Testiranje klijentskih komponenti web aplikacija pomoću alata Playwright

Cetina, Kristijan

SVEUČILIŠTE JURJA DOBRILE U PULI
FAKULTET INFORMATIKE

Kristijan Cetina

Testiranje klijentskih komponenti web aplikacija pomoću alata Playwright

DIPLOMSKI RAD

Pula, 2024.

SVEUČILIŠTE JURJA DOBRILE U PULI
FAKULTET INFORMATIKE

Kristijan Cetina

Testiranje klijentskih komponenti web aplikacija pomoću alata Playwright

DIPLOMSKI RAD

JMBAG:	**2424011721, izvanredni student**
Studijski smjer:	**Informatika**
Kolegij:	**Razvoj IT rješenja**
Znanstveno područje:	**Društvene znanosti**
Znanstveno polje:	**Informacijske i komunikacijske znanosti**
Znanstvena grana:	**Informacijski sustavi i informatologija**
Mentor:	**dr.sc. Nikola Tanković**

Pula, rujan, 2024. godine

Zahvala

Zahvaljujem svojem mentoru na izdvojenom vremenu i podršci, kako na izradi ovog rada, tako i tijekom cijelog studiranja na Fakultetu informatike.

Zahvaljujem se i svojim timskim kolegama, s kojima sam od samog početka sudjelovao na svim timskim zadatcima i njihovoj pomoći pri individualnom radu. Bez vas ovo iskustvu ne bi bilo niti upola zabavno kao što je bilo.

Zahvaljujem se i svim ostalim profesorima i djelatnicima na nesebičnoj potpori kada je god to bilo potrebno.

Naposljetku veliko hvala mojoj obitelji na potpori i razumijevanju tijekom mojeg ponovnog studiranja.

Sažetak

Kako bi se osigurala isporuka kvalitetnog programskog rješenja, testiranje softvera je jednako važno kao i samo kodiranje. Kako se kompleksnost aplikacija povećava, raste i vjerojatnost pojave pogrešaka koje mogu negativno utjecati na korisničko iskustvo. Kako bismo to spriječili, koristimo razne tehnike testiranja, a jedna od njih je testiranje klijentskih komponenti.

Ovaj rad se fokusira na korištenje Microsoftovog alata Playwright za testiranje klijentskih komponenti. Playwright je popularan izbor za end-to-end testiranje modernih web aplikacija jer omogućuje pouzdano testiranje na različitim preglednicima i platformama. Njegove glavne prednosti uključuju podršku za najnovije web tehnologije, jednostavnu sintaksu i mogućnost pisanja robusnih i održivih testova.

Ključne riječi

Playwright, JavaScript, open-source

Sommario

Per garantire la consegna di una soluzione software di qualità, il test del software è importante tanto quanto la programmazione stessa. Con l'aumentare della complessità delle applicazioni, cresce anche la probabilità di errori che possono influire negativamente sull'esperienza utente. Per evitare ciò, utilizziamo diverse tecniche di test, una delle quali è il test delle componenti client-side.

Questo lavoro si concentra sull'uso dello strumento Playwright di Microsoft per testare le componenti client-side. Playwright è una scelta popolare per il testing end-to-end delle applicazioni web moderne perché consente di testare in modo affidabile su diversi browser e piattaforme. I suoi principali vantaggi includono il supporto per le tecnologie web più recenti, una sintassi semplice e la capacità di scrivere test robusti e sostenibili.

Parole chiave:

Playwright, JavaScript, open-source

Abstract

To ensure the delivery of a quality software solution, software testing is just as important as coding itself. As the complexity of applications increases, so does the

likelihood of errors that can negatively impact the user experience. To prevent this, we use various testing techniques, one of which is client-side component testing.

This paper focuses on using Microsoft's Playwright tool for testing client-side components. Playwright is a popular choice for end-to-end testing of modern web applications because it allows reliable testing across different browsers and platforms. Its main advantages include support for the latest web technologies, simple syntax, and the ability to write robust and maintainable tests.

Keywords:

Playwright, JavaScript, open-source

Popis oznaka i kratica

Kratica	**Opis**
POM	Page Object Model - Objektni model stranice
DOM	Document Object Model - Objektni model dokumenta
JSON	JavaScript Object Notation - format datoteke
.md	Markdown datoteka

Korišteni strani pojmovi

Pojam	**Opis**
Product owner	Po Scrum metodologiji rada "Vlasnik proizvoda" je osoba posvećena maksimiziranju vrijednosti proizvoda
Scrum	Scrum je agilni okvir za timsku suradnju koji se obično koristi u razvoju softvera i drugim industrijama

Sadržaj

Poglavlje 0

Uvod

U kontekstu sve bržeg razvoja softverskih rješenja i povećanih zahtjeva za kvalitetom, automatizacija testiranja se etablirala kao nezaobilazni dio razvojnog procesa. Među različitim vrstama testiranja, end-to-end (E2E) testiranje zauzima posebno mjesto, simulirajući realistične scenarije korištenja kako bi se osiguralo da softverski sustav funkcionira kao koherentna cjelina.

E2E testiranje predstavlja sveobuhvatan pristup provjeri funkcionalnosti aplikacije od početka do kraja, uključujući interakciju s različitim komponentama, bazama podataka, vanjskim sustavima i korisničkim sučeljima. Glavni cilj ovakvog testiranja je potvrditi da su svi dijelovi sustava integrirani i da zajedno ispunjavaju postavljene funkcionalne zahtjeve.

S obzirom na sve veću kompleksnost modernih softverskih sustava, koji često uključuju mnoštvo međusobno povezanih komponenti i usluga, ručno provođenje E2E testova postaje sve zahtjevnije i sklono pogreškama. Automatizacija ovih testova omogućuje značajnu uštedu vremena i resursa, te osigurava dosljednost i ponovljivost testiranja. x Međutim, implementacija automatiziranih E2E testova nije bez izazova. Neki od najčešćih problema uključuju:

- Krhkost testova: Promjene u korisničkom sučelju ili podacima mogu dovesti do čestih prekida testova, što zahtijeva kontinuirano održavanje.
- Složenost postavke: Konfiguracija okruženja za E2E testiranje može biti kompleksna, posebno za velike i distribuirane sustave.
- Vrijeme izvršavanja: E2E testovi su često dugotrajni, što može usporiti razvojni proces.
- Odabir pravih alata: Izbor odgovarajućih alata za automatizaciju E2E testiranja ovisi o specifičnim potrebama projekta.

Usprkos navedenim izazovima, prednosti automatiziranog E2E testiranja su neosporne. Ovakav pristup omogućuje rano otkrivanje i otklanjanje grešaka, poboljšava kvalitetu softvera, smanjuje troškove održavanja i ubrzava proces izdanja.

Opis i definicija problema

Automatizirano end-to-end (E2E) testiranje predstavlja integralni dio suvremenog razvoja softvera, omogućavajući sveobuhvatnu provjeru funkcionalnosti aplikacije od početka do kraja. Međutim, implementacija i održavanje učinkovitih E2E testova suočava se s nizom izazova koji ograničavaju njihovu sveobuhvatnu primjenu.

Složenost dizajna testova predstavlja jedan od ključnih problema. Kreiranje robusnih E2E testova zahtijeva duboko razumijevanje kompleksnih interakcija unutar sustava, što se posebno ističe u velikim i distribuiranim arhitekturama. Svaka promjena u sustavu može utjecati na višestruke testove, zahtijevajući njihovu reviziju ili ponovno pisanje.

Održivost testova je još jedan izazov koji se javlja zbog dinamične prirode razvoja softvera. Česte promjene u kodu, korisničkom sučelju i podacima mogu dovesti do zastarjelosti postojećih testova, što zahtijeva kontinuirana ulaganja u njihovo održavanje.

Vrijeme izvršavanja predstavlja značajnu prepreku za široku primjenu E2E testova. Zbog sveobuhvatne prirode, ovi testovi obično zahtijevaju značajno više vremena za izvršavanje u usporedbi s drugim vrstama testova (npr. unit testovima). Dugo trajanje izvršavanja može usporiti razvojni proces i otežati brzu povratnu informaciju.

Pouzdanost rezultata je ključna za uspjeh automatiziranih testova. Lažno pozitivni ili negativni rezultati mogu dovesti do gubitka povjerenja u testnu infrastrukturu i uzrokovati nepotrebne troškove. Faktori koji utječu na pouzdanost uključuju stabilnost testnog okruženja, kvalitetu testnih podataka i robusnost same testne logike.

Integracija s CI/CD procesima je još jedan aspekt koji treba pažljivo razmotriti. Uspješna implementacija automatiziranih E2E testova zahtijeva njihovu tijesnu integraciju s kontinuiranim integracijskim i isporučnim (CI/CD) procesima. Ovo podrazumijeva visok stupanj automatizacije, orkestracije i usklađenosti s postojećim razvojnim praksama.

Struktura rada

Struktura ovoga rada podijeljena je u logičke cjeline. Nakon uvoda i objašnjavanja rada, u poglavlju 1 - Testiranje programskog rješenje i osiguranje kvalitete objašnjen je proces testiranje i osiguranja kvalitete programskog rješenja.

Poglavlje 2 - Alat Playwright detaljnije objašnjava korištenu biblioteku kao i njezino korištenje.

Poglavlje 3 - Testiranje nakon nadogradnje na novu verziju objašnjava proces testiranja nakon objave nove verzije kao i problem koji se pokušava riješiti ovim radom.

Poglavlje 4 - Studija slučaja pruža detaljniji uvid u samo rješenje koje je implementirano unutar kompanije u kojoj autor trenutno radi.

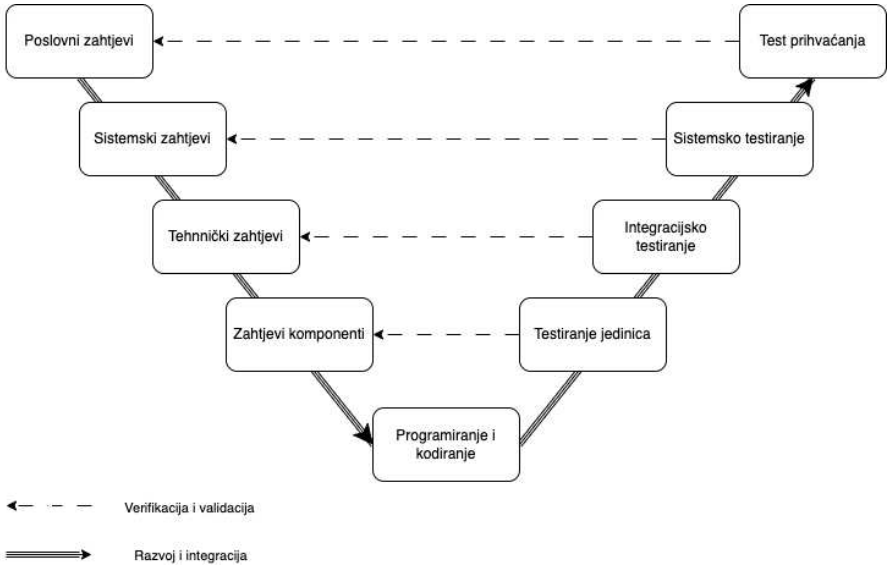
Poglavlje 5 - Zaključak je završni zaključak rada.

Kompletan Git repozitorij ovog rada javno je dostupan na `https://github.com/KristijanCetina/jsTesting`

Poglavlje 1

Testiranje programskog rješenje i osiguranje kvalitete

Testiranje softwareskog proizvoda se provodi u cilju osiguravanja kvalitete samog proizvoda. Svaka faza razvoja ima svoje specifične zahtjeve i načine testiranja, a njihov pregled je dan na slici 1.1. Pojedine funkcije u kodu se pokrivaju unit testovima koji se brinu da pojedine komponente rade ono što su namijenjene na najnižoj razini. To su testovi koji se u pravilu izvršavaju vrlo često prilikom pisanja koda kako bi se osiguralo da promjena neke funkcije ili komponente nije poremetila njen rezultat.



Slika 1.1: Proces testiranja u Scrum metodologiji rada

1.1 Proces testiranja

Svaki proces testiranja započinje izradom plana testiranja, unutar kojeg se precizno definira što će se testirati, na koji način, te koji su uvjeti za uspješno prolazak testa, poznati kao kriteriji prihvatljivosti (*engl. Acceptance criteria*). Ovaj plan služi kao temeljni dokument koji vodi cijeli proces testiranja, osiguravajući da su svi sudionici projekta usklađeni s ciljevima i metodologijama testiranja.

Tijekom razvoja nove funkcionalnosti u softverskom proizvodu često se provodi ručno testiranje. Ovo testiranje omogućava timu da prati napredak razvoja i osigura da funkcionalnost napreduje u skladu sa zadanim smjernicama. Ručno testiranje često uključuje provjeru da li se implementirane značajke ponašaju prema dogovorenim specifikacijama, bilo interno unutar tima, bilo s klijentom. Ovaj korak je ključan za rano otkrivanje i ispravljanje potencijalnih problema prije nego što postanu preveliki ili skupi za rješavanje.

Nakon završetka razvoja nove funkcionalnosti, slijedi završno funkcionalno testiranje. U ovoj fazi, funkcionalnost se testira kako bi se osiguralo da ispunjava sve zadane kriterije i da je spremna za implementaciju u produkciju. Pored toga, izrađuju se i automatizirani testovi, koji se zatim redovito izvršavaju u zadanom intervalu. Ovi automatizirani testovi imaju ključnu ulogu u osiguravanju da uvođenje novih funkcionalnosti ne izaziva nepredviđene pogreške u postojećim, ispravno funkcionirajućim dijelovima softvera. Ovaj oblik testiranja poznat je kao regresijsko testiranje.

Regresijsko testiranje služi kao zaštitni mehanizam, osiguravajući da svaka promjena u softveru ne uvodi nove greške ili probleme. To je posebno važno u složenim sustavima, gdje čak i mala promjena u kodu može imati dalekosežne posljedice. Automatizirani regresijski testovi omogućuju kontinuiranu provjeru stabilnosti softvera kroz cijeli njegov životni ciklus, smanjujući rizik od neočekivanih problema prilikom uvođenja novih značajki. Osim tehničkog testiranja, bitno je provoditi testiranje na temelju realnih scenarija koji se očekuju u stvarnoj upotrebi softvera. Ovi scenariji uključuju testove koji simuliraju stvarne korisničke interakcije kako bi se osiguralo da softver zadovoljava sve funkcionalne zahtjeve.

Također, važno je provoditi testove koji simuliraju neispravne ili ekstremne uvjete, poznate kao negativni scenariji, kako bi se potvrdilo da sustav pravilno rukuje pogrešnim unosima ili neuobičajenim situacijama. Testiranje u negativnim scenarijima je od ključne važnosti za osiguranje robusnosti softvera. Cilj ovih testova je potvrditi da sustav ispravno prepoznaje i odbacuje neispravne ili nevažeće unose, te da ne dolazi do neželjenih ponašanja ili grešaka unutar softvera. Time se osigurava da softver ne samo da ispravno funkcionira u idealnim uvjetima, već da je otporan i na nepravilne ulazne podatke ili nepredviđene situacije.

Proces testiranja softvera je iterativan i kontinuiran. Kako se softver razvija i nadograđuje, tako se i testovi moraju prilagođavati i proširivati kako bi obuhvatili nove funkcionalnosti i promjene. Kroz ovaj kontinuirani ciklus testiranja, osigurava se da softver ostaje pouzdan, funkcionalan i spreman za korisničke potrebe, čak i dok se dinamično mijenja i razvija. Uspješan proces testiranja stoga nije samo pitanje otkrivanja grešaka, već i strateški alat za kontinuirano poboljšanje kvalitete softverskog proizvoda.

1.2 Ciljevi testiranja

Ciljevi testiranja moraju zadovoljavati nekoliko kriterija, a to su:

- Specifičnost
- Mjerljivost
- Ostvarljiv
- Realističan
- Vremenski ograničen

Neki od ciljeva testiranja su:

Verifikacija i validacija

Cilj testiranja nije samo otkrivanje pogrešaka u kodu ili dizajnu, već i potvrđivanje da softver ispunjava svoju namjenu te radi onako kako je zamišljen. U tom kontekstu, verifikacija i validacija predstavljaju dvije ključne aktivnosti koje osiguravaju ispravnost i pouzdanost softverskog proizvoda.

Verifikacija je proces kojim se provjerava je li softver pravilno implementiran u skladu s definiranim zahtjevima i specifikacijama. Ova faza testiranja fokusira se na pitanje: "Da li gradimo proizvod na ispravan način?" Verifikacija se odnosi na sve aspekte razvoja softvera, uključujući pregled kodova, provjere dizajna, statičke analize, i jedinična testiranja. Kroz ove aktivnosti, tim osigurava da su svi zadani kriteriji i specifikacije ispunjeni prije nego što softver bude pušten u uporabu.

S druge strane, validacija se odnosi na potvrđivanje da softver radi ono što bi trebao raditi u stvarnim uvjetima korištenja, odnosno da ispunjava očekivanja krajnjih korisnika. Validacija odgovara na pitanje: "Jesmo li izgradili pravi proizvod?" U ovoj fazi, fokus je na funkcionalnim testovima, testovima performansi, korisničkom testiranju i integracijskim testovima. Validacija se

provodi kako bi se osiguralo da softver ispunjava svoje ciljeve u stvarnom svijetu te da zadovoljava potrebe korisnika i poslovnih zahtjeva.

Jedan od ključnih rezultata testiranja, bilo da se radi o verifikaciji ili validaciji, je izvještaj o testiranju (*test report*). Ovaj dokument pruža detaljan pregled provedenih testova, uključujući informacije o tome koji su testovi izvršeni, rezultati tih testova, te sve otkrivene greške ili nedostaci. Izvještaj o testiranju služi kao službeni zapis koji omogućava praćenje statusa softvera i donošenje informiranih odluka o daljnjem razvoju ili puštanju softvera u proizvodnju.

Izvještaj o testiranju također igra ključnu ulogu u komunikaciji između različitih dionika u projektu, uključujući razvojni tim, voditelje projekata, i klijente. Kroz jasno dokumentirane rezultate, izvještaj omogućava svim stranama da razumiju trenutačno stanje softvera, procijene rizike, te definiraju daljnje korake potrebne za osiguranje konačne kvalitete proizvoda.

Verifikacija i validacija su stoga suštinski komplementarni procesi u testiranju softvera. Dok verifikacija osigurava da je softver tehnički ispravan i usklađen sa specifikacijama, validacija osigurava da softver ispunjava svoju namjenu i zadovoljava potrebe korisnika. Kombinacija ovih procesa omogućava isporuku kvalitetnog, pouzdanog i funkcionalnog softverskog proizvoda, čime se minimiziraju rizici i osigurava dugoročni uspjeh projekta. Verifikacija i validacija kroz testiranje pružaju sveobuhvatnu provjeru ispravnosti softvera, omogućujući da se otkriju i isprave svi problemi prije nego što proizvod dospije do krajnjih korisnika. Ove aktivnosti nisu samo tehnička nužnost, već su i ključne za postizanje visoke razine kvalitete i zadovoljstva korisnika, što u konačnici doprinosi uspjehu softverskog proizvoda na tržištu.

Prioritiziranje pokrivenosti

U idealnom svijetu s neograničenim resursima, svaki dio izvornog koda i svaka funkcionalnost softvera bili bi pokriveni testovima, osiguravajući maksimalnu razinu pouzdanosti i kvalitete. Međutim, u stvarnosti, resursi poput vremena, radne snage i financija su ograničeni, što zahtijeva pažljivo planiranje i prioritiziranje pri odlučivanju o tome koje će funkcionalnosti i dijelovi koda biti pokriveni testovima. Ova odluka je ključna za održavanje učinkovitosti i usmjeravanje napora prema onim aspektima softvera koji su najkritičniji za njegovu ispravnost i korisničko iskustvo.

Prioritizacija testiranja obično se temelji na nekoliko ključnih kriterija. Prvi kriterij je kritičnost funkcionalnosti za osnovnu svrhu softvera. Funkcionalnosti koje su ključne za glavni rad aplikacije trebaju biti među prvima pokrivene testovima, jer svaki propust u tim dijelovima može imati ozbiljne posljedice po korisnike i poslovanje. Na primjer, u financijskim sustavima, algoritmi za

izračunavanje i prijenos novca zahtijevaju visoku razinu pouzdanosti, te je stoga njihovo testiranje prioritet.

Drugi važan kriterij je složenost koda. Kompleksniji dijelovi koda, koji sadrže više uvjetnih grana, petlji ili složenih logičkih izraza, podložniji su pogreškama i teže ih je ručno provjeriti. Stoga, upravo ti dijelovi trebaju biti prioritetno pokriveni testovima, kako bi se smanjio rizik od neočekivanih grešaka koje mogu biti teške za dijagnosticiranje i otklanjanje. Takvi složeni dijelovi koda često se nalaze "ispod površine", odnosno nisu vidljivi odmah prilikom otvaranja programa, te zahtijevaju pažljivo testiranje kako bi se osigurala njihova ispravnost.

Treći aspekt koji utječe na prioritizaciju je učestalost korištenja određenih funkcionalnosti. Funkcionalnosti koje korisnici najčešće koriste trebaju biti dobro testirane, jer svaka greška u tim dijelovima može značajno narušiti korisničko iskustvo i povjerenje u softver. Testiranje takvih funkcionalnosti omogućava rano otkrivanje i otklanjanje problema prije nego što oni negativno utječu na većinu korisnika.

Uz tehničke aspekte, potrebno je uzeti u obzir i vremenske i resursne ograničenosti. Beskonačno testiranje može dovesti do iscrpljivanja resursa, zbog čega je važno postaviti jasne granice i definirati što je potrebno testirati, a što ne. Ovdje dolazi do izražaja važnost balansiranja između pokrivenosti testovima i vremena potrebnog za njihovo izvršavanje. Testovi koji su preopsežni mogu oduzeti previše vremena, usporiti razvojni ciklus i povećati troškove bez proporcionalne koristi.

Konačno, treba uzeti u obzir i povijest problema i grešaka u softveru. Funkcionalnosti i dijelovi koda koji su u prošlosti bili problematični trebaju biti prioritetno pokriveni testovima kako bi se smanjio rizik od ponovnog pojavljivanja istih problema. Također, nova funkcionalnost ili značajke koje se tek uvode u softver zahtijevaju dodatnu pažnju, budući da još nisu dovoljno ispitane u stvarnim uvjetima. Pravilno određivanje prioriteta omogućava učinkovito korištenje resursa, osigurava visoku kvalitetu kritičnih dijelova softvera, i smanjuje rizik od neotkrivenih grešaka koje bi mogle imati značajne posljedice po korisnike i poslovanje.

Sljedivost

Sljedivost je ključni cilj testiranja softvera koji osigurava da se sve faze razvoja softverskog proizvoda mogu pratiti unatrag, od krajnjeg rezultata do početnih zahtjeva. Dokumentiranje testova, uključujući kada je, kako i što testirano, ima presudnu ulogu u omogućavanju ove sljedivosti. U slučaju pojave problema, sljedivost omogućuje timu da identificira specifične promjene koje su dovele do neželjenog ponašanja proizvoda, čime se olakšava proces ispravljanja grešaka i unaprjeđenja kvalitete.

Ovaj aspekt sljedivosti postaje posebno značajan u određenim kategorijama softvera, poput onih korištenih u financijskom sektoru, medicinskim uređajima, zrakoplovstvu, ili sigurnosno osjetljivim sustavima, gdje se može zahtijevati dodatna odgovornost samog proizvoda. U takvim kontekstima, testna dokumentacija ne služi samo kao interni alat za praćenje kvalitete, već može biti i pravno ili regulatorno obvezujući dokument. Organizacije često moraju dokazati da su sve komponente sustava testirane prema određenim standardima te da su promjene u kodu pažljivo praćene i evaluirane.

Sljedivost omogućuje vezu između različitih artefakata u procesu razvoja softvera, kao što su poslovni zahtjevi, tehničke specifikacije, kod, testni slučajevi, i rezultati testiranja. Na primjer, sljedivost može osigurati da su svi poslovni zahtjevi pokriveni odgovarajućim testnim slučajevima, te da su svi otkriveni problemi povezani s određenim dijelovima koda ili specifikacija. Ova povezanost je ključna za omogućavanje potpune vidljivosti nad razvojem proizvoda i osiguranje da se niti jedan aspekt proizvoda ne zanemari tijekom testiranja.

Kao dio sveobuhvatnog pristupa upravljanju kvalitetom, sljedivost također pomaže u upravljanju rizicima. Praćenjem svakog koraka u procesu razvoja i testiranja, timovi mogu identificirati potencijalne izvore rizika i brzo reagirati na promjene koje bi mogle utjecati na kvalitetu ili sigurnost proizvoda. Na primjer, ako promjena u jednom dijelu sustava izazove nepredviđene probleme, sljedivost omogućuje timu da brzo identificira korijenski uzrok i minimizira negativne posljedice.

Osim toga, sljedivost igra važnu ulogu u dugoročnoj održivosti softverskih sustava. U projektima s dugim životnim ciklusima, gdje može doći do promjena u timu ili arhitekturi sustava, detaljna dokumentacija i sposobnost praćenja svih testova unatrag postaju neophodni za održavanje kvalitete i razumijevanje povijesti razvoja proizvoda. To omogućuje novim članovima tima da brzo shvate kontekst i povijest određenih odluka, smanjujući vrijeme potrebno za prilagodbu i povećavajući ukupnu učinkovitost tima.

Sljedivost nije samo tehnička potreba, već strateški cilj testiranja softvera koji osigurava visoku razinu kontrole, transparentnosti i odgovornosti tijekom cijelog životnog ciklusa softverskog proizvoda. U složenim i reguliranim industrijama, sljedivost postaje temeljni aspekt osiguranja kvalitete, omogućujući organizacijama da održe visoke standarde i isporuče sigurne i pouzdane proizvode svojim korisnicima. [1]

1.3 Uloga testera u timu

Glavna uloga testera, kao i samog procesa testiranja, je pružiti dodatnu sigurnosnu mrežu, što čini zadnju kariku u lancu procesa testiranja i osiguranja

kvalitete proizvoda. Dok su programeri (*developeri, engl. developers*) zaduženi za pisanje unit testova, testeri su odgovorni za pisanje i izvršavanje funkcionalnih testova. Osim toga, testeri surađuju s vlasnicima proizvoda (*product ownerima, engl. product owners*) na definiranju kriterija za prihvaćanje, koji određuju kada je određeni dio softvera spreman za puštanje u produkciju [2].

Bitno je napomenuti kako su i developeri i testeri dio istog tima, te kao takvi dijele cilj - isporuku najkvalitetnijeg proizvoda moguće unutar zadanih parametara. Iako su njihove uloge različite, one se međusobno nadopunjuju. Dok developeri fokusiraju svoje napore na razvoj funkcionalnosti, testeri osiguravaju da te funkcionalnosti rade prema očekivanjima i zadanim specifikacijama.

Testeri igraju ključnu ulogu u ranom otkrivanju grešaka koje bi mogle imati značajan utjecaj na korisničko iskustvo i na stabilnost samog proizvoda. Njihova odgovornost nije samo identificirati greške, već i raditi na tome da se one isprave prije nego što proizvod dospije do krajnjih korisnika. Ova preventivna uloga testera može značajno smanjiti troškove i vrijeme potrebno za naknadne popravke, što dugoročno doprinosi uspjehu projekta.

Osim tehničkih aspekata, testeri također imaju važnu ulogu u komunikaciji unutar tima. Njihova sposobnost da jasno i precizno komuniciraju o otkrivenim problemima te da surađuju s developerima i product ownerima osigurava da se svi problemi riješe na vrijeme i na odgovarajući način. Tako, testeri ne samo da pomažu u održavanju tehničke kvalitete proizvoda, već također doprinose i općoj koheziji i učinkovitosti tima.

U kontekstu agilnih metoda razvoja softvera, uloga testera postaje još značajnija. Agilni timovi često rade u kratkim iteracijama (*sprintovima*), gdje je brzina isporuke novih funkcionalnosti ključna. U takvim uvjetima, testeri moraju biti uključeni od samog početka razvoja kako bi osigurali kontinuirano testiranje i osiguranje kvalitete kroz cijeli razvojni ciklus. Njihova proaktivnost i fleksibilnost u pristupu testiranju omogućuju timu da brzo reagira na promjene i isporučuje funkcionalan i stabilan softver na kraju svake iteracije.

Uloga testera u timu nije samo tehnička, već i strateška. Oni ne samo da osiguravaju da proizvod radi prema specifikacijama, već također pridonose stvaranju kulture kvalitete unutar tima. Kroz svoje aktivnosti, testeri omogućuju timu da isporuči visoko kvalitetan proizvod koji zadovoljava potrebe korisnika i održava reputaciju tvrtke na tržištu.

Poglavlje 2

Alat Playwright

Playwright je open-source biblioteka za automatizaciju testiranja web preglednika i web skrapanja koju je razvio Microsoft. Omogućuje automatizaciju testiranja web aplikacija na Chromiumu, Firefoxu i WebKit-u s jednim API-jem.

Playwright se ističe nizom prednosti koje ga čine pogodnim alatom za testiranje web aplikacija u akademskom i industrijskom okruženju. Prvo, jednostavnost korištenja predstavlja ključnu značajku Playwrighta. Njegovo intuitivno sučelje (API) podsjeća na dobro poznate alate poput jQuery-ja i Cypressa, čime olakšava učenje i implementaciju testova čak i za manje iskusne korisnike.

Nadalje, Playwright je visoko optimiziran za brzinu i pouzdanost, što je osobito važno u kontekstu testiranja produkcijskih okruženja. Njegova efikasnost omogućava smanjenje vremena izvršenja testova, uz istovremeno povećanje točnosti rezultata, što ga čini preferiranim alatom u dinamičnim razvojnim okruženjima.

Svestranost alata očituje se u njegovoj sposobnosti testiranja širokog spektra web aplikacija. Od jednostavnih web stranica do složenih arhitektura poput jednostrukih (SPA) i višestrukih (MPA) web aplikacija, Playwright uspješno odgovara na različite zahtjeve modernih web tehnologija.

Posebnu vrijednost predstavlja i njegova podrška za više programskih jezika. Playwright omogućava pisanje testova na popularnim jezicima kao što su JavaScript, TypeScript, Python, Java i C#, čime omogućava integraciju u različite razvojne ekosustave te zadovoljava potrebe širokog spektra korisnika, od početnika do iskusnih profesionalaca.

Playwright je alat široke primjene, a njegove funkcionalnosti omogućuju njegovu uporabu u različitim aspektima razvoja i testiranja web aplikacija. Jedna od ključnih primjena Playwrighta jest automatizacija testiranja korisničkog sučelja (UI). Korištenjem ovog alata moguće je kreirati automatizirane testove koji provjeravaju funkcionalnost web aplikacija, čime se osigurava konzistentnost

i pouzdanost softvera, te se smanjuje mogućnost pojave pogrešaka.

Osim toga, Playwright se može koristiti za web scraping, tehniku prikupljanja podataka s web stranica. Ova funkcionalnost omogućava programerima i istraživačima učinkovito dohvaćanje i analiziranje podataka iz različitih izvora na internetu, čime se olakšava proces prikupljanja informacija u raznim istraživačkim ili poslovnim kontekstima.

Nadalje, Playwright nudi mogućnost generiranja snimki zaslona (screenshotova) i videozapisa web stranica, što je korisno u svrhe dokumentacije ili analize. Ova značajka omogućava vizualnu inspekciju korisničkog sučelja i praćenje tijeka rada aplikacije, što može biti od presudne važnosti u razvojnim i testnim fazama softverskih projekata.

2.1 Opis i pregled paketa

Sistemski zahtjevi za pokretanje Playwrighta su: [1]

- Node.js 18 ili noviji
- Windows 10 ili noviji, Windows Server 2016 ili noviji ili Windows Subsystem for Linux (WSL),
- MacOS 12 Monterey ili noviji
- Debian 11, Debian 12, Ubuntu 20.04 ili Ubuntu 22.04, sa x86-64 ili arm64 arhitekturom.

Omogućava testiranje na Chromium, Firefox i WebKit engenima [2] koji se koriste u modernim web preglednicima.

Paket omogućava izvršavanje testova u UI načinu rada kao i u *headless* načinu rada prilikom kojeg se ne vide koraci kako se kreće po web stranici nego se na kraju testa dobije izvještaj o uspješnosti testiranja. To je vrlo korisno kada se koriste automatski načini objavljivanja koda koji onda može izvršiti testiranje prilikom svake promjene koda.

2.2 Instalacija

Najjednostavniji način za instalaciju Playwright paketa je putem npm alata koristeći naredbu

[1] `https://playwright.dev/docs/intro#system-requirements`
[2] `https://www.npmjs.com/package/playwright#documentation--api-reference`

```
npm init playwright@latest
```

te će to instalirati paket i pokrenuti postupak inicijalizacije paketa. Osim `npm`, može se koristiti i `yarn` ili `pnpm`, ovisno o osobnim preferencijama. Tokom inicijalizacije može se birati nekoliko postavki:

- Odabrati TypeScript ili JavaScript (standardno je TypeScript)
- Odabrati ime direktorija koji će sadržavati testove (standardno je 'test' ili 'e2e' - end to end, ako 'test' već postoji)
- Dodati GitHub Action workflow za automatsko izvršavanje testova prilikom objave izvornog koda na GitHub servisu
- Instalirati potrebne preglednike koji će se koristiti za testiranje

Playwright će nakon toga kreirati potrebne direktorije i datoteke za konfiguraciju kao i primjer jednog testa za lakši početak

```
playwright.config.ts
package.json
package-lock.json
tests/
  example.spec.ts
tests-examples/
  demo-todo-app.spec.ts
```

Primijetimo kako je uvriježena norma da se datoteke koje sadrže testove imaju `.spec` ispred oznake tipa datoteke uz zadržavanje istog imena. Čak ih i razni editori koda označavaju s drugim ikonama kako bi bili vizualno lakše raspoznatljivi od datoteka koje sadrže izvorni komponenti kao što je vidljivo na slici 2.1.

Slika 2.1: Izgled ikona s izvornim kodom i testom za komponentu

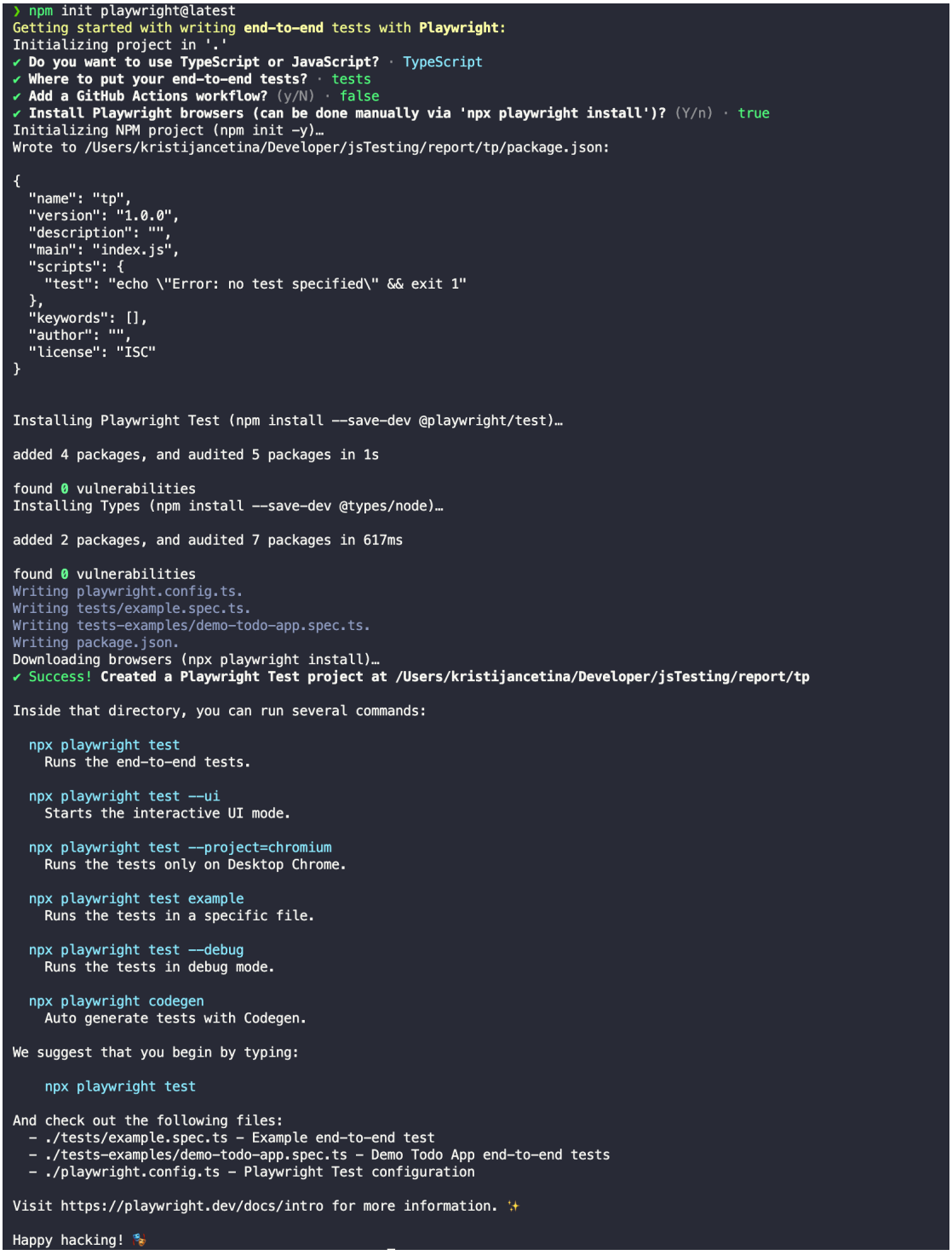
Ako se inicijalizacija vrši unutar već postojećeg projekta, što je najčešće i slučaj, konfiguracija zavisnih paketa će biti dodana u postojeću `package.json` datoteku.

`playwright.config.ts` datoteka sadrži konfiguracije testova kao npr

- koji se preglednik koristi,
- koja je veličina prozora preglednika,
- koji se mobilni uređaj koristi u slučaju testiranja na mobilnim preglednicima,
- standardno očekivano vrijeme ispunjenja testa (timeout)

te mnogi drugi preddefinirane i prilagođene opcije konfiguracije.

Na slici 2.2 vidimo kako izgleda uspješna instalacija i inicijalizacija Playwright paketa.



Slika 2.2: Ekran nakon uspješne instalacije i inicijalizacije Playwright paketa

2.3 Lokatori

Lokatori su ključni element u radu s pronalaženjem elemenata nad kojima želimo izvršiti neku radnju. Ukratko, lokatori predstavljaju način pronalaženja elemenata na stranici u bilo kojem trenutku. Kroz različite metode, Playwright omogućuje precizno i učinkovito lociranje elemenata, čime se omogućava pouzdanost testova.

Ovo su preporučeni ugrađeni lokatori:

- `page.getByRole()` za lociranje prema eksplicitnim i implicitnim atributima pristupačnosti.
- `page.getByText()` za lociranje prema tekstualnom sadržaju.
- `page.getByLabel()` za lociranje kontrola obrasca prema tekstu pridružene oznake.
- `page.getByPlaceholder()` za lociranje unosa prema rezerviranom mjestu.
- `page.getByAltText()` za lociranje elemenata, obično slika, prema njihovom alternativnom tekstu.
- `page.getByTitle()` za lociranje elemenata prema atributu naslova.
- `page.getByTestId()` za lociranje elemenata na temelju atributa data-testid (moguće je konfigurirati i druge atribute).

Primjer korištenja:

```
await page.getByLabel('User Name').fill('John');
await page.getByLabel('Password').fill('secret-password');
await page.getByRole('button', { name: 'Sign in' }).click();
await expect(page.getByText('Welcome, John!')).toBeVisible();
```

Playwright dolazi s više ugrađenih lokatora. Kako bi testovi bili otporniji, preporučuje se prioritiziranje atributa usmjerenih prema korisniku i eksplicitnih ugovora, poput `page.getByRole()`.

Na primjer, za sljedeću DOM strukturu:

```
<button>Sign in</button>
```

Element se može locirati prema njegovoj ulozi `button` s nazivom "Sign in":

```
await page.getByRole('button', { name: 'Sign in' }).click();
```

Lociranje prema ulozi

Lokator `page.getByRole()` odražava kako korisnici i asistivne tehnologije percipiraju stranicu, primjerice je li neki element gumb ili potvrdni okvir. Kada se locira prema ulozi, obično treba navesti i dostupno ime kako bi lokator točno odredio element.

Primjer DOM strukture:

```
<h3>Sign up</h3>
<label>
  <input type="checkbox" /> Subscribe
</label>
<br/>
<button>Submit</button>
```

Elementi se mogu locirati prema implicitnim ulogama:

```
await expect(page.getByRole('heading', { name: 'Sign up' })).toBeVisible();
await page.getByRole('checkbox', { name: 'Subscribe' }).check();
await page.getByRole('button', { name: /submit/i }).click();
```

Lociranje prema oznaci

Većina kontrola obrasca ima pridružene oznake koje se mogu koristiti za interakciju s obrascem. Ova metoda se koristi kada lociramo polja obrasca:

Primjer DOM strukture:

```
<label>Password <input type="password" /></label>
```

Unos se može ispuniti nakon lociranja prema tekstu oznake:

```
await page.getByLabel('Password').fill('secret');
```

Lociranje prema rezerviranom mjestu (placeholder)

Unosi mogu imati placeholder atribut koji korisnicima sugerira koji bi se vrijednosti trebali unijeti. Ova metoda se koristi kada lociramo elemente obrasca koji nemaju oznake, ali imaju tekstove rezerviranog mjesta:

Primjer DOM strukture:

```
<input type="email" placeholder="name@example.com" />
```

Unos se može ispuniti nakon lociranja prema tekstu rezerviranog mjesta:

```
await page.getByPlaceholder('name@example.com')
  .fill('playwright@microsoft.com');
```

Lociranje prema tekstu

Lokatori prema tekstu omogućuju pronalaženje elementa prema tekstu koji sadrži. Možete koristiti podstring, točan niz ili regularni izraz.

Primjer DOM strukture:

```
<span>Welcome, John</span>
```

Element se može locirati prema tekstu koji sadrži:

```
await expect(page.getByText('Welcome, John')).toBeVisible();
```

Za točno podudaranje:

```
await expect(page.getByText('Welcome, John',
  { exact: true })).toBeVisible();
```

Za podudaranje s regularnim izrazom:

```
await expect(page.getByText(/welcome, [A-Za-z]+$/i)).toBeVisible();
```

Lociranje prema alternativnom tekstu

Sve slike trebale bi imati atribut alt koji opisuje sliku. Ova metoda se koristi za lociranje slika prema alternativnom tekstu:

Primjer DOM strukture:

```
<img alt="playwright logo" src="/img/playwright-
logo.svg" width="100" />
```

Slika se može kliknuti nakon lociranja prema alternativnom tekstu:

```
await page.getByAltText('playwright logo').click();
```

Lociranje prema naslovu

Elementi se mogu locirati prema odgovarajućem atributu naslova koristeći **page.getByTitle()**.

Primjer DOM strukture:

```
<span title='Issues count'>25 issues</span>
```

Broj problema može se provjeriti nakon lociranja prema tekstu naslova:

```
await expect(page.getByTitle('Issues count')).toHaveText('25 issues');
```

Lociranje prema test id-u

Testiranje prema test id-evima je najotporniji način testiranja jer će test proći čak i ako se tekst ili uloga atributa promijene. No, testiranje prema test id-evima nije usmjereno prema korisniku. Ako je vrijednost uloge ili teksta važna, valja razmotriti korištenje lokatora usmjerenih prema korisniku poput `page.getByRole()` ili `page.getByText()`.

Primjer DOM strukture:

```
<button data-testid="directions">Itinéraire</button>
```

Element se može locirati prema njegovom test id-u:

```
await page.getByTestId('directions').click();
```

Podešavanje prilagođenog atributa test id-a

Prema zadanim postavkama, `page.getByTestId()` će locirati elemente na temelju atributa `data-testid`, ili može biti drukčije konfigurirati u testnoj okolini ili pak pozivom `selectors.setTestIdAttribute()`.

Primjer konfiguracije:

```
import { defineConfig } from '@playwright/test';

export default defineConfig({
  use: {
    testIdAttribute: 'data-pw'
  }
});
```

Te se onda može koristiti dati atribut za lociranje elemenata:

```
\begin{verbatim}
<button data-pw="directions">Itinéraire</button>
```

Element se može locirati kao i obično:

```
await page.getByTestId('directions').click();
```

Lociranje prema CSS-u ili XPath-u

Ako je apsolutno nužno koristiti CSS ili XPath lokatore, može se koristiti `page.locator()` za kreiranje lokatora koji uzima selektor opisujući kako pronaći element na stranici. Playwright podržava CSS i XPath selektore te ih automatski prepoznaje ako izostavite prefiks `css=` ili `xpath=`.

Primjeri korištenja:

```
await page.locator('css=button').click();
await page.locator('xpath=//button').click();

await page.locator('button').click();
await page.locator('//button').click();
```

CSS i XPath selektori mogu biti vezani za strukturu DOM-a ili implementaciju, što znači da se mogu prekinuti kada se struktura DOM-a promijeni. Dugi CSS ili XPath lanci su primjer loše prakse koja vodi do nestabilnih testova:

```
await page.locator('#tsf > div:nth-
child(2) > div.A8SBwf > div.RNNXgb > div > div.a4bIc > input').click();
await page.locator('//*[@id="tsf"]/div[2]/div[1]/div[1]/div/div[2]/input')
  .click();
```

Preporuka je izbjegavati CSS i XPath lokatore kad god je to moguće i umjesto toga koristiti lokatore koji su bliži percepciji korisnika stranice, poput `page.getByRole()` ili definirati eksplicitne testne atribute koristeći test id-ove.

Lociranje u Shadow DOM-u

Svi lokatori u Playwrightu prema zadanim postavkama rade s elementima u Shadow DOM-u, s iznimkom:

- Lociranje prema XPath-u ne prolazi kroz shadow root.
- Shadow root u zatvorenom načinu rada nije podržan.

Primjer s prilagođenom web komponentom:

```
<x-details role="button" aria-expanded="true" aria-controls="inner-
details">
  <div>Title</div>
  #shadow-root
  <div id="inner-details">
    <button>Submit</button>
  </div>
</x-details>
```

Elementi u Shadow DOM-u mogu se locirati prema njihovim ulogama:

```
await page.getByRole('button', { name: 'Submit' }).click();
```

Elementi u zatvorenom Shadow rootu nisu dostupni lokatorima:

```
#shadow-root(mode=closed)
  <button>Submit</button>
```

Metode lokatora

Lokator je centar svih akcija i asertacija u Playwrightu:

- `locator.click()` klikne na element.
- `locator.fill(value)` ispunjava unos.
- `locator.count()` vraća broj elemenata.

Ponašanje lokatora koristi se za automatsko čekanje elemenata. Playwrightova `expect(locator).toBeVisible()` metoda čeka dok element ne postane vidljiv prije nego što nastavi. Lokatori su također robusniji prema nestalnosti DOM-a, čekajući automatski da elementi postanu dostupni i ponovno pokušavajući u slučaju grešaka u mreži ili spore dinamike stranice.

Primjeri:

```
await page.locator('text=Submit').click();
await page.locator('input[type="password"]').fill('password');
await expect(page.locator('button')).toHaveCount(3);
```

Više metoda za rad s lokatorima:

- `locator.first()` locira prvi element.
- `locator.last()` locira zadnji element.
- `locator.nth(index)` locira n-ti element.

Primjeri:

```
await page.locator('button').first().click();
await page.locator('button').last().click();
await page.locator('button').nth(2).click();
```

Testiranje dinamike

Lokatori u Playwrightu automatski čekaju da element postane dostupan i ponovo pokušavaju ako se element mijenja, što ih čini otpornima na dinamiku stranica. Ovdje su neki primjeri kako se to koristi:

Primjeri:

```
// Čeka da gumb postane vidljiv i klikne ga
await page.locator('button').click();

// Ispunjava unos kada postane dostupan
await page.locator('input[type="password"]').fill('password');

// Čeka da gumb postane nevidljiv
await expect(page.locator('button')).not.toBeVisible();
```

Playwrightova `expect` metoda može se koristiti za različite asertacije, što dodatno povećava fleksibilnost i robusnost testova.

Primjeri naprednog lociranja

Lokatori mogu biti kombinirani za složenije scenarije:

`Chaining locators`: Kombinacija više lokatora za precizno ciljanje elementa.

```
await page.locator('form').locator('input[name="username"]').fill('john_doe');
```

`Filtering locators`: Korištenje filtera kao što su `has` i `hasText` za daljnje sužavanje rezultata.

```
await page.locator('div').locator('text=Submit').click();
await page.locator('div', { hasText: 'Important' }).click();
```

`Working with child locators`: Ciljanje potomaka unutar određenog elementa.

```
await page.locator('ul').locator('li >> text=Item 2').click();
```

`Relative locators`: Korištenje relativnih lokatora za ciljanje elemenata u odnosu na druge elemente.

```
await page.locator('text=Name').locator('..')
  .locator('input').fill('Jane Doe');
```

Složeniji scenariji lociranja

Za složenije scenarije, Playwright nudi dodatne mogućnosti: Korištenje `nth()` se kada postoji potreba za ciljanjem određenog pojavljivanja elementa među više istovrsnih elemenata.

```
await page.locator('button').nth(2).click(); // Klik na treći gumb u nizu
```

Kombinacija više lokatora za precizno ciljanje.

```
await page.locator('section').locator('button',
  { hasText: 'Submit' }).click();
```

Rad s elementima unutar Shadow DOM-a.

```
await page.locator('my-
component').locator('button', { hasText: 'Submit' }).click();
```

Praktične primjene

Evo nekoliko praktičnih primjera kako se lokatori koriste u stvarnim scenarijima testiranja:

Automatsko popunjavanje obrazaca:

```
await page.getByLabel('Username').fill('test_user');
await page.getByLabel('Password').fill('password123');
await page.getByRole('button', { name: 'Login' }).click();
```

Validacija sadržaja

:

```
await expect(page.getByText('Welcome, test_user!')).toBeVisible();
```

Interakcija s pop-up prozorima

:

```
await page.getByRole('button', { name: 'Open modal' }).click();
await expect(page.getByRole('dialog')).toBeVisible();
await page.getByRole('button', { name: 'Close' }).click();
```

Navigacija kroz elemente

:

```
await page.getByRole('link', { name: 'Next' }).click();
await expect(page).toHaveURL('/next-page');
```

2.4 Generiranje testova

Microsoft Playwright pruža moćan alat za automatizaciju testiranja web-aplikacija, omogućujući automatsko generiranje testova putem snimanja korisničkih interakcija, poput klikanja mišem i unosa teksta. Ova mogućnost čini Playwright izuzetno pristupačnim alatom za početnike, omogućujući im da brzo započnu s izradom automatskih testova bez potrebe za opsežnim znanjem programiranja. Snimanje korisničkih radnji i njihovo automatsko prevođenje u izvršni kod ne samo da ubrzava proces testiranja, već i smanjuje potrebu za ručnim pisanjem ponavljajućih dijelova koda, što je često zamorno i podložno pogreškama.

Generirani kod od strane Playwrighta često je dovoljno robustan za upotrebu u jednostavnijim scenarijima, kao što su osnovne funkcionalnosti web-aplikacija. U ovim slučajevima, automatski generirani testovi mogu poslužiti kao osnovna testna pokrivenost koja omogućuje brzo otkrivanje regresijskih problema ili osnovnih grešaka u aplikaciji. Na primjer, testovi za provjeru rada forme za prijavu ili osnovnih navigacijskih funkcionalnosti mogu biti brzo sastavljeni pomoću ove metode, čime se osigurava da ključne funkcionalnosti aplikacije ispravno funkcioniraju bez potrebe za detaljnim ručnim pisanjem testova.

Jedna od ključnih prednosti ovog pristupa je eliminacija potrebe za pisanjem prilagođenih funkcija koje su često potrebne u dugotrajnim projektima gdje se očekuje stalno održavanje i nadogradnja. Ova prednost postaje očita u situacijama kada je potrebno brzo sastaviti testove za funkcionalnosti koje možda neće biti dugoročno održavane ili u slučajevima kada je cilj samo brza provjera osnovne funkcionalnosti. U takvim okolnostima, automatski generirani testovi mogu značajno smanjiti vrijeme potrebno za izradu i održavanje testne suite, omogućujući razvojnim timovima da se usmjere na složenije aspekte razvoja i testiranja.

Međutim, unatoč praktičnosti i brzini koju pruža automatsko generiranje testova, važno je biti svjestan potencijalnih nedostataka ovog pristupa. Jedan od glavnih rizika je prekomjerno oslanjanje na automatski generirane testove, što može dovesti do problema u kasnijim fazama razvoja. Naime, iako su automatski generirani testovi korisni za brzo sastavljanje osnovne pokrivenosti, često im nedostaje fleksibilnost i prilagodljivost koja je potrebna za složenije testne

scenarije. Također, generirani kod može sadržavati nepotrebne ili redundantne korake koji mogu otežati održavanje i povećati složenost testne suite tijekom vremena.

Stoga, iako automatsko generiranje testova s pomoću Playwrighta može značajno ubrzati proces testiranja, potrebno je pažljivo balansirati između brzine i kvalitete. Razvojni timovi bi trebali koristiti ovu mogućnost s oprezom, uzimajući u obzir dugoročne ciljeve projekta i složenost aplikacije. U slučajevima kada se očekuje dugoročno održavanje i nadogradnja, može biti korisno dodatno rafinirati i optimizirati generirane testove, ili čak napisati prilagođene testove kako bi se osigurala maksimalna pouzdanost i održivost testnog paketa.

2.5 Pokretanje alata za generiranje testova

Alat za generiranje testova u Microsoft Playwrightu pokreće se putem naredbe `codegen`, koja kao argument može primiti URL web stranice za koju se želi generirati testove. Iako unos URL-a nije obavezan prilikom pokretanja alata, on može biti dodan kasnije izravno u prozoru preglednika koji se otvori. Ova fleksibilnost omogućuje korisnicima da započnu generiranje testova na različite načine, ovisno o specifičnim potrebama projekta.

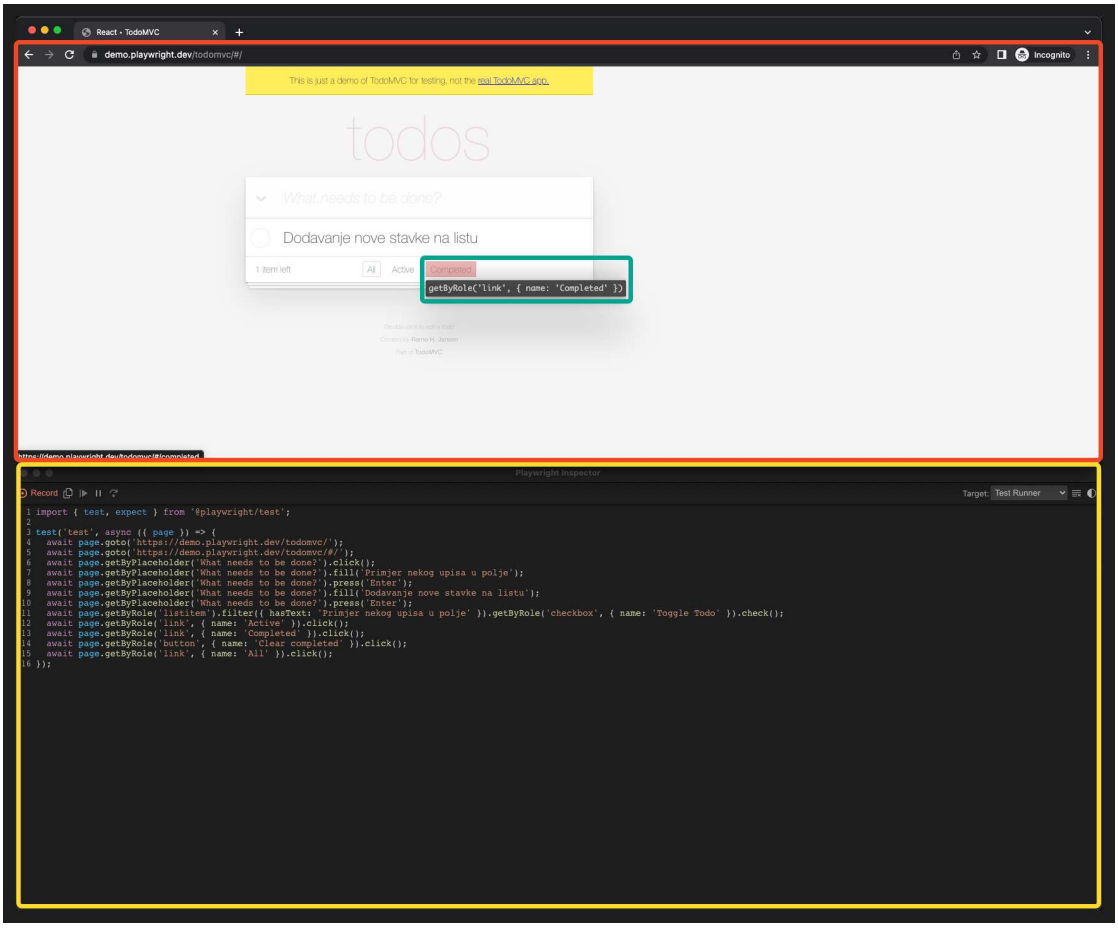
Kao primjer, koji je preuzet iz službene dokumentacije [3], razmotrit ćemo naredbu:

```
npx playwright codegen demo.playwright.dev/todomvc
```

Ova naredba pokreće alat za generiranje testova i otvara zadanu web stranicu unutar sučelja za generiranje. Na slici 2.3 prikazan je izgled sučelja za generiranje testova koje se sastoji od nekoliko dijelova:

- Prozor preglednika - označen crvenim okvirom, unutar kojeg se prikazuje i izvršava aplikacija. Ovaj prozor omogućava korisniku interakciju s web stranicom kao što bi to učinio u bilo kojem drugom pregledniku.
- Prozor za generirani kod - označen žutim okvirom, prikazuje kod koji Playwright automatski generira na temelju korisničkih interakcija s aplikacijom u prozoru preglednika. Ovaj kod se može odmah koristiti ili dodatno prilagoditi prema potrebama.
- Lokator - označen zelenim okvirom, prikazuje se kada korisnik pomakne miš preko elementa unutar prozora preglednika. Lokator je ključan jer definira elemente na stranici s kojima Playwright testovi trebaju interagirati, kao što su tipke, polja za unos, ili linkovi.

[3] `https://playwright.dev/docs/codegen-intro#running-codegen`



Slika 2.3: Izgled sučelja za generiranje testova

Ova struktura sučelja omogućava korisniku da brzo i jednostavno generira testove s minimalnim unosom koda, fokusirajući se na interakcije s korisničkim sučeljem. Generirani kod je odmah vidljiv i može se modificirati u realnom vremenu, što olakšava proces prilagodbe testova specifičnim potrebama aplikacije.

Jedan od glavnih benefita korištenja ovog alata jest njegova intuitivnost i pristupačnost, čak i za korisnike s minimalnim iskustvom u pisanju automatiziranih testova. Sučelje omogućava neposrednu povratnu informaciju o generiranom kodu, što olakšava učenje i razumijevanje načina na koji Playwright funkcionira. Ovo je posebno korisno za timove koji tek započinju s implementacijom automatizacije testiranja i žele brzo izraditi osnovnu testnu suite.

Osim što omogućava brzo generiranje testova, `codegen` naredba također

podržava fleksibilnost prilikom testiranja različitih scenarija. Korisnici mogu mijenjati URL-ove unutar sučelja, ponovno pokretati testove, te dodavati ili uklanjati dijelove koda prema potrebi, sve unutar istog alata. Ova funkcionalnost čini Playwright izuzetno moćnim alatom za automatizaciju testiranja u dinamičnim okruženjima gdje se zahtjevi i aplikacije često mijenjaju.

Pokretanje alata za generiranje testova s pomoću Microsoft Playwrighta predstavlja jednostavan, ali vrlo učinkovit način za kreiranje automatiziranih testova. Kombinacija intuitivnog korisničkog sučelja i mogućnosti trenutačnog generiranja koda omogućava razvojnim timovima da brzo postignu visoku razinu pokrivenosti testovima, čak i u ranim fazama razvoja, što može značajno poboljšati ukupnu kvalitetu softverskog proizvoda.

2.6 Kontinuirana integracija i testiranje

Uvođenje kontinuirane integracije (CI) i kontinuirane dostave (CD) predstavlja jedan od ključnih elemenata suvremenog razvoja softvera, omogućujući timovima bržu, pouzdaniju i konzistentniju isporuku softverskih rješenja. U nastavku će biti opisano kako i zašto napraviti jednostavan CI/CD pipeline-a za automatizirano testiranje s pomoću Playwrighta.

CI/CD pipeline je automatizirani niz koraka koji omogućuje brzu i pouzdanu isporuku aplikacija. CI/CD pipeline se sastoji od niza automatiziranih procesa koji uključuju izgradnju (build), testiranje i distribuciju (deployment) aplikacije. Kroz automatizaciju ovih koraka, CI/CD pipeline pomaže u smanjenju rizika od grešaka, ubrzava proces isporuke te osigurava dosljednost i kvalitetu softverskih rješenja.

Integracija Playwright-a u CI/CD pipeline omogućava automatizirano izvođenje testova pri svakoj promjeni koda, čime se osigurava da sve funkcionalnosti aplikacije rade ispravno prije nego što se promjene implementiraju u produkciju.

Definicija CI/CD Pipeline-a: Nakon konfiguracije repozitorija, potrebno je definirati CI/CD pipeline. To se obično radi s pomoću YAML datoteka koje sadrže instrukcije za izgradnju, testiranje i distribuciju aplikacije. U nastavku je primjer osnovne konfiguracije za GitHub Actions, a može biti generirana automatski prilikom inicijalizacije projekta kao što je prikazana na slici 2.4:

```
name: Playwright Tests
on:
  push:
    branches: [ main, master ]
  pull_request:
```

Slika 2.4: Upit za kreiranje GitHub Action workflowa prilikom inicijalizacije projekta

```
    branches: [ main, master ]

jobs:
  test:
    timeout-minutes: 60
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: lts/*

    - name: Install dependencies
      run: npm ci

    - name: Install Playwright Browsers
      run: npx playwright install --with-deps

    - name: Run Playwright tests
      run: npx playwright test

    - uses: actions/upload-artifact@v4
      if: always()
      with:
        name: playwright-report
        path: playwright-report/
        retention-days: 30
```

Ova konfiguracija definira akcije koje će se pokrenuti pri svakom push-u ili pull request-u na glavnu granu repozitorija. Akcije uključuju preuzimanje koda, postavljanje Node.js okruženja, instalaciju zavisnosti i pokretanje Playwright testova.

Nakon definiranja CI/CD pipeline-a, svaki push ili pull request pokreće automatiziranu izgradnju i testiranje aplikacije. Playwright testovi se izvršavaju unutar pipeline-a, čime se osigurava da sve promjene koda ne narušavaju postojeću funkcionalnost.

Posljednji korak CI/CD pipeline-a je distribucija aplikacije. Ako svi testovi prođu uspješno, aplikacija se automatski distribuira na produkcijsko okruženje. Ovaj korak može uključivati različite metode distribucije, kao što su deployment na cloud platforme, generiranje Docker datoteke (image) ili distribucija na serverske klastere.

Prednosti CI/CD Pipelinea za Playwright

Implementacija CI/CD pipelinea za Playwright donosi brojne prednosti:

- Automatizacija: CI/CD pipeline automatizira proces testiranja i distribucije, čime se smanjuje potreba za ručnim intervencijama i povećava produktivnost tima.
- Konzistentnost: Automatizirano testiranje osigurava dosljednost i kvalitetu aplikacije, jer se testovi izvršavaju pri svakoj promjeni koda.
- Brža isporuka: CI/CD pipeline ubrzava proces isporuke, omogućujući brže uvođenje novih funkcionalnosti i popravaka grešaka.
- Rano otkrivanje problema: Redovito izvršavanje testova omogućava rano otkrivanje problema, čime se smanjuje rizik od grešaka u produkcijskom okruženju.

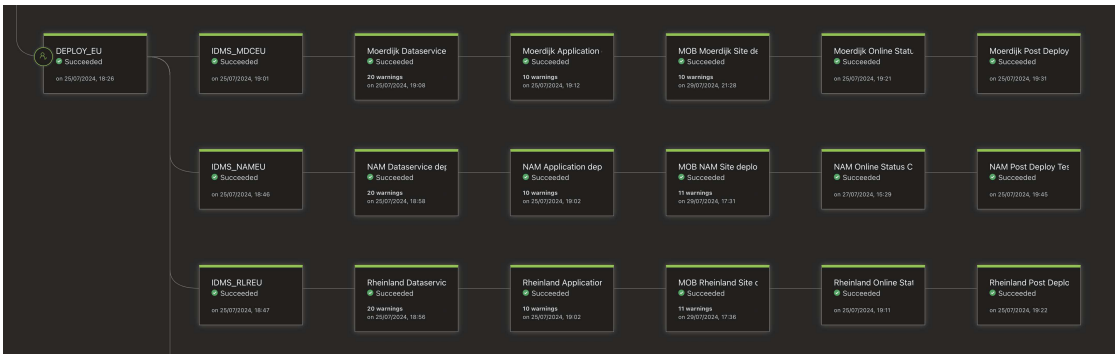
Implementacija kontinuiranog testiranja unutar Azure DevOps okruženja

Kontinuirano testiranje predstavlja ključnu komponentu u modernim DevOps procesima, omogućavajući timovima da osiguraju kvalitetu softvera kroz automatizirane testove koji se izvršavaju tijekom cijelog ciklusa razvoja. Unutar Azure DevOps okruženja, implementacija kontinuiranog testiranja se može efikasno postići putem konfiguracije u YAML formatu, koja omogućuje definiranje koraka za automatsko pokretanje testova unutar CI/CD pipelinea.

Jedan od primjera konfiguracije prikazan je u nastavku:

```
  steps:
- script: 'npx playwright test tests/postDeployPipelineQA.test.ts -
-workers 1'
  workingDirectory: '$(System.DefaultWorkingDirectory)/Playwright_Source/
    postDeployTests'
  displayName: 'Run The Test'
```

Grafički prikaz konfiguracije je prikazan na slici 2.8 Ova konfiguracija definira korak unutar Azure Pipelinea koji pokreće Playwright testove nakon što se softver implementira. Script naredba koristi `npx` za izvršavanje Playwright testova, a `workingDirectory` definira direktorij u kojem se testovi nalaze. Na ovaj način osigurava se da se testovi pokreću u točno određenom kontekstu, čime se smanjuje rizik od neuspjeha testova zbog neodgovarajućih radnih okruženja.

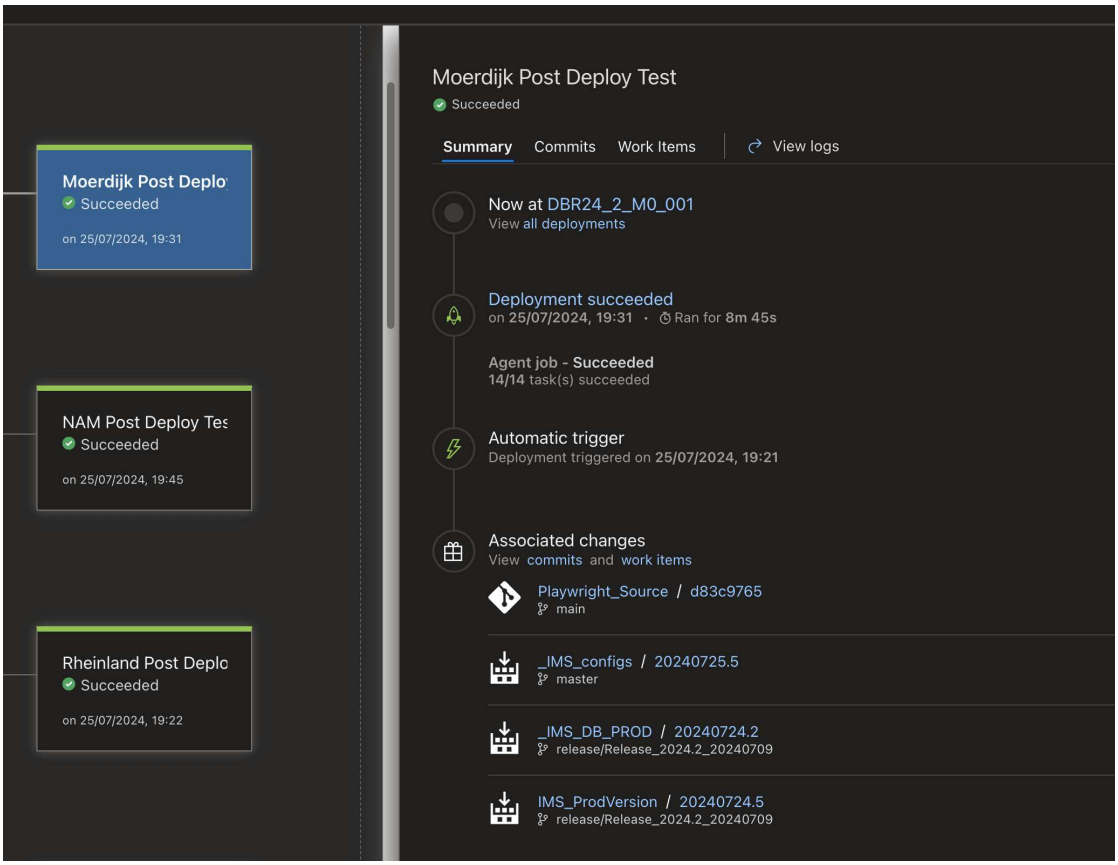


Slika 2.5: Izgled Azure Pipelinea sa zasebnim koracima

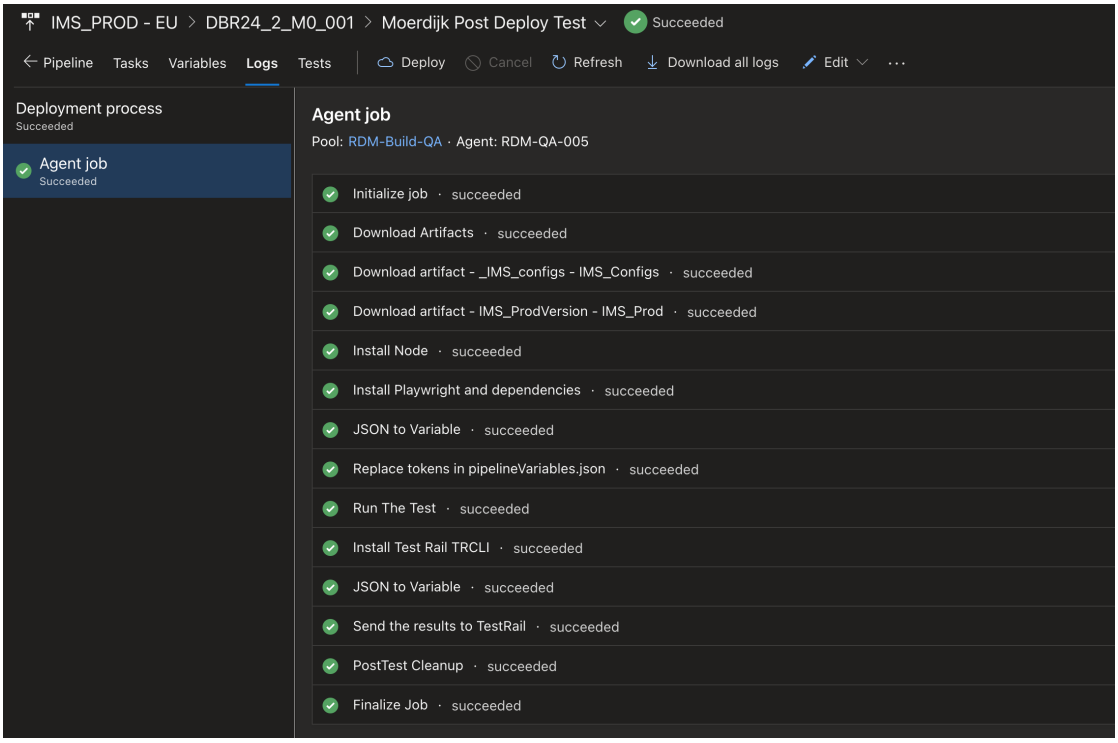
2.5 daje pregled cjelokupnog pipelinea.

Slike 2.6, 2.7 i 2.8 prikazuju detalje specifičnih koraka unutar pipelinea, s posebnim fokusom na korake za testiranje. Ovi prikazi omogućuju vizualni uvid u konfiguraciju i strukturu pipelinea, što olakšava razumijevanje i daljnje prilagodbe procesa testiranja.

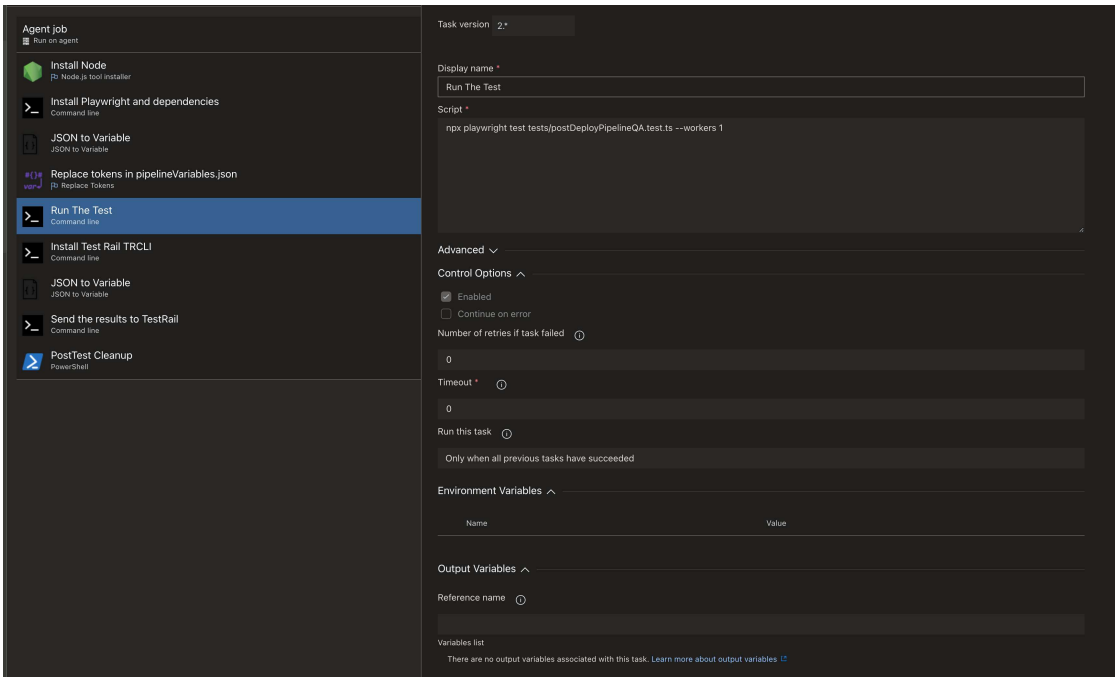
Implementacija ovakvog pristupa omogućuje dosljedno provođenje testiranja tijekom cijelog razvojnog ciklusa, što pomaže u ranom otkrivanju problema i održavanju visoke razine kvalitete softverskog proizvoda. Azure DevOps pruža fleksibilnost i moćne alate za integraciju automatiziranih testova, čime se podržava kontinuirana isporuka i brza reakcija na promjene unutar razvojnog tima.



Slika 2.6: Detalji koraka za testiranje - posljednji u nizu unutar pipelinea



Slika 2.7: Detalji koraka za testiranje - akcije koraka za testiranje



Slika 2.8: Detalji konfiguracije samog testa

Poglavlje 3

Testiranje nakon nadogradnje na novu verziju

Jedna od ključnih aktivnosti svakog razvojnog tima je redovita nadogradnja softvera na novu verziju. U našem poduzeću, uobičajena praksa je implementacija nove verzije softvera jednom mjesečno za glavnog klijenta. S obzirom na to da klijent ima postrojenja na šest kontinenata, uključujući dodatna postrojenja smještena u međunarodnim vodama svih svjetskih oceana, proces nadogradnje podijeljen je po regijama.

Te regije su:

- Americas (Sjeverna i Južna Amerika),
- EMEA (Europa, Bliski istok i Afrika),
- APAC (Azija i Pacifik).

Svaka od ovih regija obuhvaća više od 15 postrojenja koja koriste zasebne poslužitelje, što znači da se nadogradnja mora provesti na više od 15 lokacija unutar svake regije. Već iz ove strukture vidljivo je da proces nadogradnje nosi sa sobom značajan rizik od potencijalnih problema i zastoja koji se mogu pojaviti tijekom implementacije. Stoga je od ključne važnosti osigurati da proces nadogradnje bude što optimalniji, uz minimalne prekide u radu i bez nepotrebnih zastoja.

3.1 Postojeći način testiranja

rije uvođenja automatskih testova, standardna procedura uključivala je detaljno testiranje AM regije koje je provodilo 3 do 4 testera, odnosno inženjera za kontrolu kvalitete (QA). Svaki tester bi provjeravao 3 do 4 postrojenja (site). U prosjeku, za provjeru jednog postrojenja bilo je potrebno između 20 i 30 minuta kako bi se

osiguralo da su sve promjene ispravno primijenjene te da nisu izazvale neželjene nuspojave ili prestanak rada postojećih funkcionalnosti.

S obzirom na velik broj funkcionalnosti i ograničeno vrijeme, bilo je nužno selektivno pristupiti odabiru onih funkcionalnosti koje će se testirati, pri čemu je ključni kriterij bila kritičnost pojedine funkcionalnosti. Cijeli proces trajao je između 1,5 i 2 sata po članu testnog tima, dok je ukupni trošak procijenjen na 6 do 7 radnih čovjek-sati. Jednostavnom množenjem s cijenom rada dolazi se do ukupnog troška testiranja nakon nadogradnje.

Nakon što bi se verificiralo da je novi paket programa ispravan, te u slučaju da klijenti nisu prijavili ozbiljne probleme koji bi onemogućili daljnji rad (*showstopper*), pristupilo bi se nadogradnji sljedeće regije, Azije i Pacifika.

U tom bi se slučaju postupak ponovio s mogućim izmjenama, poput smanjenja broja članova testnog tima za jednog, kako bi se izbjegli prekovremeni sati. Ova redukcija testiranja značila je da se provjeravao samo smanjeni set funkcionalnosti, što je zahtijevalo maksimalno 20 minuta po postrojenju, odnosno ukupno oko 5 čovjek-sati po regiji. Iako se na taj način smanjio trošak, on je i dalje bio značajan, pogotovo jer se rad na AP regiji djelomično odvijao izvan redovnog radnog vremena, što je povećavalo trošak prekovremenog rada. Ovaj problem bio je posebno izražen za EU regiju, gdje se radovi obavljaju potpuno izvan radnog vremena, pa je sav rad bio prekovremeni.

Uz sve navedeno, dodatni problem ručnog testiranja bila je točnost i preciznost, pri čemu su različiti testeri provodili testiranja na različitim razinama kvalitete. To je razumljivo, s obzirom na to da se radi o ljudima, a ne o strojevima. Stoga smo odlučili taj posao, u što većoj mjeri, prepustiti strojevima.

Iz navedenog je jasno kako je bilo nužno unaprijediti proces testiranja nadogradnje, prvenstveno radi povećanja kvalitete rada.

3.2 Automatsko testiranje procesa nadogradnje

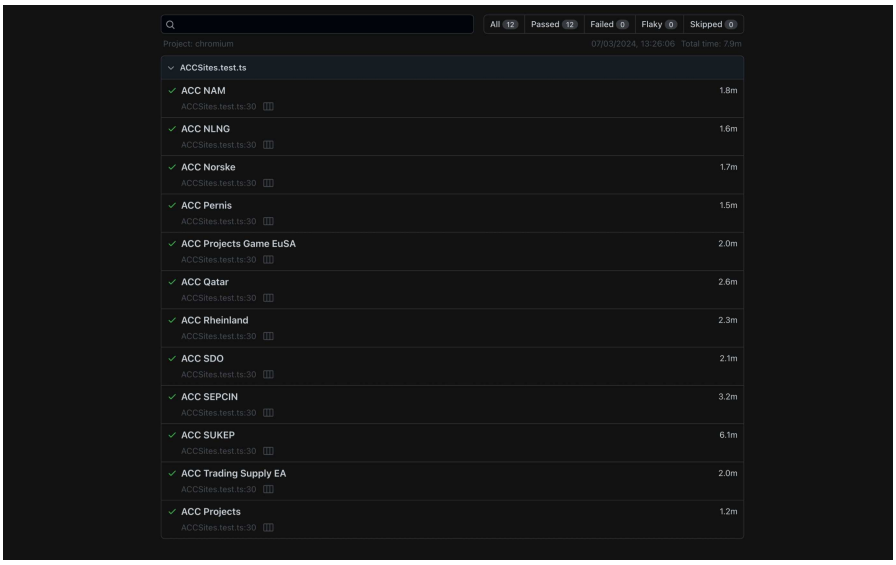
Glavni cilj prelaska na automatski način testiranja je bio povećati preciznost testova i osigurati da se ne izostavi niti jedan korak koji se provjeravao pri ručnom testiranju.

Drugi cilj je bio smanjiti vrijeme potrebno za testiranje s prethodnih 30 minuta na ispod 5 minuta.

Dodatan cilj je bio omogućiti daljnji rast broja klijenata i instalacija koja se mogu nadograditi istovremeno, a bez da se mora uložiti značajno više vremena. Idealno smo htjeli postići $\mathcal{O}(\log n)$ rast potrošenog vremena za svakog novog klijenta.

Nakon što je provedena nadogradnja s automatskim testiranjem potvrđeno je da su glavni ciljevi ostvareni te da će se nastaviti s implementacijom primijenjene

tehnologije i za ostale potrebe. Konkretno, vrijeme testa po postrojenju je iznosilo od 2-4 minuta, ovisno o performansama servera i količini podataka koje klijent ima za razne izvještaje. Na slici 3.1 je prikazan izvještaj koji Playwright generira nakon završenog testiranja.



Slika 3.1: Izvještaj nakon završenog automatskog testiranja

Niti jedan test nije pokazivao znakove problema, ako ih zaista nije bilo (*false-negative* testovi). Svi uočeni problemi nakon isporuke nove verzije su bili riješeni u rekordnom roku, a tome je pridonijelo vrlo kratko vrijeme do otkrivanja problema.

Kao dodatan benefit je primijećeno da se sada testeri mogu više fokusirati na specifične rubne slučajeve koje je potrebno verificirati te tako dodatno smanjiti broj prijavljenih nedostataka od strane korisnika te posljedično povisiti kvalitetu isporučenog proizvoda.

Poglavlje 4

Studija slučaja

Proučavanje teme automatizacije testiranja web aplikacija putem Playwright alata predstavlja značajan korak u unapređenju kvalitete softvera i učinkovitosti testnih procesa. Ovaj slučajni studij je motiviran stvarnim problemom automatizacije testiranja opisanom u poglavlju 3.1. U nastavku poglavlja, detaljno će biti opisan izvorni kod korišten za implementaciju automatiziranih testova, naglašavajući ključne aspekte korištenih metoda i tehnika.

Glavna svrha prikazane skripte jest demonstracija strukturiranog pristupa automatizaciji testiranja web aplikacija koristeći Playwright. Korištenjem modela objekata stranica (POM - Page Object Model) i modularnih funkcija, osigurava se čitljivost i održivost testova. Svaka stranica, definirana u JSON konfiguraciji, prolazi kroz slijed provjera funkcionalnosti, omogućujući fleksibilnost u uključivanju ili isključivanju određenih testnih koraka prema potrebi.

Ovo poglavlje će se detaljno baviti procesom uvođenja i inicijalizacije potrebnih komponenti, deklaracijom varijabli za instanciranje POM klasa, konfiguracijom testova, te glavnim testnim skriptama. Također, uključit će se i objašnjenje pomoćnih funkcija koje povećavaju modularnost i održivost koda, kao i pregled baznih objekata stranica korištenih u testovima.

Kroz ovu studiju slučaja, cilj je pružiti sveobuhvatan uvid u praktične aspekte automatizacije testiranja, pokazujući kako se može postići visoka razina pouzdanosti i učinkovitosti u provjeri funkcionalnosti web aplikacija.

4.1 Uvođenje i inicijalizacija

Skripta započinje serijom uvoženja pomoćnih komponenti potrebnih za testiranje

```
import { test, expect, Page } from '@playwright/test';
import { MainGridToolbarPage } from '../pom/MainGridToolbar';
import { MainNavTabsPage } from '../pom/MainNavTabs';
```

```
import { FLOCForm } from '../pom/FLOCForm';
import { LoginPage } from '../pom/Login';
import { DevToolsPage } from '../pom/DevTools';
import { DashboardPage } from '../pom/Dashboard';
import { MainGridPage } from '../pom/MainGrid';
import { ReportsLightboxPage } from '../pom/ReportsLightbox';
import { FeedbackPage } from '../pom/Feedback';
import { CorrosionLoopPage } from '../pom/CorrosionLoop';
import { ModulePickerPage } from '../pom/ModulePicker';
import { BaseMainOptions } from '../pom/BaseMain';
import { MyAccount } from '../pom/MyAccount';
import { MainGridFiltersPage } from '../pom/MainGridFilters';
import sites from '../data/ListOfSites.json';
import { format } from 'date-fns/format';
import { enGB } from 'date-fns/locale';
```

Navedene komande uvoze dijelove Playwright biblioteke potrebne za učitavanje stranice kao i provjeru dobivenih rezultata s očekivanim vrijednostima. Nadalje, cijeli proizvod je podijeljen u manje komponente koje se dijele unutar programa te su isti definirani u POMu. Podaci o stranicama koje treba testirati su učitani putem JSON formata te također se koristi `date-fns` biblioteka za manipulaciju i formatiranje datuma.

4.2 Deklaracija varijabli

Sljedeće varijable su deklarirane za spremanje instanci POM klasa

```
let loginPage: LoginPage;
let modulePickerPage: ModulePickerPage;
let mainGridPage: MainGridPage;
let mainGridToolbarPage: MainGridToolbarPage;
let baseMain: BaseMainOptions;
let mainNavTabsPage: MainNavTabsPage;
let devTools: DevToolsPage;
let reportsLightboxPage: ReportsLightboxPage;
let myAccount: MyAccount;
```

4.3 Konfiguracija testova

Globalna konfiguracija testova je definirana u `/playwright.config.ts` datoteci, ali je moguće za svaki test definirati dodatne parametre ili nadvladati globalne

kada je to potrebno. Za ovaj test je dodatno definirano da želimo da se izvršava paralelnom načinu izvođenja kako bi se poboljšale performanse s obzirom na to da nema potrebe da se čeka završetak jednog testa da bi se pokrenuo naredni. To je jednostavno napravljeno sljedećom linijom koda:

```
test.describe.configure({ mode: 'parallel' });
```

4.4 Glavni test

Glavna petlja koda koji se izvršava za svakog klijenta (site) je:

```
    for (const site in sites) {
    const currentDate = format(new Date(), 'd MMM yyyy, HH:mm:ss', {
        locale: enGB,
    });
    const siteObj = sites[site];

    test(site, async ({ page }) => {
        const url: string = siteObj.url;
        // Instantiate POM classes with the current page instance
        loginPage = new LoginPage(page);
        modulePickerPage = new ModulePickerPage(page);
        mainGridPage = new MainGridPage(page);
        mainGridToolbarPage = new MainGridToolbarPage(page);
        baseMain = new BaseMainOptions(page);
        mainNavTabsPage = new MainNavTabsPage(page);
        devTools = new DevToolsPage(page);
        reportsLightboxPage = new ReportsLightboxPage(page);
        myAccount = new MyAccount(page);

        // Perform login and initial setup
        await test.step('Login Check', async () => {
            await checkConsoleLog(page, site);
            await page.goto(url + '#fcmfloc');
            await login(page);
            await mainGridToolbarPage.clearAllFilters();
            await mainGridToolbarPage.clearScoping();
        });

await test.step('Feedback Check', async () => {
const feedbackPage = new FeedbackPage(page);
```

```
await feedbackPage.sendFeedback();
});

await test.step('HD Tool Check', async () => {
await devTools.checkHDTool(url);
});

await test.step('Reset Survey Provider', async () => {
// Must be called immediately after HD Tool Check,
// until the page refresh is fixed.
await devTools.resetSurveyProvider();
});

if (siteObj.PEI == true) {
await test.step('Go To PEI', async () => {
await modulePickerPage.goToPEI(url);
});

await test.step('PEI Main Grids Check', async () => {
await baseMain.goToMyAccount();
await myAccount.checkPEIMainGrids();
});

await test.step('Checklist Findings Check', async () => {
if (siteObj.CIVIL == true) {
await modulePickerPage.selectCivil();
}
await checkChecklistFindings(page);
});

await test.step('PI Service Check', async () => {
if (siteObj.PIService == true) {
await checkPIService(page);
}
});

await test.step('Dynamic Forms Check', async () => {
await checkDynamicForms(page);
});
```

```
        if (siteObj.FCM == true) {
            await test.step('Go To FCM', async () => {
                await modulePickerPage.goToFCM(url);
            });

            await test.step('FCM Main Grids Check', async () => {
                await baseMain.goToMyAccount();
                await myAccount.checkFCMMainGrids();
            });
        }

        if (siteObj.MobUrl) {
            await test.step('Mobile Site Check', async () => {
                await page.goto(siteObj.MobUrl + '#peifloc');
                await login(page);
                console.log('Mobile Site Check Successful');
            });
        } else {
            console.log(site + " doesn't have a mobile site url");
        }
    });
}
```

Unutar petlje se provjerava niz krucijalnih funkcionalnosti koji moraju raditi nakon svakog ažuriranja, a praksa je pokazala da ponekad, iz raznih razloga, to nije slučaj. Primarno su to logiranje u program, navigacija na specifične module, provjera logging alata i resetiranje analitičkih alata. Svaki korak testa je definiran kao asinkroni `await test.step` koji objedinjuje jedan korak.

4.5 Pomoćne funkcije

Više asinkronih pomoćnih funkcija je definirano koje sadrže specifične radnje koje se ponovno koriste u različitim koracima testiranja, poboljšavajući modularnost koda i mogućnost održavanja.

```
async function login(page: Page) {
    await loginPage.verifyVersion();
    await loginPage.loginCorpAdm();
    const okButton = page.locator("//button[text()='OK']");
    if (await okButton.isVisible()) {
        await okButton.click();
```

```
    }
    console.log('Login Check Successful');
}
```

4.6 Bazni objektni model stranice

U nastavku slijedi objašnjenje definicije definicije klase TypeScript za `BaseMainOptions` koja se koristi za pristupanje elementima na stranici.

Definiranje i korištenje zasebnih klasa poboljšava mogućnost ponovne upotrebe koda kao i njegovo održavanje što za posljedicu ima lakše pisanje automatiziranih testova.

4.6.1 Svojstva klase

```
export class BaseMainOptions {
    readonly page: Page
    readonly buttonActionItem: Locator
    readonly buttonMyAccount: Locator
    readonly notificationText: Locator
    readonly notificationClose: Locator
```

Klasa `BaseMainOptions` sadrži svojstva koja predstavljaju različite elemente na web stranici. Ova su svojstva inicijalizirana kao samo za čitanje, što znači da su postavljena jednom i da se nakon toga ne mogu mijenjati. Svojstva uključuju:

- page: Instanca klase Page koja predstavlja web stranicu.
- buttonActionItem: Lokator za "Action Items" gumb.
- buttonMyAccount: Lokator za "My Account" gumb.
- notificationText: Lokator za tekst obavijesti.
- notificationClose: Lokator za gumb za zatvaranje obavijesti

4.6.2 Konstruktor

```
constructor(page: Page) {
    this.page = page
    this.buttonActionItem = page.locator("//button[@title='Action Items']")
    this.buttonMyAccount = page.locator("//span[.='My Account']")
    this.notificationText = page.locator('p[class="CenNotificationText"]')
```

```
    this.notificationClose =
        page.locator('//div[@class="CenNotificationClose"]').first()
}
```

Konstruktor inicijalizira klasu `BaseMainOptions` instancom `Page` te dodjeljuje lokatore za različite elemente njihovim odgovarajućim svojstvima, koristeći XPath i CSS selektore.

4.6.3 Metode

```
async checkNotification(notification: string) {
    await expect(this.notificationText).toBeVisible()
    const content = await this.notificationText.textContent()
    expect(content).toBe(notification)
}
```

Ova asinkrona metoda provjerava je li obavijest s određenim tekstom vidljiva na stranici. Čeka da element obavijesti bude vidljiv te provjerava njegov tekstualni sadržaj i potvrđuje da sadržaj odgovara očekivanom tekstu.

```
async closeNotification() {
    await this.page.waitForTimeout(200)
    if (await this.notificationClose.isVisible()) {
        await this.notificationClose.click()
    }
}
```

Ova metoda čeka kratko razdoblje (200 milisekundi), a zatim provjerava je li gumb za zatvaranje obavijesti vidljiv. Ako jest, metoda klikne gumb za zatvaranje kako bi zatvorili pop-up s obavijesti.

Ostale metode su definirane na sličan način.

Poglavlje 5

Zaključak

U ovom radu istražena je važnost testiranja klijentskih komponenti u procesu razvoja softvera, s posebnim naglaskom na korištenje Microsoftovog alata Playwright. Kako se složenost modernih web aplikacija povećava, pouzdano i efikasno testiranje postaje ključno za održavanje kvalitete i funkcionalnosti proizvoda. Playwright se istaknuo kao moćan alat za end-to-end testiranje zbog svoje podrške za najnovije web tehnologije, jednostavne sintakse i mogućnosti provođenja robusnih i održavanih testova na različitim preglednicima i platformama.

Rezultati ovog istraživanja pokazuju da korištenje Playwrighta može značajno smanjiti broj grešaka koje prolaze nezamijećene do korisničkog iskustva, čime se poboljšava ukupna kvaliteta proizvoda. Nadalje, implementacija Playwrighta omogućuje timovima za razvoj softvera da brže i efikasnije identificiraju i otklone pogreške, što rezultira smanjenjem troškova i vremena potrebnog za održavanje i nadogradnju aplikacija.

U zaključku, integracija Playwrighta u proces razvoja web aplikacija predstavlja vrijednu investiciju koja može donijeti dugoročne benefite u smislu pouzdanosti, kvalitete i zadovoljstva korisnika. Buduća istraživanja mogu se fokusirati na dodatne tehnike i alate za testiranje kako bi se još više poboljšao proces osiguranja kvalitete u razvoju softvera.

Literatura

[1] S. Quadri and S. U. Farooq, "Software testing–goals, principles, and limitations," *International Journal of Computer Applications*, vol. 6, no. 9, p. 1, 2010.

[2] A. Mundra, S. Misra, and C. A. Dhawale, "Practical scrum-scrum team: Way to produce successful and quality software," in *2013 13th International Conference on Computational Science and Its Applications*, pp. 119–123, IEEE, 2013.

[3] Microsoft, "Playwright homepage." `https://playwright.dev/`. (3.4.2022.).

[4] Microsoft, "Yaml schema reference for azure pipelines," 2024. (7.7.2024.).

[5] Microsoft, "Creating test plan in azure devops," 2024. (8.7.2024.).

[6] Matt Pocock, "Beginner's typescript," 2024. (8.2.2024.).

[7] Node JS, "Introduction to node.js," 2023. (15.12.2023.).

[8] B. J. Sauser, R. R. Reilly, and A. J. Shenhar, "Why projects fail? how contingency theory can provide new insights–a comparative analysis of nasa's mars climate orbiter loss," *International Journal of Project Management*, vol. 27, no. 7, pp. 665–679, 2009.

[9] T. Linz, *Testing in scrum: A guide for software quality assurance in the agile world*. Rocky Nook, Inc., 2014.

[10] R. Löffler, B. Güldali, and S. Geisen, "Towards model-based acceptance testing for scrum," *Softwaretechnik-Trends, GI*, 2010.

[11] S. Dileepkumar and J. Mathew, "Optimize continuous integration and continuous deployment in azure devops for a controlled microsoft. net environment using different techniques and practices," in *IOP Conference Series: Materials Science and Engineering*, vol. 1085, p. 012027, IOP Publishing, 2021.

[12] R. Hundhausen, *Professional Scrum Development with Azure DevOps*. Microsoft Press, 2021.

[13] M. Sauerová, "Web browser recorder," 2022.

[14] D. Flanagan, *JavaScript: The definitive guide: Activate your web pages*. " O'Reilly Media, Inc.", 2011.

[15] D. Vanderkam, *Effective TypeScript*. " O'Reilly Media, Inc.", 2024.

[16] S. Baumgartner, *TypeScript Cookbook*. " O'Reilly Media, Inc.", 2023.

Popis slika