OULUN AMMATTIKORKEAKOULU

Joni Virvalo

WEB-SOVELLUSTEN TESTIAUTOMAATIO PYTHONILLA: VERTAILUSSA SELENIUM JA PLAYWRIGHT

WEB-SOVELLUSTEN TESTIAUTOMAATIO PYTHONILLA: VERTAILUSSA SELENIUM JA PLAYWRIGHT

Joni Virvalo
Opinnäytetyö
Syksy 2023
Tradenomi, Tietojenkäsittely
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma

Tekijä: Joni Virvalo
Opinnäytetyön nimi: Web-sovellusten testiautomaatio Pythonilla: vertailussa Selenium ja Playwright
Työn ohjaaja: Ilpo Virtanen
Työn valmistumislukukausi ja -vuosi: Syksy 2023 Sivumäärä: 37 + 11 liitesivua

---

Tässä opinnäytetyössä tutkittiin testiautomaation roolia ohjelmistotestauksessa. Testiautomaatio on menetelmä, joka käyttää testausohjelmistoa testien suorittamiseen ja tulosten vertaamiseen ennustettuihin tuloksiin. Sen avulla voidaan automatisoida toistuvia ja monimutkaisia testejä, mikä tehostaa ohjelmiston laadun varmistamista.

Opinnäytetyössä keskityttiin tutkimaan kahta tunnettua testausohjelmistoa, Seleniumia ja Playwrightia, ja niiden käyttöä web-sovellusten automatisoidussa testauksessa. Testausohjelmistoilla suoritettiin testejä, jotka perustuivat AAA- ja sivuobjektimalleihin. Pythonin valinta ensisijaiseksi ohjelmointikieleksi perusteltiin sen laadukkailla kolmannen osapuolen kirjastoilla, jotka tarjoavat monipuolisia ominaisuuksia testitapauksien toteuttamiseen.

Asiasanat: Python, Pytest, Selenium, SeleniumBase, Playwright

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Business Information Systems

Author: Joni Virvalo
Title of thesis: Automated Testing of Web Applications with Python: A Comparison of Selenium and Playwright
Supervisor: Ilpo Virtanen
Term and year when the thesis was submitted: Autumn 2023
Number of pages: 37 + 11 appendices

This thesis explores the role of test automation in software testing. Test automation is a method that utilizes testing software to execute tests and compare results to expected outcomes. It allows for the automation of repetitive and complex tests, thereby enhancing the assurance of software quality.

The thesis primarily focuses on the examination of two well-established testing tools, Selenium and Playwright, and their application in automating the testing of web applications. Tests were conducted based on the AAA (Arrange, Act, Assert) and Page Object Model (POM) patterns. The choice of Python as the primary programming language was motivated by its rich ecosystem of third-party libraries, which facilitate diverse test case implementations.

Keywords: Python, Pytest, Selenium, SeleniumBase, Playwright

SISÄLLYS

1 JOHDANTO

Testiautomaatio on ohjelmistotestauksen menetelmä, jossa käytetään erillistä testausohjelmistoa testien suorittamiseen ja tulosten vertaamiseen ennustettuihin tuloksiin. Tämä mahdollistaa toistuvien ja monimutkaisten testien automatisoinnin, mikä säästää aikaa ja resursseja ohjelmiston laadun varmistamisessa.

Automatisoitujen testien suunnitteluvaiheessa määritellään, mitä osa-alueita halutaan testata ja millaisia testitapauksia tarvitaan näiden osa-alueiden kattavaan testaamiseen. Suunnittelun perusteella valitaan testausympäristöön sopiva testausohjelmisto, jonka avulla suoritetaan testitapaukset. Nämä testitapaukset tarkastavat ohjelmiston toiminnallisuudet ja varmistavat, että ne vastaavat odotettuja vaatimuksia ja toimivat oikein.

Opinnäytetyön tavoitteena on tutkia kahden tunnetun testausohjelmiston, Seleniumin ja Playwrightin, käyttöä web-sovellusten automatisoidussa testauksessa. Testausohjelmistoilla suoritetaan automatisoituja testejä käyttämällä AAAC- ja sivuobjektimallia. Python on valittu opinnäytetyöhön sen laadukkaiden kolmannen osapuolen kirjastojen ansiosta. Nämä kirjastot tarjoavat ominaisuuksia, joita voi laajasti hyödyntää erilaisissa testitapauksissa ja tehostavat testausohjelmistojen käyttöä.

Opinnäytetyön aihetta on rajattu Pythonin osalta käymällä läpi keskeiset ominaisuudet testitapausten luomiseen. Testitapaukset toteutetaan mustalaatikkotesteinä, joissa tarkastellaan web-sovellusten ominaisuuksia ilman yksityiskohtaista tietoa ohjelmiston sisäisestä rakenteesta. Mustalaatikkotestaus tarjoaa arvokasta tietoa testausohjelmistojen soveltuvuudesta web-sovellusten loppupään testaukseen.

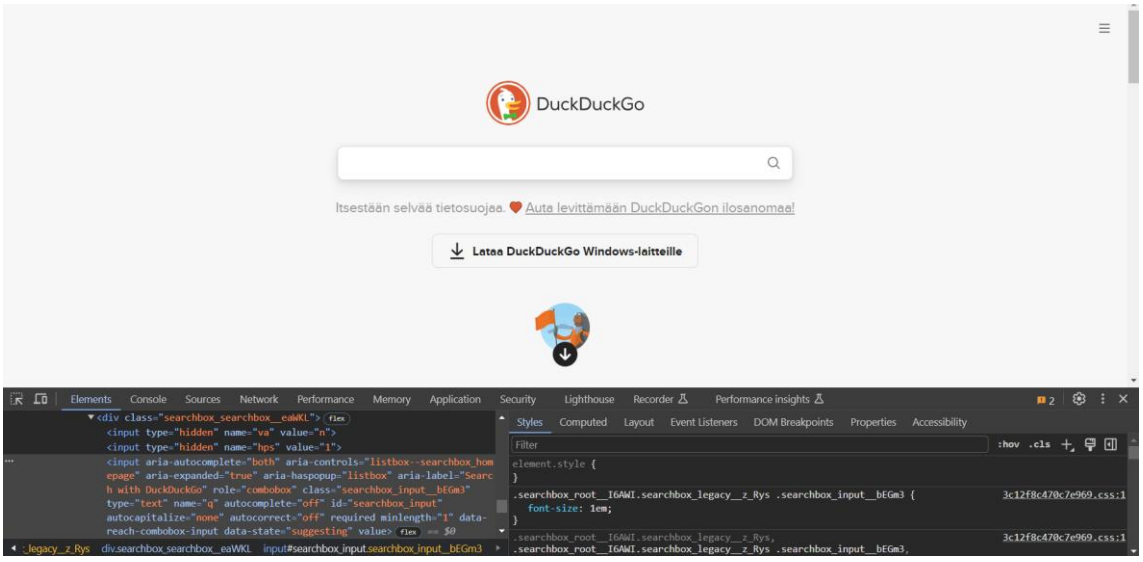
.

2 WEB-SOVELLUKSEN TESTIAUTOMAATION EDUT

Web-sovellus on ohjelmisto, joka sijaitsee etäpalvelimella ja johon päästään käsiksi web-selaimen välityksellä. Tällaisia sovelluksia voivat olla esimerkiksi sähköpostipalvelut, sosiaalisen median alustat ja verkkokaupat. Web-sovelluksia ei tarvitse erikseen ladata laitteelle, koska ne toimivat internet-yhteyden avulla. Asiakas pääsee käyttämään web-sovellusta web-selaimen avulla, kuten Googlen Chromella, Mozillan Firefoxilla ja Applen Safarilla.

Web-sovelluksen toiminta edellyttää yleisesti verkkopalvelinta, sovelluspalvelinta ja tietokantaa. Verkkopalvelimet käsittelevät käyttäjien lähettämiä pyyntöjä, kun taas sovelluspalvelin suorittaa pyydetyt tehtävät. Tietokanta tallentaa tarvittavat tiedot.

Käyttöliittymä käsittää web-sovelluksen näkyvän osan käyttäjälle. Käyttöliittymä sisältää erilaisia elementtejä, kuten sivun rakenteen (HTML), visuaalisen ulkoasun ja tyylin (CSS) sekä toiminnallisuudet ja interaktiiviset toiminnot (JavaScript). Automaattisessa käyttöliittymätestauksessa keskitytään pääasiassa CSS-valitsimien paikantamiseen, jotta haluttuun elementtiin voidaan vaikuttaa. Jokainen elementti, johon halutaan vaikuttaa, täytyy tunnistaa yksilöllisesti. Valitsin voi perustua elementin ominaisuuksiin tai sen sijaintiin DOM-hierarkiassa. (Techtarget 2023.)



Kuva 1. Havainnollistava kuva DuckDuckGo hakukoneesta. Valitsemalla CSS-valitsimen name="q" testausohjelma paikantaa hakukentän.

Automatisoidut testit ovat nopeammin suoritettavia ja tehokkaampia kuin manuaaliset testit. Manuaalisten testien suorittaminen kiireisinä aikoina rajoittaa mahdollisuutta käyttää aikaa muihin keskeisiin tehtäviin. Testausohjelmistot lyhentävät suoritusaikaa, joka vapautuu enemmän aikaa testien suunnitteluun ja uusien testien kehittämiseen (Ron Patton 2004, sivu 232.)

Yksi käytännöllinen tapa nopeuttaa web-sovellusten testien suorittamista on automatisoida ne hyödyntämällä päätöntä selainta. Päätöntä selainta ohjataan taustalla samalla tavalla kuin tavallista selainta, mutta suorituksesta ei näe mitään näytöllä päättömän selaintestin aikana. Vaikka selainten toteutukset voivat vaihdella, päätön selain kykenee luotettavasti analysoimaan ja tulkkaamaan verkkosivuja. (BrowserStack 2023.)

Manuaalisten testien suorittamisen aikana pitkä ja toistuva testaus saattaa heikentää keskittymiskykyämme, mikä voi johtaa virheisiin ja puutteellisiin tuloksiin. Testaajan subjektiivisuus ja vaihtuminen voivat myös vaikuttaa testien luotettavuuteen.

Testityökalut eivät kärsi väsymyksestä eivätkä menetä testaajan tavoin tarkkaavaisuuttaan. Ne suorittavat saman testin samalla tavalla joka kerran, ja niiden tarkkaavaisuus pysyy vakaana. Tämä tarkoittaa, että testitulokset ovat yhdenmukaisempia ja luotettavampia manuaaliseen testaukseen verrattuna. Testityökalut voivat myös havaita pienimmätkin erot odotettujen ja todellisten tulosten välillä, mikä tekee niistä erinomaisen vaihtoehdon varmistamaan ohjelmiston tarkkuuden ja suorituskyvyn.

Lisäksi automaatiotestaus vähentää resurssien tarvetta. Se tarkoittaa käytännössä työmäärän vähentämistä, koska automaattisia testejä voidaan suorittaa samanaikaisesti eri selaimilla ja emulaattoreilla, jolloin web-sovelluksen toimintaa voidaan testata eri laitteilla ja näytön erilaisilla resoluutiolla. (Ron Patton 2004, sivu 232.)

3 MUSTALAATIKKOTESTAUS

Mustalaatikkotestaus on ohjelmistotestauksen menetelmä, jossa testaaja tarkastelee testattavaa ohjelmistoa ulkopuolisena havainnoijana ilman tietoa sen sisäisestä rakenteesta. Testaajan tehtävänä on syöttää tietoa ohjelmistolle ja arvioida sen tuottamia tuloksia.

Toiminnallinen testaus on mustalaatikkotestauksen osa-alue, joka keskittyy varmistamaan, että ohjelmisto toimii odotetulla tavalla ja täyttää asetetut toiminnalliset vaatimukset. Tämä testausmenetelmä arvioi, kuinka hyvin ohjelmisto suorittaa tiettyjä toimintoja ja varmistaa, että se noudattaa sovelluksen toiminnallisia määrityksiä. Toiminnallinen testaus voi kattaa useita eri testausalueita, kuten käyttöliittymän, rajapintojen, tietokannan ja palvelimen testauksen.

Mustalaatikkotestaus voi myös olla ei-toiminnallista testausta, joka ei keskity suoraan sovelluksen toiminnallisiin ominaisuuksiin vaan arvioi sen suorituskykyä, luotettavuutta, skaalautuvuutta ja muita ei-toiminnallisia tekijöitä. Tavoitteena on varmistaa, että sovellus toimii tehokkaasti ja luotettavasti erilaisissa olosuhteissa ja kuormituksissa ilman suorien toiminnallisuuksien arviointia. (Guru99 2023.)

4 PYTHON

Python on korkean tason avoimen lähdekoodin ohjelmointikieli, jota voi käyttää monipuolisesti erilaisiin ohjelmointitehtäviin. Kielen loi Guido Van Rossum 1990-luvulla, ja sen suosio on kasvanut tasaisesti. Nimensä Python on saanut Monty Pythonin lentävän sirkuksen komediaohjelman inspiroimana.

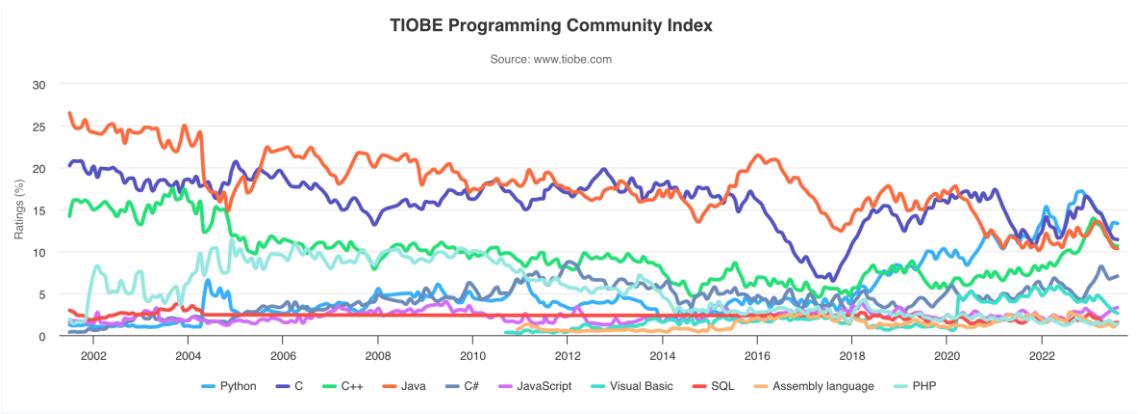
Python on tulkattu ohjelmointikieli, mikä tarkoittaa, että se muunnetaan automaattisesti tavukoodiksi ennen suoritusta. Tavukoodi tallennetaan yleensä automaattisesti levylle, joten erillistä kääntämistä ei tarvitse toistaa, ellei alkuperäistä lähdekoodia muuteta. Python on dynaamisesti tyypitetty kieli ja sisältää oliopohjaisia ominaisuuksia ja rakenteita.

Yksi Pythonin merkittävimmistä piirteistä on sen poikkeuksellinen tapa käyttää sisennystä lohkojen määrittämiseen ja erottamiseen. Tämä eroaa monista muista ohjelmointikielistä, joissa käytetään lohkosulkeita. Pythonissa lohkot erotellaan sen sijaan sisentämällä koodirivejä. Käytännössä tämä tarkoittaa, että ohjelmassa käytetyt lohkot määritellään sisentämällä niihin kuuluvat koodirivit tiettyyn tasoon. Esimerkiksi kuvan 2 skriptin voi kirjoittaa ja suorittaa Python-ympäristössä. (Python Programming 2022.)

```
import random
import string

def generate_password(length=12):

    characters = string.ascii_letters + string.digits +
string.punctuation

    password = ''.join(random.choice(characters) for _ in
range(length))

    return password

password = generate_password(16)
print("Random Password:", password)
```

Kuva 2. Pythonin skripti luo 16 merkin mittaisen satunnaisen salasanan, joka sisältää ASCII-merkkejä käyttäen Pythonin sisäänrakennettua merkkijonometodia.

Pythonin suosio on tasaisesti kasvanut lähes sen koko olemassaolon ajan, mutta erityisesti vuoden 2018 jälkeen suosion kasvuvauhti on kasvanut merkittävästi.



Kuva 3. TIOBEn akitiivisesti päivittyvä index suosituista ohjelmointikielistä.

Pythonin suosion kasvun taustalla on sen helppokäyttöisyys, laaja yhteisö, akateeminen käyttö ja kysyntä yritysmaailmassa. Pythonin syntaksi mahdollistaa tehokkaan koodin kirjoittamisen, ja sillä on laaja käyttäjäyhteisö, joka tarjoaa tukea ja resursseja. Python on myös suosittu akateemisissa piireissä, ja sillä on monipuolista käyttöä STEM-aloilla. Yritysmaailmassa Pythonin osaaminen on arvostettua, ja se oli vuonna 2022 kolmanneksi eniten kysytty ohjelmointikieli rekrytoijien keskuudessa. (Github Blog 2023.)

Pythonin sisältää monipuolisen standardikirjaston ja kolmannen osapuolen kirjaston laajalla moduulien ja pakettien tarjonnalla. Kolmannen osapuolen paketit löytyvät PyPI:stä (The Python Package Index). Jokaisessa Pythonin projektissa on suositeltavaa asentaa kolmannen osapuolen paketit Pythonin standardikirjastoon kuuluvan venv-moduulin sisällä, joka on kevyt virtuaaliympäristö. Tämä mahdollistaa Python-pakettien hallinnan paikallisesti projektikohtaisesti sen sijaan, että ne olisivat asennettuina globaalisti koko käyttöjärjestelmälle. (Python Venv 2023.)

4.1 Pytest

Holger Krekel julkaisi Githubissa vuonna 2007 Pytestin tarjotakseen paremman vaihtoehdon Pythonin sisäänrakennettuun testikehykseen. Pytestin suosio on kasvanut ajan myötä, ja se on nykyään yksi Pythonin suosituimmista avoimen lähdekoodin kirjastoista. (Pytest History 2015.)

Pytestin yksi keskeisimmistä ominaisuuksista on fikstuurien (fixture) käyttö. Fikstuurit ovat dekoraattoreita eli funktioon kiinnitettäviä resurssien hallintaan ja siivoamiseen tarkoitettuja toimintoja, jotka mahdollistavat testiskriptin resurssien eristämisen tai jakamisen. Toinen keskeinen ominaisuus on metatiedon kiinnittäminen testimoduulin funktioihin, mikä mahdollistaa monipuolisten testien suorittamisen. (Pytest Explanation Fixtures 2015.)

Pytestillä on myös tehokas testien etsintäjärjestelmä, joka automatisoi testien etsimisen ja suorittamisen. Kehys tarjoaa laajan liitännäisjärjestelmän, jonka avulla testisuorituksia voidaan laajentaa ja mukauttaa tarpeen mukaan. Tämä mahdollistaa ominaisuuksien lisäämisen, kuten testitapausten hallinnan, testikattavuuden raportoinnin ja testien rinnakkaisen suorittamisen. (Pytest Invoke 2015.)

4.2 Testin rakenne

Pytestin dokumentissa “Anatomy of a Test" ohjeistetaan lyhyesti suunnittelemallista Arrange-Act-Assert-Cleanup, joka tulee sanoista valmistele, toimi, varmista ja siivoa. Tämä suunnittelumalli on siivousta lukuun ottamatta tunnettu yksikkötestauksesta. Valmistelu on testin konteksti, jossa valmistellaan tarvittavat resurssit ympäristöön. Toimi on yksittäinen toimenpide, joka käynnistää testattavan toiminnallisuuden ja muuttaa järjestelmän tilaa. Varmistus on muuttuneen tilan vertaaminen odotettuun lopputulokseen. Siivous on testin jälkeinen resurssien palauttaminen, jotta testi ei vaikuta muihin testisarjan suorituksiin. Siivous mahdollistaa laajempien testisarjojen suunnittelun. (Pytest Anatomy of a Test 2015; Freecontent 2020.)

```
import pytest

@pytest.fixture
def initial_value():
    return 1

def test_one_plus_two(initial_value):

    # Arrange
    one = initial_value

    # Act
    result = one + 2

    # Assert
    assert result == 3
```

Kuva 4. Havainnollistava kuva, jossa fikstuuri valmistelee funktion resurssit.

Kuvassa 4 on esimerkki siitä, miten hyödynnetään fikstuuria testin alustamiseen. Fikstuurit on suositeltavaa lisätä omaan Pythonin konfigurointimoduuliin. Laajempi esimerkki, jossa fikstuurit jakavat resursseja, löytyy liitteestä 1. Liitteen 1 rajapintatestissä yield-lause suorittaa testin jälkeisen siivouksen, joka vapauttaa resurssit ja palauttaa testiympäristön alkuperäiseen tilaansa.

Fikstuurin laajuus (scope) on parametri, joka määrittää sen, kuinka kauan fikstuurin tila pysyy voimassa. Tilan laajuudet muodostavat hierarkian korkeimmasta alimpaan: istunto, paketti, moduuli, luokka ja funktio. Fikstuurien suunnittelussa on otettava huomioon, että ne tukevat testien itsenäistä suorittamista ja tuottavat luotettavia tuloksia. (Pytest fixture 2020.)

4.3 Pytest-funktion merkintä

Pytestin sisäänrakennettu dekoraattori @pytest.mark.parametrize mahdollistaa testitapauksen funktion merkitsemisen niin, että testi suoritetaan useilla eri syötteillä. (Pytest parametrize 2015.)

```
import pytest


numbers = [
    (3, 3, 9),
    (5, -5, -25),
    (33, 3, 99)
]

# Parametrit 'a', 'b', ja 'result' saavat arvonsa numbers-
listasta
@pytest.mark.parametrize('a, b, result', numbers)
def test_multiplication(a, b, result):
    assert a * b == result
```

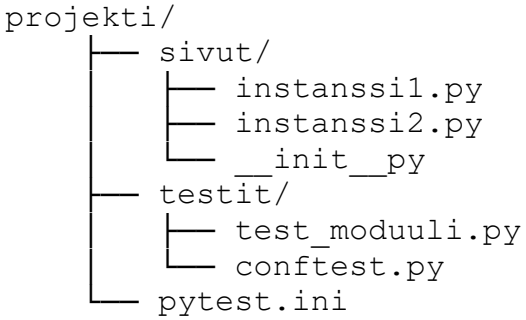
Kuva 5. Havainnollistava kuva Pytestin @pytest.mark.parametrize dekoraattorin käytöstä.

Pytestillä voidaan luoda myös omia kustomoituja merkintöjä tai käyttää merkintöjä testiluokkiin tai moduuleihin. Liitännäisohjelmat voivat käyttää näitä merkkejä, ja niitä käytetään yleisesti myös testien valitsemiseen komentorivillä. Opinnäytetyössä käytetään testin parametrisointiin pytest-xdist-liitännäisohjelmaa, jossa hyödynnetään @pytest.mark.parametrize-dekoraattoria.

Liitteessä 4 ja 5 on suunniteltu testitapaus, jossa voidaan lisätä product-taulukkoon merkkijonoja, joita haetaan verkkokaupasta rinnakkain suoritettavissa testeissä. Liitteessä 3 on käytetty pytest-repeat-liitännäisohjelmaa, joka mahdollistaa testin suorittamisen useassa sarjassa peräkkäin. (Pytest how to mark 2015.)

4.4 Hakemistorakenne

Hakemistorakenteella ja testien nimeämisellä on merkitystä pytest-projektissa, koska ne vaikuttavat siihen, miten pytest tunnistaa ja suorittaa moduulit. Hyvin suunniteltu hakemistorakenne auttaa erottamaan olioluokkien moduulit sekä testi- ja konfigurointimoduulin toisistaan. Pytest käyttää nimeämisstandardia, jossa testimoduulin nimi alkaa sanalla "test_" tai "_test"



Kuva 6. Havainnollistava kuva Pytestillä luodusta hakemistorakenteesta.

Yleinen käytäntö on sijoittaa kaikki moduulit projektin omaan hakemistoon ja luoda erillinen kansio testimoduulille. Testimoduulin kansioon ei suositella lisäämään Python-paketteja, eli __init__.py -tiedosto lisätään hakemistoon erikseen vain testimoduulin kutsuja varten.

Fikstuurit sijoitetaan conftest.py-moduuliin, mikä mahdollistaa resurssien jakamisen kaikkien projektin testien kesken. Pytest tunnistaa automaattisesti konfigurointiin tarkoitetun moduulin. Juuritiedostossa oleva pytest.ini ei ole pakollinen, mutta se mahdollistaa yhden keskitetyn paikan projektin testiympäristön mukauttamiseksi. Yksi käytännöllinen tapa on lisätä juuritiedostoon suurin osa komennosta, jotta komennon voi suorittaa pelkästään kirjoittamalla pytest. (Pytest good practice 2015.)

5 PLAYWRIGHT

Playwright on Microsoftin kehittämä avoimen lähdekoodin kirjasto ja kehys, joka on suunniteltu web-sovellusten automaattiseen testaamiseen ja selainten automatisointiin. Se sai alkunsa Googlen julkaisemasta avoimen lähdekoodin automaatiotyökalusta nimeltä Puppeteer. Vuonna 2019 osa Puppeteerin kehittäjistä siirtyi Microsoftille kehittämään Playwrightia, joka julkaistiin vuonna 2020. (Playwright Tech 2020.)

```python
# Tuodaan tarvittavat Playwrightin toiminnot
from playwright.sync_api import sync_playwright

# Alustetaan Playwright-ympäristö
with sync_playwright() as p:
    # Käynnistetään selain
    browser = p.chromium.launch()

    # Luodaan uusi konteksti
    context = browser.new_context()

    # Käytetään kontekstia sivun luomiseen
    page = context.new_page()

    # Siirrytään Pythonin verkkosivulle
    page.goto("https://www.python.org/")

    # Haetaan sivun otsikko ja tulostetaan se konsoliin
    title = page.title()
    print(f"Sivun otsikko: {title}")

    # Suljetaan selain
    browser.close()
```

Kuva 7. Esimerkki Playwrightin käytöstä Pythonilla ilman Pytest-kehystä

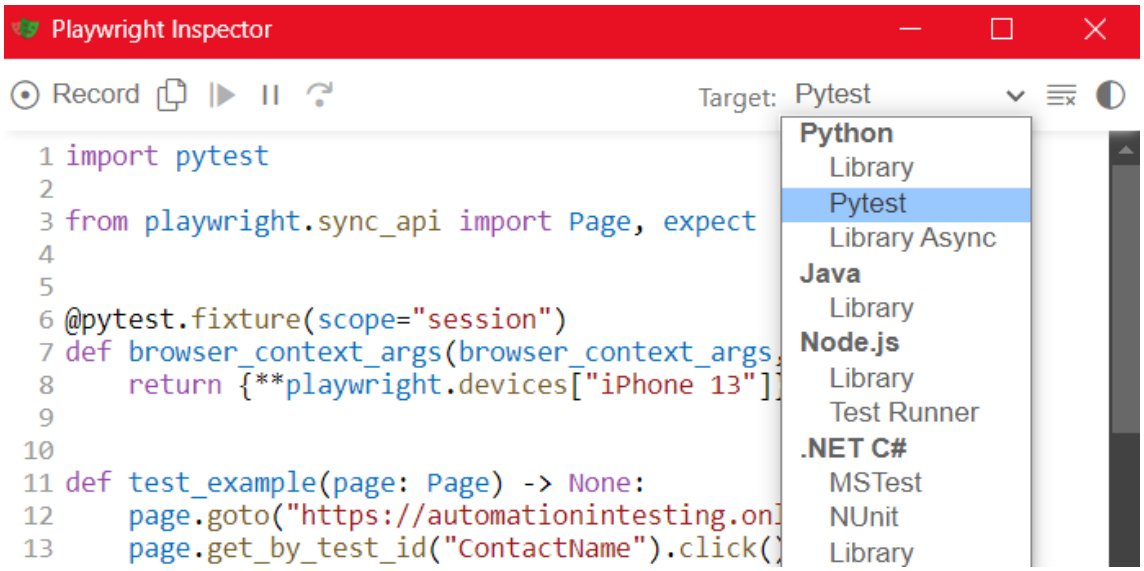
Kuvassa 7 on Playwrightilla toteutettu synkronisesti suoritettava testiskripti. Tässä esimerkissä määritellään selainohjelma ja luodaan testiskriptille konteksti selaininstanssiin tulostamaan Pythonin etusivun otsikko.

Konteksti luodaan käyttämällä new_context() -metodia, joka kuuluu selaimen (browser) objektiin. Konteksti toimii kuin erillinen selainikkuna, mutta sille voidaan asettaa omia erityisiä asetuksia, kuten mobiililaitteen määrittely testin suorittamista varten, mitä havainnolistetaan kuvassa 10. Kontekstin sisällä luodaan uusi eristetty selainikkuna käyttämällä new_page() -metodia. Kontekstin käyttö ei ole tässä esimerkissä välttämätön, mutta kontekstin käyttö voi olla hyödyllistä monimutkaisemmissa testiskripteissä.

```
from playwright.sync_api import Page

def test_page_title(page: Page):
    # Siirrytään Pythonin verkkosivulle
    page.goto("https://www.python.org/")

    # Haetaan sivun otsikko
    title = page.title()

    # Tarkistetaan, että "Python" löytyy otsikosta
    assert "Python" in title, f"Odotettiin 'Python' otsikossa:
{title}"
```

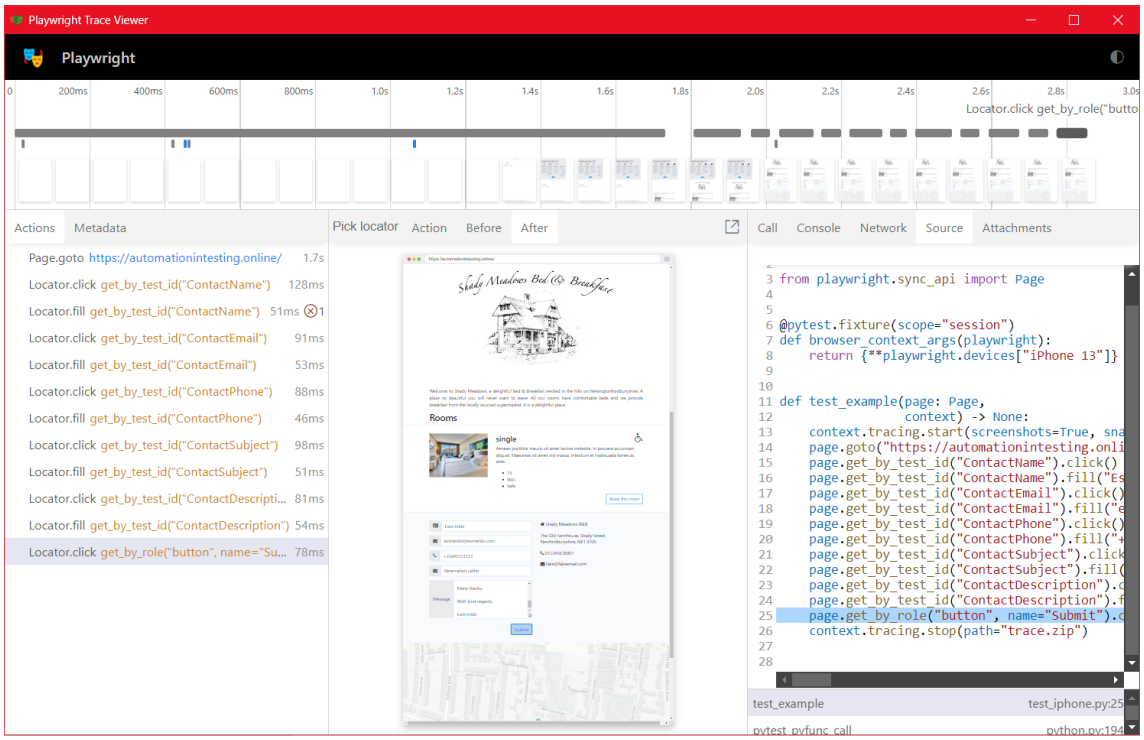
Kuva 8. Playwright Pytest-kehyksessä. Page on luokka, joka edustaa selaininstanssia. Page-luokkaa käytetään fikstuurina resurssien esivalmisteluun ja puhdistamiseen.

Microsoft on kehittänyt Playwrightille Pytest-kehyksen integraatioon pytest-playwright-lisäosan, jota suositellaan käyttämään Playwrightin päästä päähän -testien toteuttamisessa. Playwrightille on luotu Pytest-kehyksessä omia komentoriviparametreja ja fikstuureja. Liitteessä 4 on toteutettu sivuobjektimalli, jossa Page-oliota kutsutaan sivuobjekteista testimoduuliin. (Playwright Pytest 2023.)



Kuva 9. Playwright Inspector -ikkuna, johon on tallennettu automationintesting.online -sivulla toteutettu manuaalinen sivun elementtien paikantaminen mobiilinäkymällä.

Playwright sisältää monipuolisen kokoelman erilaisia työkaluja testitapausten toteuttamiseen ja tarkistamiseen, kuten Codegenin, Playwright Inspectorin ja Trace Viewerin. Codegen luo testejä tallentamalla manuaalisesti suoritetun toiminnan verkkosivulta. Codegen paikantaa automaattisesti käyttöliittymän CSS-valitsimet, jotka voidaan nauhoittaa Playwright Inspector -ikkunaan. Playwright parantaa automaattisesti paikantimen laatua havaitessaan verkkosivun elementissä useita tyylitiedostoja kuten kuvassa 11. (Playwright Codegen 2023.)



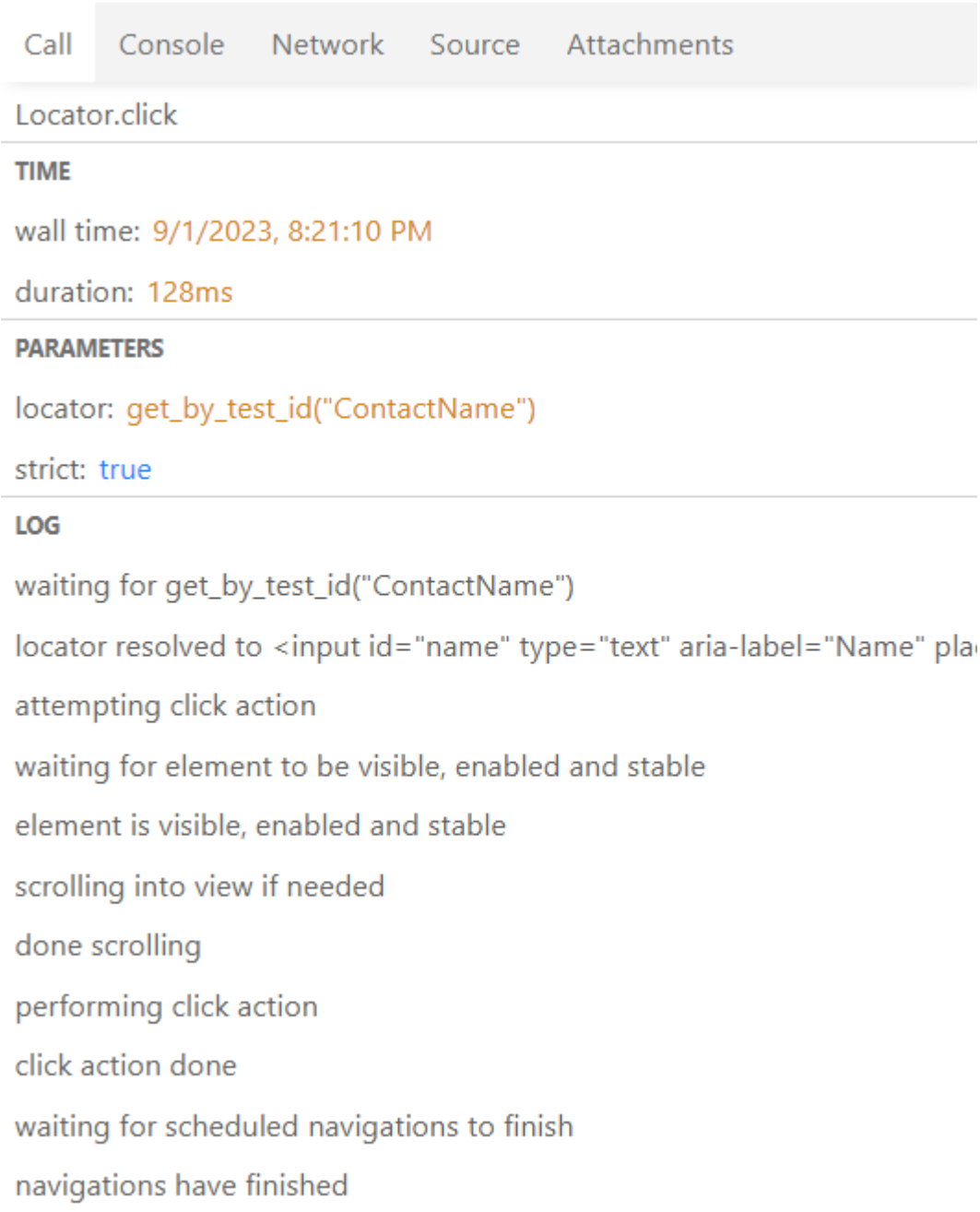
Kuva 10. Trace Viewer -ikkunnassa ruutukaappaus on ylhäällä näkyvä aikajana, tilannekuvat keskellä testitapauksen käyttöliittymän yläpuolella ja lähdekoodi oikealla.

Playwright Trace Viewer on graafinen työkalu, joka tallentaa ja visualisoi Playwrightin suorittaman skriptin jäljitettävät tapahtumat verkkosivun käyttöliittymässä sekä taustalla tapahtuvat pyynnöt palvelimelle. Kuvassa 10 näkyvä skripti on verkkosivustolla suoritettu testiskripti. Testiskriptissä on määritelty konteksti jäljitykselle ja jäljityksen asetuksille, jotka näkyvät kuvassa 11.

```
context.tracing.start(screenshots=True, snapshots=True, sources=True)
```

Kuva 11. Suomennettuna screenshots, snapshots, ja sources ovat ruutukaappaukset, tilannekuvat ja lähdekoodi. Nämä lisäykset eivät ole pakollisia Trace Viewerin käytön kontekstissa, mutta ne tuovat arvokasta tietoa.

Ruutukaappaukset tallentavat testin visuaalisen suorituksen aikajanalle. Ne tallentavat suorituksen alusta asti ja näyttävät kaikki testin aikana tapahtuneet toiminnot. Tilannekuvat puolestaan esittävät jokaisen skriptin suorituksen erikseen. Ne sisältävät kolme osaa: tilanteen ennen elementin käsittelyä, elementin käsittelyn tilanteen ja tilanteen käsittelyn jälkeen. Trace Viewer tallentaa jokaisen sivuelementin sijainnin Pick Locator -valikkoon. (Playwright Trace Viewer 2023.)



Kuva 12. Playwright suorittaa automaattisesti odotuksen ja elementin tarkemman paikantamisen. Kuvassa Playwright paikantaa ContactName CSS-valitsinta käyttöliittymästä.

5.1 Rajapintatestaus Playwrightilla

Rajapintatestien automatisoinnissa voidaan hyödyntää Playwrightin ominaisuuksia, jotka mahdollistavat automatisoitujen HTTP-pyyntöjen suorittamisen, vastausten tarkistamisen ja erilaisten tarkastusten tekemisen rajapinnan toimivuuden ja odotettujen tulosten varmistamiseksi. (Playwright ApiRequestContext 2023.)

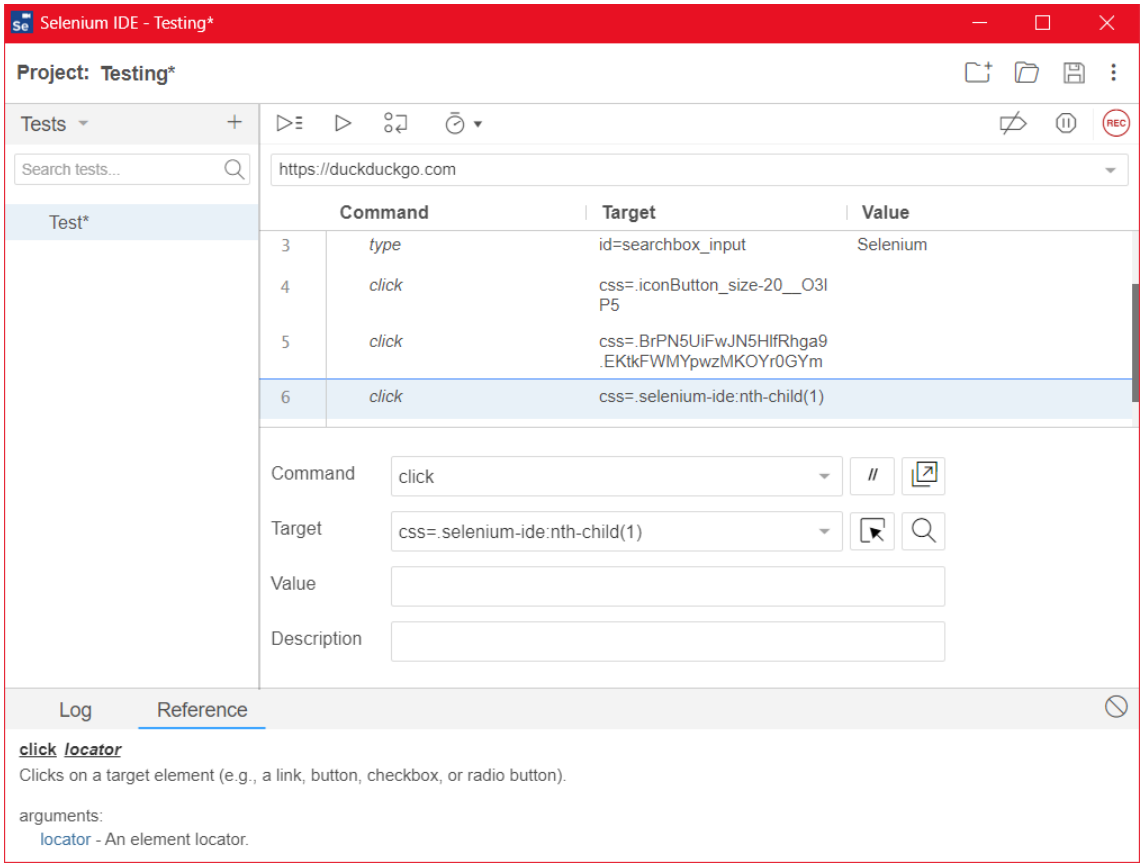
Liitteessä 1 esitellään testi, jossa suoritetaan rajapintatestaus Playwrightin APIRequestContext- ja APIResponse-luokan metodilla. Testi suoritetaan Restful-booker alustalla, joka muodostaa aamiaismajoitusten varaamisjärjestelmän automaatiotestien harjoitteluun. (Restful-booker 2020.)

6 SELENIUM

Jason Huggins esitteli alun perin Seleniumin vuonna 2004, ja siitä lähtien projekti on saanut yli 30 000 kommittia GitHubissa vapaaehtoisilta kehittäjiltä. Selenium on suosittu avoimen lähdekoodin kokoelma erilaisia työkaluja ja kirjastoja verkkosivujen automaatiotestaamiseen. Sen ekosysteemistä tunnetuimmat työkalut ovat W3C-standardiksi vakiintunut WebDriver, Selenium Grid ja Selenium IDE. (Selenium History 2023.)

Selenium WebDriver on ohjelmointirajapinta ja työkalu, joka mahdollistaa web-selaimen automaattisen ohjaamisen. Se suorittaa web-sovelluksen käyttöliittymässä erilaisia toimintoja, kuten sivujen hallintaa, lomakkeiden täyttämistä ja klikkauksia sivun elementteihin. (Selenium WebDriver 2023.)

Selenium Grid on keskitetty resurssienhallintajärjestelmä, joka mahdollistaa Selenium-testien suorittamisen samanaikaisesti useissa eri selainympäristöissä ja laitteissa. Tämä tehdään käyttämällä keskuspalvelinta (Hub) ja erillisiä testiympäristöjen solmuja (Node). (Selenium Grid 2023.)



Kuva 13. Selenium IDE Firefox selaimeen asennettuna.

Selenium IDE on selaimeen asennettava työkalu, joka mahdollistaa verkkosivun elementtien paikantamisen ja sivun elementtien käsittelyyn sopivan komennon tunnistamisen. Testisuoritusten tallentaminen ja niiden toistaminen Selenium IDE:n avulla auttavat nopeuttamaan testausta ja vähentämään virheitä. (Selenium IDE 2021.)

Seleniumin merkittävistä eduista on sen laaja kielituki, mikä mahdollistaa sen yhteensopivuuden eri testauskehyksien kanssa. Opinnäytetyötä varten käytettiin Seleniumia Pytestin lisäksi SeleniumBase-kehyksessä.

SeleniumBase on avoimen lähdekoodin Python-kehys, joka yksinkertaistaa ja tehostaa web-sovellusten automatisoitujen testien luomista ja suorittamista Seleniumilla. Se hyödyntää Selenium WebDriver -kirjastoa web-selainten automatisointiin ja tarjoaa lisäominaisuuksia, kuten testien järjestämistä, rinnakkaista suoritusta, mukautettua testiraportointia ja web-automaatiokomentoja. SeleniumBaseen on integroitu Pytest- ja Behave-kirjastot, joka mahdollistavat Pytestin ja Behaven ominaisuuksien käytön SeleniumBasella.

BaseCase-luokka on keskeinen osa SeleniumBase-kehystä. Pytestin ominaisuuksien käytössä SeleniumBase hyödyntää Pythonin ”unittest.TestCase” -luokkaa, joka on Pythonin standardikirjastoon kuuluvia ominaisuuksia. Yksi BaseCasen tärkeimmistä hyödyistä on parannukset WebDriverin komentoihin, jotka tekevät siitä vakaamman, luotettavamman ja joustavamman. Sivuston elementeille annetaan riittävästi aikaa latautua, ennen kuin WebDriver suorittaa niihin liittyviä toimintoja, mikä varmistaa testien luotettavuuden.

SeleniumBasen asennus on yksinkertaista: se voidaan suorittaa yhdellä komennolla virtuaaliympäristöön käyttäen komentoa ”pip install seleniumbase”. Tällöin asennetaan myös automaattisesti uusimmat selainajurit, Pytest, Pytestin lisäosat, Seleniumin kirjastot ja Behave. (SeleniumBase 2023.)

```python
import pytest
from selenium import webdriver

@pytest.fixture
def driver():
    driver = webdriver.Chrome()
    yield driver
    driver.quit()

def test_google_title(driver):
    driver.get("https://www.google.com")
    assert "Google" in driver.title
```

Kuva 14. Selenium Pytest-kehyksessä: skripti aukaisee selaimen Chromella ja tarkistaa sivun otsikon.

```python
from seleniumbase import BaseCase

class TestGoogleSearch(BaseCase):

    def test_google_title(self):
        self.open("https://www.google.com")
        self.assert_title("Google")
```

Kuva 15. SeleniumBasen skripti aukaisee automaattisesti sivun Chromella ja tarkistaa sivun otsikon. Komentorivillä voi määrittää selaimenohjelman sivun aukaisuun.

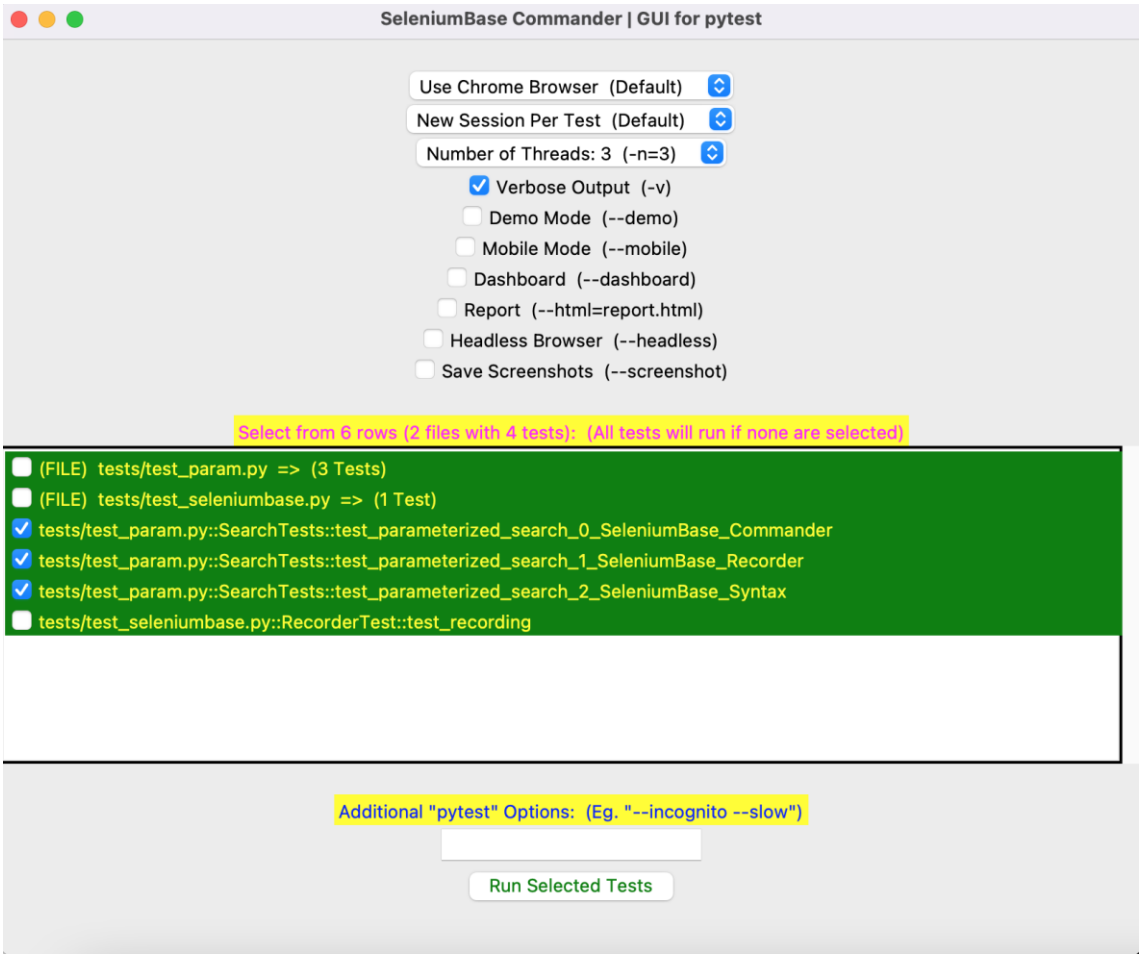
SeleniumBase Dashboard ja raportit tarjoavat visuaalisen yhteenvedon testien suorittamisesta. Näissä raporteissa voi tarkastella testien tuloksia, virheitä ja suorituskykyä graafisesti. Ne auttavat nopeasti tunnistamaan ja diagnosoimaan ongelmia testiajoissa ja tarjoavat yksityiskohtaisen yleiskuvan testien tilasta ja suorituskyvystä. (SeleniumBase Dashboard 2023.)

SeleniumBase Recorder -tila mahdollistaa käyttäjien vuorovaikutusten tallentamisen web-sovelluksen käytön aikana selaimessa ja tuottaa niiden perusteella Python-koodia. Tätä generoitua koodia voi muokata ja käyttää automatisoitujen testiskriptien luomiseen. Recorder-tila tekee Selenium-testiskriptien laatimisesta helpompaa, kun se tallentaa käyttäjän toimia, kuten klikkauksia ja lomakkeiden lähetyksiä, ja tuottaa niihin vastaavan koodin automaattisesti

SeleniumBase Commander mahdollistaa pytest- ja behave-skriptien suorittamisen graafisen käyttöliittymän avulla. Ohjelma käynnistetään komennolla ”sbase gui” tai ”sbase behave-gui”, minkä jälkeen voit valita ja mukauttaa testejä sekä asetuksia. Tämä sisältää eri selainten valinnan ja niihin

liittyvät asetukset. Lisäksi ohjelma tukee mobiililaitteiden testausta, tarjoaa raportointiin visuaalisen näkymän testien suorituksista ja mahdollistaa Pytestillä testien suorittamisen rinnakkain.

Mobiililaitteiden testauksessa SeleniumBase hyödyntää Chromen selaimen Device Modea, jossa voidaan käyttää selaimen tarjoamia mahdollisuuksia tarkastella sivua mobiililaitteella. Mobiililaitteiden testaukseen voidaan muissa selaimissa käyttää Seleniumin Gridiä (SeleniumBase GUI 2023.)



Kuva 16 SeleniumBase Commander -ikkuna.

6.1 SeleniumBase Behave

Behave on Pythonille suunniteltu avoimen lähdekoodin kehys, joka mahdollistaa käyttäytymisperusteisen kehityksen (BDD) noudattamista. Sen tarkoituksena on helpottaa kehittäjien, testaajien

ja ei-teknisten sidosryhmien yhteistyötä ohjelmiston käyttäytymisen testaamisessa ihmisläheisellä kielellä.

Behave hyödyntää Gherkin-kieltä, joka tarjoaa helppolukuisen muodon skriptille. Gherkin mahdollistaa ohjelmistomoduulien ja skenaarioiden määrittämisen strukturoidusti käyttämällä avainsanoja, kuten "Given", "When" ja "Then", kuvaamaan alkutilanteen, toiminnot ja odotetut tulokset.

SeleniumBase hyödyntää Behave-kehyksessä BaseCase-luokan metodeja samalla tavalla kuin Pytestillä. SeleniumBase on luonut omat behave_sb -komennot, jotka helpottavat testisarjojen luontia Behave-kehyksessä.

```
features/
├── __init__.py
├── behave.ini
├── environment.py
├── bookstore.feature
└── steps/
    ├── __init__.py
    ├── imported.py
    └── bookstore.py
```

Kuva 17. Esimerkki SeleniumBasella luodun BDD-testin rakenteesta.

Ominaisuustiedostot (.feature) kuvaavat testitapaukset luonnollisella kielellä ja määrittelevät, mitä testattava ohjelmisto tekee. Askeleiden kansiosta (steps) löytyvät Python-tiedostot, jotka määrittelevät, miten kunkin ominaisuustiedoston askel suoritetaan ohjelmistossa.

Ympäristötiedosto (environment.py) mahdollistaa SeleniumBasen integroinnin Behaven kanssa ja asettaa SeleniumBasen BaseCase-luokan Behave-testeille. Se myös tarjoaa metodeja, jotka mahdollistavat toimintojen suorittamisen eri vaiheissa testien aikana. Kuvan 17 skripti löytyy liitteestä 2.

Vaikka konfiguraatiotiedoston behave.ini käyttö ei ole välttämätöntä, siihen voi asettaa Behave-testikehyksen asetukset ja vaihtoehdot, jotka ohjaavat testien suorittamista. (SeleniumBase Behave 2023; Behave 2023.)

7 SELENIUMIN JA PLAYWRIGHTIN VERTAILU

Taulukossa 1 käydään läpi Seleniumin ja Playwrightin keskeiset ominaisuudet web-sovellusten testaamiseen. Taulukosta voidaan huomata, että Playwright käyttää hyödykseen selainautomaatiossa selainmoottoreita, kun taas Selenium käyttää hyödykseen selainajureita. Aikaisemmissa kappaleissa olemme voineet huomata, että Selenium on enemmänkin kokoelma erilaisia työkaluja, kun taas Playwright on täysi testauskehys, joka soveltuu myös rajapintatestaukseen. (Selenium vs Playwright 2023.)

Ominaisuudet	Selenium	Playwright
Selaintuki	Chrome, Firefox, Safari ja Edge WebDriverin kautta	Chromium, Firefox (Gecko) ja WebKit
Selaimen konsolin käyttö	Rajallinen pääsy selaimen konsoliin	Täysi pääsy selaimen konsolilokiin
Verkkopyynnöt	Rajallinen pääsy verkkopyyntöihin	Täysi pääsy verkkopyyntöihin
Mobiilitestaus	Voidaan käyttää mobiilitestaukseen Appiumin avulla	Mobiilitestaus mahdollista emulaattorilla
Kielituki	Java, Python, Ruby, C# ja JavaScript	TypeScript/JavaScript, Python, .NET ja Java
Lisenssi	Avoin lähdekoodi, Apache License 2.0	Avoin lähdekoodi, Apache License 2.0

Taulukko 1. Taulukossa on kuvailtu Seleniumin ja Playwrightin ominaisuudet.

Tässä kappaleessa on tarkoituksena vertailla Selenumia ja Playwrightin selainhallintaa, CSS-paikantimien käyttöä ja suoritusnopeutta Python-ohjelmointikielellä. Suoritusnopeuden vertailussa käytetään Pytest-kehystä Seleniumilla ja Playwrightilla.

7.1 Selainhallinta

Selenium tukee selaimia käyttämällä WebDriver-ajureita. Tämä tarkoittaa, että Seleniumin käyttöön tarvitaan erillisten WebDriver-ajureiden asentamista ja konfigurointia (esimerkiksi ChromeDriver Chrome-selaimelle ja GeckoDriver Firefox-selaimelle). Selenium hyödyntää näitä ajureita hallitsemaan selainohjelmaa verkkosivun käyttöliittymässä.

Seleniumin selainhallinta mahdollistaa jokaisen testin suorittamisen omassa selaininstanssissa, jonka suoritus on erillään muista testeistä. Tämä eristäminen mahdollistaa useiden testien samanaikaisen suorituksen. Selenium ei kuitenkaan tyhjennä automaattisesti selainohjelman välimuistia, minkä vuoksi testit eivät ole täysin erillisiä ilman erikseen määrittelyä. (Selenium Getting Started 2022; Selenium Medium 2023.)

Playwright käyttää sisäänrakennettuja selainmoottoreita sen sijaan, että se hallitsisi suoraan kolmannen osapuolen selainohjelmaa. Jokainen testi luo oman ainutlaatuisen selainkontekstin selaininstanssissa, mikä vastaa käytännössä selaimen incognito-tilaa. Yhdellä selaininstanssilla voi olla samanaikaisesti useita konteksteja, ja jokainen konteksti toimii täysin erillään muista. Jokaisella selainkontekstilla on myös omat evästeet ja istuntotiedot, jotka eivät vaikuta muihin konteksteihin. Tämä mahdollistaa testien suorittamisen rinnakkain samalla selaininstanssilla, koska jokainen konteksti toimii erillään toisistaan.

Playwrightin toimintatapa eroaa perinteisestä selainhallinnasta merkittävästi ja tarjoaa paremman kontrollin selainprosessien hallintaan ja rinnakkaiseen testaukseen, koska testit ovat täysin eristettyjä selaimen välimuistista ja evästetiedoista. (Playwright Browser 2023; Playwright Broswer-Contexts 2023.)

7.2 Paikantimet

Elementtien paikantamiseksi Seleniumin paikantimet keskittyvät verkkosivun DOM-rakenteeseen ja sen attribuutteihin, kuten id:hen, nimeen ja xpath:iin. Paikantimien tarkkuudesta huolimatta Selenium ei sisällä automaattista odottamista verkkosivun DOM-muutosten varalta. Ilman automaat-

tista odottamista mahdollistavilla kehyksillä, kuten SeleniumBasella, Seleniumin testeillä on lisättävä eksplisiittisiä odotuksia, jotta testin pystyy suorittamaan DOM-puun hitaiden muutosten aikana. (Selenium Locators 2023; Selenium Waits 2023.)

Playwright puolestaan painottaa käyttäjille näkyviä attribuutteja, kuten tekstin sisältöä ja elementtien rooleja, kun se paikantaa elementtejä. Playwrightin sisäänrakennettu automaattinen odotus paikantaa elementtejä tekee testeistä vakaampia ja vähemmän herkkiä DOM-puun muutoksille. Kuva 12 havainnollistaa sitä, kuinka monta eri toimintaa Playwright saattaa tehdä automaattisesti yhdessä paikannuksessa. (Playwright Locators 2023.)

7.3 Suoritusnopeus

Checklyn vertailu eri automaatiotyökalujen suoritusnopeuksista havaitsi, että Playwrightilla on nopeampi suoritusnopeus kaikilla osa-alueilla Seleniumiin verrattuna. Mediumin artikkeli teki vastaavan havainnon, kun se vertaili Playwrightia ja Seleniumia verkkoharavoinnissa. (Checklyhq 2023; Medium 2023.)

Opinnäytetyössä havaittiin sama asia. Suoritusnopeuden testaaminen toteutettiin samankaltaisilla skripteillä päättömässä tilassa samalle sivulle 10 kertaa peräkkäin. Vertailussa huomattiin selkeä ero Seleniumin selainhallinnan toiminnassa, jossa uudestaan suoritettava skripti käynnistyy hitaasti. Seleniumin suoritusnopeus oli testissä 43,99 sekuntia, kun taas Playwrightin nopeus oli 12,46 sekuntia. Suoritusnopeuden vertailun skriptit ovat liitteessä 3.

8 SIVUOBJEKTIMALLI

Opinnäytetyössä on luotu sivuobjektimalli (Page Object Model) verkkokauppojen käyttöliittymän testaamista varten. Suunnittelumalli on jaettu kolmeen osaan, jotka tarkastelevat verkkokaupan hakuominaisuuden, hakutulosten ja ostoskorin toiminnallisuuden testaamista useilla syötteillä. Testin tehtävänä on arvioida suunnittelumallin avulla saavutettuja tuloksia käyttäjän toiminnan näkökulmasta ja varmistaa niiden vastaavan odotuksia.

Suunnittelumallissa jokainen verkkosivu mallinnetaan luokkapohjaisesti, jolloin sivun toiminnot ja elementit liitetään tähän luokkaan. Jokaisen sivun luokka sisältää toimintoja, jotka mahdollistavat web-sovelluksen elementtien hallinnan ja tarkastuksen.

Sivuobjektimallin tavoitteena on eristää testitapaukset sivujen rakenteesta ja toiminnoista. Tämä tekee testien kirjoittamisesta helpommin ylläpidettävämpää, koska muutokset sivujen rakenteessa tai toiminnoissa eivät vaikuta suoraan testitapauksiin. Lisäksi sivuobjektimalli edistää testien uudelleenkäytettävyyttä, koska sivuja ja niiden toimintoja voidaan käyttää useissa eri testeissä.

```
Testit
├──pages
│  search.py
│  result.py
│  cart.py
│  __init__.py
│
├──tests
│ │conftest.py
│ │test_case.py
│
│pytest.ini
```

Kuva 18. Havainnollistava sivuobjektimallin kuvaus projektissa.

Suunnittelumalli on toteutettu SeleniumBasella ja Playwrightilla Pytest-kehyksessä. Suunnittelumallin testiskenaarion lisäksi tarkoituksena on tarkastella, miten SeleniumBasella ja Playwrightilla toteutetaan monimutkaisempia käyttöliittymän päästä päähän -testejä Pythonilla.

Pages-hakemistorakenteessa on Python-paketti, josta test_case.py-moduuli kutsuu paketin moduulien luokkia. Liitteissä 4 ja 5 on kummankin projektin toteutus.

SeleniumBase sisältää valmiiksi kaikki ajurit ja testiin tarvittavat lisäosat kirjaston asennuksessa. Pytest-kehykseen asennetaan Playwrightin kirjaston ja playwright-pytest integroinnin lisäksi lisäosa pytest-xdist, joka mahdollistaa testien suorittamisen rinnakkain.

9 POHDINTA

Opinnäytetyön tavoitteena oli objektiivinen vertailu Playwrightin ja Seleniumin välillä. Vertailun tuloksena havaittiin, että useissa eri ominaisuuksissa Playwright osoittautui selvästi käytännöllisemmäksi kuin Selenium.

Tämä ero voi osittain johtua Seleniumin avoimen hallinnoinnin lähestymistavasta, joka mahdollistaa yhteisön osallistumisen päätöksentekoon ja avoimen keskustelun projektin kehittämisestä. Playwright taas on Microsoftin hallinnoima projekti, jossa käyttäjillä ei ole samanlaista vaikutusvaltaa suunnitteluprosessiin. On tärkeää huomata, että näiden projektien rahoituksessa voi olla eroja, mutta tämä ei välttämättä ole ratkaiseva tekijä.

Opinnäytetyön tuloksena Playwrightin ominaisuudet ja käytettävyys osoittautuivat monilla mittareilla ylivoimaisiksi Seleniumiin verrattuna. Lopullinen valinta näiden kahden välillä saattaa riippua käyttäjän painotuksista, kuten pragmaattisuudesta tai avoimuudesta projektin suhteen.

Opinnäytetyön tekemisen aikana tutkin myös Seleniumin historiaa ja sen merkitystä selainten automaattisessa testauksessa. Vaikka Seleniumilla on ollut puutteita, se ansaitsee silti tunnustusta sen historiallinen panos selainohjelmien yhteistyöhön avoimen lähdekoodin testausohjelmissa ansaitsee tunnustusta.

Käytännön tasolla havaitsin, että Playwright tarjoaa käyttäjälle kattavan testaustohjelman yhden asennuksen avulla, kun taas Selenium vaatii erillisten ohjelmistojen asentamisen. Lisäksi Playwrightin suorituskyky vaikutti nopeammalta. Vaikka käyttöliittymätoiminnot saattavat hieman vaihdella, ne eivät näyttäneet olevan ratkaisevan tärkeitä, jotta selainohjelmien käyttö voitaisiin perustella selainmoottoreiden sijaan.

LÄHTEET

Behave 2023. Behavior Driven Development.
Hakupäivä 17.10.2023. https://behave.readthedocs.io/en/latest/philosophy/

BrowserStack 2023. What is Headless Browser and Headless Browser Testing?
Hakupäivä 27.8.2023. https://www.browserstack.com/guide/what-is-headless-browser-testing

Checkhq 2021. Cypress vs Selenium vs Playwright vs Puppeteer speed comparison.
Hakupäivä 25.9.2023. https://www.checklyhq.com/blog/cypress-vs-selenium-vs-playwright-vs-puppeteer-speed-comparison/

Freecontent 2020. Making Better Unit Tests: part 1, the AAA pattern.
Hakupäivä 4.10.2023. https://freecontent.manning.com/making-better-unit-tests-part-1-the-aaa-pattern/

Ghostinspector 2023. CSS Selector Cheat Sheet: Strategies for Automated Browser Testing.
Hakupäivä 4.10.2023. https://ghostinspector.com/blog/css-selector-strategies-automated-browser-testing/

Github Blog 2023. Why Python keeps growing, explained.
Hakupäivä 2.9.2023. https://github.blog/2023-03-02-why-python-keeps-growing-explained/

Guru99 2023. What is BLACK Box Testing? Techniques, Types & Example.
Hakupäivä 4.10.2023. https://www.guru99.com/black-box-testing.html

Playwright ApiRequestContext 2023. APIRequestContext.
Hakupäivä 6.9.2023. https://playwright.dev/python/docs/api/class-apirequestcontext

Playwright Browsers 2023. Browsers.
Hakupäivä 19.9.2023. https://playwright.dev/python/docs/browsers

Playwright Browser-contexts 2023. Isolation.

Hakupäivä 19.9.2023. https://playwright.dev/python/docs/browser-contexts

Playwright Codegen 2023. Test generator.
Hakupäivä 27.9.2023. https://playwright.dev/python/docs/codegen

Playwright Pytest 2023. Pytest Plugin Reference
Hakupäivä 4.9.2023. https://playwright.dev/python/docs/test-runners

Playwright Tech 2020. Introduction and walkthrough into the Playwright framework: a Node.js library to automate Chromium, Firefox and WebKit with a single API.
Hakupäivä 27.9.2023. https://playwright.tech/blog/what-is-playwright

Playwright Trace Viewer 2023. Trace Viewer Intro.
Hakupäivä 9.9.2023. https://playwright.dev/python/docs/trace-viewer-intro

Playwright 2023. Playwright enables reliable end-to-end testing for modern web apps.
Hakupäivä 4.9.2023. https://playwright.dev

Pytest Anatomy of Test 2015. Anatomy of a test.
Hakupäivä 4.9.2023. https://docs.pytest.org/en/7.3.x/explanation/anatomy.html

Pytest Explanation Fixtures 2015. About fixtures.
Hakupäivä 4.10.2023. https://docs.pytest.org/en/7.1.x/explanation/fixtures.html?highlight=fixture

Pytest Fixture 2020. Pytest Fixtures: Explicit, Modular, Scalable.
Hakupäivä 4.9.2023. https://docs.pytest.org/en/6.2.x/fixture.html

Pytest Good Practice 2015. Good Integration Practices.
Hakupäivä 4.9.2023. https://docs.pytest.org/en/7.1.x/explanation/goodpractices.html

Pytest History 2015. History.
Hakupäivä 7.9.2023. https://docs.pytest.org/en/7.1.x/history.html

Pytest How To Mark 2015. How to mark test functions with attributes.

Hakupäivä 4.10.2023. https://docs.pytest.org/en/latest/how-to/mark.html

Pytest Invoke 2015. How to invoke pytest.
Hakupäivä 4.10.2023. https://docs.pytest.org/en/7.1.x/how-to/usage.html?highlight=name

Pytest Parametrize 2015. How to parametrize fixtures and test functions.
Hakupäivä 4.9.2023. https://docs.pytest.org/en/7.3.x/how-to/parametrize.html

Python Venv 2023. Creation of virtual environments.
Hakupäivä 29.8.2023. https://docs.python.org/3/library/venv.html

Python Programming 2022. Python Programming/Overview.
Hakupäivä 2.9.2023. https://en.wikibooks.org/wiki/Python_Programming/Overview

Restful-Booker 2023. API documentation for the playground API restful-booker.
Hakupäivä 6.9.2023. https://restful-booker.herokuapp.com/apidoc/index.html

Ron Patton, 2004, The Art of Software Testing, Second Edition. Wiley.

SeleniumBase Dashboard 2023. Dashboard / Reports.
Hakupäivä 6.9.2023. https://seleniumbase.io/examples/example_logs/ReadMe/

SeleniumBase GUI 2023. SeleniumBase Commander.
Hakupäivä 14.9.2023. https://seleniumbase.io/help_docs/commander/

Selenium Grid 2023. Want to run tests in parallel across multiple machines? Then, Grid is for you.
Hakupäivä 17.10.2023. https://www.selenium.dev/documentation/grid/

Selenium History 2023. Selenium History.
Hakupäivä 8.9.2023. https://www.selenium.dev/history/

Selenium IDE 2023. The Selenium IDE is a browser extension that records and plays back a user's actions.
Hakupäivä 17.10.2023. https://www.selenium.dev/documentation/ide/

Selenium WebDriver 2023. WebDriver drives a browser natively, learn more about it. Hakupäivä 8.9.2023. https://www.selenium.dev/documentation/webdriver/

SeleniumBase 2023. SeleniumBase. Hakupäivä 14.9.2023. https://seleniumbase.io/

Selenium Getting Started 2023. If you are new to Selenium, we have a few resources that can help you get up to speed right away. Hakupäivä 19.9.2023. https://www.selenium.dev/documentation/webdriver/getting_started/

Selenium Locators 2023. Locator strategies. Hakupäivä 22.9.2023. https://www.selenium.dev/documentation/webdriver/elements/locators/

Selenium Medium 2023. How to handle Cookies in Selenium WebDriver. Hakupäivä 4.10.2023. https://www.browserstack.com/guide/how-to-handle-cookies-in-selenium

Techtarget 2023. What is Web Application (Web Apps) and its Benefits. Hakupäivä 4.10.2023. https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app.

LIITTEET:

Liite 1: Playwright rajapintatestaus

Liite 2: Seleniumbase Behave

Liite 3: Suoritusnopeus Playwrightila ja Seleniumilla

Liite 4: Sivuobjektimalli Playwright

Liite 5: Sivuobjektimalli Selenium

```
#test_booker.py

from playwright.sync_api import APIRequestContext

#test_post
def test_post(api_context: APIRequestContext):
    data = {
        "firstname": "John",
        "lastname": "Doe",
        "totalprice": 99,
        "depositpaid": True,
        "bookingdates": {
            "checkin": "2023-09-05",
            "checkout": "2023-09-06"
        },
        "additionalneeds": "English breakfast"
    }
    new_booking = api_context.post(
        "/booking", data=data
    )

    print(new_booking.body())
    assert new_booking.ok
    assert new_booking.status == 200

# Add the ID from the 'post' command to the 'get', 'put' or
'delete' command ("/booking/" + id_here)

#test_get
def test_get(api_context: APIRequestContext):
    bookings = api_context.get(
    "/booking/"
    )

    print(bookings.body())
    assert bookings.ok
    assert bookings.status == 200
```

```
#test_put
def test_put(api_context: APIRequestContext):

    data = {
        "firstname": "Jane",
        "lastname": "Doe",
        "totalprice": 99,
        "depositpaid": True,
        "bookingdates": {
            "checkin": "2023-09-05",
            "checkout": "2023-09-06"
        },
        "additionalneeds": "English breakfast"
    }
    put_booking = api_context.put(
        "/booking/", data=data
    )
    print(put_booking.body())
    assert put_booking.ok
    assert put_booking.status == 200


#test_delete.py
def test_delete(api_context: APIRequestContext):


    delete_booking = api_context.delete(
        "/booking/"
    )

    print(delete_booking)
    assert delete_booking.ok
    assert delete_booking.status == 201
```

```
#conftest.py

import pytest
from playwright.sync_api import Playwright, APIRequestContext


# Get API Authentication Token
@pytest.fixture(scope="session")
def get_authenticated(
        playwright: Playwright,
) -> [APIRequestContext, None, None]:
    headers = {
        "Content-Type": "application/json"
    }
    request_context = playwright.request.new_context(
        base_url="https://restful-booker.herokuapp.com", ex-
tra_http_headers=headers
    )
    data = {
        "username": "admin",
        "password": "password123"
    }
    auth_response = request_context.post("/auth", data=data)
    token = auth_response.json()["token"]
    request_context.token = token
    yield request_context
    request_context.dispose()



## API Context for Authenticated Requests
@pytest.fixture(scope="session")
def api_context(
        playwright: Playwright,
        get_authenticated: APIRequestContext
) -> [APIRequestContext, None, None]:
    headers = {
        "Content-Type": "application/json",
        "Accept": "application/json",
        "Cookie": f"token={get_authenticated.token}",
    }
    request_context = playwright.request.new_context(
        base_url="https://restful-booker.herokuapp.com", ex-
tra_http_headers=headers
    )
    yield request_context
    request_context.dispose()
```

```
#bookstore.feature

Feature: Test Automation Search

  Scenario: User searches for "Test Automation" on Automation
Bookstore
    Given Open the Google homepage
    When Perform a search for "Test Automation"
    Then Verify search results for "Test Automation"


#enviroment.py

from seleniumbase import BaseCase
from seleniumbase.behave import behave_sb
behave_sb.set_base_class(BaseCase)
from seleniumbase.behave.behave_sb import before_all
from seleniumbase.behave.behave_sb import before_feature
from seleniumbase.behave.behave_sb import before_scenario
from seleniumbase.behave.behave_sb import before_step
from seleniumbase.behave.behave_sb import after_step
from seleniumbase.behave.behave_sb import after_scenario
from seleniumbase.behave.behave_sb import after_feature
from seleniumbase.behave.behave_sb import after_all


#bookstore.py
from behave import step


@step("Open the Google homepage")
def open_google_homepage(context):
    sb = context.sb
    sb.open("https://automationbookstore.dev/")
    sb.clear_local_storage()


@step('Perform a search for "{search_term}"')
def perform_search(context, search_term):
    sb = context.sb
    sb.type("input#searchBar", search_term)


@step('Verify search results for "{expected_results}"')
def verify_search_results(context, expected_results):
    sb = context.sb
    sb.assert_text(expected_results)

#imported.py
from seleniumbase.behave import steps
```

```
#Seleniumin skripti

from selenium import webdriver
from selenium.webdriver.common.by import By
import pytest

@pytest.mark.repeat(10)
def test_sauce():
    chrome_options = webdriver.ChromeOptions()
    chrome_options.add_argument("--headless")
    driver = webdriver.Chrome(options=chrome_options)
    driver.get("https://www.saucedemo.com/")
    driver.find_element(By.ID, "user-name").send_keys("stand-
ard_user")
    driver.find_element(By.ID, "password").send_keys("se-
cret_sauce")
    driver.find_element(By.ID, "login-button").click()



#Playwrightin skripti

import pytest
from playwright.sync_api import Page


@pytest.mark.repeat(10)
def test_sauce(
        page: Page):
    page.goto("https://www.saucedemo.com/")
    page.fill("input#user-name", "standard_user")
    page.fill("input#password", "secret_sauce")
    page.click("input#login-button")
```

```
#Playwright test_case

import pytest
from pages.search import SearchPage
from pages.result import ResultPage
from pages.cart import CartPage
from playwright.sync_api import Page

products = [

]

@pytest.mark.parametrize('phrase', products)
def test_ecommerce(
        phrase: str,
        page: Page,
        cart_page: CartPage,
        result_page: ResultPage,
        search_page: SearchPage) -> None:

    search_page.load()

    search_page.search(phrase)

    assert result_page.result_link_titles_con-
tain_phrase(phrase)

    result_page.click_result_link()

    cart_page.click_add_to_cart_button()

    cart_page.click_cart_icon()

    cart_page.click_delete_button()

    assert cart_page.is_cart_empty()
```

```
#Playwright conftest.py

import pytest
from playwright.sync_api import Page
from pages.search import SearchPage
from pages.result import ResultPage
from pages.cart import CartPage

@pytest.fixture
def result_page(page: Page) -> ResultPage:
    return ResultPage(page)

@pytest.fixture
def search_page(page: Page) -> SearchPage:
    return SearchPage(page)

@pytest.fixture
def cart_page(page: Page) -> CartPage:
    return CartPage(page)


#Playwright search.py

from playwright.sync_api import Page

class SearchPage:

    def __init__(self, page: Page) -> None:
        self.page = page
        self.search_button = page.locator('')
        self.search_input = page.locator('')

    def load(self) -> None:
        self.page.goto('/')

    def search(self, phrase: str) -> None:
        self.search_input.fill(phrase)
        self.search_button.click()
```

```
#Playwright result.py

from playwright.sync_api import Page


class ResultPage:

    def __init__(self, page: Page) -> None:
        self.page = page
        self.result_links = page.locator('')

    def result_link_titles_contain_phrase(self, phrase: str,
minimum: int = 5) -> bool:
        titles = self.result_links.all()
        text_titles = [title.text_content() for title in ti-
tles]
        matches = [title for title in text_titles if
phrase.lower() in title.lower()]
        return len(matches) >= minimum

    def click_result_link(self):
        self.result_links.click()


#Playwright cart.py

from playwright.sync_api import Page

class CartPage:
    def __init__(self, page: Page) -> None:
        self.page = page
        self.empty_cart = page.locator('')

    def click_add_to_cart_button(self):
        self.page.click('')

    def click_cart_icon(self):
        self.page.click('')

    def click_delete_button(self):
        self.page.click('')


    def is_cart_empty(self):
        cart_text = self.empty_cart.inner_text()
        return cart_text == '0'
```

```
#Selenium test_case.py

import pytest
from pages.search import SearchPage
from pages.result import ResultPage
from pages.cart import CartPage

products = [

]

@pytest.mark.parametrize('phrase', products)
def test_ecommerce(
        search_page: SearchPage,
        result_page: ResultPage,
        cart_page: CartPage,
        phrase: str
        ) -> None:

    search_page.load()

    search_page.search(phrase)

    assert result_page.result_link_titles_con-
tain_phrase(phrase)

    result_page.click_result_link()

    cart_page.click_add_to_cart_button()

    cart_page.click_cart_icon()

    cart_page.click_delete_button()

    assert cart_page.is_cart_empty()
```

```
#Selenium conftest.py

import pytest
from pages.search import SearchPage
from pages.result import ResultPage
from pages.cart import CartPage


@pytest.fixture
def search_page(sb):
    return SearchPage(sb)

@pytest.fixture
def result_page(sb):
    return ResultPage(sb)

@pytest.fixture
def cart_page(sb):
    return CartPage(sb)

#Selenium search.py

from seleniumbase import BaseCase


class SearchPage:

    def __init__(self, base_case: BaseCase) -> None:
        self.base_case = base_case
        self.search_button = ''
        self.search_input = ''

    def load(self):
        self.base_case.open_start_page()

    def search(self, phrase: str):
        self.base_case.type(self.search_input, phrase)
        self.base_case.click(self.search_button)
```

```
#Selenium result.py

from seleniumbase import BaseCase

class ResultPage:

    def __init__(self, base_case: BaseCase) -> None:
        self.base_case = base_case
        self.result_links = ''

    def result_link_titles_contain_phrase(self, phrase: str,
minimum: int = 5) -> bool:
        titles = self.base_case.find_elements(self.re-
sult_links)
        text_titles = [title.text for title in titles]
        matches = [t for t in text_titles if phrase.lower() in
t.lower()]
        return len(matches) >= minimum

    def click_result_link(self):
        self.base_case.click(self.result_links)


#Selenium cart.py
from seleniumbase import BaseCase

class CartPage:

    def __init__(self, base_case: BaseCase) -> None:
        self.base_case = base_case

    def click_add_to_cart_button(self):
        self.base_case.click('')

    def click_cart_icon(self):
        self.base_case.click('')

    def click_delete_button(self):
        self.base_case.click('')


    def is_cart_empty(self):
        cart_text = self.base_case.get_text('')
        return cart_text == '0'
```