Testno ogrodje za razvijalce - ali kako doseči, da razvijalci celovito testirajo

Mitja Krajnc, Tadej Ciglič, Boris Ovčjak

Databox, Ptuj, Slovenija
mitja.krajnc@databox.com, tadej.ciglic@databox.com, boris.ovcjak@databox.com

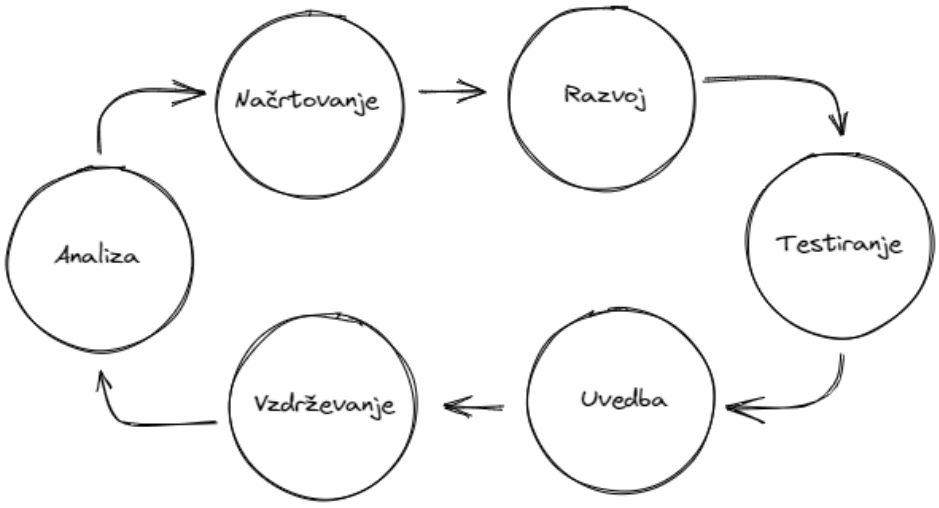
Pred izdajo rešitve v produkcijsko okolje je zagotavljanje kakovosti in testiranje pomemben korak, vendar je hkrati tudi postopek, ki se ga razvijalci z vso silo otepamo. Če je testiranje enot že ustaljen del razvojnega procesa pa se drugih vrst testiranj, kot sta npr. dimno testiranje in integracijski testi, pogosto že na daleč izognemo in to predamo testerjem. Da bi celoten proces testiranja izboljšali, smo se odločili pripraviti ogrodje, ki bi razvijalcem omogočilo pisanje in izvajanje kompleksnih testov na enostaven način - podobno kot testiranje enot. V članku bomo predstavili ogrodje za testiranje, ki je v prvi vrsti namenjeno prav razvijalcem za enostavno uporabo pri razvoju. Ogrodje temelji na odprtokodni rešitvi Playwright, ki je namenjena avtomatskemu testiranju spletnih rešitev v brskalniku in omogoča izdelavo testov v različnih programskih jezikih. Ker smo želeli zagotoviti čim širšo uporabo ogrodja v podjetju, smo uporabili različico, ki temelji na jeziku TypeScript. Struktura ogrodja je sestavljena iz abstrakcij, ki so že znane iz objektno orientiranega programiranja in so zato razvijalcem blizu, s čimer omogoča, da preko zaporedne uporabe razredov in metod ustvarijo testni scenarij, ki ga lahko na preprost način večkrat zapored hitro in učinkovito uporabijo.

Ključne besede:
testiranje
avtomatizacija testiranja
Playwright
ogrodje
razvoj

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.14

1 Uvod

Pri razvoju programske opreme se je tekom let v praksi izoblikoval nek tipičen razvojni cikel, ki ga z manjšimi prilagoditvami in spremembami poznamo in uporabljamo vsi. Sestavljen je iz korakov kot so analiza, načrtovanje, razvoj, testiranje, uvedba in vzdrževanje. Na sliki 1 so omenjeni koraki postavljeni v ciklično strukturo, kjer se po vzdrževanju že prične analiza za nadaljnji razvoj.



Slika 1: Tipičen razvojni cikel.

V našem članku se bomo podrobneje lotili faze testiranja, ki bi se naj izvajala po razvoju oziroma že v sami fazi razvoja. Razvijalci bi naj tukaj pisali in izvajali teste za preverjanje pravilnega delovanja kode. Testiranje lahko vključuje enote testiranja (unit testing), integracijsko testiranje (integration testing), funkcionalno testiranje (functional testing) in sprejemno testiranje (acceptance testing). V pomoč je zagotovo tudi dimno testiranje (smoke test), pri katerem imajo glavno vlogo testerji in ekipa za zagotavljanje kakovosti. Kljub temu, da se na papirju vse to lepo bere pa se je potrebno zavedati, da pisanje testov s strani razvijalcev ni ravno najbolj priljubljena aktivnost. Čeprav ni res, da vsi razvijalci ne marajo pisanja testov, obstaja več razlogov, zakaj nekateri razvijalci morda ne uživajo v pisanju testov ali jim ne namenjajo prednosti v svojem razvojnem procesu. Tu je nekaj pogostih razlogov:

- **Omejenost časa**: Razvijalci se pogosto soočajo s tesnimi roki in pritiskom, da hitro dostavijo funkcionalnosti ali posodobitve. Pisanje obsežnih testov lahko vzame veliko časa, še posebej pri kompleksnih sistemih. Nekateri razvijalci morda menijo, da lahko prihranijo čas tako, da preskočijo ali zmanjšajo pisanje testov ter se osredotočijo le na pisanje kode.
- **Pomanjkanje takojšnjega zadovoljstva**: Pisanje testov običajno ne zagotavlja takoj vidnih ali oprijemljivih rezultatov. Zahteva trud in pozornost do podrobnosti, ne da bi neposredno prispevala k funkcionalnosti ali vidnim spremembam za končne uporabnike. Posledično se zdi, da pisanje testov deluje manj nagradno v primerjavi s pisanjem kode, ki proizvaja vidne spremembe.
- **Kompleksnost in učna krivulja**: Učinkovito pisanje testov zahteva znanje in razumevanje testnih okolij, orodij ter najboljših praks. Osvojiti zapletenosti testiranja je lahko izziv, še posebej za razvijalce, ki so relativno novi na tem področju ali jim manjka izkušenj pri pisanju zanesljivih testnih paketov.
- **Nadležnost vzdrževanja**: Teste je treba vzdrževati in posodabljati, ko se koda razvija in spreminja skozi čas. Dodajanje novih funkcionalnosti ali preoblikovanje obstoječe kode lahko zahteva ustrezne posodobitve testov. To stalno vzdrževanje lahko razvijalcem predstavlja dodatno breme, še posebej, ko se že ukvarjajo z drugimi prioritetami.

- **Prioritizacija drugih nalog**: V nekaterih primerih se razvijalci morda počutijo pritisnjene v kot, da dajo prednost razvoju funkcionalnosti ali popravkom napak pred pisanjem testov. Vodstvo ali zahteve projekta lahko poudarjajo hitro dostavo novih funkcionalnosti, kar vodi k temu, da razvijalci ne postavljajo pisanja testov na prednostno mesto, da bi izpolnili te zahteve.

V ta namen smo se odločili da s sodelovanjem med razvijalci in testerji razvijemo ogrodje, ki bi naslovilo zgoraj omenjene točke. Se pravi bi omogočilo hitro pripravo kompleksnih testov, s čimer bi že sami razvijalci prevzeli del kontrole pred izdajo. Takšno ogrodje bi moralo biti tudi del neprekinjene integracije, pri čemer se testi izvedejo v skladu z vnaprej definiranimi scenariji, ki jih definirajo testerji. Na podlagi teh zahtev smo izdelali ogrodje za testiranje »Platform Testing Service for Databox« ali PTSD, ki ga vzdržuje ekipa za zagotavljanje kakovosti in je s svojim širokim spektrom razvijalcem prijaznih operacij namenjen prav njim za uporabo pri razvoju.

2 Testno ogrodje za razvijalce PTSD

Ena izmed prvih odločitev pri izdelavi ogrodja PTSD je bila, kakšno ogrodje vzeti za osnovo. Znotraj podjetja smo že uporabljali ogrodje Katalon [2], ki omogoča avtomatizacijo testiranja spletnih aplikacij, API-jev, mobilnih aplikacij in namiznih aplikacij ter vključuje funkcionalnosti, kot so snemanje in predvajanje testov, upravljanje podatkov, generiranje poročil o testiranju. Orodje ponuja uporabniku prijazen vmesnik, ki olajša ustvarjanje, izvajanje in upravljanje testnih primerov in podobno. Kaj kmalu smo ugotovili, da sam Katalon Studio sicer ima uporabniku prijazen vmesnik, vendar pa za naše razvijalce ni bil ugodna rešitev. Zato smo iskali rešitev, ki bila bolj pisana na roko razvijalcem, bila lahkotnejša in po načinu uporabe bolj podobna razvijalčevemu vsakdanu. V ta namen smo ocenili priljubljena orodja kot so Selenium, Cyprsess, Puppeteer in Playwright [3,4]. Za naš primer je bila še posebej pomembna podpora različnim brskalnikom, več zavihkom in oknom, hitrost, stabilnost, uporaba lokatorjev in kompatibilnost z obstoječim programskimi jeziki. Po pregledu ogrodij in preizkusu smo se za to odločili za ogrodje Playwright.

2.1 Playwright

Playwright je razmeroma novo odprtokodno orodje za avtomatizacijo testiranja v brskalniku, njegovo prvo različico je Microsoft izdal leta 2020. Izdelala ga je ekipa, ki stoji za ogrodjem Puppeteer. To je ogrodje brezglavega testiranja za Chrome/Chromium. Playwright. Playwright sam po sebi presega Puppeteer in med drugimi spremembami zagotavlja podporo za več brskalnikov. Zasnovan je za samodejno testiranje spletnih aplikacij od konca do konca (end-to-end). Posebej je prilagojen za sodoben splet in na splošno deluje zelo hitro, tudi pri kompleksnih projektih testiranja. Čeprav je precej novejši od Seleniuma, Playwright hitro pridobiva na moči in ima vse več privržencev. Zaradi svoje mladosti sicer podpira manj brskalnikov/jezikov kot Selenium, vendar po istem principu vključuje tudi novejše funkcije in zmogljivosti, ki so bolj usklajene s sodobnim spletom. Aktivno ga razvija Microsoft. Med značilnosti ogrodja Playwright, ki so ključne za naše potrebe spadajo:

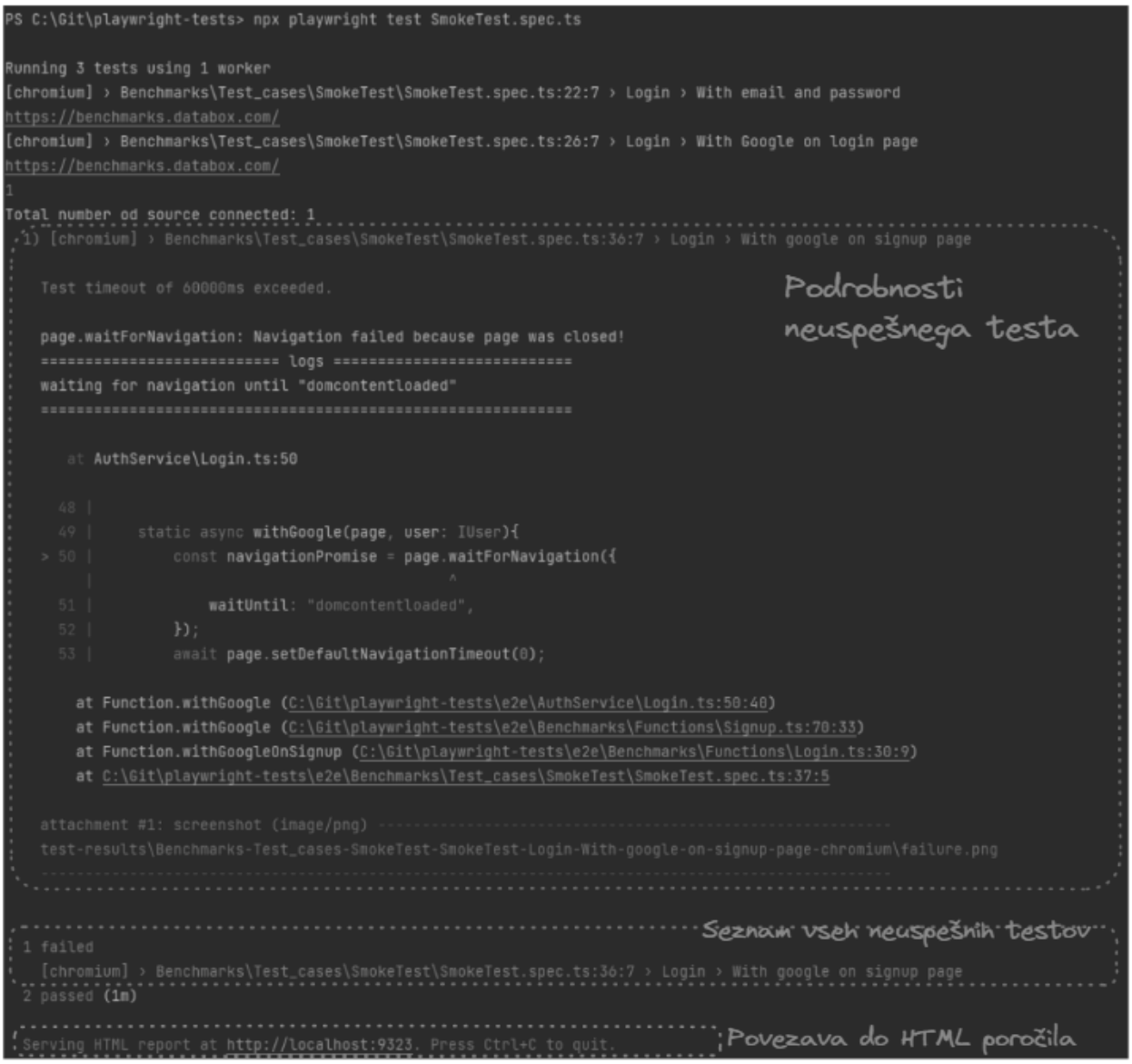
- **Podpora za več brskalnikov**: Kot že omenjeno Playwright omogoča avtomatizacijo brskanja in testiranje spletnih aplikacij v več brskalniki. To so Chrome, Firefox, Safari in Edge. S čimer nam omogoča, da preverimo, ali naša aplikacija deluje pravilno v vseh brskalnikih, ki jih tudi uradno podpiramo.
- **Podpora več platformam:** Poleg podpore za različne brskalnike, Playwright omogoča tudi avtomatizacijo na različnih platformah, vključno z operacijskimi sistemi Windows, macOS in Linux.
- **Hitrost in zanesljivost**: Playwright je zasnovan tako, da omogoča hitro in zanesljivo izvajanje testov. Avtomatizacija brskalnikov je optimizirana, kar pripomore k učinkovitemu izvajanju testov in zmanjšanju časa potrebnega za zaključek testnih sklopov. Pri evaluaciji ogrodij smo pri prepisu nekaterih testov iz Katalona zaznali občutno povečano hitrost, in sicer tudi do 300 odstotkov.

- **Podpora za več jezikov:** Playwright podpira več programskih jezikov, kot so JavaScript, TypeScript, Python in C#. Zaradi te podpore, smo lahko omogočili, da smo izbrali jezik, ki je najbolj ustrezal našim razvijalcem.

Glede na uporabo programskih jezikov v različnih ekipah v podjetju smo se odločili za izdelavo ogrodja v jeziku TypeScript. S tem smo našli najbližji skupni imenovalec tako ekipam, ki delajo na čelnem delu sistema (frontend) kot ekipam, ki delajo na zaledju sistema (backend). Za poganjanje Playwright potrebuje samo okolje Node.js in namestitev paketa preko enega izmed paketnih upraviteljev (npm, yarn, pnpm). Poganjanje poteka preko konzole z ukazom, kjer določimo kateri test v kateri zbirki poženemo. Tekom poteka testa se v konzolo lahko izpisujejo tudi kakšne dodatne informacije, ki smo si jih pripravili v sklopu priprave testa. Primer poganjanja testov je viden na sliki 2.

Slika 2: Poganjanje testov znotraj konzole z ogrodjem Playwright.

V kolikor testi niso bili uspešni, lahko pridobimo poročilo z ukazov "show-report", pri čemer dobimo detajlni pogled kaj je šlo narobe. Eno izmed takih poročil je vidno na sliki 3.



Slika 3: Prikaz neuspelega testa.

Iz takega poročila lahko razvijalec izlušči kaj je šlo narobe in kaj mora popraviti oziroma če je tekom razvoja prišlo do neželenih sprememb na nepričakovanih delih aplikacije.

2.2 Struktura ogrodja PTSD

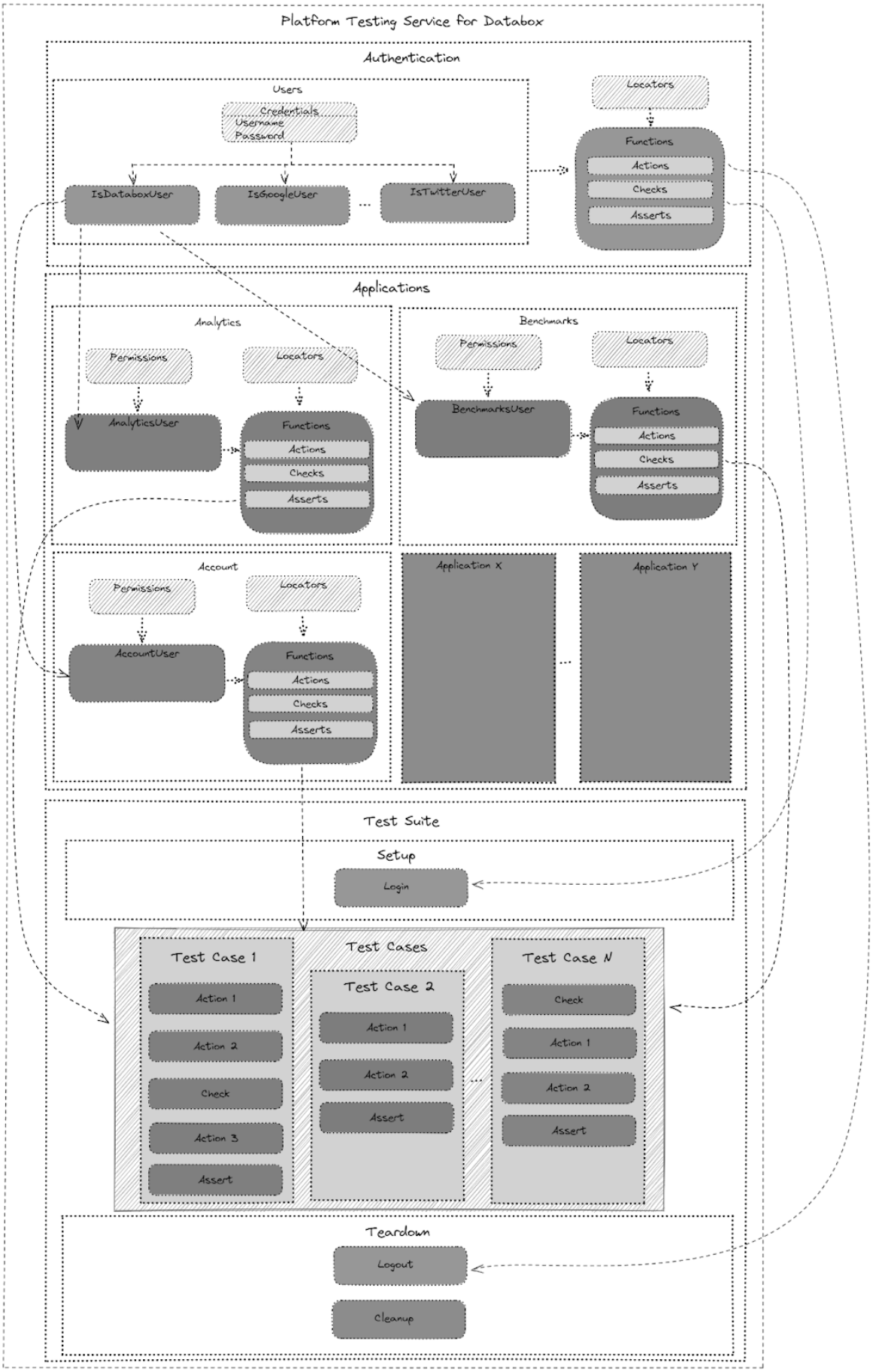
Da bi uporabo ogrodja PTSD čim bolj približali razvijalcem, smo ga zasnovali tako, da je sestavljen iz abstrakcij, ki so že znane iz objektno orientiranega programiranja in so zato blizu razvijalcem in sami po sebi niso revolucionarni. Testi oziroma testni scenariji so sestavljeni s pomočjo razredov, vmesnikov, metod, kot da bi uporabljali kakšno programsko ogrodje, s to izjemo da na koncu, ko kodo poženejo, se izvede celotni test v brskalniku. S pomočjo zastavljene strukture smo poskrbeli da je celotno ogrodje na takšnem nivoju abstrakcije, da se razvijalec ne rabi zavedati specifik samega ogrodja (Playwright). Vse specifike, kot so lokatorji, metode za manipulacije s kontrolami, stranmi, so skrite za primerno poimenovanimi metodami.

Da smo dosegli smiseln nivo abstrakcije, smo najprej ogrodje razdelili na več logičnih enot in sicer na avtentikacijo in na aplikacije. Na tak način smo ločili komponente, ki so skupne našim produktom od specifik, ki jih ima vsaka aplikacija:

- **Komponenta avtentikacije** omogoča uporabo celotnega avtentikacijskega dela naših produktov, od prijave, odjave do funkcionalnosti pozabljenega gesla itd.. V ta namen izpostavlja nabor funkcionalnosti, ki logično uporabniku ogrodja povedo, kaj bodo dosegli s katero akcijo v sistemu. Ta del ogrodja vzpostavi tudi prve vmesnike za nadaljnjo uporabo. Tukaj znotraj so definirani tipi uporabnikov glede na način prijave. Ločimo tako uporabnika, ki se prijavi z uporabniškim imenom in geslom ali preko katerega od ponudnikov identifikacijskih storitev (google, twitter, facebook,...).
- **Komponente aplikacij** predstavljajo posamezne aplikacije znotraj našega ekosistema. Sestavljene so iz funkcionalnosti, ki zagotavljajo nabor akcij v posamezni aplikaciji in manipulacijo z njimi. Vsaka aplikacija ima tako nabor svojih funkcionalnosti in tudi svoj nabor vmesnikov za uporabnike, saj s tem zagotovimo pravilno obnašanje glede na tip uporabnika.

Kot je že lahko zgoraj razvidno so torej končni testni scenariji sestavljeni iz definicije uporabnikov in uporabo funkcionalnosti posameznih aplikacij ter interakciji med uporabniki in funkcionalnostmi, kar je tudi prikazano na sliki 4. Uporabnike in funkcionalnosti znotraj ogrodja definirali sledeče:

- **Uporabniki (angl. Users)** so razredi in vmesniki, ki predstavljajo uporabnike sistema in so na voljo za uporabo v funkcijah. Vsak uporabnik vsebuje nabor dovoljenj in poverilnic, ki vplivajo na tokove in interakcijo s komponentami. Če vedenje komponente temelji na vrsti uporabnika, ki z njo komunicira, je treba uporabnika takega tipa tudi posredovati funkciji. Dovoljenja za uporabnika so vnaprej določena za vsako komponento znotraj aplikacije, zato je za uporabo na voljo več vrst uporabnikov (npr. skrbnik, urednik itd.)
- **Funkcije (angl. Functions)** so metode, ki predstavljajo komponente aplikacije, s katerimi je mogoče komunicirati. Temeljijo na strukturi uporabniškega vmesnika aplikacije. Če vedenje temelji na specifičnem uporabniškem profilu, mora funkcija prejeti ustreznega uporabnika, kar je definirano v sami funkciji. Podobno, če želimo biti v interakciji na določeni komponenti, moramo posredovati identifikator v funkcijo (npr. uporabniško ime, če iščemo uporabnika v iskalnem polju). Glede na način interakcije smo funkcije razdelili na tri skupine:
    - **Akcija (angl. Actions):** To so funkcije, ki predstavljajo dejanja na uporabniškem vmesniku.. Predstavljajo navigacije, klike na gumbe, pisanje besedila v vnosna polja, izbiranje elementov s spustnega seznama itd. Predstavljajo jedro testnega primera.
    - **Preverjanja (angl. Checks):** Funkcije, ki predstavljajo logiko za preverjanje stanja v toku, da se določijo nadaljnji koraki. To je funkcija za nadzor poteka testnega primera. Te funkcije se uporabljajo za ugotavljanje, ali je nekaj v pravilnem stanju za nadaljevanje testa (npr. uporabnik obstaja).
    - **Preizkusi (angl. Asserts):** Funkcija, ki jo je mogoče preizkusiti in zajema logiko enega preizkusa dejanja. Predstavlja pričakovanje, kaj se mora zgoditi. V testnem primeru je treba uporabiti vsaj eno preizkusno metodo. Uspešnost ali neuspeh te metode pa potem določa uspeh testnega primera.



Slika 4: Struktura ogrodja PTSD.

3 Uporaba ogrodja PTS

Kot že omenjeno so, zaradi nivoja abstrakcije, uporabnikom ogrodja (razvijalcem), prikrite specifike testov. Primer take abstrakcije lahko vidimo na sliki 5, kjer smo znotraj razreda *LoginPage* za metodo *actionLoginWithCredentials* skrili specifike ogrodja Playwright, na kak način pridemo do primernih tekstovnih polj in kateri gumb se stisne.

```
export class LoginPage {
    2 usages
    static async actionLoginWithCredentials(page: Page, user: IDataboxUser) : Promise<void> {
        await page.locator(authLoginLocators.INPUT_LOGIN_EMAIL).fill(user.getCredentials().email);
        await page.locator(authLoginLocators.INPUT_LOGIN_PASSWORD).fill(user.getCredentials().password);
        await page.locator(authLoginLocators.BUTTON_SIGN_IN).click();
        await page.waitForLoadState();
    }
```

Slika 5: Prikaz izseka kode iz razreda LoginPage skupaj z metodo actionLoginWithCredentials.

Predpriprava metod in razredov je v domeni testerjev in ekipe za zagotavljanje kakovosti, ki poskrbijo, da se napovedane spremembe uporabniškega vmesnika zrcalijo tudi v samem ogrodju. S tem pristopom smo skrili tudi kompleksnost avtomatskega testiranja v brskalniku pred razvijalci in poskrbeli, da se ne rabijo ubadati z morebitnimi spremembami na uporabniškem vmesniku. To je še posebej dobrodošlo za razvijalce zaledja (backend), ki niso tako dovzetni za spremembe uporabniškega vmesnika.

Nek testni scenarij, lahko razvijalec spiše sam ali uporabi že vnaprej sestavljen testni scenarij. Tak je denimo scenarij uporabljen pri dimnem testiranju, je enostavno berljiv in samo razlagalen kaj se kje dogaja in kaj se bo testiralo. Dokler nismo tega skrili za abstrakcijo pa je bilo to za razvijalca neprimerno težje razbrati. Na sliki 6 vidimo izsek testnega scenarija brez uporabe abstrakcij v primerjavi z izsekom enakovrednega scenarija z uporabo abstrakcij, tako kot ga uporablja razvijalec. Že na prvi pogled je spodnji izsek kode, ki predstavlja naše ogrodje, krajši in bolj berljiv. Kljub temu, da samo ogrodje ne predstavlja nič drugega kot uporabo znanih konceptov objektnega programiranja pa smo dosegli izboljšavo procesa, predvsem za to, ker so testerji že prej vključeni v proces načrtovanja in ogrodje te posledično testi pripravljeno dosti prej kot je to bilo pred vpeljavo tega ogrodja.

```
export const EditorPermission =async (page) => {
    //Login
    await page.goto(Url.BENCHMARKS_PROD);
    await page.locator(BenchmarksLoginLocators.REDIRECT_LOGIN_BUTTON).click();
    await page.locator(BenchmarksLoginLocators.LOGIN_EMAIL_INPUT).fill(BenchmarksLoginCredentials.USER_EDITOR);
    await page.locator(BenchmarksLoginLocators.LOGIN_PASSWORD_INPUT).fill(BenchmarksLoginCredentials.USER_PASSWORD);
    await page.locator(BenchmarksLoginLocators.LOGIN_SIGN_IN_BUTTON).click();
    await expect(page.locator(BenchmarksHomeLocators.NAVIGATION_BAR_HOME)).toBeVisible();

    //Check permissions on account
    await page.locator(BenchmarksHomeLocators.BUTTON_ACCOUNT_OPTIONS).click();
    await page.waitForTimeout(500);
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_PROFILE)).toBeVisible();
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_DATA_MANAGEMENT)).toBeVisible();
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_USER_MANAGEMENT)).toBeVisible();
    await expect(await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_COMPANY_INFORMATION)).toBeHidden();

    //Check permissions on Data source management page
    await page.locator(BenchmarksHomeLocators.ACCOUNT_OPTION_DATA_MANAGEMENT).click();
    await page.locator(DataSourceManagerLocators.BUTTON_EDIT_SOURCE_OPTIONS).click();
    await expect(await page.locator(DataSourceManagerLocators.EDIT_DATA_SOURCE)).toBeVisible();
    await expect(await page.locator(DataSourceManagerLocators.EDIT_DATA_SOURCE_METADATA)).toBeVisible();
    await expect(await page.locator(DataSourceManagerLocators.DELETE_DATA_SOURCE)).toBeVisible();

    //Check permissions on group that was created by Admin role
    await page.goto(Url.BENCHMARKS_PROD + '/groups/joined');
    await page.locator(PrivateGroupLocators.ADMIN_GROUP_CHECK).click();

    async function isFinished(response) {
        return response.url().includes('/benchmarks/') && response.status() === 200 || response.status() === 400
    }
    const response = await page.waitForResponse(async (response) => await isFinished(response));
    console.log(response.status());

    await page.locator(PrivateGroupLocators.GROUP_OPTIONS).click();
    await expect(await page.locator(PrivateGroupLocators.SHARE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.LEAVE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.DELETE_GROUP)).toBeHidden();

    await expect(await page.locator(PrivateGroupLocators.METRICS)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.MEMBERS)).toBeHidden();
    await expect(await page.locator(PrivateGroupLocators.DATA_SOURCES)).toBeVisible();

    await page.locator(PrivateGroupLocators.DATA_SOURCES).click();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_ADD_DATA_SOURCES)).toBeEnabled();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_REMOVE_DATA_SOURCE)).toBeVisible();

    await page.locator(PrivateGroupLocators.EDIT_GROUP).click();
    await page.locator(PrivateGroupLocators.SAVE_GROUP).click();
    await expect (await page.locator(BenchmarksHomeLocators.INFO_TOAST)).toBeVisible();

    //Check permissions on group that was created by Editor role
    await page.goto(Url.BENCHMARKS_PROD + '/groups/joined');
    await page.locator(PrivateGroupLocators.EDITOR_GROUP_CHECK).click();

    async function isFinished1(response) {
        return response.url().includes('/benchmarks/') && response.status() === 200 || response.status() === 400
    }
    const response1 = await page.waitForResponse(async (response) => await isFinished1(response));
    console.log(response1.status());

    await page.locator(PrivateGroupLocators.GROUP_OPTIONS).click();
    await expect(await page.locator(PrivateGroupLocators.SHARE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.LEAVE_GROUP)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.DELETE_GROUP)).toBeVisible();

    await expect(await page.locator(PrivateGroupLocators.METRICS)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.MEMBERS)).toBeVisible();
    await expect(await page.locator(PrivateGroupLocators.DATA_SOURCES)).toBeVisible();

    await page.locator(PrivateGroupLocators.DATA_SOURCES).click();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_ADD_DATA_SOURCES)).toBeEnabled();
    await expect(await page.locator(PrivateGroupLocators.BUTTON_REMOVE_DATA_SOURCE)).toBeVisible();

    await page.locator(PrivateGroupLocators.EDIT_GROUP).click();
    await page.locator(PrivateGroupLocators.SAVE_GROUP).click();
    await expect (await page.locator(BenchmarksHomeLocators.INFO_TOAST)).toBeVisible();
}
```

```
export const testPermissions =async (page: Page, user: User.IBenchmarkUser) => {
    await Benchmarks.LoginPage.actionLoginWithCredentials(page, user);
    await Benchmarks.ProfileMenu.assertPermissions(page, user);
    if(user instanceof User.Admin || user instanceof User.Editor){
        await Benchmarks.SourcesPage.assertPermissions(page, user);
        await Benchmarks.GroupsJoinedPage.assertAdminPermissionsInGroup(page, user, 'Admin Permission Group');
        await Benchmarks.GroupsJoinedPage.actionEditPrivateGroup(page, 'Admin Permission Group');
        await Benchmarks.GroupsJoinedPage.assertEditorPermissionsInGroup(page, user , 'Editor Permission Group');
        await Benchmarks.GroupsJoinedPage.actionEditPrivateGroup(page, 'Editor Permission Group');
        await Benchmarks.GroupsJoinedPage.assertUserPermissionsInGroup(page, user, 'User Permission Group');

    }
    if(user instanceof User.User){
        await Benchmarks.SourcesPage.assertPermissions(page, user);
        await Benchmarks.GroupsJoinedPage.assertUserPermissionsInGroup(page, user, 'User Permission Group');

    }
    if(user instanceof User.Viewer){
        await Benchmarks.GroupsJoinedPage.assertAdminPermissionsInGroup(page, user, 'User Permission Group');
    }
}
```

Slika 6: Primerjava testnih scenarijev napisan v čistem Playwrightu in v našem ogrodju po abstrakciji.

Poleg razvoja samega ogrodja smo čas posvetili tudi vpeljavi ogrodja v sam proces neprekinjene integracije (continuous integration). Vnaprej pripravljeni scenariji poskrbijo, da se pred uvedbo novih funkcionalnostih avtomatično zaženejo čim bolj podrobni testi, ki so nastali pod okriljem ekipe za zagotavljanje kakovosti. Tudi tukaj se je izkazalo ogrodje PTSD več kot primerno, saj je pisanje scenarijev kot zlaganje sestavljanke in lahko na enostaven način veliko programske kode uporabimo ponovno, kar pa ni presenetljivo, glede na to da so zadaj koncepti objektnega programiranja. Pri vpeljavi nove aplikacije v ekosistem podjetja smo lahko vse kar se tiče avtentikacije v aplikacijo preprosto ponovno uporabili iz prej omenjene komponente avtentikacije. Tudi pri testnih scenarijih, ki so zahtevali takšno sosledje dogodkov, da je bilo potrebno prehajanje iz ene v drugo aplikacijo, smo lahko enostavno ponovno uporabili že obstoječe funkcije.

4 Zaključek

Z vpeljavo ogrodja PTSD v podjetju smo razvijalcem ponudili možnost pisanja celovitih in dimnih testov na enak način kot bi pisali teste enot. Kakovost rešitev preden pridejo v roke ekipi za zagotavljanje kakovosti se je v ekipah, kjer smo to ogrodje vpeljali občutno dvignila. Zaznali smo več kot 50% zmanjšanje zaznanih napak, ki jih najde ekipa za zagotavljanje kakovosti. Odziv razvijalcev je prav tako bil pozitiven, saj se je zmanjšal čas pri pridobivanju prvih informacij glede testiranja. S tem ko so sami dobili možnost, da lahko enostavno in hitro poženejo celovite teste ter si sami sestavijo testni scenarij brez potrebnega znanja o testnih ogrodij se je povečala tudi njihova samozavest pri prehodu na nadaljnje faze razvojnega cikla.

Potrebno je pa tudi omeniti še eno pozitivno stran, ki jo je prineslo ogrodje PTSD. Sodelovanje med ekipo za zagotavljanje kakovosti, razvijalci in projektnimi vodji se je okrepilo, saj je prišlo do pretoka informacij do vseh akterjev prej že v fazi analize in načrtovanja. Ekipa za zagotavljanje kakovosti je zaradi zagotavljanja skladnosti ogrodja z želenimi načrti tako že vnaprej sodelovala pri načrtovanju in nudila povratne informacije glede na vpliv na celotno rešitev in morebitne posledice, ki jih razvijalci in projektni vodje niso predvideli.

Literatura

[1] https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/, SDLC (Software Development Life Cycle) Phases, Process, Models, obiskano 4.7.2023

[2] https://katalon.com/, Katalon, obiskano 4.7.2023

[3] https://www.testim.io/blog/puppeteer-selenium-playwright-cypress-how-to-choose/ , Puppeteer, Selenium, Playwright, Cypress – how to choose?, obiskano 5.7.2023

[4] https://www.browserstack.com/guide/cypress-vs-selenium-vs-playwright-vs-puppeteer, Cypress vs Selenium vs Playwright vs Puppeteer: Core Differences, obiskano 5.7.2023

[5] https://playwright.dev/, Playwright, obiskano 5.7.2023