. dok se još učitava. Slika 5.7 prikazuje funkciju *waitSpinner* koja provjerava da taj element nije vidljiv te tada nastavlja s testiranjem.

```
201 Cypress.Commands.add("waitSpinner", () => {
202     cy.get('clr-progress-spinner').should('not.be.visible');
203 });
204
```

Slika 5.7. Funkcija waitSpinner

Ako bilo koji od ovih testova padne, testiranje se zaustavlja te su ostali testovi isto pali. Ovo su nužni uvijeti koji moraju biti ispunjeni za izvođenje testova. Osim funkcije *before*, funkcija *beforeEach* (slika 5.8) se također koristi. Ova funkcija se pokreće prije svakog testa te provjerava postojanje sesije, posjećuje poveznicu putem *visit* i čeka da se sve na stranici učita (funkcija *waitSpinner*) prije početka izvođenja testiranja.

```
22 beforeEach(() => {
23     if(document.getElementsByTagName('h2')){
24         cy.checkSession(Cypress.env('ApplicationBaseUrl'), link.links.Board1);
25         cy.intercept("GET", interceptLink.controlling.Board1).as("Board1");
26         cy.visit(Cypress.env('ApplicationBaseUrl') + link.links.Board1);
27         cy.waitSpinner();
28         }
29 });
```

Slika 5.8. Funkcija beforeEach

Funkcija *checkSession* vidljiva na slici 5.9 provjerava postoji li sesija, te ako ne postoji ona se kreira. Funkcija *intercept* je ugrađena naredba Cypress-a putem koje je moguće presresti zahtjeve i odgovore na strani poslužitelja.

```
39  Cypress.Commands.add('checkSession', (baseUrl, link) => {
40      cy.session('session', ()=> {
41          cy.login(baseUrl);
42      });
43      console.log(document.URL);
44      if(document.URL.startsWith(baseUrl))
45          cy.visit(baseUrl + link);
46
47  });
```

Slika 5.9. Funkcija checkSession

Nakon postavljanja metoda koje se pokreću prije testova, vrijeme je i za same testove. Prvi test koji ispituje dim je test za provjeru prikaza elemenata odabira datuma te se nalazi na slici 5.10. Test se provodi za svaki takav element putem ugrađene naredbe *forEach*. Funkcija *verifyDate* je vidljiva na slici 5.11 Putem ove funkcije provjerava se jesu li elementi vidljivi i imaju li početni zadani tekst koji glasi „DD/MM/YYYY“. Nakon toga se provjerava da pored elementa postoji ikona kalendara.

```
34          allDates.forEach((date) => {
35              it(`Verify date ${date}`, () => {
36                  cy.verifyDate(Board1.paths.date[date], Board1.status.date[date]);
37              });
38          });
```

Slika 5.10. Provjera prikaza elemenata odabira datuma

```
151 Cypress.Commands.add('verifyDate', (path, state) =>  {
152     cy.get(path).find('input')
153         .should('be.visible')
154         .and(state)
155         .and('have.attr', 'placeholder', 'DD/MM/YYYY');
156     cy.get(path).find('    icon').should('have.attr', 'shape', 'calendar');
157 });
```

Slika 5.11. Funkcija verifyDate

Testovi vidljivi na slici 5.12 tiču se perioda filtara. Prvo se provjerava jesu li prikazana točno pet elementa, a onda se putem funkcije *verifyPeriod*, vidljive na slici 5.13, provjerava ispisani tekst i vidljivost elemenata.

```
40      it('Verify number of period filters', () => {
41          cy.get(Board1.paths.period.mtdPeriod)
42              .find('clr-radio-wrapper').should('have.length', 5);
43      });
44
45      allPeriods.forEach((period) => {
46          it(`Verify period ${period}`, () => {
47              cy.verifyPeriod(Board1.paths.period[period], Board1.position.period[period],
48                  Board1.translation.period[period]);
49          });
50      });
```

Slika 5.12. Testovi vremenskih perioda

```
143 Cypress.Commands.add('verifyPeriod', (path, index, translation) => {
144     cy.get('clr-radio-wrapper').eq(index).within(() => {
145         cy.get("span[class='text-position']").within(($name) => {
146             expect($name.text().trim()).to.be.equal(translation);
147         });
148     }).should('be.visible');
149 });
```

Slika 5.13. Funkcija verifyPeriod

Test vidljiv na slici 5.14 provjerava broj vrijednosti padajućih izbornika za svaki HTML element *select* na stranici. Provjerava se točnost prijevoda odabrane opcije putem funkcije *verifyDropdowns* vidljivoj na slici 5.16. Funkcija *insertLastDayOfPreviousMonth* sa slike 5.15 postavlja vrijeme na početni datum prethodnog mjeseca.

```
52      defaultFilters.forEach((filterName) => {
53          it(`Verify filter ${filterName}`, () => {
54              cy.insertLastDayOfPrevousMonth(Board1.paths.date.dateTo);
55              cy.waitSpinner();
56              cy.verifyDropdowns(Board1.paths.filters[filterName], Board1.translation.filters[filterName]);
57          });
58      });
```

Slika 5.14. Test filtara

```
211 Cypress.Commands.add('insertLastDayOfPrevousMonth', (path) => {
212     cy.insertdate(path, moment(new Date()).subtract(1,'months')
213         .endOf('month').toDate().toLocaleDateString('en-GB'));
214 });
```

Slika 5.15. Funkcija insertLastDayOfPreviousMonth

Funkcija *verifyDropdowns* provjerava jesu li elementi vidljivi i imaju li minimalno jednu vidljivu opciju. Nakon toga se provjerava prijevod teksta odabrane opcije.

```
132 Cypress.Commands.add('verifyDropdowns', (path, translation) => {
133     cy.get(path)
134         .should('be.visible').find('option').should('have.lengthOf.at.least', 1);
135     cy.get(path + " option:selected")
136         .within(($filterName) => {
137             cy.compareTranslation(translation, $filterName.text().trim())
138         });
139 });
```

Slika 5.16. Funkcija verifyDropdowns

Test više odabira, slika 5.17, izvodi se za svaki element *select* koji ima odabir više opcija. Funkcija *verifyMultiselect* na slici 5.18 provjerava vidljivost i prijevod takvih elemenata. Slika 5.18 prikazuje drugi način kreiranja novih funkcija u Cypress-u, van *commands.ts* datoteke.

```
60         allMultiselect.forEach((multiselect) => {
61             it(`Verify multiselect ${multiselect}`, () => {
62                 cy.insertLastDayOfPrevousMonth(Board1.paths.date.dateTo);
63                 cy.waitSpinner();
64                 Boards.verifyMultiselect(Board1.paths.multiselect[multiselect],
65                     Board1.translation.multiselect[multiselect]);
66             });
67         });
```

Slika 5.17. Test više odabira filtara

```
57 export function verifyMultiselect(path, translation){
58     cy.get(path)
59         .should('be.visible')
60         .find('label')
61         .within(($message) => {
62             cy.compareTranslation(translation, $message.text());
63         });
64 }
```

Slika 5.18. Funkcija verifyMultiselect

Test gumbova sa slike 5.19 provodi se za svaki gumb na stranici putem metode *verifyButton* sa slike 5.20. Funkcija *verifyButton* uspoređuje stanje i pravilan prijevod teksta na gumbu.

```
69     allButtons.forEach((button) => {
70         it(`Verify button ${button}`, () => {
71             cy.verifyButton(Board1.paths.buttons[button], Board1.status.buttons[button],
72                 Board1.translation.buttons[button]);
73         });
74     });
```

Slika 5.19. Test gumbova

```
160  Cypress.Commands.add('verifyButton', (path, state, translation) =>  {
161      cy.get(path)
162          .should(state)
163          .find('span').within(($buttonName) => {
164              cy.compareTranslation(translation, $buttonName.text().trim());
165          });
166  });
```

Slika 5.20. Funkcija verifyButton

Test na slici 5.21 prikazuje test za widgete i tablice. Ovaj test je jednostavan jer samo provjerava da svi takvi elementi nisu vidljivi.

```
76          allWidgets.forEach((widget) => {
77              it(`Verify widget ${widget}`, () => {
78                  cy.get(Board1.paths.widgets[widget]).should('not.be.visible');
79              });
80          });
81
82          it('Verify table', () => {
83              cy.get(Board1.paths.table.table).should('not.be.visible');
84          });
```

Slika 5.21. Test widgeta i tablice

Osim specifičnosti stranice, potrebno je provjeriti jesu li logo, naziv i elementi unutar navigacijske trake vidljivi. Slika 5.22 prikazuje test za naziv gdje se provjerava postoji li zadana klasa i odgovara li naziv zadanom nazivu. Slika 5.23 prikazuje provjeru loga tvrtke da ima odgovarajuću klasu, sliku i alternativan naziv (u slučaju da se slika ne učitava).

```
22          it('Verify title', () => {
23              cy.get(list.path.logoAndTitle)
24                  .find('span')
25                  .should('have.class', 'title')
26                  .within(($title) => {
27                      expect($title.text().trim()).to.be.equal(list.translation.title)
28                  });
29      });
```

Slika 5.22. Test naziva

```
31          it('Verify logo', () => {
32              cy.get(list.path.logoAndTitle)
33                  .find('img')
34                  .should('have.class', list.logo.class)
35                  .and('have.attr', 'alt', list.logo.alt)
36                  .and('have.attr', 'src', list.logo.src);
37          });
```

Slika 5.23. Test loga

Slika 5.24 prikazuje provjeru broja elemenata na navigacijskoj traci koji imaju padajuće vrijednosti. Te padajuće vrijednosti se provjeravaju na slici 5.25 s metodom sa slike 5.26, *verifyDashboardInList.*

```
39      it('Verify number of elements', () => {
40          cy.get(list.path.dashboardList).find('li').should('have.length', 3);
41      });
```

Slika 5.24. Test broja elemenata navigacijske trake

```
43  dashboards.forEach((dashboard) => {
44      it(`Verify ${dashboard} link `, () => {
45          verifyDashboardInList(list.path.dashboardList, list.position[dashboard],
46              list.links[dashboard], list.translation[dashboard])
47      });
48  })
```

Slika 5.25. Test elemenata navigacijske trake

Putem metode *verifyDashboardInList* provjerava se postoje li poveznice na elementima i odgovaraju li zadanim poveznicama. Provjerava se jesu li vidljive i omogućene, te je li im tekstualni naziv točan. Svi navedeni testovi u ovom poglavlju odgovaraju *smoke* testovima.

```
11  export function verifyDashboardInList(path, index, href, translation) {
12      cy.get(path).find('li')
13          .eq(index).find('a')
14          .should('have.attr', 'href', href)
15          .and('be.visible')
16          .and('not.be.disabled')
17          .within(($dashboardName) => {
18              cy.compareTranslation(translation, $dashboardName.text().trim());
19          });
20  }
```

Slika 5.26. Funkcija verifyDashboardInList

5.2.2. Drugi scenarij – automatizirano (Cypress)

Drugi scenarij obrađuje dio funkcionalnosti stranice namijenjen za korisnikove greške pri pretrazi. Slika 5.27 prikazuje provjeru verifikacijskih poruka. Prvi test na slici provjerava prikazuje li se verifikacijska poruka, a drugi je li se verifikacijska poruka uklonila odmah nakon odabira jedne ili više opcija elementa *select*. Funkcija *verifyVerificationMessage* na slici 5.28 provjerava tekst, atribute i ikonu ispred teksta.

```
24      describe('Verification message', () => {
25          it('Brand verification message on Board1', () => {
26              cy.verifyVerificationMessage(Board1.paths.multiselect.brand,
27                  Board1.paths.verificationMessages.brand,
28                  Board1.translation.verificationMessages.brand);
29          });
30
31          it('Brand verification message on Board1 after user select one', () => {
32              cy.verifyVerificationMessage(Board1.paths.multiselect.brand,
33                  Board1.paths.verificationMessages.brand,
34                  Board1.translation.verificationMessages.brand);
35
36                  Boards.selectAllInMultiselect(Board1.paths.multiselect.brand);
37              cy.get(Board1.paths.multiselect.brand).should('be.visible')
38                  .find(Board1.paths.verificationMessages.brand).should('not.exist');
39          });
40      });
```

Slika 5.27. Test verifikacijskih poruka

```
176  Cypress.Commands.add("verifyVerificationMessage", (path, errorPath, translation) => {
177      cy.get(path).find(errorPath).within(($text) => {
178              cy.compareTranslation(translation, $text.text().trim());
179              cy.get('    icon').should('have.attr', 'shape', 'exclamation-circle')
180                  .and('have.attr', 'status', 'danger').and('have.attr', 'size', 'sm');
181          })
182  });
```

Slika 5.28. Funkcija verifyVerificationMessage

5.2.3. Treći scenarij – automatizirano (Cypress)

Za kreiranje testova ostalih funkcionalnosti i obradu trećeg scenarija gledati će se stranica „Učinkovitost prodaje po prodajnom savjetniku“ zbog većeg broja filtara. Za početak se testiraju periodni filtri te slika 5.29 prikazuje primjer testiranja za periodni filtar „YTD“. Putem naredbe *intercept* se presreće odgovor stranice. U tom odgovoru se nalaze datumi koji su postavljeni te preko njih možemo lagano provjeriti je li postavljen na odgovarajući datum. Metoda *clickPeriod* simulira pritisak na gumb danog periodnog filtra te čeka da stranica dovrši učitavanje. Metoda *verifyDates* na slici 5.30 koristi prethodno presretnuti odgovor stranice te provjerava je li ga u tom vremenu primila pa uspoređuje datume koje iščitava iz odgovora i one koji su predani kao argument. Prethodno objašnjena biblioteka *moment* olakšava proces određivanja i usporedbe datuma sa svojim ugrađenim metodama kao što su *startOf* i *format*.

```
40          it('verify WTD Period', () => {
41              cy.intercept("GET", interceptLink.boards.Board3).as('Board3');
42              Boards.clickPeriod(Board3.translation.period.wtdPeriod);
43              Boards.verifyDates('@Board3',
44                  moment(new Date()).startOf('week').format('DD/MM/YYYY'),
45                  moment(new Date()).format('DD/MM/YYYY'));
46          });
```

Slika 5.29. Test perioda „YTD“

```
114 export function verifyDates(waitResponse, startDate, endDate) {
115     cy.wait(waitResponse).then((response) => {
116         expect(response.response.statusCode).eq(200);
117         expect(parseDate(response.request.url)[0]).to.contain(startDate);
118         expect(parseDate(response.request.url)[1]).to.contain(endDate);
119     });
120 }
```

Slika 5.30. Funkcija verifyDates

Na slici 5.31, na liniji 69 se vidi uporaba funkcije *startOf* gdje se uzima početak godine od današnjeg datuma. Osim početka godine moguće je uzeti i početak mjeseca, tjedna i kvartila što je korišteno pri testiranju ostalih periodnih filtara. Važno je za naglasiti da je potrebno postaviti koji dan se gleda kao prvi dan tjedna jer neke države imaju za prvi dan tjedna nedjelju. Putem slike 5.31 se prikazuje funkcija *moment.locale* gdje se postavlja lokalitet i odabrani dan za početak tjedna.

```
moment.locale('en-gb', {
    week : {
        dow : 1 // Ponedjeljak je prvi dan tjedna
    }
});
```

Slika 5.31. Postavljanje prvog dana u tjednu

Test prelaska mišem preko periodnih filtara prikazan je na slici 5.32. Test se ponavlja za svaki period te se koristi metoda *verifyHover* sa slike 5.33. Ta metoda simulira pritisak na periodni filtar i provjerava putem funkcije *realHover* da se tijekom prelaska miša preko gumba pojavi tekst. Funkcija *realHover* je funkcija iz biblioteke *Cypress Real Events.*

```
24  describe('Hover functionality', () => {
25      periods.forEach((period) => {
26          it(`Verify hover message with period ${period}`, () => {
27              cy.waitSpinner();
28              Boards.verifyHover(Board3.translation.period[period], Board3.paths.period[period],
29                  Board3.translation.hover.period[period]);
30          })
31      });
32  });
```

Slika 5.32. Test prelaska mišem preko perioda

```
123  export function verifyHover(period , path, translation) {
124      cy.waitSpinner();
125      cy.contains(period).click();
126      cy.contains(path).realHover().should('be.visible');
127      cy.waitSpinner();
128      cy.contains(path).find('#tooltip-id')
129          .within($text => {
130              cy.compareTranslation(translation, $text.text().trim());
131          });
132  }
```

Slika 5.33. Funkcija verifyHover

Test sa slike 5.34 prikazuje provjeru prikaza widgeta prilikom pretrage koja se obavi pritiskom na gumb „Pretraga“. Prvo se unosi krajnji datum pretrage putem funkcije *insertdate* sa slike 5.35. Funkcija prvo čisti prethodne zapise i upisuje novi datum. Nakon toga se klikom na gumb „MTD“ kroz funkciju *clickPeriod* postavlja krajnji datum pretrage. Potrebno je i postaviti prodajnog savjetnika što se provodi funkcijom *clickOnSalesPerson*. Funkcija odabire prvu osobu u listi odabira. Nakon toga se pritisne gumb „Pretraga“ putem funkcije *clickOnButton*, a zatim se provjeravaju svaki widgeti putem funkcije *verifyWidget.*

```
30  describe('Test calculate button', () => {
31      it('Test calculate button', () => {
32          cy.insertdate(Board3.paths.date.dateTo, moment(new Date()).subtract(1,'months')
33          .endOf('month').format('DD/MM/YYYY'));
34
35          Boards.clickPeriod(Board3.paths.period.mtdPeriod);
36          Boards.clickOnSalesPerson(Board3.paths.filters.salesperson1, 1);
37          Boards.clickOnButton(Board3.paths.buttons.calculate);
38
39          Board3Widgets.forEach((widget) => {
40              Boards.verifyWidget(Board3.paths.widgets[widget],
41                  Board3.status.widgets.mtdPeriod,
42                  Board3.translation.widgets[widget], Board3.icon.widget[widget]);
43          });
44      });
```

Slika 5.34. Test gumba „Pretraga“

```
122  Cypress.Commands.add('insertdate', (path, date) => {
123      cy.get(path).find('input').clear().type(date).type('Cypress.io{enter}');
124  });
```

Slika 5.35. Funkcija insertdate

Funkcija *verifyWidget* sa slike 5.36 provjerava ima li widget u sebi vrijednost koja definira treba li se pokazati ili ne. Prema tome se odabiru widgeti koji će se pokazati. Ako se widgeti prikazuju onda se provjerava imaju li točnu ikonu te prikazuju li točan naziv na točnom prijevodu. Postoji

dodatni test koji koristi funkciju *verifyWidget*, no u njemu se provjeravaju ponašanja widgeta za sve periodne filtre.

```
90  export function verifyWidget(path, status, translation, icon)  {
91      if(status.valueOf().startsWith('not'))
92          cy.get(path).should('not.exist');
93      else {
94          cy.get(path).should(status)
95              .find('img').should('have.attr', 'src', icon).next()
96              .within(($labelName) => {
97                  console.log($labelName.text().trim())
98                  cy.compareTranslation(translation, $labelName.text().trim());
99              });
100     }
101 }
```

Slika 5.36. Funkcija verifyWidget

Osim periodnih filtara, stranica „Učinkovitost prodaje po prodajnom savjetniku“ ima i dodatne filtre koji se provjeravaju na slikama 5.37 i 5.38. Slika 5.37 prikazuje provjeru datuma kod odabira dodatnog filtra „prošla godina“. Filtri usporedbe prošle godine i prošlog tjedna rade na istom principu. Dohvaća se odgovor stranice koji se kasnije proslijedi metodi *verifyDates* koja uspoređuje jesu li datumi ispravno postavljeni. No prije te provjere potrebno je pritisnuti na dodatni filtar, i kao dodatna provjera, provjeriti da se drugi prodajni savjetnik nije omogućio za odabir. Slika 5.38 prikazuje provjeru je li drugi prodajni savjetnik omogućen kod odabira filtra „usporedi s prodavačem“.

```
95          it.only('verify radio button: year', () => {
96              cy.intercept("GET", interceptLink.boards.Board3).as('Board3');
97              cy.get(board3.paths.radio_buttons.compareToPreviousYear).click();
98
99              cy.get(board3.paths.filters.salesperson2).should('be.disabled');
100             cy.waitSpinner();
101             Boards.verifyDates('@Board3',
102                 moment(new Date()).startOf('year').format('DD/MM/YYYY'),
103                 moment(new Date()).format('DD/MM/YYYY'));
104         });
```

Slika 5.37. Test filtra usporedi s prošlom godinom

```
106         it('verify radio button: compare to salesperson2', () => {
107             cy.intercept("GET", interceptLink.boards.Board3).as('Board3');
108             cy.get(board3.paths.radio_buttons.compareToSalesPerson).click();
109             cy.waitSpinner();
110             cy.get(board3.paths.filters.salesperson2).should('not.be.disabled');
111         });
```

Slika 5.38. Test filtra usporedi s prodavačem

Pošto stranica „Učinkovitost prodaje po prodajnom savjetniku“ ne sadrži tablicu, uzima se stranica „Izvješće prodaje po markama“ za kratak prikaz testiranja tablice. Nakon provedbe *verifyWidget* naredbe kao i kod prethodne stranice, provjerava se i tablica kod druge dvije stranice. Slika 5.39 prikazuje provjeru vidljivosti tablice nakon odabira ispravnih filtara i pretrage. Slika 5.40 prikazuje test provjere naziva bitnih oznaka u tablici. Sličan tekst se provodi za ostale oznake unutar tablice.

```
25          Boards.clickPeriod(Board2.translation.period.qtdPeriod);
26          Boards.selectAllInMultiselect(Board2.paths.multiselect.salespersons);
27          Boards.clickOnButton(Board2.paths.buttons.calculate);
28          cy.waitSpinner();
29          cy.get(Board2.paths.table.table).should('be.visible');
```

Slika 5.39. Provjera vidljivosti tablice

```
35          it('Verify table - main labels', () => {
36              allMainLables.forEach((lable) => {
37                  Boards.verifyLabel(Board2.paths.table.lable.main[lable],
38                      Board2.translation.table.lable.main[lable]);
39              });
40          });
```

Slika 5.40. Test glavnih oznaka tablice

Funkcija *verifyLabel*, slika 5.41, provjerava stvarne i očekivane nazive unutar tablice ako se vide u trenutačnom okviru preglednika. Ako se ne vide, onda se putem funkcije *scrollIntoView* okvir preglednika pomakne kako bi se vidjeli elementi.

```
36  export function verifyLabel(path, translation){
37      cy.get(path).scrollIntoView().then((text) => {
38          if(document.getElementsByClassName('overflowing-cell indent') !== undefined) {
39              {
40                  cy.get(path).then(el => {
41                      const actualTranslation = el.get(0).innerText.trim();
42                      cy.compareTranslation(translation, actualTranslation);
43                  });
44              }
45          }
46          else {
47              cy.compareTranslation(translation, text.text());
48          }
49      });
50  }
```

Slika 5.41. Funkcija verifyLabel

5.2.4. Četvrti scenarij – automatizirano (Cypress)

Test na slici 5.42 provjerava vezu između stranica povezanih poveznicom, prema tome predstavlja testiranje integracije komponenti. Linije prije 77 postavljaju filtre, dok linija 77 odabire nasumični element u tablici koji da se pritisne. Odabire se nasumični element kako bi pri svakom pokretanju testa dobili novu pritisnutu osobu ili marku. Ovo je jedan od načina izbjegavanja dugih vremena izvođenja testova gdje bi se prolazilo kroz pritisak na svaku osobu ili marku. Unutar linija 78 i 80 provjerava se naziv poveznice i pritišće se na nju putem *click* metode. Linija 80 pamti vrijednost namještenog filtra. Linije 82 do 84 provjeravaju da novo otvorena stranica ima odgovarajuću poveznicu, naziv *h2* HTML elementa i da su filtri postavljeni na one koji su bili odabrani na prethodnoj stranici putem metode *verifySalesPersonDropdowns* vidljive na slici 5.43. Prikazani isječak kôda vrijedi za spoj između stranica „Izvješće prodaje po markama“ i „Učinkovitost prodaje po prodajnom savjetniku“, a za spoj „Izvješće prodaje po prodajnim savjetnicima“ i „Izvješće prodaje po markama“ se provodi sličan test sa promjenom filtara.

```
70      it('Test salesperson hyperlink', () => {
71          Boards.clickOnMultiselect(Board2.paths.multiselect.salespersons);
72          let arr = new Array();
73          cy.get("div[class='drop-show']").find('label').each(($element) => {
74              arr.push($element.text());
75          }).then(() => {
76              Boards.clickOnMultiselect(Board2.paths.multiselect.salespersons);
77              let randomNumber = Math.floor(Math.random() * 'brandList'.length);
78              Boards.returnHyperlinkName(arr[randomNumber], false).then(value => {
79                  cy.get("a[id='column" + value[0] + "Id']").click({force: true});
80                  let selectedSalesPerson = value[2];
81
82                  cy.url().should('include', link.links.Board3);
83                  cy.verifyDashboardName(Board3.translation.boardName);
84                  Boards.verifySalesPersonDropdowns(Board3.paths.filters.salesperson1, selectedSalesPerson);
85              });
86          });
87      });
```

Slika 5.42. Provjera poveznice između prve dvije stranice

Razlog nasumičnih odabira filtara je izbjegavanje iscrpnog testiranja jer ima puno filtara i njihovih kombinacija. Kako bi se izbjeglo izvođenje svakog od njih, odabiru se nasumični filtri. To znači da se pri svakom pokretanju testa provodi testiranje na drugoj kombinaciji filtara.

```
154 export function verifySalesPersonDropdowns(path, translation) {
155     cy.get(path)
156         .should('be.visible').find('option').should('have.lengthOf.at.least', 1);
157     cy.get(path + " option:selected")
158         .within(($filterName) => {
159             expect(translation).to.be.eq($filterName.text().trim())
160         });
161 }
```

Slika 5.43. Funkcija verifySalesPersonDropdowns

5.3. Automatizirano testiranje web aplikacije u alatu Selenium

Prije početka bilo kojeg testa unutar Seleniuma potrebno je prvo postaviti upravljački program (*eng. driver*) specifični za preglednik koji će se koristiti pri testiranju, u ovom slučaju to je *ChromeDriver*. Postavljanje upravljačkog programa vidljivo je na slikama 5.44 i 5.45. Slika 5.45 prikazuje postavljanje opcija upravljačkog programa. Moguće je postaviti veličinu, vidljivost, anonimnost i slično. Neke od opcija utječu na sam rad upravljačkog programa, kao što je opcija na liniji 108. Putem ove opcije izbjegava se kvar ili pad preglednika pri premaloj particiji /dev/shm za određena okruženja virtualne mašine.

```
ChromeOptions options = setupChromeDriver();
driver = new ChromeDriver(options);
WebDriverWait wait = new WebDriverWait(driver, Duration.ofMillis(5000000L));
```

Slika 5.44. Postavljanje upravljačkog programa

```
105     private ChromeOptions setupChromeDriver() {
106         WebDriverManager.chromedriver().setup();
107         ChromeOptions options = new ChromeOptions();
108         options.addArguments("--disable-dev-shm-usage"); // rj za probleme s ograničenim resursima
109         options.addArguments("--remote-allow-origins=*");
110         //options.addArguments("--headless");
111         options.addArguments("--incognito");
112         options.addArguments("--start-maximized");
113         //options.addArguments("--start-fullscreen");
114         options.addArguments("window-size=1980,960");
115         return options;
116     }
```

Slika 5.45. Postavljanje opcija upravljačkog programa

Nakon što je napravljen upravljački program, njegova instanca se dohvaća putem funkcije *getDriver()* te putem nje upravljamo upravljačkim programom i time i preglednikom. Prije svih napravljenih testova poziva se funkcija *prepareReport* koja se nalazi na slici 5.46. Funkcija *ensureLoggedIn* posjećuje predani URL i prijavljuje se s danim podacima za korisnika. Funkcija *navigateToPage* se vraća na URL određene stranice koju se testira. Slika 5.47 prikazuje metodu *navigate* putem koje se posjećuje URL. Metoda *navigate* se poziva na instancu upravljačkog programa kao prethodno spomenuto. Dodatno, kako se ne bi čekalo dovijeka, potrebno je postaviti vremensko ograničenje čekanja. Ukoliko se stranica ne učita u tom periodu, test se neće dalje nastavljati i javit će se greška korisniku.

```
71      @Override
72      public void prepareReport() throws IOException {
73          ensureLoggedIn();
74          navigateToPage();
75      }
```

Slika 5.46. Funkcija prepareReport

```
getDriver().navigate().to(getBaseUrl());
getDriver().manage().timeouts().pageLoadTimeout(25, TimeUnit.SECONDS);
```

Slika 5.47. Posjećivanje stranice

5.3.1. Prvi scenarij – automatizirano (Selenium)

Testiranje periodnih filtara se obavlja putem testa vidljivog na slici 5.48. Putem funkcije *getPeriodNumber* (slika 5.49) dohvaćaju se svi periodni filtri te testira ima li ih pet, dok funkcija *verifyPeriodFilters* (slika 5.50) uspoređuje svaki naziv periodnog filtra s tekstualnim nazivima koje treba prikazivati.

```
16      @Test
17      public void validatePeriodFilters() throws IOException {
18          boards.getPeriodNumber();
19          boards.verifyPeriodFilters();
20      }
```

Slika 5.48. Testiranje periodnih filtara

Slike 5.49 i 5.50 obje sadrže funkciju *getWait().until(ExpectedConditions...)*. Ova funkcija omogućuje da se testiranje zaustavi dok se ne pronađe element s odgovarajućim uvjetom. Uvjet koji se koristi u obje ove funkcije je da je pronađeni element vidljiv.

```
105     public void getPeriodNumber(){
106         getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.xpath(period_xpath)));
107         List<WebElement> element = getDriver().findElements(By.xpath(period_xpath));
108         Assert.assertEquals(element.size(), 5);
109     }
```

Slika 5.49. Funkcija getPeriodNumber

```
120     public void verifyPeriodFilters(){
121         SoftAssert softAssert = new SoftAssert();
122         getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.xpath(period_xpath)));
123         List<WebElement> elements = getDriver().findElements(By.xpath(period_xpath));
124         for(WebElement element: elements){
125             softAssert.assertEquals(element.getText(), periods.get(elements.indexOf(element)));
126         }
127         softAssert.assertAll();
128     }
```

Slika 5.50. Funkcija verifyPeriodFilters

Provjera birača datuma se odrađuje putem funkcije *verifyDateFilters* na slici 5.51. Funkcija čeka dok se ne vidi prvi birač datuma te onda provjerava jesu li oba vidljiva. Osim vidljivosti samog elementa provjerava i je li vidljiva ikona te ima li zadani oblik i status kao atribut.

```
147     public void verifyDateFilters() {
148         SoftAssert softAssert = new SoftAssert();
149         getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.id(datePickers.get(0))));
150         for(String dateP: datePickers){
151             WebElement datePicker = getDriver().findElement(By.id(dateP));
152             WebElement datePicker_icon = datePicker.findElement(By.tagName("cds-icon"));
153             softAssert.assertTrue(datePicker.isDisplayed());
154             softAssert.assertTrue(datePicker_icon.isDisplayed());
155             softAssert.assertEquals(datePicker_icon.getAttribute("shape"), "calendar");
156             softAssert.assertEquals(datePicker_icon.getAttribute("status"), "info");
157         }
158         softAssert.assertAll();
159     }
```

Slika 5.51. Funkcija verifyDateFilters

Provjera ostalih filtara za prvu stranicu se odrađuje putem testa sa slike 5.52. Funkcija *verifyMultiFilter*, slika 5.53, čeka vidljivost filtara te nakon toga ga dohvaća i provjerava njegovu vidljivost i zadano stanje.

```
26      @Test
27      public void validateFilters() throws IOException {
28          boards.verifyMultiFilter();
29          boards.verifyMultiselectFilter();
30      }
```

Slika 5.52. Provjera filtara

Funkcija *verifyMultiFilter* vrijedi samo za prvu stranicu pošto ona ima samo jedan filtar s mogućnosti odabira svega ili jedne opcije, dok druge dvije stranice imaju dva takva filtra pa se za njih dodaje još jedna provjera za taj drugi filtar unutar funkcije *verifyMultiFilters*.

```
196     public void verifyMultiFilter(){
197         //Board1
198         SoftAssert softAssert = new SoftAssert();
199         getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.id(BoardFilters.get(0))));
200         softAssert.assertTrue(getDriver().findElement(By.id(BoardFilters.get(0))).isDisplayed());
201         Select siteSelect = new Select(getDriver().findElement(By.id(BoardFilters.get(0))));
202         softAssert.assertEquals(siteSelect.getFirstSelectedOption().getText(), "All Sites");
203         softAssert.assertAll();
204     }
```

Slika 5.53. Funkcija verifyMultiFilter

Funkcija *verifyMultiselectFilter* sa slike 5.54 prikazuje provjeru vidljivosti elementa i njegovog potvrdnog okvira putem kojeg se može odraditi brzi odabir više vrijednosti filtra.

```
438     public void verifyMultiselectFilter(){
439         getWait().until(ExpectedConditions.elementToBeClickable(getDriver()
440             .findElement(By.xpath(checkbox_xpath)).findElement(By.xpath(".."))));
441         Assert.assertTrue(getDriver().findElement(By.xpath(checkbox_xpath))
442             .findElement(By.xpath("..")).isDisplayed());
443     }
```

Slika 5.54. Funkcija verifyMultiselectFilter

Funkcija *verifyButton* na slici 5.55 se koristi za testiranje gumba za pretragu. Prvo se čeka da se gumb može pritisnuti, nakon toga se provjerava vidljivost, naziv gumba te da je li on omogućen.

```
206     public void verifyButton(){
207         SoftAssert softAssert = new SoftAssert();
208         getWait().until(ExpectedConditions.elementToBeClickable(By.id(buttonId)));
209         WebElement button = getDriver().findElement(By.id(buttonId));
210         softAssert.assertTrue(button.isDisplayed());
211         softAssert.assertEquals(button.getText(), "CALCULATE");
212         softAssert.assertTrue(button.isEnabled());
213         softAssert.assertAll();
214 
215     }
```

Slika 5.55. Funkcija verifyButton

Funkcija *verifyWidgetsBoards12*, slika 5.56, se koristi pri testiranju stanja vidljivosti widgeta. Stanje koje se utvrđuje je predano kao argument funkcije. Ukoliko trebamo utvrditi njihovu vidljivost, onda se prvo čeka dostupnost prvog widgeta pa se za svaki widget sa stranice provjerava je li vidljiv. No ukoliko se treba utvrditi da ne postoji taj element čeka se bilo koji vidljivi element sa stranice pa se provjerava nevidljivost elemenata. Kada widgeti nisu vidljivi, oni se ne nalaze nigdje u kôdu stranice što znači da ih se ne može pronaći kako bi se utvrdilo da ih nema. Time smo izjavom *try catch* osigurali da ukoliko je element prikazan kada ne treba biti, test pada, a ukoliko se ne prikazuje, onda će grešku za nepostojeći element uhvatiti blok *catch*. Vidljivost tablice ispituje se na sličan način putem metode *validateVisibityOfTable*. Osim vidljivosti widgeta testira se i naziv i slika ikone koji se pojavljuju uz svaki widget. Za provjeru slike koristila se metoda *contains* koja provjerava postoji li ključna riječ unutar izvora slike.

```
241    public void verifyWidgetsBoards12(boolean value){
242        SoftAssert softAssert = new SoftAssert();
243        if(value){
244            getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.id(board12Widgets.get(0))));
245            for(String widget: board12Widgets){
246                WebElement widgetElement = getDriver().findElement(By.id(widget));
247                softAssert.assertTrue(widgetElement.isDisplayed());
248                softAssert.assertEquals(widgetElement.findElement(By.tagName("span"))
249                    .getAttribute("innerHTML").trim(), board12WidgetsNames.get(board12Widgets.indexOf(widget)));
250                softAssert.assertTrue(widgetElement.findElement(By.tagName("img")).getAttribute("src")
251                    .contains(board12WidgetsIcons.get(board12Widgets.indexOf(widget))));
252            }
253        }
254        else {
255            getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.id(datePickers.get(0))));
256            try {
257                for (String widget : board12Widgets) {
258                    softAssert.assertFalse(getDriver().findElement(By.id(widget)).isDisplayed());
259                }
260            } catch (NoSuchElementException exception) {}
261        }
262        softAssert.assertAll();
263    }
```

Slika 5.56. Funkcija verifyWidgetsBoards12

5.3.2. Drugi scenarij – automatizirano (Selenium)

Provjera verifikacijskih poruka za prvu stranicu vidljiva je na slici 5.57. Metoda *setPeriodFiler*, slika 5.58, čeka vidljivost periodnih filtara te odabire predani periodni filtar. Metoda *pressCalculate* radi na istom principu osim što poziva metodu *waitOnPage* nakon što se pritisne gumb. Metoda *waitOnPage* čeka da nestane element za prikaz učitavanja stranice koristeći *getWait().until(ExpectedConditions...)* gdje je uvjet nevidljivost elementa.

```
11      @Test
12      public void NoBrandTeam() throws IOException {
13          boards.setPeriodFilter("YTD");
14          boards.pressCalculate();
15          boards.validateErrorMessage("There is no brand team selected");
16      }
```

Slika 5.57. Provjera verifikacijske poruke pri ne odabiru marke

```
111    public void setPeriodFilter(String period) {
112        getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.xpath(period_xpath)));
113        getDriver().findElement(By.xpath(period_xpath+ "//span[text() = '" + period+ "']")).click();
114    }
```

Slika 5.58. Funkcija setPeriodFilter

Zadnja funkcija koja se poziva na slici 5.57 je funkcija *validateErrorMessage* sa slike 5.59. Ova metoda dohvaća tekst elementa te uklanja moguće prazne znakove kako bi se tekst mogao usporediti s tekstom predanim u argumentu funkcije. Osim teksta se provjeravaju i atributi.

```
65      public void validateErrorMessage(String message){
66          SoftAssert softAssert = new SoftAssert();
67          WebElement errorMessage = getDriver().findElement(By.tagName("clr-control-error"));
68          WebElement errorIcon = errorMessage.findElement(By.tagName("    icon"));
69          String error = errorMessage.getText().trim();
70          softAssert.assertEquals(error, message);
71          softAssert.assertEquals(errorIcon.getAttribute("shape"), "exclamation-circle");
72          softAssert.assertEquals(errorIcon.getAttribute("status"), "danger");
73          softAssert.assertAll();
74      }
```

Slika 5.59. Funkcija validateErrorMessage

Drugi tip provjere je vidljiv na slici 5.60, a to je provjera je li se verifikacijska poruka uklonila nakon što je korisnik odabrao opciju filtra zbog verifikacijske poruke. Funkcija *setMultiselectFilter* odabire sve opcije u obveznom više odabirskom filtru.

```
18      @Test
19      public void SelectedBrandTeam() throws IOException {
20          boards.setPeriodFilter("YTD");
21          boards.pressCalculate();
22          boards.validateErrorMessage("There is no brand team selected");
23          boards.setMultiselectFilter();
24          boards.validateNoError();
25      }
```

Slika 5.60. Provjera uklanjanja verifikacijske poruke

Funkcija *validateNoErrorMessage* sa slike 5.61 je još jedan od načina kako bi se provjerilo nepostojanje elemenata. Ključno je da se dohvaća više elementa i da se provjerava je li to polje elemenata prazno. Ukoliko je prazno, verifikacijska poruka se uklonila, ukoliko nije, postoji greška.

```
178     public void validateNoErrorMessage(){
179         Assert.assertTrue(getDriver().findElements(By.tagName("clr-control-error")).isEmpty());
180     }
```

Slika 5.61. Funkcija validateNoError

Još jedan primjer verifikacijskih poruka je sa stranice „Učinkovitost prodaje po prodajnom savjetniku“ kada se pri odabiru filtra usporedbe s prodajnim prodavačem ne odabere drugi prodavač. Test je vidljiv na slici 5.62. Potrebno je postaviti periodni filtar putem metode *setPeriodFilter* te zatim odabrati filtar usporedbe s prodajnim prodavačem putem funkcije *setRadioFilter* koja dohvaća filtar putem predanog argumenta i pritišće na njega. Metoda *setSelectFilter* koja je vidljiva na slici 5.63 čeka da se odabrani element može pritisnuti te nakon toga ga se dohvaća i odabire se vrijednost prema onoj iz argumenta funkcije. Kada su filtri

namješteni, putem funkcije *pressCalculate* se odrađuje pritisak na gumb pretraga pa je onda samo potrebno provjeriti je li se pokazao točan verifikacijski tekst.

```
19      @Test
20      public void NoSalesPerson2() throws IOException {
21          boards.setPeriodFilter("YTD");
22          boards.setRadioFilter(4);
23          boards.setSelectFilter("sp1", 1);
24          boards.pressCalculate();
25          boards.validateErrorMessage("Sales Person 2 is not selected");
26      }
27  }
```

Slika 5.62. Provjera verifikacijske poruke pri ne odabiru drugog prodajnog savjetnika

```
405     public void setSelectFilter(String elementId, int value) {
406         getWait().until(ExpectedConditions.elementToBeClickable(By.id(elementId)));
407         Select select = new Select(getDriver().findElement(By.id(elementId)));
408         waitOnPage();
409         select.selectByIndex(value);
410     }
```

Slika 5.63. Funkcija setSelectFilter

5.3.3. Treći scenarij – automatizirano (Selenium)

Kao i prije, treći scenarij se obrađuje na stranici „Učinkovitost prodaje po prodajnom savjetniku“ zbog većeg broja filtara. Prvi filtri koji se testiraju su periodni filtri. Slika 5.64 prikazuje primjer testiranja periodnog filtra kada korisnik pritisne na „QTD“ što se obavlja putem funkcije *setPeriodDate*. Funkcija *getDate*, slika 5.65, uspoređuje datum početnog birača datuma sa predanim datumom. Putem klase *LocalDate* i metode *now* moguće je dohvatiti trenutačni datum sistemskog sata u zadanoj vremenskoj zoni. Pomoću klase *LocalDate* moguće je dohvatiti broj mjeseca datuma te njegov odgovarajući kvartil što pomaže pri testiranju „QTD“ filtra. Osim mjeseca moguće je dohvatiti i dan u tjednu koji odgovara nazivu npr. kao na slici 5.66 kod testiranja prvog dana u tjednu koristeći „*DayOfWeek.MONDAY*“.

```
31      @Test
32      public void testPeriod_QUARTER() throws IOException {
33          boards.setPeriodFilter("QTD");
34          boards.getDate("startDate", LocalDate.now().with(LocalDate.now().getMonth().firstMonthOfQuarter()).withDayOfMonth(1));
35      }
```

Slika 5.64. Testiranje periodnog filtra „QTD“

Vrijednosti birača datuma su vidljive na stranici, ali ih nije moguće dohvatiti putem metoda klase *Select*. U Cypressu se čitao odgovor zahtjeva sa poslužiteljske strane kako bi se dobili datumi

navedeni u biraču datuma. U Seleniumu, radi jednostavnosti, se uzimaju pojedinačno elementi iz komponente *Calendar* putem metode *getDate*. Ta funkcija radi na taj način da prvo dohvati onemogućeni element početnog birača datuma te ga omogući. Onemogućenom elementu se uklanja atribut *disabled* putem sučelja *JavascriptExecutor* koje omogućuje izvršavanje JavaScript naredbi putem Selenium WebDriver-a metodom *executescript*. Nakon toga klikom na ikonu za kalendar se otvara kalendar preko kojega se može dohvatiti vrijednost odabranog datuma. Drugo rješenje ovog problema dohvaćanja vrijednosti birača datuma bi zahtijevalo ovisnost o *Rest Assured* API-ju.

```
665     public void getDate(String path, LocalDate expected) {
666         DateTimeFormatter parser = DateTimeFormatter.ofPattern("MMM").withLocale(Locale.ENGLISH);
667         getWait().until(ExpectedConditions.visibilityOfElementLocated(By.tagName(datePickers_xpath)));
668         WebElement clicky = getDriver().findElement(By.xpath("//"+datePickers_xpath+"[@id = '"+ path +"']//button[@id = 'startDateButton']"));
669         JavascriptExecutor js = (JavascriptExecutor) getDriver();
670         js.executeScript("arguments[0].removeAttribute('disabled', 'disabled')", clicky);
671         WebDriverWait wait = new WebDriverWait(getDriver(), Duration.ofSeconds(20));
672         wait.until(ExpectedConditions.elementToBeClickable(clicky));
673         clicky.click();
674         WebElement datePicker = getDriver().findElement(By.tagName("clr-daypicker"));
675         String header = getDriver().findElement(By.className("calendar-pickers")).getText();
676         String[] list = header.split("\n");
677         String month = list[0];
678         TemporalAccessor accessor = parser.parse(list[0]);
679         int year = Integer.valueOf(list[1]);
680         int day = Integer.valueOf(datePicker.findElement(By.xpath("//button[@aria-label = 'clarity.datepickerSelectedLabel']")).getText());
681         getDriver().findElement(By.xpath("//"+datePickers_xpath+"[@id = '"+datePickers.get(0)+"']//button[@id = 'startDateButton']")).click();
682         LocalDate date = LocalDate.of(year, accessor.get(ChronoField.MONTH_OF_YEAR), day);
683         Assert.assertEquals(date, expected);
684         System.out.println(formatDate(date));
685     }
```

Slika 5.65. Funkcija getDate

```
21      @Test
22      public void testPeriod_WEEK() throws IOException {
23          boards.setPeriodFilter("WTD");
24          boards.getDate("startDate", LocalDate.now().with(DayOfWeek.MONDAY));
25      }
```

Slika 5.66. Testiranje periodnog filtra „WTD“

Slika 5.67 prikazuje testiranje prelaskom miša na periodne filtre. Svi periodni filtri osim „DATE“ filtra zahtijevaju tekst koji se pojavljuje kada se pređe preko njih s mišem. Stoga postoje dvije funkcije, *verifyNoHover* i *verifyHover*, slika 5.68.

```
45      @Test
46      public void validateHover() throws IOException {
47          boards.verifyNoHover("DATE");
48          boards.verifyHover("WTD", "week to date");
49          boards.verifyHover("MTD", "month to date");
50          boards.verifyHover("QTD", "quarter to date");
51          boards.verifyHover("YTD", "year to date");
52      }
```

Slika 5.67. Testiranje prelaskom miša na periodne filtre

Funkcija *verifyNoHover* provjerava da tekst prilikom prelaska miša nije vidljiv, dok funkcija *verifyHover* provjerava da je tekst vidljiv i jednak predanom tekstu. Klasa *Actions* omogućuje izvršavanje složenih korisničkih gesta umjesto izravnog korištenja tipkovnice ili miša. Putem te klase moguće je napraviti prelazak miša preko filtra uz metodu *moveToElement* koja pomiče miš na sredinu elementa. Metoda *perform* služi za izvođenje zadanih radnji.

```
356     public void verifyHover(String period, String label){
357         Actions act = new Actions(getDriver());
358         getWait().until(ExpectedConditions.visibilityOfElementLocated(By.xpath(period_xpath+"//span[text() = '" + period+ "']")));
359         WebElement element = getDriver().findElement(By.xpath(period_xpath+"//span[text() = '" + period+ "']"));
360         act.moveToElement(element).perform();
361         Assert.assertEquals(getDriver().findElement(By.id("tooltip-id")).getAttribute("innerHTML"), label);
362     }
```

Slika 5.68. Funkcija verifyHover

Na slici 5.69 se nalazi testiranje prikaza widgeta nakon pritiska gumba „Pretraga“. Putem metoda *setPeriodFilter* i *setSelectFilter* se postavljaju svi potrebni filtri te se putem metode *pressCalculate* izvodi pritisak na gumb. Metoda *verifyWidgets* je metoda za treću stranicu koja funkcionira na istom principu kao i prethodno objašnjena metoda *verifyWidgetsBoards12* za prve dvije stranice. Razlika je u podacima koji se koriste. Provjerava widgete, nazive i slike prikazanih ikona. Za razliku od testa na slici 5.70, ovaj test se više fokusira da se klikom na gumb prikazuju widgeti, dok se na slici 5.70 nalazi funkcija koja provjerava prikaz widgeta za svaki periodni filtar.

```
12      @Test
13      public void validateCalculateButtonWidgets() throws IOException {
14          boards.setPeriodFilter("YTD");
15          boards.setSelectFilter("salesPerson1Id", 1);
16          boards.pressCalculate();
17          boards.verifyWidgets(true);
18      }
```

Slika 5.69. Testiranje prikaza widgeta nakon pritiska gumba „Pretraga“

```
20      @Test
21      public void validatePeriodWidgets() throws IOException {
22          boards.validateAllPeriodWidgetsBoard3();
23      }
```

Slika 5.70. Testiranje widgeta za svaki periodni filtar

S obzirom na to da prve dvije stranice ne prikazuju widgete pri periodnim filtrima „DATE“ i „WTD“, funkcija validate *AllPeriodWidgetsBoard3*, slika 5.71, ne odgovara za te dvije stranice. Stoga postoji metoda *validateAllPeriodWidgets* koja za ta dva periodna filtra provjerava neprikazivanje widgeta. Obje metode prvo postavljaju periodne filtre pa izvode pritisak na gumb

za pretragu. Nakon toga pozivaju prethodno objašnjenu metodu *verifyWidgets* za treću stranicu ili *verifyWidgetsBoards12* u slučaju prve i druge stranice.

```
284     public void validateAllPeriodWidgetsBoard3(){
285         //valid for Board3
286         for (String period : periods) {
287             if(period.equals("DATE")){
288                 setPeriodFilter(period);
289                 setFilterDate(LocalDate.now().minusDays(3));
290             }
291             else
292                 setPeriodFilter(period);
293             setSelectFilter(BoardFilters.get(2), 1);
294             getDriver().findElement(By.id(buttonId)).click();
295             waitOnPage();
296             verifyWidgets(true);
297         }
298     }
```

Slika 5.71. Funkcija validateAllPeriodWidgetsBoard3

Sljedeće je potrebno testirati dodatne filtre za usporedbu kao što su usporedba s prošlom godinom sa slike 5.72 i usporedi s prodavačem na slici 5.73. Nakon pritiska na filtar „usporedi s prošlom godinom“ putem metode *getDate* se provjerava jesu li se pravilno promjenili datumi. Provjera promjene periodnog filtra se provodi naredbom *validatePeriodFilter* na način da se pronađe klasa *activePeriod* i provjeri odgovara li naziv aktivnog periodnog filtra onom predanom u argumentu.

```
28      public void testPeriod_R_py() throws IOException {
29          boards.setRadioFilter(3);
30          boards.getDate("startDate", LocalDate.now().minusDays(1).withDayOfYear(1));
31          boards.validatePeriodFilter("YTD");
32          boards.setSelectFilter("sp1", 1);
33          boards.pressCalculate();
34          boards.verifyWidgets(true);
35      }
```

Slika 5.72. Test filtra usporedi s prošlom godinom

```
190     public void validatePeriodFilter(String period) {
191         getWait().until(ExpectedConditions.visibilityOfAllElementsLocatedBy(By.xpath(period_xpath)));
192         Assert.assertEquals(getDriver().findElement(By.className("activePeriod")).findElement(By.tagName("span")).getText(), period);
193     }
```

Slika 5.73. Funkcija validatePeriodFilter

Slika 5.74 prikazuje provjeru filtra usporedi s prodavačem. Nakon odabira filtra potrebno je provjeriti da se odabir drugog prodajnog savjetnika omogućio putem funkcije *verifyEnabledSP2*. Funkcija *verifyEnabledSP2* testira omogućenost putem metode *isEnabled* na dohvaćeni element. Odabirom opcija filtra za oba prodajna savjetnika i obavljanjem pretrage provjerava se jesu li se widgeti pokazali.

```
36      @Test
37      public void testPeriod_R_sp2() throws IOException {
38          boards.setRadioFilter(4);
39          boards.verifyEnabledSP2();
40          boards.setSelectFilter("sp1", 1);
41          boards.setSelectFilter("sp2", 1);
42          boards.pressCalculate();
43          boards.verifyWidgets(true);
44      }
```

Slika 5.74 Test filtra usporedi s prodavačem

Kao prije spomenuto, treća stranica ne sadrži tablicu tako da se provjera njene funkcionalnosti odrađuje na prve dvije stranice putem funkcije *verifyTableLook*. Funkcija *verifyTableLook* sa slike 5.75 prolazi kroz svaki periodni filtar i postavlja ostale filtre kako bi se odradila pretraga. Nakon toga, potrebno je pričekati da se stranica učita pa odraditi provjeru redova i stupaca tablice. Stupci tablice se provjeravaju imaju li veličinu veću od jedan jer njihov broj ovisi o broju filtara koje je korisnik odabrao (npr. broj marki), dok se redovi provjeravaju u funkciji *verifyTableRows* sa slike 5.76.

```
309     public void verifyTableLook(){
310         SoftAssert softAssert = new SoftAssert();
311         for (String period : periods) {
312             if(period.equals("DATE")){
313                 setPeriodFilter(period);
314                 setFilterDate(LocalDate.now().minusDays(3));
315             }
316             else
317                 setPeriodFilter(period);
318             setMultiselectFilter();
319             getDriver().findElement(By.id(buttonId)).click();
320             waitOnPage();
321             verifyTableRows();
322             List<WebElement> columns = getDriver().findElements(By.xpath("//*[contains(@id,'column')]"));
323             softAssert.assertTrue(columns.size()>0);
324         }
325         softAssert.assertAll();
326     }
```

Slika 5.75. Funkcija verifyTableLook

Funkcija *verifyTableRows* dohvaća sve redove, provjerava njihovu vidljivost te uspoređuje pravilanost prikazanog teksta.

```
346     public void verifyTableRows(){
347         SoftAssert softAssert = new SoftAssert();
348         getWait().until(ExpectedConditions.visibilityOfElementLocated(By.id(tableRows.get(0))));
349         int i=-1;
350         for(String rows: tableRows){
351             i++;
352             softAssert.assertTrue(getDriver().findElement(By.id(rows)).isDisplayed());
353             softAssert.assertEquals(getDriver().findElement(By.id(rows)).getText().trim(), tableRowsNames.get(i));
354         }
355         softAssert.assertAll();
356     }
```

Slika 5.76. Funkcija verifyTableRows

5.3.4. Četvri scenarij – automatizirano (Selenium)

Testiranje poveznica unutar tablice se obavlja u testu na slici 5.77. Prvo je potrebno postaviti filtre i obaviti pretragu. Nakon toga putem *validateVisibilityOfTable* se utvrđuje vidljivost tablice. Nakon toga se poziva funkcija *clickOnTableLink* koja vraća kliknuti element za daljnju usporedbu. Funkcija *verifyDashboardName* provjerava naziv stranice kako bi se utvrdilo (osim putem URL-a) da je otvorena ispravna stranica. Naziv se provjerava putem *h2* HTML elementa. Putem funkcije *checkBrandFilter* (za drugu stranicu) ili *checkFiltersBoard3* (za treću stranicu) se provjeravaju filtri koji moraju biti automatski zadani. Marka koja se odabere u tablici na prvoj stranici mora se prikazati kao zadana vrijednost filtra za marke. Odabirom na prodajnog savjetnika u tablici na drugoj stranici se otvara treću stranicu koja ima zadane vrijednosti odabrane sa prve (marka) i druge (prodajni savjetnik) stranice.

```
16      @Test
17      public void validateHyperlink() throws IOException {
18          boards.setPeriodFilter("YTD");
19          boards.setMultiselectFilter();
20          boards.pressCalculate();
21          boards.validateVisibityOfTable(true);
22          String brand = boards.clickOnTableLink();
23          boards.verifyDashboardName("Board 2");
24          boards.checkBrandFilter(brand);
25          boards.setMultiselectFilter();
26          boards.pressCalculate();
27          boards.validateVisibityOfTable(true);
28          String salesPerson = boards.clickOnTableLink();
29          boards.verifyDashboardName("Board 3");
30          boards.checkFiltersBoard3(brand, salesPerson);
31      }
32  }
```

Slika 5.77. Testiranje poveznice između stranica

Funkcija *clickOnTableLink*, slika 5.78, dohvaća sve stupce te uzima nasumični broj unutar domene broja stupaca. Nakon toga, potrebno je uzeti vrijednost stupca na čiju će se poveznicu pritisnuti te se onda izvodi pritisak na poveznicu. Ta vrijednost se vraća za testiranje u funkciji *checkBrandFilter*. Razlog nasumičnog broja je opet kao i prethodno u Cypressu, izbjegavanje iscrpnog testiranja.

```
266     public String clickOnTableLink(){
267         Random rand = new Random();
268         List<WebElement> columns = getDriver().findElements(By.xpath("//*[contains(@id,'column')]"));
269         int rand_link = rand.nextInt(columns.size());
270         String name = columns.get(rand_link).getText();
271         columns.get(rand_link).click();
272         return name;
273     }
```

Slika 5.78. Funkcija clickOnTableLink

Funkcija *checkBrandFilter* provjerava odgovara li marka dana u argumentu onoj koja je vidljiva na stranici. Funkcija *checkFiltersBoard3* sa slike 5.79 provjerava i marku i prodajnog savjetnika.

```
307     public void checkFiltersBoard3(String brand, String sp){
308         SoftAssert softAssert = new SoftAssert();
309         getWait().until(ExpectedConditions.visibilityOfElementLocated(By.id(BoardFilters.get(1))));
310         Select selectBrand = new Select(getDriver().findElement(By.id(BoardFilters.get(1))));
311         Select selectSP = new Select(getDriver().findElement(By.id(BoardFilters.get(2))));
312         waitOnPage();
313         softAssert.assertEquals(selectBrand.getFirstSelectedOption().getText(), brand);
314         softAssert.assertEquals(selectSP.getFirstSelectedOption().getText(), sp);
315     }
```

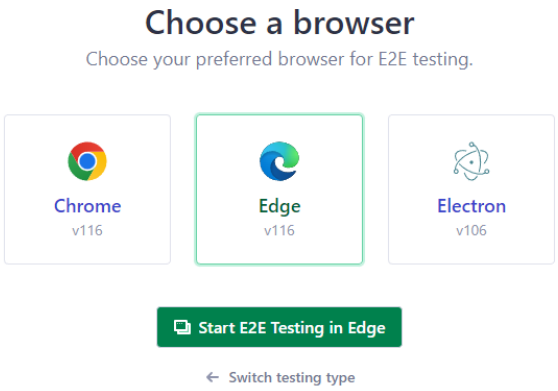
Slika 5.79. Funkcija checkFiltersBoard3

5.4. Nefunkcionalno testiranje

Nefunkcionalno testiranje samo po sebi većinski je nemoguće odraditi ručno, ali postoje neka koja se mogu. Jedno od takvih testiranja su upotrebljivost aplikacije i testiranje kompatibilnosti s više Internet preglednika. Upotrebljivost aplikacije odnosi se na lakoću učenja i korištenja aplikacije i unosa ulaza. Korisnik mora lagano razumijeti kako koristiti aplikaciju bez dodatnih uputa. Ovo testiranje se izvodi tako da se tijekom testiranja ostalih funkcionalnosti mora obazirati na njihov izgled i objašnjenje. Na primjer, poveznice u tablici moraju biti druge boje kako bi naveli korisnika da uopće klikne na njih. Verifikacijske poruke pri pretraživanju pomažu korisniku da razumije ono što mora učiniti kako bi uspješno odradio pretragu. Nije od pomoći ako se korisniku samo ispiše „Neispravni podaci“ bez da se objasne koji podaci su neispravni.

S obzirom na jednostavnost aplikacije, nefunkcionalno automatizirano testiranje može se izvesti samo u gledanju vremena izvedbe pretraživanja i dohvaćanja podataka. Prihvatljivo vrijeme predlaže klijent te se aplikacija toga drži. To znači da svi funkcionalni testovi imaju vremenski limit u kojem se moraju izvršiti te ako neki od njih prekrši to mora se znati zašto. Ako je riječ o dugom vremenskom dohvaćanju onda aplikacija ne ispunjava uvjete koje je klijent zadao. Osim vremena izvođenja dodatno može izvesti i testiranje kompatibilnosti s više Internet preglednika. Kod Cypressa je potrebno samo pritisnuti na odgovarajući preglednik prije pokretanja testova (slika 5.80), ili ako se pokreće u terminalu IntelliJ aplikacije, onda je potrebno samo odrediti koji preglednik se koristi pri pokretanju testova. Cypress ne omogućuje istovremeno testiranje u različitim preglednicima, dok Selenium to omogućuje. Selenium ima opciju za odabir željenih preglednika na kojem će se istovremeno vrtiti odabrani testovi. Cypress omogućuje lakšu izmjenu između željenih preglednika na kojima će se provoditi testovi i nije potrebno izmijeniti kôd kako

bi se izmijenio preglednik. Selenium ipak zahtijeva promjene kôda kako bi se odabrali željeni preglednici.



Slika 5.80. Prikaz odabira Internet preglednika u Cypress sučelju

5.5. Izvješća automatiziranog testiranja

Oba alata za automatizirano testiranje imaju mogućnosti izrade izvješća rezultata testova u XML zapisu. Selenium provodi izvješća kroz TestNG, dok ih Cypress provodi kroz Mocha Junit Reporter, kao što je prethodno spomenuto. Oba izvješća prikazuju broj padova testova tj. grešaka, ukupno vrijeme izvedbe testova, pojedinačno vrijeme izvedbe testova te nazive testova i gdje se nalaze. Slika 5.81 prikazuje izvješće izrađeno putem Cypressa, a slika 5.82 prikazuje izvješće izrađeno putem Seleniuma. Oba izvješća su ispisi rezultata testova za periodne filtre.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="Mocha Tests" time="44.143" tests="5" failures="0">
  <testsuite name="Root Suite" timestamp="2023-09-11T20:32:26" tests="0" file="cypress\e2e\boards\Board1\functionality\dateBehaviour.cy.ts" time="0.000" failures="0">
  </testsuite>
  <testsuite name="Board1" timestamp="2023-09-11T20:32:26" tests="0" time="0.000" failures="0">
  </testsuite>
  <testsuite name="Date behaviour" timestamp="2023-09-11T20:33:01" tests="5" time="8.637" failures="0">
    <testcase name="Board1 Date behaviour verify Date Period" time="34.470" classname="verify Date Period">
    </testcase>
    <testcase name="Board1 Date behaviour verify WTD Period" time="1.583" classname="verify WTD Period">
    </testcase>
    <testcase name="Board1 Date behaviour verify MTD Period" time="1.319" classname="verify MTD Period">
    </testcase>
    <testcase name="Board1 Date behaviour verify QTD Period" time="1.600" classname="verify QTD Period">
    </testcase>
    <testcase name="Board1 Date behaviour verify YTD Period" time="1.525" classname="verify YTD Period">
    </testcase>
  </testsuite>
</testsuites>
```

Slika 5.81. Izvješće automatiziranih testova za periodne filtre - Cypress

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by org.testng.reporters.JUnitReportReporter -->
<testsuite hostname="Andreja-Laptop" failures="0" tests="5" name="controller.tests.Board1.functionality.DateBehaviourTest" time="81.998" errors="0" timestamp="2023-09-11T22:35:51 CEST" skipped="0">
  <testcase classname="controller.tests.Board1.functionality.DateBehaviourTest" name="testPeriod_DATE" time="22.791"/>
  <testcase classname="controller.tests.Board1.functionality.DateBehaviourTest" name="testPeriod_YEAR" time="12.841"/>
  <testcase classname="controller.tests.Board1.functionality.DateBehaviourTest" name="testPeriod_MONTH" time="21.254"/>
  <testcase classname="controller.tests.Board1.functionality.DateBehaviourTest" name="testPeriod_WEEK" time="12.292"/>
  <testcase classname="controller.tests.Board1.functionality.DateBehaviourTest" name="testPeriod_QUARTER" time="12.820"/>
</testsuite> <!-- controller.tests.Board1.functionality.DateBehaviourTest -->
```

Slika 5.82. Izvješće automatiziranih testova za periodne filtre - Selenium

6. USPOREDBA, ANALIZA I PRIMJENA REZULTATA TESTIRANJA

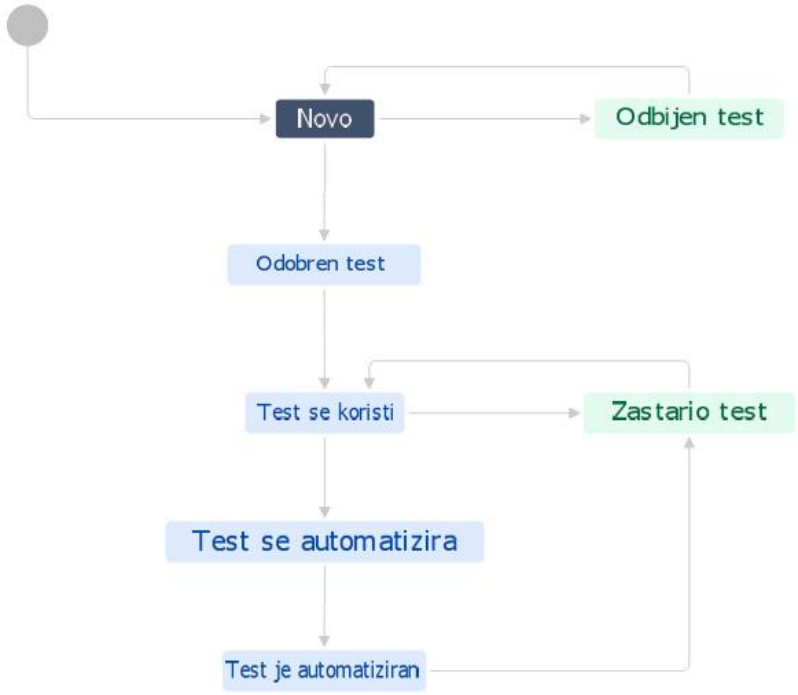
U ovom poglavlju se analiziraju rezultati i prikazuju se načini na koje se primjenjuju rezultati testiranja. Ovo poglavlje uspoređuje tipove testiranja i prikazuje rezultate izvedbe alata Cypress i Selenium.

6.1. Ručni i automatizirani testovi

Slika 6.1 prikazuje faze razvoja web aplikacije tvrtke rađene u *Sprintovima*. *Sprint* predstavlja kratko, vremenski ograničeno razdoblje od dva tjedna u kojem tvrtka radi na dovršavanju zadane količine posla. Na kraju svakog *Sprinta* nastaje potpuno funkcionalan i potencijalno isporučiv proizvod ili komponenta proizvoda. Ideja *Sprinta* je osiguranje brze i česte povratne informacije klijenta. Faze u kojima se provodi testiranje su faza „Razvoj i testiranje“ i „Testiranje“. Faza razvoja i testiranja predstavlja fazu u kojoj se provodi ručno testiranje te se pišu i implementiraju automatizacijskih testovi za regresijsko testiranje. Faza testiranja predstavlja fazu u kojoj se provodi automatizirano regresijsko testiranje. U ovoj fazi nema ručnog testiranja. U fazama „Razvoj i testiranje“ i „Testiranje“ provode se verifikacije, dok se u fazi „Proizvodnja“ provodi validacija. Validaciju provodi sam klijent i u slučaju da se prilikom proizvodnje pronađe greška, smatra se nedostatkom ili *defektom*. Greške vezane uz zahtjeve su rijetke u fazi testiranja, a poželjno je da su nedostaci u fazi proizvodnje nepostojeći.

Slika 6.1. Faze razvoja web aplikacije - Sprint tvrtke

Slika 6.2 prikazuje životni vijek testa. Nakon što je test odobren, on se koristi dok god ne zastari. Tijekom korištenja ručnog testa počinje i proces automatizacije. Nakon što je test automatiziran, ručni test se više ne koristi.



Slika 6.2. Životni vijek testa

Ručni testovi se izvode sporije, pogotovo u slučaju gdje se mora prolaziti kroz sve moguće kombinacije filtara pri pregledavanju widgeta i/ili tablice. U trećem scenariju pokazalo se da je lakše automatizirati nego prolaziti kroz sve funkcionalnosti kad god se dogodi promjena na aplikaciji. Najveća razlika između obje metode testiranja je regresijsko testiranje. Automatizacijom je jednostavno pokrenuti sve zapisane testove u bilo kojem trenutku dok je kod ručnog testiranja nužno da se tester ponovno upozna s testovima. Zapisane korake morat će pratiti i provoditi svaki od njih ponovno. Tablica 6.1 prikazuje vrijeme potrebno za izvođenje regresijskih testova tijekom razvoja web aplikacije tvrtke. Potrebno je ručno odabrati je li test prošao ili ne, dok bi se isto automatski povuklo automatiziranim testiranjem putem XML datoteke koja sadrži rezultate izvođenja testova. Ta XML datoteka se može podići na Internet usluge kao što je JIRA. Dizanjem te datoteke na JIRA stranicu brže se prikazuju rezultati nego pri ručnom uvođenju rezultata.

Tablica 6.1. Brzina izvođenja testova po alatu za automatizaciju

	Ručno [h]	**Automatizirano (Cypress) [h]**
Sprint	4	0,4
Mjesec	8	0,6
6 mjeseci	48	3,6
Godina	96	7,2

6.2. Selenium i Cypress automatizirani testovi

Tablica 6.2 prikazuje vrijeme izvođenja odabranih testova unutar Cypress i Selenium programa. Vrijeme je izraženo u milisekundama [ms]. Vidljivo je iz tablice da Cypress brže izvodi testove. Uočeno je da jedino prvi Cypress test traje najduže zbog postavljanja sesije i ostalih komponenti, dok se svaki idući test provodi brže. Isključujući prvi testi, svi testovi su znatno brže izvedeni nego pojedini testovi unutar Seleniuma. To se može vidjeti na izvješćima prikazanim na slikama 5.81 i 5.82 gdje su svi Selenium testovi iznad 12 ms, dok je kod Cypress testova prvi iznad 34 ms, a svi ostali ispod 2 ms.

Tablica 6.2. Brzina izvođenja testova po alatu za automatizaciju

	Cypress [ms]	**Selenium [ms]**
Stranica 1 - Smoke test	48.6	56.364
Stranica 1 – Verifikacijske poruke	25.181	34.653
Stranica 1 – periodni filtri	44.143	81.998
Stranica 3 – widgeti	43.485	46.299

Tablica 6.3 prikazuje broj linija kôda potrebnih za izvođenje istih testova navedenih u tablici 6.2. Prema tablici 6.3 vidljivo je da su Cypress testovi zahtijevali manje linija kôda za rješavanje problema.

Tablica 6.3. Broj linija testova po alatu za automatizaciju

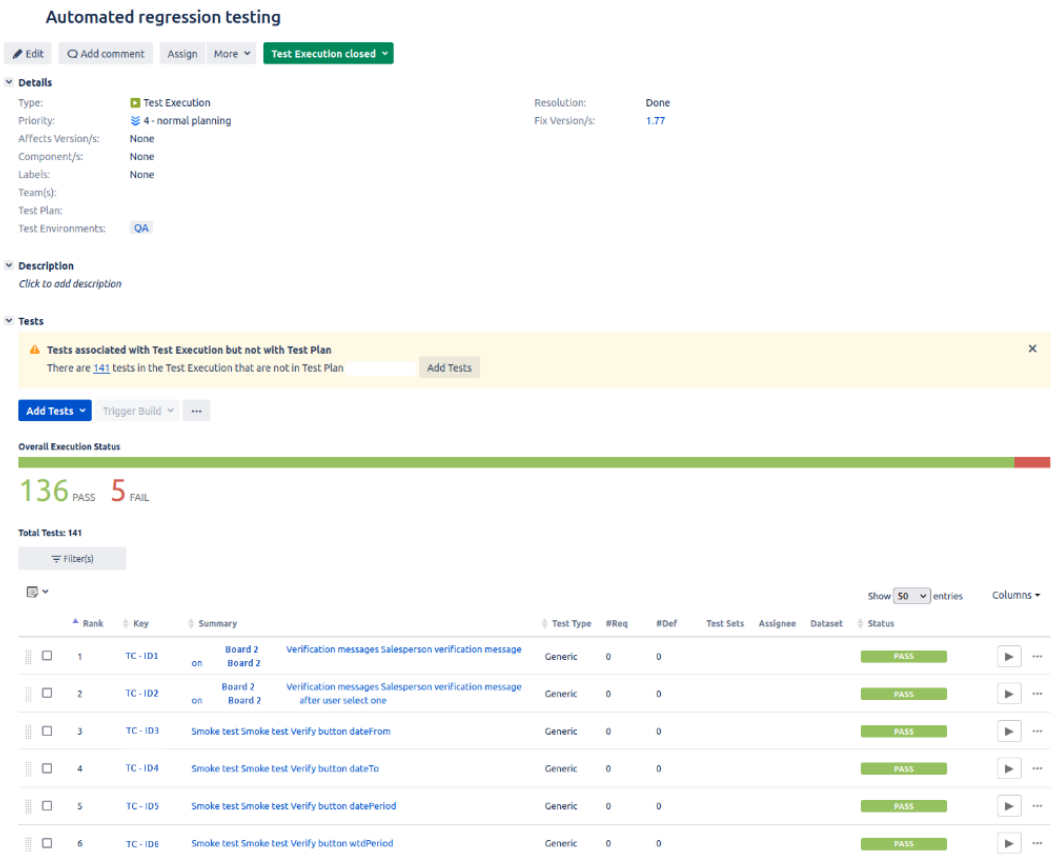
	Cypress	**Selenium**
Stranica 1 - Smoke test	72	89
Stranica 1 – Verifikacijske poruke	31	64
Stranica 1 – periodni filtri	64	117
Stranica 3 – widgeti	41	68

Za nefunkcionalno testiranje kompatibilnosti s raznim preglednicima Selenium prednjači svojom opcijom istovremenog testiranja u različitim preglednicima.

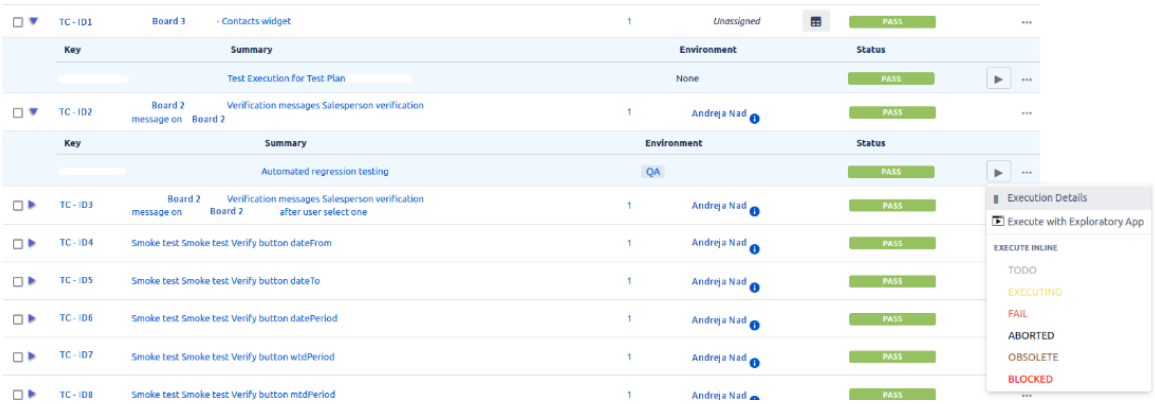
6.3. Primjena rezultata testiranja

Tijekom izvođenja testiranja pravila su se izvješća čiji bi se rezultati postavili na stranicu. Rezultati testiranja su se primjenjivali tijekom razvoja stranice na način da su svi testovi imali svoje nazive, rezultate i identifikacijske oznake, kao prikazano na slici 6.3. Slika 6.4 prikazuje druge opcije osim prolaska testa, a to su: napravit će se, u izvršavanju, pao, prekinut, zastario i blokiran. Ovi statusi se automatski stavljaju u odnosu na izvješća, no prilikom ručnog testiranja, oni se postavljaju ručno. Prilikom testiranja prvobitni testovi su bili ručni jer automatizacija, iako smanjuje vrijeme

izvedbe testova, zahtijeva vrijeme da se napišu svi potrebni testovi. Osim vremena, ponekad se nemaju sve informacije za izradu testova. Primjerice, prilikom izrade testova neki od elemenata nisu imali klase ili identifikacijske oznake. U takvom slučaju elementi se mogu dohvaćati putem jedinstvenih HTML oznaka, no u slučaju i njihova izostanka, dohvaćanje elementa postaje otežano. Zbog takvih incidenata se prvo odradilo korak-po-korak ručno testiranje. Ručno su se napisali testovi na stranici i u trenutku kada su bili zapisani, izvršili su se i postavili na odgovarajući status.

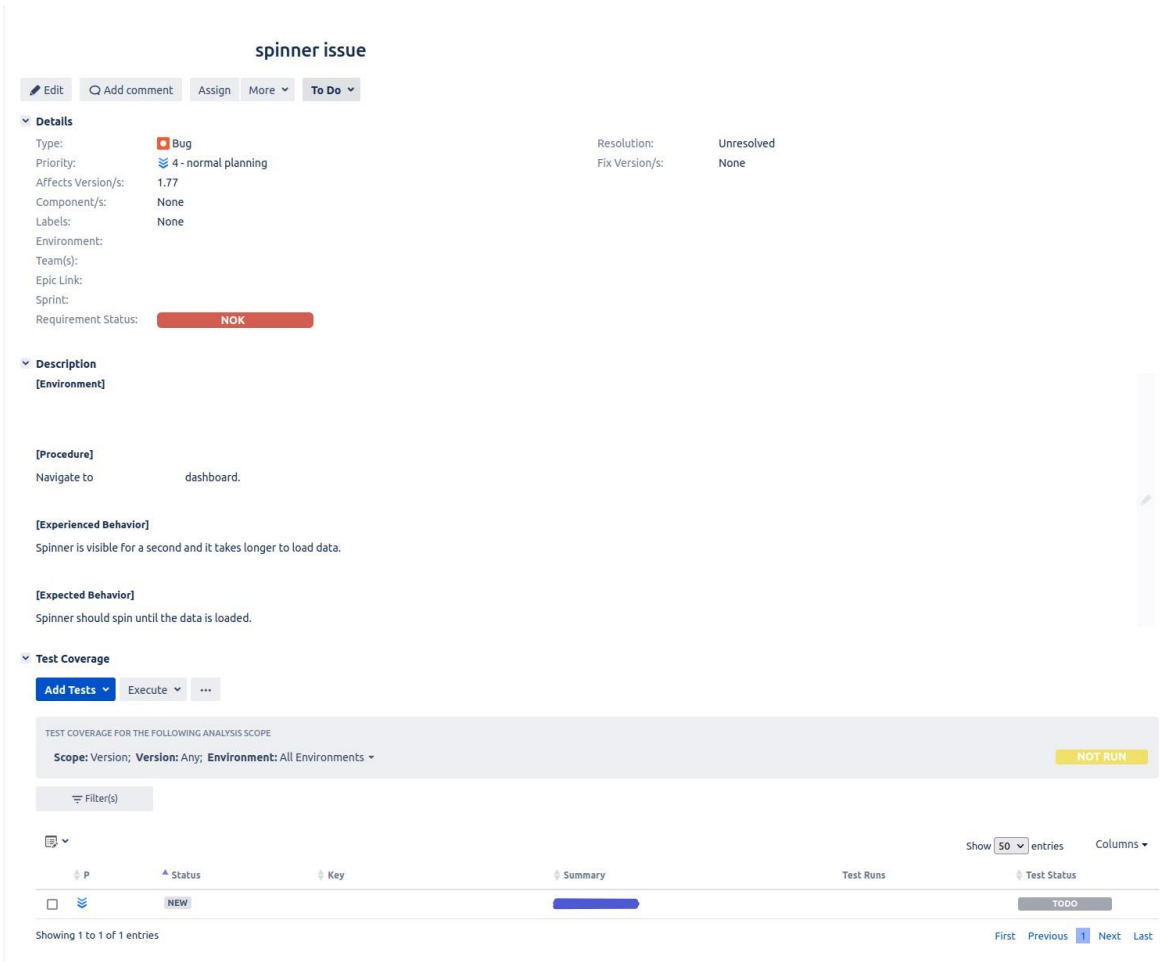


Slika 6.3. Prikaz automatiziranih testova u stranici JIRA



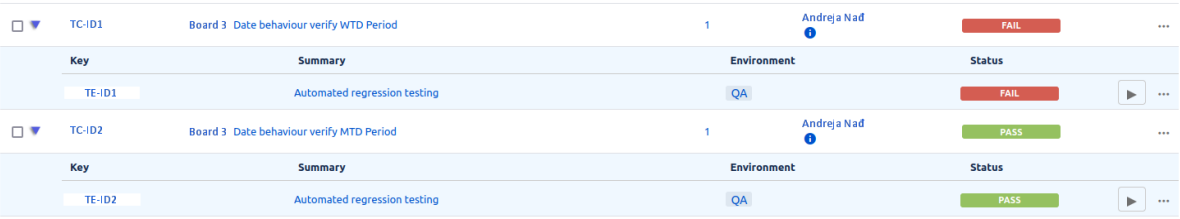
Slika 6.4. Opcije statusa rezultata testova

Jedna od grešaka pronađenih tijekom ručnog testiranja vidljiva je na slici 6.5. Na slici se nalazi primjer izvješća o grešci. Potrebno je zapisati proceduru kojom se dogodila greška, što se očekivalo da će se dogoditi te stvarni rezultat. Na slici se vidi oznaka „Requirement Status: NOK“ što predstavlja status neispunjenog zahtjeva. Zahtjev je, u ovom slučaju, da element za učitavanje stranice mora biti vidljiv dok god se stranica učitava. Element je nestao prije nego li su se svi podaci učitali i rezultirao greškom. Tijekom automatiziranog testiranja to je predstavljalo veliki problem. Nakon ispravka greške, kreirano izvješće o grešci se postavlja u stanje testiranja gdje se ponovno testira ručno ili putem automatiziranog regresijskog testiranja radi utvrđivanja nestanka greške.



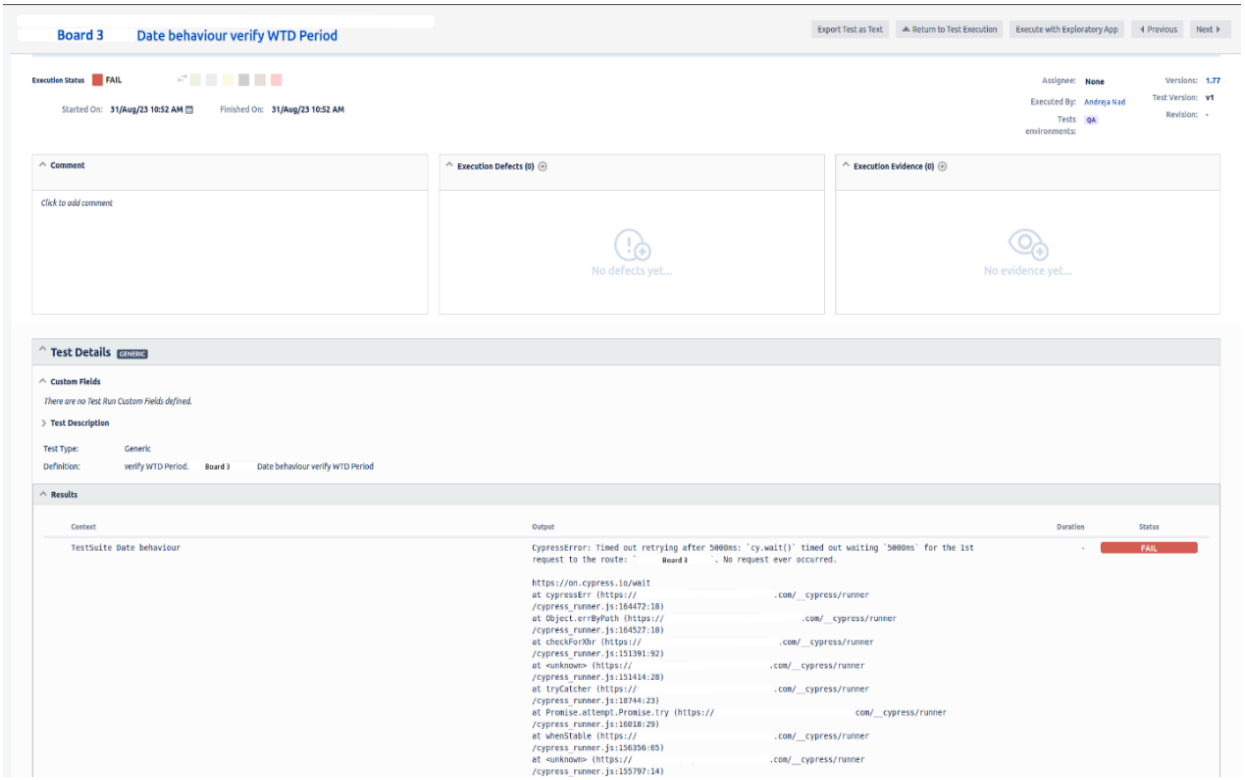
Slika 6.5. Primjer prijavljene greške

Osim pada testa elementa za učitavanje, par testova su pali i kod izvršavanja automatiziranih testova. Slika 6.6 prikazuje primjer zapisa pada automatiziranog testa, dok slika 6.7 prikazuje detaljan opis greške koja se dobila tijekom pada testa.



Slika 6.6. Primjer pada testa

U ovom slučaju, iako je test pao za odabir periodnog filtra „WTD“, kroz Cypress grešku na slici 6.7 vidljivo je da je uzrok pada testa isteklo vrijeme prilikom korištenja metode *cy.wait* za dohvaćanje URL-a stranice.



Slika 6.7. Prikaz detalja za test koji ima grešku

Dodatni testovi koji su imali padove su: testiranje poveznice i usporedba s prošlim tjednom. Testiranje poveznice je u više testnih slučajeva imalo grešku. Prilikom odabira prodajnog savjetnika putem poveznice iz tablice s druge stranice, filtar na trećoj stranici prikazivao je krivo ime. U slučaju usporedbe s prošlim tjednom, nastala greška još nije ispravljena u vrijeme pisanja ovoga rada. Kada se pritisne na „WTD“ filtar, ideja je da se datumi promjene na početak tjedna i trenutačni datum (trenutačni datum se koristi ako nije prethodno postavljen krajnji datum). Problem nastaje kada je početni datum tjedna jednak trenutačnom datumu. U tom se slučaju „WTD“ datumi prebacuju na prethodni tjedan, no pri testiranju filtra „usporedi s prošlim tjednom“ datumi se ne prebacuju već početni i konačni datumi prikazuju isti datum. Slika 6.8 prikazuje istu

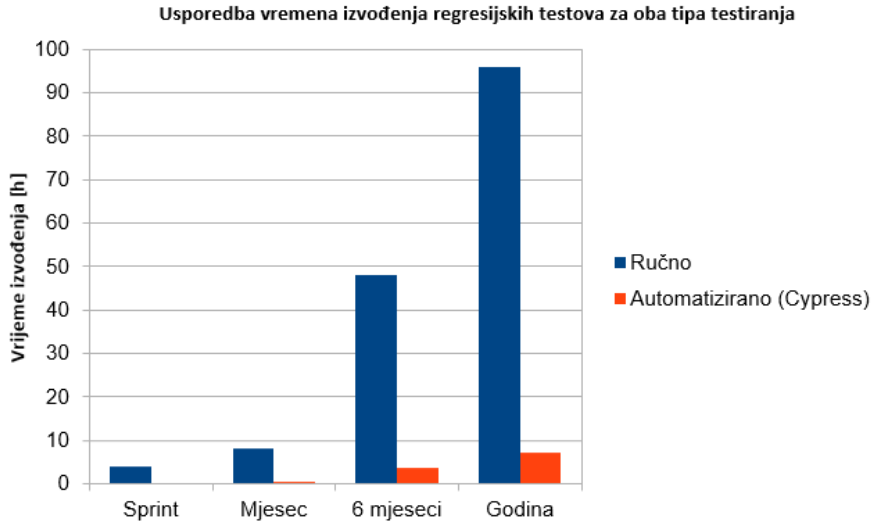
grešku unutar izvješća. Prema tome, granične vrijednosti kod usporednih filtara se trebaju dodatno pregledati i ispraviti kako bi se izbjegle greške.

```
6    <testcase classname="boards.tests.Board3.functionality.Board3FunctionalityTest" name="testPeriod_R_pw" time="10.287">
7      <failure message="expected [2023-09-04] but found [2023-09-11]" type="java.lang.AssertionError">
8        <![CDATA[java.lang.AssertionError: expected [2023-09-04] but found [2023-09-11]
9    at org.testng.Assert.fail(Assert.java:111)
10   at org.testng.Assert.failNotEquals(Assert.java:1578)
11   at org.testng.Assert.assertEqualsImpl(Assert.java:150)
12   at org.testng.Assert.assertEquals(Assert.java:132)
13   at org.testng.Assert.assertEquals(Assert.java:644)
```

Slika 6.8. Primjer greške unutar izvješća – Selenium

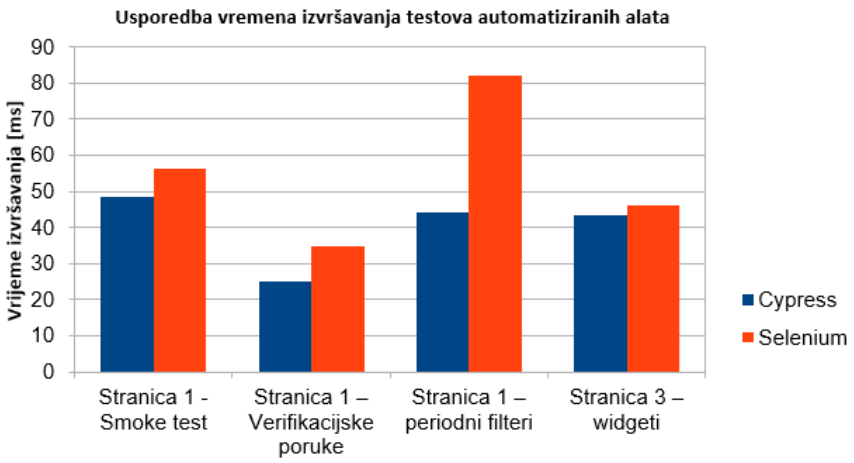
6.4. Osvrt na testiranje

Slika 6.9 prikazuje graf usporedbe vremena izvođenja regresijskih testova za ručno i automatizirano testiranje. Iz grafa je vidljivo da automatizacija ubrzava proces testiranja, ponajviše u slučaju regresijskog testiranja. Regresijsko testiranje se izvodi nad svakom novom inačicom aplikacije. To znači da bi se, ukoliko je odabrano, tijekom godine izgubilo otprilike 88 sati provedbom ručnog testiranja.



Slika 6.9. Graf usporedbe vremena izvođenja regresijskih testova za oba tipa testiranja

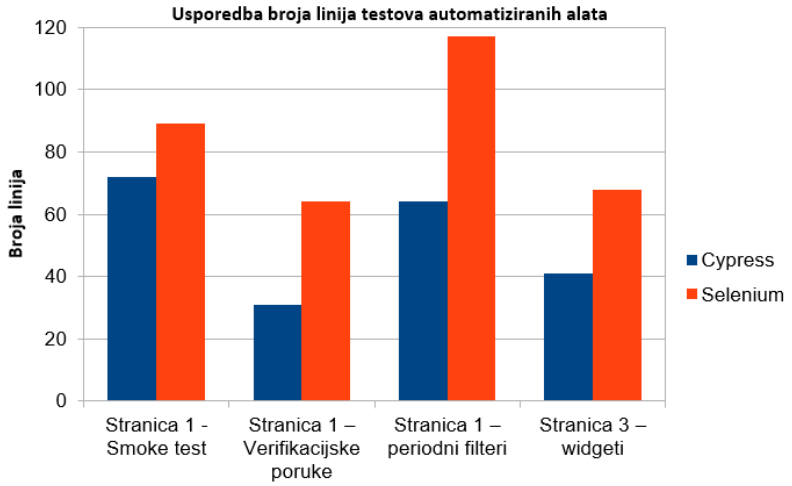
Iz grafa prikazanog na slici 6.7 može se zaključiti da je Cypress brži od Seleniuma pri izvršavanju testova. Osim što je brz, Cypress je i lagan za namjestiti i koristiti. Njegova jednostavnost u instalaciji i pokretanju čini ga odličnim izborom alata za brzi početak testiranja.



Slika 6.10. Graf usporedbe vremena izvršavanja testova automatiziranih alata

Unatoč svemu, Selenium testovi su bili lakši za napisati i ispravljati. Glavnu ulogu za to je imala naredba za čekanje određenog uvjeta elementa (npr. vidljivost) i proces otklanjanja grešaka unutar Seleniuma. Otklanjanje grešaka unutar Seleniuma tijekom pravljenja testova je slično kao i kod ispravljanja programskog kôda. Mogu se odabrati linije na kojima će program stati te je moguće pisati liniju po liniju kôda unutar terminala dok terminal automatski dohvaća rezultate sa stranice. Cypress pokaže liniju na kojoj se greška dogodila, te njeno ispravljanje zahtijeva duže vremena dok se razumije razlog padanja testova. Npr. kod testiranja poveznica unutar Cypressa, prilikom provjere filtara, odabrana vrijednost opcije je vratila „undefined“ iako kada Cypress dođe do kraja, vrijednost je vidljiva na stranici. To znači da se metoda odradila prije nego se učitala vrijednost filtra na stranici.

Cypress pravi slike stranice prilikom izvođenja svake metode što olakšava ispravljanje grešaka, no ne daje mogućnost ispitivanja zašto i kako je došlo do neke pogreške. U Seleniumu je tijekom izvedbe moguće provjeriti je li u određenom koraku učitana vrijednost te ispisati njenu vrijednost. Tako se olakšava proces utvrđivanja točnog izvora greške. Iako je Selenium osjetno lakši za napisati i ispravljati, on je imao i više linija kôda nego Cypress testovi, kao što je vidljivo na slici 6.11.



Slika 6.11. Graf usporedbe broja linija testova automatiziranih alata

Prilikom testiranja pronađene su greške koje su otkrile nedostatke u dokumentaciji. U dokumentaciji su navedeni problemi oko odabira periodnih filtara. Tako primjerice, prilikom odabira ponedjeljka i odabira „WTD“ filtra, očekuje se da se tjedan automatski prebaci na prethodni kako bi se izbjegla greška da početni i završni datum su isti datumi. U suprotnom, pretraga se ne može izvršiti. Prilikom testiranja treće stranice uočeno je da svi dodatni filtri pretrage nisu pravilno postavljeni, tj. dolazi do greške koja se namjerno izbjegavala kod periodnih filtara. Problem je što je verifikacijska poruka greške koja se izbacuje prilikom takve pretrage „Niste odabrali niti jednog prodajnog savjetnika“. Pri tom je problem u tome što se prodajni savjetnik ni ne može odabrati ukoliko razlika između početnog i završnog datuma pretrage nije barem jedan dan. Prema tome, trebaju se popraviti dodatni filtri za usporedbu s prošlim tjednom, mjesecom i godinom. Dodatno, prilikom testiranja je uočeno da za neke dane nema vrijednosti za pretragu. S obzirom da se vrijednosti u filtrima dohvaćaju ovisno o datumima, znači da postoji mogućnost da neki datumi uopće nemaju vrijednosti, pogotovo datumi koji padaju u subotu i nedjelju. No, prilikom odabira ovakvih datuma događa se isti problem kao što je već navedeno, odnosno korisniku se ne izjavljuje da odabrani datumi nisu ispravni ili da odabrani datumi ne sadrže vrijednosti. Tom verifikacijskom porukom bi se dalo korisniku do znanja u čemu je problem i da pokuša uzeti druge datume kako bi uspješno obavio pretragu.

7. ZAKLJUČAK

Proces testiranja aplikacije nužan je proces koji se treba započeti što prije u razvoju aplikacije. U radu su objašnjene vrste, tehnike, strategije i razine testiranja koje se mogu primjenjivati u fazi testiranja. U ovome radu testiranje započinje početkom razvoja web aplikacije. Obavljeno je ručno i automatizirano testiranje web aplikacije prema testnom planu koji sadrži više korisničkih scenarija s ciljem povećanja pokrivenost radnji koje bi korisnik izveo koristeći web aplikaciju. Testni plan dodatno sadrži vrste, tehnike, strategije i razine testiranja koje se moraju provoditi u oba tipa testiranja. Ručni testovi su zapisivani korak po korak prema redoslijedu izvođenja i tako su izvedeni. Automatizirani testovi su grupirani prema funkcionalnostima za lakše ponovno izvršavanje i razumijevanje pri regresijskom testiranju. Nakon završetka zapisivanja i izvođenja testova, analizirani su tipovi testiranja i uspoređeni su rezultati za dva odabrana alata za automatizirano testiranje.

Automatizirani testovi izvedeni uz korištenje Cypressa su brži i zahtijevaju manje testnog programskog kôda u odnosu na Selenium testove. S druge strane, Selenium ima prikladnije sučelje za ispravljanje grešaka prilikom pisanja testova, a Cypress je pogodniji za vizualno provođenje testiranja. Prilikom izvođenja testova, Cypress stvara slike za svaku izvedenu metodu što olakšava proces nefunkcionalnog testiranja vezanog za izgled stranice. Može se reći da su testovi pokrili sve zahtjeve i funkcionalnosti web aplikacije. Tijekom testiranja pronađene su greške koje su prijavljivane putem izvješća o greškama s ciljem daljnjeg ispravljanja. Web aplikacija ima još prostora za nadogradnje i poboljšanja nekih problema koji su uočeni tijekom testiranja. Osim web aplikacije, testovi se mogu dodatno nadograditi prilikom čekanja određenog stanja elementa web aplikacije i dohvaćanja odgovora s poslužitelja.

LITERATURA

[1] D., Graham, E., Veenendaal, I. Evans, R. Black, Foundations Of Software Testing, Cengage Learning EMEA, 2008.

[2] M.A. Friedman, J. M. Voas, Software Assessment: Reliability, Saftey And Testability, John Wiley & Sons, New York, 1995.

[3] M.D. Ernst, Static And Dynamic Analysis: Synergy And Duality, WODA 2003: Workshop on Dynamic Analysis, pp. 24 – 27, Portland, USA, 2003.

[4] G.D. Everett, R. McLeod, Software Testing: Testing Across The Entire Software Development Life Cycle, Wiley-IEEE Press, New York, 2007.

[5] J. Pan, Software Testing, 18-849b Dependable Embedded Systems, Carnegie Mellon University, USA, 1999.

[6] M. Kassab, J. F. DeFranco, P. A. Laplante, Software Testing: The State Of The Practice, IEEE Software, No. 5, Vol. 34, pp. 46 – 52, 2017.

[7] ISTQB Foundation Certification: 1.3 Testing Principles (K2) [online], Blogger, dostupno na: http://istqbfoundation.blogspot.com/p/13-testing-principles-k2.html [25. lipnja 2023.]

[8] A. Spillner, T. Linz, H. Schaefer, Software Testing Foundations: A Study Guide For The Certified Tester Exam, Rocky Nook, USA, 2014.

[9] A. Mateen, Q. Zhu, S. Afsar, Comparitive Analysis Of Manual vs. Automotive Testing For Software Quality, Proceedings of The 7th International Conference on Software Engineering and New Technologies, pp. 1 – 7, New York, 2018.

[10] A.B. Mailewa, J. Herath, S. Herth, A Survey Of Effective and Efficient Software Testing, Midwest Instuction And Computing Symposium, Vol. 48, MIC Symposium, USA, 2015, dostupno na: https://www.micsymposium.org/mics2015/ProceedingsMICS_2015/Mailewa_2D1_41.pdf [25. lipnja 2023.]

[11] G.J. Myers, C. Sandler, T. Badgett, The Art Of Software Testing, John Wiley & Sons, New Jersey, 2012.

[12] E. Dustin, T. Garret, B. Gauf, Implementing Automated Software Testing: How To Save Time and Lower Cost While Raising Quality, Pearson Education, USA, 2009.

[13] B. Homes, Fundamentals Of Software Testing, Wiley - ISTE, USA, 2012

[14] Testware [online], ISTQB Glossary, dostupno na: https://glossary.istqb.org/en_US/term/testware-11 [26. lipnja 2023.]

[15] Test Strategy [online], ISTQB Glossary, dostupno na: https://glossary.istqb.org/en_US/term/test-strategy-3 [26. lipnja 2023.]

[16] R. Pham, 5.2.6 Test Strategy, Test Approach: Test Strategy, Test Approach [online], ISTQB Foundation, Dostupno na: https://istqbfoundation.wordpress.com/2017/09/18/test-strategy-test-approach/ [26. lipnja 2023.]

[17] What Is Test Strategy? Types Of Strategies With Examples [online], Try QA, dostupno na: https://tryqa.com/what-is-test-strategy-types-of-strategies-with-examples/ [26. lipnja 2023.]

[18] Test Strategy Tutorial: Comprehensve Guide With Best Practices [online], LambdaTest, dostupno na: https://www.lambdatest.com/learning-hub/test-strategy [26. lipnja 2023.]

[19] I. Hooda, R.S. Chhillar, Software Test Process, Testing Types And Techniques, International Journal of Computer Applications, Vol. 111, No. 13, pp. 10 – 14, 2015.

[20] P. C. Jorgensen, Software Testing: A Craftsman's Approach, Auerbach Publications, New York, USA, 2014.

[21] R. D. Craig, S. P. Jaskiel, Systematic Software Testing, Artech House, USA, 2002 [27. lipnja 2023.]

[22] L. Luo, Software Testing Techniques: Technology Maturation and Research Strategy, Carnegie Mellon University, USA [27. lipnja 2023.]

[23] I. Jovanović, Software Testing Methods And Techniques, IPSI Transactions on Internet Research, Multidisciplinarna, međudisciplinarna pitanja u računarstvu i inžinjerstvu, Vol 5, No. 1, pp. 30 – 41, IPSI Bgd Internet Research Society, 2009.

[24] What Is End-To-End Testing? [online], Katalon, dostupno na: https://katalon.com/resources-center/blog/end-to-end-e2e-testing [20. kolovoza 2023.]

[25] P. Muller-Earl, Integration Testing and End-To-End Testing - What's The Difference? [online], headspin, 2023, dostupno na: https://www.headspin.io/blog/primary-difference-between-integration-testing-and-end-to-end-testing [20. kolovoza 2023.]

[26] What is End To End Testing: E2E Testing Framework With Examples [online], Software Testing Help, 2023, dostupno na: https://www.softwaretestinghelp.com/what-is-end-to-end-testing/ [20. kolovoza 2023.]

[27] M. Sharma, R. Angmo, Web based Automation Testing and Tools, International Journal of Computer Science and Information Technologies, Vol. 5, No. 1, pp. 908 – 912, 2014.

[28] D. Atesogullari, A. Mishra, Automation Testing Tools: A Comparative View, International Journal on Information Technologies & Security, Vol. 12, No. 4, pp. 63 – 76 2020.

[29] H. V. Gamido, M.V. Gamido, Comparative Review of the Features of Automated sSftware Testing Tools, International Journal of Electrical and Computer Engineering, Vol. 9, No. 5, pp. 4473 – 4478, 2019.

[30] F. Mobaraya, S. Ali, Technical Analysis of Selenium and Cypress as Functional Automatization Framework for Modern Web Application Testing, 9th International Conference on Computer Science, Engineering and Applications, pp. 27 – 46, 2019.

SAŽETAK

Ovaj diplomski rad bavi se usporedbom ručnog i automatiziranog testiranja, analizom razlika među njima, razloga uporaba i mogućnosti za testiranje na složenoj web aplikaciji. U teorijskom dijelu diplomskog rada objašnjeni su problemi, izazovi i mogućnosti ručnog i automatiziranog testiranja programa. Objašnjeni su alati Cypress i Selenium za automatizirano testiranje, korištene biblioteke i metode kao i njihove mogućnosti prilikom testiranja web aplikacije na klijentskoj i poslužiteljskoj strani. U praktičnom dijelu rada, predloženi su testni plan, scenariji i slučajevi testiranja za provedbu ručnog i automatiziranog testiranja na funkcionalnoj i nefunkcionalnoj razini. Navedeni testni plan sadrži scenarije, vrste, tehnike, strategije i razine testiranja koje se provode tijekom testiranja web aplikacije. Provedena je usporedba brzine izvedbe ručnog i automatiziranog testiranja te automatiziranog testiranja s alatima Cypress i Selenium. Objašnjene su faze testiranja web aplikacije i prikazan proces testiranja unutar tvrtke te preložena unaprjeđenja programskog rješenja. Obavljena je supredna analiza oba tipa i alata testiranja na primjeru izvedbe testiranja tijekom razvoja web aplikacije tvrtke.

Ključne riječi: automatizirano testiranje, Cypress, ručno testiranje, Selenium, web aplikacija.

ABSTRACT

This thesis deals with the comparison of manual and automated testing, their differences, reasons for use and possibilities for testing a complex web application. In the theoretical part of the thesis, the problems, challenges and possibilities of manual and automated program testing are explained. Cypress and Selenium automated tools, libraries and methods used, as well as their capabilities when testing web applications on the client and server side are explained. In the practical part of the work, a test plan, scenarios and test cases are proposed for the implementation of manual and automated testing at the functional and non-functional level. The specified test plan contains scenarios, types, techniques, strategies and levels of testing that are performed during web application testing. A comparison of the performance speed of manual and automated testing and Cypress and Selenium automated testing was performed. The stages of testing the web application were explained and the testing process within the company was presented, as well as the proposed improvements to the software solution. An analysis was performed between both types and both testing tools on the example of testing performance during the development of the company's web application.

Keywords: automated testing, Cypress, manual testing, Selenium, web application.

ŽIVOTOPIS

Andreja Nađ rođena je 19. lipnja 1999. godine u Osijeku, Hrvatska. Pohađala je Elektrotehničku i prometnu školu Osijek, gdje je uspješno završila smjer za tehničara za mehatroniku. U 2016. godini završila je stručno osposobljavanje u Šolskom centru Škofja Loka, u Sloveniji, za elektropneumatiku i električno upravljanje te je iz iste teme sljedeće godine osvojila deveto mjesto na državnom natjecanju. Isto tako, 2016. godine sudjeluje u Robotics Camp 2016. U 2018. godini je odradila praksu sa Erasmus+ projekom ponovno u Šolskom centru Škofja Loka. Iste te godine je sudjelovala na Erasmus+ projektu Wonderland gdje je u suradnji sa ostalim zemljama partnerima poboljšala svoje komunikacijske vještine i engleski jezik. Nakon završetka srednje škole, upisuje Preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija koji uspješno završava te nastavlja Diplomski studij Izborni blok Informacijske i podatkovne znanosti na istom fakultetu. Vješta na području engleskog jezika, odgovorna i uporna.

_____

Potpis autora

PRILOZI

Prilog 1. Diplomski rad u datoteci docx.

Prilog 2. Diplomski rad u datoteci pdf

Prilog 3. Programski kôdovi automatiziranih testova