УДК 004.8:004.9 **doi: 10.26906/SUNZ.2025.1.83-86**

О. Ю. Заковоротний, О. А. Сапальський

Національний технічний університет "Харківський політехнічний інститут", Харків, Україна

ВИРІШЕННЯ ПРОБЛЕМ ПРОДУКТИВНОСТІ ЗА ДОПОМОГОЮ АСИНХРОННИХ МЕТОДІВ В JAVASCRIPT

Анотація. Сучасні веб-додатки часто стикаються з викликами продуктивності, пов'язаними з блокуванням інтерфейсу користувача та повільною обробкою даних. Оскільки JavaScript є однопотоковою мовою програмування, браузер не здатен одночасно виконувати обчислення та оновлювати інтерфейс. Це може спричиняти "лаги" — ситуації, коли додаток зависає, і користувач не може взаємодіяти з інтерфейсом. Асинхронне програмування в JavaScript надає інструменти для вирішення цих проблем, що дозволяє підвищити відгукливість додатків і ефективність виконання завдань. **Мета** цієї роботи полягає у дослідженні можливостей підвищення продуктивності веб-додатків за допомогою асинхронного програмування в JavaScript. Це включає аналіз ефективності обробки завдань та відгукливості інтерфейсу користувача, а також оцінку доцільності використання асинхронних підходів для вирішення проблем блокування інтерфейсу та запобігання "лагам", що часто виникають у сучасних однопотокових середовищах. **Отримані наступні результати:** використання асинхронного програмування в JavaScript значно підвищило продуктивність веб-додатків. Відгукливість інтерфейсу покращилася за рахунок зменшення кількості випадків блокування UI під час обробки важких завдань. У місцях, де раніше спостерігалися відчутні лаги, додаток тепер працює безперервно, забезпечуючи плавну взаємодію з користувачем. **Висновки.** Було доведено доцільність використання асинхронного програмування в JavaScript для підвищення продуктивності веб-додатків. Це дозволяє значно покращити відгукливість інтерфейсу та уникнути блокування під час виконання складних операцій, забезпечуючи плавний досвід взаємодії для користувача. Визначено перспективи подальших досліджень у напрямку оптимізації асинхронних процесів для різних типів завдань та інтеграції з багатопотоковими рішеннями.

Ключові слова: асинхронне програмування, JavaScript, продуктивність веб-додатків.

Вступ

В останні роки веб-додатки стали невід'ємною частиною нашого повсякденного життя, забезпечуючи швидкий доступ до інформації та інструментів для вирішення різних завдань. Однак зі зростанням складності сучасних додатків з'являються нові виклики, пов'язані з їх продуктивністю. Однією з основних проблем є блокування інтерфейсу користувача під час виконання ресурсоємних операцій, що значно погіршує користувацький досвід. Це особливо помітно в однопотоковому середовищі, яким є JavaScript, коли браузер не може одночасно виконувати обчислення і оновлювати інтерфейс. Такі затримки в роботі програми можуть викликати "лаги" — моменти, коли додаток тимчасово не реагує на дії користувача.

Асинхронне програмування пропонує ефективні рішення для вирішення цієї проблеми. Воно дозволяє розділити завдання на менші, не блокуючи основний потік, що сприяє збереженню відгукливості інтерфейсу та загальному підвищенню продуктивності веб-додатків. Незважаючи на очевидні переваги, використання асинхронних підходів вимагає уважного планування, оскільки помилки в їх реалізації можуть призвести до непередбачуваних результатів.

Ця робота присвячена дослідженню можливостей асинхронного програмування в JavaScript для підвищення продуктивності веб-додатків і покращення взаємодії користувача з інтерфейсом.

Постановка проблеми. JavaScript, як однопотокова мова програмування, має суттєві обмеження, що впливають на продуктивність веб-додатків. Однією з основних проблем є те, що мова не може виконувати дві задачі одночасно. Це означає, що під час виконання ресурсомістких обчислень браузер не здатен оновлювати інтерфейс.

В результаті, інтерфейс працює плавно лише при виконанні задач тривалістю до ~50 мс [1] в іншому випадку виникають переривання, а при дуже тривалих операціях можливі зависання.

Варто пам'ятати, що 0,1 секунди це максимальний час, протягом якого користувачі відчувають, що вони безпосередньо взаємодіють з об'єктами в інтерфейсі. 1 секунда: — максимальний час, протягом якого користувачі можуть вільно переміщатися в командному просторі, не відчуваючи непомірного очікування.

Затримка від 0,2 до 1,0 секунди означає, що користувачі помічають цю затримку і, отже, відчувають, що комп'ютер "працює" над командою, а не що команда є безпосереднім наслідком їх дій. Велика затримка додає суттєвого ризику, що користувач може втратити увагу і, як наслідок, інтерес [2].

Таким чином, існує нагальна потреба у вирішенні цих проблем, щоб забезпечити безперервну і плавну взаємодію з веб-додатками, що є критично важливим для підтримки інтересу користувачів та забезпечення високої якості користувацького досвіду.

У зв'язку з цим, **метою дослідження** є вивчення того, як асинхронні методи в JavaScript можуть покращити продуктивність веб-додатків на практиці, та визначити підходи що дозволяють зменшити блокування інтерфейсу та підвищити відгукливість, забезпечуючи тим самим кращий користувацький досвід.

Основна частина роботи

У цьому розділі розглянемо, як асинхронне програмування може покращити продуктивність веб-додатку на прикладі системи розумного будинку. Зокрема, ми будемо розглядати проблемами, які виникають при використанні синхронного XMLHttpRequest для отримання даних.

Припустимо, у нас є система умного дому, яка періодично отримує дані з датчиків, таких як температура, вологість та стан системи безпеки.

Використовуючи синхронний XMLHttpRequest, ми можемо реалізувати запит таким чином, наведено на рис. 1.

```
const request = new XMLHttpRequest();

// синхронний запит
request.open("GET", "https://api.smart-home.com/temperature", false);
request.send();

if (request.status === 200) {
  // Обробка отриманих даних
  processTemperatureData(request.responseText);
}

```

Рис. 1. Синхронний XHR запит

Використання синхронного запиту означає, що браузер буде заблокований на час виконання запиту, що призводить до заморожування інтерфейсу. Це викликає "лаги" у користувацькому інтерфейсі, і користувач не зможе взаємодіяти з додатком, що може негативно вплинути на його досвід. Таким чином, користувач може відчувати розчарування та відмовитися від використання програми.

Натомість, асинхронні запити є кращим підходом, оскільки вони дозволяють браузеру продовжувати реагувати на дії користувача. Таким чином, незалежно від тривалості запиту інтерфейс залишатиметься інтерактивним. У однопоточному Javascript це досягається за рахунок того, що всі запити обробляються не самим двигуном мови, а безпосередньо браузером, що додатково дозволяє робити кілька запитів одночасно. Інструменти на кшталт "xhr" або "fetch" лише дають інструкцію браузеру запланувати запити та колбеки, які мають бути виконані після їх завершення. У потрібний момент браузер відправити ці колбеки в Event loop, де вони будуть оброблені та виконані в порядку черги після звільнення основного стека виклик (рис. 2).

```
const request = new XMLHttpRequest();
// true - асинхронний запит
request.open("GET", "https://api.smart-home.com/temperature", true);
// Обробка отриманих даних
request.onload = () => {
  if (request.status === 200) {
    processTemperatureData(request.responseText);
  }
};
// Обробка помилки
request.onerror = (error) => handleError(error);

request.send();
```

Рис. 2. Асинхронний XHR запит

Ідеальним підходом для обмеження асинхронної логіки буде використання синтаксису async/await. Підхід не збільшить продуктивність, але зробить код більш читаним і організованим (рис. 3).

```
async function getEnvironmentData() {
    try {
        const temperature = await fetch('https://api.smart-home.com/temperature');
        const humidity = await fetch('https://api.smart-home.com/humidity');
        // Обробка отриманих даних
        processTemperatureData(temperature);
        processHumidityData(humidity);
    } catch (error) {
        // Обробка помилки
        handleError(error);
    }
}
```

Рис. 3. Запит з використанням синтаксису async/await

Таким чином покращується робота програми. Поки запит, користувач може виконувати у додатку інші завдання, а браузер може відобразити тимчасовий компонент заглушку. Це значно покращує користувальницький досвід. Інша поширена причина проблем з продуктивністю – це довгі завдання. Як вже було сказано раніше, завдання повинні бути не більше 50 мс, щоб не викликати проблем з продуктивністю. В іншому випадку користувач зможе помітити лаги. Розв'язанням цього завдання виглядає очевидним: необхідно розбити одну велику задачу на безліч менших (рис. 4).

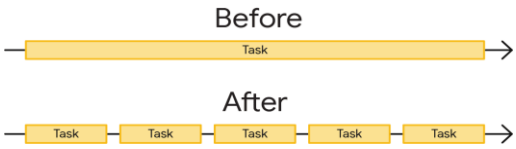

Рис. 4. Розбиття задачі

Але проблема полягає в тому, що безліч синхронних завдань, як і раніше, блокуватимуть браузер, а саме процес реакції на користувацькі події та малювання інтерфейсу [3]. Цю проблему можна вирішити за допомогою функцій відкладеного виклику. Якщо дати браузеру час або вікно для виконання завдань взаємодії з інтерфейсом, то можна виключити появу лагів і зависань. Найпростіший випадок це використання setTimeout. Цей метод дозволяється запланувати виклик функції не раніше вказаного часу. У контексті розробки програми "розумного дому" його можна використовувати, щоб запланувати перевірку трекінгових систем раз на п'ять хвилин. Це дозволяє різко скоротити кількість викликів (рис. 5).

Іноді немає можливості вказати часовий проміжок для початку виконання функції. У цьому випадку може статись в нагоді requestAnimationFrame. requestAnimationFrame вказує браузеру на те, що ви хочете зробити анімацію, і просить його запланувати перемальовку на наступному кадрі анімації [4]. Як параметр метод отримує функцію, яка буде викликана перед перемальовуванням.

```
async function checkEnvironment() {
  // Отримання даних про навколишнє середовище
  await getEnvironmentData();
  // Виклик функції через 5 хвилин
  setTimeout(checkEnvironment, 300000);
}

// Запуск перевірки
checkEnvironment();
```

Рис. 5. Виклик через setTimeout

Він синхронізується з частотою оновлення екрана (зазвичай 60 кадрів на секунду). Замість запуску анімації з фіксованим інтервалом (як при використанні setInterval), браузер сам вирішує, коли краще виконати наступний кадр, пропускаючи непотрібні оновлення та тим самим знижуючи навантаження на процесор. Так, наприклад, додаток smart home може використовувати його для малювання графіків.

Подібний метод, requestIdleCallback, використовується для виконання фонових завдань, які не критичні для продуктивності і можуть бути відкладені доти, доки браузер не стане "вільним" від інших пріоритетних операцій [5]. Так реальна програма може використовувати цей метод для роботи з трекінгом активності користувача та збору статистики.

Бувають випадки, коли необхідно виконати якісь складні обчислення, які не можна розділити на менші частини. У такому разі може допомогти інший підхід - використання WebWorkers (рис. 6).

```
// main.js
const worker = new Worker("worker.js");
// Обробка середньої температури
worker.onmessage = ({ data }) => handleAverageTemperature(data);
// Повідомлення для Worker
worker.postMessage("Обчислити середню температуру");
//=============================================================
// worker.js
onmessage = ({ data }) => {
  // Складні обчислення
  const averageTemperature = calculateAverageTemperature(data);
  // Відправка результату назад
  postMessage(averageTemperature);
};

// Обчислення для знаходження середньої температури
function calculateAverageTemperature() {
  for (let i = 0; i < 1_000_000; i++) {
    // Складна логіка
  }

  // Повертаємо обчислене значення
  return someCalculatedValue;
}
```

Рис. 6. Код WebWorker

Web Workers — це API для запуску JavaScript у фоновому потоці, окремо від основного потоку, який відповідає за взаємодію з DOM та обробку подій користувача.

Це дозволяє виконувати важкі обчислення або тривалі завдання без блокування інтерфейсу користувача.

Однак, Web Workers не мають доступу до DOM безпосередньо.

Це означає, що будь-які результати обчислень повинні передаватися назад до основного потоку через механізм обміну повідомленнями (postMessage()), що може створити невелику затримку для інтерактивних застосунків.

Так, наприклад, додаток може розраховувати різні показання датчиків без негативного впливу на продуктивність сторінки.

Висновки та перспективи подальших досліджень

На основі проведеного аналізу та використання асинхронних методів програмування в JavaScript для додатку "розумного дому" вдалося досягти значного підвищення продуктивності та покращення взаємодії з користувачем.

Перехід від синхронних запитів до асинхронних дозволив уникнути блокування інтерфейсу, що робить роботу програми плавною і безперервною. Використання `async/await` значно спрощує читабельність коду та підтримку складних запитів.

Методи відкладеного виконання, такі як

– `setTimeout`,

– `queueMicrotask`,

– `requestAnimationFrame`

– `requestIdleCallback`,

дозволяють ефективніше розподіляти навантаження між критичними і неважливими задачами, підвищуючи загальну продуктивність.

Також застосування Web Workers для фонових обчислень виводить важкі обчислення з головного потоку, що дозволяє уникнути "лагів" навіть при великій кількості даних.

Дослідження показує, що асинхронні техніки є критичними для створення відгукливих та продуктивних вебзастосунків.

Перспективи подальших досліджень можуть включати глибшу оптимізацію алгоритмів обробки даних та інтеграцію більш складних фонових завдань у Web Workers для ще більшої стабільності та ефективності роботи застосунків..

СПИСОК ЛІТЕРАТУРИ

1. Addy Osmani. A comprehensive iot attacks survey based on a building-blocked reference mode. URL: https://web.dev/articles/long-tasks-devtools?hl=en (дата звернення 20.10.2024).
2. Miller R. B. Response time in man-computer conversational transactions //Proceedings of the December 9-11, 1968, fall joint computer conference, part I. – 1968. – С. 271-277.
3. MDN. Window: requestAnimationFrame() method. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/requestAnimationFrame (дата звернення 20.10.2024).
1. 4 MDN. Window: requestIdleCallback() method. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallback (дата звернення 20.10.2024).
4. Jeremy Wagner. Optimize long tasks. URL: https://web.dev/articles/optimize-long-tasks?hl=en (дата звернення 20.10.2024).

Received (Надійшла) 30.11.2024
Accepted for publication (Прийнята до друку) 05.02.2025

Solving productivity problems using asynchronous methods in JavaScript

Oleksandr Zakovorotnyi, Oleskandr Sapalskyi

Abstract. The **subject of study in the article** are the performance issues in modern web applications caused by the single-threaded nature of JavaScript. Specifically, it investigates how blocking operations, like synchronous XMLHttpRequests (XHR), can negatively impact the responsiveness of user interfaces. The study explores how asynchronous programming can improve the user experience by preventing interface lag and optimizing task handling. **The task** is to explore and demonstrate how various asynchronous techniques, including callbacks, promises, async/await, and methods such as setTimeout, queueMicrotask, requestAnimationFrame, and requestIdleCallback, can be employed to address performance bottlenecks in a web-based smart home application. Additionally, the use of Web Workers to offload intensive calculations to background threads is examined. **The following results were obtained:** The study shows that transitioning from synchronous XHR to asynchronous methods significantly reduces the occurrence of interface freezing. Callbacks and promises provide basic improvements in responsiveness, while async/await makes the code more readable and easier to manage. Methods like setTimeout and requestAnimationFrame allow deferring non-critical tasks, improving performance during active user interaction. Web Workers were effectively used to handle complex computations, preventing the main thread from being blocked and allowing the application to remain responsive even during intensive processing. **Conclusions:** Asynchronous programming techniques are essential for optimizing the performance of JavaScript-based web applications. By adopting these methods, developers can prevent interface blocking and enhance the overall user experience. The use of Web Workers for background processing further improves performance, making these approaches highly applicable to complex applications like smart home systems. Future research could focus on refining these techniques for specific use cases and exploring additional strategies for performance optimization.

Keywords: asynchronous programming, JavaScript, web application performance.