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:	**Diplomski rad**
Znanstveno područje:	**Društvene znanosti**
Znanstveno polje:	**Informacijske i komunikacijske znanosti**
Znanstvena grana:	**Informacijski sustavi i informatologija**
Mentor:	**dr.sc. Nikola Tanković**

Pula, srpanj, 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.

Izjava o samostalnosti izrade završnog rada

Ja, dolje potpisani Kristijan Cetina, 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 nedozvoljeni 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.

U Puli, 7. kolovoza 2024.

Student: Kristijan Cetina

Izjava o korištenju autorskog djela

Ja, Kristijan Cetinadajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom Testiranje klijentskih komponenti web aplikacija pomoću alata Playwrightkoristi 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, 7. kolovoza 2024.

Student: Kristijan Cetina

Sažetak

Često se kaže da je testiranje softvera jednako važno kao i samo kodiranje. Kako se kompleksnost aplikacija povećava, tako raste i vjerojatnost pojave pogrešaka koje mogu negativno utjecati na korisničko iskustvo. Da 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 održavanih testova.

Ključne riječi

Playwright, JavaScript, open-source

Sommario

Si dice spesso che testare il software sia importante quanto la codifica stessa. Con l'aumento della complessità delle applicazioni, aumenta anche la probabilità di errori che possono influire negativamente sull'esperienza utente. Per prevenirlo, utilizziamo diverse tecniche di test, una delle quali è il test dei componenti client-side.

Questo documento si concentra sull'utilizzo di Playwright di Microsoft per testare i componenti client-side. Playwright è una scelta popolare per i test end-to-end delle moderne applicazioni web in quanto consente test affidabili su diversi browser e piattaforme. I suoi principali vantaggi includono il supporto per le ultime tecnologie web, una sintassi semplice e la capacità di scrivere test robusti e manutenibili.

Parole chiave:

Playwright, JavaScript, open-source

Abstract

It's often said that testing software is equally important as coding itself.

As application complexity grows, 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 testing client-side components.

This paper focuses on using Microsoft's Playwright for testing client-side components. Playwright is a popular choice for end-to-end testing of modern web applications as it enables 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 i opis zadatka

U dinamičnom svijetu razvoja softvera, automatizacija testiranja igra ključnu ulogu u osiguravanju kvalitete proizvoda. End-to-end (E2E) testiranje je pristup koji omogućuje simulaciju stvarnog korisničkog iskustva kroz cijeli sustav, od početka do kraja. Cilj ovih testova je potvrditi da svi dijelovi sustava funkcioniraju zajedno na predviđeni način. Sa sve većom složenošću softverskih sustava i potrebom za bržim izdanjima, automatsko E2E testiranje postaje sve relevantnije. Međutim, ovo testiranje donosi sa sobom niz izazova koje je potrebno adresirati kako bi bilo učinkovito i korisno.

Opis i definicija problema

Automatsko end-to-end testiranje softvera obuhvaća proces kreiranja, izvršavanja i održavanja testova koji provjeravaju funkcionalnost aplikacije kao cjeline. Problem koji se istražuje u ovom radu može se definirati kroz sljedeće ključne aspekte:

Složenost Kreiranja Testova: Kreiranje automatskih E2E testova zahtijeva duboko razumijevanje svih dijelova sustava i njihovih međusobnih interakcija. Ovo može biti izuzetno složeno u velikim i distribuiranim sustavima.

Održavanje Testova: S obzirom na česte promjene u kodu i sistemskim zahtjevima, automatski E2E testovi zahtijevaju redovno održavanje kako bi ostali relevantni. Svaka promjena može potencijalno zahtijevati modifikaciju ili kreiranje novih testova.

Izvršavanje Testova: Automatski E2E testovi često traju duže od drugih vrsta testova (kao što su unit testovi ili integracijski testovi) zbog svoje prirode koja obuhvaća cijeli sustav. Ovo može rezultirati dugim vremenom izvršavanja i problemima s performansama.

Pouzdanost Testova: Testovi moraju biti pouzdani, tj. rezultati testiranja moraju biti točni i konzistentni. Lažno pozitivni ili negativni rezultati mogu dovesti do gubitka povjerenja u testove i dodatnih troškova.

Integracija sa CI/CD Procesima: Automatsko E2E testiranje mora biti integrirano s kontinuiranim integracijskim i kontinuiranim isporučnim (CI/CD) procesima kako bi podržalo agilne prakse razvoja softvera. Ovo zahtijeva visok nivo automatizacije i orkestracije.

Struktura rada

Struktura ovoga rada podjeljena je u logičke cjeline. Nakon uvoda i objašnjavanja rada, u poglavlju 1 - Uvod u testiranje programskog rješenje i osiguranje kvalitete objašnjen

je proces testiranje i osiguranja kvalitete programskog rješenja.

Poglavlje 2 - 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 rješiti ovim radom.

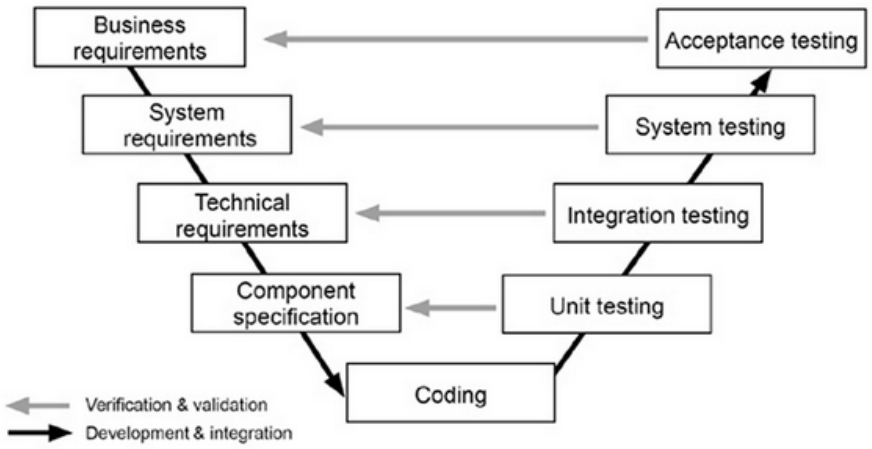
Poglavlje 4 - Studija slučaja pruža detaljniji uvid u samo rješenje koje je implementirano unutar kompanije u kojoj autor trenutno radi.

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

Poglavlje 1

Uvod u testiranje programskog rješenje i osiguranje kvalitete

Testianje 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 namjenjene 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 sa izradom plana testiranja unutar kojeg se definira šte se testira na koji način te koji su uvjeti da se zadani test smatra uspješnim (*engl. Acceptance critera*).

Tokom samog razvijanja nove funkcionalnosti u softwareskom prozvodu često se izvršava ručno testiranje kako bi se utvrdilo da li proces razvoja ide u zadanom smjenu i u konačnici da se radi ono što je dogovoreno bilo interno s timom ili čak i sa klijentom.

Nakon što se završi sam razvoj nove funkcionalnosti onda se radi i završno funkcionalno testiranje i izrada automatskih testova koji će se u budućnosti izvršavati automatski u zadanom intervalu kako bi se osiguralo da uvođenje nove funkcionalnosti ne uvode nove pogreške na već ispravnim funkcionalnostima. Poznato kao i regresijsko testiranje.

Bitno je testirati realne scenarije koji se očekuju da moraju zadovoljiti, kao i one scenarije od kojih se očekuje da nesmiju proći test. To se radi u svrhu potvrde da test zaista radi ono što je namijenjen, a ne da imamo propust u samom testu koji uvijek vraća pozitivan rezultat ili da testirana funkcija nema implementiranu validaciju ulaznih parametara koji onda mogu izazvati nepoželjno ponašanje programa.

1.2 Ciljevi testiranja

Ciljevi testiranja moraju zadovoljavati nekoliko kriterija, a to su:

- Specifičnost
- Mjerljivost
- Ostvarljiv
- Realističan
- Vremenski ograničen

Neki od ciljeva testiranja su: [1]

Verifikacija i validacija

Cilja testiranja nije samo pronaći pogreške u kodu ili dizajnu. Cilj je verificirati da software zaista radi ono što je namjenjen i kako je zamišljen. Jedan od rezultata testiranja je i izvještaj (*test report*).

Prioretiziranje pokrivenosti

U idealnom svijetu sa neograničenim resursima svaki dio koda i funkcionalnosti bi bio pokriven testovima, ali nažalost to nije moguće. Zato je bitno pravilno odrediti što je prioritet te što će se pokriti testovima. U pravilu su to one funkcionalnosti i značajke koje nisu vidljive na prvi pogled čim se otvori program jer su takvi problemi lako uočljivi svakome. Isto tako, beskonačni testovi uzimaju mnogo dragocjenog vremena pa je i u tom pogledu bitno odrediti što se treba testirati.

Sljedivost

Dokumentiranje testova kada se nešto i kako testiralo je bitno kako bi se u slučaju pojave problema moglo odrediti kada je i koja promjena uzrokovala neželjeno ponašanje proizvoda. To je posebno bitno u određenim kategorijama softwarea kao npr. u financijskom poslovanju gdje se može tražiti dodatna odgovornost samog proizvoda.

1.3 Uloga testera u timu

Glavna uloga testera, kao i samog procesa testiranja, je dodatna sigurnosna mreža koja je samo zadnja karika u lancu procesa testiranja i osiguranja kvalitete proizvoda. Dok su programeri (*developeri, engl. developers*) zaduženi za pisanje unit testova, testeri su zaduženi za pisanje i izvršavanje funkcionalnih testova. Testeri također pomažu product owneru sa izradom kriterija za uspješno prihvaćanje rezultata testova [2]. Bitno je napomenuti kako su i developeri i testeri dio istog tima te kao tim imaju zajednički cilj - isporuka najkvalitetnijeg proizvoda moguće unutar zadanih parametara.

Poglavlje 2

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.

Prednosti Playwrighta:

- Jednostavan za korištenje: Playwright ima intuitivan API koji je sličan JQuery-ju i Cypress-u.
- Brz i pouzdan: Playwright je optimiziran za brzinu i pouzdanost, što ga čini idealnim za testiranje web aplikacija u produkciji.
- Svestran: Playwright se može koristiti za testiranje različitih tipova web aplikacija, uključujući jednostavne web stranice, jednostruke web aplikacije (SPA) i višestruke web aplikacije (MPA).
- Podržava više jezika: Playwright se može koristiti s raznim jezicima programiranja, uključujući JavaScript, TypeScript, Python, Java i C#.

Playwright se može koristiti za:

- Automatizaciju UI testova: Playwright se može koristiti za pisanje automatiziranih UI testova koji provjeravaju funkcionalnost web aplikacija.
- Web skrapanje: Playwright se može koristiti za prikupljanje podataka sa web stranica.
- Generiranje screenshot-ova i videozapisa: Playwright se može koristiti za generiranje screenshot-ova i videozapisa web stranica.

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),

[1] `https://playwright.dev/docs/intro#system-requirements`

- 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 koristno 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

```
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
```

Primjetimo kako je uvrijeđena 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.

Ukoliko 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

[2] `https://www.npmjs.com/package/playwright#documentation--api-reference`

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

- koji se preglednik koristi,
- koja je veličina prozora preglednika,
- koji se mobilni uređaj koristi u slučaju tesitiranja 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 >
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

Playwright ima mogućnost generiranja testova tako što snima klikanje miša korisnika te to prevodi u kod koji se može izvršavati. To je vrlo jednostavan i praktičan način da čak i totalni početnici mogu krenuti s izradom automatskih testova, a kasnije s iskustvom se ti testovi mogu rafinirati i poboljšavati. Često je i tako generirai kod dovoljno dobar za upotrebu kod jednostvanijih slučajeva, a zasigurno je dobar za brzo sastavljanje testova da se izbjegne često dosatno tipkanje djelova koda koji se neće kasnije ponovno upotrebljavati. To uklanja potrebu za pisanje prilagođenih funkcija kao što radimo prilikom izrade projekta koji planiramo dugo vremena održavati i na taj način se može uštediti dosta vremena. Iako, treba biti oprezan s time jer često ušteda vremena na početku vodi do puno utrošenog vremena kasnije, ali to je tema koja je izvan okvira ovog rada.

2.5 Pokretanje alata za generiranje testova

Alat za generiranje testova se pokreće putem `codegen` naredbe koja prima argument URL web stranice za koju se želi generirati testovi. URL nije obavezan i može se pokrenuti alat bez njega, a zatim dodati URL izravno u prozoru preglednika.

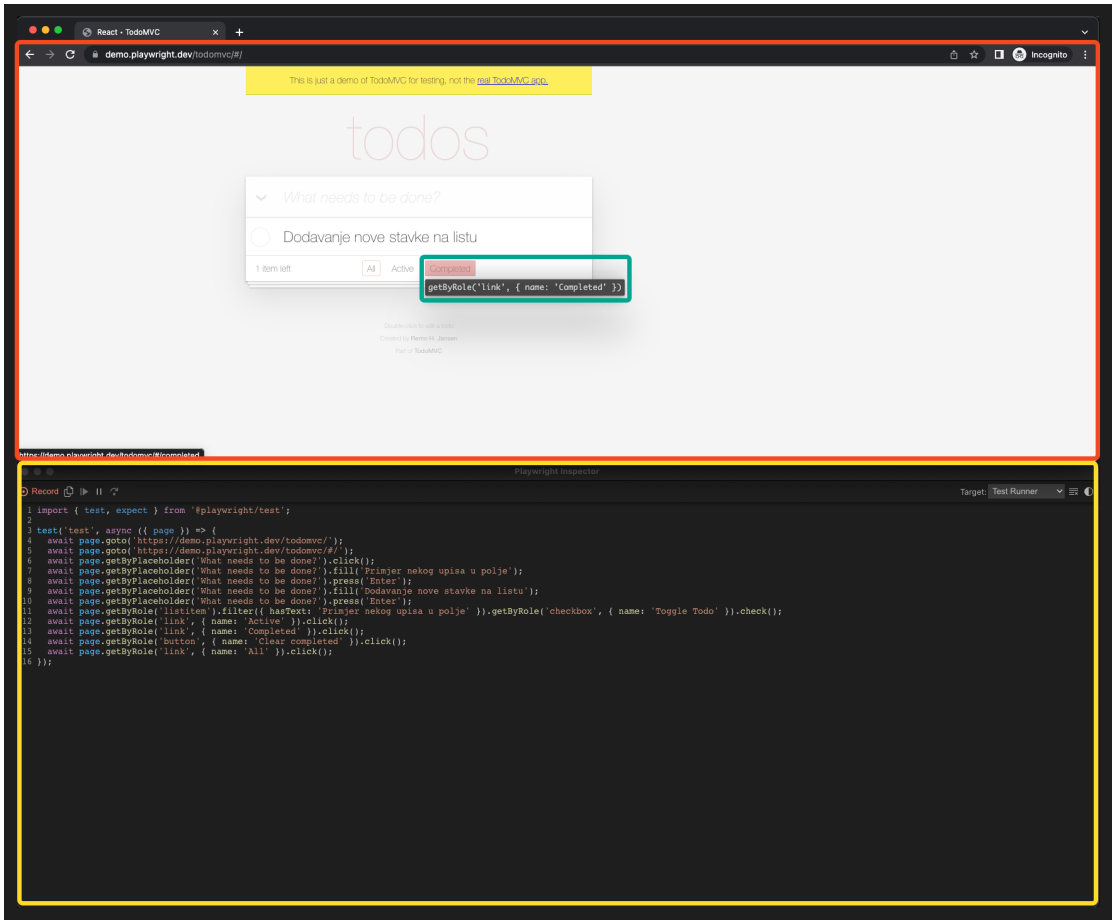
Pokazati ćemo to na promjeru koji se nalazi u službenoj dokumentaciji [3] koristeći naredbu

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

Na slici 2.3 je prikazan izgled sučelja za generiranje testova koji se sastoji od nekoliko djelova:

- prozor preglednika unutar kojeg se izvršava aplikacija - označen s crvenim okvirom (gore)
- prozor unutar kojeg se prikazuje generirani kod - označen s žutim okvirom (dolje)
- Lokator koji će Playwright koristiti se prikazuje kada se stavi miš preko elementa - označen sa zelenim okvirom (unutar prozora preglednika)

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



Slika 2.3: Izgled sučelja za generiranje testova

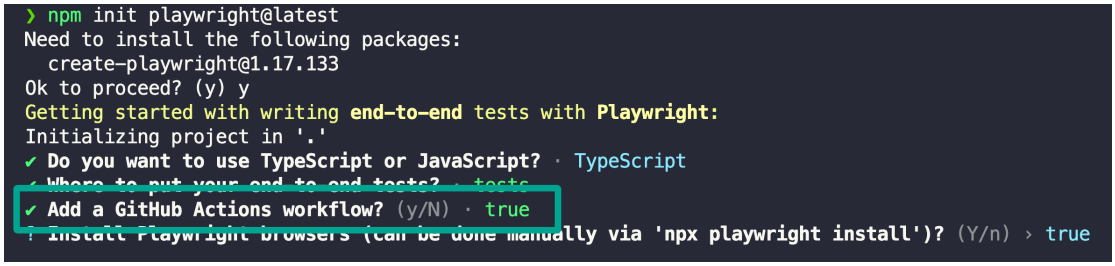
2.6 Kontinuirana integracija i testiranje

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 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 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:



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

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

Konfiguracija u YAML formatu:

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

Poglavlje 3

Testiranje nakon nadogradnje na novu verziju

Jedna od čestih aktivnosti svakog razvojnog tima je nadogradnja programa na novu verziju. U poduzeću je praksa da se pušta u produkcijski rad nova verzija programa jednom mjesečno za glavnog klijenta. Kako klijent ima postrojenja na 6 kontinenta uz dodatna postrojenja koja se nalaze u međunarodnim vodama svih svjetskih oceana proces nadogreadnje je podijeljen po regijama. To su Americas (Sjeverja i Južna Amerika), EU (Europa i Bliski istok + Afrika) te AP (Azija i Pacifik). Svaka od regije ima više od 15 postrojenja koja koriste zasebne servere što znači da se nadogradnja vrši na više od 15 lokacija po regiji. Već iz ovoga je izvjesno kako je to puno potencijalnih problema i zastoja koja se mogu pojaviti te treba osigurati da je proces nadogranje čim je više optimalan bez nepotrebnih zastoja i prekida u radu.

3.1 Postojeći način testiranja

Do uvođenja automstkih testova standardna procedura se sastojala od toga da AM regiju detaljno testira 3-4 testera (inženjera za kontrolu kvalitete - QA) koji bi svaki od njih provjerio 3 do 4 postrojenja (site). U prosjeku, za provjeriti jedno postrojenje je trebalo 20 do 30 minuta kako bi se osiguralo da su sve promjene aplicirane ispravno te da nisu izazvale neželjene nuspojave i prestanak rada postojećih funkcionalnosti. Naravno, zbog velikog broja funkcionalnosti te nedostatka vremena da se svaka od njih ručno provjeri bili smo vrlo izbirljivi što će se testirati s obzirom na kritičnost neke funkcionalnosti. Cijeli taj proces bi trajao oko 1:30 do 2 sata po članu test tima, a sve ukupno bi se trošak procijenio na 6-7 radnih čovjek-sati. Ako to pomnoćimo s cijenom rada lako se dolazi do ukupnog troška testiranja nakon nadogradnje.

Nakon što se verificiralo da je novi paket programa ispravan, te ako klijenti nisu prijavili ozbiljne probleme koji ih spriječavaju u radu putem sustava za prijavu kvara (*showstopper*), moglo se pristupiti nadogradnji sljedeće regije Asia Pacific.

Tada bi se ponovila opet ista procedura, uz eventualne izmjene da bi bio 1 član testinog tima manje kako bi se smanjili prekovremeni sati jer nije bilo potrebe za provjerom svih funkcionalnosti, već je bilo prihvatljivo da se provjeri i reducirani set za koji bi trebalo maksimalno 20 minuta po postrojenju što bi ukupno iznosilo oko 5 čovjek-sati po regiji. Iako je to manji trošak, i dalje je značajni trošak. Čim više što se je termin za radove na AP regiji djelomično izvan rednovnog radnog vremena poduzeća tako da treba uračanti i trošak prekovremenog rada. To je posebno izraženo za EU

regiju kod koje pak termin za radove je u potpunosti izvan redovnog radnog vremena pa je u tom slučaju sav rad je prekovremeni rad.

Uz sve navedeno dodatan problem ručnog testiranja je točnost i preciznost istog. Isto tako je normalno za očekivati kako će različiti testeri odraditi posao na različitoj razini kvalitete jer ipak nisu strojevi. Zato smo odlučili taj posao prepustiti strojevima. Barem u mjeri koliko je to moguće.

Iz gore navedenog je razvidno kako je bilo potrebno poboljšati proces testiranja nadogradnje primarno iz razloga povećanja kvalitete posla.

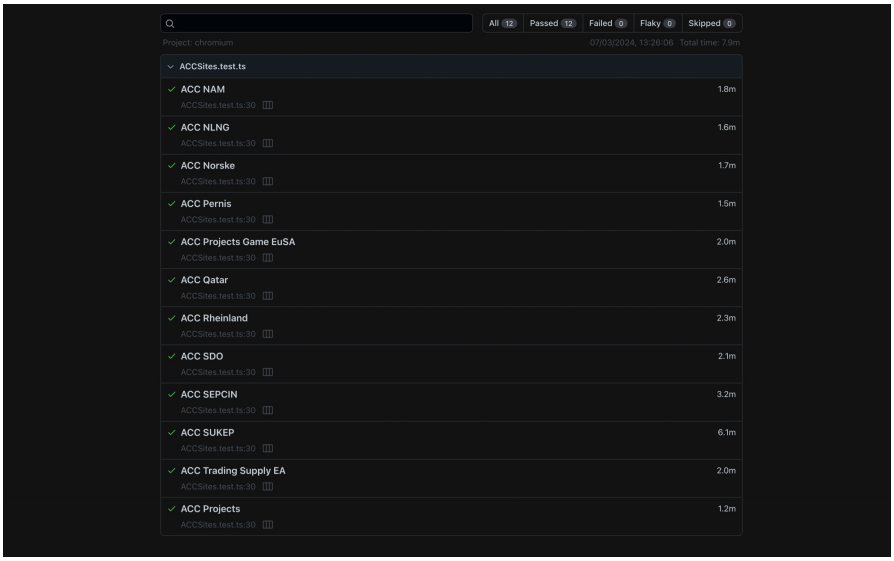
3.2 Automatsko tesiranje procesa nadogradnje

Glavni cilj prelaska na autimatski 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 smanjti vrijeme potrebano za testiranje sa prethodnih 30 minuta na ispod 5 minuta.

Dodatan cilj je bio omogućiti daljni 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 tesiranjem potvrđeno je da su glavni ciljevi ostvareni te da će se nastaviti s implementacijom primjenjene 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 tesiranja

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

Kao dodatan benefit je primjećeno da se sada testeri mogu više fokusirati na specifične rubne slučajeve koje je potrebano verificirati te na taj način dodatno

smanjiti broj prijavljenih nedostataka od strane korisnika te poslijedično 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, definisana 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 djelove 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 performase s obzirom 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 objašanjenje 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 Inc., "Playwright homepage." `https://playwright.dev/`. (3.4.2022.).

[4] 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.

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

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

Popis slika