Automatizirano testiranje mobilne Android aplikacije korištenjem alata Appium

Aščić, Barbara

Master's thesis / Diplomski rad

2022

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:200:040228

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-11**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
ELEKTROTEHNIČKI FAKULTET RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

AUTOMATIZIRANO TESTIRANJE MOBILNE ANDROID APLIKACIJE KORIŠTENJEM ALATA APPIUM

Diplomski rad

Barbara Aščić

Osijek, 2022.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 15.09.2022.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Barbara Aščić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1102R, 13.10.2020.
OIB studenta:	36882794194
Mentor:	Prof.dr.sc. Goran Martinović
Sumentor:	,
Sumentor iz tvrtke:	Nikolina Mihić, mag.math.
Predsjednik Povjerenstva:	Prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	Prof.dr.sc. Goran Martinović
Član Povjerenstva 2:	Doc.dr.sc. Ivan Vidović
Naslov diplomskog rada:	Automatizirano testiranje mobilne Android aplikacije korištenjem alata Appium
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U radu je potrebno analizirati i opisati načela, razine, vrste, izazove, metodologiju i okoline za testiranja mobilnih aplikacija. Također, potrebno je opisati specifičnosti testiranja mobilnih web, nativnih i hibridnih aplikacija, te objasniti ulogu i utjecaj testiranja u razvoju mobilnih aplikacija. Posebno je važno istaknuti ulogu regresijskog testiranja u razvojnom ciklusu programskog rješenja. Za složenu mobilnu aplikaciju Azil Osijek čija je svrha olakšati korisnicima postupak volontiranja i udomljavanja štićenika Azila, potrebno je napraviti plan testiranja i dizajnirati testne slučajeve, a u programskom jeziku Java koristeći alat Appium automatizirati regresijske testove i po potrebi druge oblike testiranja. Dizajn i implementaciju, te rezultate testova
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	15.09.2022.

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, 30.09.2022.

Ime i prezime studenta:	Barbara Aščić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1102R, 13.10.2020.
Turnitin podudaranje [%]:	3

Ovom izjavom izjavljujem da je rad pod nazivom**: Automatizirano testiranje mobilne Android aplikacije korištenjem alata Appium**

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. AUTOMATIZIRANO TESTIRANJE I PRIKAZ STANJA U PODRUČJU 3
2.1. Važnost testiranja 3
2.2. Testna načela 3
2.3. Razine i vrste testiranja mobilnih aplikacija 4
2.3.1. Funkcionalno testiranje 5
2.3.2. Nefunkcionalno testiranje 6
2.3.3. Regresijsko testiranje 7
2.4. Automatizirano testiranje 7
2.5. Alati za automatizirano testiranje 9
2.6. Stanje u području automatiziranog testiranja 10
3. DEFINIRANJE TESTNE OKOLINE I POSTAVKE TESTIRANJA 12
3.1. Programsko rješenje aplikacije Azil Osijek 12
3.1.1. Sloj domene 13
3.1.2. Podatkovni sloj 14
3.1.3. Prezentacijski sloj 15
3.2. Korisničko sučelje aplikacije Azil Osijek 16
3.3. Testna okolina 19
3.3.1. Appium 19
3.3.2. Cucumber 20
3.3.3. Java 21
3.3.5. Appium Inspector 23
3.3.6. Virtualni uređaji 23
3.4. Testni plan i testni slučaj 23
3.4.1. Testni plan 23
3.4.2. Testni slučajevi 25
3.5. Veza između testera i programera 27
4. PROVEDBA TESTIRANJA 28
4.1. Korištenje Appium Inspectora 28
4.2. Postavljanje Appiuma 29
4.3. Klase Pages 30
4.4. Step definition 31
4.4.1. Gherkin 32
4.5. Runners 33
4.6. Datoteke feature 34
4.7. Pokretanje testova 36
5. ANALIZA REZULTATA TESTIRANJA 37
5.1. Prikaz rezultata testiranja 37

5.2. Analiza koraka 42
5.3. Utjecaj testiranja na razvoj programskog rješenja 43
5.3.1. Izvješće o greški 43
6. ZAKLJUČAK 45
LITERATURA 46
SAŽETAK 48
ABSTRACT 49
ŽIVOTOPIS 50
PRILOZI 51

1. UVOD

Velika rasprostranjenost mobilnih uređaja i njihova prisutnost u svakodnevnom životu doveli su do razvoja mobilnih aplikacija koje se upotrebljavaju u različitim područjima. Sukladno s time potrebno je osigurati kvalitetu s novim potrebama takvog razvoja mobilnih aplikacija [1]. Kako bi svaka aplikacija bila prihvatljiva korisniku za korištenje mora proći ciklus koji se naziva životni ciklus aplikacije. Životni ciklus najčešće je podijeljen u šest etapa: planiranje, definiranje zahtjeva, dizajn proizvoda, implementacija i dokumentacija, testiranje i održavanje. Razvoj aplikacije odvija se u suradnji više osoba koje zajedno rade u timu, gdje svaki član tima ima određeni zadatak koji mora obaviti. U praksi se testiranje provodi od samog početka razvoja proizvoda pa do njegove isporuke na tržište kako bi se otklonilo što više grešaka i povećala kvaliteta programske podrške. Testiranje je važno jer greške programske podrške mogu biti skupe i opasne te mogu potencijalno uzrokovati novčane i ljudske gubitke a povijest je puna takvih primjera. Kako ručno testiranje može biti skup i vremenski dugotrajan proces, raste potreba za automatiziranim testovima koja umanjuje potrebno vrijeme za testiranje. Automatizirano testiranje omogućeno je pisanjem testova koji opisuju ponašanje korisnika, testovi se pokreću u programu i testiraju mobilnu aplikaciju umjesto da se testovi provode ručno. Postoje razni alati za provedbu automatiziranih testova, te se pri odabiru zadovoljavajućeg alata uzima u obzir radi li se o web ili mobilnoj aplikaciji, koliko je velik sustav koji se testira, koliko korisnika će se služiti sustavom i slično.

U ovom radu bit će opisane tehnike testiranja i glavna razlika između funkcionalnog i nefunkcionalnog testiranja, provedba i važnost regresijskog testiranja s naglaskom na automatizirano testiranje, koje član tima zadužen za testiranje mora obaviti. Objasnit će se sedam testnih načela i važnost samog testiranja u razvoju proizvoda. Analizirat će se alati za automatizirano testiranje te objasniti zašto je izabran alat Appium. Opisat će se korisničko sučelje aplikacije Azil – Osijek te arhitektura *clean*. Dat će se primjer testnog plana i testnih slučajeva koji se koriste prilikom testiranja aplikacije. Objasnit će se arhitektura Appium projekta s naglaskom na klase *Pages, StepDefinition* i datoteku značajki. Detaljno će se analizirati jezik Gherkin koji se koristi za pisanje testova. Nakon pokretanja testova analizirati će se rezultati testiranja koji će se prikladno dokumentirati. Nakon toga dati će se prijedlog za poboljšanje programskog rješenja aplikacije i osvrt automatiziranog testiranja s obzirom na ručno testiranje.

U drugom poglavlju rada objašnjena je važnost testiranja, opisana su testna načela i vrste testiranja, objašnjena je veza između regresijskog i automatiziranog testiranja te su analizirani najčešći alati korišteni za automatizirano testiranje. U trećem poglavlju definirana je testna okolina i svi potrebni programi i alati za provedbu automatiziranih testova. Raspisan je testni plan i primjer testnih slučajeva za aplikaciju. U četvrtom poglavlju provedeno je testiranje aplikacije Azil te je detaljno objašnjena struktura projekta. U petom poglavlju analizirani su rezultati te je dana analiza o isplativosti provedbe automatiziranih testova s obzirom na ručno testiranje.

2. AUTOMATIZIRANO TESTIRANJE I PRIKAZ STANJA U PODRUČJU

U ovom poglavlju objašnjena je važnost testiranja te testna načela koja se moraju poštovati prilikom testiranja. Objašnjene su vrste testiranja s naglaskom na funkcionalno i nefunkcionalno testiranje, objašnjena je veza između regresijskog i automatiziranog testiranja. Detaljno je analizirano automatizirano testiranja i alati koji služe za njihovu provedbu.

2.1. Važnost testiranja

Testiranje programske podrške je metoda kojom se provjerava zadovoljava li proizvod definirane zahtjeve i osigurava da taj proizvod bude bez grešaka. Prema [2], testiranje ima mnogo prednosti, a neke od njih su :

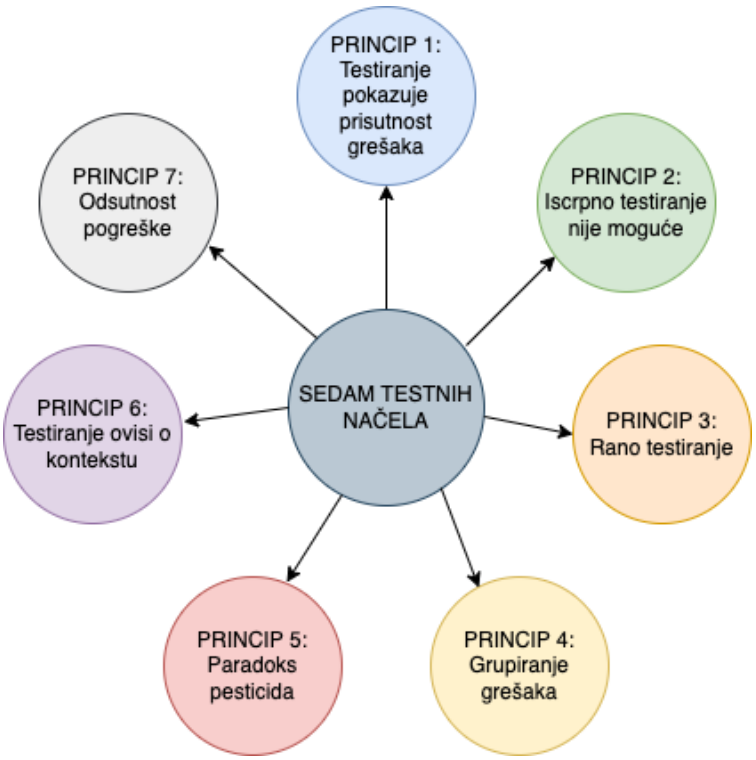
- Isplativost – testiranje pomaže za dugoročnu uštedu novca, u slučaju da se greške otkriju u ranijoj fazi testiranja programske podrške, njihovo ispravljanje manje košta
- Brža implementacija aplikacije – smanjuje se životni ciklus razvoja aplikacije ranim otkrivanjem grešaka, što dovodi do brže implementacije aplikacije
- Sigurnost – najranjivija i najosjetljivija prednost testiranja programske podrške, korisnici proizvoda traže proizvode od povjerenja
- Kvaliteta proizvoda – testiranje osigurava isporuku kvalitetnog proizvoda kupcima
- Zadovoljstvo kupaca – UI/UX testiranje osigurava najbolje korisničko iskustvo

2.2. Testna načela

Prema [3], tijekom testiranja bitno je dobiti točne rezultate, a kako bi se to moglo ostvariti testeri se trebaju držati sedam testnih načela (slika 2.1).

1. načelo: Testiranjem se dokazuje prisutnost grešaka – no ne može se tvrditi obrnuto, ukoliko greške nisu pronađene, da one ne postoje.
2. načelo: Iscrpno testiranje je moguće – testiranje svega (svih mogućih ulaza i kombinacija) nije izvedivo, zahtjeva mnogo vremena i novca, umjesto toga testira se po prioritetima
3. načelo: Rano testiranje – testne aktivnosti trebale bi započeti u najranijoj fazi životnog ciklusa programske podrške kako bi se uštedili novac i vrijeme
4. načelo: Grupiranje grešaka – mali broj komponenata sadrži većinu grešaka otkrivenih tijekom testiranja prije isporuke klijentu.

5. načelo: Paradoks pesticida – ukoliko se isti testovi iznova ponavljaju nije moguće pronaći greške. Kako bi se izbjegao ovaj paradoks, potrebno je redovno nadzirati testove i pisati nove.
6. načelo: Testiranje ovisi o kontekstu – testiranje se drugačije radi ukoliko se radi o drugačijem okruženju, npr. sigurnosno testiranje internetske trgovine ne provodi se na jednaki način kao i sigurnosno testiranje komercijalne web stranice koja služi samo za gledanje.
7. načelo: Zabluda o odsutnosti pogrešaka – pronalaženje i uklanjanje pogrešaka je beznačajno ukoliko je izgrađeni proizvod neupotrebljiv i ne ispunjava potrebe i očekivanja korisnika.



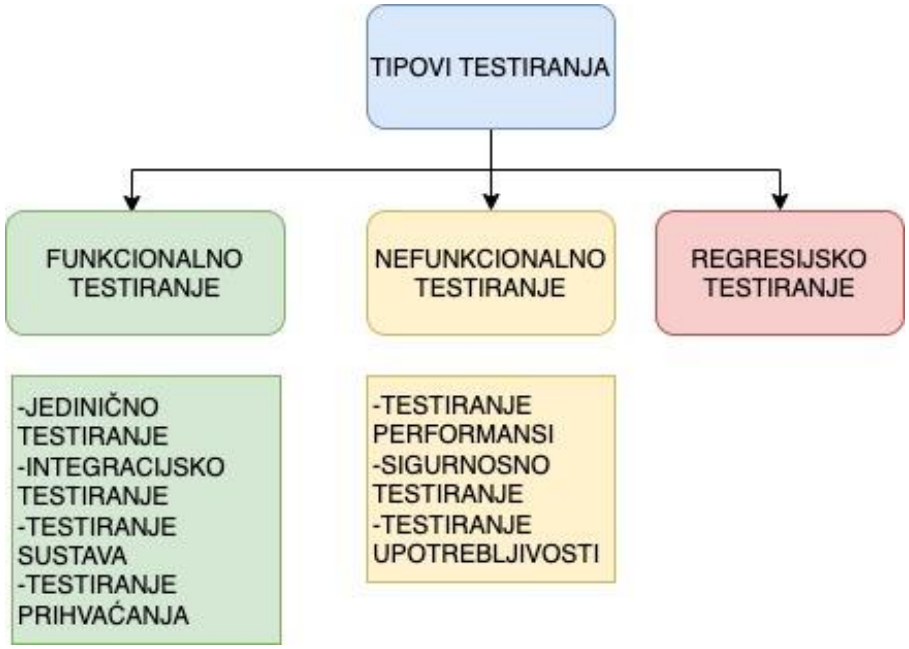
Slika 2.1. Sedam testnih načela [3]

2.3.Razine i vrste testiranja mobilnih aplikacija

U području mobilnih aplikacija često se čuje izraz: nativna aplikacija, hibridna aplikacija i web aplikacija. Glavna razlika je što se nativne i hibridne aplikacije mogu instalirati u trgovini aplikacija, dok su web aplikacije zapravo web stranice optimizirane za mobilne uređaje te izgledaju kao aplikacija. Iz perspektive osiguravatelja kvalitete mobilna aplikacija definira se

kao aplikacija koja se pokreće korisničkim unosima na mobilnom uređaju s ograničenim resursima. S druge strane web aplikacija se definira kao aplikacija koja se nalazi na poslužitelju i kojoj se može pristupiti preko interneta [3].

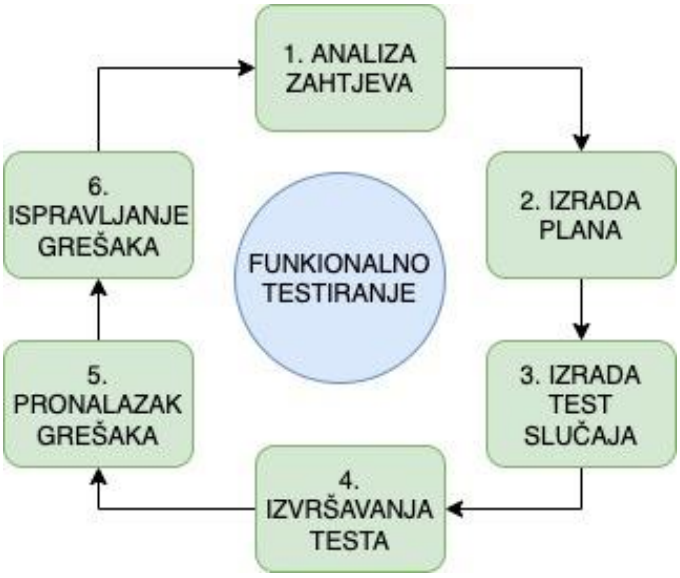
Aplikacija koja je testirana u ovom radu nativna je aplikacija. Nativna mobilna aplikacija izrađena je isključivo za jednu platformu i može u potpunosti koristiti sve značajke uređaja (GPS, kameru, kompas, akcelometar itd.). Prednosti nativnih aplikacija su te što mogu raditi bez internetske veze, imaju veću brzinu i bolje korisničko iskustva. Nedostatak je taj što održavanje može biti komplicirano i za korisnike i za programere (više inačica aplikacije, ponovno postavljanje na trgovinu aplikacija, ponovna instalacija itd.) [4].
Uobičajeno se testiranje mobilnih aplikacija klasificira u 3 kategorije: funkcionalno, nefunkcionalno i testiranje održavanja. Slika 2.2 prikazuje podjelu tipova testiranja.



Slika 2.2. Tipovi testiranja [3]

2.3.1. Funkcionalno testiranje

Funkcionalno testiranje još se naziva i *black-box* testiranje, ali ono može uključivati i nefunkcionalno testiranje. Proces funkcionalnog testiranja sastoji se od analize zahtjeva, izrade plana, izrade test slučajeva, izvršavanja testova, pronalaska grešaka i ispravljanja grešaka. Na slici 2.3 prikazan je proces funkcionalnog testiranja.



Slika 2.3. Proces funkcionalnog testiranja

Funkcija sustava ili komponenta sustava obično se opisuje u specifikacijama zahtjeva te se funkcionalno testiranje temelji na tim funkcijama. Svaka komponenta mora biti dokumentirana i shvaćena od strane ispitivača.

Testiranje se može provesti na svim razinama:

- jedinično testiranje
- integracijsko testiranje
- testiranje sustava
- testiranje prihvaćanja.

Testiranje komponenti, poznatije kao testiranje jedinica, modula i programa traži nedostatke i provjerava funkcioniranje pojedine komponente. Njegova specifičnost je što se svaka komponenta može testirati zasebno i izolirano od ostatka sustava. Integracijsko testiranje testira sučelja između komponenti te interakciju s različitim dijelovima sustava kao što su datotečni sustav, operativni sustav i drugo. Testiranje sustava odnosi se na provjeru ponašanja cijelog sustava s obzirom na definirane zahtjeve. Testiranje prihvaćanja i testira valjanost programske podrške s obzirom na potrebe korisnika kako bi se utvrdilo je li sustav prihvatljiv [3].

2.3.2. Nefunkcionalno testiranje

Nefunkcionalno testiranje opisuje koliko dobro sustav radi. Najpoznatije metode nefunkcionalnog testiranja su: testiranje performansi, sigurnosno testiranje, testiranje upotrebljivosti. Testiranje performansi provodi se mjerenjem vremena odziva, određivanjem

najvećeg broja korisnika kojeg sustav može podnijeti, stabilnosti sustava pod različitim opterećenjima, itd. Sigurnosno testiranje osigurava siguran i stabilan sustav te pronalazi sigurnosne nedostatke zbog kojih se može doći do napada i ugrožavanja podataka sustava. Sigurnosno testiranje uključuje procjenu rizika, procjenu držanja te sigurnosno skeniranje. Testiranje upotrebljivosti poznato je još kao testiranje korisničkog iskustva te služi za mjerenje koliko je sustav jednostavan i intuitivan za korištenje. Još neke od vrsta nefunkcionalnog testiranja su instalacijsko testiranje kojim se provjerava hoće li sustav raditi ispravno na svakom uređaju te testiranje oporavka kojim se testira koliko brzo se sustav može opraviti nakon neočekivanog ispada ili kvara [5].

2.3.3. Regresijsko testiranje

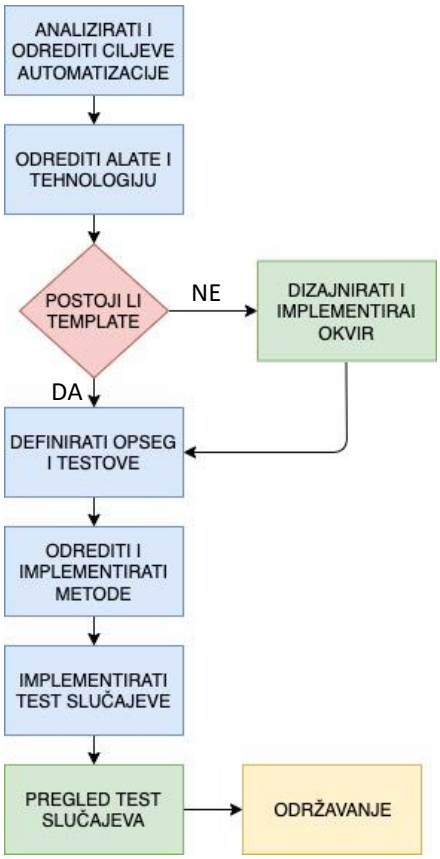
Regresijsko testiranje [3] je potpuni ili djelomičan odabir već izvršenih testnih slučajeva koji se ponovno izvode kako bi se osiguralo da izmjena u programskom kodu nije negativno utjecala na postojeće funkcionalnosti sustava. Regresijsko testiranje treba se provoditi pri svakoj izmjeni koda, može uključivati testiranje cijelog sustava ili testiranje pojedinih komponenti koje imaju vezu sa kodom koji je izmijenjen ili dodan u program. No provođenje regresijskih testova može zahtijevati puno vremena ukoliko se izvode ručno. Zbog toga testeri mogu upotrijebiti automatizirane testove koji će testiranje odraditi umjesto njih. U ovom su radu nadalje korišteni regresijski testovi koji uključuju funkcionalne i nefunkcionalne testove te njihova automatizacija.

2.4. Automatizirano testiranje

Svakom novom promjenom ili novom implementacijom u projektu inženjeri osiguranja kvalitete moraju proći kroz cijelu aplikaciju kako bi se uvjerili jesu li nove značajke utjecale ili prouzročile nove greške u radu cjelokupnog projekta. Provedbom automatiziranih testova štedi se vrijeme i novac, postiže se veća točnost u provođenju testova zbog izbjegavanja ljudskog faktora te se osiguravaju radnje koje se ponavljaju. Automatizirano testiranje karakteristično je za agilni razvoj proizvoda, te kako se agilni način sve više koristi u razvoju programske podrške tako raste i potreba za automatizacijom testova [6]. Regresijski testovi se izvode svakodnevno jer je agilnim načinom rada omogućeno više izmjena tijekom razvoja, stoga je provođenje automatiziranih testova nužno za ubrzanje razvoja proizvoda. Kao što se navodi u [7]: „Automatizirano testiranje uspješno je u tradicionalnom i agilnom razvoju, ali agilni razvoj ne može uspjeti bez automatiziranog testiranja“.

Svrha automatiziranog testiranja je osiguravanje automatizirane podrške za postupke u testiranju. U razvoju programske podrške tim osiguranja podrške, zadužen za dizajn i izvođenje testova razvija okruženje za automatizirano testiranje. U usporedbi s ostalim testovima, automatizirano testiranje olakšava sigurnosno testiranje, testiranje opterećenja i konfiguracijsko testiranje. Planiranje u automatiziranom testiranju neophodno je kao i planiranje u ručnom testiranju. Prema [8] planiranje automatiziranog testiranja uključuje:

- Analizu i planiranje
- Odabir alata i tehnologija
- Odabir i dizajn testnog okruženja
- Definiranje opsega testiranja
- Identifikaciju zahtjeva
- Implementaciju testnih slučajeva
- Pregled testnih slučajeva
- Održavanje



Slika 2.4. Dijagram planiranja automatiziranog testiranja

Automatizirano testiranje zahtjeva pisanje računalnih programa za pronalaženje grešaka ili nedostataka. To je idealan pristup za zamjenu dugotrajnog i iscrpnog ručnog testiranja. Prema [9], automatizirano testiranje vrlo je kritičan proces; potrebno je odrediti koje funkcionalnosti je potrebno automatizirati, potrebna su značajna ulaganja u smislu analize značajki programske podrške koja će se automatizirano testirati, nabavi alata, pisanja skripti, obuke testera i drugo. Stoga je potrebna precizna analiza ulaganja i povrata prije početka automatiziranog testiranja. No ono ima razne prednosti poput poboljšanja kvalitete proizvoda, izbjegavanja ljudske pogreške, ubrzavanja procesa izvršavanja testova čime se povećava vjerojatnost poštivanja strogih rokova isporuke.

Prema usporedbi s ručnim testiranjem, automatizirano testiranje dovodi do potvrdnog ponašanja. Uzrok tome je težina kodiranja nedosljednih testova i ograničenje alata za automatizaciju u tom pogledu. „Automatsko testiranje je više o sretnim slučajevima, može se tako reći. Naravno, može se nositi i s negativnim (nedosljednim) testnim slučajevima ali ručno testiranje može bolje rukovati s negativnim, slučajnim ili neočekivanim rezultatima. S automatskim testnim slučajevima teže je nositi se s neočekivanim rezultatima“ [10]. No kako bi se započelo automatizirano testiranje potrebno je savladati tehnike ručnog testiranja. Ukoliko se žele razviti kvalitetni automatizirani testovi, tester mora znati sve o aplikaciji s kojom radi, a jedini način da to postigne je da prvo aplikaciju testira ručno.

2.5. Alati za automatizirano testiranje

Uspjeh provedbe automatiziranih testova ovisi o prepoznavanju pravog alata za projekt. Automatizirani alati koriste se za olakšavanje općenito svih dizajna i procesa testiranja. Najbolji alati za testiranje programske podrške su oni koji se mogu modificirati prema svojim potrebama testiranja. Postoje različiti alati za testiranje različitih aplikacija, aplikacije mogu biti nativne, hibridne i web. Stoga prije odabira alata treba razmisliti o sljedećim čimbenicima: [11]

- Platformi koja se testira
- Dostupnosti alata, potrebi za licencom
- Zahtjevima programske podrške i sklopovlju računala
- Jednostavnosti korištenja

Selenium je jedan od najboljih alata za web automatizaciju osiguranja kvalitete, može automatizirati više operativnih sustava poput Windowsa, Maca i Linuxa te preglednika poput

Fire Foxa, Google Chrome-a itd. Selenium nudi značajke snimanja i reprodukcije sa svojim dodatkom za Selenium IDE. Prema [12] najpoznatiji alati za testiranje mobilnih aplikacija su: Appium, Calabash, XCUITest, EarlGrey i TestGrid. S mnoštvom otvorenih i komercijalnih alata koji se mogu izabrati za automatizirano testiranje proces odabira može biti težak.

Alat Appium objašnjen je u trećem poglavlju jer se koristiti za provedbu automatiziranih testova. Calabash je mobilno okruženje za pisanje automatiziranih testova koji podržava više jezika: Ruby, Java, Flex i .NET. Sadržava biblioteke koje omogućuju programsku interakciju testnih skripti s nativnim i hibridnim aplikacijama. Otvorenog je koda i podržava Cucumber što omogućuje pisanje testova na prirodnom jeziku koji mogu razumjeti ne tehnički stručnjaci i testeri. Među alatima za testiranje mobilnih uređaja, XCUITest najpoznatiji je po testiranju iOS aplikacija. Pokrenut je od strane Apple-a 2015. godine. Namijenjen je za pokretanje UI testova na iOS aplikacijama koristeći Swift/Objective C programske jezike. Poznat je po brzom izvršavanju testova, intuitivnom načinu rada i jednostavnim upravljanjima testova. EarlGrey razvijen je od strane Google-a kao okvir za testiranje, posebno koristan za izradu korisničkog sučelja i funkcionalnosti testova. Koristi Unit Testing Target i ugrađenu sinkronizaciju koja omogućava lakše kreiranje i održavanje testova. Također omogućuje automatizirano testiranje nativnih i hibridnih aplikacija. TestGrid omogućuje korisnicima ručno i automatizirano testiranje mobilnih aplikacija na stvarnim uređajima ili virtualnim uređajima. Bazira se na ponovnoj upotrebi gotovih testova na različitim inačicama aplikacije, kao i na drugim aplikacija jer je automatizacija bazirana na umjetnoj inteligenciji. Nije otvorenog koda te se plaća mjesečno.

2.6. Stanje u području automatiziranog testiranja

Budući da većina poduzeća treba dostaviti mobilnu aplikaciju što je brže moguće na tržište, mobilne aplikacije razvijaju se u agilnom razvojnom modelu. Agilni razvoj karakterizira njegova brzina razvoja i prihvaćanja promjena. S ograničenim vremenom ne može se očekivati da se pokriju svi mogući slučajevi korištenja aplikacije. Testovi koji su nužni za izvođenje su: testiranje performansi, funkcionalno testiranje, testiranje instalacije i sigurnosno testiranje [13]. Svi navedeni testovi mogu se provesti automatiziranim testiranjem, no potrebno je odabrati odgovarajući alat. Kada ljudi razmišljaju o alatu za testiranje obično na umu imaju alat za izvođenje testova, alat koji može pokretati testove. No postoje različita područja, a s time i različite vrste alata za testiranje. Alati su grupirani prema aktivnostima testiranja ili područjima koja su podržana skupom alata, na primjer, alati koji podržavaju aktivnosti upravljanja, alati za

podršku u statičnom testiranju. Neki alati obavljaju vrlo specifične i ograničene funkcije ali mnogi komercijalni alati pružaju podršku za niz različitih funkcija.

Prema [3] postoje mnoge prednosti koje se mogu dobiti korištenjem alata za testiranje:

- smanjenje rada koji se ponavlja
- veća dosljednost i ponovljivost
- objektivno ocjenjivanje
- jednostavnost pristupa informacijama o testovima ili testiranju

Ponavljajući posao zamoran je obavljati ručno, sklon je greškama ukoliko se uvijek iznova rade isti zadatci. Primjer ponavljajućeg rada uključuje regresijske testove, unošenje istih testnih podataka iznova, a oboje se može obaviti pomoću alata za izvođenje testova. Iako postoje značajne koristi koje se mogu postići korištenjem alata za testiranje, postoje mnoge organizacije koje nisu postigle koristi koje su očekivale. Samo korištenje alata nije jamstvo za postizanje veće kvalitete proizvoda. Svaka vrsta alata zahtijeva ulaganje truda i vremena kako bi se postigla potencijalna korist [3]. Mnogo je rizika koji su prisutni kada se uvede i koristi podraška alata za testiranje, bez obzira na vrstu alata. Rizi uključuju:

- nerealna očekivanja od alata
- podcjenjivanje vremena, troškova i truda za početno uvođenje alata
- podcjenjivanje vremena i truda potrebnih za postizanje značajnih i trajnih koristi od alata
- pretjerano oslanjanje na alat, smanjenje ručnog testiranja

Dobra je ideja koristiti računala za obavljanje stvari u kojima su računala stvarno dobra, a u kojima ljudi nisu baš dobri. Dakle, korištenje alata vrlo je korisno za zadatke koji se stalno ponavljaju - računalo se neće umoriti i moći će točno ponoviti ono što je učinjeno prije. Testiranje različitih uređaja poseban je izazov u provedbi testiranja aplikacije. Naročito aplikacija na Android operacijskom sustavu gdje različiti mobilni uređaji pružaju različite značajke i sklopovske komponente. Do 2014. godine razvijeno je oko 130 različitih mobilnih uređaja sa sustavom Android, te sedam inačica operacijskog sustava Android [14]. Automatiziranim testiranjem rješava se problem testiranja na različitim uređajima jer je omogućeno korištenje emulatora.

3. DEFINIRANJE TESTNE OKOLINE I POSTAVKE TESTIRANJA

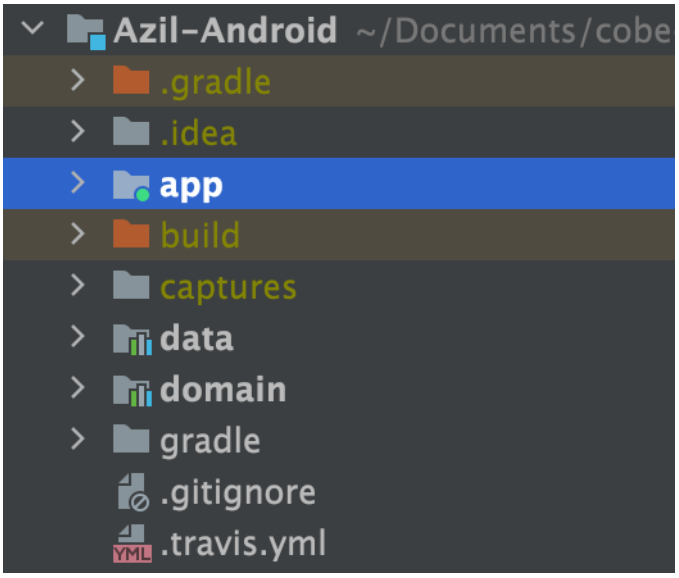
U ovom poglavlju opisana je mobilna aplikacija Azil Osijek na kojoj je provedeno automatizirano testiranje, programsko rješenje aplikacije te korisničko sučelje aplikacije. Opisani su alati koji su potrebni za pisanje i izvršavanje automatiziranih testova te jezici Java i Gherkin. Opisana je testna okolina, testni plan i testni slučajevi. Navedeni su ciljevi testiranja i objašnjena je veza između programera i testera u praktičnijem smislu.

3.1. Programsko rješenje aplikacije Azil Osijek

Aplikaciju Azil Osijek izradila je osječka firma Cobe. Aplikacija Azil Osijek je Android aplikacija pisana u programskom jeziku Kotlin te je izrađena korištenjem arhitekture *Clean*. Takva arhitektura uključuje tri modula [15]:

- prezentacijski sloj – modul koji prikazuje podatke
- podatkovni sloj – modul za dohvaćanje podataka s lokalne ili udaljene baze podataka
- sloj domene – modul koji sadržava svu poslovnu logiku

Na slici 3.1 vidljiva je arhitektura *Clean* gdje datoteka *app* predstavlja prezentacijski sloj, datoteka *data* podatkovni sloj te *domain* datoteka sloj domene.



Slika 3.1. *Clean* arhitektura Azil aplikacije

Ovisnosti ili tijek podataka unutar arhitekture *Clean* može se podijeliti u pet etapa.

1. Korisnički pozivi iz prezentacijskog sloja
2. Prezentacijski sloj izvršava *usecase*
3. *Usecase* iz sloja podataka šalje upit podatkovnom sloju
4. Podatkovni sloj se pokreće i šalje rezultat *usecase*-u
5. Informacija se šalje natrag korisniku

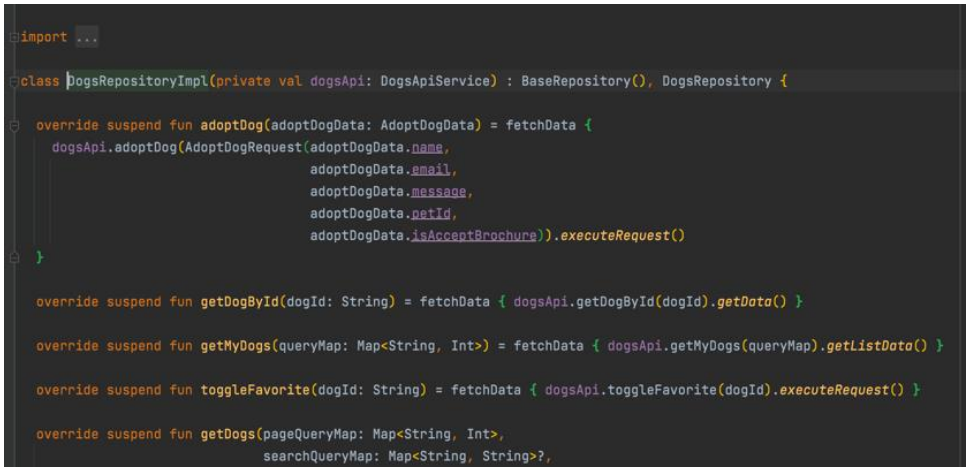
3.1.1. Sloj domene

Sloj domene smatra se jezgrom aplikacije. Sloj domene može sadržavati *Usecaseve*, entitete, i *DataLayer* sučelja. Preko sučelja se pristupa konkretnim implementacijama koje se nalaze u podatkovnom sloju. Na slici 3.2 prikazano je sučelje *DogsRepository* koje sadrži deklaraciju metoda za dohvaćanje pasa i dohvaćanje jednog psa po imenu. Također vidljive su metode za udomljavanje pasa i odabir najdražeg psa. Sve navedene metode mogu biti blokirajuće, a to se ostvaruje ključnom riječju *suspend.*

```
import ...

interface DogsRepository {

    suspend fun getDogs(pageQueryMap: Map<String, Int>, searchQueryMap: Map<String, String>?,
                        filterQueryMap: Map<String, String>?, sort: String?): Result<DogsDomain>

    suspend fun getDogById(dogId: String): Result<Dog>

    suspend fun getMyDogs(queryMap: Map<String, Int>): Result<List<Dog>>

    suspend fun toggleFavorite(dogId: String): Result<Unit>

    suspend fun adoptDog(adoptDogData: AdoptDogData): Result<Unit>

}
```

Slika 3.2. Sučelje *DogsRepository* u sloju domene

Na slici 3.3 prikazana je konkretna implementacija *DogsRepository*-a u podatkovnom sloju aplikacije. Implementirana je logika metoda *adopt*, *getDogById*, *getDogs* i *toggleFavorite.* Repozitorij prima varijablu *dogsApi* za dohvaćanje liste pasa. Navedene metode izvršavaju zahtjev na backend pozivanjem metode *fetchData()* koja vraća željene podatke. Svaki repozitorij nasljeđuje sučelje.



Slika 3.3. Implementacija klase *DogsRepositoryImp* u podatkovnom sloju.

Ono što još sadržava datoteka *domain* su objekti slučajeva korištenja i njihova sučelja. Sve implementacije slučajeva korištenja su apstraktne i ne ovise ni o prezentacijskom sloju niti o podatkovnom sloju. Na slici 3.4 prikazano je sučelje slučaja korištenja *GetMyDogsUseCase*. Sučelje sadrži definiciju funkcije *invoke* koja kao atribut prima varijablu *page* tipa *Int*, a kao rezultat vraća listu pasa.

Slika 3.4. Sučelje *GetMyDogsUseCase*

3.1.2. Podatkovni sloj

Ovaj sloj odgovoran je za dohvaćanje podataka. To može biti lokalno dohvaćanje, dohvaćanje iz memorije ili dohvaćanje preko oblaka. Sadrži implementaciju apstraktnih definicija koje se nalaze u sloju domene. Sadrži repozitorije i implementaciju izvora podataka te definiciju baze podataka. Slika 3.5 prikazuje implementaciju klase slučaja korištenja *GetMyDogsUseCaseImpl* koja nasljeđuje sučelje *GetMyDogsUseCase* koje je objašnjeno u prethodnom poglavlju. U klasi se definira metoda *invoke* koja vraća listu pasa. Lista pasa dohvaća se iz *dogsRepository* klase pozivanjem funkcije *getMyDogs*.

```
package com.cobeisfresh.domain.interaction.dogs.impl

import ...

class GetMyDogsUseCaseImpl(private val dogsRepository: DogsRepository) :
        GetMyDogsUseCase {
    override suspend operator fun invoke(page: Int) = dogsRepository.getMyDogs(getPagesQueryMap(page = page))
}
```

Slika 3.5. Implementacija klase *GetMyDogsUseCaseImpl*

U podatkovnom sloju nalaze se i definiraju API-a te maperi koji služe za pretvaranje mrežnih API modela u modele baza podataka i obrnuto. Na slici 3.6 prikazano je sučelje *DogsApiService* koje sadrži metode za dohvaćanje različitih api-a. Za dohvaćanje pasa koristi se zahtjev GET , a za ažuriranje tablice koristi se zahtjev POST. Svaki zahtjev vraća tip podatka karakterističan za taj zahtjev.

```
interface DogsApiService {

    @GET("pets/all")
    suspend fun getDogs(@QueryMap pageQueryMap: Map<String, Int>, @QueryMap searchQueryMap: Map<String, String>?,
                        @QueryMap filterQueryMap: Map<String, String>?, @Query("sort") sort: String?): Response<DogsResponse>


    @GET("pets/pet")
    suspend fun getDogById(@Query("id") dogId: String): Response<DogDetailsResponse>

    @GET("pets/favorites")
    suspend fun getMyDogs(@QueryMap queryMap: Map<String, Int>): Response<List<DogModelResponse>>

    @GET("pets/favorites/toggle")
    suspend fun toggleFavorite(@Query("id") dogId: String): Response<Unit>

    @POST("me/adopt")
    suspend fun adoptDog(@Body request: AdoptDogRequest): Response<Unit>
}
```

Slika 3.6. *DogsApiService* sučelje

3.1.3. Prezentacijski sloj

Prezentacijski sloj specifičan je za Android. Sadrži fragmente, modele prikaza, adaptere, aktivnosti itd. Također sadrži lokator, uslugu za upravljanje ovisnostima. U ovom sloju korišten je obrazac MVVM. Prezentacijski kod aplikacije razdvojen je na tri dijela: View, ViewModel i Model [16].

- Model – Ovaj sloj odgovoran je za apstrakciju podataka. Model i ViewModel rade zajedno kako bi dobili i spremili podatke.
- View – Svrha ovog sloja je informiranje ViewModel o radnji korisnika. Ovaj sloj promatra ViewModel i ne sadrži nikakvu logiku aplikacije.
- ViewModel – Izlaže one tokove podataka koji su važni za View. Služi kao poveznica između View-a i Modela.

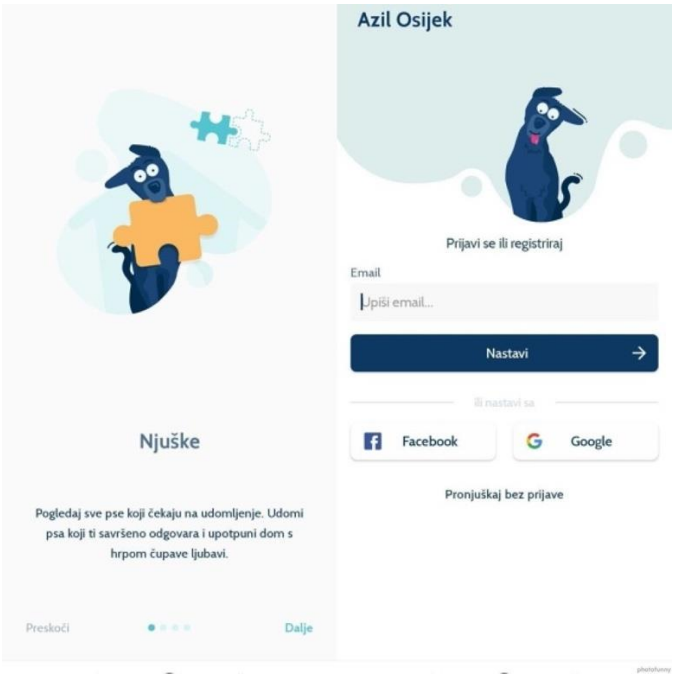
Na slici 3.7. prikazan je primjer ViewModela korišten u izradi aplikacije. U njemu se nalazi korisnički slučaj koji provjerava je li korisnik prijavljen. Ako je prijavljen dohvaćaju se svi psi te se oni prikazuje, u suprotnome otvara se zaslon koji korisniku daje do znanja da nije prijavljen.

```
class MyPetsViewModel(private val getMyDogsUseCase: GetMyDogsUseCase,
                      private val addRemoveFavoriteUseCase: AddRemoveFavoriteUseCase) :
        BaseViewModel<List<Dog>, MyPetsViewEffects>() {

    private var page = FIRST_PAGE

    init {
        resetPage( shouldReset: true)
        if (sharedPrefs.isLoggedIn()) {
            getMyDogs()
        } else {
            _viewEffects.value = ToggleNoDataLayout( visible: true)
        }
    }
```

Slika 3.7. ViewModel *MyPetsViewModel*

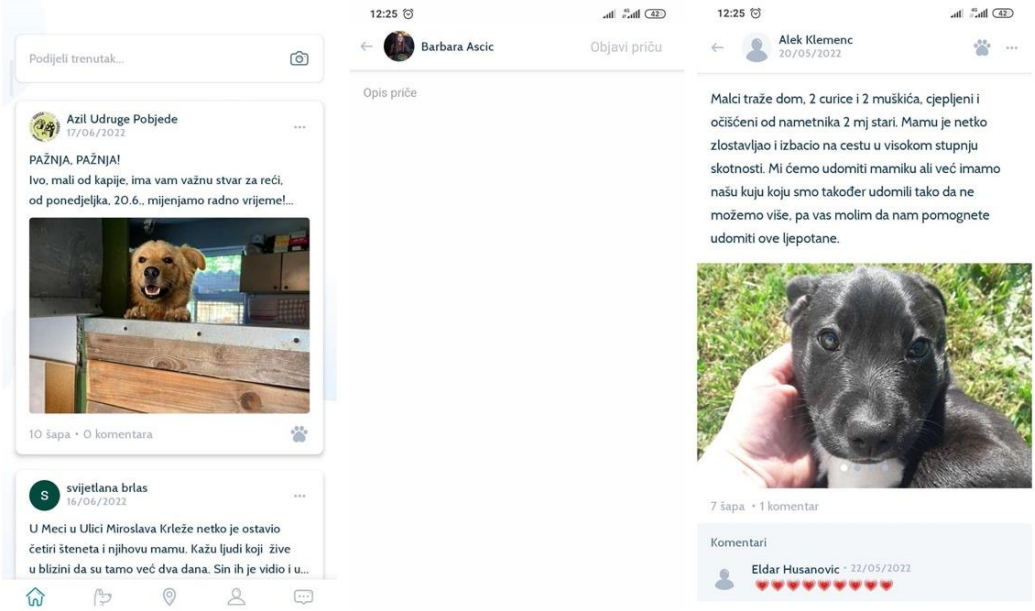
3.2.Korisničko sučelje aplikacije Azil Osijek

Aplikacija koja je odabrana za provedbu automatiziranih testova je Azil Osijek čija je svrha olakšati korisnicima postupak volontiranja i udomljavanja štićenika Azila. Može se pronaći na Trgovina Play-u ukoliko se koristi Android mobilni uređaj ili na App trgovini ukoliko se koristi iOS mobilni uređaj. Za provedbu automatiziranih testova koristit će se Android aplikacija. Prilikom pokretanja aplikacije otvara se „onboarding“ zaslon koji sadrži četiri etape (Slika 3.1). Nakon što se prođu „onboarding“ zasloni otvara se zaslon za prijavu u aplikaciju. Moguće je prijaviti se putem društvene mreže Facebook ili putem g-maila, no postoji i mogućnost ulaska u aplikaciju bez prijave (Slika 3.8).



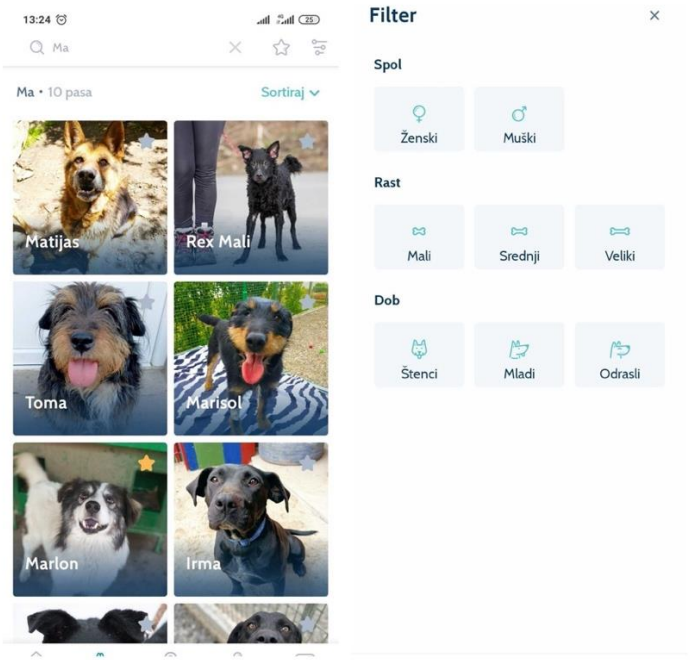
Slika 3.8. Početni zaslon i login aplikacije Azil Osijek

Aplikacija sadrži pet glavnih zaslona. Prvi zaslon „Novosti“ otvara se nakon što se korisnik prijavi u aplikaciju. Zaslon „Novosti“ prikazan je kao neka društvena mreža gdje korisnik može objavljivati svoj sadržaj ili gledati sadržaj drugih korisnika, taj sadržaj može komentirati, kliknuti ikonu šape kako bi rekao da mu se nešto sviđa itd. (Slika 3.9).



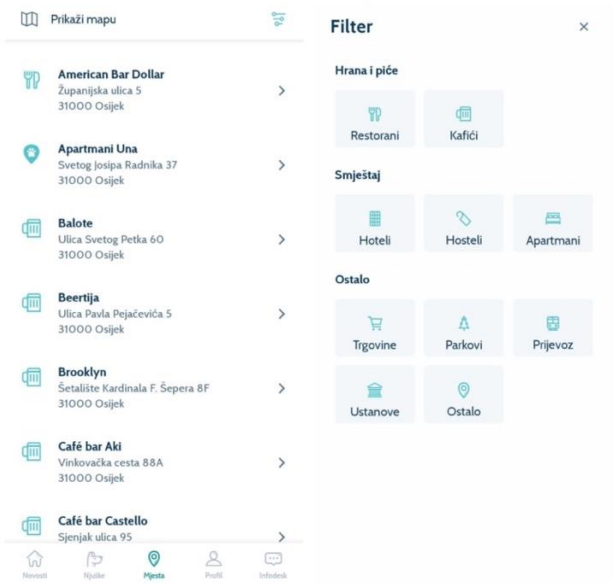
Slika 3.9. Zaslon „Novosti“

Zaslon „Njuške“ sadrži popis svih pasa u azilu te se mogu sortirati nasumično, abecedno ili vremenski, mogu se pretražiti po izrazu, filtrirati po spolu, rastu i dobi te označiti kao favoriti (Slika 3.10).



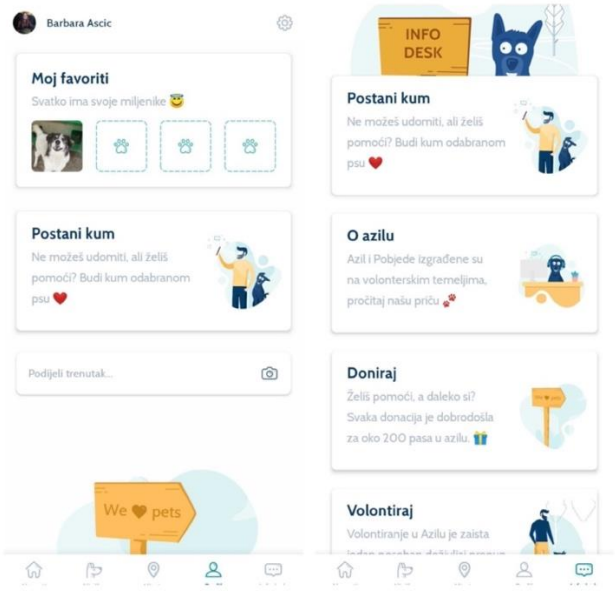
Slika 3.10. Zaslon „Njuške“

Zaslon „Mjesta“ prikazuje mjesta koja su prijateljski nastrojena prema kućnim ljubimcima i koje im dozvoljavaju ulazak. Također se mogu filtrirati prema vrsti objekta, smještaja i ostalom kao što su parkovi, trgovine, ustanove i drugo (Slika 3.11).



Slika 3.11. „Profil“ zaslon

Zaslon „Profil“ sadrži najvažnije informacije korisnika kao što su favoriti i objave. Zaslon „Infodesk“ sadrži informacije o Azilu, načinu kako postati kum, kako donirati i kako volontirati (Slika 3.12).



Slika 3.12. Zasloni „Profil“ i „InfoDesk“

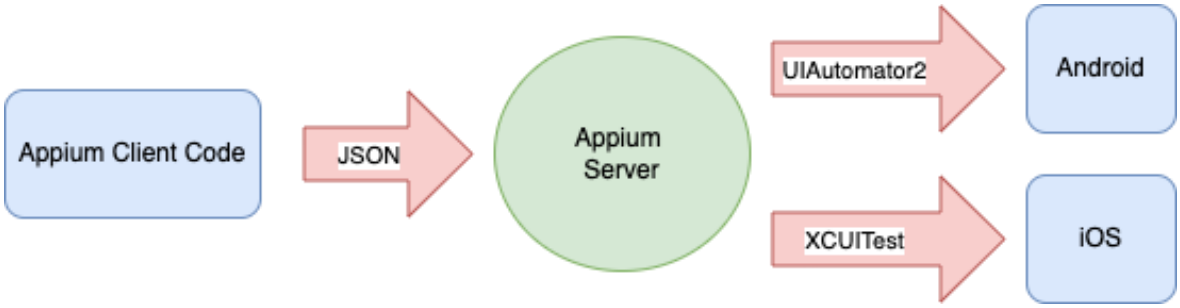
3.3. Testna okolina

Za izradu automatiziranih testova korišteno je prijenosno računalo MacBook Pro s procesorom 2,6 GHz 6-Core Intel Core i7. Prostorija u kojoj se odvijalo testiranja je laboratorij na fakultetu te radna soba. U poglavlju 3.2.1 upoznaje se s programskim alatom Appium i Cucumber koji su korišteni za provedbu automatiziranih testova. U poglavlju 3.2.2 opisan je jezik Java koji se koristi U poglavlju 3.2.2 opisuje se razvojno okruženje ItelliJ IDE CE u kojem će se pisati automatizirani testovi. U iduća dva pod poglavlja objašnjena je uloga virtualnih uređaja i alat Appium Inspector.

3.3.1. Appium

Korišten je alat Appium, alat otvorenog koda koji omogućuje provedbu automatiziranih testova na različitim platformama kao što su Android, iOS i Windows. Također omogućuje testiranje na fizičkim uređajima ili virtualnim uređajima poput emulatora i simulatora. Appium podržava sve jezike koji imaju Selenium biblioteku poput jezika: Java, Objective-C, PHP, JavaScript, C#, Python itd. Appium je HTTP poslužitelj napisan korištenjem Node.js-a i pokreće iOS i Android sesije koristeći WebDriver JSON. Prema tome, prije inicijalizacije poslužitelja

Appium, Node.js mora biti unaprijed instaliran na sustav. Appium ne dozvoljava testiranje Android inačice koja je manja od 4.2. Slika 3.13 preuzeta iz [3], prikazuje arhitekturu Appiuma.

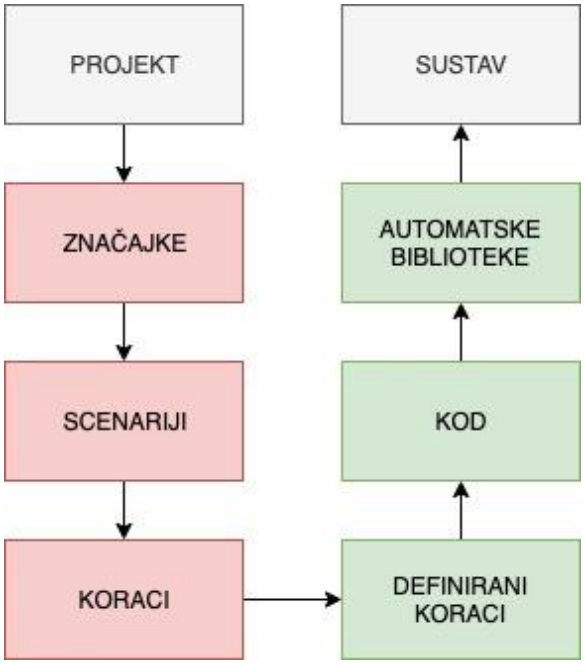


Slika 3.13. Arhitektura Appium-a [3]

Appium Client Code predstavlja kod koji se može pisati u bilo kojem od jezika koji su prethodno navedeni. Appium omotava kod i konvertira ga u JSON format te ga šalje Appium Severu. Appium Server prihvaća samo JSON objekte. Ima sposobnost tumačenja JSON objekata i razumije koje se akcije trebaju izvesti na mobilnim uređajima te ih prosljeđuje mobilnim uređajima/simulatorima.

3.3.2. Cucumber

Cucumber je alat koji podržava BDD (Behaviour-Driven-Development). BDD je proširenje Test-Driven-Developmenta (TDD) koje naglašava razvoj značajki temeljenih na korisničkoj priči i pisanju koda koji pruža rješenja stvarnih problema. Cucumber čita izvršene specifikacije napisane u običnom tekstu i potvrđuje da programska podrška radi ono što te specifikacije kažu. Specifikacije se sastoje od više primjera ili scenarija [17]. Kako bi Cucumber mogao razumjeti scenarije mora slijediti osnovna pravila sintakse koja se zovu Gherkin. *Step definition* (definicije koraka) povezuju Gherkin s programskim kodom, dakle definiranje koraka povezuje specifikaciju s implementacijom. Arhitektura Cucumber prikazana je dijagramom na slici 3.14 [18].



Slika 3.14. Arhitektura Cucumber-a

Crveno označeni elementi predstavljaju poslovnu okolinu, zeleno označeni elementi predstavljaju tehnološku okolinu. Poslovnu okolinu određuju projekt menadžeri i testeri dok tehnološku okolinu određuju programeri i testeri.

3.3.3. Java

Java je programski jezik opće namjene koji se koristi za razvoj mobilnih aplikacija. Kako tvrtka Oracle navodi jezik Java radi na više od tri milijarde uređaja diljem svijeta što ju čini jednim od najpopularnijih programskih jezika. Javu je izvorno razvio James Gosling u svibnju 1995. godine. Jezik se u početku zvao 'Oak', također se zvao 'Green' da bi se zatim preimenovao u 'Java'. Karakteristike Java programskog jezika:

- neovisan o platformi – Moguće je pisati kod na jednom operativnom sustavu, a pokretati ga na drugom operativnom sustavu bez ikakvih izmjena.
- objektno orijentiran – Java je objektno orijentirani jezik, što pomože da kod postane fleksibilniji i višekratno upotrebljiv.
- brzina – Dobro optimiziran kod Jave gotovo je jednako brz kao jezici niže razine poput C++, a mnogo je brži od Pythona i PHP-a.

2019. godine Java je bila jedan od najpoznatijih jezika koji se koristio u razvoju mobilnih aplikacija

3.3.4. IntelliJ IDEA CE

U IDE - Integriranom razvojnom okruženju razvija se sav kod. U praksi ljudi najčešće koriste Eclipse i IntelliJ IDEA. Za potrebe ovog rada koristio se IntelliJ IDEA CE. To je razvojno integrirano okruženje za jezike JVM, dizajnirano za maksimiziranje produktivnosti programera. Razvija ga i održava JetBrains. Omogućuje brz pristup svim značajkama i integriranim alatima te ima širok raspon mogućnosti prilagodbe. Prema [19] neke od najvažnijih značajki su:

- pametno dovršavanje koda
- dovršenje lančanog koda
- dovršenje statičnog člana
- otkrivanje duplikata
- uredničko-centrično okruženje
- inspekcije i brzi popravci
- prečaci za sve
- ugrađeni program za ispravljanje pogrešaka

Potrebno je postaviti varijable okoline kao što je prikazano na slici 3.15. Varijable se postavljaju tako da se u terminalu upiše naredba *nano ~/.bash_profile*. Kako bi se osiguralo da je upisana dobra putanja JAVA_HOME varijable može se provjeriti naredbom *source ~ /.bash_profile*

```
export ANDROID_HOME=/Users/{YourMacUserName}/Library/Android/sdk
export ANDROID_SDK_ROOT=/Users/{YourMacUserName}/Library/Android/sdk
export ANDROID_AVD_HOME=~/.android/avd
export JAVA_HOME=/Library/Java/JavaVirtualMachines/corretto-15.0.2/Contents/Home
export PATH=${JAVA_HOME}/bin:$PATH
export PATH=${PATH}:${ANDROID_HOME}/tools
export PATH=${PATH}:${ANDROID_HOME}/platforms-tools
export PATH=${PATH}:${ANDROID_HOME}/build-tools/27.0.1
export PATH=${PATH}:${JAVA_HOME}
```

Slika 3.15. Enviroment varijable

Potrebno je instalirati maven naredbom *mvn -v*. Apache Maven [20] je alat za upravljanje proizvodima programske podrške. Na temelju koncepta objektnog modela projekta (POM), Maven upravlja izgradnjom projekta, izvješćivanjem i dokumentacijom.

3.3.5. Appium Inspector

Appium Inspector je alat koji se koristi za postupak identifikacije elemenata korisničkog sučelja mobilne aplikacije. Radi na stvarnim uređajima, simulatorima ili emulatorima. On radi kao Appium klijent (poput Webdriver-a) s korisničkim sučeljem. Prije pokretanja Appium Inspectora potrebno je lokalno pokrenuti Appium Server. Kada Appium Server pokrene skriptu za automatizirano testiranje, Appium Inspector može identificirati elemente korisničkog sučelja svake aplikacije koja se testira. Temeljna struktura Appium Inspectora je osigurati da se otkrije svaki vidljivi element aplikacije kada se razvijaju testne skripte.

3.3.6. Virtualni uređaji

Virtualni uređaj simulira najvažnije značajke mobilnih pametnih uređaja. Oni služe za pokretanje aplikacija kako bi se prikazalo kako će aplikacija raditi na stvarnom uređaju. Postoje dvije vrste virtualnih uređaja: simulatori i emulatori [21]. Za testiranje Android aplikacije koriste se emulatori. Sva testiranja biti će izvršena na emulatorima Nexus i Pixel koji se pokreću pomoću Android Studij-a.

3.4.Testni plan i testni slučaj

Testni plan i scenarij ovisi o brojnim čimbenicima [3] uključujući razinu, ciljeve i objekte testiranja koje se planiraju provesti. Pisanje plana, priprema i odabir testnih strategija odvija se tijekom razdoblja planiranja cjelokupnog projekta iako je moguća izmjena tijekom razvoj projekta nakon što se dobije više informacija ili dođe do nekih izmjena.

3.4.1. Testni plan

Testni plan je projektni plan testiranja koje se mora izvršiti, on se može razlikovati s obzirom na veličinu i kompleksnost projekta. Testni plan aplikacije Azil Osijek sadrži podatke o:

1) Tip testiranja: regresijsko testiranje
2) Funkcionalnosti koje se testiraju – prikazane u tablici 3.1.

Tab. 3.1. Funkcionalnosti aplikacije Azil Osijek

Ime modula	**Opis**
Prolazak kroz onboarding	Korisnik treba moći proći kroz tri zaslona onboardinga
Preskok onboarding-a	Korisnik treba moći preskočiti onboarding
Registracija	Korisnik se treba moći registrirati svoj račun
Prijava	Korisnik se treba moći prijaviti u aplikaciju
Ulazak u aplikaciju bez prijave korisnika	Korisnik treba moći ući u aplikaciju bez prijave
Otvaranje navigacijske trake	Korisnik klikom na bilo koju stavku navigacijske trake treba moći otvoriti potreban zaslon
Sortiranje pasa abecednim redom	Korisnik treba moći sortirati pse abecednim redom
Otvaranje Google karte	Korisnik treba moći vidjeti mjesta na Google karti

3) Rizici i problemi testiranja – prikazani u tablici 3.2

Tab. 3.2. Rizi i problemi prilikom testiranja

Rizik	**Rješenje**
Tester nema sve potrebne vještine za provedbu automatiziranih testiranja	Isplanirati tečaj za usavršavanje automatiziranog testiranja
Raspored projekta je pretijesan, teško je završiti projekt na vrijeme	Postaviti prioritete testiranja za svaku aktivnost testiranja

4) Logistika

Tko provodi testiranje? Testiranje provodi jedan tester koji je osiguravatelj kvalitete i koji piše automatizirane testove.

Kada se izvode testovi? Testovi se izvode kada je aplikacija dovršena, kada su definirane funkcionalnosti koje se testiraju

5) Resursi

a) Resursi sustava – tablica 3.3.

Tab 3.3. Resursi potrebni za vrijeme testiranja

Resursi	**Opis**
Računalo	Macbook Pro, 2.6 GHz 6 Core Intel Core i7
Indeks mreža	Internetska brzina mora biti najmanje 5 Mb/s
Alat za testiranje	Razviti alat za testiranje koji automatski generira rezultate testiranja

b) Ljudski resursi – prikazani u tablici 3.4.

Tab. 3.4. Ljudski resursi potrebni za testiranje

Uloga	**Zadatak**
Voditelj testiranja	Upravlja cijelim projektom Nabavlja odgovarajuće resurse
Ispitivač/Tester	Opisuje testne alate/arhitekturu za automatizaciju Izvršava testove, bilježi rezultate i prijavljuje nedostatke
Programer	Implementira testne slučajeve, testne programe itd.

6) Raspored i estimacija

Za potrebe ovog rada nije bilo potrebno napraviti raspored i estimaciju jer sva testiranja obavlja jedna osoba. Procijenjeno vremena testiranja je 60 sati.

7) Rezultati ispitivanja – dokumenti, prikazani su u tablici 3.5.

Tab. 3.5. Dokumenti potrebni za testiranje

Prije faze testiranja	**Tijekom ispitivanja**	**Nakon završetka ciklusa testiranja**
Dokument s planovima ispitivanja	Alati za testiranje	Rezultati testiranja/izvješća
Dokument testnih slučajeva	Virtualni uređaji	

3.4.2. Testni slučajevi

Prema [22], testni slučajevi se pišu prema unaprijed definiranim zahtjevima proizvoda te se dokumentiraju u specifikacijama testnih slučajeva. Oni pružaju potrebnu osnovu za provedbu i izvođenje testova te su zapravo skup radnji koji se izvode za provjeru određene značajke ili specifikacije aplikacije. Stoga su specifikacije testnih slučajeva temeljni dio procesa testiranja mobilne aplikacije.

Testni slučajevi moraju biti detaljizirani kako bi se mogao provjeriti rezultat i kako bi se moglo utvrditi je li dobiveni rezultat ispravan odgovor sustava. Ako se testovi trebaju automatizirati, alat za testiranje treba točno znati s čime usporediti izlaz sustava [3]. U slijedećim tablicama raspisani su neki od testnih slučajeva koji su se automatizirali testovima.

Tablica 3.6 prikazuje testni slučaj za registraciju. Testni slučaj sadrži korake: otvaranje aplikacije, preskakanje onboardinga, unos adrese e-pošte i druge.

Tablica 3.6. Testni slučaj- registracija

	Opis koraka	**Očekivani rezultat**	**Dobiveni rezultat**	**Prolaz/Pad**
1.	Otvori aplikaciju Azil Osijek	Aplikacija se otvorila	Očekivan	Prolaz
2.	Preskoči onboarding	Otvara se zaslon za prijavu/registraciju	Očekivan	Prolaz
3.	Unesi adresu e-pošte.	Email se može unijeti	Očekivan	Prolaz
4.	Klikni na *Continue* gumb	Otvara se zaslon za unos lozinke i imena	Očekivan	Prolaz
5.	Unesi lozinku	Lozinka se može unijeti	Očekivan	Prolaz
6.	Unesi ime	Ime se može unijeti	Očekivan	Prolaz
7.	Klikni na *Continue registration* gumb	Korisnik je registriran.	Očekivan	Prolaz

Tablica 3.7 prikazuje testni slučaj za otvaranje Google karte na zaslonu Mjesta. Kao i prethodna tablica sadrži podatke o opisu koraka, očekivanom rezultatu, dobivenom rezultatu i prolaznosti testnog koraka.

Tablica 3.7. Testni slučaj – otvaranje Google karte na zaslonu Mjesta

	Opis koraka	**Očekivani rezultat**	**Dobiveni rezultat**	**Prolaz/Pad**
1.	Otvori aplikaciju Azil Osijek	Aplikacije se otvorila	Očekivan	Prolaz
2.	Preskoči onboarding	Otvara se zaslon za prijavu/registraciju	Očekivan	Prolaz
3.	Klikni na *without login* gumb	Otvara se zaslon *Novosti*	Očekivan	Prolaz
4.	Klikni na *Mjesta* na navigacijskoj traci	Otvara se zaslon *Mjesta*	Očekivan	Prolaz
5.	Klikni na *showMap* gumb	Otvara se Google karta s prikazanim mjestima	Očekivan	Prolaz

Tablica 3.8 prikazuje testni slučaj za sortiranje pasa abecednim redom. Testni slučaj je pao jer nije dobiven očekivani rezultat testa.

Tablica 3.8. Testni slučaj – sortiranje pasa abecednim redom

	Opis koraka	**Očekivani rezultat**	**Dobiveni rezultat**	**Prolaz/Pad**
1.	Otvori aplikaciju Azil Osijek	Aplikacija se otvorila	Očekivan	Prolaz
2.	Preskoči onboarding	Otvara se zaslon za prijavu/registraciju	Očekivan	Prolaz
3.	Klikni na *withoutlogin* gumb	Otvara se zaslon Novosti	Očekivan	Prolaz
4.	Klikni na Njuške na navigacijskoj traci	Otvara se zaslon Njuške	Očekivan	Prolaz
5.	Klikni na *sort* gumb	Otvara se modal za odabir vrste sortiranja	Očekivan	Prolaz
6.	Klikni na *aplhabetically* gumb	Modal se zatvorio	Očekivan	Prolaz
7.	Klikni na prvog psa	Zaslon s prvim psom se otvorio. Ime psa počinje sa slovom „a“	Neočekivan	Pad

3.5.Veza između testera i programera

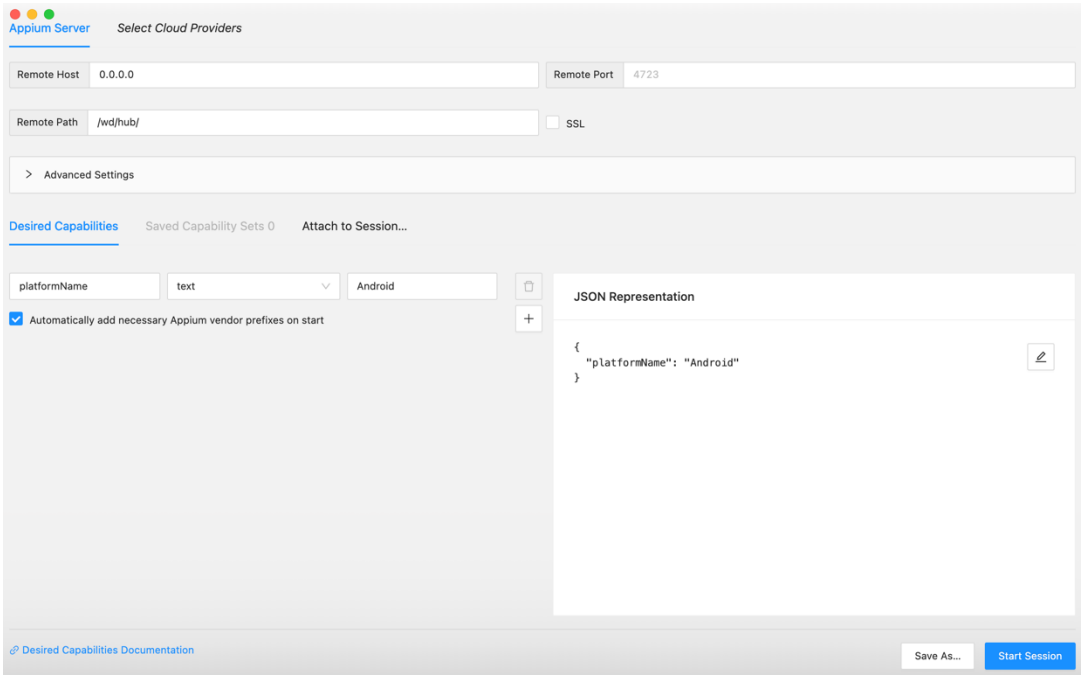
Niti jedan projekt nije bio uspješan zbog alata, proračuna, koda ili infrastruktura već zbog ljudi koji su radili na tom projektu i učinili taj projekt uspješnim, a kako bi nešto uspjelo potreban je rad cijelog tima, a ne pojedinca. Kada dođe do suradnje i komunikacije između testera i programera projekti se brže završavaju i promiče se vrhunska kvaliteta agilnog načina rada [23]. Testeri se kao i programeri trebaju uključiti u razvojnu fazu projekta, time se izgrađuju čvrsti temelji za daljnju suradnju te to uključuje:

- učenje o klijentima
- dogovor oko zajedničkih podataka za testiranje te dogovor o tome kako izgleda gotov proizvod
- kreiranje zajedničko razumijevanje „gotovog“ proizvoda
- dizajniranje testova u dogovoru s programerom
- uklanjanje koncepta „mi“ protiv „njih“ između programera i testera
- fokusiranje na zajednički rad cjelina povećava šansu za uspješnom isporukom proizvoda

4. PROVEDBA TESTIRANJA

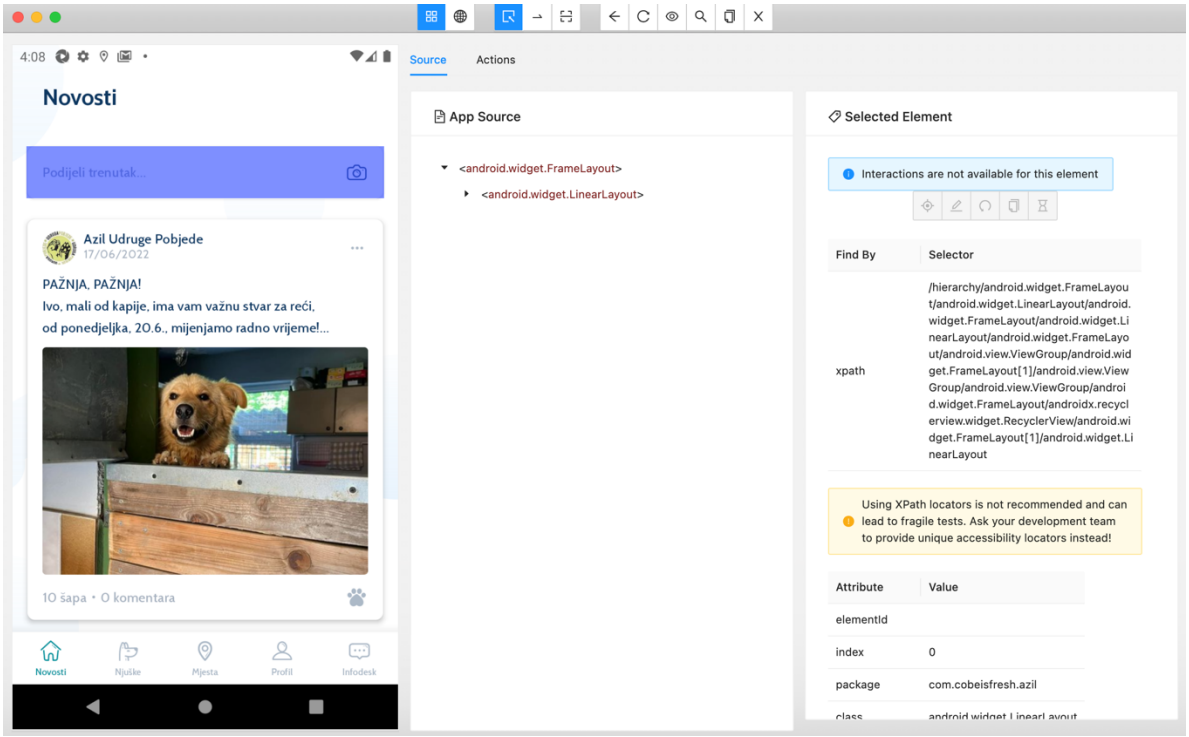
4.1. Korištenje Appium Inspectora

Kako bi se identificirali elementi korisničkog sučelja aplikacije Azil Osijek koristiti se alat Appium Inspector. Prvo se pokreće emulator na kojem je instalirana aplikacija Azil Osijek, zatim se u terminalu pokreće naredba *appium*, nakon toga se pokreće program Appium Inspector. Postavlja se varijabla *platformName* kao Android i pokreće se sesija. (Slika 4.1).



Slika 4.1. Pokretanje Appium Inspectora

Kada se pokrene Appium Inspector mogu se pretraživati elementi i pronaći varijable poput *id*, *xpath*, indeks, test itd. koje će se koristiti u pisanju testova u Appiumu (Slika 4.2). Kada se promijeni zaslon u aplikaciji za ažuriranje zaslona u Appium Inspectoru potrebno je kliknuti na gumb za osvježavanje.



Slika 4.2. Pretraživanje elemenata

4.2.Postavljanje Appiuma

Prije nego se počnu pisati testovi u Appium-u, potrebno je skinuti ovisnosti za Appium Maven sa stranice https://mvnrepository.com/. Sve ovisnosti dodaju se u pom.xml datoteku. Osim ovisnosti za maven, za potrebe ovog projekta dodane su ovisnosti za Javu, Json, Cucumber, itd. Nakon što se ovisnosti dodaju potrebno je učitati promjene koje su napravljene. Primjer jedne ovisnosti može se vidjeti na slici 4.3.

```
1. <dependency>
2.             <groupId>io.appium</groupId>
3.             <artifactId>java-client</artifactId>
4.             <version>7.6.0</version>
5. </dependency>
```

Slika 4.3. Ovisnosti u pom.xml datoteci

TestNG je okruženje koje se koristi u Javi i bit će korišten za provjeru valjanosti testova. Treba se dodati kao i appium kroz datoteku pom.xml. U testng.xml datoteci nalaze se vrste virtualnih uređaja odnosno emulatora. Potrebno je navesti naziv platforme, ime emulatora, ime uređaja i vrste portova što je prikazano na slici 4.4.

```
<test name="Nexus_5_API_30">
  <parameter name="platformName" value="Android"/>
  <parameter name="udid" value="emulator-5554"/>
  <parameter name="deviceName" value="Nexus_5_API_30"/>
  <parameter name="systemPort" value="10000"/>
  <parameter name="chromeDriverPort" value="10001"/>
  <classes>
   <class name="com.qa.runners.Nexus_5_API_30_TestRunner"/>
  </classes>
</test>
```

Slika 4.4. testng.xml datoteka

Potrebno je dodati apk Azil-Osijek aplikacije u datoteku *apps*.

4.3.Klase Pages

U datoteci *pages* nalaze se sve klase Jave u kojima su definirani Android elementi koristeći anotaciju *@FindByAndroid* i kreirane metode koje se izvode na tim elementima. Osnovna klasa koju nasljeđuju sve klase *Page* je klasa *BasePage*. U klasi *BasePage* događa se inicijalizacija Android pogonskog programa i definirane su pomoćne metode koje se koriste u klasama *Page*. Jedna od pomoćnih metoda je *sendKeys()* koja kao atribut prihvaća mobilni element i *String* varijablu te služi za slanje varijable tipa *String* mobilnom elementu. Metoda *sendKeys()* koristi se za slanje podatka, primjerice upisivanje e-maila, lozinke, imena u input elemente. Još neke od pomoćnih metoda koje su korištene u ovom radu su *click(), waitForVisibility(), getAttribute(), scrollToElement()* itd. Na slici 4.5 nalaze se metode *sendKeys()* i *click()*.

```
public void click(MobileElement e) {
        waitForVisibilityI;
        e.click();
    }

public void sendKeys(MobileElement e, String txt) {
        waitForVisibilityI;
        e.sendKeys(txt);
   }
```

Slika 4.5. Metode *click()* i *sendKeys()*

Kako se aplikacija Azil Osijek sastoji od više zaslona, za svaki zaslon potrebno je kreirati klasu *Page*. Klasa *Page* treba sadržavati sve Android elemente koji će se koristiti na tom zaslonu u testiranju. Za pronalazak Android elementa koristi se Appium Inspector, elementi se mogu pretraživati po *id*-u, *xpath*-u, atributima itd. Preporučljivo je u dogovoru s programerom koristiti pretraživanje po id-u. Ukoliko id nije prisutan, u ovom radu korišteno je pretraživanje po xpath-u. Za inicijalizaciju Android elemenata korištena je anotacija @AndroidFindBy koja služi za lociranje jednog mobilnog elemenata ili popisa mobilnih elemenata. Ono omogućuje brzo i jednostavno stvaranje objekata *Page*. Nije potrebno inicijalizirati elemente koji se nalaze na zaslonu a neće se koristiti u testiranju. Na slici 4.6 prikazan je primjer inicijalizacije gumba za registraciju i input polja za ime na zaslonu *Registracija*.

```
1. @AndroidFindBy(id=”com.cobeisfresh.azil:id/userFullNameInput”)
2.   MobileElement nameInput;
3.
4.  @AndroidFindBy(id=”com.cobeisfresh.azil:id/continueRegister”)
5.   MobileElement continueRegistrationButton;
```

Slika 4.6. Inicijalizacija gumba za registraciju i input polja za unos imena

Zbog kompleksnosti pojedinih zaslona bilo je potrebno jedan zaslon rastaviti na više objekata *Page*. Tako postoji objekt *RegistrationLoginPage* koji se pojavljuje i prilikom registracije i prijave dok se objekt *LoginPageAzi*l pojavljuje samo prilikom prijave, a objekt *RegistrationPage* samo prilikom registracije. Napravljeni su objekti za zaslone Onboarding, Novosti, Njuške, Mjesta, Profil i Infodesk.

4.4.Step definition

Datoteka *Step definition* sadrži klasu *Hooks* i klase *StepDefinition*. U klasi *Hooks* definirani su blokovi koda koji se izvode prije i poslije svakog scenarija. Mogu se definirati bilo gdje u programu koristeći metode *@Before* i *@After*. Ono omogućuje bolje upravljanje tijeka rada koda i pomaže smanjiti dupliciranje koda. Metoda *initialize()* ima anotaciju *@Before* te se poziva prije svakog scenarija. Ona gasi pogonski program ukoliko nije prethodno ugašen i postavlja ga na vrijednost *null*, pokreće server, inicijalizira i pokreće pogonski program. Metoda *quit()* ima anotaciju *@After* te se poziva nakon svakog scenarija. Ona zatvara pogonski program i zatvara server. Objekt *StepDefintion* služi za preslikavanje koraka testnih slučajeva u datoteci *feature*. On izvršava korake na aplikaciji koja se testira i provjerava rezultate u

odnosu na očekivane rezultate. Da bi se „definirani koraci“ izvršili moraju odgovarati danoj komponenti u datoteci *feature*.

Treba izbjegavati pisanje sličnih koraka jer mogu dovesti do konflikata, koraci se mogu koristiti u različitim datotekama *feature*. Tako se primjerice korak 'I press skip button' koristi na onboardingu, prijavi, pretraživanju pasa itd. Jedna klasa *StepDefinition-a* sadrži funkcije i anotacije Gherkin uz njih. Koraci su 'ljepilo' za stvarni kod, koriste se za skrivanje detalja implementacije pozivanjem nekoliko pomoćnih metoda iz jedne definicije koraka. Na slici 4.7 dan je primjer objekta *RegistrationStepDef*.

```
1. public class RegistrationStepDef {
2.
3.     @When("^I press Continue registration button$")
4.     public void iPressContinueRegistrationButton(){
5.         new RegistrationPage().pressContinueRegistrationButton();
6.     }
7.
8.     @When("^I Enter name as \"([^\"]*)\"$")
9.     public void iEnterName(String name) {
10.        new RegistrationPage().enterName(name);
11.    }
12.
13.    @When("^I Scroll to ContinueRegistration button$")
14.    public void iScrollToContinueRegistrationButton() {
15.       new RegistrationPage().scrollToContinueRegistrationButton();
16.    }
17.
18.}
```

Slika 4.7. RegistrationStepDef klasa

Klasa koja je prikazana na slici 4.7 sadrži tri funkcije. Svaka funkcija prvo sadrži anotaciju s ključnom riječju 'When' što predstavlja radnju koja će se izvršavati. Nakon ključne riječi slijedi prirodnim jezikom napisan korak koji će se izvoditi. Zatim se navodi ime funkcije čije je ime preporučljivo odabrati kao i opis koraka. Potom se poziva klasa *Page* određenog zaslona na kojoj se poziva funkcija koja izvodi određenu radnju.

4.4.1. Gherkin

Svi testni slučajevi generiranu su u meta jeziku Gherkin-u. To je jezik koji Cucumber razumije. Prema [24], Gherkin omogućuje opis ponašanja programske podrške bez detalja kako se to

ponašanje implementira. Datoteke Gherkin-a podijeljene su na značajke koje sadrže popis scenarija, scenarija koji se sastoji od popisa koraka i koraka koji počinju jednom od ključnih riječi. Ključne riječi koje Gherkin koristi su: *Feature, Scenario, Scenario Outline, When, And, Then* itd.

- Feature – Opisuje trenutnu testnu skriptu koja se mora izvršiti.
- Scenario – Opisuje korake i očekivani ishod za određeni testni slučaj.
- Scenario Outline – Koristi se za ponavljajuće scenarije koji slijede isti obrazac ponašanja na različitim skupovima podataka. Scenario Outline se koristio u pisanju koraka za provedbu testa registracije.
- Given – Određuje kontekst teksta koji će se izvršiti, predstavlja preduvjet koji se mora ispuniti kako bi se test mogao izvesti.
- When – Navodi radnju testiranja koja se mora izvršiti.
- Then – Očekivani ishod testa.
- Examples – Koristi se ako se koristi Scenario Outline, predstavljaju skupove podataka nad kojim se izvode testovi.

4.5.Runners

Klasa MyRunnerTest sadrži dvije funkcije: *intialize()* koja ima anotaciju *@BeforeClass* i izvodi se prije svakog testa i funkciju *quit()* koja ima anotaciju *@AfterClass* te se izvodi nakon svakog testa. Za provedbu automatiziranih testova koriste se dva emulatora stoga postoje dvije klase *Runner*. Klasa *Nexus_5_API_30_TestRunner* služi za pokretanje emulatora Nexus, klasa *Pixel_2_API_30_RunnerTest* služi za pokretanje emulatora Pixel. Najbitnija stavka koja se nalazi u klasama *Runner* je CucumberOptions plugin, te je prikazana na slici 4.8.

```
1. @CucumberOptions(
2.         plugin = {“pretty”
3.         , “html:target/Nexus_5X/cucumber.html”
4.         , “summary”
5.         , “me.jvt.cucumber.report.PrettyReports:target/Nexus5/cucu
   mber-html-reports”}
6.         ,features = {“src/test/resources”}
7.         ,glue = {“com.qa.stepdef”}
8.         ,tags = “@test”
9.         ,dryRun=false
10.        ,monochrome=true
11. )
```

Slika 4.8. Klasa Runner – Cucumber plugin

@CucumberOptions se može koristiti za pružanje dodatne konfiguracije runner-u. Za formatiranje korišten je plugin 'pretty' i 'html'. Što bi značilo da će izvješća automatiziranih testova biti formatirani i spremljeni u dokumente *cucumber.html* i *cucumber-html-reports* Korišten je i plugin 'summary' koji služi za ispisivanje isječaka koda koji nedostaju. Ono što se još mora dodati je putanja datoteka *features* i *glue* što predstavlja 'ljepilo' za korake izvršavanja. Ono što je opcionalno za predati je vrijednost *tags*. *Tags* ili oznake se u Cucumberu koriste za povezivanje testova, poput *smoke* testiranja, regresije itd., s određenim scenarijima. Oznake služe kao pomoć pri izvršavanju testova kada datoteka značajki sadrži mnogo scenarija, pomoću njih mogu se pripremiti izvješća za određene scenarije pod istom oznakom. Također oznaka omogućuje preskakanje scenarija tako što će se prije navođenja imena oznake koristiti znak ~. Oznake se stavljaju kao anotacije prije naziva scenarija u datoteci značajki. Za korištenje Junita za izvršavanje Cucumber scenarija treba se dodati ovisnost u datoteku pom.xml.

4.6.Datoteke feature

Datoteka *feature* ili datoteka značajki sadrži opis testnog scenarija visoke razine na jednostavnom jeziku. Taj se jezik zove Gherkin i objašnjen je u prethodnom poglavlju. Značajka se može još objasniti kao parametar koji se koristi za testiranje zahtjeva korisnika. U značajci se može zvati bilo koji korak koji je definiran, koraci se ne moraju nalaziti u istim klasama, čak se i potiče ponovno korištenje koraka u što je moguće većoj mjeri kako bi se smanjilo dupliciranje koda. Na slici 4.9 prikazan je scenarij koji je lako čitljiv bilo kojoj osobi koja sudjeluje u procesu razvoja aplikacije.

```
1. Scenario Outline: Login with email
2.     Given I skip Onboarding
3.     And I tap on screen
4.     And I Enter email as “<email>”
5.     And I press Continue button
6.     And I Enter password as “<password>”
7.     And I press Continue button
8.     Then I should be on Novosti page with title “<title>”
9.
10.    Examples:
11.      | email | password | title|
12.      |  1999barbaraa@gmail.com| password | Novosti|
```

Slika 4.9. Značajka prijava korisnika

To je najbitnija stavka Cucumbera jer je jednostavan i pregledan, sakrivena je sva dodatna implementacija te nije vidljiva u scenarijima. Prvi scenarij služi za testiranje prijave u aplikaciju. Riječ Gherkin s kojom počinje test je Scenarion Outline jer je to scenarij koji prima podatke te sadrži podatke Example na kraju. Preduvjet izvršavanja je da je preskočen onboarding te se otvara zaslon za prijavu.

Scenarij imitira očekivano ponašanje korisnika, stoga će se u daljnjem opisu koristiti on kao izvršitelj radnje. Korisnik prvo mora kliknuti na zaslon kako bi mogao unijeti potrebne podatke. Nakon toga dolazi korak za unos e-maila koji se predaje kao '1999barbaraa@gmail.com'. Zatim korisnik pritišće gumb *'Continue'* kako bi otvorio novi zaslon za nastavak prijave. Ukoliko je unesena e-mail adresa koja ne postoji test će pasti jer će se otvoriti zaslon za registraciju. Nakon što je otvoren zaslon za nastavak prijave, korisnik unosi lozinku koja je 'password'. Potom korisnik treba kliknuti na gumb '*Continue'*. Nakon toga očekivano je da će se otvoriti zaslon Novosti te se time potvrđuje da je test uspješno prošao. Ukoliko test završi, a nije otvoren zaslon Novosti, test će pasti. Drugi test prikazan na slici 4.10 je sortiranje pasa abecednim redom.

```
1. Scenario Outline: Sort alphabetically
2.       When I skip Onboarding
3.       And I press withoutLogin button
4.       And I click on Njuske navigation
5.       And I click on sort button
6.       And I click on sort alphabetically button
7.       And I click on first dog
8.       Then First letter of dogs name should be “<firstLetter>”
9.       Examples:
10.          |firstLetter|
11.          | a |
```

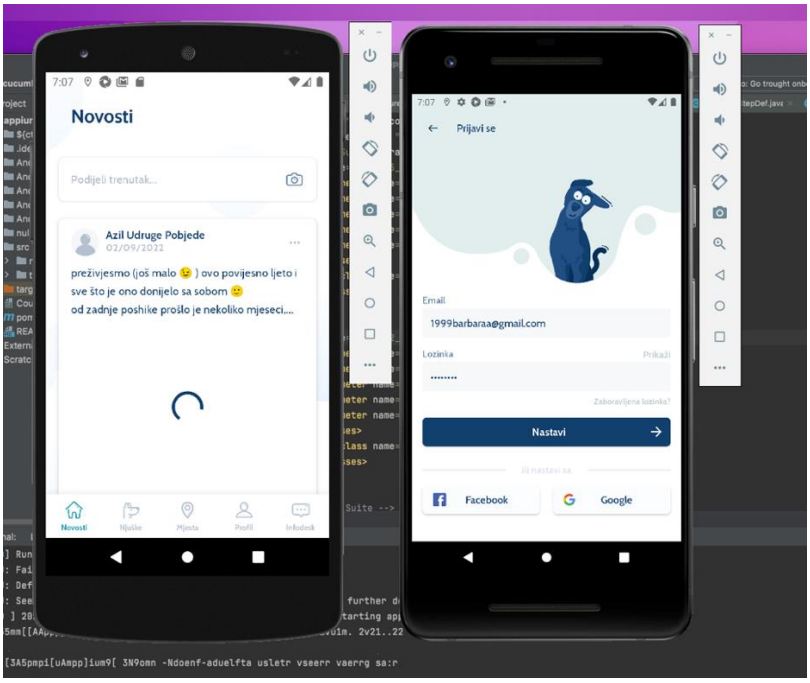
Slika 4.10. Značajka sortiranje pasa abecednim redom

Kao i prethodni scenarij preduvjet za izvršavanje je da se preskoči onboarding zaslon, još jedan preduvjet koji se mora ispuniti je ulazak u aplikaciju bez prijave kako bi se ubrzao proces automatiziranog testiranja. Nakon toga otvara se zaslon Novosti te se na navigaciji klikne na stavku 'Njuske'. Potom se treba kliknuti na gumb za sortiranje te odabrati abecedno sortiranje. Zatim se otvara zaslon 'Njuske' te korisnik treba kliknuti na prvog psa. Očekivano je da ime prvog psa počinje na slovo 'a' jer su psi poslagani abecednim redom i znamo da postoji pas čije

ime počinje na slovo 'a'. Kao i u prethodnom scenariju sakrivena je sva implementacija u klasama *StepDefiniton* i *Page*.

4.7. Pokretanje testova

Testovi se pokreću preko terminala naredbom: „*mvn clean test -Dsurefire.suiteXmlFiles= /Users/barbara/Desktop/appium-cucumberbdd-testng/src/test/resources/testng.xml*“. Gdje *mvn clean* test predstavlja pokretanje *mavena*, a ostatak lokaciju datoteke projekta *testng.xml* koja se pokreće. Nakon što je pokrenuta naredba pokreću se emulatori Pixel i Nexus koji vrte testove koji su im zadani. Koji testovi će se pokrenuti ovisi o oznaci koja se nalazi uz scenarij. Paralelno testiranje omogućeno je dodavanjem *„parallel=tests“* navoda u dokument testng.xml. Paralelno testiranje omogućuje izvođenje različitih testova na različitim uređajima. Ukoliko se emulatoru Pixel preda oznaka *regression*, na tom emulatoru izvoditi će se samo scenariji koji imaju oznaku *regression*. Paralelno se na drugom emulatoru Nexus mogu izvoditi testovi za scenarije koji imaju oznaku smokeTesting. Time se skraćuje proces testiranja jer se ne mora čekati da se prvo izvrte testovi na jednom emulatoru kako bi započeli testovi na drugom emulatoru. Na slici 4.11 prikazano je pokretanje testova na emulatorima.



Slika 4.11. Pokretanje testova na emulatorima

5. ANALIZA REZULTATA TESTIRANJA

5.1.Prikaz rezultata testiranja

Za potrebe ovog rada izrađeno je desetak testova, no kako bi se pokazali različiti rezultati odabrano je osam testova za prikaz rezultata testiranja. Detaljno izvješće svih rezultata može se pronaći u prilogu. Pokrenuto je pet testova s oznakom *regression* na emulatoru Nexus, i tri testa s oznakom *smokeTesting* na emulatoru Pixel.

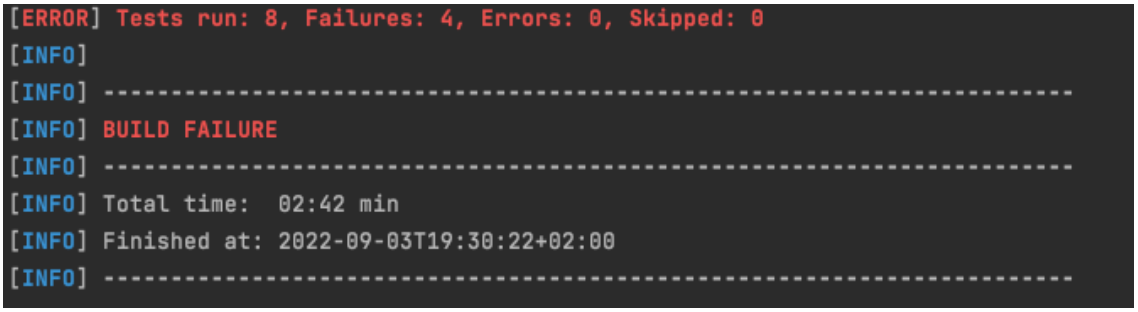
Testovi koji su pokrenuti na emulatoru Nexus su:

- otvaranje zaslona Njuške
- abecedno sortiranje pasa
- otvaranje Google karte
- prijava korisnika
- ulazak u aplikaciju bez prijave

Testovi koji su pokrenuti na emulatoru Pixel su:

- prolazak kroz onboarding
- preskok onboarding-a
- registracija korisnika

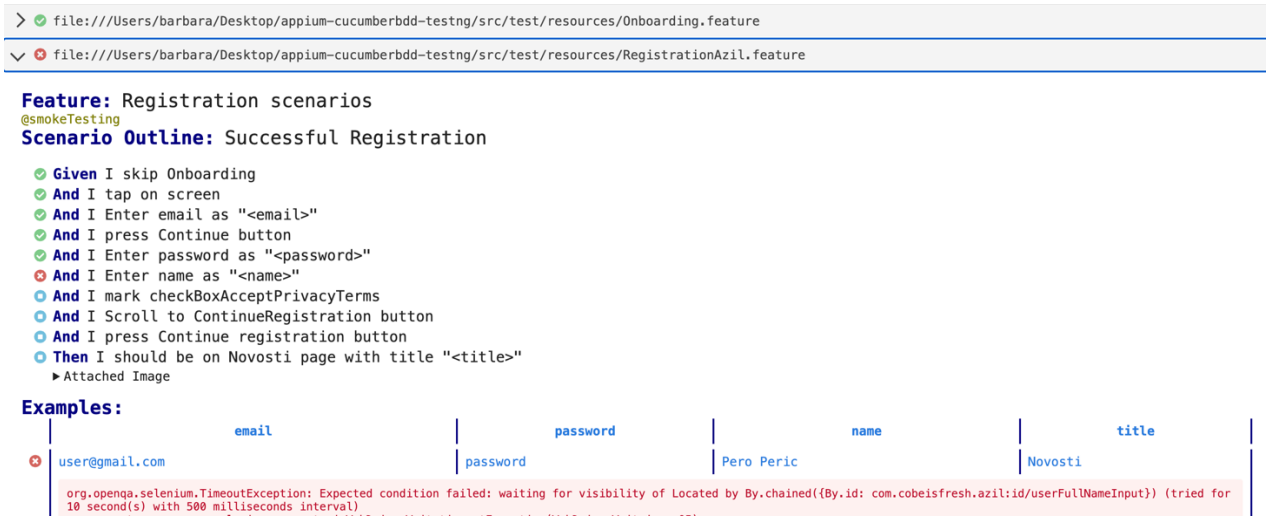
Nakon što su se testovi izvršili, Appium kreira dokument cucumber.html koji prikazuje rezultate testiranja. Rezultati se mogu vidjeti i u terminalu u kojem se pokrenuo Appium. Na slici 5.1 može se vidjeti da je pokrenuto osam testova, od kojih su četiri testa pala. Nije se dogodio nijedan error i nijedan test nije preskočen.



Slika 5.1. Rezultati testiranja prikazani u terminalu

Vrijeme izvršavanja je dvije minute i 42 sekunde. Prikazan je i datum i točno vrijeme kada su testovi izvršeni. U terminalu je moguće vidjeti korake izvršavanja testova no za pregledniji

prikaz potrebno je otvoriti generirani dokument cucumber.html. U dokumentu se može vidjeti koji testovi su pali i zbog čega. Dokument je kreiran za svaki emulator posebno. Na slici 5.2 prikazano je izvješće testa 'registracija novog korisnika' pokrenutog na emulatoru Pixel.



Slika 5.2. Cucumber izvješće emulatora Pixel – registracija

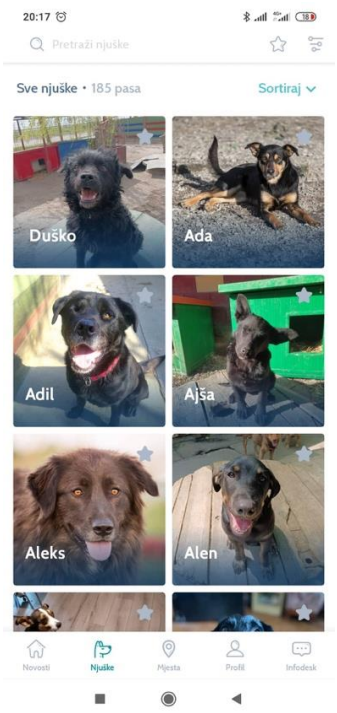
Vidljivo je da su svi koraci uspješno izvršeni sve do koraka 'Unos imena', nakon tog koraka nijedan korak nije izvršen te je test pao. Razlog zbog kojeg je test pao je što nije pronađen element 'nameInput'. To se dogodilo jer je korisnik s korisničkim imenom 'user@gmail.com' već registriran i nije otvoren zaslon za registraciju korisnika nego za nastavak prijave. Kao što je objašnjeno u prethodnom poglavlju registracija i prijava korisnika dijele isti zaslon, ukoliko se korisnik želi registrirati treba unijeti nepostojeću e-mail adresu, a ukoliko se želi prijaviti u aplikaciju treba unijeti postojeću e-mail adresu. Moguće je da testovi padnu zbog nekih nepredviđenih scenarija kao što je ovaj te je zbog toga uvijek potrebno pogledati izvješće kako bi se utvrdilo što se točno dogodilo. Na slici 5.3 vidljivo je da je test registracija uspješno izvršen nakon što je promijenjeno korisničko ime.

Slika 5.3. Cucumber izvješće emulatora Pixel – registracija

Na emulatoru Nexus pala su tri testa: sortiranje pasa abecednim redom, prijava u aplikaciju i otvaranje aplikacije bez prijave. Razlog zbog kojeg je pao test sortiranje pasa je taj što ime psa nije počelo na slovo 'a'. Kako bi se provjerilo je li to zaista tako može se testirati aplikacija ručno i nakon što se usporedi vidljivo je da funkcionalnost sortiranje abecednim redom ne radi ispravno. Na slici 5.4 prikazano je izvješće testa 'sortiranje pasa abecednim redom' na emulatoru Nexus.

Slika 5.4. Cucmber izvješće emulatora Nexus – sotiranje pasa abecednim redom

Na slici 5.5 vidljivo je da sortiranje pasa ne radi ispravno, jer prvi pas ne počinje slovom 'a', ova greška treba se prijaviti programeru kako bi ju ispravio.



Slika 5.5. Sortiranje pasa

Druga dva testa su pala jer je isteklo vrijeme potrebno za pronalazak određenog elementa. Takvi scenariji su vrlo mogući jer može doći do gubitka internetske veze ili sporosti u izvođenju emulatora. Zbog toga je uvijek potrebno pogledati zbog čega je test pao. Nakon što su testovi pokrenuti još jednom, uspješno su prošli kao što je prikazano na slici 5.6.

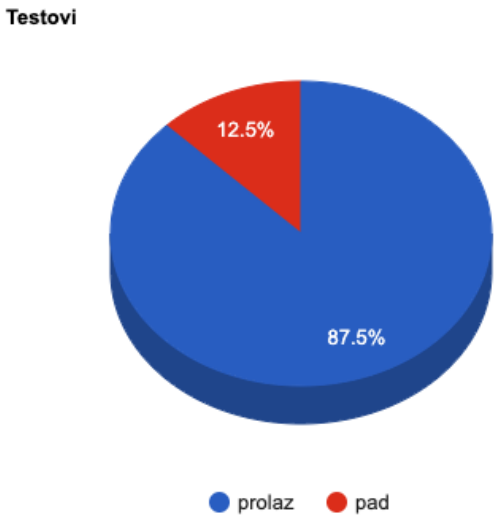
```
Feature: Login scenarios
@regression
Scenario Outline: Login with email

  Given I skip Onboarding
  And I tap on screen
  And I Enter email as "<email>"
  And I press Continue button
  And I Enter password as "<password>"
  And I press Continue button
  Then I should be on Novosti page with title "<title>"

Examples:
    |          email          | password |  title  |
  | 1999barbaraa@gmail.com | password | Novosti |
@regression
Scenario Outline: Enter without login

  Given I skip Onboarding
  And I press withoutLogin button
  Then I should be on Novosti page with title "<title>"

Examples:
    |  title  |
  | Novosti |
```

Slika 5.6. Prijava korisnika, ulazak u aplikaciju bez prijave

Moguće je pokrenuti sve testove na oba emulatora. Ispred svakog scenarija nalazi se oznaka *@test* i u klasama *Runner* potrebno je promijeniti oznaku u *@tag*. Izvješće svih testova vidljivo je na slici 5.7.

Slika 5.7. Izvršavanje svih testova

Samo je jedan test pao, i to test u datoteci značajki 'Njuske', test za sortiranje pasa po abecednom redu, kao što je i očekivano. Na slici 5.8 prikazana je prolaznost testova.



Slika 5.8. Grafički prikaz prolaznosti testova

Za izvođenje svih šesnaest testova, emulatorima su bile potrebne tri minute i 42 sekunde. Ručnim testiranjem to bi trajalo znatno duže. Ovo testiranje izvelo se na različitim uređajima te je smanjilo trošak upotrebe fizičkih uređaja. Ručno testiranje dugotrajan je proces i zamoran je za ponavljanje, dok je automatizirano testiranje vrlo brzo i nije problem izvršavati iste testova više puta. Proces ručnog testiranja može biti pogrešan zbog ljudskog faktora, dok je proces automatizacije pouzdan jer se temelji na kodu i skriptama. Međutim može doći do pogreške kao što je pokazano u prethodnom poglavlju.

Ukoliko testovima treba dugo vremena za izvođenje ili se unesu pogrešni podaci za testiranje može doći do netočnih rezultata. Zbog toga je potrebno nadzirati izvođenje testova i provjeriti generirano izvješće testiranja. Premda je za izvođenje automatiziranih testova potrebno

značajno manje vremena, u obzir se treba uzeti vrijeme i napor za stvaranje početnih skripti za testiranje. Automatizirani testovi djeluju i kao dokumentacija koji pruža povijest izvršavanja testova, novi programer ili tester na projektu može pogledati generirana izvješća testiranja i brzo razumjeti bazu koda. Ručno testiranje prikladnije je za testiranje upotrebljivosti i pronalazak UI/UX grešaka.

5.2. Analiza koraka

Cijelo izvješće analize koraka može se pronaći u prilogu diplomskog rada. Tablica analize koraka sadrži:

- naziv koraka
- učestalost pojavljivanja koraka
- prosječno trajanja koraka
- maksimalno trajanja koraka,
- ukupno trajanje koraka
- prolaznost koraka.

Na slici 5.8 vidljivo je da je najčešće korišten korak 'preskoči onboarding' koji se pojavio u devet testova, zatim korak 'klikni na withoutLogin gumb' koji se pojavio u sedam testova, te nakon toga korak 'klikni na skip gumb' koji se pojavio u četiri testa. Koraci inicijalizacije pogonskog programa i gašenja pogonskog programa pojavljuju se u svakom testu.

Implementation	Occurrences	Average duration	Max duration	Total durations	Ratio
com.qa.stepdef.Hooks.initialize()	11	7.630	9.512	1:23.938	100.00%
com.qa.stepdef.Hooks.quit(io.cucumber.java.Scenario)	11	5.782	7.024	1:3.604	100.00%
com.qa.stepdef.LoginAzilStepDef.iPressSkipButton()	9	2.680	3.388	24.126	100.00%
com.qa.stepdef.LoginAzilStepDef.iPressWithoutLoginButton()	7	1.006	1.594	7.042	100.00%
com.qa.stepdef.OnboardingStepDef.iPressSkipButton()	4	3.475	5.400	13.902	100.00%

Slika 5.9. Analiza koraka po učestalosti pojavljivanja

Prolaznost svih koraka je stopostotna. Prosječno trajanje funkcije *initalize()* je devet sekundi što je zadovoljavajuće s obzirom da se radi o pokretanju emulatora i aplikacije. Prosječno trajanje funkcije *iPressWithoutLoginButton()* je nešto više od jedne sekunde, s obzirom da se ovaj test pojavljuje sedam puta rezultat je zadovoljavajući. Moguće je sortirati korake po prosječnom vremenu trajanja koraka. Najduže izvođenje ima korak 'klikni na abecedno sortiranje' na Njuska zaslonu čije je trajanje skoro 6 sekundi. Najkraće izvođenje ima korak

'potvrdi prihvaćanje uvjeta privatnosti' na zaslonu za registraciju koje traje manje od jedne sekunde. Na slici 5.10 prikazana je tablica analize koraka po prosječnom vremenu trajanja. Kao i na prethodnoj slici prolaznost svih koraka je stopostotna.

com.qa.stepdef.NjuskeStepDef.IClickOnSortAlphabeticallyButton()	1	5.916	5.916	5.916	100.00%
com.qa.stepdef.Hooks.quit(io.cucumber.java.Scenario)	11	5.782	7.024	1:3.604	100.00%
com.qa.stepdef.ProfilStepDef.iShouldBeOnProfilPageWithTitle(java.lang.String)	1	5.721	5.721	5.721	100.00%
com.qa.stepdef.LoginAzilStepDef.iEnterPassword(java.lang.String)	2	5.599	5.602	11.198	100.00%
com.qa.stepdef.MjestaStepDef.iShouldBeOnMjestsPageWithTitle(java.lang.String)	1	5.141	5.141	5.141	100.00%

Slika 5.10. Analiza koraka po prosječnom vremenu trajanja

Vidljivo je da prosječno trajanje ne odstupa puno od maksimalnog trajanja koraka. Prosječno trajanje koraka *quit()* je više od pet sekundi dok je maksimalno trajanje sedam sekundi.

5.3.Utjecaj testiranja na razvoj programskog rješenja

Razvoj kvalitetnog proizvoda programske podrške ključna je potreba za industriju programske podrške. Kvalitetan proizvod ne može se razviti bez uključivanja osiguravatelja kvalitete u razvojni ciklus programske podrške [25]. Greška koja je pronađena može utjecati na korisničko zadovoljstvo te ju je potrebno ispraviti u što kraćem vremenu. Automatizirano testiranje pomaže prilikom izmjene koda ili nadogradnje programske podrške novim funkcionalnostima. Regresijsko testiranje služi za provjeru, da nije narušeno ponašanje aplikacije.
Kreiranjem automatiziranih testova primijećeno je da na nekim mobilnim elementima nedostaje atribut *id* te se za dohvaćanje elementa morao koristit *xpath.* Preporučljivo je koristiti identifikator za dohvaćanje elemenata jer je to podatak jedinstven za pojedini element. Ukoliko programer doda novi element, moguće je da će doći do izmjene varijable *xpath* te će tako nesvjesno utjecati na izvršavanje testova.

5.3.1. Izvješće o greški

Kako bi se obavijestilo programera o greški potrebno je napraviti izvješće o greški. Treba sadržavati naslov, opis greške, korake kako reproducirati grešku, očekivano ponašanje i stvarno ponašanje te prioritizaciju. Potrebno je navesti inačicu sustava i uređaj na kojem je greška pronađena. Preporučljivo je dodati i neki vizualni dokaz o greški, može biti slika ili videozapis. U tablici 5.1. prikazano je izvješće o greški.

Tab 5.1. Izvješće o greški

Naslov:	Sortiranje pasa abecednim redom ne radi ispravno
Vrsta uređaja:	Emulatori Nexus i Pixel
Koraci:	1) Otvori aplikaciju 2) Preskoči onboarding 3) Klikni na Njuške na navigaciji 4) Klikni na gumb 'Sortiraj' 5) Klikni na gumb 'abecedno' 6) Klikni na prvog psa
Očekivani rezultat:	Ime psa počinje na slovo 'a'
Stvarni rezultat:	Ime psa počinjen na slovo 'd', nakon njega slijede abecednim redom poslagani psi
Prioritizacija:	*Medium*

6. ZAKLJUČAK

Cilj ovog diplomskog rada bila je izrada automatiziranih testova za Android mobilnu aplikaciju Azil Osijek. Koristio se programski jezik Java u programskom okruženju IntelliJ IDE. Za provedbu testiranja korišteni su alati Appium i Cucumber. Napravljeni su testni planovi i testni slučajevi, te su implementirani u provedbi automatiziranih testova. Objašnjena je potreba za testiranjem te također i potreba za automatiziranim testiranjem zbog agilnog načina rada koje je usko povezano s regresijskim testovima. Provedeni su automatizirani testovi na emulatorima Nexus i Pixel, te su generirana izvješća koja prikazuju rezultate automatiziranog testiranja. Izvješća su analizirana te su testovi po potrebni nadograđeni. Izrađeno je izvješće o greški koje se daje programeru na pregled. Uspoređeno je automatizirano i ručno testiranje te je dan zaključak kako je automatizirano testiranje isplativo ukoliko se pojavi česta potreba za regresijskim testiranjem aplikacije.

Najvažnija prednost automatiziranog testiranja je isključivanje ljudskog faktora i vrijeme izvođenja testova, mogućnost izvođenja na različitim operativnim sustavima te paralelno izvođenje testova na više uređaja. Automatiziranim testovima rješava se problem čestih ponavljanja testova jer računalo svaki test izvodi brzo i točno. Nedostatak automatiziranog testiranja je nemogućnost testiranja upotrebljivosti. Moguće je analizirati prosječno trajanje testova, maksimalno trajanje testova, učestalost pojavljivanja koraka itd. Analizom navedenog moguće je refaktorirati kod i ubrzati provedbu automatiziranih testova.

LITERATURA

[1] S. Braun, F. Elzberzhager, K. Holl, Automation Support for Mobile App Quality Asscurance - A Tool Landscape, Procedia Computer Science, Volume 110, pages 117-124, 2017.

[2] What is Software Testing? https://www.guru99.com/software-testing-introduction-importance.html [pristup: 25.6.2022.]

[3] D. Graham, E. van Veenendaal, I. Evans, R. Black, Foundations of Software Testing, Thomson, 2014.

[4] R. Budiu, Mobile: Native Apps, Web Apps, and Hybrid Apps, Nielsen Norman Group, 2013. https://www.nngroup.com/articles/mobile-native-apps/

[5] What is Non Functional Testing? https://www.perforce.com/blog/alm/what-non-functional-testing [pristup: 26.6.2022.]

[6] I. Hmelik, Getting Started With Automated Testing, https://www.cobeisfresh.com/blog/getting-started-with-automated-testing [pristup: 15.7.2022.]

[7] D. Graham, M. Fewster, Experiences of test automation, 2011.,

[8] Perfect Planning for Your Test Automation https://www.innominds.com/blog/perfect-planning-for-your-test-automation [pristup: 4.8.2022.]

[9] D. Kumar, The Impact of Test Automation on Software's Cost, Quality and Time to Market, Procedia Computer Science, Volume 79, Pages 8-15, 2016.

[10] I. Salman, P. Rodriguez, B. Turhan What Leads to a Confirmatory or Disconfirmatory Behavior of Software Testers, IEEE Transactions on Software Engineering, pp. 1351-1368, vol. 48, 2022.

[11] P. Jurvanen, M. Mantyla, V. Garousi, Choosing The Right Test Automation Tool: A Grey Literature Review Of Practitioner Sources, Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, Pages 21-30, Karlskrona, Sweden 2017.

[12] 5 Test Automation Problems Even Modernqa Teams Face, 2019 https://smartbear.com/blog/5-test-automation-problems-qa-teams-face/ [pristup: 2.8.2022.]

[13] J. Wang, J. Wu, Research on Mobile Application Automation Testing Technology Based on Appium, 2019 International Conference on Virtual Reality and Intelligent Systems, Jishou, China, 2019.

[14] B. Kirubakaran, V. Karthikeyani, Mobile Application Testing – Challenges and Solution Approach Trough Automation, 2013 International Conference on Pattern Recogition, Informatics and Mobile Engineering, Salem, India 2013.

[15] A. Sehgal, Clean Architecture, 2019. https://anmolsehgal.medium.com/clean-architecture-fef10b093ad0 [pristup: 5.9.2022.]

[16] MVVM (Model View ViewModel) Architecture Pattern in Android, 2022. https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/ [pristup: 5.9.2022.]

[17] Cucumber, https://cucumber.io/docs/guides/overview/ [pristup: 1.8.2022.]

[18] How Cucumber Works?, http://www.seleniumframework.com/cucumber-2/make-a-case/how-cucumber-works-2/ [pristup: 3.8.2022.]

[19] Vodič za IntelliJ IDEA – Razvoj Jave s Intellij IDE, https://hr.myservername.com/intellij-idea-tutorial-java-development-with-intellij-ide [pristup: 24.7.2022.]

[20] Maven, https://maven.apache.org/ [pristup:25.7.2022.]

[21] A. Angappa, What are Virtual Devices and How to Use Them for Testing, 2022., https://www.lambdatest.com/blog/virtual-device/ [pristup: 10.8.2022.]

[22] K. Juhnke, M. Tichy, Challenges With Automotive Test Case Specifications, Proceedings of the 40th International Conference on Software Engineering: Companion Processdings, Pages 131-132, 2018.

[23] J. Kaur, Bridging The Gap Between Testers and Developers by Collobration, 2019., https://dzone.com/articles/bridging-the-gap-between-testers-and-developers-by [pristup: 2.8.2022.]

[24] A. Paiva, End-To-Eng Automatic Business Process Validation, Procedia Computer Science, Volume 130, Pages 999-1004, 2018.

[25] M.D. Haiderzai, M.I. Khattab, How Software Testing Impact the Quality of Software Systems, International Journal of Engineering in Computer Science, Pages 5-9, 2019.

SAŽETAK

Cilj ovog diplomskog rada je provedba automatiziranih testova za aplikaciju Azil Osijek. Za provedbu automatiziranih testova izrađen je testni plan i testni slučajevi. Detaljno su opisane vrste i razine testiranja mobilnih aplikacija, izazovi koji se pojavljuju prilikom testiranja. Posebno je istaknuta uloga regresijskog testiranja i njena povezanost s automatiziranim testovima. Za izradu testova korišteno je programsko okruženje IntelliJ IDE te alat Appium s Cucumber dodatkom te su svi testovi pisani jezikom Java. Nakon što je aplikacija testirana automatiziranim testovima, rezultati su analizirani i prikazani grafički. Analizirano je prosječno trajanje testova, maksimalno trajanje testova, učestalost pojavljivanja koraka u testovima. Kreirano je izvješće o grešci koji se daje programeru na pregled kako bi ispravio grešku. Predloženo je poboljšanje za testiranu mobilnu aplikaciju.

Ključne riječi: Android, Appium, automatizirano testiranje, Cucumber, mobilna aplikacija.

ABSTRACT

The goal of this thesis is the implementation of automatic tests for the application Azil Osijek. A test plan and test cases were created for the implementation of automatic tests. The types and levels of mobile application testing, the challenges that arise during testing are described in detail. It is important to emphasize the role of regression testing and it's connection with automatic tests. The IntelliJ IDE programming environment and the Appium tool with the Cucumber plugin were used to create the tests, and all tests were written in the Java language. After the application was tested with automatic tests, the results were analyzed and presented graphically. Average duration of tests, maximum duration of tests, frequency of occurrence of steps in tests were analyzed. A bug report is created and given to the developer to review in order to fix the bug. An improvement was proposed for testing the mobile application.

Keywords: Android, Appium, automation testing, Cucumber, mobile application.

ŽIVOTOPIS

Barbara Aščić rođena je 19.02.1999. godine u Đakovu. Pohađala je Osnovnu školu Tenja u Tenji. Nakon toga upisuje I. Gimnaziju Osijek u Osijeku. Nakon završetka srednjoškolskog obrazovanja s odličnim uspjehom, upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku na sveučilištu Josipa Jurja Strossmayera, smjer preddiplomski studij računarstva te ga završava 2020.godine. Iste godine upisuje diplomski studij računarstva, smjer programsko inženjerstvo na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Na drugoj godini preddiplomskog studija dobiva STEM stipendiju. Na prvoj godini diplomskog studija dobiva sveučilišnu stipendiju Josipa Jurja Strossmayera. Od programskih jezika usavršila se u korištenju Jave i JavaScript-a. Od 2021. godine radi u osječkoj tvrtki Cobe kao inženjer osiguranja kvalitete, gdje usavršava svoje znanje u pisanju i provođenju ručnih i automatiziranih testova.

PRILOZI

Prilog 1. Diplomski rad u datoteci docx

Prilog 2. Diplomski rad u datoteci pdf

Prilog 3. Izvješće testiranja u datoteci docx