Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – магістр

на тему: **«Підсистема для динамічного тестування доступності вебдодатків у системі CI/CD»**

Виконав: студент 2-го року навчання
освітньої програми «Інженерія
програмного забезпечення»,
спеціальності 121 Інженерія
програмного забезпечення

Огир Вадим Дмитрович

Рецензент: Нагірна А. М.,
кандидат фіз.-мат. наук, доцент

Рецензент: Афонін А. О.,
кандидат фіз.-мат. наук, доцент

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК _____
«____» _____ 20____ р.

Київ – 2024

Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики,
доцент, к.ф-м.н., С. С. Гороховський
_____
(підпис)
„____”_____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студенту 2-го курсу, факультету інформатики
Огиру Вадиму Дмитровичу

Дослідити та порівняти існуючі рішення для перевірки стану доступності та реалізувати застосунок для динамічного тестування доступності вебдодатків

Зміст ТЧ до кваліфікаційної роботи:

Зміст
Анотація
Вступ
1 Дослідження предметної області
2 Аналіз існуючих рішень для тестування доступності
3 Розробка підсистеми для динамічного тестування вебдоступності у системі CI/CD
4 Опис роботи підсистеми на прикладі реального вебдодатку
Висновки
Список використаних джерел

Дата видачі „___” _____ 2023 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Графік підготовки кваліфікаційної роботи до захисту

Графік узгоджено «_____» _____ 2023 р.

№ з/п	Перелік робіт	Термін	Підпис	Дата	Примітка
1.	Отримання теми кваліфікаційної роботи	16.10.2023			
2.	Ознайомлення з існуючою інформацією за темою кваліфікаційної роботи	20.10.2023			
3.	Розробка плану та структури роботи	01.11.2023			
4.	Ознайомлення з існуючими рішеннями для статичного тестування доступності	03.11.2023			
5.	Робота з дослідженням про алгоритм динамічного тестування доступності	01.01.2024			
6.	Початок створення власного рішення, реалізація методів статичного аналізу доступності	15.01.2024			
7.	Реалізація алгоритму визначення типу активного контенту	05.02.2024			
8.	Закінчення створення рішення для динамічного тестування вебдоступності	01.03.2024			
9.	Виконання порівняння роботи реалізованого алгоритму з аналогами на ринку	15.04.2024			
10.	Початок написання текстової частини	20.04.2024			
11.	Подання проміжної версії текстової частини	10.05.2024			
12.	Остаточне завершення написання текстової частини роботи	15.05.2024			
13.	Створення презентації	20.05.2024			
14.	Захист кваліфікаційної роботи	11.06.2024			

ЗМІСТ

АНОТАЦІЯ **6**

ВСТУП **7**

РОЗДІЛ 1: ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ **9**

1.1 Визначення вебдоступності, актуальність проблеми 9

1.2 Категорії користувачів з обмеженими можливостями 10

1.3 Класифікація типових проблем доступності 12

Висновки до розділу 1 19

РОЗДІЛ 2: АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ТЕСТУВАННЯ ДОСТУПНОСТІ **20**

2.1 Спеціалізовані рішення для тестування окремих правил доступності 20

2.2 Комплексні рішення для статичного тестування доступності 22

2.3 Динамічний підхід у тестуванні доступності 30

2.4 Постановка задачі 33

Висновки до розділу 2 35

РОЗДІЛ 3: РОЗРОБКА ПІДСИСТЕМИ ДЛЯ ДИНАМІЧНОГО ТЕСТУВАННЯ ВЕБДОСТУПНОСТІ У СИСТЕМІ CI/CD **36**

3.1 Опис алгоритму 36

3.2 Вибір інструментів 39

3.3 Архітектура підсистеми 43

3.4 Реалізація статичного аналізу доступності 45

3.5 Реалізація динамічного аналізу доступності 48

Висновки до розділу 3 55

РОЗДІЛ 4: ОПИС РОБОТИ ПІДСИСТЕМИ НА ПРИКЛАДІ РЕАЛЬНОГО ВЕБДОДАТКУ **56**

4.1 Особливості конфігурації новоствореної системи 56
4.2 Аналіз роботи створеного рішення на прикладі реального вебдодатку 57
Висновки до розділу 4 61
ВИСНОВКИ 62
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 64

АНОТАЦІЯ

Роботу присвячено аналізу існуючих методів статичного і динамічного тестування доступності сучасних вебзастосунків, виділенню їх переваг та недоліків на прикладі комплексних багатосторінкових вебдодатків. Також у роботі запропоновано метод динамічного тестування доступності, що поєднує у собі зазначені методи статичного тестування з алгоритмом автоматичного визначення типу активного контенту вебсторінки і його перевірки на співпадіння з відповідними групами правил доступності, і реалізовано підсистему для аналізу доступності вебдодатків у системі CI/CD на базі запропонованого методу. Отримані результати показали, що застосунок здатний виявляти нетипові помилки вебдоступності і аналізувати контент, недосяжний для алгоритмів статичного тестування.

ВСТУП

Безперечно, інтернет є одним з найважливіших винаходів людства, кількість користувачів якого перетнула половину популяції планети. Вже важко знайти сферу людської діяльності яку інтернет не змінив: від доступу до новин, пошуку роботи та простої комунікації до покупки товарів, здобуття освіти і навіть одруження. Втім, як відомо, чим більша система, тим більше в ній проблем, і інтернет не є виключенням – якщо раніше вебсайти були примітивними і однотипними, то зараз у веброзробників є абсолютна свобода дій, якою вони нерідко зловживають, а вебсайти перетворились у повноцінні вебдодатки. На жаль, через такі зловживання страждають в першу користувачі з обмеженими можливостями, які вже не здатні взаємодіяти з сайтами і сприймати інформацію нарівні з пересічними користувачами. Саме тому з еволюцією вебтехнологій зародилося поняття вебдоступності, що означає створення будь-якого контенту з думкою про те, що він має бути простим для сприйняття і використання всіма існуючими категоріями користувачів, включаючи людей з вадами зору, слуху, моторики, когнітивних здібностей, тощо.

Сьогодні зробити сучасний вебзастосунок доступним – непроста задача, яка часто починається з повної переробки дизайну та бібліотеки компонентів і закінчується ручним виправленням найвитонченіших проблем у коді застосунку. Звісно, існує низка інструментів, які значно полегшують аналіз, до прикладу, доступності контрасту тексту, або ж і повністю перевіряють вебсторінку на наявність поширених недоліків. Втім, для справді великих застосунків, цього недостатньо, адже з’являється людський фактор: функціональність, що вже була доступною, може перестати бути такою через неуважність або необізнаність інженера, що її змінив.

Отож, метою даної роботи є порівняння існуючих методів аналізу вебдоступності, виділення їх переваг та недоліків на прикладі роботи з сучасними вебдодатками, а також реалізація рішення для динамічного

тестування доступності вебзастосунків у системі CI/CD з метою недопущення деградації стану доступності з плином часу.

Об’єктом дослідження є вебдоступність і методи аналізу доступності у сучасних вебдодатках. Предметом дослідження є аналіз статичних методів тестування вебдоступності, створення алгоритму динамічного тестування вебдоступності, а також його реалізація і порівняння з існуючими аналогами на реальних вебсайтах.

Робота складається з чотирьох розділів, які містять підрозділи, висновків, а також списку використаних джерел.

Перший розділ містить детальне визначення вебдоступності, класифікацію користувачів з обмеженими можливостями, а також типові помилки, які впливають на загальний стан доступності у вебдодатках. Окрему увагу приділено останнім трендам і нормативно-правовим актам у даній сфері, що підкреслюють її важливість.

У другому розділі висвітлено існуючі методи статичного аналізу доступності у вебдодатках, які, зокрема, використовуються у великих компаніях, виділено їх переваги та недоліки. Також розглянуто рішення, що реалізують перераховані методи. Окрім цього, виділено дослідження одного з динамічних методів аналізу вебдоступності і сформовано постановку задачі даної роботи.

У третьому розділі описано і реалізовано власний алгоритм аналізу вебдоступності, який комбінує вищезазначені методи статичного аналізу з динамічним, що дає змогу досягти більшого відсотку виявлення помилок доступності.

Наостанок, четвертий розділ містить опис конфігурації і роботи створеної підсистеми для динамічного тестування вебдоступності у системі CI/CD на прикладі реального вебдодатку.

РОЗДІЛ 1: ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Визначення вебдоступності, актуальність проблеми

Доступність (англ. accessibility, a11y) – це практика створення будь-якого контенту (включно з вебсайтами і мобільними додатками) таким чином, щоб він був можливим для сприйняття і використання якомога більшою кількістю людей. При цьому, доступність – це не лише про людей з обмеженими можливостями: адаптованість контенту є значною перевагою також для пересічних людей, що користуються мобільними пристроями або ж мають повільне інтернет-з’єднання.

Втім, сьогодні, доступність вже не є суто моральною чи етичною нормою. Багато країн мають законодавство, яке вимагає виконання норм вебдоступності. Найбільш відомими прикладами є США (Закон про американських інвалідів, ADA та Розділ 508 Закону про Реабілітацію США, Section 508) та Європейський Союз (Директива про доступність вебсайтів та мобільних додатків). І якщо раніше ці вимоги стосувались лише державного сектору, то зараз їх вплив відчули і звичайні приватні компанії – кількість судових позовів щодо доступності вебсайтів у 2022 році побила новий рекорд [1], що свідчить про стрімку актуалізацію даного напрямку. Найбільш відомими прецедентами є позов проти компанії Netflix у січні 2012 року, що призвів до виплати компанією компенсації у розмірі близько $800,000, а також колективний позов проти Массачусетського технологічного інституту (MIT) і Гарвардського університету у 2015 році на суму більше 1.5 мільйонів доларів США, що звинувачував їх у ненаданні точних і вичерпних субтитрів для матеріалів онлайн-курсів.

Насправді, очевидно також, що адаптація вебдодатків для людей з обмеженими можливостями є економічно доцільною для бізнесу. У своєму маркетологічному дослідженні «A Hidden Market: The Purchasing Power of Working-Age Adults With Disabilities», Michelle Yin, Dahlia Shaewitz, Cynthia Overton та Deeza-Mae Smith дійшли до висновку, що великі компанії

позбавляють себе мільярдних прибутків, поки їх ресурси залишаються недоступними [2].

З технічної ж точки зору, доступність – це комбінація низки вимог до контенту, основними з яких є:

1. Дизайн: контент має бути логічно зв'язаним і однаково доступним та виразним у всіх його станах і їх можливих комбінаціях, включаючи особливі для людей з обмеженими можливостями. Типовий приклад – кнопка з іконкою, що має унікальний вигляд у таких станах: звичайний, звичайний у фокусі, вимкнений, активний, активний у фокусі, а також у різних формах вад зору, зокрема – дальтонізму;
2. Семантика: всі елементи, особливо інтерактивні, мають бути реалізовані з використанням стандартних засобів мови розмітки HTML, або ж повністю дублювати всю їх функціональність і логіку;
3. Навігація: при використанні будь-якого методу взаємодії з контентом (як-от лише з допомогою клавіатури), вся функціональність має бути однаково доступною;
4. Дублювання контенту, що може бути ефективно сприйнятий лише певною групою користувачів. Типові приклади: субтитри для відео, додаткові пояснення для контенту, що важко сприймається без контексту, альтернативний текст для картинок, графіків, тощо.

Отже, доступність – це не лише про просту ідею зробити світ кращим, а про великі перспективи як для людей з обмеженими можливостями, так і для бізнесу.

1.2 Категорії користувачів з обмеженими можливостями

За даними Центрів з контролю та профілактики захворювань в США за 2023 рік, до 1 з 4 людей у Сполучених Штатах Америки мають хоча б одну інвалідність [3]. Якщо говорити про доступність, найбільш поширеними є наступні категорії користувачів:

1. Люди з когнітивними вадами і обмеженою здатністю до навчання. Насправді, таких користувачів серед людей з обмеженими можливостями – більшість. Вади цієї категорії включають проблеми з увагою, швидкістю оброблення інформації, коротко- і довготривалою пам’яттю, здатністю до логічного мислення і заключення висновків, обробкою мови і математичних операцій. Цікавим фактом є те, що синдром дефіциту уваги з гіперактивністю, який особливо притаманний молодому поколінню, також відноситься до цієї категорії. Користувачі даної категорії не потребують жодного спеціального програмного забезпечення для використання вебсайтів.
2. Користувачі з вадами зору. Сюди відносяться такі інвалідності: повна або часткова сліпота, частковий (протанопія, дейтеранопія, тританопія і їх комбінації – погіршене сприйняття червоного, зеленого і синього кольорів відповідно) або повний дальтонізм (чорно-біла палітра). Не можна забувати також про епілепсію, збудниками якої можуть бути багато сучасних вебсайтів. Люди з вадами зору користуються програмами, що призначені для читання екрану (screen readers) – VoiceOver, JAWS, NVDA – які дозволяють з використанням клавіатури навігувати вмістом вебсайту так само, як виконується перемикання між інтерактивними елементами з клавіатури.
3. Користувачі, що користуються програмним забезпеченням «speech to text».
4. Люди з вадами слуху. Ця категорія користувачів також користується послугами ПЗ «speech to text», але у більшості випадків це зводиться до читання (можливо, автоматично згенерованих) субтитрів.
5. Люди з порушеннями моторики. Це можуть бути як захворювання, особливо притаманні людям похилого віку: артрити, артрози, параліч, так і, безперечно, відсутність кінцівок. Такі люди не можуть користуватися комп’ютерними мишами чи трекпадами і змушені взаємодіяти з інтерактивними елементами з використанням механізму фокусу (стан

елементу, коли він готовий приймати ввід від користувача), клавіатури, зокрема – клавішами Tab, Space та Enter, або пристрою, що її емулює.

Втім, варто також пам’ятати про тимчасові інвалідності: обмежена моторика кінцівок після операції, тимчасова сліпота, глухота, тощо. І навіть це – ще не все. У кожного була в житті хоч одна ситуація, коли, до прикладу, треба було подивитись відео або послухати голосове повідомлення, проте обставини або оточення робили це неможливим – як-от шум в метро робить неможливим прослуховування подкастів без субтитрів або текстової альтернативи. Справді, це також вважається проблемою з доступністю.

1.3 Класифікація типових проблем доступності

Варто зазначити, що існують всесвітньо визнані стандарти вебдоступності, які і диктують правила доступного контенту. Найбільш відомим з цих стандартів є World Wide Web Consortium Web Content Accessibility Guidelines (W3C WCAG) [4], і саме ним будемо керуватись у даній роботі. Отож, даний стандарт містить 78 критеріїв доступного контенту, які поділяються на 4 групи: сприйняття (perceivable), керованість (operable), зрозумілість (understandable) і сумісність (robust), що разом формують 4 принципи такого контенту і відомі за акронімом POUR. Окрім категорій, критерії також поділяються на 3 рівні: A, AA, і AAA, де A – найосновніший і найважливіший рівень, без якого відповідні групи користувачів взагалі не зможуть сприймати інформацію і взаємодіяти з вебсайтом, а AAA – рівень «найкращих практик», що зазвичай ігнорується. Таким чином, у сумі маємо 87 критеріїв доступних вебсайтів (Рисунок 1.1).

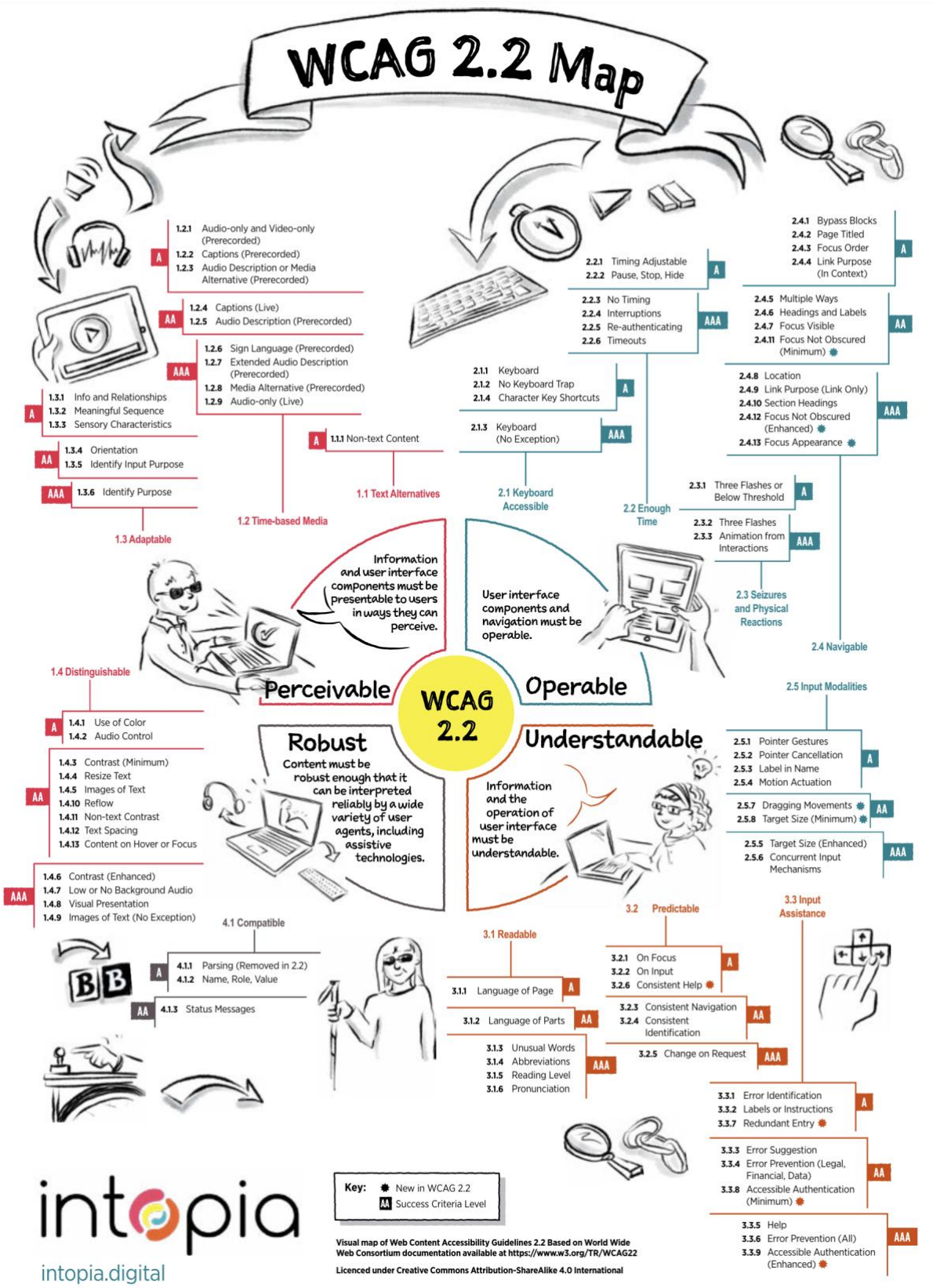


Рисунок 1.1 – Мапа критеріїв доступного контенту WCAG, поділених за категоріями

Звісно, частина цих критеріїв є специфічними для окремих видів вебдодатків або обов’язковими для виконання лише у випадку, коли необхідно досягти максимального рівня доступності (наприклад, для сайтів державного сектору). Втім, можемо виділити 10 найбільш поширених проблем, за версією ADA Site Compliance [4].

1. Навігація. Цей пункт включає навігацію, що імплементована з використанням посилань, які не мають відповідної ролі або не згруповані у єдиний компонент, що робить її визначення такими програмами, як читачі екрану, неможливим, а навігацію з клавіатури – неефективною, адже при кожному завантаженні сторінки користувачам необхідно пройти через безліч посилань щоб нарешті дібратись до основного контенту. Сюди ж відноситься відсутність у сайтів механізму під назвою Skip Links – це спеціальний блок посилань, який дозволяє миттєво пропустити елементи навігації і перейти до бажаного блоку вебсторінки (Рисунок 1.2).

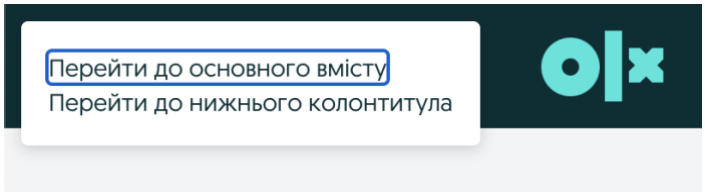


Рисунок 1.2 – Приклад блоку Skip Links

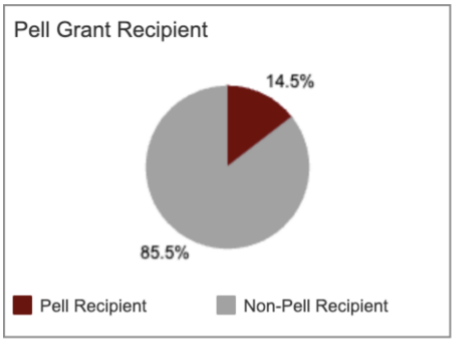
2. Таблиці. Пересічні розробники часто нехтують правилами реалізації таблиць в HTML, не використовуючи спеціальні теги для відповідних частин таблиці, які автоматично структурують її і для читачів екрану. Особливо гостро ця проблема стоїть у складних таблицях зі змішаними даними, багаторівневими і об’єднаними комірками, і нарешті – вкладеними таблицями. Сюди ж відносяться таблиці з можливістю сортування, зміни порядку рядків/стовпців і відсутнім тегом <caption>, що має містити короткий текстовий опис таблиці.
3. Недостатній контраст елементів (Рисунок 1.3). В термінології доступності, контраст елементів включає мінімальне співвідношення контрасту

кольору переднього плану (текст, іконка, тощо) до заднього плану (фон) як 3:1. До того ж, задовільне значення цього співвідношення може бути ще більшим для малих елементів. Якщо ж елемент інтерактивний – додається вимога достатнього співвідношення контрасту фону кнопки до загального фону. Насправді, до цього пункту також відносяться погано видимі індикатори активних інтерактивних елементів і колір інформативних іконок.

Рисунок 1.3 – Перелік правил для доступного контрасту елементів

4. Альтернативний текст для графічного контенту. На жаль, ця проблема є актуальною навіть для ресурсів, що турбуються про доступність, з двох причин: часто контент створюють самі користувачі, які не замислюються про необхідність альтернативного тексту для їх зображень, а також через те, що створення альтернативного тексту для картинок і, особливо, складних графіків або мап, є досить трудомістким завданням, особливо якщо ці елементи є інтерактивними. Ба більше, часто динамічну графіку неможливо описати простим текстом – для цього її треба дублювати таблицею або списком (Рисунок 1.4), що досі може сприйматись розробниками абсурдно. Втім, варто зазначити, що з поширенням AI-технологій, ця проблема може бути з легкістю вирішена: у пакеті програм

Microsoft Office, до прикладу, при вставці користувачем картинки, програма автоматично генерує її точний текстовий опис.



Alt Text: A pie chart of pell grant recipients with two slices.

Data Table:

Pell Grant Status	Percent of Total Enrollment	Enrollment Headcount
Pell Recipient	14.5%	5,162
Non-Pell Recipient	85.5%	30,428

Рисунок 1.4 – Приклад альтернативного тексту для простого графіку

5. Неправильний порядок інтерактивних елементів. Ця проблема з’явилась з появою у технології CSS можливості вільного позиціонування елементів на сторінці з використанням, до прикладу, властивості position. Як наслідок, це призводить до того, що порядок елементів у коді сторінки і на екрані відрізняються: користувач очікує, що наступним інтерактивним елементом буде певна кнопка, яка візуально знаходиться поруч з поточним елементом, але насправді в дереві сторінки вона є останньою, і натиснувши Tab, активним елементом стає не очікувана кнопка, а зовсім інший інтерактивний елемент.
6. Відсутні ARIA-атрибути. Це спеціальні HTML-атрибути, що починаються з акроніму aria-* (accessible rich internet applications) і призначені для того, щоб надати або деталізувати призначення і роль елементу для читачів екрану, зокрема у випадках, коли немає можливості використати семантично правильні відповідники. Типовим прикладом використання цих атрибутів є кнопка, яка відкриває модальне вікно. Без додаткових даних, вона буде інтерпретована читачем екрану як звичайна кнопка. Втім,

додавши лише 2 атрибути – aria-expanded=”false” і aria-haspopup=”dialog” – читач екрану сповістить, що ця кнопка відкриває діалогове вікно і в даний момент воно є закритим.

7. Недосяжні інтерактивні елементи. Часто розробники зловживають атрибутом tabindex, що відповідає за порядок елементу у списку інтерактивних елементів, тим самим ненароком повністю виключаючи його з цього списку. Насправді, ситуація може бути і прямо протилежною – невидимі або сховані інтерактивні елементи отримують фокус.
8. Відсутній або недостатній вміст інтерактивних елементів. Ця проблема є водночас поширеною і маловідомою, адже перед нею вразливі лише люди з вадами зору і когнітивних здібностей. Типовий приклад – посилання з текстом «Learn more», яке часто зустрічається у контексті нової функціональності або повідомлення про певну помилку. Для користувачів з вадами зору, такий текст є недостатнім, бо їм важко визначити його контекст, особливо якщо до цього вони вже зустрічали у застосунку посилання з ідентичним текстом. Користувачі ж з когнітивними вадами взагалі рідко можуть зрозуміти суть цього посилання, адже просто не мають змоги згадати, який текст передував цьому посиланню.
9. Недоступні форми. Безперечно, найважливіша частина більшості вебсайтів, адже саме через форми користувачі передають свої дані. Тут проблеми завжди починаються з відсутності асоціації між елементами форми, як-от назвами полів, самими інтерактивними елементами, допоміжними інструкціями і помилками валідації, і закінчуються слабким візуальним зв’язком між елементом форми і відповідною помилкою валідації. Окремої уваги заслуговують такі складові форм, як селектори з випадаючими списками, drag & drop елементи і повноцінні текстові редактори, оскільки такі елементи переважно є «самописними» через відсутність або примітивність семантичних аналогів.

10. Вставка контенту відмінного від HTML. До такого відносяться PDF-файли, Word-документи, презентації, тощо, які мають обмежену підтримку користувачів з обмеженими можливостями.

Отже, проаналізувавши типові проблеми доступності, можна зробити висновок про те, що для їх вирішення варто завжди починати з дизайну (UI/UX). Саме дизайн є найбільш вагомою і «неповороткою» складовою вебсторінок, оскільки на його переробку у великому застосунку можуть піти роки, адже дизайн – це не лише про візуальну складову застосунку, а і про спосіб використання кожного компоненту додатку окремо і в цілому. До того ж, дизайн має бути консистентним, що не дозволяє змінити один екземпляр компоненту в ізоляції, при цьому не зачіпаючи інші. Після візуального дизайну і користувацького досвіду іде контентний дизайн (CD) – це, за своєю суттю, вся текстова частина застосунку, включно зі створенням додаткового/альтернативного тексту для людей з обмеженими можливостями. І лише після цього естафету приймає безпосередньо код вебзастосунку, де мають задовольнятися вимоги семантики, правильного порядку інтерактивних елементів, і адаптації контенту для людей з обмеженими можливостями шляхом додавання різноманітних атрибутів і механізмів.

Висновки до розділу 1

Перший розділ роботи присвячено дослідженню предметної області вебдоступності. В ньому детально розглянуто визначення вебдоступності та її актуальність у сучасному світі, що зумовлено як морально-етичними аспектами, так і законодавчими вимогами багатьох передових країн. Проаналізовано категорії користувачів з обмеженими можливостями і відповідні проблеми, з якими вони зіштовхуються на сучасних вебсторінках. Також розглянуто основні критерії вебдоступності згідно з міжнародним стандартом W3C WCAG і наведено найбільш поширені проблеми доступності вебсайтів. На основі аналізу цих проблем зроблено висновок, що вирішення питань доступності варто розпочинати з перевірки і переробки дизайну, оскільки саме він є найвагомішою складовою, що впливає на зручність користування вебзастосунком для всіх категорій користувачів. Таким чином, забезпечення вебдоступності є не лише етичним обов'язком, але й економічно вигідною стратегією, що сприяє залученню ширшої аудиторії, включаючи людей з обмеженими можливостями. Це дозволяє бізнесу уникати потенційних юридичних проблем, судових позовів та втрати прибутків, а також сприяє створенню інклюзивного цифрового середовища.

РОЗДІЛ 2: АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ТЕСТУВАННЯ ДОСТУПНОСТІ

Ми вже дізналися, що вебдоступність – це досить об’ємне поняття, що передбачає виконання десятків різноманітних правил, на ручну валідацію яких навіть для невеликого вебсайту можуть піти тижні. Саме тому у цій сфері вже давно сформувався набір створених ентузіастами і великими компаніями спеціалізованих інструментів, як для тестування окремих правил так і стану доступності в цілому.

2.1 Спеціалізовані рішення для тестування окремих правил доступності

Інструменти для тестування окремих правил доступності, хоч і є вузькоспеціалізованими, та все ж варті окремої згадки, адже вузька спеціалізація робить їх простими у використанні і напрочуд якісними. Тож найпопулярнішим інструментом для перевірки співвідношення контрасту кольорів серед інженерів з доступності є Color Contrast Analyzer (CCA) [5] від компанії TGPi. Окрім самого значення, програма надає результати перевірки для можливих типів контенту (текст різних розмірів, іконки, тощо) з указаним кольором (Рисунок 2.1).

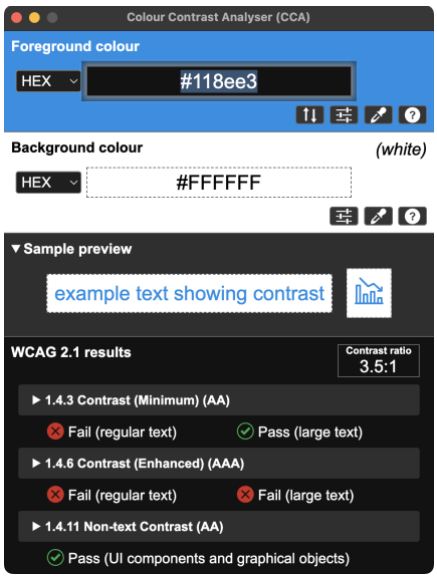


Рисунок 2.1 – Інтерфейс програми Color Contrast Analyzer

Альтернативним інструментом для перевірки співвідношення контрасту є Color Contrast Checker від WebAIM [6]. На відміну від CCA, цей інструмент представлений у вигляді вебсайту (Рисунок 2.2), втім має ідентичну функціональність.

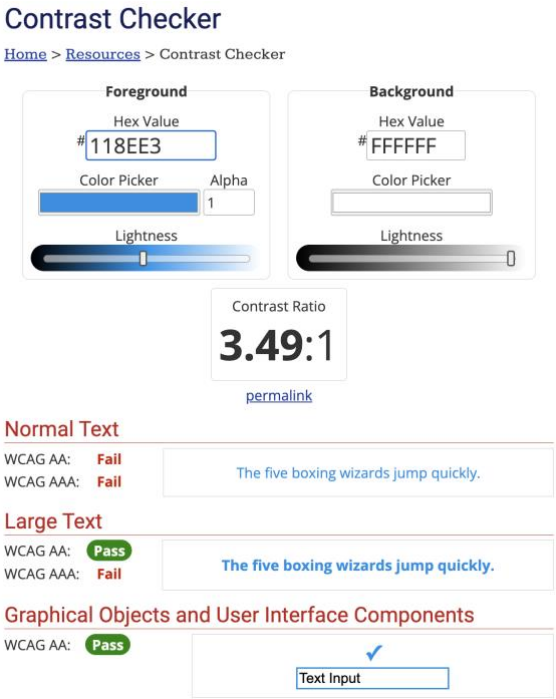


Рисунок 2.2 – Інтерфейс вебсторінки Color Contrast Checker

Ще одним інструментом для роботи з контрастом є Tanaguru Contrast-Finder [6]. Цей застосунок працює за зворотнім принципом: вказавши бажані кольори фону і переднього плану разом з цільовим значенням співвідношення контрасту, програма підбере найближчі до вказаного кольори, які будуть задовольняти вказаним вимогам. Проте, логічно, що цей застосунок може використовуватись лише для роботи з невеликими вебсайтами, які не мають своєї дизайн-системи і палітри кольорів.

Нарешті, останнім застосунком для перевірки контрасту є Button Contrast Checker від Aditus [7]. За своєю моделлю, Button Contrast Checker є стороннім API, що дозволяє проаналізувати відповідність кольорів кнопок та посилань на наданій вебсторінці правилам вебдоступності. Перевагою застосунку є те, що він

намагається виокремити CSS-стилі, застосовані до інтерактивних елементів, і вже за ними проводити аналіз того, чи є кольори доступними. Втім, такий підхід також має низку недоліків:

- Стилі часто містять властивості, як-от тіні, що заважають коректному аналізу. Ба більше, різні браузери надають власні стандартні стилі фокус-індикаторів, і може виникнути така ситуація, коли в Google Chrome фокус-індикатор є абсолютно доступним, а в Safari від Apple він взагалі відсутній. Саме тому раціонально буде перевіряти елементи у стані фокусу в кожному окремому браузері і не робити висновок лише за наявними стилями.
- У сучасних вебсайтах нерідко використовується псевдоклас :focus-within, що дає можливість перенести фокус-індикатор на будь-який батьківський елемент інтерактивного елементу, що зазвичай використовується у нестандартних компонентах. Таким чином, аналізуючи стилі самого інтерактивного елементу, можна прийти до неправильного результату.

2.2 Комплексні рішення для статичного тестування доступності

Окрім вузькоспеціалізованих рішень, існує велика кількість повноцінних (часто комерційних) рішень для комплексного тестування доступності вебсайтів, імплементованих за однією з трьох моделей:

- Розширення для браузера (англ. devtools) – допоміжний інструмент для розробників, що дозволяє проаналізувати стан доступності відкритої у браузері вебсторінки і згенерувати текстовий звіт;
- Стороннє API – на вхід приймає адресу вебсторінки, оброблює її на сторонніх серверах і повертає текстовий звіт. При цьому, з очевидних причин, не підтримує вебсайти/додатки, запущені локально. Достеменно невідомо, у якому середовищі відбувається тестування – у браузерному, чи лише шляхом валідації коду сторінки;

- Автономний застосунок із вбудованим інтерфейсом командного рядка (англ. CLI) – також приймає на вхід адресу вебсторінки (підтримує локальні вебсторінки/додатки) і повертає текстовий звіт. При цьому, може як емулювати поведінку браузера, так і виконувати лише валідацію коду сторінки.

Тож перший представник комплексного рішення – Wave від WebAIM [8]. Реалізує одразу всі моделі: браузерне розширення, стороннє API і автономний застосунок (Wave Engine). Насправді, це є великою перевагою, адже з усіх моделей лише остання може використовуватися для постійного відслідковування розробниками стану доступності вебдодатку, зокрема у системі CI/CD. Втім, у даній роботі розглянемо лише браузерне розширення Wave Chrome Extension, адже інші моделі цього рішення розповсюджуються на комерційній основі і працюють за ідентичним з браузерним розширенням принципом. Отже, розширення Wave може виявляти великий спектр помилок доступності, серед яких: картинки з недостатнім або відсутнім альтернативним текстом, недостатній контраст текстових елементів, неправильний порядок заголовків, відсутній або недостатній текст/контент кнопок і посилань, а також неправильне використання атрибутів і ролей. Окрім цього, користувачу надається можливість вручну оцінити порядок інтерактивних елементів (англ. focus order) та структуру документу (заголовки, навігація, основний контент, нижня навігація, тощо). Наостанок, розширення пропонує експортувати звіт у форматах HTML, JSON та XML, або ж скористатися вбудованою функціональністю, яка є інтерактивною, що значно спрощує пошук цільових елементів на сторінці, адже у звітах їх репрезентація є досить обмеженою і майже завжди представляє собою XPATH або CSS селектор. Проте, на жаль, у комплексних вебдодатках з великою кількістю елементів, цей інтерактивний звіт є непридатним для використання (Рисунок 2.3).

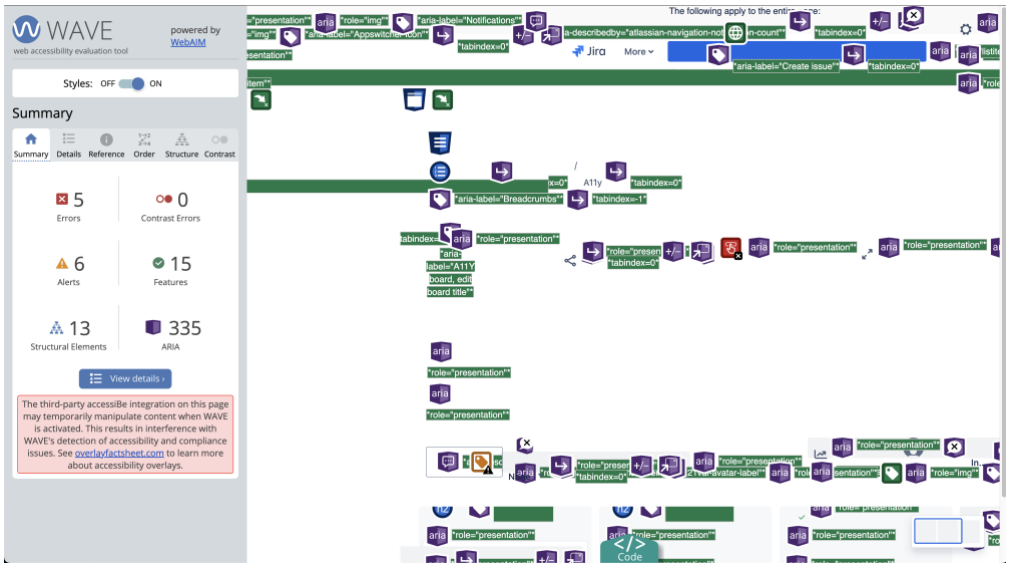


Рисунок 2.3 – Приклад невдалого інтерфейсу звіту роботи розширення Wave

Наступним відомим рішенням для комплексного аналізу доступності є браузерне розширення Lighthouse [9] з відкритим вихідним кодом, яка насправді стало популярним завдяки іншій своїй функції – оцінці продуктивності завантаження і роботи вебсторінки. Тож це розширення також може виявляти помилки доступності, зумовлені відсутнім альтернативним текстом у зображень, недостатнім текстом посилань, низьким контрастом текстових елементів, неправильними комбінаціями ролей і ARIA-атрибутів, та багатьма дрібнішими помилками. Окрім цього, розширення пропонує користувачу вручну перевірити відповідність сторінки наступним критеріям: коректний порядок інтерактивних елементів, всі інтерактивні елементи можуть отримати фокус, фокус не є заблокованим або «загубленим» (така ситуація, коли після взаємодії з певним елементом, цей елемент зникає з дерева сторінки і фокус повертається на її початок), а також те, що всі невидимі елементи не можуть приймати фокус.

Ще одним безкоштовним інструментом для комплексного тестування доступності є Accessibility Insights [10]. Розроблене компанією Microsoft, рішення швидко здобуло популярність на ринку, адже окрім браузерного розширення воно пропонує окремий застосунок для тестування доступності Windows-додатків. Насправді, це є досить сильною перевагою, адже такий

застосунок потенційно може мати доступ до внутрішнього API операційної системи, що значно полегшить взаємодію з такими її модулями, як режим підвищеного контрасту і читачі екрану, проте у даній роботі фокусуємось на вебдоступності. Тож розширення Accessibility Insights має 3 основні режими роботи, що в цілому виявляють ті ж помилки, що і попередні рішення:

- FastPass – режим для автоматизованої перевірки трьох найбільш популярних проблем (контраст, альтернативний текст, текст посилань) за максимум 5 хвилин;
- Quick Access – режим для змішаної перевірки 10 поширених проблем, що займає до 30 хвилин;
- Assessment – ще один режим для повного керованого (ручного) тестування доступності сторінки.

Окрім цього, розширення має вбудовані прості, натомість дуже важливі утиліти, які дозволяють швидко візуалізувати компоненти навігації, рівні заголовків, альтернативні тексти, і, що вкрай важливо, те, як буде виглядати сторінка для людей з різними формами дальтонізму. Ба більше, також є можливість візуалізувати порядок інтерактивних елементів (Рисунок 2.4), хоч і не без допомоги користувача, що допоможе виявити інтерактивні елементи, які не отримують фокус або ж знаходяться за межами видимої області екрану.

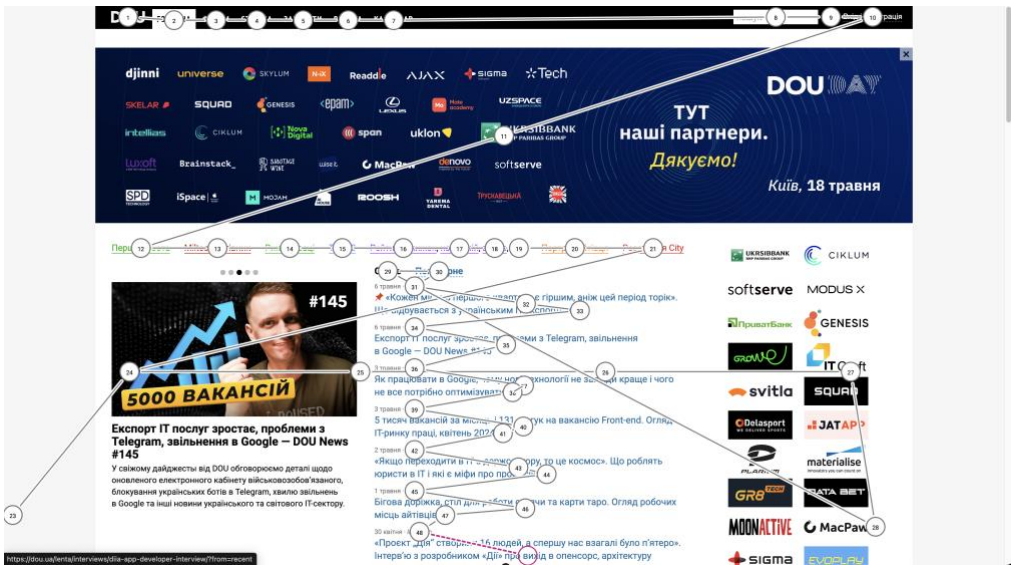


Рисунок 2.4 – Приклад візуалізації порядку інтерактивних елементів

Окремої уваги заслуговує той факт, що розширення Accessibility Insights працює на базі ПЗ axe-core, про яке піде мова далі.

Нарешті, останнє рішення для комплексної перевірки доступності вебсайтів – лідер ринку Axe від компанії Deque Systems [11]. Фактично, це є набір рішень: браузерне розширення axe DevTools, повноцінний застосунок для мануального тестування axe Auditor, а також сервіс для моніторингу стану доступності вебдодатків axe Monitor. Окрім програмних продуктів, компанія також відома своїми послугами з проведення аудитів доступності і виправлення помилок, для чого має окремі команди інженерів. Важливо також те, що Deque Systems бере активну участь у розвитку і популяризації сфери доступності, зокрема – проводить регулярні вебінари і конференції. Отож всі програмні рішення компанії, включно з axe DevTools, яке, насправді, не вирізняється серед попередніх рішень за функціональністю, оглянутих у цьому розділі, базуються на бібліотеці під назвою axe-core. Бібліотека реалізована за моделлю автономного застосунку з інтерфейсом командного рядка і має безліч варіацій для різноманітних мов програмування і середовищ виконання, таких як Selenium, Puppeteer і Playwright, що безперечно робить її простою у використанні для налаштування автоматизованого тестування доступності в системі CI/CD. І саме тому надалі у даній роботі будемо розглядати axe-core як базовий інструмент для тестування доступності, адже він розповсюджується на безоплатній основі, а також має відкритий вихідний код і активно підтримується спільнотою розробників.


```
{
  "documentTitle": "Scrum: Teams in Space - Agile Board - Jira",
  "pageUrl": "jira-active-sprint.html",
  "issues": [
    {
      "code": "aria-allowed-attr",
      "type": "error",
      "typeCode": 1,
      "message": "Elements must only use allowed ARIA attributes
(https://dequeuniversity.com/rules/axe/4.2/aria-allowed-attr?application=axeAPI)",
```

```
    "context": "<div class=\"ghx-swimlane-header\" aria-label=\"Swimlane for custom: TIS Developer
Love\" tabindex=\"0\" data-swimlane-id=\"18\"><div class=\"ghx-heading\" title=...</div>",
    "selector": "#ghx-pool > div:nth-child(1) > div",
    "runner": "axe",
    "runnerExtras": {
      "description": "Ensures ARIA attributes are allowed for an element's role",
      "impact": "critical",
      "help": "Elements must only use allowed ARIA attributes",
      "helpUrl": "https://dequeuniversity.com/rules/axe/4.2/aria-allowed-attr?application=axeAPI"
    }
  }
}
```


Приклад 2.1 – Зразок результату аналізу доступності інструменту axe-core

Отже, ми розглянули найбільш відомі сервіси для комплексного автоматизованого тестування доступності вебзастосунків, деякі з яких навіть можуть бути інтегрованими в систему CI/CD. Проте, на жаль, їх всіх об’єднує наступний факт: ці інструменти у автоматизованому режимі без допомоги користувача здатні виконувати лише примітивний статичний аналіз вебсторінки, тобто перевірку HTML-коду із застосованими до елементів стилями. Це означає, що за стандартного способу використання, такі інструменти як axe-core будуть аналізувати лише ті вебсторінки, шляхи до яких були надані їм у якості вхідних даних, що призводить до глобальної проблеми – виконується аналіз лише того контенту, що присутній на сторінці після її завантаження. Тобто весь контент діалогових вікнон, випадаючих списків, спливаючих повідомлень, будь-яких інших прихованих компонентів що стають доступними після певної взаємодії зі сторінкою, загалом – динамічних елементів – проаналізовано не буде. Ба більше, всі перераховані у розділі інструменти не здатні перевіряти і взаємодіяти з елементами форм – однією з основних складових взаємодії користувачів з вебресурсами. І хоч здається, що існує просте рішення цієї проблеми – створити тести, де після початкового аналізу сторінки виконується певна дія, а далі проводиться повторний аналіз – воно не буде повноцінним, адже перераховані інструменти не призначені для тестування динамічного контенту і не зможуть

виявити такі проблеми, як відсутність спеціальних ARIA-атрибутів у кнопки, що відкриває модальне вікно, або ж відсутність сповіщень для читачів екрану про кількість знайдених опцій випадаючого списку після здійсненого пошуку. Насправді, це наштовхує нас на ще один висновок: комплексному аналізу доступності вебсайту має піддаватись весь існуючий контент і функціональність вебзастосунку в цілому, адже лише таким чином можна досягти повного покриття всього спектру проблем доступності і передбачити можливі складнощі, з якими стикаються користувачі з обмеженими можливостями під час активної роботи з вебресурсами. Цей факт, до речі, підтверджується дослідженням Deque Systems про те, що автоматизоване тестування доступності здатне виявити лише 57% всіх існуючих проблем [13].

Проблему з тестуванням обмеженої частки контенту вебсторінки намагається вирішити комплекс рішень Pa11y від Pa11y Team [14], що налічує такі інструменти: Pa11y – інтерфейс командного рядку для тестування доступності наданої сторінки чи сторінок, Pa11y CI – адаптований CLI для системи CI/CD, а також Pa11y Dashboard – аналог axe Monitor для динамічного моніторингу стану доступності вебсторінок. За сам аналіз доступності у цих інструментах відповідає вже знайомий нам axe-core, втім Pa11y реалізує важливе доповнення – можливість побудови «скрипту» з використанням вбудованого набору інтуїтивно зрозумілих команд, що буде виконуватись перед безпосереднім аналізом сторінки. Це дозволяє автоматизувати виконання простих дій на вебсторінці, як-от натискання кнопки або заповнення полів форми заданими значеннями, що значно спрощує написання тестів з перевірки доступності. Втім, варто зазначити, що ці дії будуть виконані саме перед початком аналізу, тобто проблема перевірки динамічного контенту все ще залишається актуальною.

```
pa11y(url, {
  actions: [
    'set field #username to exampleUser',
    'set field #password to password1234',
```

```
            'click element #submit',
            'wait for path to be /myaccount'
        ]
    });
```

Приклад 2.2 – Зразок команд для виконання перед аналізом доступності у Pa11y

Наостанок, ще одним підходом до вирішення проблеми тестування недоступного контенту, що набирає популярності у великих компаніях, є автоматичне створення тестів з використанням axe-core для так-званих Storybook'ів – атомарних ізольованих середовищ для тестування окремих компонентів користувацького інтерфейсу (Рисунок 2.5).

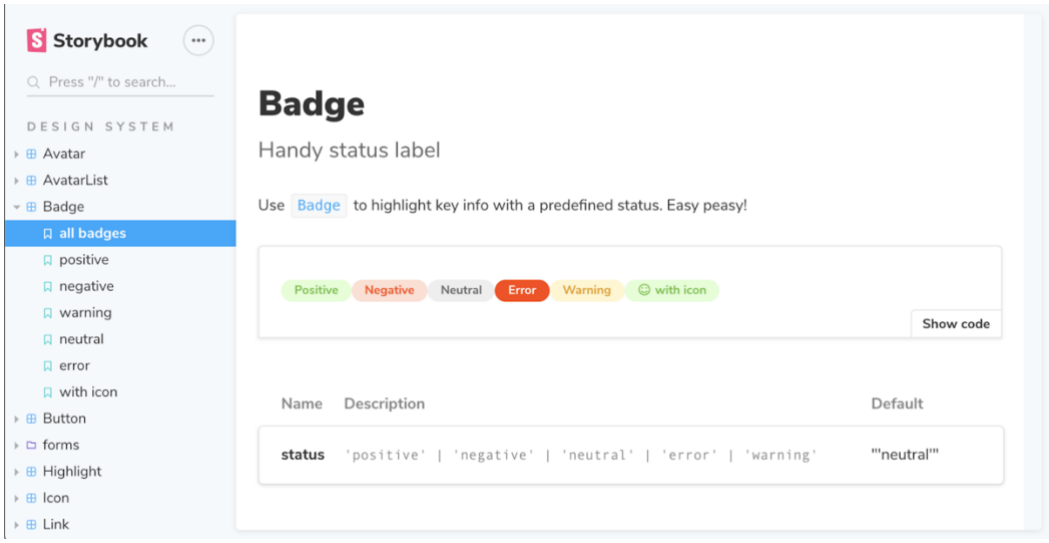


Рисунок 2.5 – Приклад storybook'у для дизайн системи

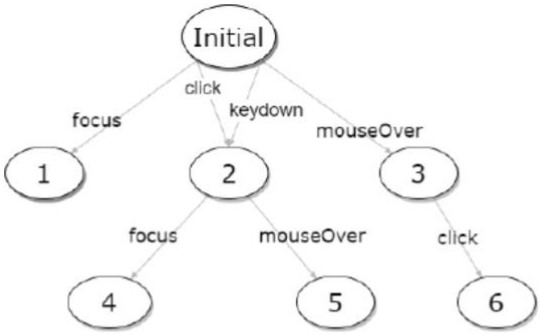
Storybook'и не є новинкою для IT-індустрії і вже довгий час використовуються інженерами для демонстрації роботи їх вебкомпонентів, що є досить зручною альтернативою документації і шляхом взаємодії різних команд розробників, коли виникає потреба швидко розібратись з новим компонентом, його можливими станами і способом використання. Таким чином, покривши тестами доступності ці середовища, можна дійсно швидко зібрати статистику і перелік проблем доступності, особливо якщо піддавати тестуванню всі можливі стани цих компонентів (як приклад, випадаючий список у наступних станах: закритий,

відкритий, у процесі завантаження опцій, з помилкою, тощо) і відфільтрувати результати на наявність дублікатів. Втім, якщо цей підхід і справді вирішує проблему з тестуванням неактивного динамічного контенту, ми все ще не можемо гарантувати, що будуть протестовані всі можливі стани перелічених компонентів. Але більш важливо те, що порушується принцип тестування кінцевого користувацького досвіду, адже компоненти перевіряються на доступність в ізоляції один від одного, в той час як доступність передбачає тісний зв'язок між пов'язаними елементами.

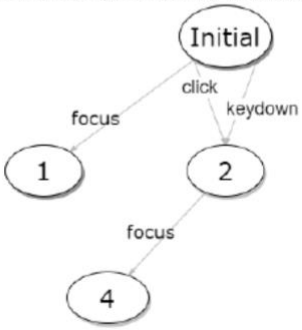
2.3 Динамічний підхід у тестуванні доступності

Ми вже дійшли висновку про те, що окрім статичного тестування доступності має також відбуватись динамічне – взаємодія з усіма інтерактивними елементами (або хоча б найбільш важливими) і перевірка того, як вебсайт реагує на ці дії. Адже лише такий підхід зможе виявити навіть таку тривіальну проблему як відсутність на кнопці обробників натискання клавіш Enter/Space, що призводить до ситуації, коли звичайний клік мишкою на кнопці активує обробник події, прив'язаний до неї, а аналогічна дія з клавіатури – вже ні. Отож, цим підходом зацікавились розробники з The MITRE Corporation, і запропонували власне дослідження [15], що фокусується на теорії і математичному обґрунтуванні ідеї порівняння дерева станів взаємодії з інтерактивними елементами. У своїй роботі, Trevor Bostic, Jeff Stanley, John Higgins, Daniel Chudnov, Justin F. Brunelle та Brittany Tracy використали підхід вебсканування аби розробити фреймворк під назвою Demodocus, який здатен автоматично виявляти і взаємодіяти з інтерактивними елементами на сторінці тими методами, які доступні звичайним користувачам і людям з обмеженими можливостями. Таким чином, у результаті маємо великий граф станів (Рисунок 2.6), вершини якого представляють стани контенту вебсторінки, а ребра – можливі дії, що були застосовані до сканованих інтерактивних елементів. Отже,

розділивши отриманий граф на два окремих, що будуть мати лише ті ребра, які відповідають типовим взаємодіям звичайних людей і користувачів з обмеженими можливостями, та порівнявши їх на еквівалентність, зможемо дійти висновку про те, чи є певна функціональність доступною.



(a) The graph generated by simulating a user without a disability.



(b) The graph generated by simulating a user with a disability. States 3, 5, and 6 are not included.

Рисунок 2.6 – Приклад порівняння дерев станів взаємодії з контентом

Окрім порівняння еквівалентності графів взаємодії з інтерактивними елементами, фреймворк також надає можливість визначення складності таких взаємодій для кожного елементу. Тобто маємо вже зважений граф (Рисунок 2.7), де вага ребра може мати значення у проміжку від 0 до 1 включно, де 0 – це абсолютно недоступний для людей з обмеженими можливостями стан (як-от такий, що досяжний лише дією mouseOver, тобто наведення мишкою), а 1 – абсолютно доступний. Очевидно, що перемноживши ваги ребер, що ведуть до бажаного стану, можемо дізнатись індекс його доступності.

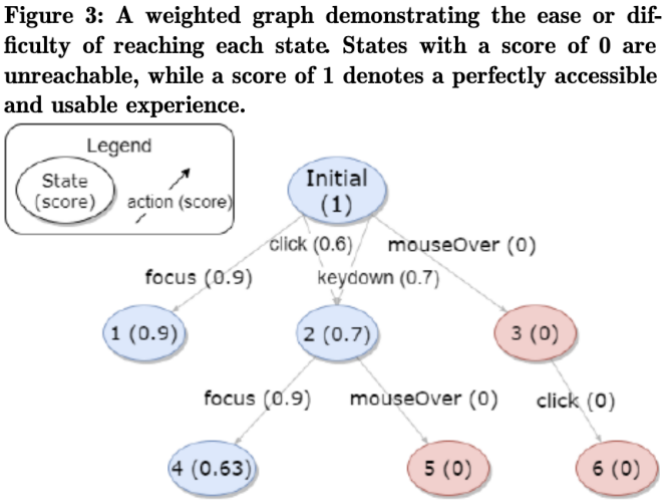


Рисунок 2.7 – Приклад зваженого дерева станів взаємодії з контентом

Фреймворк Demodocus є важливим кроком у розвитку динамічного аналізу вебдоступності і відкриває нові можливості у автоматизованому тестуванні контенту без достеменного розуміння його природи. Втім, на жаль, практика показує що це дослідження поки є занадто теоретичним, і запропонований підхід тестує буквальну досяжність контенту, у той час як всі інші складові вебдоступності, як-от прості ARIA-атрибути, залишаються поза увагою інструменту. До того ж, без з’ясування приблизної природи аналізованого контенту, буде складно визначити як список можливих дій, що можуть бути застосовані до нього, так і бажаний результат цих дій. Нетривіальним прикладом таких елементів може бути механізм drag & drop, який вимагає клопіткої роботи для того, щоб зробити його доступним.

Якби ми все ж мали змогу визначати тип активного контенту вебсторінки, перед нами відкрилися б нові можливості у виявленні нетривіальних проблем доступності, адже знаючи тип контенту і згрупувавши загальновідомі правила вебдоступності за типами контенту, його можна перевірити на виконання відповідної групи правил. У якості прикладу візьмемо певну кнопку, що при активації відкриває діалогове вікно. Ми знаємо, що така кнопка, окрім

стандартної семантики кнопки, має мати атрибути aria-haspopup=”dialog” і aria-expanded=”false”. А діалогове вікно повинне відповідати наступним критеріям:

- Заголовок вікна має мати перший рівень, а також унікальний ідентифікатор;
- Контейнер модального вікна має мати атрибут role=”dialog” і містити ідентифікатор його заголовку у якості значення атрибуту aria-labelledby;
- Весь вміст вебсторінки, окрім самого модального вікна, має бути невидимим для допоміжних технологій (англ. Assistive technology, AT), що забезпечується атрибутом aria-modal=”true” і механізмом під назвою focus trap;
- Після появи діалогового вікна на екрані, фокус має автоматично перестрибнути або на сам контейнер вікна/заголовок (який матиме атрибут tabindex=”0”), або ж на перший інтерактивний елемент, у разі якщо перед ним немає інших важливих елементів;
- Діалогове вікно має реагувати на натискання клавіші Escape, що повинна закривати його;
- Після закриття вікна, фокус повинен повернутись на елемент, що його відкрив, або ж, якщо такий елемент вже не присутній на сторінці (у випадку, якщо він був частиною випадаючого списку, що вже закрився) – на елемент, що відкрив попередній активний елемент.

Це і є приклад групування пов’язаних правил вебдоступності.

2.4 Постановка задачі

Отже, проаналізувавши як існуючі рішення для комплексного статичного аналізу доступності вебсторінок, які здатні виявляти до 57% всіх існуючих проблем доступності, так і запропонований The MITRE Corporation підхід динамічного тестування, який може бути застосований для знаходження недосяжного динамічного контенту, можемо дійти висновку, що наразі ці два

підходи слабо пов’язані між собою, що не дозволяє виявляти повний спектр проблем вебдоступності. Тож завданням даної кваліфікаційної роботи є розробка рішення, що призначене для комбінованого аналізу вебдоступності, яке:

- зможе виконувати базовий статичний аналіз доступності;
- буде використовувати інтерфейс стандартних веббраузерів і емулюватиме роботу реального користувача;
- реалізує алгоритм вебсканування, подібний до наявного у фреймворку Demodocus, для автоматизованого обходу всіх інтерактивних елементів, що можуть бути наявними на сторінці;
- альтернативно вебскануванню, надасть опцію створення сценаріїв, подібно до Pa11y, за якими буде відбуватись тестування;
- запропонує алгоритм виявлення змін у контенті сторінки після виконання певної дії, при цьому ігноруючи елементи, що постійно змінюють свій стан, як-от слайдери/каруселі;
- запропонує алгоритм визначення типу активного контенту сторінки;
- матиме вже готові групи критеріїв доступності, поділені за типом контенту, до якого вони відносяться;
- перевірятиме активний контент на відповідність визначеній групі критеріїв;
- надаватиме звіт за результатами роботи.

Висновки до розділу 2

У другому розділі роботи було розглянуто існуючі рішення і підходи для точкового ручного і комплексного статичного аналізу доступності вебсторінок, серед яких найбільш відомим є рушій axe-core. Втім, хоч ці рішення і можуть якісно виявляти великий спектр проблем доступності, було визначено, що вони не призначені для автоматизованої взаємодії з інтерактивними елементами вебсторінок, а отже як не можуть сканувати контент, досяжний лише у результаті таких дій, так і не здатні протестувати сам формат взаємодії і його доступність.

У якості доповнення до статичного аналізу також розглянуто дослідження групи розробників з компанії The MITRE Corporation, що пропонує динамічний підхід у тестування доступності шляхом взаємодії з інтерактивними елементами вебсторінки і побудови на основі отриманих даних дерева її станів. Такий метод дозволяє точно визначити досяжність певного динамічного контенту вебзастосунку, проте ще не гарантує його доступність. Після цього сформовано постановку задачі даної роботи, що передбачає створення підсистеми для динамічного тестування вебдоступності у системі CI/CD, яка, окрім виконання статичного аналізу контенту, буде також автоматизовано взаємодіяти з інтерактивними елементами і перевіряти як досяжність активного контенту, так і виконання правил його доступності.

РОЗДІЛ 3: РОЗРОБКА ПІДСИСТЕМИ ДЛЯ ДИНАМІЧНОГО ТЕСТУВАННЯ ВЕБДОСТУПНОСТІ У СИСТЕМІ CI/CD

3.1 Опис алгоритму

Отже, у нас є мета створити алгоритм, який зможе виявляти всі можливі проблеми вебдоступності, що зустрічаються на практиці. Для цього нам необхідно поділити ці проблеми на дві категорії: ті, що можна виявити без жодних взаємодій з інтерактивними елементами сторінки (як-от відсутній або недостатній альтернативний текст, порожні посилання, недостатній контраст, тощо), і ті, що потребують певних дій з боку користувача (модальні вікна, динамічний контент, випадаючі списки, drag & drop). Нескладно помітити, що у першому випадку перераховані лише окремі правила доступності, а у другому – групи правил, або ж шаблони/патерни поведінки контенту. Таким чином, можемо поділити всі існуючі критерії доступності вебсайтів на такі, що можуть бути виявлені при статичному тестуванні, і, відповідно, лише при динамічному. Наявність двох видів аналізу означає, що і алгоритм буде поділятись на 2 частини – статичний аналіз та динамічний аналіз. При цьому, ці два види аналізу мають змінювати один одного: спочатку виконується статичне тестування доступності оригінального вмісту сторінки після її завантаження, далі взаємодіємо з першим інтерактивним елементом, очікуючи що контент зміниться – це динамічне тестування, потім, у разі появи нового контенту, проводимо статичне тестування цього контенту, і так далі. Таким чином, у варіанті з вебскануванням, у якості результату отримаємо дерево станів, схоже на те, що реалізоване в Demodocus, а у варіанті наперед створеного сценарію – ланцюг станів. Втім, якщо з користувацькими сценаріями все просто – ми виконуємо лише ті взаємодії, що вказані у них, для вебсканування маємо визначитися з наступним:

- спосіб перебору інтерактивних елементів: в ширину чи в глибину;
- спосіб опрацювання посилань;

- обмеження для уникнення нескінченних циклів;
- умова зупинки алгоритму.

Якщо говорити про спосіб перебору, то варто порівняти переваги та недоліки існуючих варіантів. У випадку перебору в ширину, значно легше контролювати хід дій, що були виконані, адже при переборі в глибину гілка взаємодії з першим інтерактивним елементом може закінчитись взагалі на сторінці з URL-адресою, відмінною від початкової. З іншого боку, при переборі в глибину значно простіше слідкувати за стеком інтерактивних елементів, а також тестувати «зворотній шлях», що чудово демонструється прикладом вкладених випадаючих списків (Рисунок 3.1).

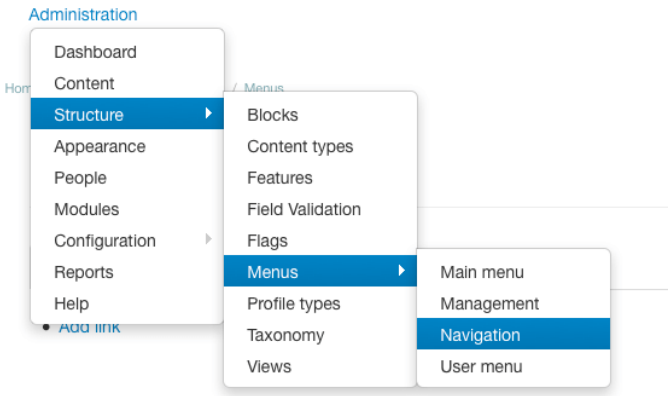


Рисунок 3.1 – Приклад вкладених випадаючих списків

Це приводить нас до висновку, що найкраще буде комбінувати методи перебору дерева інтерактивних елементів, враховуючи складність нового контенту при балансуванні дерева станів. У результаті, якщо після взаємодії з інтерактивним елементом на сторінці з'являється контент низької складності, як-от продемонстрований вище випадаючий список або модальне вікно, можемо продовжувати перебір в глибину. У випадку ж випадаючої панелі або, скажімо, панелі вкладок, які можуть містити сотні інтерактивних елементів всередині них, краще продовжити перебір в ширину перш ніж повертатись до аналізу такого контенту.

Наступною проблемою є аналіз посилань, а точніше результатів взаємодії з ними. Посилання, якщо, звісно, використані за семантичним призначенням, створені для зміни локації, що тягне за собою зміну URL-адреси і, найважливіше, контексту вебзастосунку. Саме тому логічно буде обробляти посилання вже після того, як інших непроаналізованих інтерактивних елементів на сторінці не залишилось. Звісно, на сучасних вебсторінках можна знайти багато посилань, що використані не за призначенням: вони не мають, або ж мають порожній атрибут href, що відповідає за цільову адресу, а також до них прикріплений обробник події click, що перетворює ці посилання на кнопки. Оскільки такі посилання найчастіше і справді будуть мати роль кнопки, їх аналіз можна не відкладати на останню чергу, але варто пам’ятати, що навіть у звичайної семантичної кнопки може бути обробник, що напряму звертається до API браузерної історії і змінює локацію, втім це неможливо визначити не активувавши елемент. Наостанок, варто звернути увагу на те, що багато сучасних вебдодатків реалізовані за моделлю Single Page Application (SPA), що, при переході на іншу сторінку, передбачає лише зміну URL-адреси, втім, контекст і, часто, більшість DOM-елементів, залишаються незмінними, що значно полегшує аналіз.

При переборі контенту вебсторінки після взаємодії з певними елементами легко потрапити у ситуацію, коли аналізований контент вже був протестований і тестується повторно, що також призводить до дублювання виявлених проблем цього контенту. Якщо говорити про повторний аналіз, простим і логічним рішенням буде запам’ятовувати вже протестовані елементи, або ж помічати їх такими за допомогою певного зарезервованого HTML-атрибуту. Втім, це не може бути єдиним рішенням, адже, знову ж, SPA-застосунки мають реактивну природу – вебсторінка є динамічною, а її елементи мають властивість «перерендерюватись», тобто видалятись і створюватись знову. Звісно ж, це призводить до видалення всіх даних, що були додані до елементу поза рамками коду застосунку. Саме тому зарезервовані атрибути треба дублювати, до прикладу, зберіганням унікальних селекторів вже проаналізованих елементів.

Щодо дублювання виявлених проблем, варто згадати про один з пунктів філософії рушія axe-core: «Рушій не буде звітувати про хибно позитивні результати». Це означає, що якщо алгоритм сумнівається у тому, чи проблема дійсно є реальною, він радше проігнорує її ніж скаже про неї користувачу. Дійсно, цей підхід має право на існування, особливо у великому бізнесі, втім він також піддається критиці, адже неправильні результати, по-перше, можуть бути визначеними такими помилково, а по-друге, справді помилкові результати завжди можна проігнорувати вручну, бо всім відомо, що здобути інформацію значно складніше, ніж видалити її.

Нарешті, умов зупинки алгоритму може бути кілька і вони будуть залежати від користувача. Якщо є потреба протестувати увесь вебдодаток, то найпростіше буде вважати його таким тоді, коли всі знайдені сторінки цього додатку будуть протестовані. Для цього у якості фільтру можна надати регулярний вираз, що буде перевіряти кожну нову URL-адресу на приналежність домену вебзастосунку. Якщо ж під час аналізу ми потрапляємо на іншу сторінку – просто повертаємось назад. За схожим принципом можна ще більше обмежити аналіз, вказавши конкретні URL-адреси у якості фільтру.

3.2 Вибір інструментів

Отож, для реалізації застосунку, що буде емулювати поведінку реального користувача, нам потрібен браузер, з яким ми зможемо програмно взаємодіяти, зокрема – перемикатись між інтерактивними елементами шляхом натискання клавіші Tab, заповнювати поля форм, робити скріншоти/записувати відео екрану, тощо. Окрім самого браузеру, має бути можливою взаємодія зі вмістом відкритої вебсторінки засобами мови JavaScript. Безперечно, поточну URL-адресу і історію також маємо мати змогу контролювати. Також, для тестування мають бути доступними всі популярні браузери: Google Chrome, Mozilla Firefox, Microsoft Edge, Apple Safari. Тож, рішення, що реалізують перераховану вище

функціональність називаються бібліотеками для автоматизації тестування у браузерах, і найбільш відомими їх представниками на сьогодні є Selenium WebDriver, WebdriverIO, Cypress, Puppeteer та Playwright. Проте, варто зазначити, що більшість цих рішень передбачають саме створення тестових сценаріїв з використанням тестових бібліотек, у той час як для даної роботи необхідний саме режим прямої взаємодії з рушієм, адже тести, фактично, будуть самі себе створювати.

Selenium WebDriver [16] є найстаршим прикладом рішення для автоматизації тестування у браузерах, що також означає велику спільноту розробників і широку підтримку. Випущений у 2004, Selenium WebDriver підтримує всі популярні браузери, включно з Apple Safari, має відкритий вихідний код і підтримує більшість мов програмування. Працює рішення за технологією WebDriver, що є протоколом віддаленого керування веббраузерів, запропонованим W3C. Для кожного браузера існує своя реалізація цього протоколу, і відповідно, його драйвер: для Google Chrome – ChromeDriver, Mozilla Firefox – Geckodriver, Microsoft Edge – Edge Driver, Apple Safari – SafariDriver. На жаль, протокол не має підтримки багатьох важливих функцій, як-от роботи з мережею – штучно знизити або призупинити доступ браузера до інтернету, що було б корисним для тестування доступності, є неможливим. Отже, Selenium WebDriver має кілька рівнів, що взаємодіють між собою: WebDriver-клієнт, драйвер вибраного браузера і сам браузер. Втім, така кількість рівнів суттєво впливає на швидкодію рішення, що робить його найповільнішим серед перерахованих вище. До того ж, хоч Selenium WebDriver не очікує від користувачів створених тестових сценаріїв і може використовуватись суто у режимі рушія, його API є досить громіздким і не має вбудованої підтримки деяких важливих функцій, як-от запис екрану.

WebdriverIO [17] є сучасним «спадкоємцем» Selenium WebDriver’a, що покликаний підвищити його продуктивність і зменшити поріг входу за рахунок простішого і ширшого API, хоч і має підтримку лише мов програмування,

пов’язаних з JavaScript. В цілому, WDIO базується на тому ж WebDriver протоколі, хоч і має підтримку нового протоколу DevTools для браузеру Chromium. Окрім автоматизації взаємодії з веббраузерами, WebdriverIO також неочікувано підтримує автоматизацію тестування MacOS та Electron-застосунків: у документації є приклад тестування вбудованого в операційну систему від Apple калькулятора. Окремо варто зазначити, що WebdriverIO має офіційну документацію з налаштування тестування доступності з використанням axe-core, а також вбудовану підтримку побудови візуалізації порядку інтерактивних елементів на сторінці, що наводить на думку про те, що це рішення слабо фокусується на його основній меті і допускає підтримку досить обмеженого набору можливостей як для інших платформ, так і цілих сфер тестування.

Cypress [18], заснований у 2015 році, є популярним фреймворком для End-to-End тестування, що став відомим завдяки простоті налаштування і використання. Одна з найбільших переваг цього рішення в тому, що воно передбачає написання тестів мовою JavaScript, а також не має необхідності у налаштуванні окремого проекту для конфігурації браузерів, адже технологія Cypress передбачає виконання коду тестів напряму у браузері, де працює код тестованого застосунку, на відміну від WebDriver, що зокрема позитивно впливає на швидкодію. Окрім цього, технологія реалізує підтримку низькорівневої взаємодії з рушієм браузера, тобто доступні такі функції, як штучне обмеження швидкості роботи процесора і мережі. Ще одна перевага Cypress – його інтерфейс (Рисунок 3.2), що має вбудований дебаггер і дозволяє одночасно з виконуваними тестами бачити стан тестованої вебсторінки. На жаль, технологія підтримує лише браузери Chrome, Edge та Firefox, а також доступна лише у варіанті середовища для тестування, тобто вимагає на вхід тестових сценаріїв.

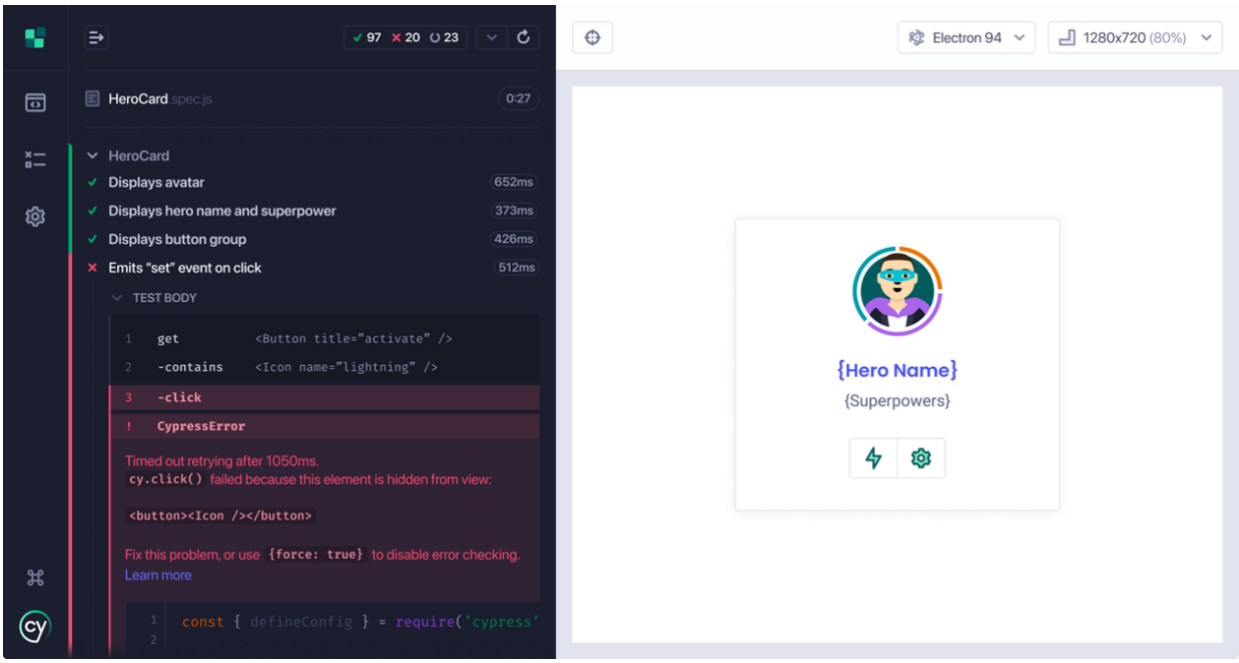


Рисунок 3.2 – Графічний інтерфейс програми Cypress

Puppeteer [19] – це Node.js бібліотека для взаємодії з браузерами на базі Chromium через протокол DevTools. Засноване у 2012 компанією Google, рішення здобуло популярність завдяки своїй простоті і орієнтованості на обмежений набір браузерів (Google Chrome та Microsoft Edge), що є вдалим вибором для проектів, які хотіли б налаштувати мінімальне E2E тестування. До того ж, підтримувана Google, технологія є добре оптимізованою для роботи з їх власними браузерами, що робить її лідером на ринку за швидкодією. Хоч Puppeteer і не вимагає роботи лише у форматі виконання тестових сценаріїв, але окрім обмеженої підтримки браузерів, також не має можливості запису екрану і контролю стану мережі.

Нарешті, Playwright від Microsoft [20], створений у 2020 – ультимативне рішення для автоматизації тестування у всіх існуючих браузерах, що побудоване на базі протоколу DevTools для браузерів на базі Chromium, а також власного аналогічного протоколу для інших браузерів. Інструмент має відкритий вихідний код і підтримку популярних мов програмування: Java, Python, C#, JavaScript і TypeScript. Окрім можливості низькорівневої взаємодії з мережею, Playwright

також підтримує емуляцію мобільних пристроїв та запис екрану. Також, хоч і створена саме для виконання тестових сценаріїв, технологія має окрему бібліотеку playwright-core, що, подібно axe-core, є рушієм, який може бути використаний для безпосередньої взаємодії з браузером. Ба більше, Playwright підтримує автогенерацію тестових сценаріїв: достатньо почати запис і програма запам’ятає послідовність дій, що була виконана на сторінці, а також дозволить в інтерактивному режимі обрати необхідні перевірки. Така функціональність стане нам у пригоді після реалізації базового алгоритму, що дозволить не запускати щоразу довгий аналіз доступності всього застосунку, а виконувати вже згенеровані тести. До того ж, Playwright також має зручний інтерфейс, подібний до Cypress, з можливістю відслідкування ходу тестування і роботи зі станом застосунку. Отже, за співвідношенням переваг і недоліків, в даній роботі будемо використовувати бібліотеку Playwright.

3.3 Архітектура підсистеми

Безперечно, архітектура має велике значення у створенні будь-якого застосунку, адже що краще він спроектований, то простіше буде його підтримувати і розширювати у майбутньому. Зокрема, з метою полегшення підтримки, для реалізації застосунку має сенс використати мову програмування TypeScript і середовище виконання Node.js, адже, окрім керування браузером через інтерфейс Playwright, нам також необхідно буде взаємодіяти з елементами вебсторінок. В цілому, застосунок буде складатися з таких компонентів:

1. Набір утиліт для точкової роботи з DOM-елементами, їх стилями, інтерактивними елементами сторінки, а також скріншотами/записами екрану.
2. Непрямі залежності, що будуть підключатись до вебсторінки перед початком аналізу.

3. Набір окремих функцій, що виконують перевірку конкретних правил або груп критеріїв доступності. Функції будуть поділятись на такі, що виконують статичний та динамічний аналіз.
4. Система логування, що буде як збирати додаткові дані для аналізу роботи застосунку, так і виявлені помилки доступності, при чому не тільки у формі текстового звіту, а і з використанням скріншотів та записів екрану.
5. Система обробки помилок.
6. Менеджер стану застосунку, що, зокрема, відповідатиме за зберігання дерева станів застосунку.
7. Основний алгоритм програми, що буде поєднувати використання функцій, які виконують аналіз доступності.

У той час як відповідні утиліти та функції для аналізу доступності будуть розглянуті у наступних підрозділах, тут пропоную зосередитись на роботі із залежностями.

Сьогодні складно уявити великий застосунок, який би не використовував сторонніх бібліотек або сервісів, і, безперечно, реалізований інструмент не є виключенням. Однак, окрім прямих залежностей додатку, як-от бібліотек для роботи зі скріншотами та файловою системою, маємо ввести поняття непрямих залежностей, що будуть імпортуватись не у застосунок, а у вебсторінку, яка буде тестуватись. Це зумовлено тим фактом, що Playwright, як і інші рішення для автоматизації тестування браузерів, надає два способи взаємодії з DOM-деревом сторінки:

- Шляхом виконання команди evaluate, що приймає на вхід функцію або стрічку коду, яка буде виконана на боці сторінки.
- Шляхом імпортування скрипта у форматі стрічки або ж локального чи віддаленого JavaScript-модулю у тестовану сторінку.

І якщо перший спосіб є вдалим вибором для виконання коротких команд, як-от отримання поточного активного елементу, то, до прикладу, складний алгоритм

виявлення змін у контенті вебсторінки після взаємодії з певним інтерактивним елементом раціонально було б винести у окремий модуль, який буде імпортований у вебзастосунок, а далі звертатися до нього у середовищі Playwright. Окрім цього, бібліотеки для взаємодії з DOM-деревом сторінки ми також не можемо використовувати у середовищі Node.js, тож їх теж маємо імпортувати на боці сторінки. Отож, для цього створимо власне поле у об'єкті window, що буде зберігати як внутрішні (власні) так і зовнішні залежності, які будуть імпортуватись з розподіленої мережі доставки контенту (CDN).

3.4 Реалізація статичного аналізу доступності

Перед початком реалізації статичного аналізу доступності, варто нагадати, що цей тип аналізу має виявляти проблеми з доступністю статичного контенту, і для його виконання має бути достатньо лише HTML-коду сторінки та стилів. Як вже було сказано, статичне тестування доступності виконує рушій axe-core, тому, насправді, тут було б достатньо просто імпортувати його як залежність і отримувати від нього знайдені проблеми. Саме тому у цьому рішенні варто в першу чергу зосередитись на тих аспектах, які axe-core виконує неякісно або ж не виконує взагалі. Тож потенційними такими аспектами є:

- Поглиблений аналіз стану фокусу інтерактивних елементів.
- Виявлення проблем з порядком інтерактивних елементів, зокрема – «пасток для фокусу» і «загубленого фокусу».
- Виявлення інтерактивних елементів, що не можуть приймати фокус або приймають фокус і водночас є невидимими.
- Виявлення інтерактивних елементів з однаковим контентом у рамках однієї сторінки.
- Аналіз контенту, що дублює один одного, зокрема – виявлення декоративних іконок, що мають альтернативний текст ідентичний до видимого поруч.

- Виявлення контенту, що є слабо помітним у режимі підвищеного контрасту.
- Виявлення інформативного контенту, що розрізняється лише за кольором і є проблемою для дальтоніків.
- Перевірка наявності механізму Skip Links.
- Виявлення контенту, що некоректно відображається з увімкненим режимом збільшених текстових інтервалів.

Найскладнішим з цього списку є аналіз інтерактивних елементів, тому у даній роботі обмежимося лише ним.

Тож перша наша задача – розглянути всі інтерактивні елементами, по ходу аналізуючи кожен з них, і переконатися, що ми успішно досягаємо кінця сторінки. Тут нам у нагоді стане браузерна бібліотека tabbable [21], що допоможе знайти всі інтерактивні елементи на сторінці, аби дізнатися їх кількість. Бібліотеку підключимо як непряму зовнішню залежність, імпортуючи її з JSDelivr CDN [22]. Отже, коли вебсторінка, яка буде аналізуватись, вже завантажена, а також підключені наші власні залежності, на що виділяємо 5 секунд, переходимо до визначення того, який інтерактивний елемент отримає фокус. Для цього емулюємо натискання клавіші Tab, у результаті чого маємо активний елемент. Перед тим, як рухатись до наступного елементу, можемо одразу проаналізувати, чи є у поточного інтерактивного елементу індикатор його стану фокусу. Для цього робимо скріншот цього елементу, додаючи по його периметру 100 пікселів, адже пам’ятаємо про псевдоклас focus-within, за допомогою якого можна перенести фокус-індикатор на батьківський елемент. Тепер необхідно зробити скріншот цього ж елементу у його початковому стані, для чого треба зробити його неактивним. Проте, тут маємо кілька проблем:

- Якщо перемістити фокус на попередній чи наступний інтерактивний елемент, має з’явитися вже його фокус-індикатор, що при порівнянні скріншотів буде зайвим шумом.

- Якщо програмно перемістити фокус на певний невидимий елемент на початку чи в кінці сторінки, ми порушимо положення прокручування сторінки, що також вплине на порівняння скріншотів.

Отже, найкращим рішенням буде перед аналізом кожного інтерактивного елементу, створювати прямо біля нього невидимий тимчасовий елемент, що будемо називати локальним орієнтиром (англ. local guide), який буде приймати фокус під час створення скріншотів, а після аналізу – видаляти його. Таким чином, до того ж, ми можемо уникнути проблеми з деактивацією вікна Skip Links, яке зазвичай ховається при втраті фокусу з його внутрішніх інтерактивних елементів. Тож, на виході маємо 2 скріншоти інтерактивного елементу у звичайному та активному станах. Використаємо бібліотеку odiff [23] для їх порівняння і створення результуючого, третього скріншоту. Далі об'єднаємо всі три скріншоти в один задля зручності аналізу результатів (Рисунок 3.3). Таким чином, базовими засобами мови JavaScript, ми маємо змогу визначити елементи, що не мають жодного фокус-індикатора. У подальшому, цей алгоритм можна покращити, проаналізувавши саму різницю між двома скріншотами, втім, засобів даної мови вже буде недостатньо.

Рисунок 3.3 – Результат роботи алгоритму з перевірки наявності індикатора фокусу

Продовжимо роботу над перебором інтерактивних елементів. Для цього нам знадобиться цикл, в тілі якого емулюватиметься натискання клавіші Tab і проводитиметься перевірка поточного активного елементу. Умова закінчення циклу – досягнення лічильником значення загальної кількості інтерактивних елементів помноженого на 1.25 (на випадок можливих циклів), або ж досягнення наперед створеного глобального орієнтира (англ. global guide) – невидимого інтерактивного елемента в кінці сторінки. Таким чином, залишилось реалізувати виявлення пасток для фокусу і загубленого фокусу. У другому випадку достатньо перевіряти, чи після натискання клавіші Tab активний елемент не дорівнює елементу body – саме він стає активним при видаленні активного елементу зі сторінки. Тепер розглянемо випадок пастки для фокусу, іншими словами – механізму обмеження інтерактивних елементів, що можуть приймати фокус. Отже, при такому сценарії фокус рано чи пізно потрапить на елемент, що вже був протестований, і найпростіший спосіб це виявити – додавати певний HTML-атрибут після завершення його аналізу. Також маємо запам’ятовувати попередній інтерактивний елемент, аби мати змогу помітити закриваючий елемент цієї пастки. Після виявлення такої пастки, можемо проаналізувати батьківський елемент всіх цих інтерактивних елементів, і визначити, чи була пастка додана цілеспрямовано.

3.5 Реалізація динамічного аналізу доступності

Після виконання статичного аналізу контенту вебсторінки, маємо переходити до взаємодії з елементами. Таким чином, динамічним аналізом доступності будемо називати взаємодію з інтерактивним елементом, визначення контенту, що змінився, його типу, а також його тестування на виконання відповідної групи критеріїв доступності. Після динамічного аналізу має відбуватися вже статичний аналіз зміненого контенту. Оскільки реалізація взаємодії не є складною задачею – достатньо лише емулювати клік на

інтерактивному елементі – одразу перейдемо до алгоритму виявлення змін у контенті сторінки.

Отже, спочатку варто визначити, які зміни в DOM-дереві нам необхідно відслідковувати після натискання певної, до прикладу, кнопки. Безперечно, перш за все це додані елементи, і елементи, що були видалені. Також бувають випадки, коли контент, як-от модальне вікно, вже є у дереві сторінки, а після натискання кнопки він лише стає видимим. Це приводить нас до висновку, що маємо відстежувати такі зміни:

- Елемент був доданий на сторінку.
- Елемент був видалений зі сторінки.
- Елемент став видимим.
- Елемент став невидимим.

Якщо перші два пункти не є проблемними, то з відстеженням видимості вже не все так очевидно, адже за неї можуть відповідати як стилі елемента, так і HTML-атрибути. Також варто зазначити, що елемент може бути як повністю прихований за допомогою таких властивостей як display: none, opacity: 0, visibility: hidden, тощо, так і просто бути позиціонованим за межами видимої області екрану завдяки властивості position. І якщо у першому випадку нам достатньо скористатися вбудованою у WebAPI функцією checkVisibility, то у другому треба перевіряти координати елемента і порівнювати їх з координатами видимої області екрану (задача про перетин двох прямокутників). До того ж, варто пам’ятати, що невидимість батьківського елемента не означає невидимість нащадка, адже CSS-властивості можна перевизначати. Тож маємо створити власну функцію, яка буде перевіряти всі наявні у DOM-дереві елементи на видимість, і зберігати результат у спеціальному HTML-атрибуті, що дозволить нам, окрім присутності елементів у коді вебсторінки, відстежувати зміни лише значень одного їх атрибуту.

Насправді, коли маємо задачу відстеження змін у DOM-дереві вебсторінки, перший інструмент, що спадає на думку – Mutation Observer API. Це частина WebAPI, що дає змогу налаштувати простий обробник події зміни стану сторінки, що буде реагувати на зміни як структури сторінки, так і атрибутів елементів. Таким чином, при кожному спрацьовуванні даного обробника, достатньо зберігати сам елемент, що змінився, і додаткові дані, у певний масив. Втім, варто одразу помітити, що один і той самий елемент за час роботи обробника може як змінитись сам кілька разів, так і його атрибути або контент. Тому логічно буде створити мапу, де у якості ключа зберігатимемо елемент, що змінився, а у якості значення – набір записів про зміни. До того ж, цей набір необхідно реалізувати з використанням структури даних Set, що передбачає зберігання лише унікальних значень, адже у майбутньому нам знадобиться можливість пошуку і видалення записів, чого значно складніше досягти з простим масивом.

Отже, після того, як всі зміни зібрані і обробник події вимкнено, значення необхідно відфільтрувати, адже під час збору даних елементи могли змінити свій стан двічі, що у кінцевому результаті означає відсутність змін. До того ж, якщо взяти до прикладу модальне вікно, у сучасних застосунках ми гарантовано зустрінемо певний індикатор завантаження, і можливо навіть не один, який спочатку з’являється, а потім зникає, що лише додає шуму до зібраних даних. Ба більше, якщо говорити про відстеження видимості елементів, може статись таке, що батьківський елемент був схований, а разом з ним – всі нащадки. Відповідно, у результаті маємо зберегти лише дані про батьківський елемент. Насправді, навіть якщо елемент в процесі роботи було додано на сторінку, значення видимості його і його нащадків також буде вважатись зміненим. Врешті-решт, запустивши кілька циклів попарних перевірок елементів, нам вдасться їх відфільтрувати. Проте, на жаль, все ж залишиться сценарій, який не вдасться коректно обробити: якщо елемент було спочатку видалено, а потім додано знову (типовий ререндер) – це буде складно виявити через необхідність порівняння не

лише посилань, а і внутрішнього контенту та стану елементів. Саме через цей та сукупність попередніх факторів, було прийняте рішення про зміну підходу на порівняння двох HTML-дерев – до взаємодії з інтерактивним елементом та після, що, зокрема, дасть нам змогу уникнути фільтрації проміжних змін контенту. Тож для цього використаємо бібліотеку diffDOM [24], яка насправді також призначена для ефективного відтворення змін у DOM-дереві. Виявлені бібліотекою зміни поділяються на зміни до атрибутів, внутрішнього контенту елементів, переміщення групи елементів та їх повну заміну на інші. Такий підхід, окрім економії ресурсів, також дасть можливість зберегти оригінальне DOM-дерево, що допоможе нам у майбутньому при порівнянні станів застосунку, а також дозволить повторно аналізувати елементи, що були видалені зі сторінки. Отож, у результаті маємо 4 списки елементів: ті, що були додані/видалені зі сторінки, а також ті, що вже були на сторінці, але стали видимими/невидимими (Рисунок 3.4).

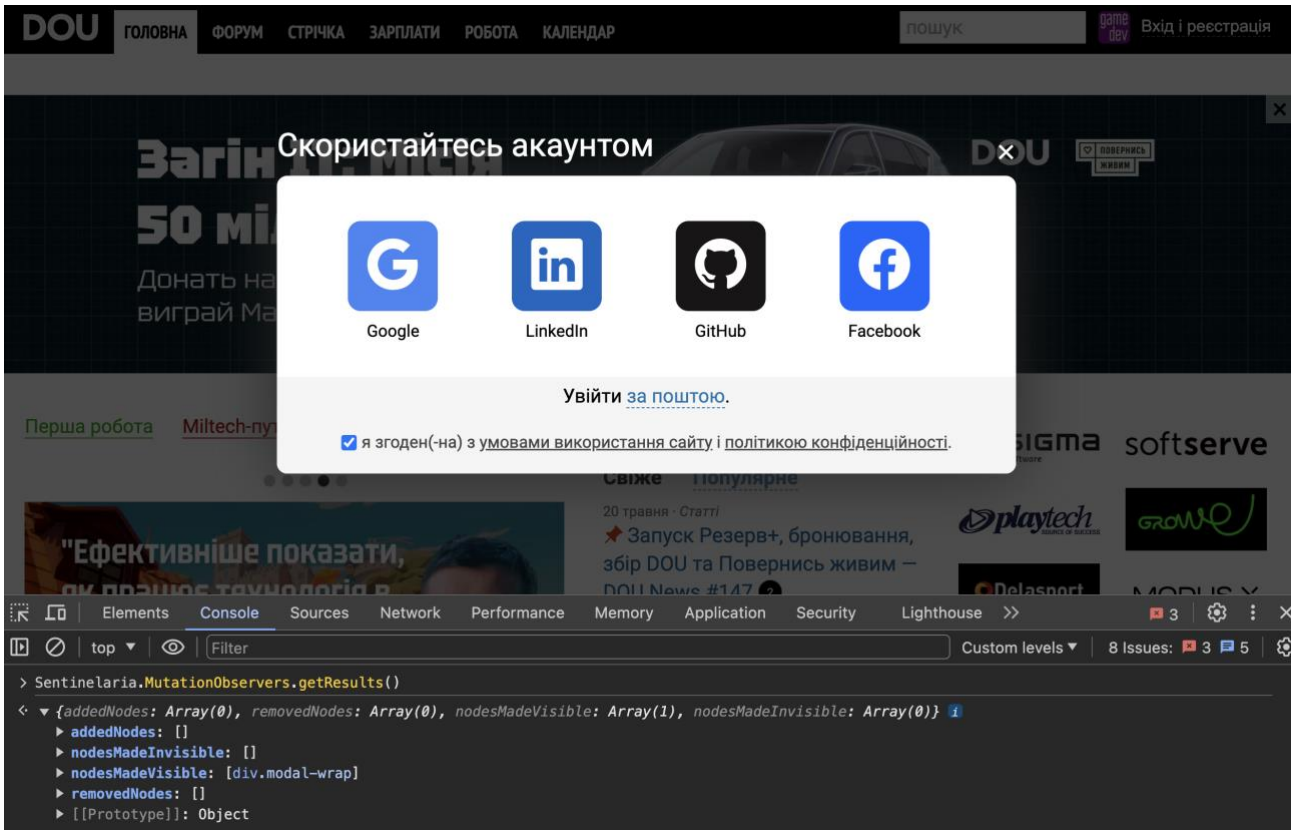


Рисунок 3.4 – Результат роботи алгоритму виявлення змін контенту після появи діалогового вікна на екрані

Ще одна проблема, що часто зустрічається на недоступних сайтах – контент, що постійно змінюється: слайдери, рекламні банери, iframe'и, тощо. На жаль, такі елементи при аналізі доступності і виявленні змін у контенті будуть постійно додавати небажаний шум у результати, тож маємо розробити стратегію для їх ігнорування. Перше, що спадає на думку – виділяти перед початком аналізу кілька секунд на виявлення елементів, що постійно змінюються, а потім маркувати їх зарезервованим HTML-атрибутом. Відповідно, під час аналізу будь-які елементи з таким атрибутом мають ігноруватися. На жаль, цей підхід має наступні недоліки:

- У випадку слайдера, найчастіше атрибут буде додано на батьківський елемент, стилі якого змінюються таким чином, щоб відображати певний слайд. Втім, зміни кожного слайду окремо, зокрема, зміни видимості, все ще не будуть ігноруватись;
- Якщо елемент постійно ререндериться, тобто видаляється і додається на сторінку, доданий атрибут одразу ж зникне.

Насправді, другий недолік можна виправити впровадивши додатковий засіб зберігання ігнорованих елементів – унікальний CSS-селектор. Використавши бібліотеку Simmer JS [25] для генерації за DOM-елементом його CSS-селектора, можна створити Set, що буде зберігати такі селектори, і перед аналізом елементу перевіряти, чи його селектор є ігнорованим. Отже, залишається перший недолік, за яким можемо зробити висновок про те, що зарезервований атрибут допоможе лише тоді, коли змінюється лише сам елемент з цим атрибутом, без урахування його нащадків, і ця проблема потребує подальшого дослідження.

Отож, ми закінчили роботу над визначенням змін у контенті після взаємодії з інтерактивним елементом. Тепер маємо розібратися з класифікацією можливих змін, аби дізнатись тип відповідного контенту. І почнемо з найгіршого випадку – коли змін виявлено не було. У такому разі можливі два варіанти: або з відповідним інтерактивним елементом треба взаємодіяти у інший спосіб (як-от з елементами механізму drag & drop, які можуть не реагувати на подію click), або

ж змінений контент був серед елементів, що ігноруються. У першому випадку – переходимо до тестування інших можливих дій, а другий випадок залишимо на подальше опрацювання. У разі ж, якщо зміни було виявлено, маємо визначити тип цього контенту з-поміж наступних:

- Діалогове вікно;
- Випадаюче меню;
- Модальне вікно (popup);
- Випадаюча панель;
- Tooltip;
- Combobox;
- Акордеон;
- Елемент <details>;
- Банер;
- Меню вкладок.

Насправді, для цього буде достатньо проаналізувати стилі, застосовані до даного контенту. Такі елементи, як діалогове вікно, випадаюче меню, модальне вікно, випадаюча панель, combobox і, у деяких випадках, банер – майже завжди будуть мати абсолютне позиціонування, реалізоване з використанням властивості position. У випадку діалогового вікна абсолютне позиціонування доповнюється так-званим backdrop-елементом, що затіняє фоновий контент. До випадаючих меню, панелей, модальних вікон і combobox’ів також зазвичай додається backdrop. Tooltip зазвичай матиме абсолютне позиціонування і відсутній backdrop. Нарешті, такі елементи як акордеон, тег <details> і меню вкладок завжди мають звичайне позиціонування, і їх можна розрізнити за наявністю схожих інтерактивних елементів поруч. Після визначення ймовірного типу активного контенту, ми будемо знати відповідні шляхи взаємодії з ним (як доступні так і ні), а також спосіб виходу з нього (як-от за допомогою клавіші Escape), що знадобиться при необхідності повернення до попереднього стану застосунку. Отож, у результаті маємо рекурсивну функцію scanNode, що на вхід

приймає елемент вебсторінки і, на прикладі аналізу діалогового вікна, повертає об’єкт, який містить тип контенту у форматі стрічки, посилання на цей елемент, а також, якщо існує – посилання на backdrop-елемент (Рисунок 3.5).

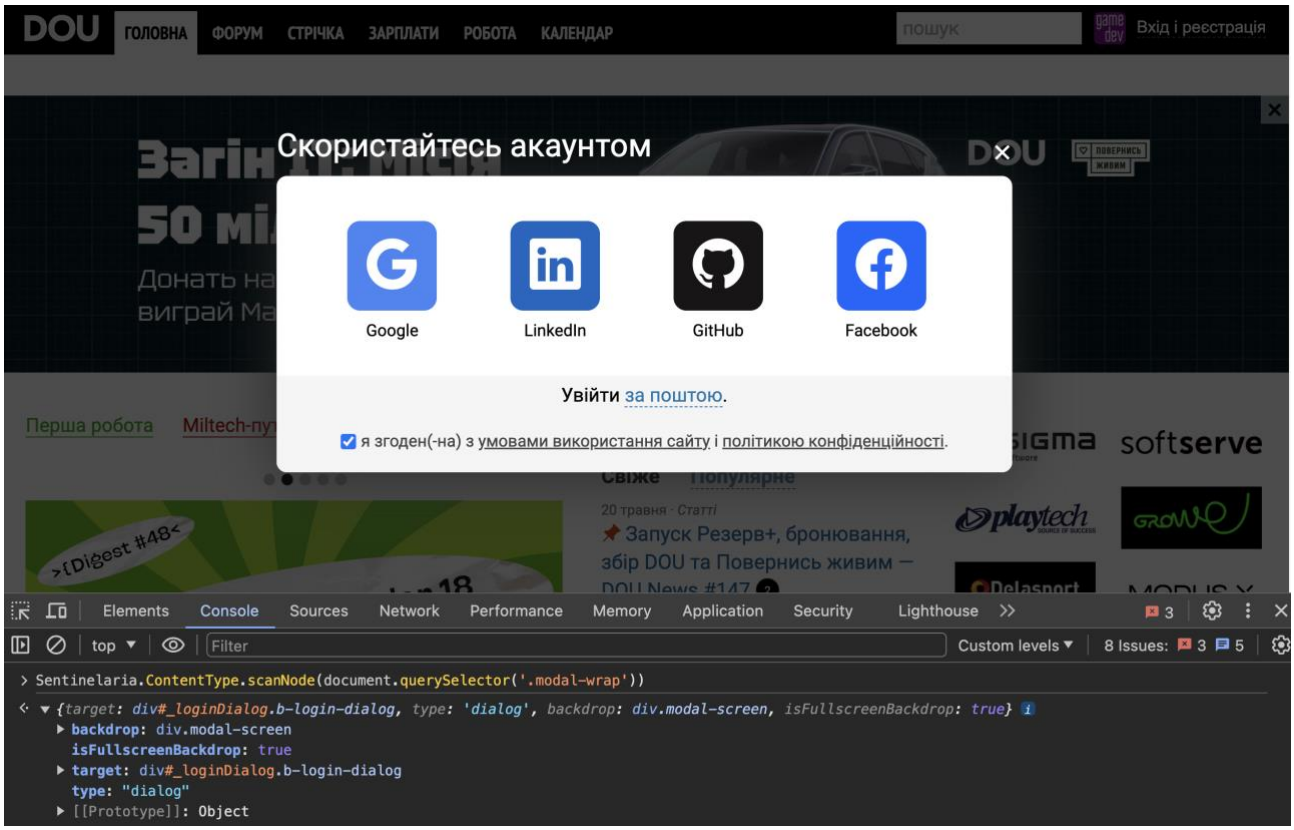


Рисунок 3.5 – Результат роботи алгоритму визначення типу контенту на прикладі діалогового вікна

Тепер, коли ми визначили тип активного контенту, можемо перевіряти його на виконання заздалегідь визначеної для діалогових вікон групи критеріїв доступності.

Висновки до розділу 3

Третій розділ даної роботи зосереджується на описі складових реалізованого рішення, зокрема, на загальному алгоритмі її роботи, архітектурі, виборі базових інструментів, а також на принципі роботи алгоритмів статичного і динамічного аналізу доступності. Визначено, що процес тестування буде реалізований у формі циклу, в якому статичний і динамічний аналіз будуть змінювати один одного після взаємодії з кожним інтерактивним елементом на сторінці. Попри певні складнощі, реалізовано складові алгоритму динамічного аналізу, що передбачає взаємодію з інтерактивним елементом, визначення змін у контенті сторінки, його типу і послідовну перевірку активного контенту на виконання відповідної групи критеріїв доступності, що дозволяє не лише впевнитись у досяжності цього контенту для користувачів з обмеженими можливостями, а і дізнатись стан його доступності. Всі зазначені алгоритми були об’єднані і реалізовані у підсистемі для динамічного тестування доступності вебдодатків у системі CI/CD.

РОЗДІЛ 4: ОПИС РОБОТИ ПІДСИСТЕМИ НА ПРИКЛАДІ РЕАЛЬНОГО ВЕБДОДАТКУ

4.1 Особливості конфігурації новоствореної системи

Настав час перевірити роботу підсистеми для динамічного тестування вебдоступності у системі CI/CD на прикладі реального застосунку. Втім, звичайно ж, перед її запуском нам належить створити мінімальну конфігурацію. Перш за все, варто відзначити, що задля забезпечення максимальної точності і швидкості тестування, тестований застосунок має бути розгорнутий або локально, або у середовищі, відмінному від того, що використовують кінцеві користувачі. Це зумовлено тим, що часто останнє передбачає додатковий захист від ботів і DDoS-атак, як-от рішення від Cloudflare [26], який значно ускладнює емуляцію дій реального користувача або ж зовсім блокує її. Також реалізована система наразі не підтримує роботу з iframe-елементами.

Отже, для тестування роботи підсистеми у даному розділі використаємо сайт відомого українського маркетплейсу Prom.ua [27], який хоч і знаходиться у production-середовищі, але вже є частково доступним, не має агресивного захисту від емульованих користувачів, має достатньо контенту різних типів і невелику кількість різних за своєю суттю сторінок. Тож для роботи системи нам необхідно вказати вхідну URL-адресу, з якої почнеться тестування: https://prom.ua/ua/. У якості фільтру URL-адрес надамо регулярні вирази, що будуть відповідати наступним сторінкам:

- https://prom.ua/ua/ - головна сторінка;
- https://prom.ua/ua/sc/military - формат URL-адреси вибраної однієї категорії товарів;
- https://prom.ua/ua/p* - формат URL-адреси певного товару;

Одразу підкреслимо, що ці адреси також враховують можливі параметри пошуку, що додаються після символу «?» - повторне сканування на таких

сторінках відбуватись не буде. Коли умови входу і зупинки надані – можемо починати роботу.

4.2 Аналіз роботи створеного рішення на прикладі реального вебдодатку

Тож робота системи починається із завантаження початкової сторінки та виконання її статичного аналізу. Окрім засобів рушія axe-core, система виконує, зокрема, перевірку фокус-індикаторів інтерактивних елементів. Як результат – зі 194 інтерактивних елементів на сторінці маємо 67, що не мають індикатора фокусу, а також 6 невидимих елементів. Після цього переходимо до динамічного аналізу першого інтерактивного елементу. Якщо не враховувати посилання і кнопки, взаємодія з якими не призводить до змін у контенті сторінки, першим таким елементом є випадаюча панель «Знайти все в одного продавця» (Рисунок 4.1).

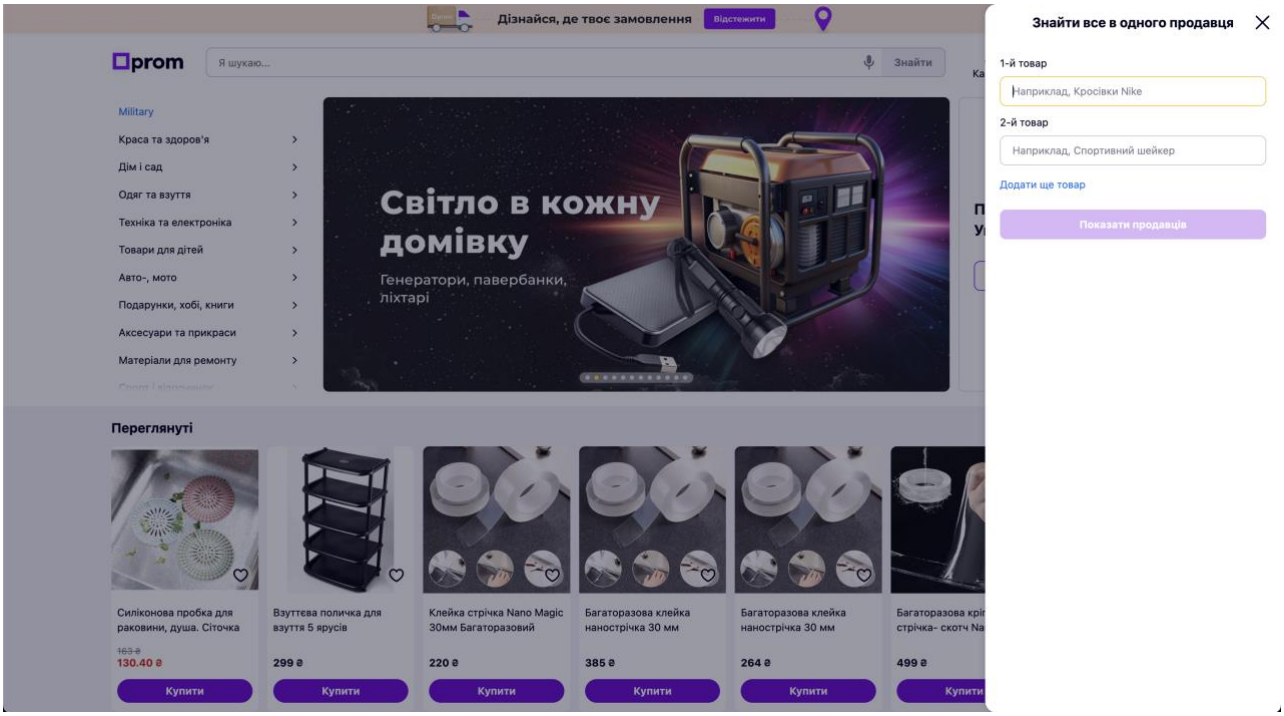


Рисунок 4.1 – Приклад зміни у контенті сторінки після взаємодії з інтерактивним елементом

Система визначила, що на екрані з’явилось випадаюча панель, після чого перевірила її і елемент, що її активував, на співпадіння з набором правил доступності для випадаючих панелей. У результаті маємо 2 відсутні атрибути на елементі-тригері і 5 порушених правил доступності у самій панелі, серед яких – відсутність «пастки для фокусу», що робить можливим перехід фокусу на невидимий контент. Далі виконується статичний аналіз вмісту випадаючої панелі, який виявив один невидимий інтерактивний елемент. Оскільки програма поки не може взаємодіяти з вебформами, перевіряємо, чи працює вихід з випадаючої панелі за допомогою клавіші Escape, а також чи стає попередній елемент активним. Друга умова не виконується, тож вручну робимо елемент активним і продовжуємо аналіз. Якщо після взаємодії з елементом змін у контенті сторінки виявлено не було – записуємо у результат попередження та продовжуємо аналіз. У випадку ж переходу на сторінку, яка не задовольняє налаштованому фільтру – переходимо на попередню сторінку.

Оскільки основна взаємодія у тестованому застосунку відбувається через випадаючі панелі, аби уникнути повторення, розглянемо елемент ще одного типу – випадаюче меню (Рисунок 4.2). Система також коректно визначила тип цього елементу і перевірила його на співпадіння з групою критеріїв для випадаючих меню. У результаті маємо наступні порушення:

- У кнопки-тригера відсутні атрибути aria-expanded, aria-haspopup і aria-owns/aria-controls;
- У контейнера меню відсутній атрибут role=”menu”, а також унікальний ідентифікатор;
- Меню не може бути відкритим клавішами Вверх/Вниз;

Натомість, навігація за посиланнями у меню працює коректно за допомогою клавіші Tab, а активація Escape призводить до виходу з меню і активації кнопки-тригера. Також меню не має реалізованого механізму «пастки для фокусу», втім, він і не є потрібним у даному випадку, адже елемент меню знаходиться в DOM-

дереві сторінки одразу після елементу, що його відкриває, таким чином зберігаючи доступний порядок інтерактивних елементів.

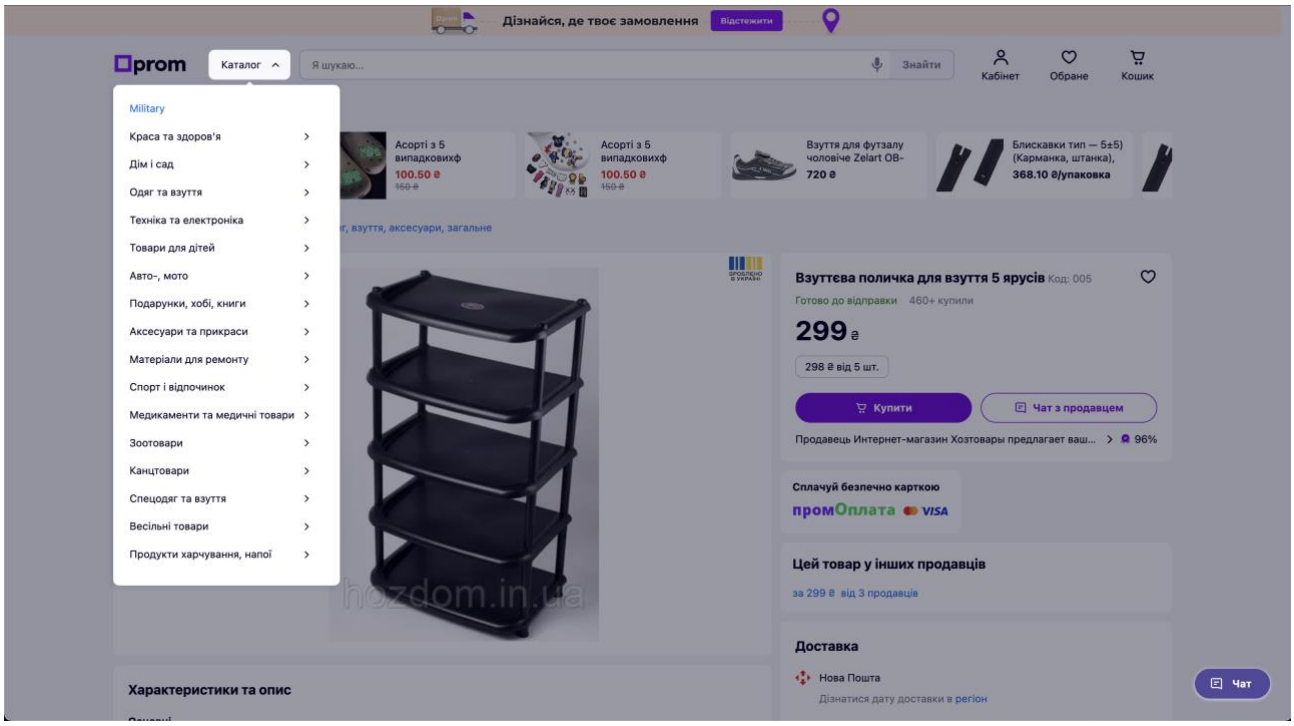


Рисунок 4.2 – Приклад появи випадаючого меню у контенті тестованої сторінки

Тож у підсумку тестування трьох вищеперерахованих сторінок маркетплейсу Prom.ua вдалося зібрати більше 300 проблем доступності без урахування знайдених рушієм axe-core. Більшість з них – відсутність фокус-індикатора у інтерактивних елементів. Втім, важливішим є те, що ми змогли як протестувати контент, недосяжний для звичайних засобів статичного тестування, як-от axe-core і подібних, а також те, що була протестована сама взаємодія з інтерактивними елементами. У якості результату система надає файл у форматі JSON з усіма знайденими помилками доступності і детальним описом кожної з них. І якщо говорити про систему CI/CD, то наданий файл може прямо порівнюватись зі створеним заздалегідь базовим результатом (англ. baseline).


```
[
  {
    "targetIndex": 2,
    "target": "<input type=\"search\" class=\"Dm7pv\" name=\"search_term\" placeholder=\"Я шукаю...\" autocomplete=\"off\" autocorrect=\"off\" autocapitalize=\"off\" value=\"\" style=\"\">",
```

```
        "message": "Interactive element doesn't have a focus ring - no-focus-ring",
        "url": "https://prom.ua/",
        "type": "error",
        "runner": "sentinelaria"
    },
    {
        "targetIndex": 3,
        "target": "<a href=\"https://prom.ua/ua/sc/dlya-mam-malyukiv\" target=\"_self\" data-
qaid=\"banner_link\" class=\"_0cNvO\"><img
srcset=\"https://images.prom.ua/5566867844_w325_h325_1268x600_4.png?fresh=1 325w,
https://images.prom.ua/5566867844_w460_h460_1268x600_4.png?fresh=1 460w,
https://images.prom.ua/5566867844_w748_h748_1268x600_4.png?fresh=1 748w,
https://images.prom.ua/5566867844_w588_h588_1268x600_4.png?fresh=1 588w,
https://images.prom.ua/5566867844_w850_h850_1268x600_4.png?fresh=1 850w,
https://images.prom.ua/5566867844_w939_h939_1268x600_4.png?fresh=1 939w,
https://images.prom.ua/5566867844_w650_h650_1268x600_4.png?fresh=1 650w,
https://images.prom.ua/5566867844_w920_h920_1268x600_4.png?fresh=1 920w,
https://images.prom.ua/5566867844_w1496_h1496_1268x600_4.png?fresh=1 1496w,
https://images.prom.ua/5566867844_w1176_h1176_1268x600_4.png?fresh=1 1176w,
https://images.prom.ua/5566867844_w1700_h1700_1268x600_4.png?fresh=1 1700w,
https://images.prom.ua/5566867844_w1878_h1878_1268x600_4.png?fresh=1 1878w\" sizes=\"(max-
width: 360px) 325px, (max-width: 480px) 460px, (max-width: 768px) 748px, (max-width: 959px) 939px,
(max-width: 1024px) 588px, 850px\"
src=\"https://images.prom.ua/5566867844_w2048_h2048_1268x600_4.png?fresh=1\" width=\"100%\"
height=\"100%\" alt=\"Для мам та малюків\" loading=\"lazy\" data-qaid=\"banner_img\"
fetchpriority=\"high\" class=\"gCAIM Lq0sj\"></a>",
        "message": "Interactive element doesn't have a focus ring - element-invisible",
        "url": "https://prom.ua/",
        "type": "error",
        "runner": "sentinelaria"
    }
]
```


Приклад 4.1 – Фрагмент результату роботи реалізованої системи

Висновки до розділу 4

Четвертий розділ розглядає роботу підсистеми для динамічного тестування вебдоступності на прикладі сайту Prom.ua – відомого українського інтернет-маркетплейсу, що вже є частково доступним і містить інтерактивні елементи різних типів. Крім того, наведено критерії, які мають виконуватись для найбільш ефективної і точної роботи реалізованого рішення.

Результати аналізу роботи системи на прикладі наведеного вебдодатку показали, що вона успішно ідентифікує різноманітні проблеми доступності, зокрема ті, які не виявляються засобами статичного тестування, такими як axe-core. Виявлені проблеми доступності зберігаються у форматі JSON для подальшого аналізу та порівняння з базовим результатом, що є типовим підходом тестування у системах CI/CD.

ВИСНОВКИ

У роботі було наведено класифікацію проблем вебдоступності, інструменти для їх виявлення, а також розглянуто підходи для автоматизованого тестування доступності у сучасних вебдодатках. Зокрема, виявлено, що наведені рішення здатні виконувати лише статичний аналіз доступності у автоматизованому режимі, що призводить до виявлення лише 57% всіх існуючих проблем вебдоступності. На противагу цьому, розглянуто дослідження The MITRE Corporation, що запропонувало динамічний підхід у тестуванні доступності шляхом побудови дерева станів можливих взаємодій з інтерактивними елементами і його порівняння з піддеревом, яке включає лише дії, доступні користувачам з обмеженими можливостями.

Запропоновано власний метод автоматизованого тестування вебдоступності, який, окрім реалізації статичного аналізу, пропонує новий підхід у тестуванні динамічного контенту шляхом взаємодії з інтерактивними елементами, виявлення активного контенту сторінки, визначення його типу і перевірки на виконання відповідної групи критеріїв доступності, що дозволяє як автоматично виконувати статичний аналіз недосяжного раніше контенту, так і перевіряти доступність пов’язаних динамічних елементів. Також реалізовано прототип підсистеми для динамічного тестування доступності вебдодатків у системі CI/CD, що виконує вебсканування інтерактивних елементів і будує дерево станів вебдодатку, перевіряючи кожен елемент з використанням зазначеного методу.

Реалізована система має значний потенціал до розвитку. Автоматичне визначення більшої кількості різноманітних типів контенту, більш точне виявлення змін у контенті після виконання дії, покращена фільтрація ігнорованого контенту – лише мала частка того, що може бути вдосконалено. До того ж, у той час як статичне тестування вже охоплює більшість поширених проблем доступності, динамічне тестування перебуває на початковому етапі розвитку і очікує свого розквіту з появою штучного інтелекту і великих мовних

моделей (англ. Large Language Models, LLM), які здатні, до прикладу, точно визначати тип активного контенту сторінки, чим можуть доповнити базові алгоритми, що засновуються на аналізі стилів. До того ж, ШІ здатний автоматично генерувати дані, необхідні для тестування доступності вебформ, що є окремою важливою частиною взаємодії користувачів з вебзастосунками, яку наразі неможливо перевірити без допомоги користувача.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. «2022 Was Another Record Year for Website Accessibility Lawsuits». AudioEye, 6 бер. 2023. Режим доступу: www.audioeye.com/post/2022-was-another-record-year-for-website-accessibility/
2. Michelle Yin, та ін. «A Hidden Market: The Purchasing Power of Working-Age Adults With Disabilities». American Institutes for Research, квітень 2018. Режим доступу: https://www.air.org/sites/default/files/2022-03/Hidden-Market-Spending-Power-of-People-with-Disabilities-April-2018.pdf
3. «Disability Impacts All of Us». Centers for Disease Control and Prevention (CDC), 15 квіт. 2023. Режим доступу: https://www.cdc.gov/ncbddd/disabilityandhealth/infographic-disability-impacts-all.html
4. W3C Web Content Accessibility Guidelines. Режим доступу: https://www.w3.org/WAI/standards-guidelines/wcag/
5. Color Contrast Checker (CCA). Режим доступу: https://www.tpgi.com/color-contrast-checker/
6. Tanaguru Contrast-Finder. Режим доступу: https://contrast-finder.tanaguru.com/
7. Button Contrast Checker. Режим доступу: https://www.aditus.io/button-contrast-checker/
8. Рішення для оцінки доступності Wave. Режим доступу: https://wave.webaim.org/
9. Браузерне розширення Lighthouse. Режим доступу: https://chromewebstore.google.com/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk
10. Accessibility Insights. Режим доступу: https://accessibilityinsights.io/
11. Комплекс рішень для тестування доступності Axe. Режим доступу: https://www.deque.com/axe/

12. Рушій для автоматизації тестування вебдоступності axe-core. Режим доступу: https://github.com/dequelabs/axe-core
13. «Automated Testing Identifies 57 Percent of Digital Accessibility Issues». Deque Systems, 10 бер. 2021. Режим доступу: https://www.deque.com/blog/automated-testing-study-identifies-57-percent-of-digital-accessibility-issues/
14. Комплекс рішень для тестування доступності Pa11y. Режим доступу: https://pa11y.org/
15. Trevor Bostic, та ін. «Automated Evaluation of Web Site AccessibilityUsing A Dynamic Accessibility Measurement Crawler». The MITRE Corporation, 27 жов. 2021. Режим доступу: https://www.researchgate.net/publication/355698816_Automated_Evaluation_of_Web_Site_Accessibility_Using_A_Dynamic_Accessibility_Measurement_Crawler
16. Selenium WebDriver. Режим доступу: https://www.selenium.dev/documentation/webdriver/
17. WebdriverIO. Режим доступу: https://webdriver.io/
18. Бібліотека для автоматизації E2E-тестування Cypress. Режим доступу: https://www.cypress.io/
19. Бібліотека Puppeteer для взаємодії з браузерами на базі Chromium. Режим доступу: https://pptr.dev/
20. Playwright від Microsoft. Режим доступу: https://playwright.dev/
21. Бібліотека tabbable для роботи з інтерактивними елементами. Режим доступу: https://github.com/focus-trap/tabbable
22. Мережа доставки контенту для бібліотек з відкритим вихідним кодом JSDelivr. Режим доступу: https://www.jsdelivr.com/
23. Бібліотека odiff для аналізу та порівняння зображень. Режим доступу: https://github.com/dmtrKovalenko/odiff
24. Бібліотека diffDOM для порівняння та відтворення змін у DOM-дереві. Режим доступу: https://github.com/fiduswriter/diffDOM

25. Рушій для побудови унікальних CSS-селекторів Simmer JS. Режим доступу: https://github.com/gmmorris/simmerjs
26. Комплекс рішень для захисту від DDoS та інших видів атак Cloudflare. Режим доступу: https://www.cloudflare.com/ddos/
27. Український маркетплейс Prom.ua. Режим доступу: https://prom.ua/ua/