УДК 004.896 **doi: 10.26906/SUNZ.2025.1.111-115**

С. Ю. Леонов, Д. А. Тиртишний

Національний технічний університет “Харківський політехнічний інститут”, Харків, Україна

РОЗРОБКА ПРОГРАМНОЇ ПЛАТФОРМИ ДЛЯ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ КЛІЄНТСЬКОЇ ЧАСТИНИ ВЕБЗАСТОСУНКУ

Анотація. У даній статті були вивчені методи та використання оригінально розробленого фреймворку для тестування продуктивності клієнтської частини вебзастосунків. Було проаналізовано завдання та виклики, які стоять перед розробниками при оптимізації взаємодії користувачів та рендерингу сторінок. Проведено моделювання взаємодії користувачів з вебзастосунками та створено фреймворк, що включає інструменти для створення сценаріїв користувацької взаємодії, моніторингу швидкодії браузера, аналізу часу завантаження сторінок, та керування тестовим процесом. Експерименти проведено на реальних вебзастосунках, що використовувалися як тестове середовище. Результати показують, що застосування пропонованого фреймворку призводить до оптимізації продуктивності клієнтської частини вебзастосунків, що значно поліпшує їхню швидкодію та надійність. В ході досліджень були визначені та підтверджені ознаки подальшого покращення взаємодії користувачів з вебзастосунками.

Ключові слова: комп’ютерна система; програмна платформа; програмне забезпечення; тестування продуктивності; клієнтська частина; вебзастосунок; фреймворк; оптимізація; експеримент; розробка.

Вступ

Тестування продуктивності клієнтської частини вебзастосунків є критично важливим аспектом забезпечення якості, який включає в себе використання автоматизованих тестів, що імітують взаємодію реальних користувачів з інтерфейсом застосунку, таким як браузер. Цей вид тестування є невід'ємною частиною процесу підвищення якості кінцевого продукту, оскільки будь-які збої або проблеми, пов’язані з низькою продуктивністю, можуть призвести до втрати клієнтів, які відмовляться від використання вашого вебзастосунку.

Уявімо ситуацію: під час великого онлайн-розпродажу, такого як Чорна п’ятниця, інтернет-магазин очікує на значне збільшення трафіку. І раптом, через проблеми з продуктивністю клієнтської частини, сайт починає зависати, сторінки завантажуються повільно, товари додаються у корзину з затримками, оплата не проводиться з першого разу - користувачі починають залишати сайт та переходити до конкурентів. Це може призвести до величезних фінансових втрат для компанії, як, наприклад, Amazon чи eBay, якщо їхні вебзастосунки не витримають навантаження у такий критичний момент.

Тестування продуктивності клієнтської частини дозволяє переконатися, що вебдодаток ефективно працює під час пікових навантажень і забезпечує користувачам безперервне та плавне керування інтерфейсом. Це допомагає виявити та оптимізувати елементи інтерфейсу, які можуть сповільнювати роботу або створювати незручності для користувачів.

Мета статті: виконати дослідження та розробити фреймворк для тестування продуктивності клієнтської частини вебзастосунків.

Основна частина

Тестування продуктивності клієнтської частини вебзастосунків є критично важливим для забезпечення оптимального користувацького досвіду. Продуктивність клієнтської частини відображає швидкість застосунку з точки зору кінцевого користувача, включаючи доставку контенту, його трансформацію та рендеринг у браузері або нативному застосунку. Важливо розуміти, що це відрізняється від часу обробки сервером, і не показує рівень навантаження, який сервер може витримати.

Для оцінки продуктивності клієнтської частини можна використовувати різні метрики, такі як час відповіді, час повного завантаження сторінки, розмір сторінки, кількість HTTP-запитів, розмір зображень, час виконання Javascript та інші. Ці метрики допомагають зрозуміти, як додаток веде себе з точки зору користувача, що є ключовим для оптимізації продуктивності. Однією з основних причин, чому тестування продуктивності клієнтської частини є таким важливим, є те, що вимірювання на стороні сервера не можуть адекватно відображати поведінку з точки зору кінцевого користувача. Наприклад, час відповіді сервера не дорівнює часу повного завантаження сторінки на стороні клієнта. Різні місця розташування клієнтів, різні конфігурації машин та використання сторонніх сервісів можуть значно вплинути на продуктивність.

Типи тестування продуктивності клієнтської частини включають мережеве тестування, тестування з різних місць, тестування користувацького досвіду, крос-браузерне тестування та мобільне тестування. Кожен з цих типів має свої особливості та допомагає виявити потенційні проблеми, які можуть вплинути на користувацький досвід. Наприклад, мережеве тестування шляхом емуляції різних пропускних здатностей та типів мереж може виявити специфічні вузькі місця для користувачів з мобільними або повільними з'єднаннями. Тестування з різних місць допомагає зрозуміти вплив мережевої швидкості та затримки, а також ефективність CDN. Тестування користувацького досвіду імітує поведінку кінцевих користувачів, дозволяючи оцінити, як додаток веде себе під час нормального та підвищеного навантаження. Крос-браузерне тестування важливе для забезпечення сумісності застосунку в різних браузерах, кожен з яких має свої особливості в обробці Javascript та рендерингу.

Загалом, тестування продуктивності клієнтської частини є невід'ємною частиною розробки вебзастосунків, оскільки воно допомагає забезпечити високий рівень задоволеності користувачів та ефективність вебзастосунку.

Для реалізації фреймворку тестування продуктивності клієнтської частини потрібно спочатку дослідити різні інструменти для тестування продуктивності, вивчити їх взаємодію та розробити архітектурну діаграму (рис. 1).

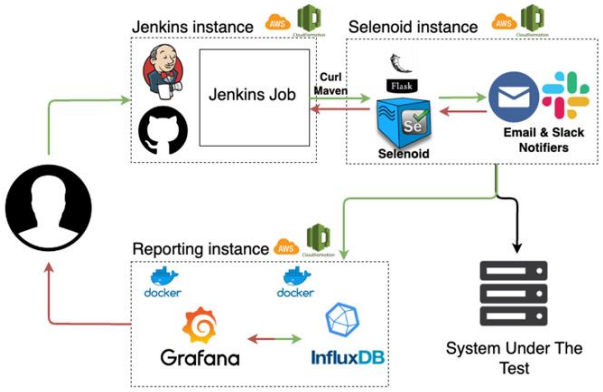

Рис. 1. Високорівнева архітектура фреймворку тестування продуктивності

Опираючись на архітектуру фреймворку тестування продуктивності клієнтської частини вебзастосунків, почнемо реалізацію поточного рішення. За основу візьмемо автотести написані на мові Java з використанням Selenium. Єдина відмінність тестів продуктивності від звичайних автотестів є те, що ми додаємо лише 1 стрічку коду для кожного методу(рис. 2) і вона виконує «магію», яка дозволяє автоматично дістати метрики продуктивності сторінки через Navigation Timing API та Resource Timing API, потім підрахувати усі важливі для візуалізації метрики, та надіслати їх до InfluxDB – в якій зберігаються всі метрики тестування продуктивності.

```
public LaptopsPage openFirstLaptop() {
    String firstLaptopInTheListName = laptopProductsList.get(0).getText();
    log.info( s: "Let's grab first laptop: " + firstLaptopInTheListName);
    laptopProductsList.get(0).click();
    wait.until(ExpectedConditions.elementToBeClickable(productDescriptionAndPricesButton));
    Assert.assertTrue(openedLaptopTitle.getText().contains(firstLaptopInTheListName),  s: "Wrong laptop was opened");
    waitUntilPageIsFullyLoaded(wait);
    perfNavigationTiming.writeToInflux( pageName: "ProductPage");
    log.info( s: "Asserted that the laptop we wanted to open and actually opened laptop are the same product");
    return this;
}
```

Рис. 2. Приклад методу з додаванням автоматичного підрахунку метрик продуктивності клієнтської частини вебзастосунку

Для того щоб досягти такого ефекту та легкості використання розробленого методу, був створений Java клас – NavigationTiming, в якому виконується декілька важливих дій:

- «Перехоплення» метрик продуктивності з браузеру, запущеного за допомогою «WebDriver».
- Підрахунок усіх важливих для візуалізації метрик.
- Надсилання метрик до InfluxDb.

Почнемо з першого пункту - «Перехопленя» метрик продуктивності з браузеру, запущеного за допомогою «WebDriver» (рис. 3).

```
1 usage
private final String javaScriptForPerformance = "var timings = performance.timing || {};return timings;";

1 usage  A Dmytro Tymkovych
public Map<String, Object> getAllTiming() {
    JavascriptExecutor jsrunner = (JavascriptExecutor) getDriver();
    if ("InternetExplorer".equalsIgnoreCase(BROWSER_NAME)) {
        Map<String, Object> ieTimings = (Map<String, Object>) jsrunner.executeScript(javaScriptForPerformanceInternetExplorer);
        timings = parseNavigationTimingDataFromIe(ieTimings);
    } else {
        timings = (Map<String, Object>) jsrunner.executeScript(javaScriptForPerformance);
    }
    return timings;
}
```

Рис. 3. Перехоплення метрик продуктивності з браузеру

За допомогою кількох команд на мові JavaScript робимо запит до браузера на повернення NavigationTiming об’єкту (рис. 4) для поточної сторінки, який потім приводимо до Java об’єкту Map<String, Object>, де String – це назва події, а Object – це timestamp. Зазвичай timestamp відображається у вигляді числа, що представляє кількість мілісекунд, які пройшли від певної епохи часу. Наприклад, у багатьох мовах програмування або базах даних використовується Unix timestamp, який показує кількість секунд, які пройшли від 1 січня 1970 року до певного моменту. Використання timestamp дозволяє легко порівнювати події, встановлювати порядок подій і виконувати інші операції, пов'язані з часом, у різних програмних системах.

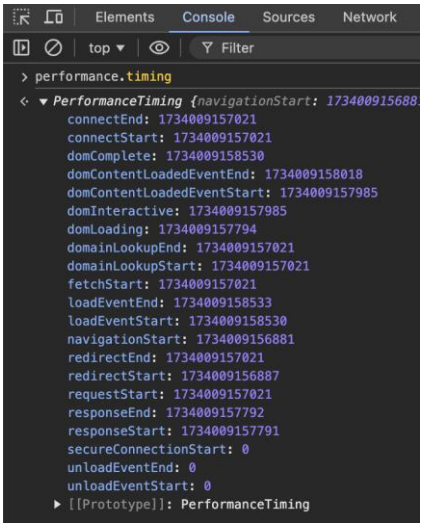

Рис. 4. Приклад Navigation Timing об’єкту

Наступним етапом є «Підрахунок усіх важливих для візуалізації метрик». Для цього підійдуть стандартні методи мови програмування Java. Достатньо перетворити отриманий timestamp у формат long, та зробити елементарні операції віднімання для того щоб отримати потрібні для візуалізації метрики. (рис. 5). Наприклад: для отримання важливої метрики «timeToInteract» достатньо відняти час події з назвою «domInteractive» від «domLoading», а для того щоб отримати повний час завантаження сторінки достатньо відняти час останньої події об’єкту Navigation timing для поточної сторінки, від першої події. А саме «Load Event End» - «Navigation Start». Останнім етапом є відправка підрахованих метрик до бази даних, в якій зберігаються результати тестування продуктивності. Для цього була використана бібліотека influxdb-java. Спочатку була піднята база даних InfluxDB, після цього я встановив зв’язок з цією базою.

```
235     //results
        2 usages  Dmytro Tyrtyshnyi
236 >   public long getLatency() { return getresponseStart() - getnavigationStart(); }
239
        1 usage  Dmytro Tyrtyshnyi
240     public long getBackend_response() {
241         return getresponseEnd() - getresponseStart();
242     }
243
        2 usages  Dmytro Tyrtyshnyi
244     public long getTimeToInteract() {
245         return getdomInteractive() - getdomLoading();
246     }
247
        2 usages  Dmytro Tyrtyshnyi
248 >   public long getTimeToLoad() { return getdomComplete() - getdomInteractive(); }
251
        2 usages  Dmytro Tyrtyshnyi
252 >   public long getOnload() { return getloadEventEnd() - getloadEventStart(); }
255
        2 usages  Dmytro Tyrtyshnyi
256 >   public long getTotal_time() { return getloadEventEnd() - getnavigationStart(); }
259
```

Рис. 5. Підрахунок важливих для візуалізації метрик.

Після цього потрібно сформувати об’єкт класу Point, в якому потрібно вказати measurement (таблицю для зберігання результатів), .time – час запису результатів в базу даних, .tag – теги для фільтрації результатів, .field – поля для підрахованих раніше метрик. Після цього кожен такий Point додається в батч, та відправляється асинхронно в базу даних. (рис. 6).

```
64      influxDB.setLogLevel(InfluxDB.LogLevel.BASIC);
65      Point point = Point.measurement("clientSide") // Формуємо об’єкт для запису в базу даних
66              .time(System.currentTimeMillis(), TimeUnit.MILLISECONDS)
67              .tag( tagName: "projectName", PROJECT_NAME)
68              .tag( tagName: "scenario", SCENARIO_NAME)
69              .tag( tagName: "env", ENV_NAME)
70              .tag( tagName: "browser", BROWSER_NAME)
71              .tag( tagName: "page", pageName)
72              .tag( tagName: "periodicity", PERIODICITY)
73              .tag( tagName: "periodicity_comment", VERSION)
74              .tag( tagName: "buildID", BUILD_ID)
75              .addField( field: "latency", this.getLatency())
76              .addField( field: "backend_response", this.getBackend_response())
77              .addField( field: "tti", this.getTimeToInteract())
78              .addField( field: "ttl", this.getTimeToLoad())
79              .addField( field: "onload", this.getOnload())
80              .addField( field: "total_time", this.getTotal_time())
81              .build();
82      batchPoints.point(point);
83      influxDB.write(batchPoints); // записуємо об’єкти в базу даних
```

Рис. 6. Запис результатів в базу даних

І найголовніше – усі ці операції займають лише декілька мілісекунд, тому на метрики та результати це ніяк не впливає.

Після розробки тестового сценарію, його можна запускати. Дуже важливою частиною розробленого інструменту є візуалізація отриманих результатів за допомогою Grafana. Для того щоб візуалізувати результати в Графані треба створити свій власний Dashboard (дошку), на яку можна додавати потрібні панелі для детальної візуалізації результатів, як в загальному на весь тестовий сценарій, так і на завантаження кожної сторінки, або виконання кожної дії окремо. Grafana dashboard - це інтерактивний інтерфейс, який використовується для візуалізації та моніторингу даних з різних джерел. Grafana - це популярний інструмент для створення і відображення дашбордів у реальному часі.

Під час побудови власної дошки, я керувався принципами того, що треба відобразити усі найважливіші метрики разом для побудови загального тренду швидкості роботи вебзастосунку, який тестується (рис. 7). Також, були враховані можливості Графани для відображення деталей та усіх важливих метрик по кожній сторінці або дії окремо. Що дозволяє знаходити проблемні ділянки в нашому застосунку і сфокусуватись на їх виправленні.

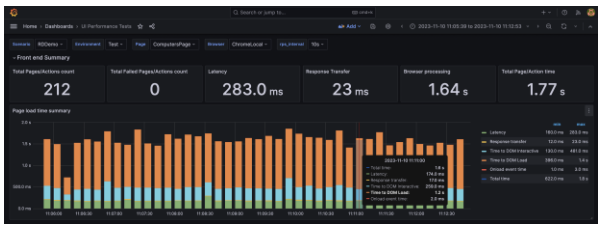

Рис. 7. Загальний тренд продуктивності по сценарію

Важливо додати, що за допомогою тегів, які ми додавали до наших записів метрик, тепер ми можемо фільтрувати відображення результатів на нашій дошці. Можна фільтрувати за різними сценаріями, тестовими середовищами, сторінками, результати для яких будуть відображатись. Також, можна фільтрувати, а також порівнювати результати між собою за різними тегам, як наприклад “browser”. Завдяки цьому тегу можна порівняти швидкість роботи одних і тих самих сторінок або дій в різних браузерах, що допоможе надавати однаково найкращу швидкодію різним користувачам. Для демонстрації роботи фільтрів, пропоную спочатку обрати одну сторінку для відображення метрик по ній (рис. 8), а потім ще кілька і порівняти результат відображення у Графані (рис. 9).

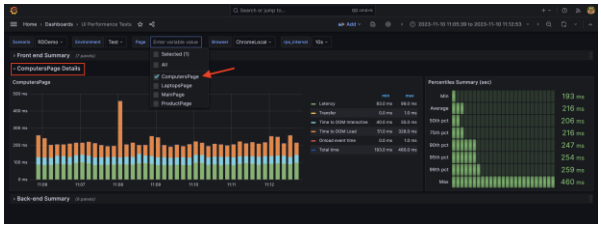

Рис. 8. Відображення панелей лише для 1 вибраної сторінки

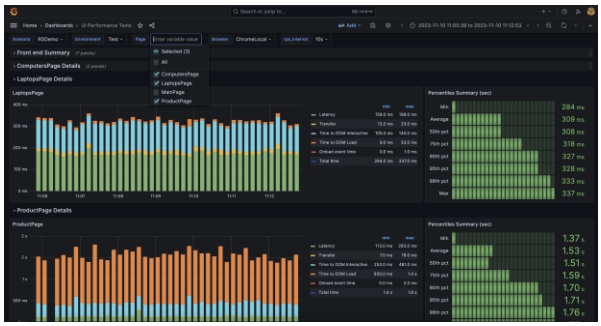

Рис. 9. Відображення панелей для кількох обраних сторінок

Як ви можете бачити на скріншоті вище, ProductPage сторінка працює приблизно в 5 разів повільніше, ніж LaptopsPage, це вже є ознакою того, що треба більш детально розглянути контент цієї сторінки та зрозуміти природу більш повільної роботи та відображенню.

Далі розглянемо більш детально кожну панель для відображення метрик по окремих сторінках. Для візуалізації метрик по окремій сторінці було створено 2 панелі, які автоматично реплікуються для кожної обраної сторінки. Це було досягнуто завдяки використанню InfluxQL – мова запитів до бази даних InfluxDB, та можливостей візуалізації в Grafana.

Перша панель відображає швидкість відкриття сторінки на часовому проміжку. Тобто на цьому графіку можна порівняти швидкість та стабільність відкриття сторінок на великому проміжку часу. Кожен стовпець з результатом відповідає за окреме відкриття сторінки та візуалізує різними кольорами швидкість виконання різних операцій з завантаження та відображення сторінки: Latency, Time to Dom Interactive, Time to DOM Load, Total Time (рис. 10).

Рис. 10. Приклад налаштування панелі зі стовпцями в Grafana

Друга панель відображає результати в різних персентилях, та була побудована по тому ж самому принципу, за виключенням різних налаштувань та типу візуалізації в Графані. Перша панель має тип «Time Series», коли друга панель – це «Bar Gauge», яка має трохи іншу візуалізацію. Головна задача другої панелі візуалізувати агреговані результати за обраний проміжок часу з цілю зрозуміти як швидко сторінка буде відображатись з різними рівнями вірогідності. Для цього в навантажувальному тестуванні використовуються персентилі. Персентиль (в контексті Графани та аналізу даних) - це статистичний термін, що вказує на значення в наборі даних, нижче якого певний відсоток спостережень. Наприклад, якщо ви маєте 90-й персентиль для швидкості завантаження вебсайту, це означає, що 90% користувачів мають швидкість завантаження, яка менше або рівна даному значенню, а лише 10% користувачів мають швидкість завантаження вище цього значення. У Графані, при аналізі метрик та візуалізації даних, використовуються персентилі для того, щоб отримати уявлення про розподіл даних і виявити екстремальні значення або аномалії. Це може бути корисно при моніторингу продуктивності, де важливо розуміти, які частини користувачів можуть відчувати найкращий чи найгірший досвід (рис. 11).

Рис. 11. Приклад налаштування панелі з персентилями в Grafana

Як бачимо, на скріншоті вище, персентилі рахуються завдяки можливостям InfluxQL, в якому є метод PERCENTILE, що підраховує персентилі для потрібних нам метрик на обраному проміжку часу. Фільтрування метрик за тегами та часом відбувається завдяки ключовому слову WHERE, де ми можемо вказати теги для фільтрації та часовий проміжок, наприклад: WHERE page =~ /^$page_name/ AND env =~ /$env/ AND scenario =~ /$scenario/ AND $timeFilter.

Також, завдяки інтеграції з системою неперервної інтеграції Jenkins в моєму інструменті тестування продуктивності, ми можемо запускати тести через дуже зручний інтерфейс, який дозволяє робити автоматичний запуск тестів за розкладом, наприклад вночі (Див. Додаток Б). Користувач має лише задати параметри запуску в зручному користувацькому інтерфейсі і натиснути кнопку Build, яка почне виконання тестів (рис. 12). Передавати можна такі параметри:

- назва тестового сценарію;
- браузер;
- тестове середовище (для розробки, для тестування);
- кількість ітерацій запуску тестового сценарію;
- чекбокс для автоматичного запису відео у разі виникнення помилок під час тесту;
- користувачі, які отримають автоматичну імейл нотифікацію після тесту.

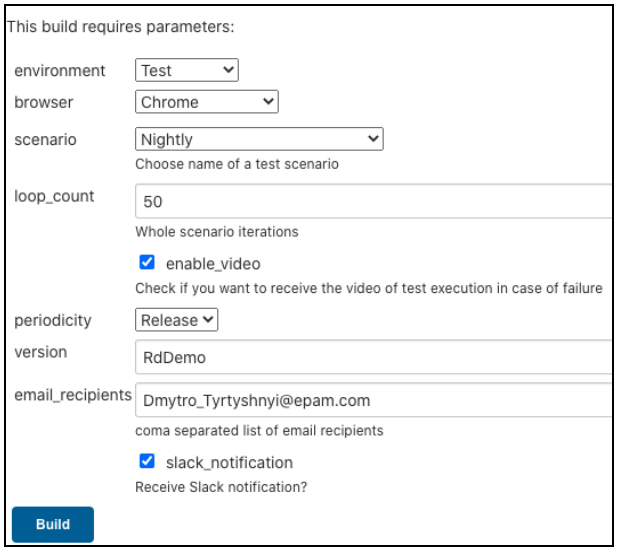

Рис. 12. Інтерфейс Jenkins

Після автоматичного запуску тестів, усі користувачі, імейли яких були вказані при запуску, отримають автоматичний звіт про тест, в якому можна буде побачити статус тесту, отримати відео для ітерацій в яких виникли помилки, та отримати посилання на відображення результатів в Графані, або відображення логів у Jenkins (рис. 13).

Обговорення результатів

В ході дослідження були виконані дослідження теоретичної частини продуктивності клієнтської частини вебзастосунків, включаючи огляд особливостей цього тестування, його переваги, та основі види.

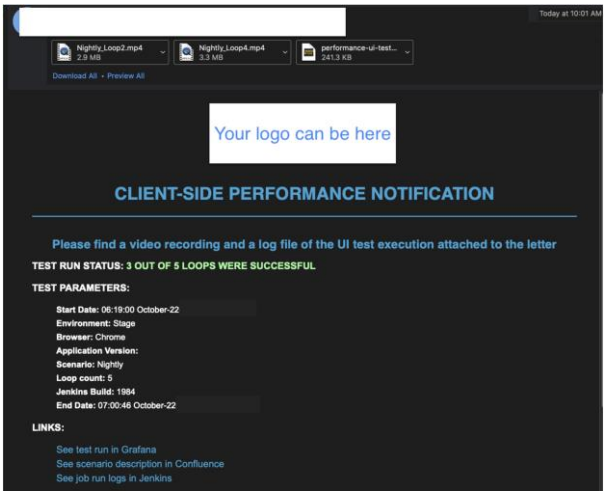

Рис. 13. Приклад автоматичного звіту на імейл

Після того були досліджені різні інструменти для розробки фреймворку тестування продуктивності клієнтської частини вебзастосунків. Усі ці інструменти були застосовані на практиці та продемонстровані у дії. Після розробки фреймворку, був розроблений тест для прикладу, та завдяки цьому тестовому сценарію був протестований інтернет-магазин.

Тести продуктивності клієнтської частини виявили найповільніші для завантаження сторінки та вказали на найпроблемніші місця, які потребують поглибленого аналізу з використанням більш потужних та деталізованих інструментів для аналізу та оптимізації проблем з продуктивністю клієнтської частини вебзастосунків.

Висновки

У результаті проведеного дослідження тестування продуктивності серверної частини вебзастосунків та розробки фреймворку потрібно зазначити, що впровадження цього фреймворку дозволяє підвищити ефективність роботи та стабільність клієнтської частини вебзастосунків.

Були детально досліджені особливості тестування продуктивності клієнтської частини вебзастосунків, продемонстровані варіанти рішень, які включають в себе набір спеціалізованих інструментів для досягнення мети дослідження.

Розроблений фреймворк включає інструменти для збіру метрик продуктивності з браузеру клієнта, моніторингу, аналізу даних, керування тестовим процесом, а також модулі для вдосконалення продуктивності. Він є зручним та гнучким інструментом для ефективного аналізу та вдосконалення продуктивності вебзастосунків.

Експерименти, проведені в рамках дослідження, підтвердили, що використання розробленого фреймворку допомагає в дослідженні з метою значного покращення продуктивності клієнтської частини вебзастосунків.

Отже, результати цього дослідження можуть бути використані розробниками вебзастосунків для підвищення продуктивності та надійності своїх продуктів.

СПИСОК ЛІТЕРАТУРИ

1. «Проблеми інформатики та моделювання (ПІМ-2023), Харків, 2023», матеріали міжнародної науково-технічної конференції, стор.69 «КЛЮЧОВІ АСПЕКТИ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ»
2. “Web performance warrior” book by Andy Still.
3. "MicroCAD-2024, 9.4 ІНФОРМАТИКА І МОДЕЛЮВАННЯ", с.1407, Леонов С.Ю, Тиртишний Д.А, Використання сучасних методів тестування та аналізу клієнтської частини вебзастосунків
4. «Проблеми інформатики та моделювання (ПІМ-2024), Харків, 2024», матеріали міжнародної науково-технічної конференції, стор.77 «Стратегії кешування контенту для оптимізації LCP у динамічних вебзастосунках»
5. Джеймс Бах та Джейсон Теслер: "Автоматизоване тестування ПЗ: Вступ до професійної практики".
6. Мартін Аббот: "Тестування продуктивності програмного забезпечення".
7. Джеймс Петер: "Тестування продуктивності та навантаження".
8. «Інформатика, управління та штучний інтелект», матеріали п’ятої міжнародної науково-технічної конференції, стор.65 «ТЕСТУВАННЯ ПРОДУКТИВНОСТІ WEB-САЙТУ ДЛЯ ЗРОСТАННЯ ЯКОСТІ ПРОДУКТУ»
9. WebDev [Електронний ресурс]: Understanding time to first byte metric : [https://web.dev/articles/ttfb]
10. WebDev [Електронний ресурс]: Understanding first contentful paint (FCP): [https://web.dev/articles/fcp]
11. WebDev [Електронний ресурс]: Understanding time to interactive metric : [https://web.dev/articles/tti]

Received (Надійшла) 08.11.2024
Accepted for publication (Прийнята до друку) 12.02.2025

Development of a software platform for testing the performance of the client part of a web application

Sergey Leonov, Dmytro Tyrtyshnyi

Abstract. This article explores methods and the use of an originally developed framework for testing the client-side performance of web applications. The tasks and challenges faced by developers in optimizing user interactions and page rendering were analyzed. Simulation of user interactions with web applications was conducted, and a framework was created that includes tools for creating user interaction scenarios, monitoring browser performance, analyzing page load times, and managing the testing process. Experiments were carried out on real web applications used as a testing environment. The results show that applying the proposed framework leads to optimizing the performance of the client-side of web applications, significantly improving their responsiveness and reliability. During the research, signs of further improvement in user interactions with web applications were identified and confirmed.

Keywords: computer system; software platform; software; performance testing; client-side; web applications; framework; optimization; experiment; development.