МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРИЗОВАНИХ СИСТЕМ ЗАХИСТУ ІНФОРМАЦІЇ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
____________ С.В. Казмірчук

«_____»____________ 2020 р.

На правах рукопису
УДК 004.056.5:510.22(043.3)

МАГІСТЕРСЬКА АТЕСТАЦІЙНА РОБОТА

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Програмний модуль відслідковування помилок у високонавантажених веб додатках.

Автор: Д.С. Сташевський

Науковий керівник: доцент кафедри КСЗІ А.В. Ільєнко

Нормоконтролер: доцент кафедри КСЗІ А.В. Ільєнко

Київ 2020

ВСТУП

Актуальність. Враховуючи стрімкий розвиток інформаційних технологій та розробку високонавантажених веб-додатків, у зв’язку з великими об’ємами кодової бази постає потреба у відслідковуванні помилок у високонавантажених веб-додатках. Оскільки велика кількість веб-додатків написані на мові програмування Javascript, у цій мові програмування вже є деякі інструменти для відслідковування помилок. Проте, коли розміри веб-додатку починають збільшуватися, відслідковувати усі помилки та аналізувати їх дуже важко.

Слід пам’ятати, що помилки можуть бути різними, від звичайних помилок які порушують цілісність інтерфейсу, до вразливостей у системі безпеки веб-додатку. Тому можна сказати, що механізм для вчасного та якісного відслідковування помилок, на сьогоднішній день – це необхідна річ у будь-якому веб-додатку.

Відомі підходи до вирішення поставленої задачі. Проаналізувавши існуючі рішення для відслідковування помилок у високонавантажених веб-додатках, можна дійти до висновку, що на сьогоднішній день існує лише два рішення які більш-менш задовольняють потреби відслідковування помилок. Проте навіть, ці два рішення мають свої недоліки. Адже у дипломній роботі будуть сформовані певні критерії для програмного модулю відслідковування помилок у високонавантажених веб-додатках.

Отже було прийнято рішення створити власний програмний модулю відслідковування помилок у високонавантажених веб-додатках.

Метою дипломної роботи є програмна реалізація модуля відслідковування помилок у високонавантажених веб-додатках. Даний програмний модуль має бути захищений від стороннього втручання та мати можливість аналізувати місце де саме сталася помилка та логувати будь-які інші помилки на серверній та клієнтській частинах.

Виходячи з мети, **завданням** даної дипломнох роботи є:

– дослідити відомі підходи відслідковування помилок у високонавантажених веб-додатках та існуючі практичні рішення які можуть відслідковувати помилки у високонавантажених веб-додатках та формування вимог до програмного модулю;

– розробити алгоритм відслідковування помилок у високонавантажених веб-додатках;

– розробити програмну реалізацію модуля відслідковування помилок у високонавантажених веб-додатках з використанням мови програмування Javascript;

– провести тестування та доцільність використання розробленого програмного модулю;

Галузь застосування. Розроблений алгоритм та програмний модуль може бути використаний у будь-якому веб-додатку написаному на мові програмування Javascript.

Об'єкт дослідження: процес відслідковування помилок у високонавантажених веб-додатках.

Предмет дослідження: існуючі методи та засоби відслідковування помилок у високонавантажених веб-додатках.

Метод дослідження. Проведені дослідження базуються на технологіях відслідковування помилок у високонавантажених веб-додатках на основі мови Javascript.

Новизна одержаних результатів полягає в наступному:

Удосконалено модуль відслідковування помилок у високонавантажених веб додатках, за рахунок використання власного алгоритму логеру, що дозволило зменшити розмір програмного додатку для завантаження близько 2-5 кілобайт, визначати конкретне місце де сталася помилка при виконанні програмного коду та зберігати історію помилок.

Практичне значення роботи полягає у створенні удосконаленого модуля відслідковування помилок у високонавантажених веб додатках на

основі використання власного алгоритму логеру, що дозволяє визначати місце у додатку де саме сталася помилка, та відправлення цієї інформації у журнал подій з використанням мови програмування Javascript**.**

Апробація результатів. Робота була апробована на наступних конференціях:

• Сташевський Д. С. Аналіз сучасного методу передачі даних між клієнтською та сервероною частинами та критерії його безпеки // Materiály XVI Mezinárodní vědecko - praktická konference «Aplikované vědecké novinky», Volume 2 : Praha. Publishing House «Education and Science». – P. 25-29.

• Сташевський Д. С. Обробка помилок у мові програмування Javascript / Сташевський Д.С., Ільєнко А.В. // Materials of the XVI International scientific and practical Conference Scientific horizons - 2020, September 30 - October 7, 2020: Sheffield. Science and education LTD – P. 60-72.

Розділ 2. ТЕХНОЛОГІЇ ВІДСЛІДКОВУВАННЯ ПОМИЛОК У ВИСОКОНАВАНТАЖЕНИХ ВЕБ-ДОДАТКАХ НА ОСНОВІ МОВИ JAVASCRIPT

2.1. Мова програмування Javascript

За основну мову програмування було взято Javascript – для клієнтської та серверної частин. На сьогоднішній день Javascript – це найбільш вдале та доцільне рішення для побудови користувацьких інтерфейсів, навіть можна сказати ця мова є незамінною на ринку. Javascript має безліч можливостей та цілком задовольняє наші потреби у побудові та написанні логіки веб-інтерфейсів та серверної частини.

JavaScript спочатку був створений для того, щоб «оживити веб-сторінки». Програми написані цією мовою називаються скриптами. Їх можна записувати прямо в HTML веб-сторінки та запускати автоматично під час завантаження сторінки. Скрипти записуються та виконуються як звичайний текст. Їм не потрібна спеціальна підготовка або компіляція для запуску. У цьому аспекті JavaScript сильно відрізняється від іншої мови, яка називається Java[9].

JavaScript є мовою програмування, яка відповідає специфікації ECMAScript. JavaScript є високорівневою мовою програмування, яка компілюється построково, тобто інтерпритується. Вона має синтаксис фігурних дужок, динамічне введення тексту, орієнтацію на об’єкти на основі прототипу та функцій першого класу.

Поряд з HTML і CSS, JavaScript є однією з основних технологій Всесвітньої мережі Інтернет. JavaScript забезпечує інтерактивні веб-сторінки та є важливою частиною будь-яких веб-додатків. Переважна більшість веб-сайтів використовують його для написання логіки на стороні клієнта, і всі основні веб-браузери мають спеціальне ядро JavaScript для його виконання.

Як мультпарадигмована мова програмування, JavaScript підтримує керовані подіями, функціональні та імперативні стилі програмування. Вона має інтерфейси прикладного програмування (API) для роботи з текстом, датами, регулярними виразами, стандартними структурами даних та об’єктною моделлю документа (DOM). Однак сама мова не включає жодного вводу / виводу (вводу / виводу), наприклад мережевих, сховищних чи графічних засобів, оскільки хостове середовище (зазвичай веб-браузер) забезпечує ці API[10].

На сьогоднішній день JavaScript може виконуватися не лише у браузері, але і на сервері, або фактично на будь-якому пристрої, що має спеціальну програму, яка називається ядром JavaScript. У браузері є вбудований механізм, який іноді називають «віртуальною машиною JavaScript».

Різні реалізації ядра мають різні “кодові назви”. Наприклад: V8 – у Chrome та Opera.SpiderMonkey – у Firefox. Є й інші кодові назви, такі як «Chakra» для Internet Explorer, «ChakraCore» для Microsoft Edge, «Nitro» та «SquirrelFish» для Safari тощо.

Як працює ядро Javascript. Механізм (вбудований, якщо це браузер) аналізує скрипт.Потім він перетворює (“компілює”) скрипт на машинну мову.І тоді машинний код працює досить швидко.Ядро застосовує оптимізацію на кожному кроці процесу. Воно навіть спостерігає за скомпільованим скриптом під час його роботи, аналізує дані, що проходять через нього, і додатково оптимізує машинний код на основі цих знань.

Сучасний JavaScript – це «безпечна» мова програмування. Вона не забезпечує низькорівневий доступ до пам’яті або центрального процесора, оскільки спочатку була створений для браузерів, які цього не потребують.

Можливості JavaScript сильно залежать від середовища, в якому він працює. Наприклад, Node.js підтримує функції, які дозволяють JavaScript читати / писати довільні файли, виконувати мережеві запити тощо.

JavaScript у браузері може робити все, що стосується управління веб-сторінками, взаємодії з користувачем та веб-сервером[11].

Наприклад, JavaScript у браузері може:

- Додавати на сторінку новий HTML, змінювати наявний вміст сторінки, змінювати стилі
- Реагувати на дії користувача, клацання миші, рухи вказівника, натисканням клавіш.
- Надсилати запити через мережу на віддалені сервери, завантажувати та відправляти файли (так звані технології AJAX та COMET).
- Отримувати та встановлювати файли cookie, ставити питання користувачу, показувати повідомлення.
- Запам’ятовувати дані на стороні клієнта (“локальне сховище”).

Що не може робити JavaScript у браузері?

Можливості JavaScript у браузері обмежені заради безпеки користувача. Мета полягає в тому, щоб запобігти доступу злочинної веб-сторінки до приватної інформації або шкоди даним користувача.

Приклади таких обмежень включають:

- JavaScript на веб-сторінці не може читати / писати довільні файли на жорсткому диску, копіювати їх або виконувати програми. Він не має прямого доступу до функцій ОС.
- Сучасні браузери дозволяють цій мові працювати з файлами, але доступ обмежений і надається лише в тому випадку, якщо користувач робить певні дії, наприклад, завантажує файл у вікно браузера або вибирає його за допомогою тегу <input>.
- Існують способи взаємодії з камерою / мікрофоном та іншими пристроями, але вони вимагають явного дозволу користувача. Тож сторінка з підтримкою JavaScript може не несподівано включати або відключати веб-камеру, спостерігати за оточенням і надсилати інформацію користувача будь-куди без його згоди.

- Різні вкладки / вікна, як правило, не знають один про одного. Іноді вони це роблять, наприклад, коли одне вікно використовує JavaScript, щоб відкрити інше. Але навіть у цьому випадку JavaScript з однієї сторінки може не мати доступу до іншої, якщо вони надходять з різних сайтів (з іншого домену, протоколу чи порту). Це називається «Політика того самого походження». Щоб обійти це, обидві сторінки повинні домовитись про обмін даними та містити спеціальний код JavaScript, який це обробляє. Це обмеження, знову ж таки, стосується безпеки користувача. Наприклад сторінка з http://anysite.com, яку відкрив користувач, не повинна мати доступу до іншої вкладки браузера з URL-адресою http://gmail.com та викрадати звідти інформацію.
- JavaScript може легко зв’язатись через мережу з сервером, звідки походить поточна сторінка. Але здатність отримувати дані з інших сайтів / доменів обмежена. Хоча це можливо, для цього потрібна явна згода (виражена в заголовках HTTP) з віддаленої сторони – це обмеження безпеки.

Такі обмеження не існують, якщо JavaScript використовується поза браузером, наприклад на сервері. Сучасні браузери також дозволяють плагіни / розширення, які можуть вимагати розширених дозволів.

У мові програмування JavaScript є принаймні декілька чудових властивостей: повна інтеграція з HTML / CSS.Дуже простий поріг входу.Підтримка всіх основних браузерів ввімкнена за замовчуванням. JavaScript – єдина технологія браузера, яка поєднує ці три речі. Саме це робить JavaScript унікальним. Ось чому це найпоширеніший інструмент для створення інтерфейсів браузера.Тим не менш, JavaScript також дозволяє створювати сервери, мобільні додатки тощо[12].

Отже, JavaScript спочатку створювався як мова лише для браузера, але зараз він використовується і в багатьох інших середовищах. Сьогодні

JavaScript має унікальну позицію як найбільш широко прийнятої мови браузера з повною інтеграцією в HTML / CSS. Є багато мов, які "перетворюються" в JavaScript і надають певні функції.

2.2. Принципи побудови сучасних високонавантажених веб-додатків на основі мови Javascript

Створюючи веб додатки, слід дуже обережно ставитися до того, щоб не надсилати занадто багато JavaScript та CSS коду нашим користувачам. Бо чим більше ми надсилаємо даних користувачам, тим повільніше може ставати наш веб-додаток.

Тож підхід зазвичай полягає в тому, щоб «завжди надсилати якомога менше коду». І це добре, чим менше коду, тим краще. Менше часу на завантаження веб-сторінки, менше часу на обробку, менше часу на її побудову, менше часу на відавання вмісту на екрані[13].

Але, важливіше, ніж намагатися максимально стиснути наш веб-додаток, це зрозуміти шлях наших браузерів від споживання HTML, CSS та JavaScript і перетворення їх у пікселі на екрані – Критичний шлях рендерингу (КШР).

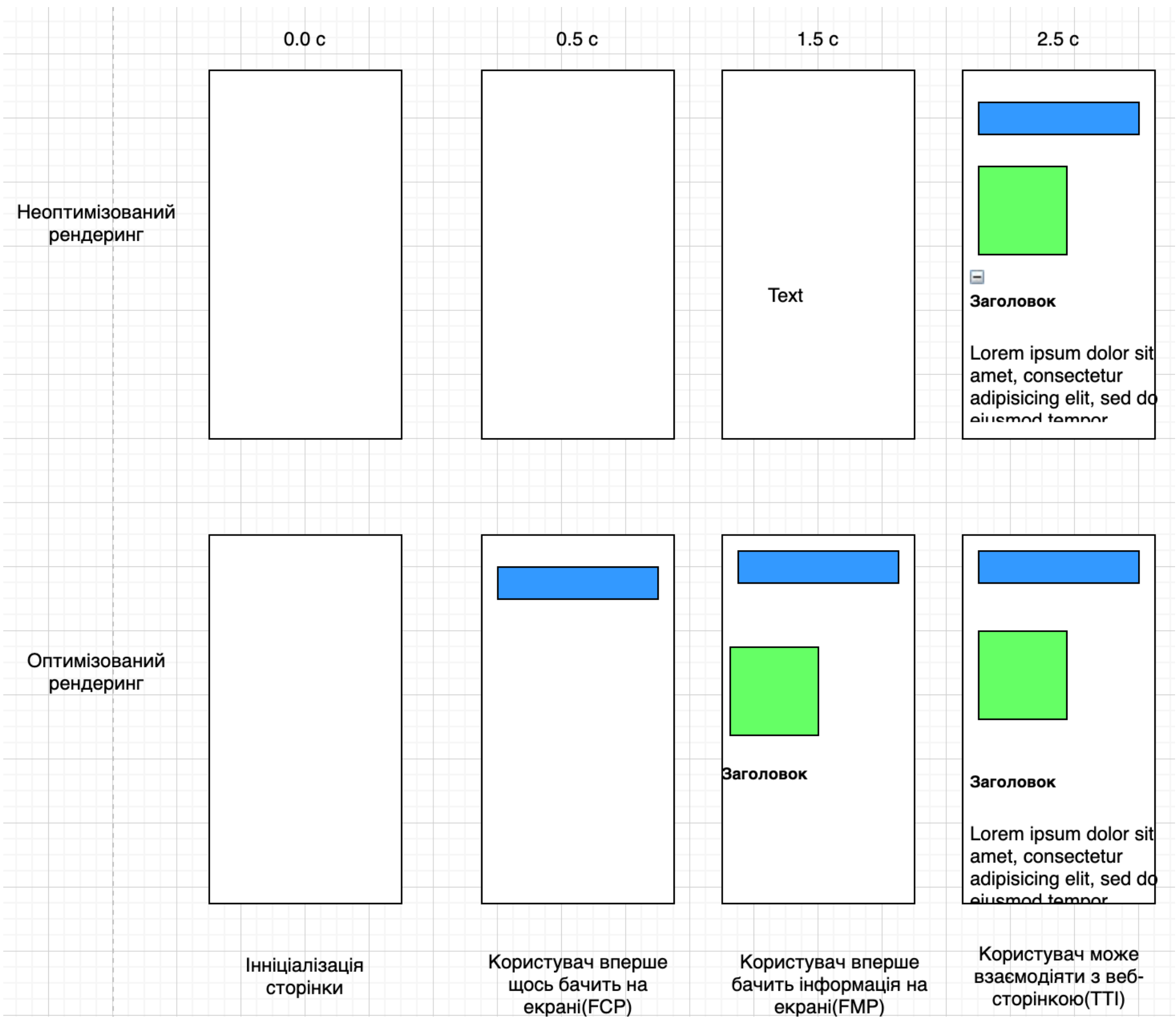

Рис 2.1 Оптимізований та неоптимізований рендеринг

Оптимізація КШР полягає у зменшенні критичних ресурсів та завантаженням некритичних неблокуючим способом, щоб досягти кращого часу завантаження сторінки.

Оскільки завантаження веб-сторінки не є одним моментом у часі, існує кілька етапів під час завантаження, які слід відстежувати, щоб можна було виміряти, наскільки „швидким” чи „повільним” є наш веб-додаток. Це:

- Коли користувач вперше бачить щось на екрані.(First contentful paint – FCP).
- Коли користувач вперше бачить щось, що несе інформацію на екрані.(First meaningful paint – FMP).

- Коли користувач може взаємодіяти з веб-сторінкою.(Time to interactive – TTI).

Існує кілька різних стратегій для вдосконалення цих показників , але оскільки не існує єдиного правильного рішення, яке задовольнятиме усі критерії, важливо розуміти КШР, щоб визначити стратегію, яка найкраще відповідає поставленій задачі[14].

Критичний шлях рендерінгу

Наступна діаграма охоплює всі основи КШР. З моменту виконання запиту та отримання першого байта:

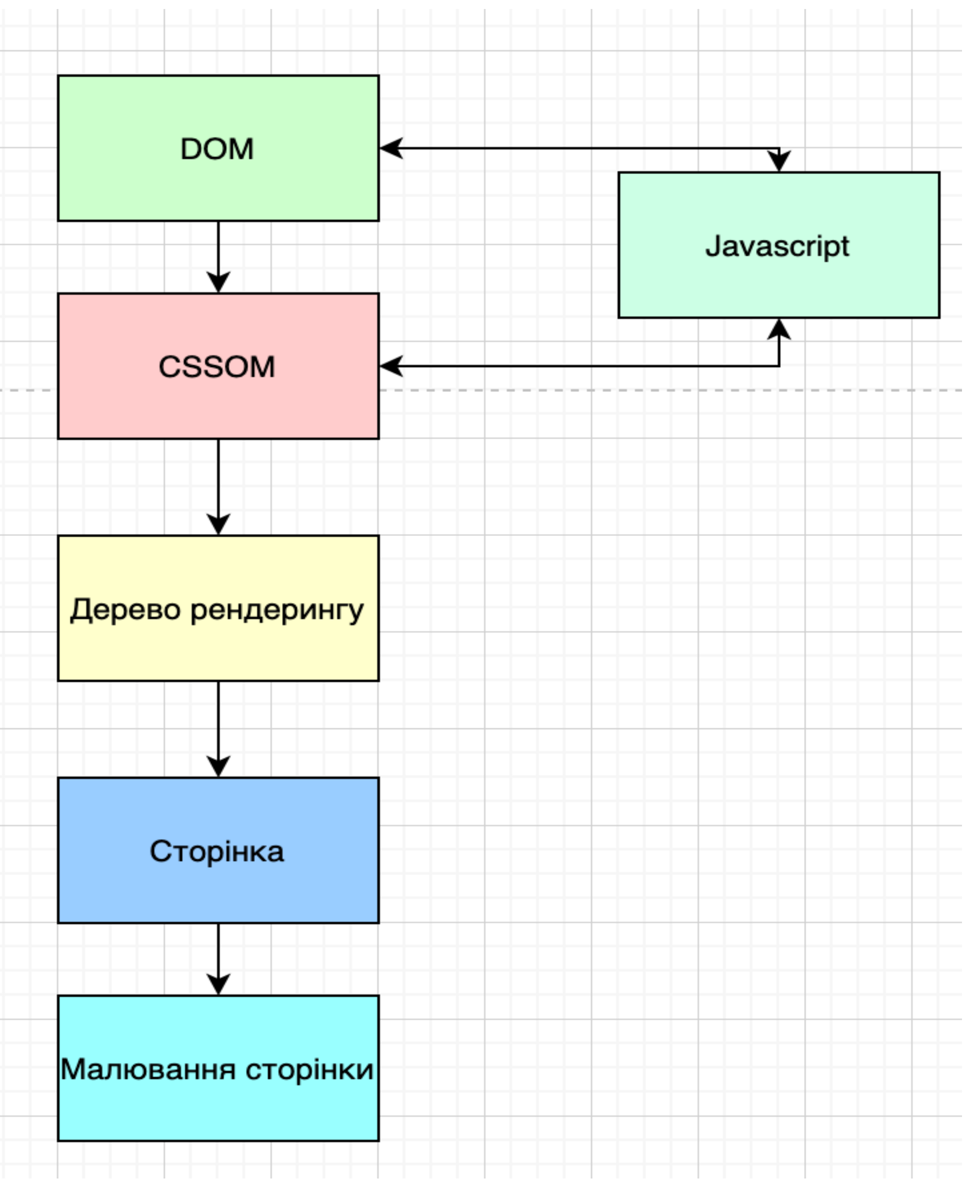

Рис. 2.2. Критичний шлях рендерингу

По-перше, береться HTML код і починає будуватися DOM-дерево, що містить всю інформацію про те, яка розмітка буде присутня.

Далі, після отримання CSS коду, іде побудова CSSOM-каркау, зберігаючи все, що пов'язано з тим, як представити вміст на екрані за допомгою стилів.

Потім об'єднуючи два вищезгадані кроки йду побудова дерева візуалізації, що містить інформацію про вміст веб-сторінки та її стилі.

Далі йде побудова макету. На цьому етапі вже відомо знаємо вміст і стилі веб-сторінки, але не вистачає одного, обчислити точну позицію та розмір кожного елемента у вікні перегляду пристрою - за це відповідальний макет[15].

Нарешті, коли макет завершено, спрацьовує подія відрисовки, перетворюючи дерево візуалізації в пікселі на екрані.

Javascript та КШР

У цьому прикладі ми маємо документ із деяким вбудованим JavaScript кодом. Проста HTML сторінка, на якій тег скрипту застосовує зміни до вмісту, що відображається на екрані.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>КШР</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
  </head>
  <body>
    <p>Тестова сторінка <span>КШР</span></p>
  </body>
  <script>
    var element = document.getElementsByTagName("p")[0];
    element.innerText = "Доброго дня";
  </script>
</html>
```

Рис. 2.3 Приклад HTML сторінки з Javascript кодом

У цьому випадку DOM-дерево починає поступово будуватися, доки не знайде тег <script>. Після цього побудова DOM-дерева призупиняється для запуску JavaScript коду, змінюючи DOM-дерево. Після цього побудова DOM-дерева відновлюється, і на екрані з'являється візуалізація даних.

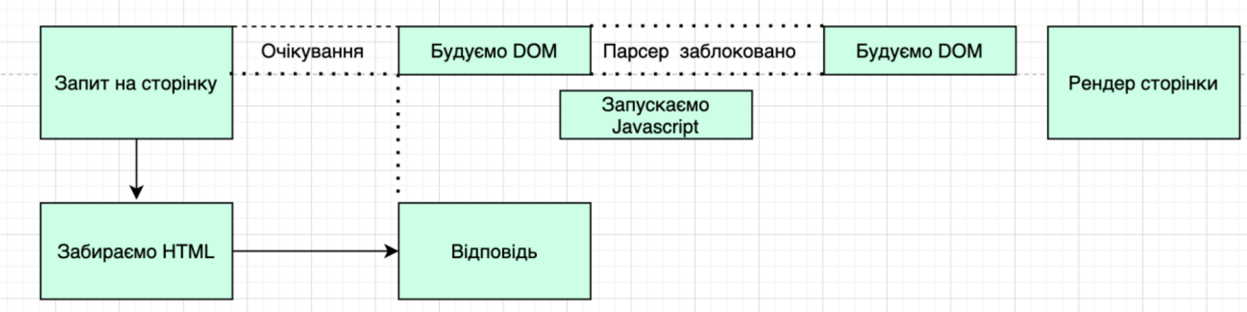

Рис. 2.4 Схема побудови веб-сторінки з HTML та Javascript

JavaScript код виконується досить швидко, здається, це не так сильно впливає на КШР. І це може бути правдою, коли йдеться про досить простий вбудований скрипт, як було згадано вище. Але для зовнішніх залежностей побудова DOM-дерева буде заблокована, поки запит HTTP не відбудеться і не запуститься отриманий Javascript код, і цей запит може потребувати деякого часу[16].

Хороша практика – уникати вбудовування ресурсів більше 1 кб і примінення стратегій кешування даних.

Було розглянуто критичний шлях рендерингу та те, як ресурси Javascript впливають на завантаження сторінки. Трохи торкнулися деяких основних стратегій покращення продуктивності завантаження сторінки, але є багато іншого, про що слід пам'ятати, говорячи про ефективність завантаження сторінки[16].

Для того, щоб високонавантажені веб-додатки не витрачали багато часу на завантаження веб-сторінки треба оптимізувати КШР.

Можна виділити 3 основі напрямки оптимізації: оптимазіція розміру заванатажуваних ресурсів(компресія та кешування Javascript, HTML, CSS), скорочення критичних ресурсів(мінімізація використання засобів блокування

візуалізації та парсингу веб-сторінки), скоротити довжину КШР(тривалість КШР є функцією графіка залежності між критичними ресурсами та їх розміром. Деякі завантаження ресурсів можуть бути ініційовані лише після обробки попереднього ресурсу, і чим більший ресурс, тим більше часу займають кругові завантаження)[17].

Оптимізація розміру завантажуваних ресурсів:

- Надсилання менш важливого коду, який можна завжди мініфікувати, стиснути та кешувати.
- Використання таких інструментів, як webpack, для мінімізації пакетів для розробки у продакшн режимі т застосування розбиття коду, щоб можна було завантажувати критичні фрагменти і відкладати некритичні.
- Використання трас396пованих пакетів та поліфілів для кожного браузера, використовуючи такі інструменти, як browserslist, core-js та “usebuiltIns” від babel.
- Застосування стиснення файлів на веб-сервері, використовуючи такі алгоритми, як Brotli або Gzip.

Скорочення критичних ресурсів:

- Використовування медіа-запитів на тегах <link>, щоб відкласти CSS, який не відповідає поточним цілі (йдется про розмір екрану девайсу з якого було відкрито веб-сторінку).
- Завантаження некритичних ресурсів неблокуючим шляхом, використовуючи атрибути async та defer на тегах скриптів.
- Попереднє завантаження та попереднє підключення скриптів, які можуть знадобитися користувачеві під час користування додатком. Це не дозволить користувачеві чекати цих ресурсів, коли вони потрібні (особливо корисно при ледачому завантаженні).

Скорочення довжини КШР. Це сильно залежить від версії HTTP протоколу, яка використовується. Зважаючи на це, треба надавати перевагу другій версії протоколу HTTP для доставки програмного коду. Але якщо немає можливості, є деякі інші шляхи, які можна застосувати під час використання HTTP протоколу першої версії.

У цьому розділі стало зрозуміло як браузер обробляє ресурси веб-програми та як це впливає на час завантаження сторінки, а отже як правильно будувати високнонавантажені веб-додатки. Які загальні основні правила для оптимізації продуктивності, як забезпечити дотримання ефективності завантаження сторінки та як це впливає на користувачів у реальному світі[20].

2.3. Помилки у Javascript

2.3.1. Інструменти відслідковування помилок у готових веб-додатках написаних на мові програмування Javascript

На сьогоднішній день постає дуже велика проблема у відслідковуванні помилок у висконавантажених веб-додатках. Адже силами мови програмування Javascript, не вдається цілком впоратися з цією проблемою. Тому потрібно шукати рішення для того, щоб якісно відслідковувати помилки у високонавантажених веб-додатках.

Для того, щоб вдало підібрати інструмент слід виділити декілька ключових проблем, які повинен вирішити цей інструмент:

- Відстеження помилок без зайвих втручань.
- Маленький розмір інструменту відслідковування помилок, для швидкого завантаження.
- Відстеження того, де саме сталася помилка(з/без стек трейсу).
- Отримання повного контексту помилок для того щоб подивитися, що саме пішло не так. Соурсмапи збиратимуться та застосовуватимуться автоматично, не вимагаючи жодних дій.

- Відстеження помилок у мові програмування JavaScript, а не користувачів. Служби, які здійснюють відстеження користувачів, зберігатимуть персональні дані і, відповідно, вимагають отримання явної згоди користувача відповідно до законодавства.
- Деталі браузеру користувача
- Інфорграфіка
- Зв’язані помилки серверної та клієнтської частин
- Відслідковування помилок тільки клієнтської частини
- Відслідковування помилок тільки серверної частини

Гарне відпрацювання програми залежить від того, наскільки швидко можна знайти, що уповільнює ситуацію. Продуктивність, моніторинг продуктивності, надає повну видимість стека, необхідну для швидшої ітерації, зменшення ризиків і, врешті-решт, постачання якісного програмного забезпечення. Завдяки виміру швидкості завантаження сторінки, команди можуть відстежувати сторінки із повільним завантаженням до їх виклику API, а також виявляти всі пов’язані помилки. Це надає розробникам вирішити вузькі місця у коді та забезпечити швидкий, надійний досвід, який вимагає клієнт. Відстеження неефективних сторінок не лише за викликом API. Деталі подій продуктивності візуалізує досвід клієнта від початку до кінця, і все це при підключенні даних пристрою користувача до очікуваної роботи[21].

За допомогою моніторингу ефективності можна переглядати транзакції за найменшим часом, пов’язані проблемами або кількістю користувачів – все в одному зведеному звіті. І маркери випуску додають ще один рівень контексту, щоб команда могла визначити, як клієнти реагують на код, який нещодавно був випущений у виробництво.

Середовища виконання допомагають краще фільтрувати проблеми, випуски та відгуки користувачів на сторінці. На сторінці можна переглянути інформацію про конкретне середовище, зосередившись на останньому релізі. Якщо використовується багатоступеневий процес релізів, також можна

вибрати інше середовище за замовчуванням і встановити умови, що відповідають атрибуту середовища, щоб обмежити попередження лише певними стадіями випуску. Навколишнє середовище доступне для більшості функцій.

Використання проектів для розділення різних служб чи програм, а середовища – для виділення різних середовищ або етапів випуску в кожному. Якщо було вибрано один або кілька проектів у загальному заголовку веб-інтерфейсу відслідковувач помилок відображає лише середовища, пов'язані з подіями з вибраних проектів.

Отже, на сьогоднішній день є два готових рішення для відслідковуваання помилок у високонавантажених веб-додатках, це Sentry.io та Catch.js. Також є ще третрій варіант – це власна програмна реалізація, яка задовольняє всі вищезагадані вимоги або навіть є кращою та більш конфігурованою. Далі буде розглянуто більш детально кожен з вищезгаданих варіантів.

2.3.2. Sentry.io та його характеристика

Sentry.io – це бібліотека для відслідковування помилок у високонавантажених веб-додатках. На даний момент це рішення є безшкотовним для пробного періоду, проте якщо використовувати його у великій екосистемі, такій як наш додаток, то його послуги та масштабування будут коштувати чималих коштів. Що пропонує sentry.io для відслідковування помилок у високонавантажених веб-додатках написаних на мові програмування Javascript?

Sentry.io без додаткових налаштувань пропонує:

- Відслідковування HTTP запитів
- Відслідковування змін у адресній строці браузера
- Кліки користувача по комонентам дом дерева у браузері
- Збір консольних метрик
- Посилання будь-яких помилок, які сталися під час користування веб-додатком

Процес налаштування Sentry.io не є дуже складним, для цього спочатку потрібно через CDN(Content delivery network), або попередньо завантаживши програмний код, підключити скрипт Raven.js. Raven.js – офіційний клієнт написаний на мові програмування JavaScript для Sentry. Він автоматично повідомляє про невідслідковані мовою програмування JavaScript помилики, винятки, викликані із середовища браузера, та забезпечує багато розширених методів API для повідомлення про власні помилки[23].

```
<body>
    <script
        src="https://cdn.ravenjs.com/3.26.4/raven.min.js"
        crossorigin="anonymous"
    ></script>
</body>
```

Рис. 2.5 Приклад підключення бібліотке Sentry.io

На перший погляд, все дуже легко, проте слід пам'ятати що будь-який сторонній скрипт підключений до сторінки високонавантаженого веб-додатку, може спричинити повільну поведінку веб-сторніки у браузері, адже перш ніж показати сторінку користувачу, відбувається завантаження всіх Javascript файлів, незважаючи на їх розмір. А оскільки, ми дбаємо про наших користувачів, то ми вірогідно захочемо одразу відслідковувати їх помилки.

Якщо заміряти розмір цього скрипту, то можна сказати, що він складає приблизно 300 кілобайт, а якщо у користувача повільний інтернет, то це достатньо великий розмір, завантаження якого спричинить певну затримку для користувача при отриманні веб-сторінки. Перейдемо далі до налаштувань.

Для того, щоб Raven скрипт слідкував за всіма помилками, ініціалізацію нашого додатку потрібно обернути у таку конструкцію:

```
Raven.context(function() {
  initMyApp();
});
```

Рис. 2.6 Приклад ініціалізації додатку з Sentry.io

Для того, щоб звітувати про певні помилики, які в нас є бажання контролювати, є спеціальна конструкція Raven.captureException(error). Де параметр, error це саме помилка, яку ми хотіли би відслідкувати, побачити її стек, для того, щоб в майбутньому виправити. Також є великий ньюанс для того, щодо того, як викидати помилки обов'язково їх обертати у конструкцію new Error(), при викиданні, тобто просто throw "Error", працювати не буде, потрібно зробити так throw new Error("broken").

Отже, можна зробити висновок, що Sentry.io, досить непогане рішення для відслідковування помилок у високонавантажених веб-додатках написаних на мові програмування Javascript. Проте є декілька недоліків, це спеціальний синтаксис для відслідковування помилок, також це рішення є платним і потрібно завантажувати на ваш веб-додаток ще один скрипт, яких і так у високонавантажених веб-додатках вистачає[24].

2.3.2. Catch.js та його характеристика

Catch.js – це бібліотека для відслідковування помилок у високонавантажених веб-додатках. На даний момент це рішення є безшкотовним для пробного періоду який триває лише 14 днів, проте якщо використовувати його у великій екосистемі, такій як наш додаток, то його послуги та масштабування будут коштувати чималих коштів. Що пропонує catch.js для відслідковування помилок у високонавантажених веб-додатках написаних на мові програмування Javascript?

Catch.js без додаткових налаштувань пропонує:

- Відслідковування HTTP запитів
- Знімки екранів користувачів при помилці
- Кліки користувача по комонентам дом дерева у браузері
- Збір консольних метрик
- Інформація про браузер користувача де сталася помилка

Інтеграція у веб-додаток досить не складна, спочатку треба зареєструватися на платформі catch.js, та вписати ваш домен на якому буде відбуватися відслідковування помилок. Після цього потрібно буде підключити скрипт:

```
<script src="//cdn.catchjs.com/catch.js"></script>
```

Рис. 2.7 підключення скрипту catch.js

Після цього можна налаштувати у власному кабінету, опції, які є бажання відслідковувати, це помилки у виконуваному Javascript коді, знаходження помилок у соурсмапах і т.д[25].

Варто зауважити, що вага скрипта catch.js складає близкь 200 кілобайт, що також досить багато, для бібліотеки, яка відслідковує помилки.

Використовуючи catch.js, слід пам'ятати про те, що замість звичайного console.log(), який надає нам Javascript, потрібно використовувати catchjs.log(), для того, щоб інформація потрапляла до особистого кабінету, та можна було дивитися статистику. Метод приймає змінну кількість параметрів будь-якого типу. Вони будуть серіалізовані та відправлені до служби CatchJS. Якщо ви передасте об'єкт у catchjs.log (), він відобразиться у вашому журналі в особистому кабінеті, подібно до того, як вони відображатимуться у інструментах розробника в браузері.

Отже, можна зробити висновок, що catch.js, також непогане рішення для відслідковування помилок у високонавантажених веб-додатках, проте воно також платне, и досить велике по розміру.

2.4. Порівняння існуючих рішень відслідковування помилок у високонавантажених веб-додатках на основі мови Javascript та формулювання власних вимог.

Вище було розгялнуто два готові рішення для відслідковування помилок у високонавантажених веб-додатках на основі мови Javascript. Проте, треба вирішити чи задвольняють вони наші вимоги? Для початку сформулюємо їх.

1. Нам потрібно мати інформацію про кожен, або майже кожен XMLHttp запит, який відбувається на стороні серверу або клієнтській частині. Тобто, якщо йде запит на завантаження нашого веб-додатку, або потрібно відслідкувати який-небудь конкретний запит по спецільному імені, ми повинні вміти це зробити. Якщо поглянути, на вищезгадані рішення, вони можуть відслідковувати запити, які викликали помилку, просто не можна за допомогою цих додатків переглянути зв’язані запити, або який запит передував помилковому.

2. Мати інформацію про браузер та user-agent користувача, щоб зрозуміти як відтворити помилку та у якому середовищі виправляти цю помилку.

3. Потрібно мати інформацію про тип помилки, яка сталася, тобто це може бути помилка при виконанні коду, або помилка при запиті на будь-який ресурс.

4. Рішення має бути гнучким та мати малий розмір, щоб не переривати виконання програмного коду веб-сторніки, тобто можна відслідковувати помилки користувача, серверу, завантаження сторонніх скриптів та бібліотек.

5. Має бути зв’язанність інформації про помилки, та збір статистики, отже кожен запис про помилку, повинен мати певний індентифікатор для того, щоб переглянути зв’язані запити або помилки.

6. Бажано, щоб при записі інформації про помилку, можна було побачити місце в кодовій базі, де саме сталася помилка, тобто виконання якого програмного коду спричинило помилку.

7. Рішення має бути безшкоштовним, та зберігати історію помилок для якісного та кількісного порівняння інформації про помилки.

З вищесформульованих вимог, можна зробити порівняннльну таблицю, та прийняти рішення чи потрібна нам власна реалізація програмного модулю

для відслідковування помилок у високонавантажених веб-додатках. А також побачити, які саме пункти відповідають чи не відповідають сформульованим вимогам вище.

Зробимо порівняльну таблицю:

Табл. 2.1. Порівняльна таблиця рішень для відслідковування помилок у високонавантажених веб-додатках.

Назва рішення / Вимога	Catch.js	Sentry.io
Інформація про кожен XMLHTTP запит	-	-
Інформація про оточення користувача	+	+
Детальна інформація про тип помилки	+	+
Гнучкість	-	-
Зв'язаність інформації про помилку з іншими запитами	-	-
Визначення конкретного місця де сталася помилка при виконанні програмного коду	-	+
Збереження історії помилок та безкоштовність рішення	-	-
Розмір програмного модулю	300кб	200кб

Отже, проаналізувавши вищезагадані вимоги, можна зробити висновок, що для того, щоб якісно слідкувати за помилками у високонавантежених веб-додатках написаних на мові програмування Javascript, потрібно розробляти власний програмний модуль, який задовольнятиме усі вимоги. Також великою перевагаю цього рішення є те, щоб в подальшому його можна буде власноруч кастомізувати та налаштовувати під потреби додатку.

2.5 Висновки до другого розділу

Можна сказати, що спочатку JavaScript створювався як мова лише для браузера, але зараз він використовується і в багатьох інших середовищах. Стало зрозуміло як браузер обробляє ресурси веб-програми та як це впливає на час завантаження сторінки, а отже як правильно будувати високнонавантажені веб-додатки. Які загальні основні правила для оптимізації продуктивності, як забезпечити дотримання ефективності завантаження сторінки, та як це впливає на користувачів у реальному світі.

На сьогоднішній день є два готових рішення для відслідковуваання помилок у високонавантажених веб-додатках написаних на мові програмування Javascript, це Sentry.io та Catch.js.

Проаналізувавши ці рішення, та склавши певні вимоги для програмного модулю, можна зробити висновок, що для реалізації задуманої мети є потреба у побудові власного програмного модулю.

Розділ 3. РОЗРОБКА АЛГОРИТМУ ТА ПРОГРАМНОГО МОДУЛЮ ВІДСЛІДКОВУВАННЯ ПОМИЛОК У ВИСОКОНАВАНТАЖЕНИХ ВЕБ-ДОДАТКАХ

3.1. Розробка алгоритму роботи програмного модулю

Перш ніж розробляти алгоритм програмного модулю, перш за все слід обрати технології для його реалізації. Для розробки данного програмного модулю було взято за основну мову програмування – Javascript. Також слід зазначити, щоб розробка алгоритму буде відбуватися, до вже функціонуючого високонавантаженого веб-додатку, написаного на мові програмування Javascript.

Для того, щоб вдало розробити алгоритм відслідковування помилок, слід знати, що у висконавантаженому веб-додатку використовуються підхід SSR(Server side Rendering) рендеринг веб-сторінки на сервері, та технологія кешування, про яку було згадано у першому розділі.

Технологія для клієнтської логіки – це Vue.js, та декілька бібліотек які допоможуть відправляти запити та взаємодіяти з користувачем такі як axios, vuex, vue-router, localStorage. Scss, lodash. Адже Vue є цілою екосистемою для побудови інтерфейсів користувача. Vue також цілком здатний запускати складні програми з однією сторінкою(SPA – single page application), використовуючи їх у поєднанні з сучасними інструментами та бібліотеками.

Технологія для серверної логіки – це Node.js, Express.js та Typescript, які на сьогоднішній день є найбільш вдалими та популярними рішеннями для побудови серверної логіки, та взаємодії з SSR технологією.

Розробку програмного модулю можна умовно поділити на дві частини:

1. Розробка алгоритму спеціального логеру, для збирання інформації про помилки та різних метрик.

2. Розробка алгоритму відслідковування місця де саме сталася помилка.

Розробка алгоритму логеру

Логер в даному випадку буде викликатися безпосередньо на стороні серверу, проте, його можна буде використовувати і на клієнтській частині, оскільки ми використовуємо технологію серверного рендерингу. Для початку слід зазначити, що даний логер буде окрім логування помилок, ще записувати, наприклад повідомлення інформаційного характеру, які можуть бути пов'язані з помилками, для того, щоб покроково відтворити сценарій помилки.

Для початку потрібно розмежувати типи повідомлень, які будуть логуватися. У нашому випадку, це буде параметр level_name, він може бути дорівнювати значенням:

- INFO, для того щоб збирати будь-яку інформацію про продукт на стороні серверу, та швидку філтрацію по ним.
- CLIENT_INFO, для того щоб збирати будь-яку інформацію про продукт на стороні клієнту, та швидку філтрацію по ним.
- WARNING, для того щоб збирати будь-яку інформацію про попередження на стороні серверу, та швидку філтрацію по ним.
- CLIENT_CRITICAL, для того щоб збирати інформацію про критичні помилки на стороні клієнту, та швидку філтрацію по ним.
- CRITICAL, для того щоб збирату інформацію про критичні помилки на стороні серверу, та швидку філтрацію по ним.
- DEBUG, для того щоб збирату інформацію про код для розробки або дебагінгу на стороні клієнту або серверу, та швидку філтрацію по ним.
- CLIENT_WARNING, для того щоб збирату інформацію про попередження на стороні клієнту, та швидку філтрацію по ним.

Далі опишемо інтерфейс, наших повідомлень про помилки, які будемо безпосередньо записувати в журнал. Інтрефейс буде складатися з параметрів:

- label, параметр, який дає загальну інформацію про повідомлення.
- message, поле, яке буде містити у собі повідомлення про помилку або будь-яку інформацію яку ми будемо передавати в логер.
- context, необов’язкове поле, яке міститтиме у собі інформацію про контекс виклику нашого логеру.
- level_name, поле, в якому потрібно вказати тип повідомлення, можливі типи (INFO, CLIENT_INFO, WARNING, CLIENT_CRITICAL, CRITICAL, DEBUG, CLIENT_WARNING).
- scope, поле в якому буде вказано об’єм помилки, та де вона була викликана.
- stack, стек, який був створений при умовах коли відбулася помилка.

Нижче приведено опис інтерфейсу для нашого логеру. Інтерфейси у мові програмування Typescript(типізований Javascript), потрібні для явного опису моделі даних, які буде приймати той чи інший метод або клас.

```
interface IMessage {
   label: string;
   message: string;
   context?: object;
   level_name?: string;
   scope?: object;
   stack?: string;
}
```

Також із допоміжних функцій для логеру, нам буде потрібен метод extractRequestParams, параметром буде приймати в себе безпосередньо запит, який ми хочемо залогувати, нижче наведена його програмна реалізація:

```
function extractRequestParams(req?: Request) {
   if (!req) {
```

```
        return {};
      }

      const cookie_auth = AuthorizationService.authTokenUid(req) || "false";

      return {
        xReqId: req.headers["x-request-id"],
        userAgent: req.headers["user-agent"],
        geoIpCountryCode: req.headers["geoip-country-code"],
        url: req.url,
        remote_addr: req.headers["user-ip"],
        host: HostHelper.get(req),
      };
    }
```

Із наведеної реалізації можга зрозімити, що перш ніж сформувати параметри нашого запиту, спочатку йде перевірка на те, чи було взагалі передано запит у функцію, якщо ні, то функція припиняє своє виконання.

Далі йде перевірка на авторизаційну куку користувача, тобто перевірка на те, чи йде запит від авторизованого користувача чи ні, якщо користувач авторизований, то ми беремо звідти його авторизаційну куку, за допомогою авторизаційоного сервісу, якщо куки немає, то просто повертаємо булеве значення false.

На останок формуємо об'єкт, який містить в собі інформацію про айді запиту, який буде потрібен для зв'язаності логів та запитів які, можуть бути пов'язані з помилкою або логом, який буде записано до журналу. UserAgent, поле яке, повідомляє нам, просто тип браузеру користувача, та пристрій з якого користувач зайшов на наш високонавантажений веб-додаток. geoIpCountryCode, параметр який визначає країну з якої було здійснено запит. Url, адреса з якої було здійснено запит користувача. Remote_addr, IP адреса користувача. Host, хост на який було здійснено запит.

Необхідність у цій допоміжній функції потрібна для того, щоб в майбутньому сформувати більш точний запис у журнал подій, для більш детальної інформації про подію.

Переходимо безспосередньо до реалізації нашого логеру:

```
const log = (message: string | any, req?: Request) => {
  const {
    xReqId,
    userAgent,
    geoIpCountryCode,
    url,
    host,
    remote_addr,
    cookie_auth,
    cookie_guest,
  } = extractRequestParams(req);
  Object.assign(message, {
    datetime: new Date().toISOString(),
    type: `${ENVIRONMENT ? "production" : "release" `,
    rid: xReqId,
    cookie_auth,
    remote_addr,
  });

  message.context = Object.assign({}, message.context || {}, {
    "project": PROJECT,
    "x-request-id": xReqId,
    "user-agent": userAgent,
    geoIpCountryCode,
    url,
    host,
```

```
    "state": {},
  });

  if (!IS_SHORT_LOGGER) {
    console.log(JSON.stringify(message));
  } else {
    console.log(message.message);
  }
};
```

Логер приймає у себе 2 парамтера, це message – повідомлення, яке ми хочему залогувати, та req – це запит який було здійснено під час запису.

Далі йде виклик фукнціі extractRequestParams, яку було описано вище, з неї беруться усі необхідні параметри про запит, який відбувся.

До взідного парметру message, дадється інформація про дату та час(datetime), для того щоб інформацію про помилки можна було ранжувати по часу та даті. Також додається поле type, яке відповідає за оточення в якому буде відбуватися записа, оточення для розробки, або оточення з яким взаємодіють користувачі.

Після цього, формується контекст запиту якщо він є, якщо не має формуємо його з переліку параметрів project(проект на якому відбувається запис), x-request-id(айді запиту для зв’язаності з іншими запитами), user-agent(інформвція про оточення користувача), geoIpCountryCode(інформація про країну користувача з якої відбувається запит).

Після того, як ми було сформовано об’єкт для запису його в журнал вібудваться виклик оператора, який логує нашу інформацію. Прото слід зазначити, що для використання по всьому проекту ми будемо формувати набор методів, яку будуть експортуватися та ранжуватися по параметру level_name, як було зазначено вище.

На виході, будемо мати по своїй суті клас, з набором методів:

```
export default {
```

```
info(message: IMessage, req?: Request) {
   message.level_name = "INFO";
   log(message, req);
},

clientInfo(message: IMessage, req?: Request) {
   message.level_name = "CLIENT_INFO";
   log(message, req);
},

warning(message: any, req?: Request) {
   message.level_name = "WARNING";
   log(message, req);
},

clientError(message: IMessage, req?: Request) {
   message.level_name = "CLIENT_CRITICAL";
   log(message, req);
},

error(message: IMessage, req?: Request) {
   message.level_name = "CRITICAL";
   log(message, req);
},

dev(message: any, req?: Request) {
   message.level_name = "DEBUG";
   log(message, req);
},
clientWarning(message: IMessage, req?: Request) {
```

```
        message.level_name = "CLIENT_WARNING";
        log(message, req);
    },
};
```

Клас має 7 методів, кожен з яких приймає 2 параметри, які потрібні для методу log, який було описано вище, це message – повідомлення, яке ми хочемо записати, та req – запит який відбувається, щоб взяти з нього інформацію про оточення користувача та виконуваного програмного коду.

Різниця лишу у значенні level_name. Це було зроблено для того, щоб експортувати логер, та мати можливість виклкиати метод логера, з тим рівнем який потрібно, для репрезентативності даних.

Тепер перейдемо до реалізації алгоритму логеру на стороні клієнта, оскільки ми хочем бачити помилки не тільки на стороні серверу, а на клієнтській стороні нашого вискогонавантаженого веб-додатку.

Для цього потрібно реалізувати клас з набором статичних методів, який буде збирати інформацію про подію, яка сталася, та відправляти її на сервер, для того, щоб відбуалася операція запису в журнал.

Метод клієнтського логеру буде приймати чотири параметри:

- isServer, параметр, який дає інформацію про те, лог на стороні серверу чи ні, адже на проекті використовується технологія серверного рендеру, зазвичай по дефолту його значення false.
- context, параметр, який дає інформацію про контекст виконання коду, під час якого буде викликаний логгер.
- callback, параметр, який потрібен, для того, щоб при запуску логеру, прив’язати функцію, котра буде пересилати наші записи на серверну частину.
- scope, параметр, який потрібен у екземплярі класу, який ми викликаємо, для того щоб прив’язати контекст виконання.

```
class Logger {
    constructor(isServer, context, callback, scope) {
```

```
        this.$isServer = isServer;
        this.$context = Object.assign({}, context);
        this.callback = callback;

        let methods = Logger.getMethods();

        methods.map((method) => {
            scope[method] = (...args) => {
                return this.handler(method, ...args);
            };
        });
    }
```

На початку ми декларуєм наш клас з назвою Logger, після цього викликаємо метод за вмовчанням у будь-якого класу це constructor, який приймає в себе 4 параметри описані вище, та присваюємо ці параметри внутрішнім властивостям нашого класу.

```
    static getMethods() {
        return [
            "info",
            "warn",
            "error",
        ];
    }
```

Далі створюється зміна methods, яка викликає метод класу gethMethods(), для того, щоб завантажити усі клієнтські методи, які були зроблені на основі параметру level_name, у нашому випадку це: info, warn, return, що еквівалентно значенням: CLIENT_INFO, CLIENT_CRITICAL, CLIENT_WARNING. Ці параметри були описані вище.

```
    processValue(obj) {
```

```
    let result = {};

    if (typeof obj === "object") {
      for (let name in obj) {
          let value = obj[name];

          if (name === "userAccessToken") {
            result[name] = value || "false";
          } else if (typeof value === "string") {
            result[name] = value;
          } else {
            result[name] = JSON.stringify(value);
          }
        }
      }
    }

    return result;
  }
```

Метод processValue приймає на вхід об’єкт, який має в собі контекст виконання коду, та користувацьку інформацію, щоб було легше індентифікувати про те, в якому стані знаходиться обліковий запис користувача. Спочатку йде перевірка на вхідний параметр, щоб він був типу oject, після цього запускається цикл, в якому створюється змінна value, в якій заберігається значення поля вхідного об’єкту. Далі йде перевірка на авторизаційний токен користувача, щоб можно зрозуміти лог був викликаний у авторизованого користувача, чи ні. Після цього йдуть перевірки на структури даних, і в кінці ми отримуємо наш результат з розподіленими полями контексту виконання.

```
  handler(method, lab, err = {}) {
```

```
let label = lab;
let error = err;

if (typeof label !== "string" && typeof error === "undefined") {
   error = label;
   label = "";
}

let obj = {
   label: label || "INITIAL_LOG_FETCH_LABEL",
   level_name: method.toUpperCase(),
   remote_addr: this.$context.userIp,
   datetime: new Date().toISOString(),
   type: this.$context.type,
   rid: this.$context["x-request-id"],
   context: {},
   message: "INITIAL_FETCH_WORKS",
};

if (err && err.context) {
   Object.assign(obj, {
      context: err.context,
   });
}

if (error && error.message) {
   Object.assign(obj, {
      message: error.message,
   });
}
```

```
            if (error && error.stack) {
                Object.assign(obj, {
                    stack: error.stack,
                });
            }

            if (isObjectEmpty(obj.context)) {
                obj.context = this.processValue(this.$context);
            }

            if (DEV) {
                if (err instanceof Error) {
                    return err;
                }
                return new Error(err);
            }

            this.callback(obj);
        }
    }
```

Метод handler викликається у констурякторі класу, для того, щоб заповнити поле scope, в якому по суті і міститься основна інформація.Метод handler приймає три параметри це – method, який буде йти у конструкторі з методу getMethods, lab, це параметр назви нашого логу, тобто під яким іменем його буде записано в наш журнал подій, та параметр err. Параметр помилки, якщо вона є, для того, щоб далі її ідентифікувати та визначити у частині, яка буде визначати де саме сталася помилка у виконвуваному коді. Після цього йдуть деякі перевірки та присвоєння значень внутршнім змінним функції. Найважливішою частиною цього коду, є формування об’єкту помилки,

поглянувши на нього, можна зробити висновок, що усі основні параметри які були описані в серверній частині тут також присутні, вони беруться з контексту виконання коду. Далі йде вибріка контексту, та фінальний крок – це виклик функції-колбеку для виконання.

```
module.exports = {
  init(config, callback) {
    new Logger(config.isServer, config.context, callback, this);
  },
};
```

Наприкінці ми експортуємо клієнтський логер з функцією init, яка приймає в себе два параметри, це config та callback. У параметрі config, міститься вся інформавція, для вхідних параметрів логеру. У параметрі callback, функція-колбек, тобто та функція, яка буде викликана після ініціалізації нашого логеру.

Розробка алгоритму відслідковування місця де саме сталася помилка

Оскільки код який отримує користувач веб-додатку має обфускований вигляд, тобто до коду застосовується механізм обфускації. Обфускація – це приведення початкового коду або виконуваного програмного коду до вигляду, який зберігає його функціональність, але ускладнює аналіз, розуміння алгоритму роботи і модифікації при декомпіляції.

Оскільки Javascript код є інтерпритованим, тобто винонується построково, то дуже легко відслідкувати який саме процес виконується і використати це в зловмисних цілях. Отже, обфускація використувується в цілях безпеки програмного коду для користувачів та для нашого висконавантаженого веб-додатку.

Також при компіляції вихідного коду, який беспосередньо отримує користувач, необхідно ховати соурсмапи, які утворюються при компіляції вихідного коду проекту, для того щоб прискорити виконання коду та полегшити процес пошуку потрібних частин коду по проекту. Проте

соурсмапи можуть бути використані зловмисниками, для того щоб побачити будь-які вразливості в коді, та скопіювати його собі, тому на проекті вони ховаються.

Проте, постає питання, як залишити соурсмапи, які потрібні для відслідковування місця де саме сталася помилка, і не видавати їх видимість в загальний доступ для всіх користувачів? Є дві опції: не використовувати соруспами при білді проекту, або дозволити їх використання, проте заборонити їх доступність по прямому URL. Отже, було обрано другий варінат, оскільки він потрібен для відслідковування місця де саме сталася помилка. Робиться це таким чином:

```
server.get("*/sourcemaps/*", suppress);
export const suppress = (req: Request, res: Response) => {
   res.status(403).end()
}
```

З даного фрагменту коду, бачимо, що по будь-якому запиту в нашому веб-додатку для користувача який буде містити у собі ключове слово sourcemaps, а сами за запитом виду “link/sourcemaps/Main.js” додаток може віддавати код соурсмапі, буде викликатися функція-колбек suppress. Реалізація якої буде віддавати 403 статус на виклик соурсмапи, проте, для використання всередині проекті соурсмапи доступні.

Далі, беспосердньо розглянему реалізацію відслідковування помилок у конкретному місці де сталася помилка:

```
import fs from 'fs';
import sourceMap from 'source-map';
import path from "path";
import {Request, Response} from "express";
```

Почнемо з імпортів, імпортуємо бібліотеку для роботи з файловою системою. Далі бібліотеку яка беспосердньо працює з соурсмапами sourceMap, та бібліотеку, яка працює з шляхом до файлів. Також ще імпортуємо типи запиту та відповіді, оскільки використується Typescript на проекті.

```
const getErrorPosition = (stack = "") => {
  try {
    const lastStack: string = stack.match(/[(].*[)]/)![0].replace(/[()]/g, "");
    const position = lastStack!.match(/\d{1,20}:\d{1,20}$/)![0].split(":");
    const isMainJs = /main[.]js/.test(lastStack);
    const name = isMainJs ?
          "main":
          lastStack.split(/.*dist\/\d*[.]/)[1].split(".bundle.js")[0];
    const path = isMainJs ?
          "dist":
          "dist" + lastStack.split(new RegExp(`\\\d*[.]${name}`, 'gi'))[0].split('/dist/')[1];
    const massage = stack.split("\n")[0];
    return {
      position: {
        line: parseInt(position[0]),
        column: parseInt(position[1])
      },
      path,
      name,
      massage,
    };
  } catch (error) {
    throw error;
  }
}
```

Функція getErrorPosition, потрібна для того, щоб визначити позицію коду, який спричинив помилку, шлях до файлу де сталася помилка, ім’я цього файлу та повідомлення яке його спричинило. Параметр який приймає ця функція це стек помилки, який передається з коду логгеру, який розміщений

на клієнтській частині. Якщо стек не передається то стандартний параметр функції це пустий рядок.

Змінна lastStack потрібна для того, щоб визначинити місце останнього виконання коду, за допомогою регулярного визару. Змінна position, потрібна для того, щоб за допомогою регулярного виразу взяти позицію помилки, яка відбулася для більшої повноти береться 20 символів від початку коду, який спричиняє помилку та 20 символів до початку, щоб зрозуміти контекст. Змінна isMainJs потрібна для того, щоб зврозуміти помилки шукати в головному бандлі коду, чи іншому, на проекті їх може бути велика кількість.

Змінна path, містить у собі шлях до файлу, де виконувася код, йду перевірка на головний виконуваний код, якщо так то повертає шлях до файду main.js, якщо ні, то до файлу бандлу певного модулю.

Далі створємо змінну massage, яка бере повідомлення з вхідного параметру stack.

Наостанок повертаємо об'єкт, який містить у собі позицію, лінію та колонку в коді, шлях до файлу, де сталася помилка, ім'я файлу, та повідомлення.

Слід зазначити, шо виконуваний код обернений в конструкцію try/catch, бо може бути таке, що станеться помилка потрібно її обробити та повернути у загальний оброник помилок на проекті, щоб також її залогувати.

Функція getErrorScope, потрібна для того, щоб визначити де саме сталася помилка та віддати цю інформацію для інтерфейсу. На вхід функція приймає параметр stack, що відповідає за стек помилки, які відбулася. Та за допомогою функціi getErrorPosition, яка була описана вище визначає позицію помилки.

```
export const getErrorScope = async ({ stack }: any) => {
  const scope: string[] = [];
  if (!stack)
    return scope;
  try {
```

```
const errorPosition = getErrorPosition(stack);
const { line, column } = errorPosition.position;
const { path: originPath, name: originName, massage } = errorPosition;
const mapPath = `../../${ originPath }/sourcemaps/${ originName }.js.map`;
```

Створюємо змінну scope, в якій буде міститися фінальна інформація про місце помилки, типо масиву строк, для того, щоб потім їх записати в журнал. Робимо перевірку на вхідний параметр stack, ящо його немає, то повертаємо пустий масив, щоб не викликати помилку, якщо параметр stack пустий. Далі щоб забезпечити обробку помилок, які можуть виникнути при виконанні коду, або при виклику функції getErrorPosition. У змінній errorPostion зберігається інформація про позицію помилки, що повертає функція getErrorPostion. Беремо з змінної errorPosition параметри: line, column, path, name, massage. Змінна mapPath, містить у собі шлях до соурсмапи, який доступний тільки нашому виконуваму коду.

```
const smc = await new sourceMap.SourceMapConsumer(fs.readFileSync(path.resolve(__dirname, mapPath), "utf8"));
const originPosition = smc.originalPositionFor({ line, column });
const originLine = originPosition.line as number;

const originFullPath = "../../" + smc.originalPositionFor({ line, column }).source!.replace("webpack:///", "")

const file = fs.readFileSync(path.resolve( __dirname, originFullPath), 'UTF-8');

const lines = file.split(/\r?\n/);

scope[0] = `${ massage } at ${ originLine } on ${ originFullPath }`;
```

Змінна smc містить у собі безпосередньо соурсмапу, за допомгою прокидування змінної mapPath у бібліотеку, яка працює з соурсмапами sourceMap, йде зчитування через файлову систему соурсмапи. Змінні originPostion, originLine та originFullPath містять у собі інформацію про позицію помилки, лінію де вона сталася та повний шлях до виконуваного у цей момент файлу. Змінна file, посилається на файл де міститься помилка і використовується у змінній lines, яка перетворює файл у масив зі строк, розбиваючи його на елементи, які починаються з нової строки. Формуємо перше повідомлення у змінну scope, яке буде записано у журнал з повідомленням про помилку, строкою де вона сталася, та шлях до файлу.

```
        lines.forEach((line: string, index: number) => {
          if (index >= originLine - 5  && index <= originLine + 5)
            scope.push(`${ index + 1 }. ${ line }`)
        });

        return scope;
      } catch (error) {
        return ["Failed parse stack:", stack, "parse error", error];
      }
    }
```

Наостанок, заповнюємо до кінця нашу змінну scope даними про помилку, за допомогою циклу forEach, по усім лініям помилки, та повертаємо фінальну інформацію у змінній scope. Також слід зазначити про блок catch, в якому відбувається обробка помилки, якщо при виконанні коду вище, щось пішло не так.

У цьому пункті було описано програмну власноруч розроблених алгоритмів логеру, та обробки помилок у високонавантажених веб-додатках. Розробка алгоритму була розділена на дві частини: алгоритм логеру для клієнтської та серверної частин, алгоритм визначення де саме сталася помилка. Основні технології: мова програмування Javascript, бібліотека

Typescipt, бібіліотка express.js та допоміжні бібліотеки для роботи з файлами та соурсмапами. Було описано як працюють соурсмапи, обфускація та для чого це потрібно.

3.2. Розробка інтерфейсної частини програми

Інтерфейсна частина програми є дуже важливою оскільки, вона поєднує у собі збір даних про помилки та безпосередньо місце, де збираються помилки, та як вони відображаються користувачу, який буде їх дивитися.

Інтрефейсну частину програми можна розділити на дві частини: збір будь-яких помилок через програмний код, адже розроблені алгоритми потрібно використати, та реалізація відображення журналу подій, де вони будуть відображатися та зберігатися.

Інтерфейс збору помилок у коді

Для збору помилок у коді, потрібно налаштувати серверну та клієнтську частини, оскільки вище було загадно, що будь-який запис буде йти через серверну частину, почнемо з неї.

Для початку потрібно створити едпоінт(роут, на який буде звертатися клієнт, щоб записати лог до журналу), до якого можна буде мати доступ з клієнтської частини, та процес його ініціалізації.

```
server.put("/log", logRouter);
```

На серверній частині, створюємо ендпоінт з назвою log, тип запиту буде PUT, для того, щоб репрезенативно позначити, що ми будемо складати дані. Прив’язуємо logRouter – роутер в якому буде описана логіка роботи серверної частини з клієнтською, та запис повідомлення у журнал.

Перейдемо до реалізації logRouter:

```
import logger from "@/server/services/LoggerService";
import { NextFunction, Request, Response } from "express";
import url from "url";
import { getErrorScope } from "@/server/helpers/ErrorParser";
```

Імпортуємо логгер, який був описаний в пункті 3.1, типи даних для Typescript з бібліотки express, url для роботи с адресами браузерними, та функцію getErrorScope, яка потріюная для того, щоб визначати місце де саме сталася помилка у веб-додатку.

```
export const logRouter = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const data = req.body;
    const scope = await getErrorScope(data);

    if (!data.label || !data.message || !data.level_name) {
      logger.clientError({
        label: "LOG_ROUTER_ERROR_MISSING_PARAM",
        message: data.message || "NO_MESSAGE_RECIEVED",
      }, req);
    }
```

LogRouter, є асинхронною функцію, бо якшо зробити його синхронним то потік виконання основного коду може зупинитися доки не виконається операція у роутері, тому потрбіно його зробити асинхронним. Приймає у себе три параметри: req, це запит який відбувся при виклику роутеру, res, це відповідь серверу, для того, щоб відправити відповідь з сервера на клієнтську частину та next, це функція яка передає виконання Javascript коду далі по ланці.

Оскільки функція виконання є асинхронною, то потрібно використовувати конструкцію try catch, для того щоб запобігти несподіваній помилці, та обробити її. У змінну data записуємо тіло запиту, для оскільки інформація про помилку міститься саме у ньому. Змінна scope містить дані, які будуть записану у неї після виконання функціїї getErrorScope, яка повертає інформацію про те, де саме сталася помилка в виконуваному коді, також є асинхронною.

Робимо перевірки, на наявність у тілі запиту обов’язкових параметрів це: lable, message, level_name, всі три параматери мають бути присутні, якщо ми їх не виявимо, то буде виклик логеру, зі стандартними повідомленнями, про те, що не було передано усіх необіхдних параметрів.

```
switch (data.level_name) {
  case "INFO":
    logger.clientInfo({
      label: data.label,
      message: data.message,
      context: data.context || {},
    }, req);
    break;
  case "WARN":
    logger.clientWarning({
      label: data.label,
      message: data.message,
      context: data.context || {},
    }, req);
    break;
  default:
    logger.clientError({
      label: data.label,
      message: data.message,
      context: data.context || {},
      scope,
    }, req);
    break;
}

res.send();
```

```
    } catch (err) {
        return next(err);
    }
};
```

Далі за допомгою конструкції switch/case, йде перевірка та запис логу у журнал подій за параметром level_name, можливі значення, які можуть приходити з клієнтської частини передаються з клієнтської реалізації логеру, в залежності від значення відбувається запис до журнали з відповідним level_name.

У кінці йде, виклик успішного виконання роутеру, тобто відповідь зі статусом 200 для клієнтської частини, щоб зрозуміти, щоб запис до журналу подій зробився успішно. Також обробка помилки у блоці catch, йде, передача виконання коду у наступні роутери.

Таку реалізацію інтерфейсу програмного модулю ми маємо на стороні серверу, тепер перейдемо до інтерфейсу клієнтської частини.

На стороні клієнта при запуску веб-додатку потрібно спочатку заініціалізувати логгер, за допомогою функції-колбеку, механізм реалізації якої був описаний у пункті 3.1.

```
import log from "@controllers/Logger";
log.init({}, (obj: any) => {
    fetch("/log", {
        method: "PUT",
        headers: {
            "Content-Type": "application/json;charset=utf-8",
        },
        body: JSON.stringify(obj)
    });
});
```

При ініціалізації клієнтської частини, спочатку імпортується клієнтський логер, та викликається метод init, який приймає у себе функцію-

колбек, в нашому випадку, цією функцією виступаю запит на серверну частину за адресою log, метод PUT. Таким чином, ми закріплюємо те, що при кожному виклику логеру у проекті, буде відбуватися запит на серверну частину, яка буде його обробляти та робити запис до журналу, щоб його можна було проаналізувати та виправити помилку. Перевага цієї реалізації полягає у тому, що оскільки ми використовуємо технології серверного рендерингу, ми не зможемо покласти наш сервер через велику кількість запитів, та вага її складаю приблизно 2-5 кілобайт. Нижче буде приведено приклад використання логеру на клієнтській частині:

```
window.onerror = function (err, info, vm) {
    const errorInfo = {
      message: err.message,
      stack: err.stack,
      level_name: "ERROR",
      context: {
        info,
        vm: vm.$el.className || "",
        uri: vm.$el.baseURI,
      },
    };

    log.error("GLOBAL_ERROR_HANDLER", errorInfo);
  }
```

Можна зрозуміти, що ми підписуємося на будь-які події помилок під того колі користувач взаємодіє з веб-додатком у браузері, стандартні інструменти Javascript, дозволяють відловити помилку, інформацію про неї, та якщо є елемент при взаємодії з яким відбулася помилка. Формується об’єкт помилки, та викликається клієнтський логер, який в свою чергу посилає запит на сервер, а сервер записує його.

Інтерфейс журналу подій

Для реалізації інтерфейсу журналу подій було вирішено взяти готовий агрегатор збору даних та їх відображення – це Kibana.

Kibana - це інтерфейсний додаток із відкритим кодом, який знаходиться на вершині Elastic Stack, забезпечуючи можливості пошуку та візуалізації даних, проіндексованих у Elasticsearch. Широко відомий як інструмент створення діаграм для Elastic Stack (раніше його називали ELK Stack після Elasticsearch, Logstash та Kibana), Kibana також виконує функції інтерфейсу користувача для моніторингу, управління та захисту кластера Elastic Stack - а також централізований концентратор для вбудованих рішень, розроблених на Elastic Stack. Розроблена в 2013 році в межах спільноти Elasticsearch, Kibana перетворилася на вікно у сам Elastic Stack, пропонуючи портал для користувачів та компаній.

Інтеграція Kibana з Elasticsearch та більший Elastic Stack роблять його ідеальним рішенням для підтримки наступних елементів: пошук, перегляд та візуалізація даних, проіндексованих у Elasticsearch, та аналіз даних шляхом створення гістограм, кругових діаграм, таблиць, гістограм та карт. Представлення інформаційної панелі поєднує ці візуальні елементи, а потім надає їх спільний доступ через браузер, забезпечуючи аналітичні подання в режимі реального часу на великі обсяги даних. Kibana дозволяє візуально аналізувати дані за допомогою індексу Elasticsearch або кількох індексів. Індекси створюються, коли Logstash (великомасштабний інгестор) або Beats (колекція одноцільових відправників даних) передає неструктуровані дані з файлів журналів та інших джерел і перетворює їх у структурований формат для функцій зберігання та пошуку Elasticsearch. Інтерфейс Kibana дозволяє користувачам запитувати дані в індексах Elasticsearch, а потім візуалізувати результати за допомогою стандартних параметрів діаграм або вбудованих додатків. Користувачі можуть вибирати між різними типами діаграм, змінювати агрегування чисел і фільтрувати за певними сегментами даних.

Інформаційна панель Kibana - це колекція діаграм, графіків, метрик, пошуків та карт, зібраних разом на одній панелі. Інформаційні панелі

забезпечують швидкий огляд даних із різних точок зору та дозволяють користувачам детальніше вивчити деталі.

Отже, рішення Kibana є ідеальним варіантом для наших вимог по збору метрик про помилки, та метрики про події у високонавантаженому веб-доадтку.

Для того, щоб встановити Kibana потрібно зробити п’ять простих кроків:

1. Завантажити ElasticSearch, адже Kibana працює на основі ElasticSearch, з веб-сайту elastic.co або з гітхаб репозиторію та встановити на серверній частині.
2. Завантажити Kibana та також встановити на сервер.
3. Далі у командній строці серверу прописати ці дві команди bin/elasticsearch, bin/kibana. Перша запускає ElasticSearch, друга запускає Kibana.
4. Відкрити Kibana, зазвичай вона відкривається на 5001 порті.
5. Доналаштувати додаткові налаштування.

Після цих кроків отримуємо рішення, яке наведено на рисунку нижче:

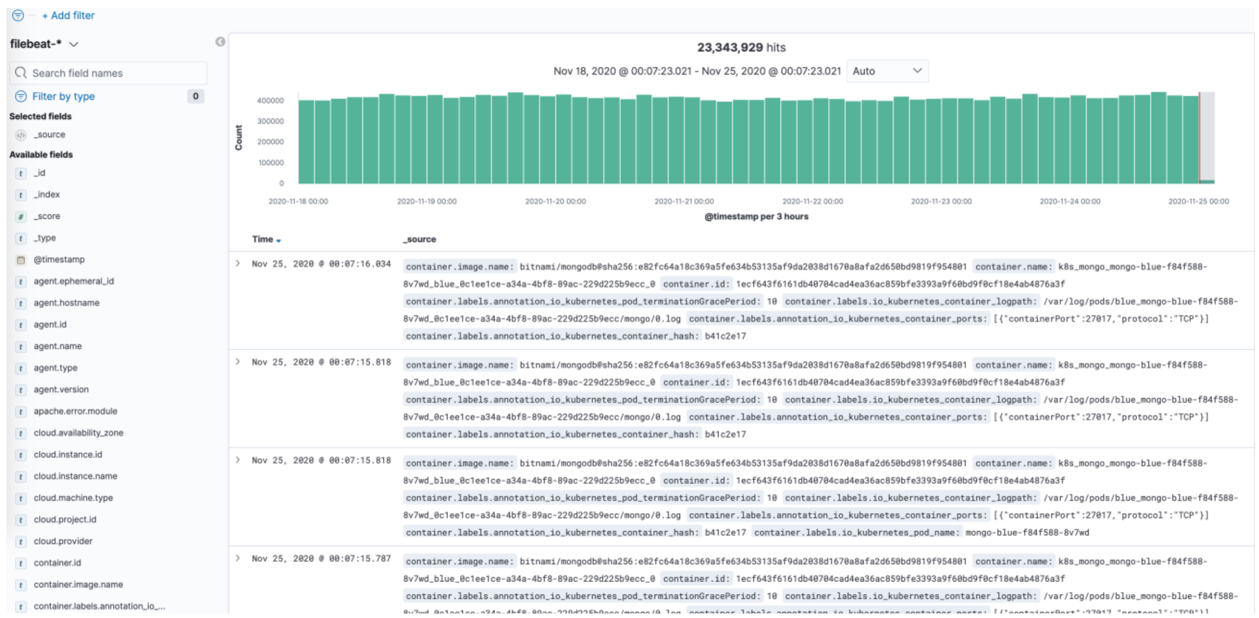

Рис.3.1 Приклад інтерфейсу Kibana

В даному розділі було описано програмний інфтерфейс та використання модулю для обробки помилок у високонавантажених веб-додатках написаних на мові програмування Javascript. Та вибрали рішення для журналу подій – це

Kibana, переваги якої, це швидка конфігурація, та дуже зручний інтерфейс для групування, агрегації та відслідковування даних.

3.3. Опис та тестування розробленого програмного модулю

Даний програмний модуль, написаний на мові програмування Javascript та Typescript, бо найбільша кількість сучасних високонавантажених веб-додатків написані на мові програмування Javascript.

Програмний модуль вирішує проблему необхідності відслідковування помилок у висконавантажених веб-додатках, та відповідає усім критеріям які було згадано вище. Проте, це рішення зроблено кокретно під потреби високонавантаженого веб-додатка.

Модуль складається з трьох частнин: алгоритм логеру, алгоритм відслідковування помилок у конкретному місці де сталася помилка та програмний інтерфейс з Kibana. Велика перевага цього модулю в тому, що кожна частина може використовуватися незалежно від інших, та може масштабуватися у майбутньому.

Проведемо тестування програмного модулю, для цього буде потрібно підключити наш модуль до вже працюючого високонавантаженого веб-додатку та навмисно зробити помилку, для того щоб перевірити модуль на працездатність та повноту інформації, яка буде потрапляти в Kibana.

Додамо фрагмент коду до головного Javascript файлу проекту main.js

```
window.onerror = function (err, info, vm) {
    const errorInfo = {
        message: err.message,
        stack: err.stack,
        level_name: "ERROR",
        context: {
            info,
            vm: vm.$el.className || "",
            uri: vm.$el.baseURI,
```

```
    },
  };

  log.error("GLOBAL_ERROR_HANDLER", errorInfo);
}
```

Після цього запустимо сторінку нашого додатку з повільним інтернет з’єднанням, для того, щоб нариклад деякі фрагменти коду повільно завантажувалися, або спрацював таймаут завантаження та була викличена помилка. Далі подивимося на наші логи в Kibana.

Nov 24, 2020 @ 23:02:33.737 agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24 bytes: 3,758 clientip: 193.185.70.83 extension: deb geo.srcdest: BI:IN geo.src: BI geo.dest: IN geo.coordinates: { "lat": 38.70042333, "lon": -87.12973222 } host: artifacts.elastic.co index: kibana_sample_data_logs ip: 193.185.70.83 machine.ram: 8,589,934,592 machine.os: win xp memory: - message: 193.185.70.83 - - [2018-08-07T21:02:33.737Z] "GET /elasticsearch/elasticsearch-6.3.2.deb HTTP/1.1" 200 3758 "-" "Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24" phpmemory: - referer: http://twitter.com/success/michael-p-anderson request: /elasticsearch/elasticsearch-6.3.2.deb response: 200 tags: error, info timestamp: Nov 24, 2020 @ 23:02:33.737

Expanded document　View surrounding documents　View single document

Рис. 3.2 Приклад логу про помилку в Kibana

Далі розгорнемо запис про помилку, щоб побачити більше деталей про помилку.

Field	Value
@timestamp	Nov 24, 2020 @ 23:02:33.737
_id	zYWc03UBJ_ynl2NmOiPp
_index	kibana_sample_data_logs
_score	-
_type	_doc
agent	Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24
bytes	3,758
clientip	193.185.70.83
event.dataset	sample_web_logs
extension	deb
geo.coordinates	{ "lat": 38.70042333, "lon": -87.12973222 }
geo.dest	IN
geo.src	BI
geo.srcdest	BI:IN
host	artifacts.elastic.co
hour_of_day	21
index	kibana_sample_data_logs
ip	193.185.70.83
machine.os	win xp
machine.ram	8,589,934,592
memory	-
message	Network error, timeout exceeded while loading google-analytics.js
phpmemory	-
referer	http://twitter.com/success/michael-p-anderson
request	/elasticsearch/elasticsearch-6.3.2.deb
response	451
tags	error, info
timestamp	Nov 24, 2020 @ 23:02:33.737
url	https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.3.2.deb
utc_time	Nov 24, 2020 @ 23:02:33.737

Рис. 3.3 Детальна інформація про помилку

Із даної помилки бачимо, дату, коли вона сталася, дані про те в якому браузері відбулася помилка, IP клієнта, географічні координати звідку було зроблено запит та країну, повідомлення про те, що сталася помилка при завантаженні скрипту google-analytics.js, та код помилки 451, також видно звідки користувач переходив на наш веб-додаток.

3.4. Оцінка ефективності впровадження програмного модулю відслідковування помилок

Щоб оцінити ефективність впровадження програмного модулю відслідковування помилок, достатньо поглянути на статистику помилок після його впровадження, та взяти порівняльну таблицю, яка була приведена у другому розділі.

Почнемо з статиски, оскільки раніше був впроваджений тільки логер, без відслідковування помилок, загальна кількість логів за один день в середньому була від 5 до 100 записів. Після впровадження даного модулю кількість помилок стала близько 500 в день.

Чому це позитивний результат? Тому що, стало відомо про багато помилок, про наявність яких не було гадки, і виявлено, що в багатьох випадках користувачі веб-додатку не отримували потрібний контент. Проте тепер стало видно в яких випадках відбуваються помилки, та те, що вони взагалі є. Таким чином, на протязі близько місяця, була проведена масштабна робота над помилками, та знижена їх кількість до 50 в день, що дало дуже значний ефект на продуктивність веб-додатку, та кількість звернень користувачів до служби підтримки з технічних питань зменшилась у два рази.

Нижче будуть приведені зображення до та після введеня модулю відслідковування помилок по кількості.

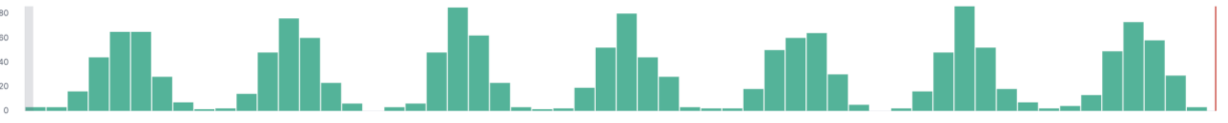

Рис. 3.4 Кількість логів до введення програмного модулю

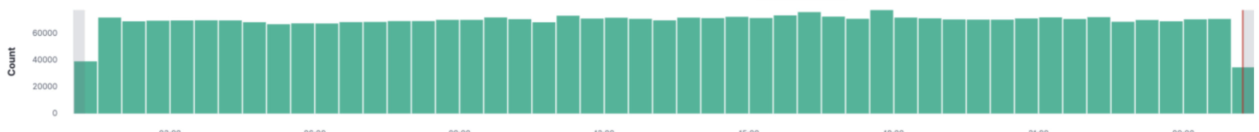

Рис. 3.5 Кількість логів після введення програмного модулю

Повернемося до порівняльної таблиці будут приведені зображення до та після введеня модулю відслідковування помилок по кількості.

Назва рішення / Вимога	Catch.js	Sentry.io	Власна реалізація
Інформація про кожен XMLHTTP запит	-	-	+
Інформація про оточення користувача	+	+	+
Детальна інформація про тип помилки	+	+	+
Гнучкість	-	-	+
Зв'язаність інформації про помилку з іншими запитами	-	-	+
Визначення конкретного місця де сталася помилка при виконанні програмного коду	-	+	+
Збереження історії помилок та безкоштовність рішення	-	-	+
Розмір програмного модулю	300кб	200кб	5кб

Табл. 2. Порівняльна таблиця рішень для відслідковування помилок у високонавантажених веб-додатках написаних на мові програмування Javascript.

З порівняльнох таблиці можна зробити висновок, що власна реалізація цілком задовольняє усі критерії, які було об'явлено у другому розділі.

Збираються метрики про кожен XMLHTTP запит, збирається інформація про оточення користувча в якому сталася помилка, детальна інформація про те, чим саме була викликана помилка, гнучкість рішення та його масштабування, зв'язанність інформації про помилку з інішми запитами за допомогою поля id, визначення конкретного місця де сталася помилка при виконанні програмного коду, за допомгою власноруч розробленого алгоритму, збережння історії помилок у журналі Kibana.

3.5 Висновки до третього розділу

У цьому розділі було розроблено та впроваджено алгоритм для відслідковування помилок у високонавантажених веб-додатках написаних на мові програмування Javascript.

Розроблено та впроваджено інтерфейс для моніторинг, збирання та агрерації записів про помилки у журнал подій за допомогою Kibana та ElasticSearch.

Описано та протестовано розроблені алгоритми та інтерфейси в умовах реального високонавантаженного веб-додатку.

Оцінено ефективність модулю, описано його переваги над аналогічними рішеннями, та те, що це рішення цілком задовольняє критерії які було сформовано у другому розділі.

Можна сказати, що з завдання, яке було поставлено на початку дипломної роботи вдалося вирішити.

Рішення має значну перевагу над уже існуючими за рахунок свого малого розміру близько п’яти кілобайт, гнучкості та гарнихперспктив для масштабування.

ВИСНОВКИ

У дипломній роботі було проведено роботу з аналізу сучасних підходів відслідковування помилок у високонавантажених веб-додатках, ознайомлено з проблемою та необхідністю відслідковування помилок у програмних модулях, адже не кожен розробник програмного забезпечення на сьогоднішній день опікується якість продукту, який розробляється, відслідковування помилок допомогає весь час цей продукт покращувати.

Розібрано поняття високонавантажених веб-додатків, навіщо вони потрібні, які проблеми вирішують, та з чим взаємодіють, адже на сьогоднішній день, у мережі Інтернет усюди використовуються саме високонавантажені веб-додатки для охоплення більшої кількості користувачів.

Проаналізовани існуючі методи та підходи відслідковування помилок у високонавантажених веб-додатках, адже для них потрібно значно більше зусиль, ніж для звичайного веб-додатку.

Описано мову програмування Javascript, її історію, походження та використання у сучасних веб-технологіях.

Було визначено основні принципи побудови сучасних високонавантажених веб-додатків на основі мови Javascript, та взаємодію з клієнтом на сервером.

Було приділено увагу основним інструментам, які існують на сьогоднішній день для відслідковування помилок у високонавантажених веб-додатках, такі як Sentry.io, Catch.js. Зроблено порівняльну характеристику цих рішень та сформовано критерії до рішення яке повністю задовольняє відслідковування помилок у високонавантажених веб-додатках.

У дипломній роботі було зупинено вибір на власному програмному модулі, адже приведені вище рішення не задовольняли основні критерії відслідковування помилок у високонавантажених веб-додатках.

Розроблено алгоритм та програмний модуль відслідковування поимлок у високонавантажених веб-додатках, який складається з логеру та модулю який визначає місце, де саме сталася помилка при виконанні програмного коду.

Розроблено інтерфейсну частину яка пов’язана з самим модулем, на основі рішення ElasticSearch – Kibana. Що цілком задовольняє основні критерії програмного модулю.

Було зроблено опис та тестування і оцінено ефективність впровадження програмного модулю, шлях порівняльння до та після вопровадження за кількістю записів у журнал, та розгорунтої інформації про помилку яка відбулася.

Було виконано досліження програмної реалізації, яке було проведено успішно.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Error handling [Electronic resource]: Eloquent Javascript. Режим доступу: World Wide Web – URL: https://eloquentjavascript.net/1st_edition/chapter5.html
2. Control flow and error handling [Electronic resource]: Mozilla Developer Network web docs. Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling
3. Handling errors in javascript [Electronic resource]: LevelUp git connected. Режим доступу: World Wide Web – URL: https://levelup.gitconnected.com/the-definite-guide-to-handling-errors-gracefully-in-javascript-58424d9c60e6
4. Error handling [Electronic resource]: OpenBook Project. Режим доступу: World Wide Web – URL: https://www.openbookproject.net/books/mi2pwjs/ch04.html
5. Swift error handling [Electronic resource]: Swift documentation. Режим доступу: World Wide Web – URL: https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html
6. Building high performance application [Electronic resource]: Hashjar. Режим доступу: World Wide Web – URL: https://www.hashjar.dev/blog/building-high-performance-scalable-applications
7. Why is error handling important [Electronic resource]: Stackoverflow. Режим доступу: World Wide Web – URL: https://stackoverflow.com/questions/368139/why-is-error-handling-important
8. Error handling in Javascript [Electronic resource]: ITnext. Режим доступу: World Wide Web – URL: https://itnext.io/error-handling-in-javascript-3e444ccae117

9. What is ERP [Electronic resource]: Oracle. Режим доступу: World Wide Web – URL: https://www.oracle.com/applications/erp/what-is-erp.html
10. Ecommerce [Electronic resource]: Shopify. Режим доступу: World Wide Web – URL: https://www.shopify.com/encyclopedia/what-is-ecommerce
11. Corporate portal [Electronic resource]: IT.integrator. Режим доступу: World Wide Web – URL: https://it-integrator.ua/en/corporate-portal
12. Error handling strategies [Electronic resource]: Dzone. Режим доступу: World Wide Web – URL: https://dzone.com/articles/error-handling-strategies
13. Technology Leaders Join OpenID Foundation to Promote Open Identity Management on the Web [Electronice resource]: IBM News Internet Portal. – Режим доступу: World Wide Web – URL: http://www-03.ibm.com/press/us/en/pressrelease/23461.wss
14. OAuth-FAQ page [Electronic resource]: Twitter developer documentation portal. – Режим доступу: World Wide Web – URL: https://developer.twitter.com/OAuth-FAQ
15. HTTP overview article [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Glossary/HTTP
16. Криптографічні системи та протоколи : навч. посіб. [для студ. баз. напряму 6.170101 "Безпека інформ. і комунікац. систем" усіх форм навчання] / А. Е. Лагун ; М-во освіти і науки України, Нац. ун-т "Львів. політехніка". – Л. : Вид-во Львів. політехніки, 2013. – 96 с. : іл. – Бібліогр.: с. 95 (10 назв).
17. Черьомушкін А. В. Криптографічні протоколи. Основні властивості і уразливості:навчальний посібник. М Ізд. центр «Академія», 2009. 272 с..
18. Index of the security protocols repository (SPORE) // Laboratorie Sp´ecification et V´erification. http://www.lsv.ens-cachan.fr/spore/table.html.
19. Clark J., Jacob J. A Survey of Authentication Protocol Literature: Version 1.0. 17 Nov. 1997. http://www.cs.york.ac.uk/ jac/papers/drareview.ps.gz, 1997.

20. Securing your site advices [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/Security/Securing_your_site
21. CORS technology overview and in depth [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
22. How to turn off form autocompletion article [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/Security/Securing_your_site/Turning_off_form_autocompletion
23. Privacy and the :visited selector article [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/CSS/Privacy_and_the_:visited_selector
24. Content Security Policy(CSP) article [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
25. Методы аутентификации и авторизации в web-приложениях [Электронный ресурс]: Интернет портал “Korzh.net” Интернет технологии, статьи по программированию – Режим доступа: World Wide Web – URL: http://korzh.net/2012-07-metody-autentifikatsii-i-avtorizatsii-v-web-prilozheniyah.html
26. Методы аутентификации [Электронный ресурс]: Интернет портал “PowerSecurity” Интернет технологии, статьи по онлайн безопасности – Режим доступа: World Wide Web – URL: https://powersecurity.org/ru/blog/authentication-methods/
27. Аутентификация пользователя. Веб [Электронный ресурс]: Интернет портал “Life-prog” Языки програмирования – Режим доступа: World Wide Web – URL: https://life-

prog.ru/view_programmer.php?id=158&page=16

28. HTTP и HTTPS [Электронный ресурс]: Интернет портал “HOSTiQ” Википедия хостинга – Режим доступа: World Wide Web – URL: https://hostiq.ua/wiki/http-https/
29. Reason: CORS disabled article [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSDisabled
30. HTTP resources and specifications article [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources_and_specifications
31. HTTP authentication article [Electronic resource]: Mozilla Developer Network web docs. – Режим доступу: World Wide Web – URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication