Testiranje, integracija i isporuka klijentskih komponenti web aplikacija

Šifner, Ivan

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:621103

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-04-01**

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

Sveučilište Jurja Dobrile u Puli

Fakultet informatika

IVAN ŠIFNER

Testiranje, integracija i isporuka klijentskih komponenti web aplikacija

Diplomski rad

Pula, travanj, 2022.

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

IVAN ŠIFNER

Testiranje, integracija i isporuka klijentskih komponenti web aplikacija

Diplomski rad

JMBAG: 0269102763**, redoviti student**

Studijski smjer: Informatika

Predmet: Izrada informatičkih projekata

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Nikola Tanković

Pula, travanj, 2022..

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Ivan Šifner, kandidat za magistra informatike ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, _____ godine

IZJAVA

o korištenju autorskog djela

Ja, Ivan Šifner dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom Testiranje, integracija i isporuka klijentskih komponenti web aplikacija koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____ (datum)

Potpis

_____

Sadržaj:

1. Uvod

Opće je poznato kako je testiranje bitna stavka kako bi se osigurala kvaliteta softvera i spriječilo stvaranje grešaka koji su poslije u proizvodnom okruženju teški i skupi za ispraviti. Kako bi se osigurala odgovarajuća razina zadovoljstva od strane korisnika provode se testovi s kraja na kraj da bi se testirale funkcijske značajke aplikacije unutar stvarnih okolnosti i podataka koji repliciraju stvarno stanje primjenjujući alate za automatizaciju testnih scenarija. Sa stajališta korisnika ona je bitna kako za sve strane koje od određenog programskog proizvoda imaju koristi, te ona ne može biti sama automatizacija testa nego mora biti i automatizacija određenog djela u procesu softverskog razvoja. Tijek rada kontinuirane integracije i isporuke zahtjeva od developera da se kod integrira rano i često kako bi se omogućilo timovima rano otkrivanje problema. Integracija i isporuka forsira developere i testere da ispoštuju određena pravila kako bi se proces razvoja softvera olakšao i automatizirao. Kada je kod prenošen npr. na Azure DevOps on se automatski gradi, testira na razini komponente ili se provode integracijski testovi, a u zadnje vrijeme prednost pronalaze i automatski testovi s kraja na kraj. Rezultati se tada vraćaju programeru, a pucanje u bilo kojoj točki tog „cjevovoda“ nam govori o korisnim informacijama na koje moramo obratiti pozornost da osiguramo konzistentnost u razvoju programskog proizvoda. U ovome radu bit će naglasak na testiranju klijentskih komponenti objašnjavajući važnost samog testiranja s kraja na kraj (eng. End to end testing, ili skraćeno E2E testing), prikaz nekih od automatskih testova i njihovih važnosti, te alata, testiranje u samom procesu razvoja softvera te cjevovodi kontinuirane integracije i isporuke klijentskih komponenti.

2. Testiranje i kontinuirana integracija

2.1 Testiranje klijentskih komponenti kao važan proces u razvoju programskog proizvoda

Testiranje klijentskih komponenti predstavlja prednji dio softverske aplikacije s kojom korisnik „komunicira“. U suštini, klijentima su vidljivi svi aspekti softvera. To obično uključuje elemente grafičkog sučelja kao što su izbornici, gumbovi, obrasci – sve što krajnji korisnici mogu vidjeti prilikom navigacije aplikacijom. Također može uključivati aspekte poput brzine učitavanja stranice, a to su čimbenici koji pridonose cjelokupnom korisničkom iskustvu. Testeri klijentskih komponenti trebali bi imati znanje o zahtjevima klijenta, ali i iskustvo u razvojnim okvirima kao što su Cypress i Selenium te bibliotekama kao što su Testcafe pomoći će testerima u organizaciji i izvršenju njihovih zadataka.Nekada se frontend testiranje i backend testiranje poistovjećuju pa je prema tome potrebno definirati i neke razlike između njih koje će biti opisane sada.

2.1.1 Razlike između backend testiranja i testiranja klijentskih komponenti.

Klijentske komponente su dio programa, odnosno aplikacije na strani klijenta. Ono uključuje sve vidljivo tijekom korištenja aplikacije. Svaka web aplikacija ima troslojnu arhitekturu koja uključuje: klijente, poslužitelje i informacijske sustave ili resurse. Prezentacijski sloj se sastoji od klijenta, a to je dio kojim testeri testiraju klijentske komponente. Oni provode testiranje sučelja i testiraju upotrebljivost te provjeravaju način kako funkcionira web stranica ili aplikacija. U ovoj fazi postavljaju se pitanja tipa u domeni onih koja su najvažnija sa stajališta klijenta. Testeri klijentskih komponenti provjeravaju jesu li izgled i dojam web stranice usklađeni sa zahtjevima klijenta. Za razliku od toga backend testiranje provjerava učinkovitost funkcionalnosti na strani poslužitelja i baze podataka softvera. To je neophodno kako bi se osigurala točna interakcija međuovisnih komponenti kako bi se podaci proizvodili, obrađivali i pohranjivali na najučinkovitiji način. Ono zahtijeva poznavanje baza podataka kao što su SQL Server, MySQL, Oracle, DB2 itd. Backend testiranje ne uključuje korisničko sučelje (eng. User Interface - UI) aplikacije. Uglavnom ono provjerava da li baza podataka pohranjuje točne podatke unesene pomoću korisničkog sučelja. Također, ako su negdje neki podaci vidljivi na korisničkom

sučelju, backend testeri provjeravaju da li upiti nad bazom podataka vraćaju točne podatke.

2.1.2 Potreba za testiranjem klijentskih komponenti

Testiranje klijentskih komponenti donosi nam mnoge benefite koje se naziru u pogledu:

- Identificiranja problema performansi (eng. performance testing); Web aplikacije posjeduju tri funkcionalna sloja – klijente, poslužitelje, resurse (ili informacijske sustave). Takvo testiranje obrađuje sloj klijenta ili dio softvera koji se prezentira klijentu
- Provjera funkcionalnosti za više uređaja tzv unakrsno testiranje uređaja (eng. Cross-device testing); Značajan aspekt testiranja klijentskih komponenti je provjera ponašanja web stranice i aplikacije na više preglednika, verzija preglednika i uređaja (mobilnog i stolnog). To ne uključuje samo značajke već i odziv na zaslonima različitih veličina i razlučivosti.
- Provjera da li su svi zahtjevi implementirani na korisnicima bitan način; To uključuje testiranje poveznica na svakoj stranici unutar specifične domene koja se testira. Ono uključuje i elemente dizajna kao što su provjere da li se događa preljev stranica (eng. Page overflow) kada većina informacija ispuni određenu stranicu do kraja, te se ovdje također provjerava da li su podnaslovi konzistentni i da li prelaze jedan preko drugoga.
- Testiranje klijentskih komponenti daje nam odgovor na pitanja da li je pojedini sustav lagan za korištenje, kakva je navigacija unutar aplikacije i generalni izgled aplikacije. On provjera i sadržaj same aplikacije. Sadržaj bi trebao biti logičan i lako razumljiv. Takvo testiranje nam u nekim slučajevima govori da li se dešavaju pojedine pravopisne pogreške unutar aplikacije, da li korištenje tamnih boja nervira korisnike i smije li se isti koristiti unutar same aplikacije. Taj se problem nastoji riješiti praćenjem raznih standarda koji potpomažu testerima u donošenju odluka. To su općeprihvaćeni standardi poput onih koji govore o neprikladnim bojama, fontovima, okvirima itd.
-

2.1.3 Tipovi testiranja klijentskih komponenti

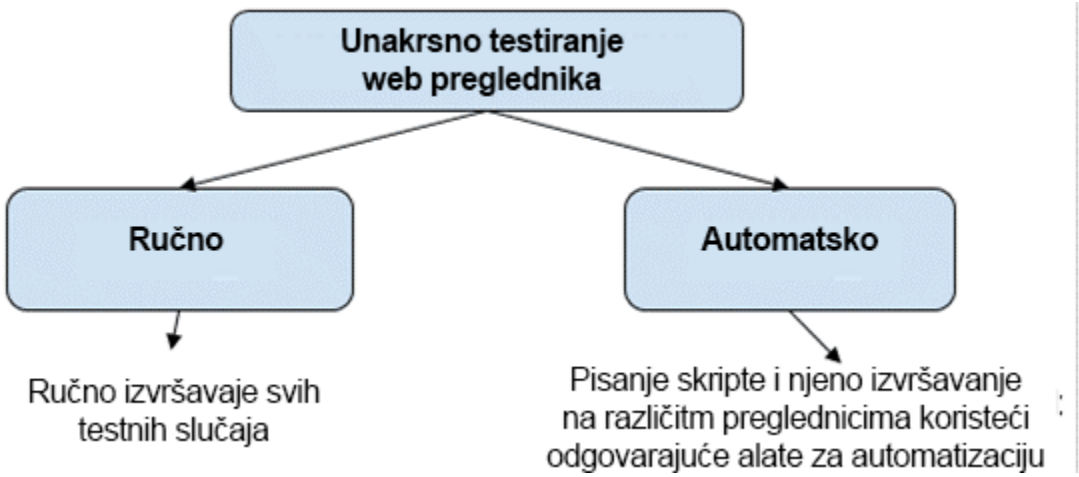
Testiranje klijentskih komponenti uključuje različite tipove testiranja od kojih su najznačajniji : testiranje komponenti, testovi vizualne regresije, unakrsno testiranje web preglednika, testiranje integracije, test pristupačnosti, te e2e testiranje ili testiranje *s kraja na kraj*.

1.) Testiranje komponenti - također poznato kao testiranje jedinica ili modula na izoliran način ima ciljeve kao što su:
   a. Smanjenje rizika
   b. Provjera jesu li funkcionalno i nefunkcionalno ponašanje komponente predstavljene na način kako je dizajnirano i specificirano u korisničkim zahtjevima
   c. Pronalazi greške u komponentama
   d. Sprečava „bježanje“ grešaka u više razine testiranja (testiranje integracije, sustava, test prihvaćenosti)

2.) Testiranje vizualne regresije - regresijsko testiranje se koristi za provjeru da bilo koje promjene sustava ne ometaju postojeće značajke i/ili strukturu koda. Uobičajeno je da programeri mijenjaju ili dodaju dio koda i da on nenamjerno poremeti nešto što je prije dobro funkcioniralo. Testiranje vizualne regresije primjenjuje istu logiku, ali ograničava testiranje na vizualne aspekte softvera. Drugim riječima, provjerava da promjene koda ne razbiju nijedan aspekt vizualnog sučelja softvera. Test vizualne regresije provjerava što će korisnik vidjeti nakon izvršenja bilo kakvih promjena koda uspoređujući snimke zaslona snimljene prije i nakon promjena koda. Implementacija iste bit će prikazana poslije u ovome radu.

3.) Unakrsno testiranje web preglednika (eng. Cross-browser testing) – predstavlja testiranje u kojem se nastoji ispitati da li različiti web preglednici i njihove pripadajuće verzije imaju isto ponašanje na našoj aplikaciji. Znači u ovome slučaju aplikacije se isto testiraju i na više uređaja pa se i ovo testiranje nekad popularno naziva „unakrsno testiranje web preglednika i uređaja“. Takva vrsta testiranja je uglavnom automatizirana korištenjem alata kao što su LambdaTest, CrossBrowserTesting, Selenium, BrowserStack, Browserling, te

je vezana za razne web preglednike kao što su : Google Chrome, Firefox, Microsoft Edge, Safari[1]. Primjer cross browser testiranja bit će prikazan pomoću LambdaTest platforme unutar testcafe implementacije kasnije u radu. Na slici br. 1 prikazano je kako se generalno izvršavaju prethodno opisani testovi. Ona nam govori kako se u 1. slučaju vrši ručno („manualno“) testiranje u kojemu se trebaju najprije odabrati preglednici na kojima će se izvoditi testiranje te test slučajevi koji se nastoje pokriti ovom vrstom testiranja. Ova vrsta „ponavljajućih“ testova najprikladnija je za automatizaciju. Stoga je isplativije provesti ovo testiranje korištenjem alata za automatizaciju.



Slika br.1: Cross Browser testiranje

4.) Integracijsko testiranje - usredotočuje se na interakcije između komponenti ili sustava. Postoje dvije vrste testiranja integracije, a to su testiranje integracije komponente, te testiranje integracije sustava. A razlike među njima su sljedeće:
   a. Testiranje integracije komponenti usredotočuje se na interakcije i sučelja između integriranih komponente.
   b. Testiranje integracije sustava fokusira se na interakcije i sučelja između sustava, paketa i mikroservisa. Testiranje integracije sustava također može pokriti interakcije sa sučeljima koje pružaju vanjske organizacije (npr. web usluge).

[1] https://www.softwaretestinghelp.com/how-is-cross-browser-testing-performed/

5.) Testiranje prihvaćenosti - Ono ocjenjuje usklađenost sustava s poslovnim zahtjevima. On nam odgovara na pitanje da li je proizvod spreman na „isporuku“.

6.) E2E testiranje (hrv. testiranje *s kraja na kraj* – govori o tome da protok informacija i korištenje aplikacije bi trebalo biti jednostavno od početka do kraja. To je jedna vrsta sveobuhvatnog testiranja koja osigurava da se aplikacija tijekom svog rada ponaša u skladu s očekivanjima. Također ona održava integritet podataka između sustava i između njegovih komponenti.

	Brzina	Opseg	Kompleksnost	Održavanje
Testovi komponente	Jako brzi	Mali	Izrazito mala	Malo/veoma lako
Testovi integracije	Brzi	Prosječan	Prosječna	Prosječno
E2E testovi	Spori	Izrazito velik	Izrazito velika	Veliko/izrazito kompleksno

Slika br.2: Razlike između tipova testiranja

Na slici br. 2 prikazane su neke razlike između tipova testiranja u pogledu brzine njihove implementacije, opsega, kompleksnosti i održavanja. Prema tome može se doći do zaključka da je e2e testiranje najkompleksnija vrsta testiranja koja je ujedno i najteža za održavanje. Opseg ovih testova kreće se od jednostavnog tijeka (prijava na web stranicu) do složenijih testnih scenarija kao što je stavljanje proizvoda u košaricu i odlazak na naplatu. Općenito je opseg vrlo velik u usporedbi s testovima komponente i integracijskim testovima[2].

[2] https://cdohotaru.medium.com/principles-of-end-to-end-web-testing-7fa7d27d7826

2.2. Testiranje i kontinuirana integracija

Fleksibilnost, brzina i kvaliteta temeljni su stupovi modernog razvoja softvera. Povećana potražnja kupaca i razvoj tehnološkog krajolika učinili su razvoj softvera složenijim nego ikad, čineći tradicionalne metode životnog ciklusa razvoja softvera (skraćeno od eng. SDLC) sporijima u odnosu na prirodu razvoja koja se brzo mijenja. Prakse poput agilnog razvoja i DevOps-a postale su popularne u olakšavanju ovih promjenjivih zahtjeva unoseći fleksibilnost i brzinu u proces razvoja bez žrtvovanja cjelokupne kvalitete krajnjeg proizvoda.[3] Brzina je ključ modernog razvoja softvera. Prijašnja paradigma tradicionalnog razvoja softvera koristeći modele kao što su vodopadni (eng. Waterfall) model uvelike je zamijenjena brzim iterativnim tehnikama koje podržavaju razvoj i isporuku klijentskih komponenti. Automatizacijom integracije i isporuke, kontinuirana integracija i isporuka omogućuje timovima za razvoj softvera da se usredotoče na ispunjavanje poslovnih zahtjeva, istovremeno osiguravajući kvalitetu koda i sigurnost softvera.[4]

2.2.1 Povijest integracije i isporuke klijentskih komponenti

Nekada bi programer proveo nekoliko mjeseci u izgradnji novih značajki aplikacije. Zatim bi prošli kroz ogroman napor integracije istih, to jest, spajanje njihovih promjena u zajedničko spremište koda, što se neizbježno promijenilo u trenutku kada su započeli svoj rad. U 80-im i 90-im godinama dominantne razvojne metodologije slijedile su proces vodopadnog modela razvoja softvera (eng. Waterfall model). To je značilo da je razvoj softvera slijedio strogo sekvencijalnu strukturu u kojoj su timovi krenuli u sljedeću fazu tek nakon što je prethodna završena. Loša strana ovog pristupa leži u načinu na koji su timovi radili. Testeri i developeri imali su različito poimanje po pitanju zašto se mnogo greški pronalazi prilikom testiranja, odnosno zašto je kod nedovoljno kvalitetno implementiran. Otklanjanje nedostataka, osobito ako su napravljeni u ranoj fazi razvoja bilo je dugotrajno i ponekad mučno. Štoviše, počelo je postajati očito da duljina cjelokupnog procesa znači da krajnji proizvod ne odgovara uvijek izvornoj poslovnoj potrebi.[5] Tada, vjerojatno 1990-ih, netko je imao dobru ideju da kontinuirano radi

[3] https://www.bmc.com/blogs/what-is-ci-cd/
[4] https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html
[5] https://www.infostretch.com/blog/the-road-to-cicd-a-short-history-of-agile-development/

integraciju. To je značajno smanjilo integracijski rizik minimiziranjem i često eliminiranjem pomaka koda koji se događa između grane glavne linije (eng. Main branch) i trenutne značajke koja se implementira za aplikaciju.[6] Godine 2001. 17 programera začetnika softvera susrelo se na poznatoj razvojnoj konferenciji u Snowbirdu, Utah (SAD). Zajedno su objavili tzv. „Agile Manifesto“, koji je imao za cilj pronaći načine za izgradnju softvera koji bi bio više kolaborativan, osjetljiviji i isporučivao češća ažuriranja. Kasnije, 2009., druga grupa je stvorila Scrum Alliance. HP-ovo istraživanje pokazuje da je otprilike u to vrijeme agilna metodologija počela stvarno uzimati maha u poduzećima. Agilno usvajanje raslo je polako, ali postojano i do 2008. dosegnulo je 13% prihvaćanja. Do iduće godine to je skočilo na 17%, a do 2010. bilo je 37%. Dok je agilni princip spojio razvoj i testiranje, poslije je značajna odgovornost pala na DevOps. DevOps je tvrdio da druga područja poslovanja moraju također biti dio procesa te je predvidio stvaranje timova koji uključuju netehničko osoblje, kao što su menadžeri proizvoda. Ovo je bio popriličan kulturni pomak. DevOps je imao i druge ciljeve. Jedan važan cilj bio je stvoriti okruženje u kojem bi se nova izdanja mogla dostavljati brže i češće. Godina 2013.-14. došlo je vrijeme DevOpsa. U Gartnerovoj studiji o analizi ankete o usvajanju DevOps-a u 2015. god pokazalo je da je 29% poduzeća do tada koristilo DevOps projekte[7].

2.2.2 Što je kontinuirana integracija, a što kontinuirana isporuka

Kontinuirana integracija (eng. Continuous integration – CI), te kontinuirana isporuka (eng. Continuous delivery – CD) uvode automatizaciju i nadzor u kompletan ciklus razvoja programskog proizvoda.

1. Kontinuirana integracija može se smatrati prvim dijelom cjevovoda (eng. Pipeline) isporuke softvera u kojem se integrira, gradi i testira aplikacijski kod.
2. Kontinuirana isporuka druga je faza cjevovoda isporuke gdje se aplikacija postavlja u svoje proizvodno okruženje kako bi je koristili krajnji korisnici.

[6] https://jhall.io/archive/2021/09/26/a-brief-history-of-ci/cd/
[7] https://www.infostretch.com/blog/the-road-to-cicd-a-short-history-of-agile-development/

Kontinuirana integracija uspostavlja automatizirani način za izgradnju, „pakiranje“ i testiranje aplikacija. Proces integracije potiče programere da češće unose promjene koda, što dovodi do bolje suradnje i kvalitete koda.

Kontinuirana isporuka počinje tamo gdje kontinuirana integracija završava i automatizira isporuku aplikacija u odabrana okruženja, uključujući okruženja proizvodnje, razvoja i testiranja. Kontinuirana isporuka je automatizirani način da se promjene koda potiskuju u ta okruženja.[8] Faza razvoja i testiranja razvoja softvera je fokus kontinuirane integracije. Kako programeri mijenjaju softverski kod, te se promjene odmah provjeravaju u središnjem sustavu kontrole izvornog koda (npr. git). Kada se kod provjeri, pokreću se automatizirani procesi izgradnje i testovi kako bi se osiguralo da promjene nisu oštetile veći softverski sustav na kojem se radi. Kada se koriste kraći i češći ciklusi razvoja-izgradnje-testiranja, pogreške kodiranja se brže pronalaze, a rizik povezan s velikim promjenama koda se smanjuje[9]. Kontinuirana isporuka znači da nove softverske značajke i popravci koda prolaze kroz ciklus razvoja, izgradnje i testiranja (tzv. Ciklus razvoja softverskog proizvoda), te postaju dostupni što je prije moguće. Kada se manje promjene češće isporučuju u proizvodnju, smanjuje se rizik od velikih promjena koje bi mogle oštetiti sustav, a kašnjenje u njihovoj isporuci kupcima je minimalizirano.

2.2.3 Način na koji se kontinuirana integracija i isporuka nadopunjuju

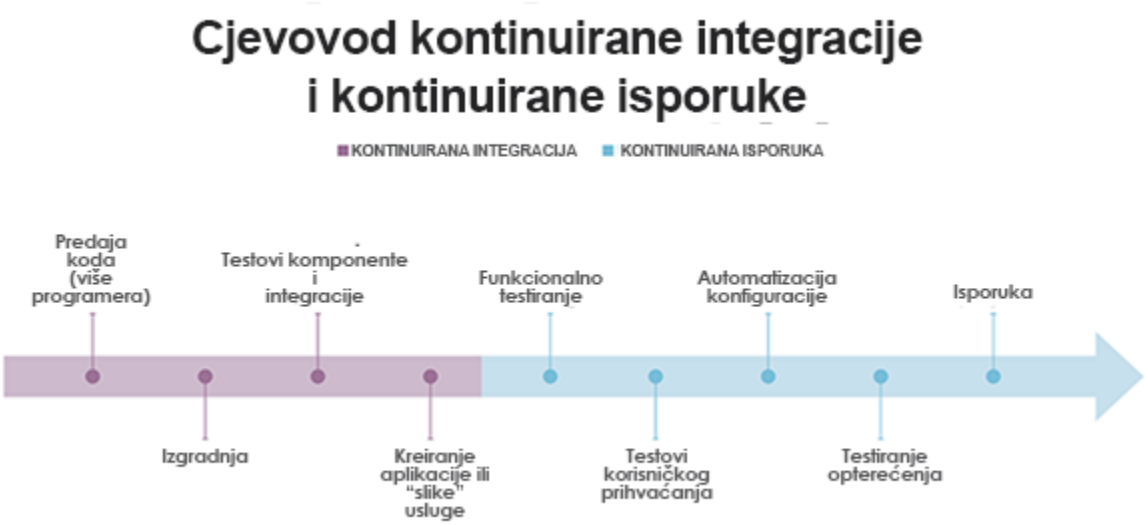
U praksi je normalno da više programera istovremeno radi na istoj bazi koda i često spremaju promjene u zajednički repozitorij koda. Učestalost izgradnje može biti svakodnevna ili čak nekoliko puta dnevno u nekim točkama životnog ciklusa projekta. Kontinuirana integracija koristi razne alate i tehnike automatizacije za stvaranje *build*-ova i njihovo praćenje kroz početno testiranje, kao što je provjera ili testiranje komponenti koda, zajedno s opsežnijim testiranjem integracije. Takva integracija je također poznata po brzim i detaljnim povratnim informacijama (eng. Feedback). Ograničena priroda svake iteracije u razvoju softverskog proizvoda znači da se greške identificiraju, lociraju, prijavljuju i ispravljaju s relativnom lakoćom[10]. Kada se kod „pakira“ (eng. Packaging), on

[8] https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html

[9] https://www.netapp.com/knowledge-center/what-is-continuous-integration-continuous-delivery-cicd/

[10] https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know

gradi neku sliku za isporuku (eng. Deployable image) koja se može implementirati, kao što je slika spremnika ili virtualnog stroja (eng. Virtual Machine - VM), prije nego što bude dostupna testerima. Nakon toga nastupa faza kontinuirane isporuke. Kontinuirana isporuka se također uvelike oslanja na alate i automatizaciju kako bi se kod izgradio putem naprednog testiranja, uključujući funkcionalno testiranje, korisničko prihvaćanje, konfiguraciju i testiranje opterećenja. Oni potvrđuju da izgrađena aplikacija (eng. *Build*) ispunjava zahtjeve i da je spremna za korištenje u proizvodnom okruženju (eng. Production Environment). Kako i kod integracije, tako i kod isporuke postoje inkrementalne iteracije koje osiguravaju da se svi problemi otkriveni tijekom testiranja identificiraju i otklone brže i jeftinije od tradicionalnih pristupa razvoju softvera. Sljedeća slika prikazuje prethodno opisano :



Slika br. 3: Prikaz nadopune integracije i isporuke koristeći pipeline

Na prethodnoj slici prikazan je uobičajen proces integracije i isporuke klijentskih komponenti. Na njoj je vidljivo kako razvojni programeri (developeri) vrše promjene koda i prenose ih do zajedničkog repozitorija koji je kontroliran od strane sustava za verzioniranje tih promjena. Nakon toga promjene se izgrađuju u cjevovodu kontinuirane integracije te se nad tim promjenama poslije provode testovi komponenti te integracijski testovi, a zatim se kreira „slika“ aplikacije koja predstavlja njeno trenutno stanje. Kao izlaz

kontinuirane integracije nastaje stanje aplikacije koje se nadalje provjerava u lancu isporuke. Nad tom aplikacijom vrše se razni funkcionalni testovi, testovi prihvaćanja, automatizacija konfiguracije softvera, testiranje opterećenja te je aplikacija konačno spremna na isporuku ovisno o uspješnosti prethodnih koraka. Uz postojeću integraciju i isporuku ovdje je važno spomenuti još jedan pojam, a to je pojam kontinuirane implementacije (eng. Continuous deployment) iako o njemu neće biti mnogo rečeno u ovome radu. Glavna razlika između kontinuirane isporuke i kontinuirane implementacije je u tome što kontinuirana implementacija automatski implementira svaku prethodno provjerenu verziju aplikacije u proizvodnju. Kod isporuke najčešće postoje etape koje se moraju zadovoljiti kako bi se provjerila izgrađena aplikacija za ručnu implementaciju ili ljudsku autorizaciju. Kontinuirana implementacija ubrzava razvoj softvera, ali međutim, takvo bi stajalište također moglo omogućiti da neotkriveni nedostaci ili ranjivosti prođu kroz testiranje i završe u proizvodnji. Za mnoge organizacije automatizirana implementacija predstavlja previše potencijalnih rizika za sigurnost i usklađenost poduzeća.[11]

2.2.4 Cjevovod kontinuirane integracije i isporuke te etape u njihovom postavljanju

Ispravno postavljanje cjevovoda isporuke i integracije ključ je za iskorištavanje svih prednosti koje ono nudi. Jedan cjevovod može imati strategiju implementacije u više faza koja isporučuje softver kao spremnike u oblaku npr. Kubernetes, a drugi može biti jednostavan cjevovod koji gradi, testira i postavlja aplikaciju kao funkciju bez poslužitelja[12]. Na sljedećoj slici prikazane su etape uspostave cjevovoda :

[11] https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know
[12] https://www.bmc.com/blogs/what-is-ci-cd/

Slika br. 4: Etape prilikom uspostave cjevovoda

Faze uspostave integracije i isporuke s prethodne slike su :

1.) Faza razvitka - u ovoj fazi se odvija razvoj aplikacije, a kod se spaja u spremište kontrole verzija te se provjerava njegova valjanost. Ovdje se razvijaju nove značajke i ponašanje aplikacije s obzirom na klijentske zahtjeve. Odabir alata u ovoj fazi ovisi o tome da li programeri rade u Javi, .NET, C# ili mnogim drugim programskim jezicima. Integrirana razvojna okruženja često se odabiru jer podržavaju određeni jezik uz razne značajke provjere koda, kao što su osnovno otkrivanje pogrešaka, skeniranje ranjivosti i pridržavanje utvrđenih standarda kodiranja. Ovdje je također bitno napomenuti alate za kontrolu verzije sustava kao što je npr. git.
2.) Faza izgradnje - aplikacija je izgrađena pomoću provjerenog koda, a kao proizvod dobiva se *artefakt* koji se koristi za testiranje. Proces izgradnje izvlači izvorni kod iz spremišta, uspostavlja veze za relevantne programske biblioteke, module i ovisnosti te izgrađuje sve te komponente u izvršnu (.exe) datoteku. Alati koji se koriste u ovoj fazi također generiraju zapisnike procesa, označavaju pogreške koje treba istražiti i ispraviti te obavještavaju programere da je izgradnja dovršena. Kao i kod stvaranja izvornog koda, alati za izgradnju obično ovise o odabranom programskom jeziku
3.) Faza testiranja – Prethodno dobiveni *artefakt* implementira se u testno okruženje, a provode se opsežni testovi kako bi se osigurala funkcionalnost

aplikacije. Testiranje automatizacije putem kontinuirane integracije može programerima uštedjeti ogromno vrijeme i trud. Kontinuirana integracija i automatizirani testovi pružaju brže povratne petlje i lakše otkrivaju probleme u novom kodu koje programeri možda nisu uspjeli predvidjeti[13].

4.) Faza isporuke - Ovo je posljednja faza cjevovoda, gdje se testirana aplikacija postavlja u proizvodno okruženje. U kontinuiranom cjevovodu isporuke, konačni proizvod šalje se ljudskim dionicima, odobrava i zatim raspoređuje. U cjevovodu kontinuirane isporuke, izgradnja se automatski implementira čim prođe testnu fazu. Ovdje se odvija osiguravanje resursa i usluga unutar podatkovnog centra i premještanje izgrađene aplikacije na neki od poslužitelja, a ova faza je također značajno automatizirana.

2.2.5 Benefiti kontinuirane integracije i isporuke

Primarni cilj integracije i isporuke pomoću cjevovoda je brzo i često isporučiti valjani softver korisnicima. Koristeći agilne i DevOps tehnike transformiraju razvojni proces i isporuku komponenti klijentu. Benefiti se ostvaruju u pogledu :

1.) Bolja kvaliteta koda - budući da razvojni tim objavljuje kod u malim serijama, on se može temeljito testirati - na primjer, korištenjem testiranja komponenti, koje programerima omogućuje da otkriju i poprave najozbiljnije greške prije implementacije softvera u proizvodnju[14]. Automatizacija testova osigurava njihovu dosljednu provedbu, čineći rezultate pouzdanijima. Budući da se automatizirani testovi izvode brže nego njihovi ručni ekvivalenti, postaje izvedivo testirati mnogo češće[15]. Redovito i temeljito testiranje koda znači da će se prije otkriti greške, što olakšava njihovo ispravljanje jer je na njima ugrađeno manje funkcionalnosti,a s vremenom to rezultira boljom kvalitetom koda.

2.) Povećana komunikacija – cjevovod kontinuirane integracije i isporuke je zajednički okvir za mnoge programere, voditelje projekata i testere koji rade na

[13] https://www.delphix.com/glossary/what-is-ci-cd-pipeline
[14] https://codete.com/blog/business-value-of-ci-cd
[15] https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/

razvoju softvera. Štoviše, ako tim slijedi ovaj proces, dobiva se na sigurnosti da programeri preuzimaju veću odgovornost i inicijativu. Svi dionici su obaviješteni za sve promjene , čak i u slučaju neuspjeha. To omogućuje vlasnicima proizvoda i programerima da učinkovito komuniciraju o rezultatima testiranja i poduzmu željene radnje na temelju kritičnosti u pogreškama softvera.[16]

3.) Povećana kreativnost razvojnog tima i zadovoljstvo klijenata - redovitom isporukom novih značajki, ispravljanjem grešaka odmah i reakcijom na povratne informacije oduševit će se krajnje korisnike aplikacije. To pomaže u odlučivanju o dojmu proizvoda na korisnika. Implementacija integracije i isporuke također omogućuje uključivanje krajnjeg korisnika i povratne informacije tijekom kontinuiranog razvoja koji dovodi do poboljšanja upotrebljivosti. Korištenjem računala za obavljanje repetitivnih zadataka, automatizirani proces također oslobađa pojedince da budu kreativni. Umjesto da se slijede ručne testirane skripte, ili postavljaju ažuriranja, razvojni tim može se usredotočiti na rješavanje problema i eksperimentiranje s rješenjima.[17]

4.) Smanjenje troškova korištenjem alata za automatizaciju i praćenje trenutnog napretka - automatizacijom radnji možemo biti sigurni da one budu manje sklone ljudskim pogreškama i da je njima lakše upravljati. U idealnom scenariju, svaka pojedinačna implementacija u proizvodno okruženje trebala bi se izvesti bez ikakve ljudske intervencije. Tu nam jako pomaže kontinuirana implementacija. Mnogi od dostupnih alata za podršku automatiziranom cjevovodu za integraciju i isporuku također potpomažu cijeli razvojni proces, pružajući niz metričkih vrijednosti od vremena izrade do pokrivenosti testiranja, stopa kvarova do vremena testiranja popravka. Na ovaj način možemo identificirati područja koja bi mogla zahtijevati pozornost kako bi mogli nastaviti poboljšavati svoj cjevovod. Sporija „izgradnja“ koda može ukazivati na potrebu

[16] https://www.lambdatest.com/blog/benefits-of-ci-cd/
[17] https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/

za povećanjem kapaciteta, dok povećanje srednjeg vremena popravljanja može biti znak procesa ili kulturoloških problema[18].

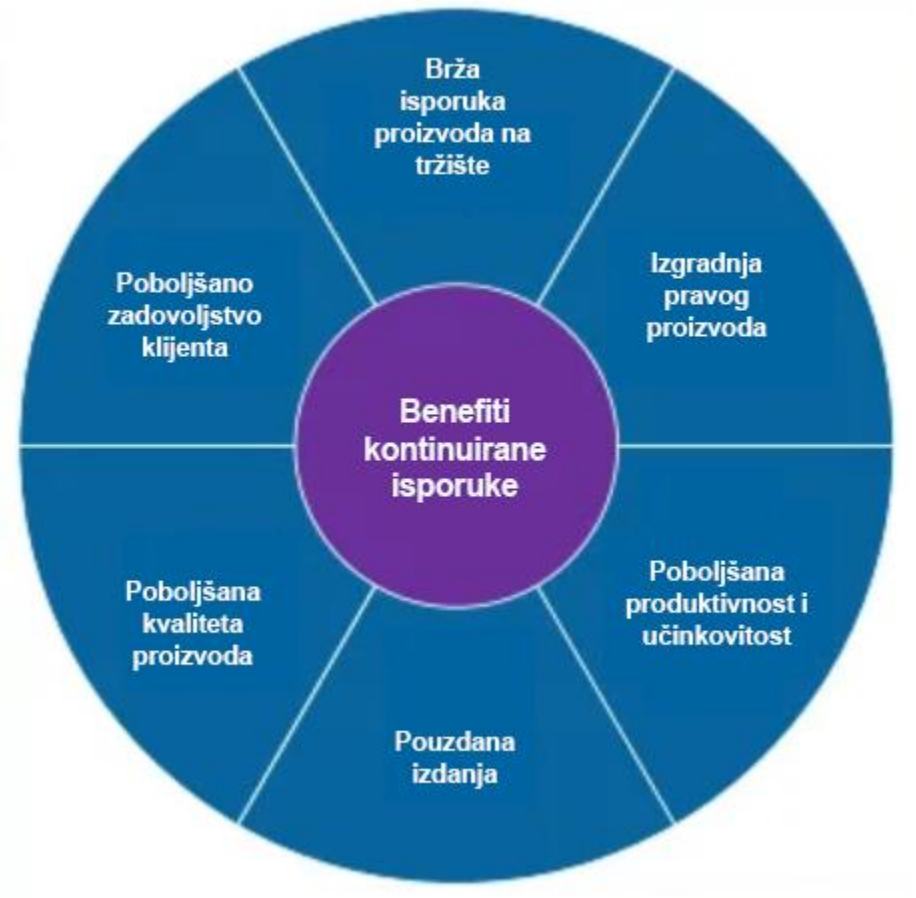
5.) Brze povratne informacije - dostupnost trenutnih povratnih informacija o kodu za novu verziju. Neuspjeh je normalna pojava, stoga vrijedi primijeniti princip brzog neuspjeha. To se može postići automatizacijom zadataka. Kada se greške identificiraju tada se mogu i brzo ispraviti.
6.) Česta ažuriranja i održavanje sustava - redovito održavanje i ažuriranja okosnica su stvaranja izvrsnog proizvoda, a to je jedna od velikih prednosti integracija i isporuke. Osigurava se da su ciklusi izdanja kraći i ciljani, što blokira manje značajki koje nisu spremne za izdavanje. U cjevovodu, održavanje se obično obavlja tijekom neradnog vremena, čime se štedi dragocjeno vrijeme za cijeli tim.

2.2.6 Ostali benefiti u pogledu kontinuirane isporuke

Na sljedećoj slici prikazani su benefiti kontinuirane isporuke. Kompanije motivirane ovim benefitima povećavaju svoje investicije u kontinuiranu isporuku. Najvažniji benefiti prikazani su : ubrzano vrijeme isporuke proizvoda na tržištu pri čemu se u prosjeku aplikacija isporučuje jednom tjedno. Nadalje izgradnja pravog proizvoda koja se očituje brzim povratnim informacijama od klijenata. Poboljšana produktivnost i učinkovitost gdje su nekada razvojni programeri provodili 20% svoga vremena postavljajući i popravljajući svoje okruženje, a danas je to u velikoj mjeri automatizirano. Pouzdana izdanja su također jedan od benefita kontinuirane isporuke pri čemu učestala testiranja postaju praksa kako bi se pogreške pronašle još u razvojnom okruženju. S češćim izdanjima broj promjena koda u svakom izdanju se smanjuje. Poboljšana kvaliteta proizvoda u kojima je broj pronađenih grešaka za aplikacije smanjen za više od 90 posto jedan je od važnih benefita.Kod kontinuirane isporuke, odmah nakon predaje koda, cijela baza koda prolazi kroz niz testova. Ako testovi pronađu problem, programeri ga popravljaju prije prelaska na drugi zadatak. Greške su toliko rijetke da timovima više nije potreban sustav za njihovo praćenje. Prije su kupci morali čekati sljedeće veliko izdanje kako bi se greške ispravile.

[18] https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/

Vremenski okvir je obično bio u trajanju od nekoliko mjeseci. Poboljšano zadovoljstvo klijenta - prije nego se kontinuirana integracija počela primjenjivati, postojalo je nepovjerenje i napetost između odjela korisnika i timova za razvoj softvera, zbog problema s kvalitetom i izdavanjem. Menadžeri su uočili da se odnos jako popravio, a povjerenje je vraćeno.



Slika br. 5: Benefiti kontinuirane isporuke

2.2.7 Izazovi kontinuirane isporuke

Osim benefita postoje i razni izazovi na koje treba obratiti pozornost, a među njima su: organizacijski izazovi, procesni izazovi, te tehnički izazovi od kojih su organizacijski ujedno i najteži.[19]

[19] https://www.infoq.com/articles/cd-benefits-challenges/

2.2.8 Organizacijski izazovi

Aktivnosti izdavanja uključuju mnoge odjele tvrtke. Svaka od njih ima svoje interese, načine rada i vlastitu percepciju kontrole svoga područja. Napetost je postojala između divizija zbog suprotstavljenih ciljeva. Na primjer, trebao je administratorski pristup poslužiteljima, a drugi je tim kontrolirao ovu dozvolu. Dolazak do rješenja u ovome slučaju uključivao je mnogo konzultacija i pregovora mjesecima.[20] Iako postoji literatura o organizacijskim promjenama, malo istraživanja se usredotočuje na uvođenje kontinuirane isporuke u organizaciju.

2.2.9 Procesni izazovi

Mnogi tradicionalni procesi ometaju isporuku. Na primjer, značajka koja je inače spremna za izdavanje mora proći kroz savjetodavni odbor za promjenu. To može odgoditi isporuku do četiri dana. Ako nekom novom poboljšanju aplikacije koje se implementira prođe samo nekoliko dana od implementacije do spremnosti za izdavanje, ovo razdoblje na kraju čini previše ukupnog vremena ciklusa značajke koja se implementira.

2.2.10 Tehnički izazovi

Korištenje mnoštvo alata i tehnika nije strano pri kontinuiranoj isporuci. Izbjegavanje propasti dobavljača jedan je od dodatnih izazova. Rad na razvoju široko prihvaćenih standarda, definiranju otvorenih API-ja i izgradnji aktivnog ekosustava dodataka pomoći će u ublažavanju izazova. Bavljenje aplikacijama koje nisu podložne kontinuiranoj isporuci kao npr. velike aplikacije također je izazovno. Ogroman broj takvih aplikacija postoji u industriji. Potrebno je dodatno istraživanje za razumijevanje njihovih karakteristika te identificiranje i razvoj najboljih strategija ili praksi za njihovo rješavanje.
Potrebno je istraživanje kako bi se identificirali ti procesi (koji pokrivaju područja poslovanja, razvoj softvera, operacije itd.) te razvili i provjerili alternative koje odgovaraju isporuci. Kako bi riješio organizacijske izazove, vodeći tim rekonstruira organizaciju kako bi razbio barijere među timovima i promovirao kulturu suradnje.

[20] https://www.infoq.com/articles/cd-benefits-challenges/

2.3 Testiranje klijentskih komponenti

Testiranje komponenti usredotočuje se na najmanje komponente aplikacije kao što su testiranje krajnje točke API-ja, testiranje funkcije itd. Iako testiranje komponenti omogućuje testeru da se usredotoči na pojedinačne komponente, važno je osigurati da svi dijelovi rade zajedno.Cilj testiranja integracije je osigurati da dijelovi rade zajedno. Premda testiranje integracije pomaže osigurati pouzdanost aplikacije, ono ne pokriva sve. Mi ne znamo kako aplikacija radi za krajnjeg korisnika. Zbog toga se rade testovi popularno nazvani „testovi *s kraja na kraj*“ ili E2E (skraćeno od eng. End to end) testovi. E2E testiranje uključuje tehnike koje simuliraju stvarnog korisnika aplikacije. Repliciranjem radnji koje bi korisnik poduzeo, test pomaže procijeniti jesu li rezultati u skladu sa zahtjevima ili očekivanim ishodom. [21] Primjeri uključuju testiranje korisničkog iskustva kao što su:

1.) Plaćanje proizvoda ili usluge na web stranici.
2.) Registracija na web stranici.
3.) Korištenje značajki aplikacije.

2.3.1 Tipovi E2E testova

Kod provođenja testova s „kraja na kraj“ postoje 2 glavna načina kako se oni odvijaju, a to su ručno i automatizirano testiranje. Ovisno o potreba organizacije, budžetu i organizacijskog kulturi kompanija se određuje na jedan od ta dva navedena tipa e2e testova. Samo korištenje alata za automatizaciju ne jamči uspjeh. Svaki novi alat uveden u organizaciju će zahtijevaju trud za postizanje stvarnih i trajnih koristi. Postoje potencijalne prednosti i mogućnosti s korištenje alata u testiranju, ali postoje i rizici. To se posebno odnosi na alate za izvršavanje testova (što se često naziva automatizacija ispitivanja).[22] O alatima za automatizaciju testova „*s kraja na kraj*“ bit će riječ poslije u radu u poglavlju 2.5, a sada će se prikazati razlike između ručnog (manualnog) i automatskog e2e testiranja.

[21] https://www.perfecto.io/blog/comprehensive-guide-end-end-e2e-testing
[22] Dorothy Graham, Erik van Veenedaal, Isabel Evans, Rex Black: Foundations of Software Testing: ISTQB Certification

2.3.2 Manualno i automatsko e2e testiranje

Ručno testiranje uključuje testera koji ručno provodi testove u kojemu ljudska pogreška može doprinijeti lošem testiranju. Za automatizirane testove, testovi se izvode automatski, odnosno putem nekog alata. Dakle, u smislu točnosti, automatizirani testovi mogu biti bolji u mnogim scenarijima. Ručno testiranje je lakše postaviti i razumjeti. U većini slučajeva, automatsko testiranje je složeno za postavljanje. Krivulja učenja je obično viša za automatizirano testiranje. Ručno testiranje na početku uključuje manje financijskih ulaganja. Iako automatizirano testiranje može uključivati više početnih ulaganja, ono nudi značajan povrat na investiciju (eng. ROI – Return Of Investment), dok se proces testiranja povećava i sazrijeva. [23] Dakle, ovo je nešto što treba uzeti u obzir pri odabiru alata za automatsko testiranje. U ručnom testiranju lakše je simulirati proizvodno okruženje, dok je kod nekih automatiziranih alata potrebno sprovesti određenu proceduru prilikom njihovog uključivanja u organizaciju. Na primjer, lako je osigurati da su ispunjeni neki uvjeti prije nego što neka pogodnost aplikacije radi ručno, ali za automatizirane alate za testiranje to može biti teško konfigurirati.

2.4 Benefiti automatizacije testova

Premda neke tvrtke radije preferiraju ručno testiranje, to nekada možda i nije najbolji izbor. Sljedeći logičan korak koji bi se trebao poduzeti je automatizacija procesa testiranja kada su testovi postavljeni. Ručno testiranje treba pod svaku cijenu svesti na minimum. Automatizacija testiranja povećava ukupnu učinkovitost razvoja softvera i omogućuje izradu snažnijih alata. U ovome dijelu bit će prikazani benefiti uvođenja automatskih testova u organizaciji. Neke od njih su brži ciklus povratnih informacija, ušteda vremena u timu, smanjenje poslovnih troškova, veća pokrivenost testova, ponovna iskoristivost test paketa, brže vrijeme do tržišta, bolji uvid u testove, poboljšana točnost, pružanje više značajki aplikacije povodom automatizacije, manje opterećenje za testni tim, brzo određivanje stabilnosti koda, te uklanjanje ljudske pogreške.

[23] https://www.perfecto.io/blog/comprehensive-guide-end-end-e2e-testing

2.4.1 Brži ciklus povratnih informacija

Bez automatizacije testiranja, povratne informacije za novorazvijene značajke mogu potrajati. Automatizacija testiranja pomaže smanjiti ciklus povratnih informacija i donijeti bržu provjeru valjanosti za faze u razvoju programskog proizvoda. Automatizacija testiranja posebno je korisna jer ona pomaže otkriti probleme ili pogreške u ranijoj fazi razvoja, što povećava učinkovitost tima.

2.4.2 Ušteda vremena u timu

Automatiziranjem postupka testiranja tim troši manje vremena na provjeru novorazvijenih značajki. Također poboljšava se komunikacija s drugim odjelima poput marketinga, dizajna ili vlasnika proizvoda koji se oslanjaju na rezultate testova. Ovi odjeli mogu jednostavno provjeriti zapisnike automatiziranih testova i vidjeti što se događa.

2.4.3 Smanjenje poslovnih troškova

Kada se koristi automatizirano testno okruženje tvrtka ušteđuje novac jer se manje resursa troši na testiranje proizvoda. Rješenje je da se ne radi nikakvo ručno testiranje. Tijekom cijelog projekta to može napraviti veliku razliku. Naravno, instalacija i postavljanje automatiziranog okruženja za testiranje zahtijeva vrijeme i resurse. Također, velike su je vjerojatnost da će se trebati dodatno platiti za odgovarajući alat za automatizaciju testiranja koji može pomoći u stvaranju stabilnog okruženja za automatizaciju testiranja.

2.4.4 Veća pokrivenost testova

Ručno testiranje postavlja ograničenja na broj testova koji se mogu provjeriti. Automatizacija omogućuje da se efektivno koristi vrijeme na pisanje novih testova i njihovo dodavanje u svoj automatizirani testni paket. To povećava pokrivenost testova za programski proizvod tako da se više značajki ispravno testiraju što rezultira većom kvalitetom aplikacije. Također, automatizirano testiranje omogućuje programerima da napišu detaljne testove koji testiraju složene slučajeve upotrebe. Dugi testovi koji se često izbjegavaju tijekom ručnog testiranja mogu se izvoditi bez nadzora.

2.4.5 Ponovna iskoristivost test paketa

U početku je izgradnja automatiziranog testnog paketa izazov. Međutim, nakon što se definira testni paket, vrlo je jednostavno ponovno koristiti testove za druge slučajeve upotrebe ili čak druge projekte. Prednost je u tome što se mogu jednostavno spojiti drugi projekti na vlastiti automatizirani testni paket. Stjecanjem znanja za postavljanje automatskih testnih paketa i postavljanje cjevovoda kontinuirane integracije može se jednostavno replicirati znanje kako bi se napravila automatizacija testiranja za novi projekt.

2.4.6 Brže probijanje na tržište

Nakon prethodno spomenutih benefita, novorazvijene značajke mogu se kontinuirano testirati i provjeriti pomoću automatizacije testiranja. To smanjuje ciklus povratnih informacija i testiranja i omogućuje tvrtkama da brže plasiraju svoje proizvode na tržište.

2.4.7 Bolji uvid u testove

Automatsko testiranje pruža bolji uvid od ručnog testiranja kada neki testovi ne uspiju. Automatizirano testiranje softvera ne daje samo uvid u aplikaciju, već ono pokazuje i sadržaj memorije, tablice podataka, sadržaj datoteka i druga unutarnja stanja programa. To pomaže programerima da utvrde što je pošlo po zlu.

2.4.8 Poboljšana točnost

Normalno je da i najveći stručnjaci griješe tijekom ručnog testiranja. Pogotovo kada se testiraju složeni slučaji upotrebe aplikacije (eng. Complex use case scenarios), može se doći do pronalaska kvarova. S druge strane, automatizirani testovi mogu izvršiti testove sa 100-postotnom točnošću jer daju isti rezultat svaki put kada se oni pokrenu.

2.4.9 Pružanje više značajki aplikacije povodom automatizacije

Automatizirani testni paket može pružiti više pogodnosti ili značajki nama, na primjer, simuliranjem tisuća virtualnih korisnika u interakciji s web aplikacijom kako bi se došlo do uvida kako se aplikacija ponaša. Nemoguće je simulirati ovakvo ponašanje ručnim testiranjem, a ovakve značajke štede programerima puno vremena.

2.4.10 Manje opterećenja za testni tim

Implementacijom automatizirane strategije testiranja testni tim i testeri za pružanje kvalitete koriste vrijeme na efektivne zadatke osim ručnog testiranja. Osim toga, opis posla QA (skraćeno od eng. Quality Assurance) inženjera postaje zanimljiviji kada se uklone ponavljajući elementi ručnog testiranja. Za mnoge QA inženjere, automatizacija testiranja stvara priliku za izgradnju novih alata za daljnju optimizaciju postojećeg testnog paketa ili ga proširuje novim značajkama.

2.4.11 Brzo određivanje stabilnosti koda

Automatizacija testiranja pomaže da se automatiziraju testovi kako bi se utvrdila stabilnost izgrađenog koda ili aplikacije. Često se ispitivanje dima (eng. Smoke testing) koristi za provjeru stabilnosti. Međutim, ispitivanje dima je sporo i zahtijeva ručni unos od inženjera za ispitivanje, kao što je postavljanje baze podataka s podacima ispitivanja. Nadalje, ispitivanje dima može se automatizirati putem automatizacije ispitivanja. Može se automatski generirati i pripremiti prave baze podataka za pokretanje testova dima i prema tome se može brzo odrediti stabilnost koda ili aplikacije. Ukratko, cilj je moći objaviti *build* što je brže moguće i automatski potvrditi njegovu stabilnost.

2.4.12 Uklanjanje ljudske pogreške

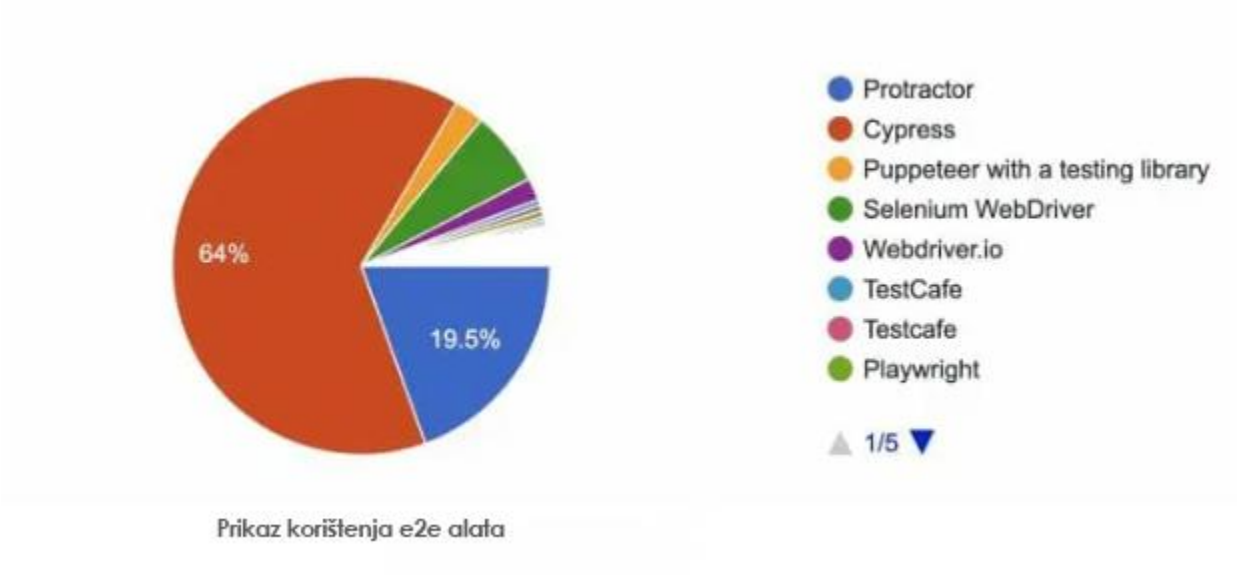
Ručno testiranje otvara mogućnost ljudima da pogriješe. Pogotovo za složene scenarije ima smisla koristiti automatizaciju testiranja kako bi se izbjegle pogreške. I dalje se može pogriješiti, čak i uz automatizaciju testiranja. Međutim, stopa pogrešaka znatno je niža kada se koristi automatizacija testiranja za neki testni paket.[24]

2.5 Alati za testiranje klijentskih komponenti

Iako je korištenje alata otvorenog koda obično bolja opcija od izgradnje vlastitog razvojnog okvira od početka, ne može se reći da su alati za automatizaciju testa otvorenog koda sve što je nekoj organizaciji potrebno. Kada se shvati kako odabrati pravi alat za automatizaciju testiranja za svaku ulogu u organizaciji, možda će organizacija na

[24] https://www.testim.io/blog/test-automation-benefits/

kraju završiti s mješavinom komercijalnih opcija i opcija otvorenog koda.[25] Na slici br. 6 prikazani su neki od najpopularnijih alata za testiranje klijentskih komponenti *s kraja na kraj*. Na njoj su prikazani udjeli korištenja pojedinih alata pa se primjećuje da se u najvećem postotku koristi Cypress i Protractor, te Selenium WebDriver, ali ne treba ni zanemariti korištenje i ostalih alata i razvojnih okvira kao što su „Puppeteer with a testing library“, „Playwrigth“, „Testcafe“. U nastavku su prikazane pogodnosti nekih od popularnih alata za testiranje *s kraja na kraj*.



Slika br. 6: Udio korištenih e2e alata u 2021. prema Protractor Githubu

2.5.1 Cypress

Cypress je jedan od razvojnih okvir koji testerima značajno olakšava posao te smanjuje probleme s kojima se susreću tijekom korištenja npr. Selenium razvojnog okvira zbog nešto drugačije arhitekture. Dok se Selenium može koristiti u testiranju komponenti i testiranju *s kraja na kraj*, Cypress se fokusira samo na testiranje *s kraja na kraj*. Cypress je svestran i učinkovito radi na aplikacijama izgrađenim pomoću različitih okvira kao što su React, Angular, itd. On ne koristi nikakav upravljački program ili kod na daljinu nego se testovi provedeni pomoću Cypressa izvode u JavaScriptu u web pregledniku. Kod korištenja Cypressa, pisanje i pokretanje testova i otklanjanje pogrešaka veoma je

[25] https://techbeacon.com/app-dev-testing/top-11-open-source-testing-automation-frameworks-how-choose

jednostavno. Ako testeri preferiraju razvoj temeljen na testu tada će Cypress biti od velike pomoći jer je napravljen da podržava takvo nešto. Može se brže razviti vlastitu aplikaciju jer čak i kada se testovi odvijaju, izmjene se mogu unijeti i vidjeti promjene koje se događaju u stvarnom vremenu.[26]

2.5.2 Cucumber

Za razliku od Cypressa, Cucumber se usredotočuje na razvoj vođen ponašanjem (eng. BDD – Behaviour-driven Development). To omogućuje izgradnju proizvoda uzimajući u obzir perspektivu različitih dionika kao što su programeri, QA, kupci itd. Glavni razlog popularnosti Cucumbera je njegova jednostavnost. Cucumber koristi Gherkin, što je vrlo slično pisanju izjava na engleskom. Za ljude koji ne vole pisati testove u složenim kodovima, Cucumber je rješenje, te on olakšava čak i netehničkoj osobi pisanje testnih skripti. To također pomaže netehničkim ljudima koji ne razumiju osnovno programiranje da razumiju testove.

2.5.3 Playwright

Playwright je alat za testiranje *s kraja na kraj* iz Microsoftovog tima. Omogućuje automatizaciju modernih web aplikacija u svim preglednicima kao što su Chrome, Firefox i Webkit. Playwright je zapravo prilično sličan Puppeteeru, gdje su neki od istih ključnih članova koji su izgradili i održavali Puppeteer zapravo prešli u Microsoft iz Googlea kako bi stvorili Playwrighta. Dok je Puppeteer bio samo alat baziran na Chromiumu, Playwright se uključuje i pruža podršku za sve moderne preglednike uključujući Google Chrome i Microsoft Edge pomoću Chromiuma, Apple Safari pomoću WebKita i Mozilla Firefox. Pomoću Playwright-a može se odabrati na kojem pregledniku se žele pokrenuti testovi (ili više preglednika) i izdati neke jednostavne naredbe koje će Playwright slijediti. Cucumber podržava različite jezike i lako se postavlja dok se Playwright usredotočuje na iskustvo krajnjeg korisnika i stoga je dobra opcija za testiranje *s kraja na kraj*[27].

[26] https://www.testim.io/blog/end-to-end-testing-frameworks/
[27] https://applitools.com/blog/migrating-protractor-tests-angular/

2.5.4 Testim

Testim je jedan od najsofisticiranijih, ali jednostavnih alata za korištenje razvojnih okvira *s kraja na kraj* za testiranje. Pogodan je za sve vrste testera, za one koji vole kodirati, ili pak one koji preferiraju drugačije načine. Testim omogućuje testiranje bez pisanja ijedne linije koda. On ima izvrsnu značajku testiranja snimanja i reprodukcije koja koristi složene algoritme i umjetnu inteligenciju za pokretanje testova na aplikaciji. Pruža vrlo jednostavno korisničko sučelje bez kompromisa u kvaliteti testiranja. Testim je dovoljno pametan da testira složene dinamičke aplikacije koristeći svoju umjetnu inteligenciju gdje mnogi drugi alati ne uspijevaju. U njemu je moguće pronaći više opcija koje olakšavaju testiranje, kao što su provjera valjanosti, snimanje zaslona, povratne informacije itd. Testim pruža paralelno izvođenje, tzv "*webhookove*" i integraciju s drugim okvirima za stvaranje okruženja za testiranje s kraja na kraj.

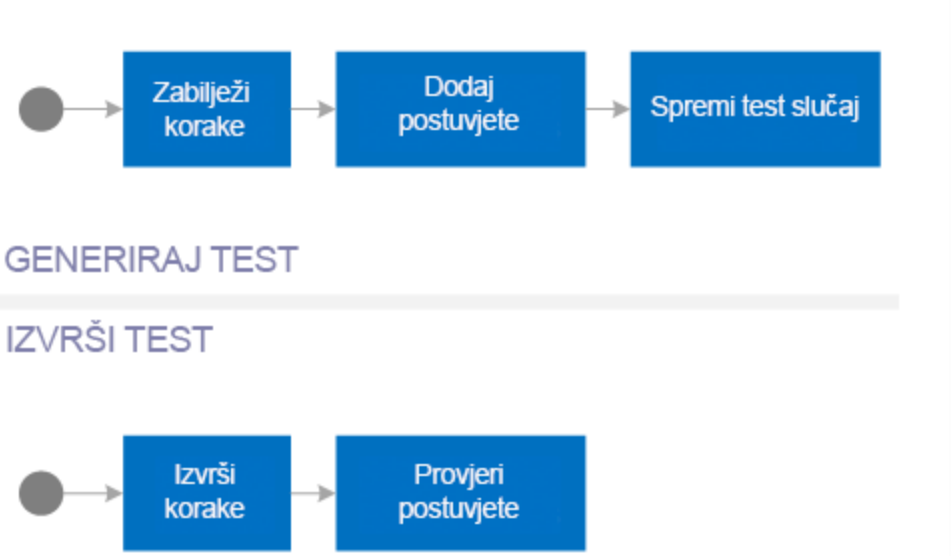
2.5.5 Protractor

Protractor je okvir za testiranje automatizacije otvorenog koda koji je napisan pomoću NodeJS-a. Nudi kombinirano testiranje *s kraja na kraja* za web aplikacije koje su izrađene pomoću AngularJS-a, a podržava i Angular i Non-Angular aplikacije. Ali budući da se može koristiti za testiranje naprednih HTML atributa, Protractor je široko poželjan za testiranje AngularJS-a. Koristi snagu različitih tehnologija kao što su NodeJS, Selenium Webdriver, Jasmine, Mocha, Cucumber, itd. kako bi ponudio snažan testni paket za automatizaciju koji je sposoban za obavljanje unakrsnog testiranja za web aplikacije. Protractor je napravljen oko Selenium Webdrivera koji pruža okvir za testiranje automatizacije, te koji simulira interakciju korisnika s Angular web aplikacijom za niz preglednika i mobilnih uređaja. On pruža sve značajke Selenium WebDriver-a sa specifičnim značajkama Angulara za besprijekorno testiranje *s kraja na kraja*. Protractor koristi JSON Wire protokol sličan protokolu Selenium WebDriver-a kako bi omogućio interakciju korisnika s preglednikom.[28]

[28] https://www.browserstack.com/guide/protractor-testing-tutorial

2.6 Alati za testiranje bez kodiranja

Jedan od glavnih nedostataka kreiranja tzv. automatiziranih e2e testova je količina koda potrebna za implementaciju raznih značajki aplikacije. U mnogim slučajevima ljudi koji su najzainteresiraniji za e2e testove nisu programeri (tj. oni koji poznaju kod ili ga zanimaju). Umjesto toga, često su dionici na poslovnoj razini – kao što su kupci, poslovni rukovoditelji ili prodajno osoblje koje je ujedno najzainteresiraniji za stvaranje i gledanje rezultata e2e testova. Osim klasičnih pogodnosti koje pruža npr. Gherkin, novija rješenja pronalaze se u testiranju bez koda. Testiranje bez koda je pristup u kojem se testni kod generira automatski na temelju niza snimljenih koraka. Na primjer, ako prodavač želi osigurati da uklanjanje artikla iz košarice, a zatim odjava ne rezultira kupnjom uklonjene stavke, može koristiti alat bez kodiranja kako bi uhvatio korake potrebne u korisničkom sučelju za uklanjanje artikla a zatim može odraditi odjavu. Prodavač tada može dodati zahtjev da uklonjeni artikl nije kupljen i slično. Alat bez koda zabilježit će radnje koje je izvršio prodavač u korisničkom sučelju kao skup diskretnih, poredanih koraka. Neki alati mogu čak dopustiti da se koraci izmijene ili preurede. Dodatni testni zahtjevi kao što je uklonjeni predmet koji se ne kupuje također bit će zabilježeni kao skup naknadnih uvjeta. Kada alat bez koda izvrši testni slučaj, on jednostavno izvršava skup određenih koraka i provjerava rezultate u odnosu na snimljene postuvjete. Ako su svi postuvjeti zadovoljeni, testni slučaj prolazi. Često je skup koraka i postuvjeta vizualno predstavljen u korisničkom sučelju, što omogućuje kategoriziranje i pregledavanje testnih slučajeva te pregled koraka i postuvjeta za pojedinačne testne slučajeve. Prethodno rečeno najbolje je prikazano na slici br. 7.



Slika br. 7: Prikaz koraka za izvođenje testova bez koda

Jedan od nedostataka automatskog testiranja bez koda je krhkost testa. Na primjer, ako je prodavač kliknuo gumb „Naplata“ na korisničkom sučelju aplikacije, alat bez koda trebao bi zabilježiti ovaj pritisak na gumb. Ako je alat zabilježio pritisak na ovaj gumb koristeći položaj gumba na mjestu X i Y na vidljivom zaslonu, pomicanje gumba u budućnosti moglo bi uzrokovati neuspjeh testa, jer se gumb više neće nalaziti na istoj koordinati X i Y na ekranu[29]. U nastavku su prikazani neki od popularnih alata za testiranje bez koda.

2.6.1 AccelQ

AccelQ automatizira web, desktop, mainframe i druge aplikacije uz minimalno vrijeme i trud, ubrzavajući ciklus testiranja. AccelQ privlači posebnu pozornost timova za testiranje i agilni pristup sa svojim prirodnim načinom kodiranja na engleskom. Može se integrirati s popularnim i postojećim alatima kao što je DevOps kako bi se pružio jedinstveni pogled na cijeli životni ciklus inženjeringa kvalitete. Kako bi poslovnim korisnicima i stručnjacima za domenu osigurao istinsko iskustvo bez koda, AccelQ nudi jednostavno korisničko sučelje koje se brzo razvija. Sa značajkama kao što su programiranje pomoću prirodnog

[29] https://dzone.com/refcardz/end-to-end-testing-automation-essentials

jezika, inteligentni istraživač elemenata, i automatiziranim generiranje testova, trenutačno je popularan među raznim organizacijama. Neke od njegovih glavnih pogodnosti su : generiranje nacrta aplikacije za bolje vizualne dizajne testova, brži razvoj i manje održavanje, sposobnost predviđanja i analize puta za razvoj testnih scenarija, maksimalno povećanje pokrivenosti testom pomoću korisničkog sučelja modela i tokova podataka, skalabilnost i sigurno korištenje u poduzeću, spremnost na kontinuiranu integraciju i automatizaciju u sprintu.

2.6.2 Katalon Studio

Prema izboru kupaca za 2020. u automatizaciji testiranja softvera i rangiran je kao alat za najbolje automatizirano testiranje softvera. Katalon je popularan izbor zbog svog širokog spektra mogućnosti testiranja preko API-ja, weba, mobilnih i desktop aplikacija. Značajke Katalon studija jednostavne su za korištenje za početnike i iskusne testere putem jednostavnog korisničkog sučelja. Neke od glavnih pogodnosti su automatizacija testiranja weba, API-ja, mobilnih i stolnih računala, a on dolazi s ugrađenim predlošcima projekta, bibliotekama testnih slučajeva, ključnim riječima i spremištima objekata. Od ostalih benefita on automatski generiraj testove na temelju promjena korisničkog sučelja te može locirati promjene u objektu i u skladu s tim generirati testove, a podržava izvorne dodatke za popularne alate kontinuirane integracije, te ugrađuje integraciju kao što su upravljanje testiranjem, upravljanje problemima, te obavijesti i komunikacije.

2.6.3 Cloud QA

S porastom tzv. razvoja u „oblaku“, agilni timovi traže ekskluzivna rješenja za testiranje temeljena na oblaku. Cloud QA ističe se kao platforma za automatizaciju testiranja bez koda temeljena na oblaku. Platforma jamči da je mnogo više od alata za snimanje i reprodukciju i pruža integrirano iskustvo testiranja *s kraja na kraj*. Njegovi ugrađeni alati za izvješćivanje brzo stvaraju i zakazuju pakete za regresijsko testiranje i mogu razvijati, graditi, održavati i izvršavati testove. Jedne od glavnih značajki su izgrađenost pomoću Seleniuma, lidera u prijenosnom okviru za testiranje web aplikacija, a on je platforma *s kraja na kraj* za kreiranje, uređivanje, održavanje, pa čak i dopuštanje ponovne upotrebe

komponenti za testiranje. Od ostalih značajki valja spomenuti besprijekornu integraciju s popularnim praćenjem grešaka, kontinuiranom integracijom, isporukom te mnogim drugima. On djeluje na infrastrukturi oblaka uz minimalno vrijeme i napore za postavljanje, te podržava širok raspon web aplikacija i okvira i ima robusne značajke testiranja s pametnim lokatorima i mogućnošću paralelnog pokretanja mnoštva testova.

2.6.4 Perfecto

Perfecto je jedna od vodećih platformi za testiranje web i mobilnih aplikacija. To je rješenje za automatizaciju temeljeno na oblaku s više podržanih uređaja, preglednika i kombinacija operacijskih sustava za testiranje mobilnih i web aplikacija. Perfecto ima mogućnost umjetne inteligencije za klasifikaciju pogrešaka, značajke koje podržavaju kontinuirano ažuriranje testnih skripti i njihovo pokretanje uz minimalno održavanje. Perfecto pokušava poboljšati digitalno iskustvo optimiziranjem kontinuiranog testiranja u procesima kontinuirane integracije i isporuke. Značajka temeljena na oblaku omogućuje testerima suradnju s bilo kojeg mjesta. Napredne mogućnosti izvješćivanja osiguravaju pronalaženje grešaka i detaljnu analizu popravaka u ranim fazama razvoja programskog proizvoda. Prema Forresteru, “Perfecto je snažan izbor za poduzeća, s poboljšanim mogućnostima paralelnog testiranja i izvođenjem testova performansi na visokoj razini uz sigurnost na razini poduzeća.”[30]

2.6.5 Curiosity Software

Curiosity Software nudi rješenje platforme otvorenog testiranja s integriranim modeliranjem testova i testne automatizacije. To timovima za testiranje daje slobodu da ga integriraju s bilo kojim alatom za automatizaciju testiranja po svom izboru, Curiosity Software se brine za automatsko generiranje testnih podataka te primjenjuje koncept modeliranja testova. Koncept modeliranja testova prevladava izazove generiranja točnih testnih skripti i podataka pomoću vizualnih modela. On se sastoji od skenera korisničkog sučelja, uvoznika testova i testnih modela koji se mogu ponovno koristiti za izradu

[30] https://www.qentelli.com/thought-leadership/insights/top-codeless-testing-tools

dijagrama toka. To stvara nove testove za sustave koji se testiraju, generirajući testove *s kraja na kraj* omogućujući timovima za testiranje da rade na smislenim zadacima. Poslovni testeri i programeri mogu generirati testove iz biblioteka kodova za ponovno korištenje. Vizualni modeli pomažu onima koji nisu testeri da generiraju testne pakete za nove kodove iz postojećih predložaka koda. To omogućuje kontinuirano testiranje za svaki sprint, eliminirajući kašnjenja u ciklusima isporuke proizvoda. Neke od glavnih značajki su mogućnosti temeljene na modelu gdje algoritmi temeljeni na umjetnoj inteligenciji koriste logiku za generiranje testova. Izrada testnih podataka u sprintu i potpune testne podatke u skladu je s promjenjivim korisničkim pričama. Sam alat je prikladan za agilno testiranje s generiranjem testnih podataka na vrijeme i automatizacijom reaktivnog testiranja. Ubrzavanje ciklusa testnih podataka s bogatim skupovima podataka za ciklus kontinuirane integracije i isporuke samo su još jedna od mnogih pogodnosti ovoga alata.

3. Pregled alata testcafe i studije slučaja

3.1 Kratka povijest testcafe-a

TestCafe je nastao zbog toga što su programeri imali problema s pokušajem automatizacije pomoću Seleniuma. Ono što se dogodilo prije otprilike osam godina je da su počeli pisati mnoge kontrole korisničkog sučelja za web. U početku je to bio samo ASP.net. U biti, aplikacije na strani poslužitelja i neke stvari na strani klijenta koje su se događale bile su relativno minimalne, i nitko se uistinu nije brinuo o testiranju tog dijela. Zatim su počeli ulaziti u HTML 5 i JavaScript stranu stvari, i tamo su počeli nailaziti na probleme u vezi s testiranjem. Budući da postoji mnogo preglednika koje su morali podržati, bio im je potreban način da automatiziraju svoje testiranje. Prvo su isprobali Selenium u kojemu nisu pronašli mnogo sreće. Zatim su odlučili isprobati da napišu vlastiti alat za testiranje za web. Zbog potrebe za testiranjem čestih isporuka aplikacija i testiranja mobilnih uređaja, trebao im je alat za testiranje koji bi mogao raditi na iOS-u, Safariju i Androidu, a tada ih je bilo teško pronaći. Tako su stvorili TestCafe i odlučili da je dovoljno dobar za prodaju kao proizvod, te se do danas zadržao u velikom broju kompanija.[31]

3.2 Jednostavne pogodnosti testcafe-a

Testni kontroler od testcafe-a uključuje akcije kao svoje metode. Ove radnje mogu se ulančati kada ih se pozove u kodu te se pomoću njih mogu lagano odvijati događaji kao što su klikanje, upisivanje teksta, navigacija, učitavanje datoteka. Osim toga testcafe ima i razne svoje selektore pomoću kojih pronalazi sve DOM elemente te nad njima može vršiti interakciju na način kao što bi i obični korisnik aplikacije to činio. Na sljedećim slikama prikazani su u kodu neki od najučestalijih događaja koji se koriste u testcafe-u.

[31] https://testguild.com/testcafe/

```
import { Selector } from 'testcafe';

fixture `Example`
    .page `https://devexpress.github.io/testcafe/example/`;

test('Press Key test', async t => {
    await t
        .click('#tried-test-cafe')
        // pressing 'Space' imitates clicking the checkbox again
        .pressKey('space')
});
```

Slika br.8: Prikaz klika na element

Na slici br.8 prikazan je tipičan klik na element na nekoj web aplikaciji u kojemu se najprije pomoću selektora u ovome slučaju preko ID-a *#tried-test-cafe* pronalazi element koji se predaje kao argument događaju za klikanje, a zatim se emitira *„space“* događaj koristeći metodu *pressKey('space')*. Na sličan način prikazani su i eventi na slici br. 9. Na njoj vidimo kako se najprije odabire selektor *#developer-name* i u njega se upisuje tekst „John Doe“, a zatim se odabire prethodno upisani tekst koristeći metodu *selectText()*, pa se daje naredba za brisanje teksta koristeći prethodno objašnjenu metodu *pressKey* koja prima argument '*delete*' u ovom slučaju.

```
import { Selector } from 'testcafe';

fixture `Example`
    .page `https://devexpress.github.io/testcafe/example/`;

test('Select Text test', async t => {
    await t
        .typeText('#developer-name', 'John Doe')
        .selectText('#developer-name')
        .pressKey('delete');
});
```

Slika br. 9: Prikaz događaja za upisivanje teksta, odabir teksta te brisanje istog

3.2.1 Načini provjere rezultata testova[32]

Provjera (eng. Assertion) se koristi za provjeru rezultata testa s očekivanim ili stvarnim rezultatima. U automatizaciji testcafea, „tvrdnja“ je stanje u kojemu se provjeravaju rezultati samog testnog scenarija. Postoji mnogo načina za korištenje tvrdnji. Jedni od najpoznatijih primjera prikaza tvrdnji opisani su ispod na slikama. Funkcija *t.expect()* koristi se za izvođenje tvrdnji u testcafeu. Ona vraća istinu ili laž ovisno o ulaznim parametrima. Sljedeća slika provjerava tvrdnju da li je broj 5 veći od broja 2.

```
1.  t.expect(5).gt(2) //Will return true
```

Slika br. 10: Primjer jednostavne expect funkcije

Na prethodnoj slici funkcija *gt()* označava tvrdnju za “veće od” nekog izraza. Skraćenica gt dolazi od engleske riječi (“Greater than”) te ona vrijedi za usporedbu teksta, brojeva i datuma. Na sličan način može se provjeriti tvrdnja za „veće ili jednako“, te će u tom slučaju funkcija biti *gte()*, od engleske riječi („Greater than or equal“). Idući primjer prikazuje malo kompleksniji scenarij u kojemu se najprije pronalazi element preko selektora za kojega želimo ustvrditi neku istinitost te zatim koristeći *expect* funkciju najprije provjeravamo da li je naslov jednak tekstu „Example“, nakon kojega slijedi tvrdnja za provjeru da li je broj 5 jednak broju 5 pa dajemo naredbu za implicitno čekanje u trajanju od 5 sekundi.

```
1.  import { Selector } from 'testcafe';
2.
3.  fixture `Testcafe Automation tutorial`
4.      .page `https://devexpress.github.io/testcafe/example/`
5.
6.  test('Testcafe Example for t.expect.eql', async t => {
7.      const PageHeadingEle = Selector('h1');
8.      var pageHeading = await PageHeadingEle.innerText
9.      console.log("Actual Page Heading: "+pageHeading)
10.     await t .expect(pageHeading).eql('Example')
11.     await t .expect(5).eql(5)
12.     .wait(5000)
13. });
```

Slika br. 11: Kompleksniji način provjere tvrdnji

[32] https://codedec.com/tutorials/assertion-in-testcafe/

Na slici br. 11 prikazan je još jedan primjer provjere koji koristi funkciju *eql()* koja prima jedan argument. Ona provjerava da li je nešto jednako onome što se tvrdi. Također još jedna bitna funkcija kada se ulanča npr. *t.expect.notContains()* koristi se u testcafe-u za provjeru da jedan niz ne sadrži drugi podniz. On vraća istinu ako tekst ne sadrži podniz i neistinu ako vrijednosti sadrže bilo koji podniz. Na sljedećem primjeru opisano je korištenje te funkcije.

```
1.  import { Selector } from 'testcafe';
2.
3.  fixture `Testcafe Automation tutorial`
4.      .page `https://devexpress.github.io/testcafe/example/`
5.
6.  test('Testcafe Example for t.expect.notContains', async t => {
7.      const PageSubHeadingEle = Selector('p').withText('This webpage is used as a sample in
    TestCafe tutorials.')
8.      var pageHeading = await PageSubHeadingEle.innerText
9.      console.log("Actual Page Heading: "+pageHeading)
10.     await t .expect(pageHeading).notContains('Hello Learner!')
11.
12.     .wait(5000)
13. });
```

Slika br. 12: Primjer korištenja notContains funkcije

Na slici br. 12 prikazan je test koji objašnjava da li tekst „Hello Learner“ nije sadržan u tekstu elementa „PageSubHeadingEle“ , nakon čega se također daje naredba za implicitno čekanje u trajanju od 5 sekundi.

```
1.  import { Selector } from 'testcafe';
2.
3.  fixture `Testcafe Automation tutorial`
4.      .page `https://devexpress.github.io/testcafe/example/`
5.
6.  test('Testcafe asserts to verify match regex', async t => {
7.
8.      await t .expect('codedec').notMatch(/^N/)
9.      .wait(5000)
10. });
```

Slika br.13: Korištenje regularnih izraza

Na prethodnom primjeru prikazano je provjere testnih tvrdnji korištenjem regularnih izraza u kojem se provjerava da li tekst „codedec“ ne sadrži nijedan element sa slovom „n“. Od ostalih važnih načina provjere tvrdnji dobro je istaknuti svojstvo „*exists*“ koje nam govori

da li traženi element postoji na web stranici, te svojstvo „*visible*“ koje provjerava da li je element na web stranici vidljiv te vraća istinu ili laž. Također jedno od često korištenih svojstava je „*within*“ koje provjerava da li se neka vrijednost nalazi u rasponu vrijednosti u kojem prvi argument određuje donju granicu, a drugi gornju. Prema tome *within(8,12)* nam određuje 8 kao donju granicu, odnosno 12 kao gornju granicu traženog izraza.

3.2.2 Moderan način izgradnje testova u testcafeu korištenjem modela objekta stranice

Model objekta stranice (eng. Page Object Model) sastoji se od selektora za odabir elemenata, URL adresa i konstantnih vrijednosti, koje se mogu koristiti u zasebnoj datoteci. Iz te datoteke odabiru se metode i elementi koji se nalaze u modelu objekta stranice, a zatim se dodjeljuje testna radnja koja će se izvršiti na svakom elementu. Zahvaljujući ovom pristupu lako je održavati testove, jer ako postoji potreba za nekim promjenama, sam test će ostati netaknut, a bit će moguće samo mijenjati selektore ili URL adrese na jednoj stranici tj. modelu objekta stranice. Ovakav način razvoja koristiti će se i u praktičnom primjeru kada budu prikazani test slučajevi u kodu. Ovamo postoje 2 bitna koraka kako se testiranje organizira a to su : izgradnja modela objekta stranice i njihovo korištenje u testnim slučajevima.

1.) Izgradnja modela objekta stranice – izgradnja se odvija tako da se u posebnoj datoteci definira model objekta koji sadrži selektore za odabir elemenata s web aplikacije, a taj model može imati i svoje metode. Slika br.14 prikazuje jedan od takvih objekata.

```
1  import { Selector } from 'testcafe';
2
3  export default class Page {
4    constructor() {
5      this.successNotification = Selector('#developer-name');
6      this.emailInput = Selector('#form_email')
7      this.submitButton = Selector('button.btn.btn-primary');
8    }
9  }
```

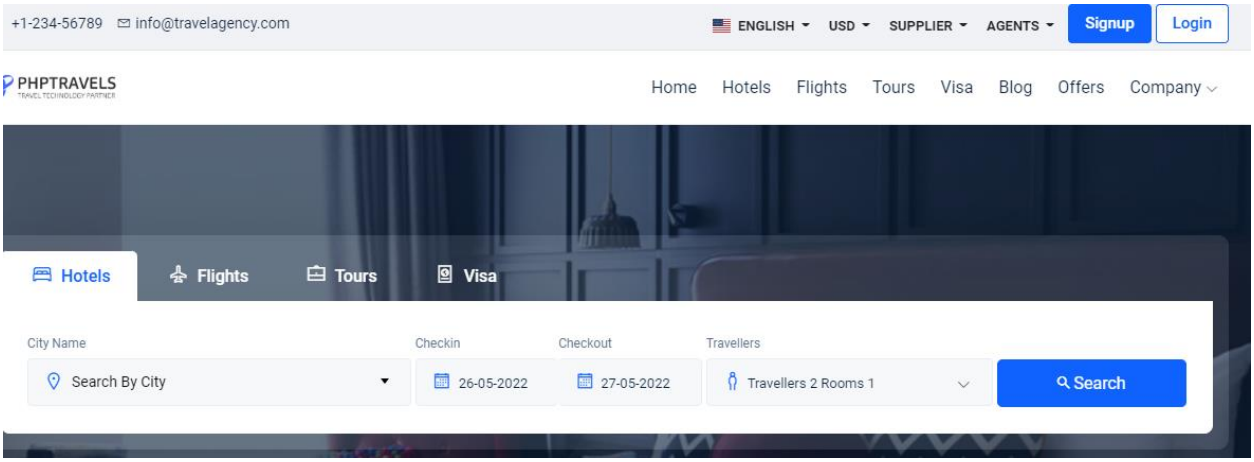
Slika br. 14: Prikaz jednostavnog objekta stranice sa selektorima

2.) Korištenje modela objekta stranice u testu – u ovom koraku se najprije dohvaća model objekta kao na slici br. 15 opisan u prvoj liniji koda. Zatim se nad svakim od selektora modela objekta koristi pripadajuća funkcija npr. za upisivanje teksta, klikanje i slično.

```
1  import Page from './pages/page-object';
2
3  const page = new Page();
4
5  fixture `My first fixture`
6    .page `https://examples.platform-os.com/full-form-example`;
7
8  test('My first test', async t => {
9
10   await t
11     .typeText(page.emailInput, 'user_test@test.com')
12     .click(page.submitButton)
13     .expect(page.successNotification.innerText).eql('This is flash notice (success)');
14 });
```

Slika br.15 Primjer korištenja modela objekta u testnom scenariju

4. Testcafe booking stranica – e2e test



Slika br. 16: Prikaz booking aplikacije za testiranje

https://phptravels.net/ je testna web aplikacija na kojoj testeri i inženjeri kvalitete uče kako implementirati i automatizirati testne scenarije. Sama aplikacija pruža pogodnosti u pogledu pronalaženja i rezervacije smještaja u hotelu ovisno o odabranom datumu i pretragu da li uopće takav smještaj postoji. Osim rezervacija smještaja u hotelu, aplikacija se dotiče i ostalih vrsta rezervacija kao što su pronalaženja obilazaka za gradove, i rezerviranje letova. Još jedna od mogućnosti je prijava za vizu u kojoj se odabire država iz koje se traži viza, te država za koju se ona traži nakon čega se popune osobni podaci te datum. Naravno autentifikacija je isto jedna od važnijih značajki aplikacije te se u kontrolnoj ploči mogu pronaći nedavno pretraženi letovi, rezervacije te stanje na računu koje se koristi prije samih rezervacija.

4.1 Određivanje testnih scenarija

Prije implementacije testova potrebno je odrediti i testne scenarije preko kojih će se odvijati automatizacija u testcafe-u. Jedan od ključnih testni scenarij je provjera da li se aplikacija uspješno učitava i da li se glavne poveznice mogu uspješno otvoriti. Osim toga prije svakog testnog scenarija provjera se prijava korisnika, te odabira jezika aplikacije iz padajućeg izbornika na početnoj stranici pošto nekada aplikacija ima tendenciju promijeniti jezike pri čemu se onda testovi ne bi uspješno mogli izvoditi za glavne

slučajeve. Za glavne mogućnosti aplikacije : rezerviranje hotela, obilaska, leta te prijave za vizu generirat će se testni scenariji, a oni su:

- Rezervacija hotela – ovaj test scenarij imat će 3 testa od kojih je jedan provjera da li se aplikacija uspješno učitava za poveznicu za rezervaciju hotela, te ispravnost poveznice, odnosno da li element web aplikacije sadrži naziv 'HOTELS', a njegov utjecaj na aplikaciju je veliki pošto je to jedna od ključnih njezinih značajki. Sljedeći test bit će odabir grada Zagreba iz padajućeg izbornika tako da se upiše 3 riječi tipa 'Zag', a aplikacija bi trebala prikazati sve gradove koji imaju tu ključnu riječ u sebi te odabrati Zagreb iz tog padajućeg izbornika. Zadnji test ovoga scenarija je provjera da li je datum polaska veći datuma povratka te odrađivane konačne pretraga. Odabrani grad u ovom slučaju bit će Zagreb na način opisan u prethodnom testu ovoga scenarija, te odabir datuma polaska i povratka nakon čega će se odrediti broj putnika za samo jednu osobu te u konačnici stisnuti gumb pretraga. Ovaj test bit će uspješan ako se nakon pretrage prikaže slika s tekstom „no results“.
- Rezervacija letova – slično kao i prethodni testni scenarij, a prvi test ovdje bit će također provjera da li se poveznica za letove uspješno učitava, odnosno da li podnaslov elementa ima u sebi tekst „SEARCH FOR BEST FLIGHT“. Poslije ovoga testa potrebno je prikazati test sa stajališta poslovne logike pri čemu se najprije iz padajućeg izbornika odabire grad polaska na način sličan kao u prethodnom scenariju te se provjerava da li je prikazani rezultat veći od broja 1, odnosno da li takav grad postoji. Testni grad polaska bit će Zagreb. Na identičan način odabire se i destinacija za koju će se koristiti testni grad New York. Također i ovdje se provjerava da li je prikazani rezultat veći od 1 odnosno da li takav grad postoji u padajućem izborniku. Zatim se odabire datum odlaska, nakon čega se provjerava da li element za odabir putnika postoji. Putnici će biti odabrani tako da se doda još jedan putnik na zadani broj od 1 putnika, tako da ih na kraju bude 2, a za to će isto postojati provjera da li je putnik uspješno dodan. Kada se odaberu svi testni podaci, tada se provjerava da li je dinamični modal o pronalasku letova uspješno prikazan. Zadnji test ovoga slučaja bit će provjera da li je datum povratka prikazan nakon klika u slučaju odabira povratnog leta, te da li su letovi poredani

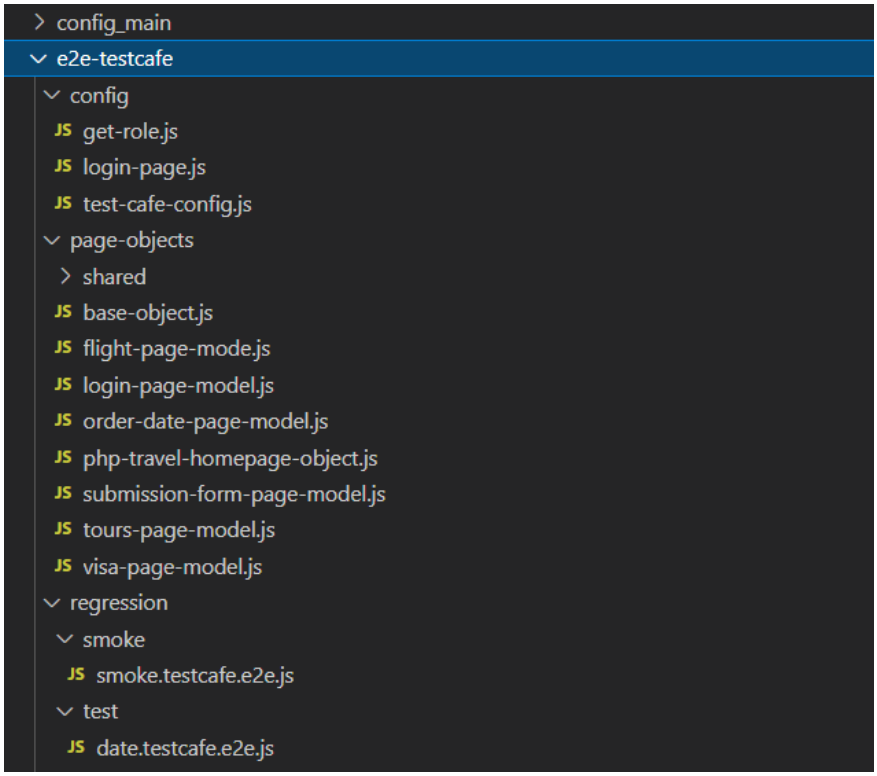
uzlazno prema cijeni nakon njihove pretrage. Na način opisan u prethodnom testu najprije se odabere grad polaska kao Zagreb te New York kao grad destinacija. Razlika od prethodnog slučaja je što se ovdje odabire i datum povratka. Kada se datumi specificiraju, odabire se količina putnika koja će isto biti uvećana za jednog putnika od zadane vrijednosti. Nakon pritiska na gumb za pretragu test mora prikazati uspješno prikazane letove sortirane uzlazno od najmanje prema najvećoj cijeni.

- Rezervacija obilazaka – jedan od testova ovog slučaja bit će kao i u prethodnim testovima provjera da li se poveznica za obilaske uspješno učitava, te da li podnaslov sadrži tekst „FIND BEST TOURS PACKAGE TODAY“. Nakon toga dolazi test za proizvoljni pronalazak obilaska. Destinacija će biti grad Zagreb za koji će se vršiti pretraga, a za isti će se odabrati i proizvoljni datum narudžbe obilaska. Test će zatim provjeriti da li postoji panel s putnicima koji će se pokušati „urediti“ tako da će se klasa putnika specificirati kao odrasla osoba, a količina će se uvećati za jedan u odnosu na zadanu vrijednost od aplikacije. Nakon klika na gumb pretrage aplikacija bi trebala uspješno odraditi navigaciju za traženi resurs. Završni test ovoga scenarija će to i provjeriti pri čemu će postojati funkcije za provjeru da li je link ispravan, odnosno da li on kao „resurs“ sadrži u sebi podatke za prethodno odabrani datum narudžbe, broj odraslih osoba, odnosno broj djece za koju se vrši pretraga.
- Prijava viza – esencijalna komponenta ovog testa kao i u prethodnim slučajevima bit će prikaz da li je aplikacija uspješno učitana te da li postoji podnaslov s tekstom koji sadrži riječi „Submit Your Visa Today“. On je naravno i ovdje bitan test, zato što u slučaju njegovog neuspjeha ni ostali se testovi neće moći uspješno izvršiti. U narednom testu nastojat će se provjeriti država za koju se traži viza i lokacija po zahtjevu za odabrani datum. To će se odvijati tako da se prvo odabere država iz koje se traži viza, gdje će postojati dodatne provjere da li aplikacija uspješno prikazuje države, te država za koju se viza traži. Testni podaci ovdje bit će za „polaznu“ državu Hrvatska, a za destinaciju država Afganistan. Nakon odabira država potrebno je provjeriti da li je datum zahtjeva prikazan unutar aplikacije. Ako je tvrdnja istinita onda se odabire jedan od budućih datuma, nakon čega se stiskom

na gumb „Submit“ završava ovaj test. Naredni test ovoga scenarija bio bi provjera da li se forma za predaju zahtjeva prikazuje te ispravnost linka na kojeg se dođe nakon što se izvrši prethodni test. Najprije se odradi navigacija na željeni link koji je spremljen u varijablu unutar klase u prethodnom testu, nakon kojega se poslije implicitnog čekanja provjerava njegova ispravnost. Nakon provjere linka, provjerava se podnaslov prijavljene forme da li zadovoljava ispravnost, točnije da li podnaslov prijavljene forme sadrži tekst lokacije odredišne države, tekst lokacije države destinacije te se provjerava s regularnim izrazima da li je datum specificiran također u podnaslovu prijavljene forme. U slučaju uspjeha ispunjavaju se dodatni podaci kao što su ime osobe koja zahtjeva vizu, prezime, email adresa, broj telefona, datum i dodatne bilješke koje korisnik odabere. Nakon uspješno unesenih podataka aplikacija bi se trebala navigirati na poveznicu https://phptravels.net/success/visa. Na sličan način potrebno je odraditi još jedan test koji u slučaju nepotpunih podataka ne smije prikazati poveznicu za uspješno zatraženu vizu.

4.2 Konstrukcija testova u visual studio codu

Za lakše snalaženje u automatizaciji testnih scenarija potrebno je napraviti dobru strukturu testa. To je najbolje postići tako da se podaci koji su potrebni za konfiguraciju i tajni podaci aplikacije odvoje u posebnu datoteku namijenjenu baš za tu konfiguraciju, a testna logika da se raspodjeli u ostale datoteke. Jedan od primjera dobre konstrukcije testova koji su dio ovoga rada prikazan je ispod na slici br. 17.



Slika br. 17: Prikaz konstrukcije testa

Na prethodnoj slici prikazani su jedni od kvalitetnih načina izgradnje testova. Na njoj vidimo kako postoje sljedeće datoteke : config_main, e2e-testcafe, page-objects, regression. Datoteka config_main najjednostavnija je datoteka i u njoj se nalaze bitni podaci kao što su korisničke tajne potrebne za prijavu na aplikaciju, te sama poveznica na resurs aplikacije. Tu datoteku proširuje datoteka test-cafe-config koja ima dodatna svojstva u praksi koja određuju ponašanje aplikacije. Primjer takve datoteke prikazan je na slici br. 18. Tamo su prikazane neke od konfiguracijskih svojstava kao što su *timeout*, navigacija, *loginPage*. Ovakve konfiguracijske datoteke naročito su dobre ako se radi brza autentifikacija korisnika pa se postavkom vrijednosti loginPage na *true*, odnosno *false* lagano može odabrati vrsta prijave koja može biti zapamćena autentifikacija putem tokena ili normalna prijava gdje se specificira korisničko ime i lozinka.

```
import ConfigData from "../../config_main/testcafe-config"

export const TestCafeConfig = {
    ...ConfigData,
    loginPage : true,
    defaultTimeout : 15000,
    loggerSettings : {
        creation: true,
    edit: true,
    navigation: true,
    viewCache: true,
    step: true,
    retryWhen: true,
    requestSuccess: true,
    requestError: true,
```

Slika br. 18: Konfiguracijska datoteka aplikacije

Od prethodno spomenutih svakako valja objasniti loginPage funkciju. U slučaju da se ono specificira na *true* provodi se normalna prijava u aplikaciju. Pošto većina aplikacija ima pristup temeljen na ulozi tada je korisno koristiti funkciju *Role()* koja dolazi unutar testcafe-a. Nju je potrebno koristiti kako bi se provjerilo je li određena funkcionalnost dostupna korisnicima kojima je dopuštena. Ako se mora prijaviti što više različitih korisnika tijekom testova, tada bi takav postupak bio ponavljajući za više tipova korisnika. Tada će testni paket biti pun dupliciranog koda, a testovi će se značajno usporiti dok se više puta prijavljujemo u sustav i izlazimo iz njega[33]. Iako u našoj aplikaciji nemamo takav slučaj gdje postoji više različitih korisnika, svejedno će biti prikazano korištenje uloga pošto je to moderna praksa koja se provodi unutar većine kompanija. Funkcija *loginPage()* u našem slučaju bit će jednostavna a u njoj se najprije vrši maksimiziranje prozora web preglednika te se pronalazi gumb za prijavu u aplikaciju (ovdje se pretpostavlja da je korisnik prethodno ručno kreiran). Ulaskom u dio aplikacije za prijavu, korisnik specificira korisničko ime i lozinku koje dohvaća iz prethodno spomenute konfiguracijske datoteke te se uspješno ostvaruje prijava. Ovdje se i koristi svojstvo „*preserveUrl*“ koje nam određuje da se korisnik ne odjavljuje nakon uspješne prijave. Ovaj spomenuti dio prijave prikazan je u kodu na sljedećoj slici.

[33] https://javascript.plainenglish.io/power-of-user-roles-in-testcafe-b88458baae17

```
export const loginPage = Role(
    TestCafeConfig.appUrl,
    async t => {
        console.log("Navigiras na", TestCafeConfig.appUrl);
        await t
            .maximizeWindow()
            .click(loginPageModel.login2)
            .typeText(loginPageModel.userNameInput, ConfigData.username)
            .typeText(loginPageModel.userNamePassword, ConfigData.password)
            .wait(1000)
            .click(loginPageModel.loginSignIn);
            console.log("Autentificiras se na sajt");
    },
        {preserveUrl : true}
);
```

Slika br. 19: Prikaz korištenja uloga za brzu prijavu u aplikaciju

4.2.1 Određivanje objekata modela stranice

Nakon što se spozna da je određivanje objekata modela stranice bitna stavka u organizaciji koda i odvajanja selektora i funkcija od testova i poslovne logike, vrijeme je za prikaz implementacije istih. Neki osnovni modeli koji su ujedno i najjednostavniji su modeli „login-page-model“, „php-travel-homepage-object“, te „base-object“, a o njima neće biti riječ u ovome radu. Objekti modela koji će biti prikazani te koji imaju najveći značaj za provedbu testova su : „order-date-page-model“, „flight-page-model“, „tours-page-model“, „visa-page-model“.

- order-date-page-model – koristi se za odabir metoda i selektora za testni scenarij rezerviranje hotela. U njemu se sadrže selektori za pronalaženje elementa za pretragu hotela, te razne metode od kojih u najvažnije metoda za dohvaćanje naslova poveznice s hotelima, dohvaćanje svih obaveznih polja za njihovu manipulaciju, dohvaćanje vrijednosti odabranog datuma, odabira unosa za grad, prikaz rezultata grada iz padajućeg izbornika i slično. Jedan skraćeni prikaz takvog modela prikazan je na sljedećoj slici.

```
export default class OrderDatePageModel {

    constructor(selector, regularDate = undefined){
        this.homepage = `${ConfigData.appUrl}/hotels`
        this.selector = selector;
        this.location = ClientFunction(() => document.location.href);
        this.dataComparement = DateComparement.get();
        this.dateOrderForm = this.selector.find('.main_search');

    }

    static get(parent = Selector('body')){
        const selector = parent.find('#hotels-search')
        return new OrderDatePageModel(selector)
    }

    getHotelTitle(){
        return this.selector.parent('div').prevSibling('h2').with({timeout : 7000});
    }

    async selectDateInput(index){
        return this.selector.find()
    }
```

Slika br. 20: Model objekta za rezerviranje hotela

- flight-page-model – model koji se koristi za testni scenarij rezervacija letova. Osim selektora za pronalazak letova, u njemu kao i u ostalim modelima objekta postoji jedna funkcija koja pronalazi sva polja koja se mogu manipulirati. Selektori koji se manipuliraju ovim objektom su selektori za lokaciju polaska, destinaciju, datum odlaska, datum povratka te putnici. Prikaz jedne zajedničke funkcije koju koriste i ostali modeli objekta za manipulaciju datumima prikazana je na slici br. 21.

```
async getAvailableCheckInOutDates(nth) {
    const datePicker = Selector('.datepicker')
        //.withAttribute('style', "display: block; top: 343.891px; left: 768.656px;")
        .withAttribute('style', /display: block;/)
        .find('tbody')
        .child('tr')
        .nth(nth)
        .child('td')
        .with({timeout : 8000})
    const day = await datePicker.nth(5).innerText;
    const monthYear = await this.getSelectedDateValue(datePicker).innerText;
    const splittedMonthYear = monthYear.split(" ");
    const connectedDate = this.dataComparement.connectDate(day, splittedMonthYear[0], splittedMonthYear[1]);
    return Promise.all([datePicker.nth(5), connectedDate])
        .then(function (data){
        return {
            'datePickr' : data[0],
            'day' : data[1]
        }
    })
}
```

Slika br. 21: Prikaz funkcije za odabir datuma iz kalendara

Prethodna funkcija bitna je zato što osim što ona pronalazi element za odabrati datum iz kalendara ona vraća i odabrani datum u tekstualnom obliku kako bi se ostale provjere mogle vršiti kao što su usporedba datuma da li su veći jedan od drugoga te da li su datumi dostupni unutar poveznice nakon što se odabere let.

- tours-page-model – ovaj model svoju primjenu pronalazi za scenarij rezervacija obilazaka. U njemu postoje selektori za poveznicu koji vodi na lokaciju s rezervacijama, te postoji uz ostale selektore i varijabla „*travellers*“ koja pokazuje za koliko putnika djece i odraslih osoba se traži obilazak. Važna funkcija koja se nalazi unutar ovog modela je ona za određivanje putnika da li se radi o djetetu ili odrasloj osobi te smanjivanje, odnosno povećavanje broja putnika za rezervaciju obilazaka. Prema pravilima aplikacije, zadano je da se na početku nalazi broj odraslih osoba jednak 1 te broj djece 0. Funkcija za uređivanje putnika „*editTravellers()* poziva funkciju „*getTypeOfTraveller*“ kako bi se za traženu osobu u padajućem izborniku pronašao odgovarajući element za njihovo povećanje ili smanjenje količine te spremanje rezultata u varijablu „*travellers*“. Prikaz takve funkcije nalazi se na slici ispod.

```
async editTravellers(person, type){
    if(person == 'adult'){
        if(type == 'qtyInc'){
            this.travellers["adults"] = this.travellers["adults"]+1;
        } else{this.travellers["adults"] = this.travellers["adults"]-1}
    }else{
        if(type= 'qtyInc'){
            this.travellers["childs"] = this.travellers["childs"]+1;
        } else{this.travellers["childs"] = this.travellers["childs"]-1}
    }
    return this.getTypeOfTraveller(person, type)
}

getTypeOfTraveller(person, adjust){
    return (Selector('.dropdown-menu')
            .withAttribute('style', /display: block;/)
            .child('.dropdown-item')
            .nth(person == 'adult' ? 0 : 1)
            .find(`.${adjust}`));
}
```

Slika br. 22: Funkcija za uređivanje količine putnika

- visa-page-model – model je koji se koristi za manipuliranje podacima vezano za testni scenarija prijava viza. Osim uobičajenih selektora za pronalazak svih obaveznih polja za unošenje podataka za prijavu viza, ovaj model omogućuje i pojedinačan odabir polja za manipulaciju kao što su ime, prezime, e-mail adresa, broj telefona, datum, te ostale bilješke koje korisnik smatra da su potrebne pri podnošenju zahtjeva za vizom. Za ovaj testni scenarij koristi se posebno dizajnirana funkcija koja pronalazi države iz koje se zahtjeva viza, te država za koju se ona zahtjeva putem direktnog odabira u padajućem izbornika umjesto prethodnog upisivanja grada/države u polje pa odabira rezultata iz padajućeg izbornika.

```
async selectCountryByText(source, country){
    console.log(`country 1 ti je ${country}`);
    console.log(Object.values((await this.getCountryAndAbbr(source, country)))[1]);
    this.countries[source] = Object.values((await this.getCountryAndAbbr(source, country)))[1]
    return (Object.values((await this.getCountryAndAbbr(source, country)))[0])
            .withExactText(country)
            .with({timeout : 7000})
}

async getCountryAndAbbr(source, country){
    const searchResults = await this.getAllSearchResults();
    const countryAbb = Selector('select')
        .withAttribute('id', source)
        .child('option')
        .withExactText(country)
        .getAttribute('value');

    console.log(`country abb ti je ${await countryAbb}`);
    return Promise.all([searchResults, await countryAbb])
        .then(function(data){
            return {
                'countrySelect' : data[0],
                'countryAbbr' : data[1]
            }
        })
```

Slika br. 23: Funkcija za odabir države iz izbornika

Na prethodnoj slici opisana je jedna od glavnih metoda za pronalazak države i skraćenice za specificiranu državu. Ona nam je potrebna ako poslije želimo provjeriti da li se skraćenice i datumi zahtjeva za vizu pronalaze u poveznici nakon što se zahtjev preda.

4.2.2 Pokretanje i prikaz testnih scenarija

Za pokretanje testnih scenarija bilo je potrebno kreirati dodatnu datoteku koja sadrži putanju do testova od interesa. Ta je datoteka u praksi nazvana .testcaferc.json i u njoj se mogu definirati razne pogodnosti testova kao što su vrijeme čekanja pri korisničkom zahtjevu za neki od resursa aplikacije koji su u ovom slučaju postavljeni na 20 sekundi, te da li greške koje se dogode u JavaScriptu preskaču, a u ovom slučaju su postavljeni na istinu, odnosno ne želimo da moguće pogreške u kodu utječu na rezultate izvođenja testova. Od ostalih stvari specificira se vrsta web preglednika koja je u našem slučaju postavljena na google chrome, te putanja do testnih scenarija. Budući da je potrebno prikazatu sve testne scenarije koji su definirani u poglavlju 4.1 važno je podesiti da ta putanja „pogađa“ njih sve. Sve prethodno opisano može se vidjeti na sljedećoj slici u konfiguracijskog datoteci .testcaferc.json.

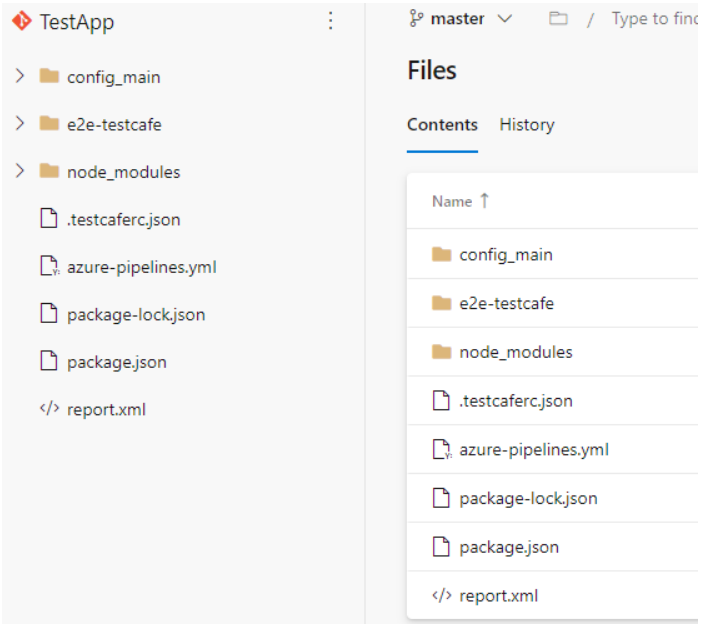
```
{
    "browser" : "chrome",
    "src" : ["e2e-testcafe/regression/**/*.e2e.js"],
    "skipJsErrors": true,
    "pageRequestTimeout": 20000
}
```

Slika br. 24: Prikaz konfiguracijske datoteke u testcafe-u

Testovi se odvijaju tako da se najprije odrađuje takozvani „test dima“ u kojemu se provjerava da li se aplikacija uspješno učitava, te da li je link ispravan, a zatim da li se glavni dijelovi aplikacije mogu otvoriti. To je inicijalni test koji ima jako veliku važnost i njegov neuspjeh odražava najveći gubitak poslovne vrijednosti za neku kompaniju. Zatim se redom vrše testovi počevši od rezervacije hotela, rezervacije letova, te pronalaska obilazaka te završni tekst provjere podnošenja zahtjeva za vizom.

5. Kontinuirana integracija e2e testova

Osiguravanje da testovi rade u lokalnom okruženju korisno je za provjeru ispravnosti sustava kojega testiramo, no ono sama po sebi ne pronalazi veliku primjenu ako proces testiranja nije kontinuiran te se automatski određuje. Tu dolazi jedna od glavnih prednosti kontinuirane integracije, a to je stalna, brza, specificirana provjera koda, odnosno vršenje testnih scenarija nad aplikacijom na jednostavan i pouzdan način. U primjeru integracije prethodno odabranih testnih scenarija koristit ćemo Azure DevOps, te postavi cjevovod na pouzdan način. Nakon registracije na azure portal te uspješnog postavljanja licenci sve je spremno za pokretanje cjevovoda na način kako mi želimo. Najprije se prolazi uobičajen postupak, a to je kreiranje organizacije, te repozitorija u koji ćemo učitati projekt prethodno kreiran u visual studio codu u JavaScriptu i node js-u. Primjer jednog takvog učitanog repozitorija može se pronaći na slici br. 25.



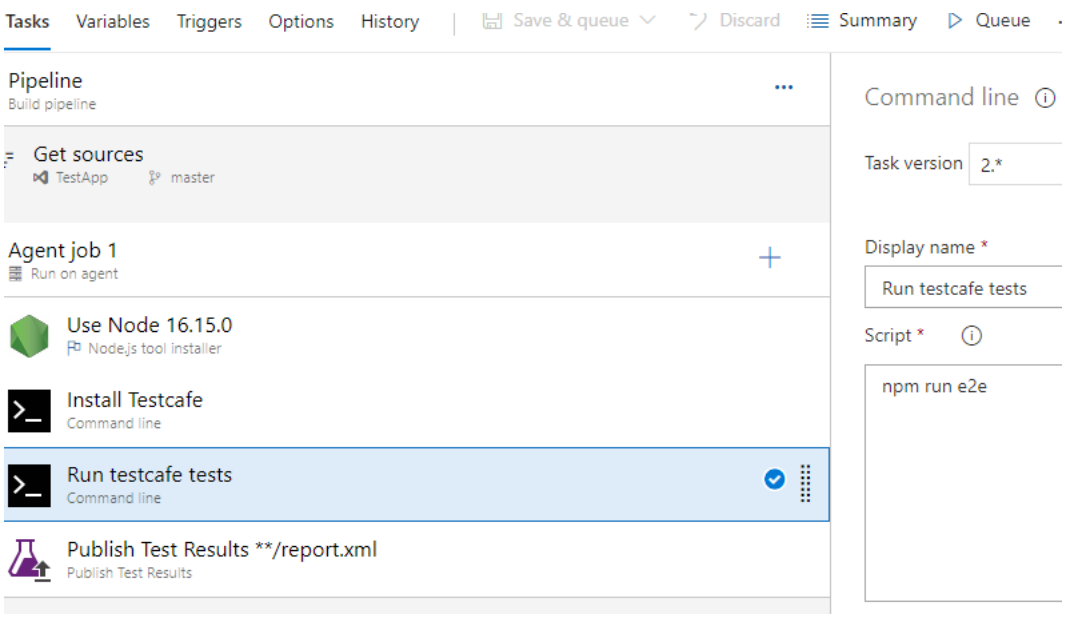
Slika br. 25: Primjer učitanog projekta na Azure repozitoriju

Poslije učitavanja projekta sve će biti spremno za uspješno postavljanje testnih scenarija za kontinuiranu integraciju korištenjem azure cjevovoda (eng. Pipeline). Postavljanje cjevovoda je zadnja prepreka ka integraciji naših testnih scenarija. Cjevovodi se

podešavaju kada se odabere izbornik „Pipelines“ unutar Azure DevOps-a. Nakon klikanja izbornika upitkuje nas se od kuda dolazi naš kod, a tu se nalaze brojne opcije kao što su Azure Repos Git, Bitbucket Cloud, GitHub, GitHub Enterprise Server, Subversion, te ostali git repozitoriji. Budući da je kod učitan u Azure repozitorij odabire se tada ta prva opcija, te iz nje projekt koji je prethodno nazvan „TestApp“. Zatim se prikazuje trenutno „stanje“ cjevovoda opisano u yaml formatu koji je poznat po proširenju .yml. YAML je format serijalizacije podataka čija glavna prednost je čitljivost i mogućnost pisanja koda za konfiguraciju. Ako postoji konfiguracijsku datoteku koja može ljudima olakšati čitanje, tada je dobra opcija koristiti YAML. YAML je format koji nije potpuna zamjena za JSON jer JSON i XML također imaju svoje mjesto, no ipak korisno je učiti YAML (ulazak u detalje YAML-a neće biti dio ovoga rada). Još jedna prednost YAML-a je njegova podrška za različite tipove podataka kao što su slučajevi, nizovi, rječnici, liste i skalari, te on ima dobru podršku za najpopularnije jezike kao što su JavaScript, Python, Ruby, Java itd. YAML podržava samo razmake, i osjetljiv je na velika i mala slova[34]. U slučaju da se ne želi koristiti YAML za konfiguraciju cjevovoda, tada se može još uvijek odabrati klasični način za postavljanje u editoru. U ovom slučaju odlučio sam se za takvo postavljanje cjevovoda kroz klasični način bez poznavanja YAML jezika. YAML će ionako biti generiran poslije te ga se može pregledati u slučaju potrebe, te podijeliti s ostalim korisnicima za koje bi takvo nešto možda bilo korisno. Nakon odabira klasičnog editora najbolje je odabrati prazni posao (eng. Empty job) gdje se naknadno dodaju zadaci redom na prethodno promišljen način kako bi se kod što bolje „obradio“. Način na koji cjevovod radi je taj da njegove zadatke vrši jedan od agenata koje pronalazimo unutar Azure DevOps-a. Agenti su ništa drugo nego operacijski sustavi koji vrše zadatke unutar cjevovoda, a oni mogu biti definirani iz Azure „bazena“ (eng. Azure Pool) ili mogu biti vlastiti agenti koji se nalaze na računalu domaćina. Jedna od bitnih razlika je ta što za agente koji se nalaze u Azure bazenu nije moguće koristiti besplatno, osim ako se ne zatraži probni period mada i tad postoje neka od ograničenja u pogledu dostupnih agenata za korištenje te broja sati na mjesečnoj bazi za njihovu uporabu. Na zatraženi probni period koji ne zahtijeva posebnu licencu mogu se koristiti odabrani agenti kao windows 2022, windows latest, windows 2019, macOS-12, ubuntu 18.04 u rasponu od maksimalno 30 sati u mjesec dana. Druga

[34] https://geekflare.com/yaml-introduction/

opcija u korištenju je malo sporija, a to je korištenje agenta instaliranog na računalu domaćina. Jedna od prednosti ovoga načina je ta što u odabiru ovakve opcije ne postoji vremensko ograničenje za koje se može upotrebljavati agent za izvršavanje specificiranih zadataka. Odlučit ćemo se za takav pristup, a njegova konfiguracija je intuitivna, treba se samo preuzeti željeni agent za Windows ili Ubuntu te podesiti na računalo domaćina te pokrenuti kao servis. Nakon postavljanja agenata i zadataka nužnih za izvršavanje testova dobije se nešto slično kao što je prikazano na sljedećoj slici.



Slika br. 26: Prikaz zadataka nužnih za pokretanje e2e testova u cjevovodu

Na prethodnoj slici vide se koraci potrebni za uspješno izvršavanje testova u cjevovodu, a njihova objašnjenja su sljedeća:

- Prvi zadatak je korištenje node verzije 16.15.0 koja je u skladu one verzije koje se nalazi na lokalnom računalu domaćina. To je zadatak koji je bitan za izvršavanje JavaScript koda izvan web preglednika, te je krucijalan za postavljanje cjevovoda.
- Drugi zadatak je instalacija testcafe-a kroz korištenje komandne linije, a on je nazvan u ovom primjeru „Install Testcafe“. Skripta koja se specificira kroz korištenje komandne linije u ovom slučaju je „*npm install*“, a u njemu se mora pripaziti gdje se nalazi package.json datoteka. Ta naredba instalirat će sve pakete

koje se nalaze u datoteci package.json. Pošto se u ovom primjeru package.json nalazi unutar projekta može se u advanced sekciji to polje ostaviti prazno.

- Treći zadatak je ujedno i najvažniji, a on je pokretanje testcafe-a također kroz komandnu liniju. Taj zadatak nazvan je u ovom slučaju „Run testcafe tests“, a skripta koja se pokreće glasi „*npm run e2e*“ pošto je ista specificirana u package.json datoteci. Ovdje također kao i u prethodnom zadatku potrebno je paziti na putanju gdje se nalazi package.json datoteka, a ona je u ovom slučaju u zadanom direktoriju pa se polje u advanced sekciji i u ovom slučaju može ostaviti praznim.
- Publish Test Results **/report.xml posljednji je zadatak ovoga cjevovoda i on je zadužen za generiranje izvještaja koji će završiti u sekciji „Test Plans“ gdje je potrebno pronaći poglavlje „Runs“ unutar Azure DevOps-a.

Nakon odabranog agenta za cjevovod te zadataka koji će se pomoću njega pokretati, moguće je dobiti uvid u samu YAML konfiguracijsku datoteku za postavljanje prethodno spomenutnog cjevovoda. Takva datoteka bit će generirana unutar samoga projekta kojega smo prethodno prenijeli u repozitorij na Azure DevOps-u. Primjer jedne takve datoteke pronalazi se na slici ispod:

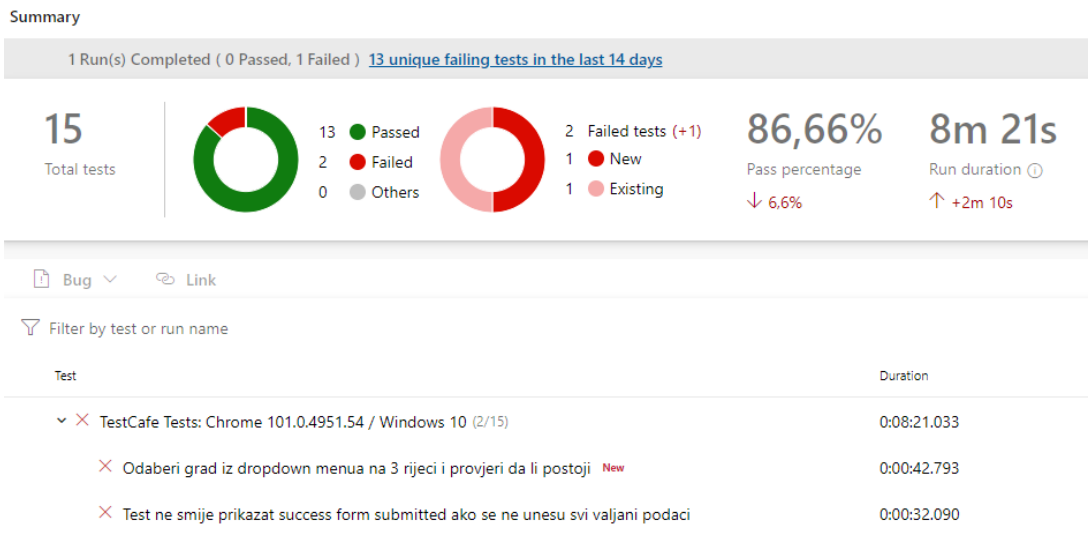
```
pool:
  name: Self-Hosted Windows Agent
steps:
- task: NodeTool@0
  displayName: 'Use Node 16.15.0'
  inputs:
    versionSpec: 16.15.0

- script: 'npm install'
  displayName: 'Install Testcafe'

- script: |
   npm run e2e

  displayName: 'Run testcafe tests'

- task: PublishTestResults@2
  displayName: 'Publish Test Results **/report.xml'
  inputs:
    testResultsFiles: '**/report.xml'
  condition: succeededOrFailed()
```

Slika br. 27: Prikaz YAML datoteke za konfiguraciju cjevovoda

- Zadnji korak tj. „PublishTestResults“ bitan je da se najprije konfigurira lokalno u package.json datoteci unutar projekta dodajući zastavicu *--reporter spec,xunit:report.xml* kod *e2e* naredbe.

5.1 Prikaz i interpretacija rezultata testova

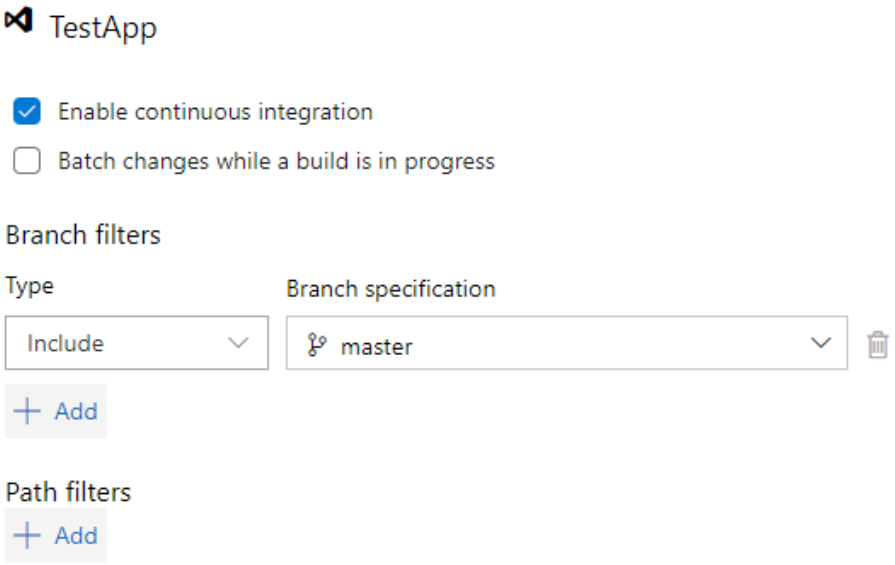
Nakon što se testovi uspješno izvrše u našem cjevovodu važno je pogledati rezultate testova. Kao što je objašnjeno u prethodnom poglavlju testove je moguće pronaći unutar sekcije „Test Plans“, te se tamo pod sekcijom „Runs“ mogu pronaći i tumačiti testovi. No to nije jedino mjesto gdje se prethodno izvršeni testovi mogu pronaći. Kada se cjevovod pokrene bit će prikazano određeno mjesto za testove unutar njega pod sekcijom „Tests“, a u njoj se mogu filtrirati testovi koji su se uspješno izvršili, odnosno oni koji su „pali“, te na koje treba posebno obratiti pozornost. Primjer jednog od takvih testnih izvještaja prikazan je na sljedećoj slici.



Slika br. 28: Prikaz rezultata testova

Na slici br.28 vidimo prikaze testova koji se mogu filtrirati prema kategorijama onih koji su prošli, koji su pali, prekinuti te ostale vrste filtera koji su važni ovisno o našim preferencijama. Na primjeru sa slike vidimo da od trenutno napisanih 15 testova koji

zadovoljavaju testne scenarije ranije definirane, samo su 2 pala, a ostalih 13 je uspješno izvršeno, odnosno aplikacija je zadovoljila testne scenarije. Osim toga primjećuje se i jedan novi dodani test prikazan s oznakom “New”. Prolazak testnih scenarija iznosila je 86.66%, odnosno 6.6% manje od onoga testa koji je ranije pokrenut. Trajanje testa iznosilo je 8 minuta i 21. Sekundu što je uvećanje od 2. minute u odnosnu na prethodno izvršeni test. Od testova koji su „pali“ pronalaze se test koji ne smije prikazati uspješno ispunjen predan zahtjev za vizu s obzirom ako nisu svi podaci pravilni uneseni. To je pogreška aplikacije za rezervaciju koja bi morala pripaziti da se takve zbunjujuće stavke u budućnosti isprave, te koje mogu negativno utjecati na korisničko zadovoljstvo samom aplikacijom. Također možemo vidjeti da i novi dodani test nije prošao provjeru, odnosno za upisani tekst u padajućem izborniku s riječi „Zag“, Zagreb nije bilo moguće pronaći zbog čega bi određivanje nepostojeće polazišne lokacije leta kao Zagreb moglo uzrokovati negativne komentare i korisnički doživljaj same aplikacije. Većina testova je uspješno prošla, a samo neki su pali što je ipak rezultat dobrog vođenja poslovnom logikom, ali ako aplikacija „misli“ biti konkurentna na tržištu tada mora ispraviti prethodno spomenute greške. Za sada je sve implementirano u cjevovod na zadovoljavajući način, ali treba postaviti još jednu bitnu stvar kako bi se kontinuirana integracija uistinu zbivala. To je jedna od prednosti Azure DevOps-a gdje se ona može lako podesiti tako da se prethodno napravljeni cjevovod odabere te se pronađe sekcija „Triggers“ kod koje se mogu podesiti stvari kao što su raspored koji označava koliko puta na dnevnoj bazi da se on pokrene ili u vremenskom intervalu kojega osoba zadužena za implementaciju cjevovoda odabere. S našeg stajališta bitno je uključiti opciju „Enable continuous integration“ što je prikazano na narednoj slici.



Slika br.29: Postavljanje kontinuirane integracije

Opcija za postavljanje kontinuirane integracije od ključnog je značenja za odvijanje testnih scenarija. Ona nam označava da će svaka promjena koda koja se napravi na kodu, potvrdi, te unese u udaljeni repozitorij na Azure DevOps-u proći cijeli prethodno definirani cjevovod kako bi se na toj promjeni koda vršili testovi. Svaki pokušaj prijenosa koda na udaljeni repozitorij npr. koristeći „git bash“ ili sličan programski proizvod za verzioniranje koda temeljito će se testirati što će dati dodatni sloj sigurnosti u provjeri stabilnosti koda te pouzdanja cijelom timu u razvojnom procesu programskog proizvoda.

6. Unakrsno testiranje i testovi vizualne regresije

Unakrsno testiranje ili „cross-testing“ provjerava kompatibilnost web aplikacije ili web stranice na različitim preglednicima, operativnim sustavima i uređajima. Testiranjem unakrsnih preglednika osigurava se da web aplikacija ili web stranica radi na ujednačen i pouzdan način na preglednicima i uređajima koje klijenti mogu koristiti. Tipovi preglednika za koje se koristi su: Chrome, Firefox, Safari, Internet Explorer, Edge, UC preglednik, Opera itd. Ova vrsta testiranja pokriva i razne operativne sustave kao što su Windows, Android, iOS, macOS itd, te mnoge uređaje kao mobilne uređaje, laptope, desktop, tablet, smart TV, itd. Korisnik može koristiti bilo koji od uređaja s bilo kojom kombinacijom preglednika ili OS-a. Stoga je potrebno provjeriti kompatibilnost, performanse i korisničko sučelje aplikacije na svim ovim kombinacijama. Kada se ono temeljito testira u svim kombinacijama, tada testiranje na različitim preglednicima pomaže u pružanju dosljednog i visokokvalitetnog korisničkog iskustva na svim uređajima i preglednicima koje koriste klijenti [35].

6.1 Svojstva unakrsnog testiranja preglednika[36]

Unakrsno testiranje ima razne značajke u kojima se mogu pronaći razni benefiti, a jedni od najznačajnijih su svakako :

- Testiranje na stvarnim testnim okruženjima : moguće je sprovesti vlastite testove kompatibilnosti među preglednicima bez napora na stvarnim preglednicima kako se ne bi propustila niti jedna greška.
- Testirati se može kada god se poželi i na lokalnim strojevima: Testiranje je lakše izvesti na vlastitom okruženju te provjeriti testne slučajeve na lokalnim strojevima iza vatrozida bez ikakvih problema.
- Različite metode izvješćivanja o testovima: mogu se odabrati metode izvješćivanja po vlastitom izboru između slika, videa itd. radi boljeg razumijevanja testa.

[35] https://testsigma.com/cross-browser-testing
[36] https://testsigma.com/cross-browser-testing

- Šira pokrivenost koda testom: razne platforme za automatizaciju nude izvršavanje testova bez koda, umjetnu inteligenciju, procesuiranje prirodnog jezika. Uklanjanjem problema u kodiranju pomaže se ostvariti veća pokrivenost testom.
- Jednostavnost otklanjanja pogrešaka: kako bi testni slučajevi bili vrhunski i savršeni, postoje razne značajke lakog otklanjanja pogrešaka.
- Kontinuirana integracija i isporuka klijentskih komponenti: moguće je ostvariti jednostavnu integraciju s cjevovodom kontinuirane integracije za brže i lakše upravljanje kontinuiranom isporukom.
- Testiranje se može izvoditi bilo kada: razne platforme unakrsnog testiranja preglednika temeljenih na oblaku pomažu u pokretanju testnih paketa bilo kada i testiranje se može završiti brže, učinkovitije i uz manje napora.

6.2 Lambda test kao platforma za unakrsno testiranje web preglednika[37]

LambdaTest je platforma za testiranje više preglednika koje se temelje na oblaku te omogućuju provođenje unakrsnog testiranja web-preglednika za web aplikacije na više od 2000 preglednika, operativnih sustava i uređaja. Pomoću LambdaTest-a može se izvesti i ručno i automatsko testiranje na više preglednika. Na primjer mogu se testirati web aplikacije na najpoznatijim web preglednicima, kao što su Chrome, Safari, Firefox, Edge, Internet Explorer, Opera i Yandex. To je skalabilna platforma za testiranje koja pomaže razvojnom timu dovodeći potrebe za procjenom softvera u infrastrukturu oblaka. Provodeći testiranje kompatibilnosti više preglednika u stvarnom vremenu pomoću LambdaTest-a, osigurava se da je web aplikacija ili web stranica kompatibilna s gotovo svim preglednicima i uređajima dostupnim na tržištu. Ona omogućuje organizacijama da testiraju svoju web aplikaciju na odziv i naprave pune snimke zaslona tijekom testiranja. Lambda test je platforma koja će biti korištena za provođenje unakrsnog testiranja web preglednika, te će ona pomoći pri daljnjoj kontinuiranoj integraciji softvera.

[37] https://thenewstack.io/how-lambdatest-automates-cross-browser-testing-from-the-cloud/

6.2.1 Glavna obilježja platforme Lambda test

Pošto je LambdaTest odabrana platforma za izvođenje unakrsnog testiranja web preglednika, bilo bi dobro posvetiti i koju riječ njegovim glavnim obilježjima. Valja istaknuti kako će se za vizualnu regresiju koristiti jedna posebna biblioteka unutar testcafe-a, tako da se neće primijeniti vizualna regresija kao dio LambdaTest platforme u ovome radu. Na sljedećoj slici nalazi se logo LambdaTest platforme.

Slika br.30: Prikaz loga LambdaTest platforme

Jedne od glavnih obilježja LambdaTest platforme su[38]:

- Testiranje kompatibilnosti internetskih preglednika – korištenjem platforme LambdaTest, može se provesti uživo testiranje kompatibilnosti više preglednika s različitim verzijama, operativnih sustava i rezolucije. Osim testiranja uživo, platforma također nudi snimanje zaslona i pružanje slika stanja u realnom vremenu (eng. Snapshot)
- Testiranje u najnovijim preglednicima za stolna računala - postoji ogroman popis preglednika na Windows i Mac operativne sustave. LambdaTest omogućuje pristup za testiranje web aplikacije čak i na najnovijim desktop preglednicima, kao što su Google Chrome, Internet Explorer, Edge, Mozilla Firefox, Safari, Opera i mnogi drugi. Također podržava najnovije verzije operativnih sustava kao što su

[38] https://thenewstack.io/how-lambdatest-automates-cross-browser-testing-from-the-cloud/

Windows, Mac, Android, iOS, itd. Testirati se može odziv na svim veličinama, tako da se testiraju web aplikacije na različitim veličinama zaslona samo jednim klikom.

- Kontinuirana suradnja tijekom testiranja - često se s testerima događa da nisu u mogućnosti surađivati s članovima svog tima tijekom provođenja unakrsnog testiranja preglednika. No, budući da je alat za testiranje temeljen na oblaku, LambdaTest omogućuje bilježenje problema tijekom testiranja web aplikacije i dijeljenje problema putem slack-a, e-pošte i slično. LambdaTest trenutno pruža integracije s gotovo 13 alata za upravljanje greškama.
- Vizualno testiranje - programeri se često susreću s problemima dok rješavaju probleme u vezi s veličinom ikona, rasporedom, razmacima između teksta, položajem elementa itd. Koristeći LambdaTest, jednostavno se može analizirati vizualnu „privlačnost“ web aplikacije. Vizualno testiranje uključuje provjeru odziva različitih veličina zaslona bez kompromisa s kvalitetom izgleda web stranice. Također, omogućuje automatsko testiranje vizualne regresije kako bi se osigurao da su izgledi web stranice savršeni i da su identificirane sve greške koje mogu utjecati na izvedbu web stranice.

6.2.2 Implementacija unakrsnog testiranja web preglednika te integracija u Azure DevOps

Za implementaciju unakrsnog testiranja web preglednika treba se najprije instalirati paket za LambdaTest za njegovu integraciju sa testcafe-om. To se postiže naredbom *npm install testcafe-browser-provider-lambdatest* nakon koje će se u datoteku package-lock.json dodati potrebni podaci za pokretanje testova. Naredbom *testcafe -b lambdatest* mogu se prikazati svi dostupni web preglednici na kojima se mogu vrtjeti testovi. Prilikom postavljanja lambda testova potrebno je dohvatiti dvije glavne stvari, a to su LT_ACCESS_KEY i LT_USERNAME koje se mogu pronaći unutar platforme LambdaTest nakon uspješne registracije. Ovisno o načinu pretplate, prikazani su dostupni preglednici, pa će tako besplatna registracija imati ograničeno vrijeme korištenja LambdaTest platforme, osim ako se ne „nabavi“ nova pretplata. Da bi lambda testovi bili valjani treba pronaći te bitne podatke kao što su LT_ACCESS_KEY i LT_USERNAME te

ih postaviti u varijable okruženja (eng. Environment variables) kako bi se mogla uspješno ostvariti veza s registriranim LambdaTest profilom, te na kraju i prikazati samo izvođenje testova.

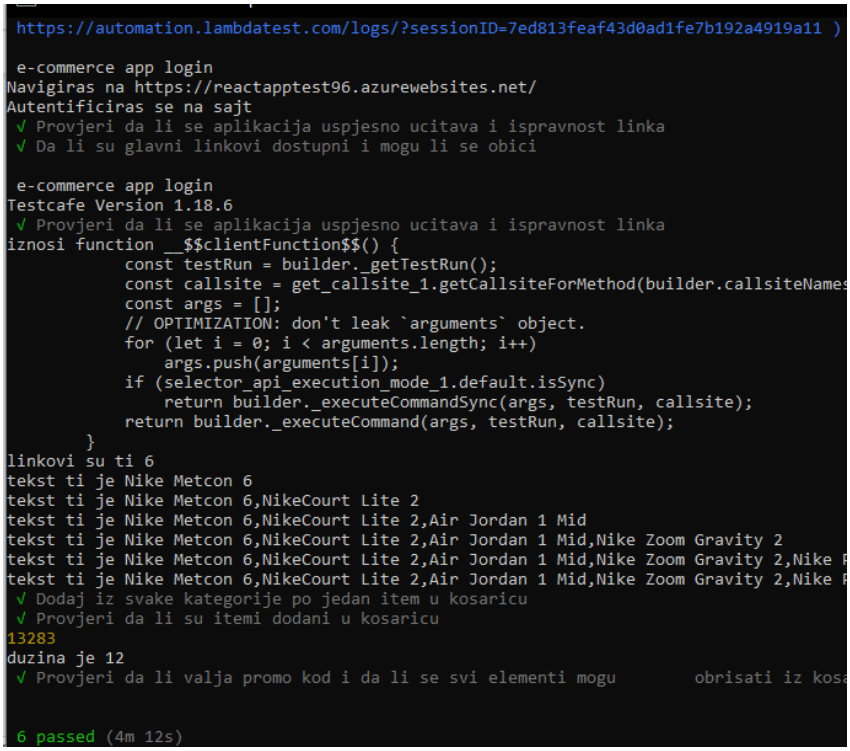
6.2.3 Kreiranje testnih scenarija za lambda testove

Testni scenariji za lambda testove bit će izvršeni oko jedne „ECommerce“ aplikacije za prodaju odjeće. Sami testovi bit će jednostavi kako bi se prikazalo korištenje lambda testova na Microsoft Edge web pregledniku kako bi se dobio uvid da li se aplikacija pravilno ponaša i u takvom okruženju. Testni scenariji za ovu aplikaciju su sljedeći:

- Test dima – predstavljat će jednostavni test scenarij u kojemu je cilj provjeriti da li se aplikacija uspješno učitava, te ispravnost linka. Aplikacija bi pri uspješnoj korisničkoj prijavi trebala u navigaciji imati poruku dobrodošlice te korisničko ime prethodno prijavljenog korisnika u obliku „Hello <korisničko ime>“. Osim toga još jedan bitan test je onaj kojime se provjerava da li se mogu otvoriti sve poveznice, odnosno resursi unutar aplikacije. Nastojat će se provjeriti link na početnoj stranici, te poveznice za mušku, žensku kategoriju, te promo kodove kao sniženja cijena proizvoda.
- Testovi provjere ispravnosti komponenti – u ovome testnom scenariju najprije se provjerava da li postoji komponenta za odjeću na početnoj strani aplikacije, te ispravnost linka. Zatim toga dolazi test provjere da li se artikli po raznim kategorijama mogu dodati u košaricu. Poslije toga dolazi test za provjeru da li su artikli uistinu dodani u košaricu što će biti prikazano brojem artikala u košarici. Završni test ovoga scenarija bit će onaj u kojemu se provjerava ispravnost kodova za sniženje ukupne cijene košarice. Provjera će se vršiti na način u kojemu će se odabrati jedan bonus za sniženje cijene npr. sniženje za 20 eura nakon kojega se provjerava da li je ukupna cijena smanjena točno za takvu vrstu sniženja. Testni scenarij bit će završen provjerom da li se neki dodani artikli u košarici mogu obrisati.

6.2.4 Pokretanje lambda testova na računalu domaćina

Nakon uspješne implementacije testova, vrijeme je da se i oni u konačnici pokrenu prvo u lokalnom okruženju na računalu domaćina, a zatim i integriraju kao jedan od ključnih faza provjere prije isporuke klijentskih komponenti. Naredba kojom će se pokrenuti testovi za Microsoft Edge preglednik u ovoj slučaju bit će *npx testcafe "lambdatest:MicrosoftEdge@91.0:Windows 10" "e2e-testcafe/regression" -c 1 -q -e --browser-init-timeout 180000*. Prethodna naredba nam govori da se koristi Microsoft Edge kao testni web preglednik, točnije njegovu verziju 91, te windows 10 kao specificirani operacijski sustav za pokretanje ovog tipa testa. Zastavice u prethodnoj komandi su razne, npr. zastavica -c označava koliko će se u jednom trenutku instanci web preglednika Microsoft Edge-a koristiti u isto vrijeme testiranja. Nadalje zastavica -q označava broj pokušaja koji su potrebni za potvrđivanje uspjeha testa, ako broj nije eksplicitno zadan, onda se podrazumijeva 3 pokušaja. Na kraju zastavica -e prikazuje da će test ignorirati pogreške u JavaScript kodu kako ne bi došlo do loše reprezentacije prikaza uspješnosti test scenarija. Zadnja zastavica –browser-init-timeout prikazuje koliko vremena dajemo web pregledniku da se poveže sa testcafe-om. U prethodnom primjeru prikazano je vremensko ograničenje za uspostavu veze za 180000 milisekundi, ili 3 minute. Nakon pokretanja prethodne naredbe u terminalu ili sučelju za naredbe dobije se sljedeći prikaz, sličan kao na narednoj slici.



Slika br. 31: Prikaz rezultata nakon lokalnog pokretanja lambda testa

S prethodne slike vidimo poveznicu na platformi na kojoj se „vrte“ lambda testovi, te se vidi isti prikaz testova kao kada bi ih pokrenuli preko testcafe-a. Sa slike je razumljivo da je svih 6 testova uspješno izvršeno što znači da je aplikacija pogodna i za korištenje za Microsoft Edge preglednik, te windows 10 operacijski sustav, te klijenti neće imati problema koristeći aplikaciju uz takve preduvjete.

6.3 Isporuka klijentskih komponenti te integracija lambda testova na Azure DevOps-u

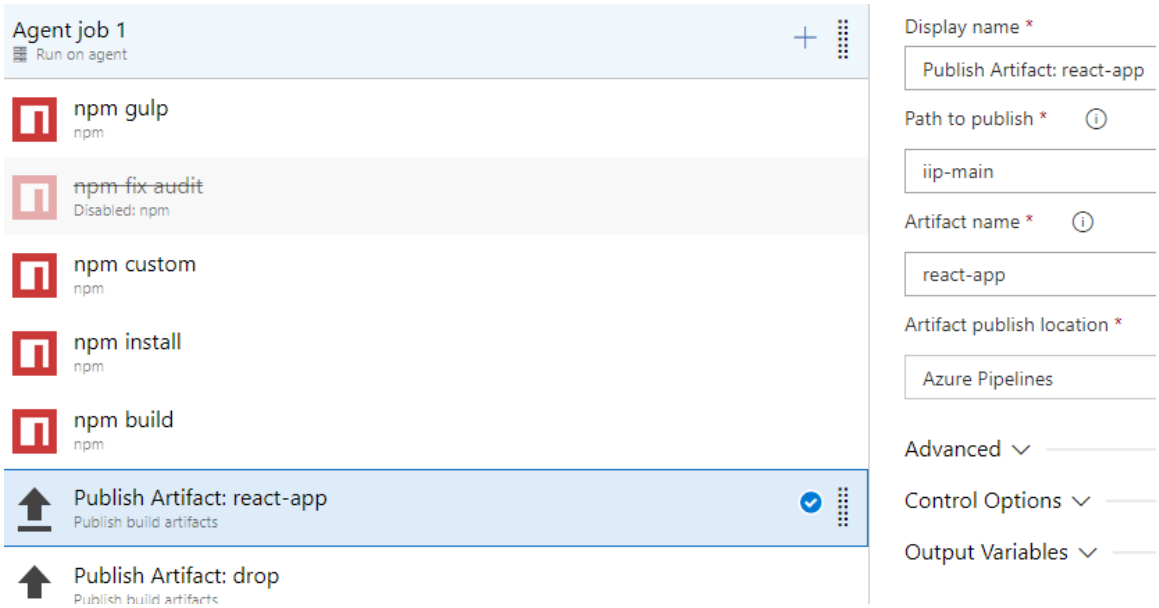
Na sličan način kao u poglavlju 5 bit će prikazana integracija testova, ali uz novo spomenuti dodatak koji se zove izdanje (eng. Release). Izdanje je „konstrukcija“ koja sadrži verzionirani skup artefakata navedenih u cjevovodu kontinuirane integracije i isporuke. Ona uključuje snimku stanja svih informacija potrebnih za izvođenje svih zadataka i radnji u cjevovodu izdanja, kao što su faze, zadaci, politike kao što su okidači i odobravatelji te opcije implementacije. Može postojati više izdanja iz jednog cjevovoda

izdanja, a informacije o svakom od njih se pohranjuju i prikazuju u Azure cjevovodima za neko proizvoljno vrijeme zadržavanja tog izdanja.[39] Kako bi se moglo pokrenuti izdanje potrebno je najprije napraviti cjevovod kontinuirane integracije na sličan način kao u poglavlju 5. U tome cjevovodu „razvit“ će se najprije aplikacija te proizvesti artefakt koji će se nadalje koristiti u izdanju koje će isto biti vlastito definirano.

6.3.1 Postavljanje cjevovoda za kontinuiranu integraciju

Nakon što se aplikacija uspješno prenese na Azure DevOps repozitorij preko nekoga alata za verzioniranje npr. gita, tada može započeti korak kreiranja cjevovoda. Kao i u prošlom primjeru, agent koji će se ovdje koristiti bit će tzv „self-hosted agent“ lokalno instaliran i pokrenut kao servis na računalu domaćina. Prvi zadatak koji će se pokrenuti je „npm gulp“ koji će instalirati gulp kao koristan alat za automatizaciju zadataka. Sljedeći zadatak je također npm zadatak, ali u njemu se specificira naredba *install --legacy-peer-deps* kako npm ne bi instalirao ovisnosti o paketima koje bi mogle poslije dovesti do problema u „sukobima“ između različitih verzija paketa naše aplikacije. Tako će se izbjeći problemi u instalaciji postojećih paketa. Nakon toga zadatka na red dolazi tipični „npm install“ zadatak koji će instalirati sve pakete te provjeriti da li se aplikacija „ruši“ do toga koraka. Kada se svi paketi uspješno instaliraju tada je na redu sljedeći zadatak u cjevovodu, a to je „npm build“ zadatak koji će izgraditi kod postojeće aplikacije. Pošto je cilj ostvariti isporuku klijentskih komponenti te provjeru koda koristeći lambda testova, važno je dodati još 2 nova zadatka. Jedan od njih bit će nazvan „Publish Artifact: react-app“ koji će proizvesti *artefakt* i učiniti ga dostupnim za vršenje lambda testova nad našim kodom. Na sličan način bit će implementiran i zadatak nazvan „Publish Artifact: drop“ i on će koristiti prethodno izgrađeni kod aplikacije, te proizvesti artefakt koji će aplikaciju isporučiti tako da ona postane javno dostupna, odnosno da je u produkcijskom okruženju. Prethodno opisani zadaci cjevovoda pronalaze se na slici br. 32.

[39] https://docs.microsoft.com/en-us/azure/devops/pipelines/release/releases?view=azure-devops



Slika br. 32: Prikaz zadataka za cjevovod kontinuirane integracije ecommerce aplikacije

Nakon definiranja cjevovoda na način opisan na prethodnoj slici potrebno je osigurati kontinuiranu integraciju tako da se u sekciji „triggers“ odabere opcija „Enable continuous integration“ koja će pri svakoj promjeni koda i prijenosu promjena na Azure DevOps repozitorij u aplikaciji automatski pokrenuti ovaj cjevovod.

6.3.2 Definiranje izdanja za kontinuiranu isporuku i lambda testova

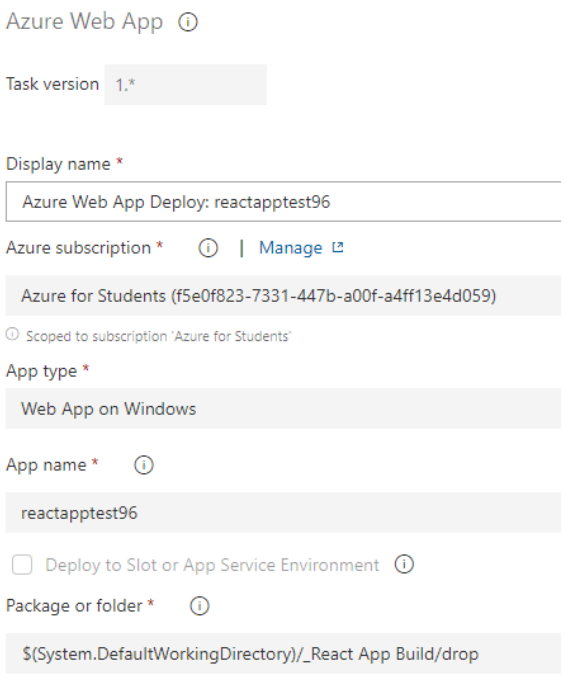
Cjevovodi za izdavanje omogućuju timovima da kontinuirano implementiraju svoju aplikaciju u različitim fazama s manjim rizikom i bržim tempom. Postavljanje u svaku fazu može se u potpunosti automatizirati korištenjem poslova i zadataka. Timovi također mogu iskoristiti prednosti i značajke odobrenja za kontrolu tijeka rada cjevovoda implementacije. Svaka faza u cjevovodu izdanja može se konfigurirati s uvjetima prije i nakon implementacije koji mogu uključivati čekanje da korisnici ručno odobre ili odbiju implementacije i provjeru s drugim automatiziranim sustavima jesu li ispunjeni specifični uvjeti. Uz to, timovi mogu podesiti ručne provjere valjanosti kako bi pauzirali cjevovod implementacije i potaknuli korisnike da izvrše ručne zadatke, a zatim nastave ili „odbiju“ implementaciju.[40] Način na koji će cjevovod izdanja biti definiran u ovome radu je onaj

[40] https://docs.microsoft.com/en-us/azure/devops/pipelines/release/approvals/?view=azure-devops

gdje se najprije pokrene cjevovod iz prošloga poglavlja, a njegov proizvod odnosno *artefakt* bit će korišten u cjevovodu izdanja tako da se aplikacija kontinuirano isporučuje, te će se nad istom vršiti lambda testovi za 3 vrste web preglednika.

6.3.3 Implementacija cjevovoda izdanja

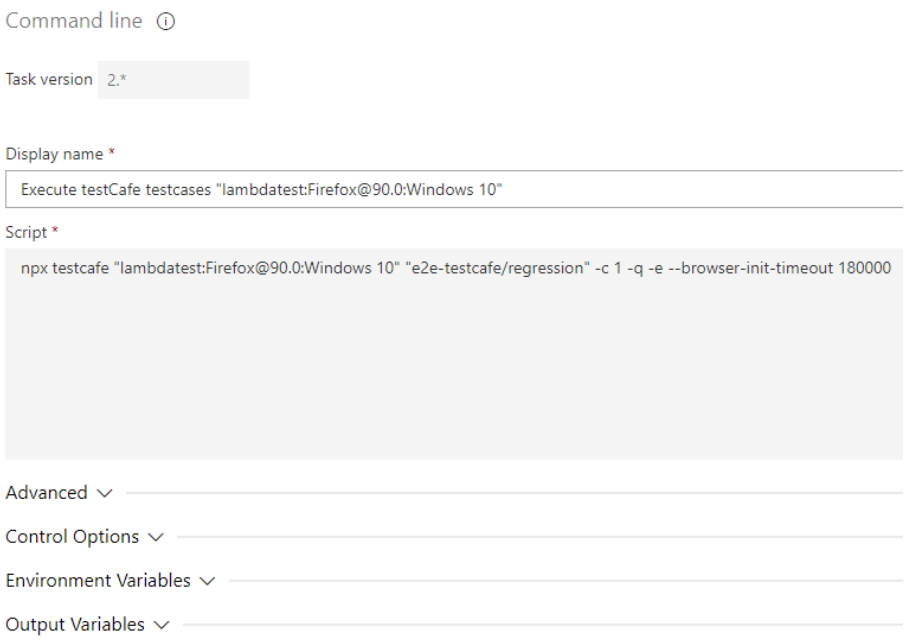
Cjevovodi izdanja pratit će prikaze na sljedećim slikama, a glavne značajke tog cjevovoda su prethodno definirana isporuka klijentskih komponenti te provedba lambda testova. Kontinuirana isporuka će imati jedan glavni zadatak, a to je razvijanje aplikacije za proizvodno okruženje. Sami postupak podešavanja web aplikacije na azure portalu neće biti dio ovoga rada. Glavni korak isporuke te aplikacije objašnjava slika br. 33.



Slika br.33: Definiranje cjevovoda za isporuku aplikacije u proizvodno okruženje

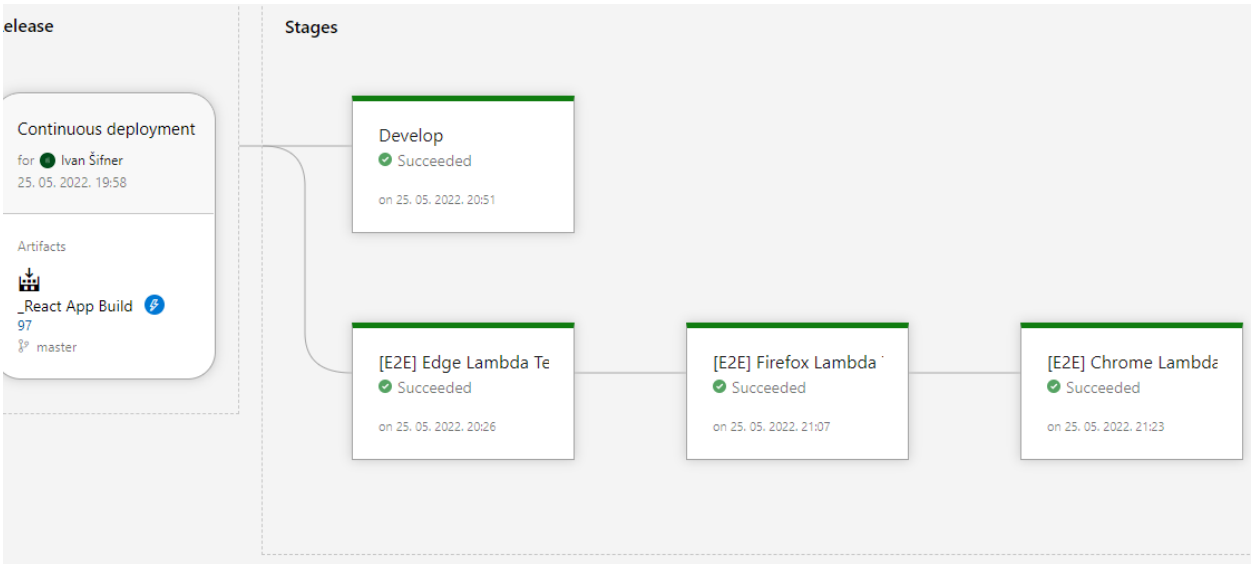
S prethodne slike vidimo zadatak koji se koristi pri isporuci, a on je “Azure Web App” koji koristi definiranu licencu konfiguriranu na azure portal, te je aplikacija namijenjena windows operacijskom sustavu, a testno je nazvana „reactapptest96“. Potrebno je napomenuti da ovaj zadatak koristi artefakt “drop” koji je proizvod cjevovoda objašnjenog u poglavlju 6.3.1. Nakon definiranja ovakvog tipa zadatka aplikacija će moći da se

isporučuje bez problema u proizvodno okruženje. Izdanja je potrebno proširiti sa zadacima za unakrsno testiranje web preglednika tako da se za svaki od web preglednika zasebno definira faza unutar cjevovoda za izdanja. Primjer od jednoga web preglednika, konkretno preglednika Firefox prikazan je na sljedećoj slici.



Slika br. 34: Postavljanje lambda testa za Firefox web preglednik

Na prethodnoj slici prikazan je zadatak za korištenje skripte iz komandne linije za pokretanje Firefox web preglednika za windows 10 operacijski sustav, te put do testnih scenarija uz zastavice definirane na sličan način na računalu domaćina opisanog u prošlom poglavlju. Osim toga potrebno je postaviti i varijable okruženja, a to su LT_ACCESS_KEY i LT_USERNAME koje se mogu definirati direktno u sekciji „Environment Variables“, a mogu se i definirati u varijablama na razini ovoga cjevovoda izdanja. Kada se definira način isporuke klijentskih komponenti te dodatni sloj za provjeru lambda testnih scenarija, tada se jamči određena doza sigurnosti da je ono što radimo uistinu ispravno i „istestirano“ na najbolji moderni način. Slika br. 35 opisuje izgled cijeloga cjevovoda izdanja, te sve faze koje su definirane da bi se prethodno opisana sigurnost i pouzdanost ostvarila.



Slika br.35: Nacrt cijeloga cjevovoda izdanja

Slika br. 35 opisuje ugrubo sve što je potrebno da se događa kako bi imali sigurnu i kvalitetnu aplikaciju. Na njoj je prikazan artefakt kojega ovo izdanje, odnosno različite faze ovoga izdanja koriste. Taj artefakt predstavlja proizvod cjevovoda kontinuirane integracije iz poglavlja 6.3.1. Faze ovoga cjevovoda su „Develop“, „[E2E] Edge Lambda Test“, „[E2E] Firefox Lambda Test“, te „[E2E] Chrome Lambda Test“. Sa slike vidimo da je svaka faza uspješno izvršena, odnosno da je kod aplikacije prošao testne scenarije za preglednike Microsoft Edge, Firefox, te u konačnici Google Chrome. Osim toga vidljivo je da je aplikacija uspješno isporučena u proizvodno okruženje pošto je faza cjevovoda izdanja „Develop“ prikazana uspješnom zelenom bojom te statusom uspjeha (eng. Succeeded).

6.4 Testovi vizualne regresije

Testiranje vizualne regresije provodi se za provođenje izgleda aplikacije iz korisničke perspektive regresijskog testiranja prikupljanjem snimki zaslona korisničkog sučelja i uspoređivanjem s izvornim ili osnovnim slikama. Ako postoje mala odstupanja od osnovne slike koja koristi snimljeno korisničko sučelje za mobilne uređaje, mehanizam vizualne regresije će na to upozoriti i istaknuti problem „piksel po piksel“. Vizualno testiranje ispituje vizualni rezultat aplikacije i uspoređuje ga s rezultatima koje je predvidio dizajner. Vizualni testovi mogu se provesti u bilo kojem trenutku na bilo kojoj aplikaciji s grafičkim korisničkim sučeljem gdje većina programera provodi vizualne testove na pojedinačnim komponentama tijekom razvoja, a tijekom testiranja *s kraja na kraj* pokreću vizualne testove na potpuno funkcionalnom programu. Dakle, regresijski test funkcionira slično kao tehnika provjere, a testni slučajevi su često automatizirani jer se moraju izvršavati više puta, gdje ručno izvršavanje istih može biti naporno. Jedni od glavnih benefita testova vizualne regresije su [41]:

1. Stotine zahtjeva, te ručnih provjera mogu se zamjeniti s jednom regresijskom „slikom“.
2. Automatizirano vizualno testiranje ispituje korisničko sučelje na raznim preglednicima, zaslonima i uređajima.
3. Mogu se istaknuti značajne ili male razlike u izglednu korisničkog sučelja
4. Pomaže poboljšati sposobnost funkcioniranja automatiziranih provjera.
5. Ono oslobađa vrijeme testnim analitičarima da se usredotoče na tumačenje težih i značajnijih izazova.
6. Sjajan vizualni dizajn doprinosi razvoju povjerenja u ispravnost proizvoda koji se razvija

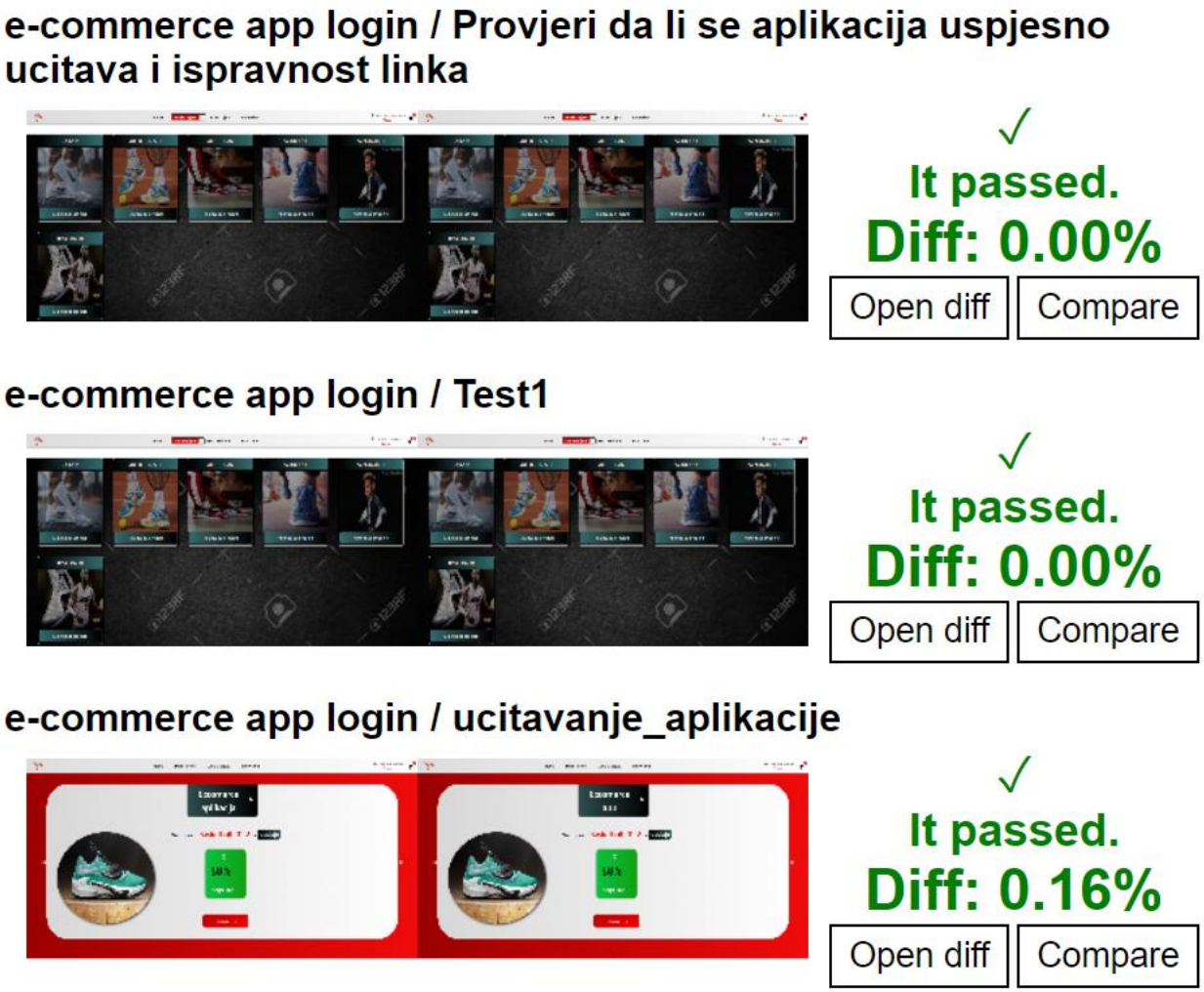
6.4.1 Implementacija testova vizualne regresije na računalu domaćina

Testovi vizualne regresije bit će primijenjeni na prethodnoj „ECommerce“ aplikaciji. Oni će se provesti tako da će se za neke testne scenarije načiniti slika stanja za koju će se

[41] https://www.qed42.com/insights/coe/quality-assurance/everything-you-need-know-about-automated-visual-regression-testing

smatrati da predstavlja očekivano stanje aplikacije, a zatim će se na isti način ispitati trenutno stanje aplikacije. Razlike između slika očekivanog i trenutnog stanja bit će prikazane na slici koja će ih označiti tako da tester u svakom trenutku može vidjeti što se na aplikaciji promijenilo ili poremetilo tijekom implementacije. Vizualna regresija provesti će se korištenjem testcafe biblioteke testcafe-blink-diff. Nakon dodavanja te biblioteke može se krenuti na implementaciju vizualne regresije. Najbolje je za takvu vrstu testiranja dodati 2 koraka u package.json datoteku. Jedan od njih bit će skripta za pokretanje prikaza očekivanog izgleda aplikacije pod nazivom test:base, a ona će definirati naredbu *testcafe chrome src/e2e-testcafe/regression/**/*.e2e.js -s src/images --take-snapshot base.* Ona nam govori da će se koristiti testovi s lokacije src/e2e-testcafe/regression te da će se odnositi na sve testove unutar te datoteke prikazano simbolom **, a koji završavaju s proširenjem .e2e.js. Osim toga koristit će se zastavica -s koja je kratica za sliku stanja (eng. Snapshot) nakon koje slijedi lokacija unutar projekta gdje će se očekivane slike stanja aplikacije spremiti. Naredbom *–take-snapshot base* pokrenuti će se komanda za snimanje stanja aplikacije. Ovu naredbu potrebno je izvesti samo jednom u onome trenutku za kojeg se vjeruje da odražava željeno stanje aplikacije, nakon čega će za svaku sliku testnog scenarija kojeg definiramo proizvesti stanje aplikacije s nazivom „*base*“. Na sličan način u package.json datoteku dodat će se skripta „test:actual“ koja će definirati naredbu *testcafe chrome src/e2e-testcafe/regression/**/*.e2e.js -s src/images --take-snapshot actual.* Ova skripta razlikuje se od prethodne samo u naredbi *–take-snapshot* actual koju je potrebno izvršiti svaki puta kada se rade promjene u izgledu korisničkog sučelja aplikacije kako bi se provjerilo da li je došlo do „rušenja“ željenog izgleda aplikacije. Naredb*a –take-snapshot* actual proizvesti će stanje aplikacije pod nazivom „actual“. Kada se u terminalu unese naredba *npx testcafe-blink-diff src/images reports/google --compare base:actual --open --threshold: 0.03 # <= 3% is OK* dobiti će se prikaz razlika između prethodno definiranih slika stanja aplikacije. U toj naredbi specificirano je da će jedni od parametara biti direktorij u kojemu se nalaze slike koje su proizvod naredbi *test:base* i *test:actual.* Osim toga definira se i željeni prag tolerancije koji u ovome slučaju iznosi 3%, odnosno razlika između slika stanja ne smije biti veća od tog broja, u protivnom će se test smatrati neuspješnim i pasti će na testiranju vizualne regresije. Odabirom te naredbe dobiti će se razlika između slika koji se mogu pronaći na

prethodno definiranom putu u datoteci *reports/google*. Razlika između slika stanja aplikacije prikazana je na sljedećoj slici.

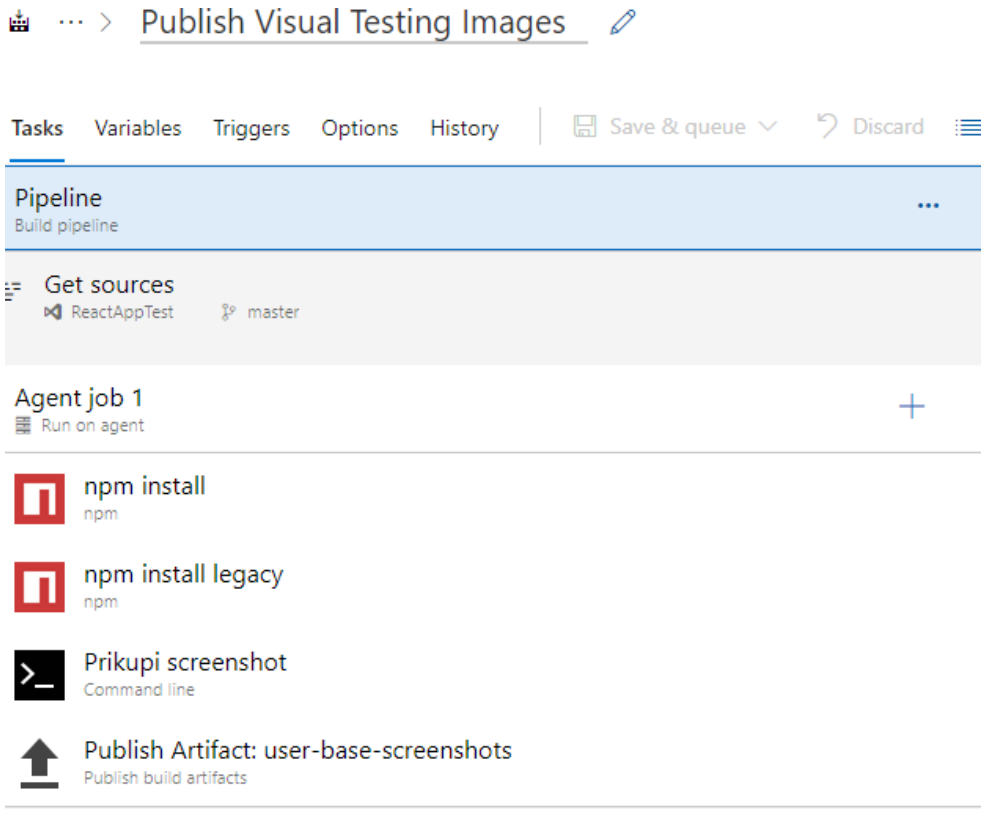


Slika br.36: Prikaz razlike između slika stanja korištenje vizualne regresije

Na prethodnoj slici vidljivo je da razlike između prve dvije slike željenog stanja aplikacije i trenutnog stanja ne postoje, dok pak u trećem slučaju postoje male razlike u udjelu od 0.16%. Ta razlika je posljedica uglavnom prilikom provođenja testova kada su se dodavali različiti artikli u košaricu tako je rastao i njihov broj u njoj pa je takva razlika minimalna, a pošto je definiran prag tolerancije da razlika između slika stanja ne smije prijeći 3%, tada se dolazi do zaključka da su definirani testni scenariji uspješno prošli test vizualne regresije.

6.4.2 Postavljanje testova vizualne regresije u cjevovodu za kontinuiranu integraciju

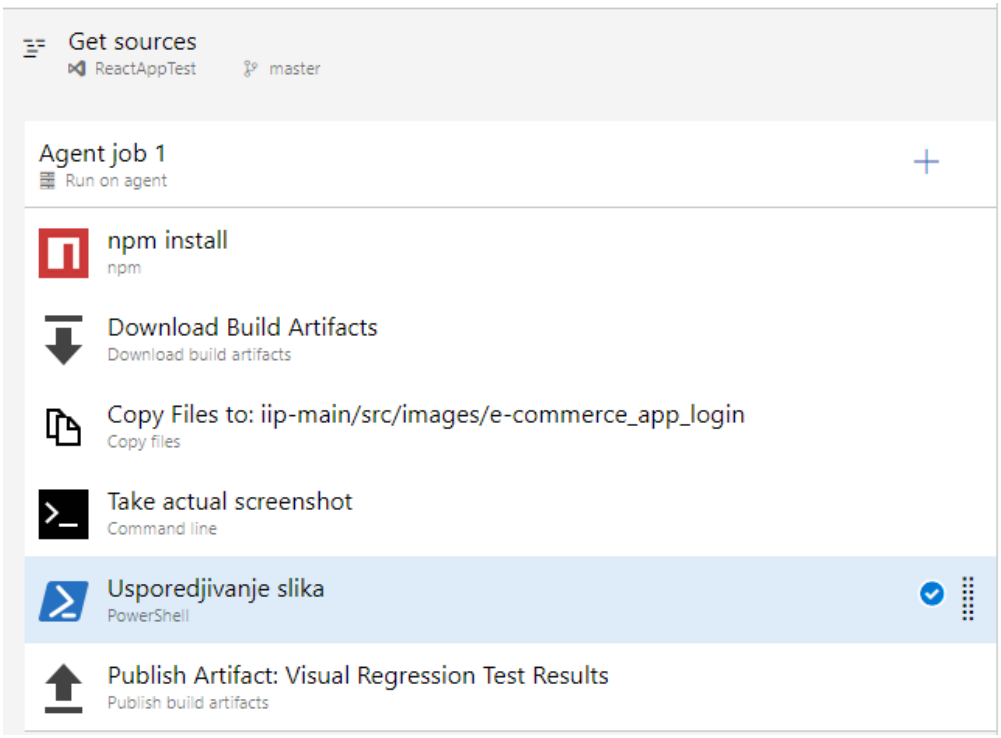
Testovi vizualne regresije odlični su u pronalaženju vizualnih odstupanja stanja aplikacije, ali njihovo ručno pokretanje s računala domaćina ne zadovoljava kriterij kontinuirane integracije, pa je za isti to potrebno i dodati. Ova implementacija cjevovoda bit će malo drugačija i u njoj će se definirati 2 cjevovoda. Prvi će biti onaj koji će se izvršiti samo jednom i on će proizvesti sliku željenog stanja aplikacije, a drugi će vršiti usporedbu tih stanja. Cjevovod za proizvodnju slike željenog stanja aplikacije prikazan je na sljedećoj slici.



Slika br. 37: Prikaz cjevovoda za proizvodnju slike željenog stanja aplikacije

Na prethodnoj slici definirani su zadatci cjevovoda, a jedni od njih su već dobro poznati „npm install“ i „npm install legacy“. Ovdje su dodani i dodatni zadaci, a jedan od najbitnijih je svakako zadatak „Prikupi screenshot“ koji će koristiti istu naredbu kao na računalu domaćina, a ona je *npm run test:base*. Nakon nje potrebno je definirati još jedan zadatak, a to je zadatak za objavu artefakta „Publish Artifact: user-base-screenshots“ koji će imati

ulogu objave željene slike stanja aplikacije koju će onda drugi cjevovod „konzumirati“ kako bi mogao vršiti usporedbu s trenutnim stanjem. Artefakt koji će biti proizveden ovim cjevovodom zvat će se „user-base-screenshots“ i upravo taj bit će korišten od narednog cjevovoda. Nakon definirana cjevovoda kontinuirane integracije za generiranje slike željenog stanja potrebno je definirati i cjevovod za usporedbu slika stanja. Prikaz takvog cjevovoda vidi se na slici br. 38.



Slika br. 38: Prikaz cjevovoda za usporedbu slika stanja aplikacije

Ovaj cjevovod sastoji se od dosta bitnih koraka. Osim spomenutog koraka koji je zajednički za svaki cjevovod „npm install“ ovdje je korišten i zadatak „Download Build Artifacts“ koji će preuzeti artefakt definiran u prethodnom cjevovodu. Nakon toga zadatka dolazi red na takozvani „Copy Files to:“ zadatak koji će imati ulogu u prijenosu preuzetoga artefakta na lokaciju gdje će dodati i svoju sliku stanja. Sliku trenutnog stanja ostvarit će pomoću zadatka „Take actual screenshot“ koji će koristiti naredbu *npm test:actual* na način kako je pokrenuta na već spomenutom računalu domaćina. Za usporedbu generiranih slika koristit će se zadatak „Uspoređivanje slika“ koji će koristiti naredbu *npx*

testcafe-blink-diff src/images reports/google --compare base:actual --open --threshold: 0.03 # <= 3% is OK koja će vršiti provjeru slika za prag tolerancije razlika od 3% na način koji je već prikazan na slici br. 37. Ako se svi zadaci uspješno izvrše tada će se moći prikazati rezultat pomoću zadatka „Publish Artifact: Visual Regression Test Results“ koji će prikazati razliku stanja slično kao i slika br. 37. Na sljedećoj slici prikazat će se kako izgleda kada aplikacija tijekom cjevovoda premaši definirani prag od 3% tako što će se promijeniti naziv aplikacije od „ECommerce app“ u „ECommerce aplikacija za praćenje elemenata u košarici tako da se provjerava test vizualne regresije“. Kao rezultat dobiti ćemo da cjevovod nije prošao test te poruku o premašenom pragu tolerancije kao što je prikazano na sljedećoj slici.

```
Processing 3 files...
    Failed with 69.54% of differences
  e-commerce app login/ucitavanje_aplikacije DONE
    Failed with 41.70% of differences
  e-commerce app login/Provjeri da li se aplikacija uspjesno ucitava i ispravnost linka DONE
    Failed with 41.75% of differences
  e-commerce app login/Test1 DONE
Write reports\google\index.html
##[error]PowerShell exited with code '1'.
Finishing: Usporedjivanje slika
```

Slika br.39: Neuspješan test vizualne regresije

Na prethodnoj slici vidimo kako su testovi premašili prag tolerancije za 3 slike stanja na tako da je prvi test imao najveću razliku od čak 69.54%, drugi 41.70%, a posljednji 41.75%. Ako se dogodi nešto ovakvo to nam govori da su aplikaciji potrebne nužne promjene te da je više vremena potrebno uložiti u kvalitetnu izgradnju koda te uspostavu praksi i standarda kako do ovoga u budućnosti ne bi došlo, kako bi aplikacija postala uspješnija.

7. Sažetak

Zadatak ovoga rada bio je prikazati problematiku načina testiranja klijentskih komponenti uz učestalo testiranje te kontinuiranu integraciju i isporuku. Osim brojnih problema s kojima se kompanije susreću u razvoju programskog proizvoda, sve više se osjeća potreba za testiranje na svakom „koraku“ u kojemu se integracija pokazuje kao ključan korak isporuke kvalitetnog softvera. Izostanak kvalitetnog testiranja može imati brojne posljedice, a jedne od najvažnijih su pogreške u proizvodnom okruženju koje će klijenti uočiti, te odlazak kod konkurencije koja provodi stalno testiranje te kontinuiranu integraciju i isporuku klijentskih komponenti. U početnom djelu rada cilj je bio istaknuti važnost integracije i isporuke te testiranja objašnjavajući neke benefite i izazove u slučaju da se kompanija odluči upotrebljavati moderne tehnologije. Osim toga bilo je riječ i o vrstama testiranja s naglaskom na testiranje klijentskih komponenti te razlika između pojedinih od njih kao što su testiranje komponente, testiranje integracije, testiranje *s kraja na kraj* i slično. Nakon teorijske podloge bilo je važno implementirati stečeno znanje u praksi i dokazati te benefite pa je drugi dio rada obuhvaćao implementaciju u modernim tehnologijama kao što su Azure DevOps te integracija testova *s kraja na kraj*. S time se željelo dokazati te isporučiti prikaz broja testova te njihove statuse o uspješnosti. Nadalje cilj ovoga rada bio je prikazati i isporuku klijentskih komponenti preko kreiranja cjevovoda izdanja, te zaključiti testiranje s testovima vizualne regresije. One kompanije koje koriste spomenute načine testiranja, integracije i isporuke svakako će imati opipljive benefite, te će se uzdići iznad konkurencije i „pridobiti“ što više korisnika.

Abstract

The task of this thesis was to present the issue of how to test client components with frequent testing and continuous integration and delivery. In addition to the many problems that companies face in software product development, it increasingly feels the need to test at every "step" in which integration takes a key part in the delivery of quality software. Lack of quality testing can have many consequences, and one of the most important are errors in the production environment that customers will notice, and going to the competition that conducts constant testing and continuous integration and delivery of customer components. In the initial part of the thesis, the goal was to emphasize the importance of integration and delivery and testing, explaining some of the benefits and challenges in case the company decides to use modern technologies. In addition, many types of testing have been explained with an emphasis on testing client components and the differences between some of them, such as component testing, integration testing, end-to-end testing etc. After the theoretical background, it was important to implement the acquired knowledge in practice and prove these benefits, so the second part of the thesis included implementation in modern technologies such as Azure DevOps and integration of *end-to-end* tests. The aim was to prove and deliver a display of the number of tests and their performance statuses. Furthermore, the aim of this thesis was to show the delivery of client components through the creation of the release pipeline, and to finish testing with visual regression tests. Those companies that use the mentioned methods of testing, integration and delivery will certainly have tangible benefits, and will rise above the competition and "gain" as many users as possible.

Popis literature:

Dorothy Graham, Erik van Veenendaal, Isabel Evans, Rex Black - Foundations of Software Testing: ISTQB Certification

https://www.softwaretestinghelp.com/how-is-cross-browser-testing-performed/

https://cdohotaru.medium.com/principles-of-end-to-end-web-testing-7fa7d27d7826

https://www.bmc.com/blogs/what-is-ci-cd/

https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html

https://www.infostretch.com/blog/the-road-to-cicd-a-short-history-of-agile-development/

https://jhall.io/archive/2021/09/26/a-brief-history-of-ci/cd/

https://www.netapp.com/knowledge-center/what-is-continuous-integration-continuous-delivery-cicd/

https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know

https://www.bmc.com/blogs/what-is-ci-cd/

https://www.delphix.com/glossary/what-is-ci-cd-pipeline

https://codete.com/blog/business-value-of-ci-cd

https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/

https://www.lambdatest.com/blog/benefits-of-ci-cd/

https://www.infoq.com/articles/cd-benefits-challenges/

https://www.perfecto.io/blog/comprehensive-guide-end-end-e2e-testing

https://www.testim.io/blog/test-automation-benefits/

https://techbeacon.com/app-dev-testing/top-11-open-source-testing-automation-frameworks-how-choose

https://applitools.com/blog/migrating-protractor-tests-angular/

https://www.browserstack.com/guide/protractor-testing-tutorial

https://dzone.com/refcardz/end-to-end-testing-automation-essentials

https://www.qentelli.com/thought-leadership/insights/top-codeless-testing-tools

https://testguild.com/testcafe/

https://codedec.com/tutorials/assertion-in-testcafe/

https://javascript.plainenglish.io/power-of-user-roles-in-testcafe-b88458baae17

https://geekflare.com/yaml-introduction/

https://testsigma.com/cross-browser-testing

https://thenewstack.io/how-lambdatest-automates-cross-browser-testing-from-the-cloud/

https://docs.microsoft.com/en-us/azure/devops/pipelines/release/releases?view=azure-devops

https://docs.microsoft.com/en-us/azure/devops/pipelines/release/approvals/?view=azure-devops

https://www.qed42.com/insights/coe/quality-assurance/everything-you-need-know-about-automated-visual-regression-testing